UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DEPARTMENT OF INFORMATION ENGINEERING

MASTER'S DEGREE
IN COMPUTER ENGINEERING

Dimensionality Reduction with Nvidia Tensor Cores

Supervisor:
Prof. Francesco Silvestri

Candidate:
Pietro Balzan

Padova, Italy

ACADEMIC YEAR 2021/2022
April 14, 2022

"Per Audacia Ad Astra"

Abstract

Thanks to the popularity and effectiveness of machine learning, the computational requirements for its development have increased beyond the limits of conventional devices. Because of this, in recent years, in order to speed up the training and inference processes of deep neural networks, a new hardware accelerator, the tensor core unit (TCU), has been introduced, allowing the computation time of matrix multiplication operations to be reduced. The aim of this work is to exploit the capabilities of tensor cores to speed up a different problem: dimensionality reduction. By making use of TCUs and certain properties of matrices, first introduced by William B. Johnson and Joram Lindenstrauss, we are able to embed a set of points having high dimensionality into a lower dimensional space with high-quality results. Throughout this paper, we will introduce the basic concepts of dimensionality reduction and explain the construction of the Johnson-Lindenstrauss matrices used in our reduction method in detail, as well as the theory involved. Following the description of Nvidia tensor cores and the Volta architecture, we will develop a number of dimensionality reduction algorithms. After testing their CUDA implementation on the Nvidia Tesla V100 GPU, we will extensively study their effectiveness and performance, both in terms of computation time and quality of results.

Contents

List of Figures

1 Introduction

One of the most important and relevant problems in the field of big data computing is the so-called *curse of dimensionality*, which refers to the issues arising when processing data in a high-dimensional space, occurring in many domains such as data analysis, statistics and machine learning.

Working with such spaces can have an impact on the quality of results as well as the performance of the algorithms used: depending on the problem, running times may have a linear or even exponential dependency on the number of dimensions. As the volume of the space increases, the available data also becomes increasingly sparse, with the negative effect of making common data organization strategies less efficient. For example, distance functions may lose effectiveness in assessing similarity between input entries. In these cases, in order to obtain a reliable result, the amount of available data would need to increase, often exponentially, with the number of dimensions.

The problems described above can be solved thanks to dimensionality reduction. This technique consists in the transformation of high-dimensional input data into a new dataset with a reduced number of dimensions, so that the new data points correlate to their originals in a meaningful way.

Many methods can be used to perform dimensionality reduction, but the one that we will be using and analyzing in this paper is the *Johnson-Lindenstrauss transform*, which has been proven to produce an output that retains many of the most important properties of the original data, such as pairwise distance and norm.

Much of the data stored and manipulated on computers, including text and images, can be represented as points in a high-dimensional Euclidean space; keeping this in mind, the problem of dimensionality reduction can be viewed as the product between a matrix (the transform) and a vector whose number of dimensions needs to be reduced.

At this point, we would like to introduce a hardware device of fundamental importance for this thesis: the tensor core unit (TCU). This family of hardware accelerators has been introduced in recent years because of the the ever-increasing challenges posed by deep neural networks and machine learning as a whole. This branch of artificial intelligence has reached unparalleled levels of accuracy and reliability in many applications, but, as the quality of this tool increased, so did its complexity: as the name suggests, deep neural networks are comprised of thousands of artificial 'neurons', interconnected within multiple layers, and their development requires a tremendous computational effort. The most common and time-consuming operations performed during the training and inference steps heavily rely on *matrix math* operations. In order to lessen this computational bottleneck, tensor core units, such as Google Tensor Processing Units or Nvidia Tensor Cores, are specifically designed to perform a dense matrix product, of a fixed $\sqrt{m} \times \sqrt{m}$ small size, as efficiently as possible.

Throughout this work, we will be using a machine containing an Nvidia Tesla V100 GPU, a graphics card equipped with hundreds of tensor cores; thanks to the CUDA API, TCUs can be programmed to aid in any task involving matrix multiplication, and this is how we can go back to the original problem of dimensionality reduction: thanks

to some additional properties of Johnson-Lindenstrauss transforms, the problem of dimensionality reduction can be treated as a matrix product, allowing us to exploit the power of tensor core units to achieve a $\sqrt{m}$ speedup over other traditional approaches [1]. These properties and the theoretical analysis of the problem will be presented in later sections.

The aim of this work is to evaluate some algorithms that perform dimensionality reduction through matrix multiplication by using both common, traditional methods, as well as implementing the idea presented by Ahle and Silvestri in [1] and leveraging the capabilities of Nvidia tensor cores, in order to compare the results both in terms of computation time and quality of the low-dimensional output vectors. In chapter 2 we will further introduce the main concepts of dimensionality reduction, while chapter 3 will describe the theory involved in the construction of Johnson-Lindenstrauss embeddings in detail. Chapter 4 will explain the Nvidia architecture, and in particular, the *Volta* generation, which was the first to introduce TCUs to the market, and also the architecure of the GPU we will be using for our tests. Next, chapter 5 will illustrate our proposed algorithms and their implementation with Nvidia's CUDA library, and, in chapter 6, we will show and analyze their practical results, both in terms of computation time and quality.

2 Dimensionality Reduction

As stated in the introduction, the issues that we face when dealing with high-dimensional data can be solved with the technique of dimensionality reduction. As a matter of fact, this tool is usually used during the pre-processing step of many problems in the field of big data. A dataset can be defined as high-dimensional when the number of features (dimensions) is greatly larger than the number of observations (input data points in the space) available. This issue is not at all theoretical: real-world datasets, such as speech signals, digital photographs, or MRI scans, often have high dimensionality. This technique lets us embed a high-dimensional input dataset within a space with a lower number of dimensions, while keeping the important properties of the data as intact as possible: ideally, the reduced representation should have a dimensionality that corresponds to the minimum number of parameters needed to account for the observed properties of the data (the *intrinsic dimensionality* of the data).

Let's give the problem a more formal definition: we can represent the inputs as a set V of vectors $v \in \mathbb{R}^d$, where d is the number of dimensions in the high-dimensional Euclidean space. The aim of the problem is to embed the original points into a space with lower dimensionality $k \ll d$ by using a suitable transform $T : \mathbb{R}^d \to \mathbb{R}^k$. Intuitively, the result is a "compressed" dataset where the valuable parts of the original data are kept, while the others are discarded.

This transformation helps to alleviate the issues caused by the curse of dimensionality: its benefits include smaller data storage requirements, reduced computation times and improvements in the quality of the data, where redundant or noisy features can be removed and only its most important aspects are captured. It is also very useful when the number of samples include a low number of observations, and a limited number of dimensions can also improve the visualization and readability of the data.

Over the years, a number of different approaches have been researched in order to generate the transform T, resulting in many different dimensionality reduction techniques being proposed.

The broadest subdivision of the different methods can be defined between *linear* and *nonlinear* techniques: the most important and commonly used linear technique is known as Principal Component Analysis (PCA), while examples of nonlinear techniques include Kernel PCA, Isomap, Hessian locally linear embedding (LLE), and many more. As shown by figure 2, different techniques allow us to tackle individual problems and deal with specific distributions of data in the optimal way.

Just as a brief overview of the characteristics, advantages and drawbacks of different dimensionality reduction techniques, let's examine a couple of the most popular ones:

PCA seeks a k-dimensional basis that best captures the variance in the data. With this construction, the first principal component (p.c.) would be the dimension having the largest projected variance, the second p.c. the orthogonal dimension with largest projected variance, and so on. Finding the first k principal components requires to find the covariance matrix of the centered data, as well as to compute and sort its

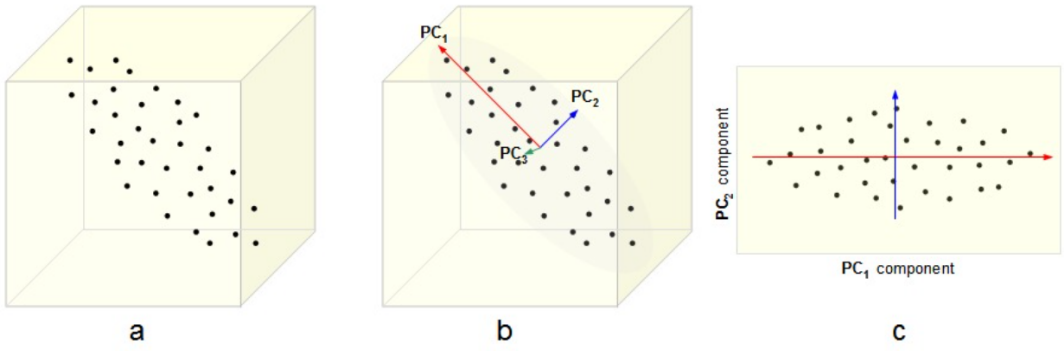

Figure 1: Visualization of a simple PCA dimensionality reduction from 3 to 2 dimensions.

eigenvalues in order to set the eigenvectors corresponding to the highest variance as the basis of the k-dimensional space. This method is certainly the most commonly used in practice, but it carries significant drawbacks related to computation time: since the size of the covariance matrix is proportional to the dimensionality of the datapoints, obtaining the eigenvectors can become infeasible for very high-dimensional data.

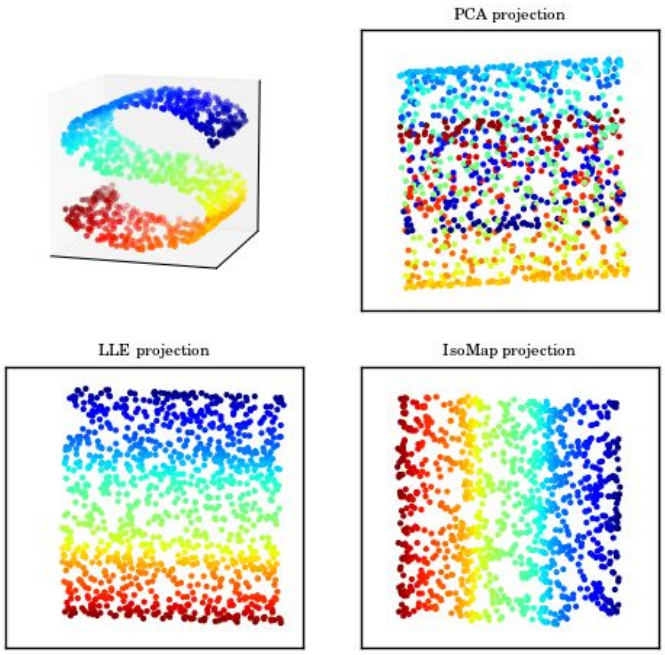

Figure 2: Linear and nonlinear dimensionality reduction techniques. The PCA projection is not able to adequately capture the features of the original points, as certain distributions of data require nonlinear reduction methods to obtain good results.

Kernel PCA is a nonlinear version of the first technique that utilizes the so-called 'kernel trick': this method computes the principal eigenvectors of the kernel matrix, instead of the covariance matrix. PCA is then applied in the kernel space, giving Kernel PCA the ability of constructing nonlinear mappings. This technique isn't immune to weaknesses either: the size of the kernel matrix is tied to the square of the number of instances that constitute the input dataset, so it's clear that finding the dimensionality reduction transform can often be a computationally taxing operation, regardless of the method we might choose.

An in-depth analysis and comparison of the most important dimensionality reduction techniques can be found at [2].

When compared to traditional techniques, one of the most important features of the method we will study here, which uses Johnson-Lindenstrauss properties, is that it allows us to generate the transform T randomly by using specific probability distributions, without further computational requirements. This is certainly a great advantage, and it has been proved that the quality of the input's transformation is comparable with the other methods, at least when considering features of the original dataset such as pairwise distance and vector norm, which are kept intact by the method with high probability.

Indeed, throughout this work we will be taking advantage of the Johnson-Lindenstrauss technique to perform dimensionality reduction, making the generation of the transform T a computationally trivial operation.

3 The Johnson-Lindenstrauss transform

In the late 1980s, William B. Johnson and Joram Lindenstrauss proved, with the so-called Johnson-Lindenstrauss lemma, that projecting points onto a random basis approximately preserves pairwise distances with high probability [3].

Lemma 1 (Johnson–Lindenstrauss): for every $d \in \mathbb{N}_1$, given a set V of n points in $\mathbb{R}^d$, $0 < \varepsilon < 1$, there exists a function $f : V \to \mathbb{R}^k$, where $k = \Theta(\varepsilon^{-2} log(|V|))$, such that for every x,$y \in V$,

$$| \, \|f(x) - f(y)\|_2^2 - \|x - y\|_2^2 \, | \leqslant \varepsilon \|x - y\|_2^2 \tag{1}$$

The proof of this lemma is based on defining the function as $f(x) := (d/k)^{1/2} Ax$, where $A \in \mathbb{R}^{d \times k}$ is a random orthogonal matrix. It was indeed shown that such a function is able to keep the norm of vectors unchanged with high probability by proving the distributional lemma, an alternative definition of lemma 1, which defines the JL property for probability distributions:

Lemma 2 (*Distributional Johnson–Lindenstrauss*): for every $d \in \mathbb{N}_1$ given a set V of n points in $\mathbb{R}^d$ and ε, $\delta \in (0, 1)$, there exists a probability distribution $\mathcal{F}$ over linear functions $f : \mathbb{R}^d \to \mathbb{R}^k$, where $k = \Theta(\varepsilon^{-2} \log(\frac{1}{\delta}))$, such that for every $x \in \mathbb{R}^d$,

$$Pr_{f \backsim \mathcal{F}} \left[\, | \, \|f(x)\|_2^2 - \|x\|_2^2 \, | \leqslant \varepsilon \|x\|_2^2 \, \right] \geqslant 1 - \delta \tag{2}$$

and we can say that such a distribution is a (ε, δ)-*Johnson-Lindenstrauss* (JL) distribution. In words, distributions with this property allow us to embed any set of n points so that all pairwise Euclidean distances are preserved up to a factor of $1 \pm \varepsilon$; the output dimension only depends on the number of points n in the dataset and the desired quality of the embedding.

Let's now analyze how one such JL-embedding could be generated: a coarse way to do so would be to simply take k of the original coordinates in $\mathbb{R}^d$ as the new coordinates, but since two points can be at a significant distance to one another while differing along just one dimension, this naive approach, by itself, is obviously fallacious. To give the reader an intuition of the real process, what we need is to find a way to have all coordinates contribute somewhat equally to the corresponding pairwise distance. To this end, the only necessary operation would be to apply a random rotation to the original d-dimensional set of points: this way, even the naive method would be viable. This is why the first methods proposed to generate the JL transform all involve randomized algorithms, based on the projection of the input points onto a spherically random hyperplane through the origin [4]. These ideas, while being fairly simple in terms of concept, also require to multiply the matrix formed by the input points with a dense matrix of real numbers; unfortunately, the computational effort required to solve this product can often be not trivial.

In a paper by Dimitris Achlioptas [5], a different method for the generation of JL embeddings was suggested. The author demonstrated that, instead of computing the spherical hyperplane projections, the same goal could be achieved with significantly faster and simpler operations. Furthermore, the random distributions proposed by Achlioptas, somewhat unintuitively, yield the same results as the other methods (and exceed them in some areas as well), without sacrificing any aspects of the embeddings' quality. This is the technique that we will be exploiting in order to generate dimensionality reduction transforms having the desired Johnson-Lindenstrauss properties.

Theorem 1 (*Achlioptas JL embedding*): Let's consider an arbitrary input set V of n points $v \in \mathbb{R}^d$, that we can represent as a $n \times d$ matrix M (every vector corresponds to a row of M), as well as two parameters $\varepsilon, \beta > 0$. We define $k_0 = \dfrac{4+2\beta}{\epsilon^2/2 - \epsilon^3/3} \log n$,

where ε controls the desired distance preservation accuracy, and β the probability of success. For any integer $k \geqslant k_0$, let T be a $d \times k$ random matrix with $T(i,j) = t_{ij}$, where t_{ij} are independent random variables from either of the following probability distributions:

$$t_{ij} = \begin{cases} +1 \text{ with probability } 1/2, \\ \text{-1 with probability } 1/2. \end{cases} \tag{3}$$

$$t_{ij} = \sqrt{3} \times \begin{cases} +1 \text{ with probability } 1/6, \\ 0 \text{ with probability } 2/3, \\ \text{-1 with probability } 1/6. \end{cases} \tag{4}$$

Let now $E = \frac{1}{\sqrt{k}} MT$ and $f : \mathbb{R}^d \to \mathbb{R}^k$ a function mapping rows of M to the corresponding rows of E. With probability at least $1 - n^{-\beta}$, for all $x, y \in V$:

$$(1-\epsilon)\|x-y\|^2 \leqslant \|f(x) - f(y)\|^2 \leqslant (1+\epsilon)\|x-y\|^2$$

As we can clearly see, this outstanding result allows us to construct an incredibly effective JL transform by using incredibly simple probability distributions. The distribution described at (4) even allows to cut down on execution times, since only a third of the entries in the transform matrix needs to be computed.

For reasons that will be made clearer in a few paragraphs, our proposed dimensionality reduction technique requires the input vectors to be lined up as the columns of the input matrix, instead of as the rows of M like shown in the above theorem. This is not a problem, as it is sufficient to take into consideration a property of matrix multiplication: $(AB)^T = B^T A^T$; that is, the transpose of a product of matrices is the product, in the opposite order, of the transposes of the factors. So, from now on, we will always refer to the JL transform T as a $k \times d$ matrix, while the input matrix M will be a $d \times n$ matrix. The outcome will be a matrix having the result vectors in its columns, but this is a further advantage because the CUDA API used in the implementation of the algorithm supports columns-major storage.

Now that we know what a JL distribution is and how to generate one, we can finally introduce the dimensionality reduction technique proposed by Ahle and Silvestri in [1]. This paper suggests that it is possible to generate a dimensionality reduction transform having the (ε, δ)-JL property, with which the *matrix-vector* product can be efficiently computed using tensor core units. In particular, when considering the TCU model, this method allows us to reach a speedup of $\sqrt{m}$ in matrix-vector multiplication.

The (m, τ)-TCU model is a simple computational model for tensor core units, designed to abstract the main hardware features of these accelerators [6]. Here, m refers to the hardware parameter indicating the size of the matrices that TCUs do calculations with (refer to chapter 4 for clarification), while τ represents the computation time of a matrix multiplication between two $\sqrt{m} \times \sqrt{m}$ matrices. With this nomenclature, we can say that regular machines using fast matrix-multiplication algorithms can compute the output in time $\tau = O(m^{3/2})$, while those equipped with TCUs have linear complexity, so $\tau = O(m)$. What needs to be highlighted is that the method we will analyze has been proven to provide a $\sqrt{m}$ speedup in *matrix-vector* multiplication, instead of *matrix-matrix* multiplication.

Theorem 2 (*Ahle-Silvestri matrix-vector TCU multiplication*): for any d,ε,$\delta > 0$, there exists a (ε, δ)-JL transform matrix $T \in \mathbb{R}^{k \times d}$ such that the matrix-vector product Tx can be computed in time $O((dk + k^2\sqrt{m}\log^3 \frac{d}{k})\tau m^{-3/2})$ in the (m, τ)-TCU model, assuming $k \geqslant m$.

The transform in question is defined as the following matrix:

$$T = (I_{r_l} \otimes A_l) \cdots (I_{r_1} \otimes A_1) \in \mathbb{R}^{r_m k_l \times r_1 c_1} \tag{5}$$

The writing $I_{r_l} \otimes A_l$ refers to the *tensor product* of the two matrices. The tensor (or Kronecker) product $C = A \otimes B$ between two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times l}$, results in a $mn \times nl$ matrix containing the B matrix mn times, multiplied in each position by an element of A that corresponds to the position of B in the $m \times n$ grid of $n \times l$ matrices represented by the output. In the case of a tensor product with an $n \times n$ identity matrix, the result is a matrix having the other input matrix repeated n times along the diagonal of the output, and all zeroes everywhere else (refer to figure below).

$$C = A \otimes B = \begin{bmatrix} A_{1,1}B & A_{1,2}B & . & A_{1,n}B \\ A_{2,1}B & A_{2,2}B & . & A_{2,n}B \\ . & . & . & . \\ A_{m,1}B & A_{m,2}B & . & A_{m,n}B \end{bmatrix}, I_n \otimes A = \begin{bmatrix} A & 0 & . & . & 0 \\ 0 & A & . & . & 0 \\ . & . & . & . & . \\ 0 & 0 & . & . & A \end{bmatrix}$$

And $(I_n \otimes A)$ is known as the *block identity matrix*.

At this point, we have to introduce a useful property of JL-distributions, proposed by Kane and Nelson [7]:

Definition 1 (*JL-moment property*): A distribution over random matrices $M \in \mathbb{R}^{k \times d}$ has the (ε, δ, p)-*JL-moment property*, when $E[\|Mx\|_2^2] = 1$ and
$(E[|\|Mx\|_2^2 - 1|^p])^{1/p} \leqslant \varepsilon \delta^{1/p}$ for all $x \in \mathbb{R}^d$, $\|x\|_2 = 1$

This property is quite useful when dealing with tensor matrix products, and more specifically, block identity matrices:

Lemma 3 (*JL Tensor lemma*): for any matrix Q with the (ε, δ, p)-*JL-moment property*, $I_k \otimes Q$ also has the (ε, δ, p)-*JL-moment property*.

It also needs to be stated that, thanks to Markov's inequality, a distribution with the (ε, δ, p)-*JL-moment property* also has the (ε, δ)-JL property. All this is important because, going back to our transform (5), $A_i \in \mathbb{R}^{k_i \times c_i}$ is defined as having the Strong JL moment property (which implies the JL-moment property [1]). Thanks to these characteristics, as well as the *JL Product lemma* found at [1], we can be certain that the transform T is a JL transform (meaning it has the JL property).

Let's now only consider a single product between an input vector and one of the sub-matrices of T: $(I_r \otimes A)x = v$. Thanks to the features of the block identity matrix, this operation can be conveniently transformed into a *matrix multiplication*, the outcome of which can be easily converted back to the vector that we would have obtained from the original matrix-vector multiplication. This essentially means that we can compute the product of a (specifically designed) matrix with a single vector, while also exploiting the power of tensor cores.

The above mentioned transformation of vector $x \in \mathbb{R}^d$ into the matrix taking part in the product, $X \in \mathbb{R}^{c \times r}$ (where $r = d/c$ is a fixed parameter representing the size of the identity matrix I_r), can be explained by making an observation on the block identity $(I_r \otimes A)$: since with the tensor product we obtain a matrix with copies of A along its diagonal and all zeroes elsewhere, when calculating a given value of the output vector, only a specific chunk of the inputs contributes to the result (because all other values outside the chunk are zero-products), and the contributing chunk shifts along the input vector x, depending on the output value to be computed. This is because, as the rows of the block identity matrix are considered, matrix A "shifts" along the diagonal. Note that every chunk is also always multiplied with the whole A. From here, we can split vector x into r chunks of length c, and use them as the columns of a new matrix $X \in \mathbb{R}^{c \times (d/c) = r}$.

Because of this, the matrix-vector product $(I_r \otimes A)x$ can be simply performed as the matrix multiplication between A and X.

$$v = (I_r \otimes A)x = \begin{bmatrix} k\{\underbrace{A}_{c} & & \\ & A & \\ & & A \end{bmatrix} x \backsimeq A \left[x_1 \quad \cdots \quad x_r\right]\} c = \left[v_1 \quad \cdots \quad v_r\right]\} k = V$$

The result of the product is a matrix $V \in \mathbb{R}^{k \times r}$ that can be easily transformed back to the vector that could be obtained from the original $(I_r \otimes A)x$ product. It's the opposite operation to what we did before to obtain X: concatenating all columns of V yields the result vector v.

A simple practical example of this operation and the transformations involved is provided in figure 3. The blue and green arrows represent the transformations applied to vector x during the construction of X, and the ones performed on matrix V to obtain v, respectively.

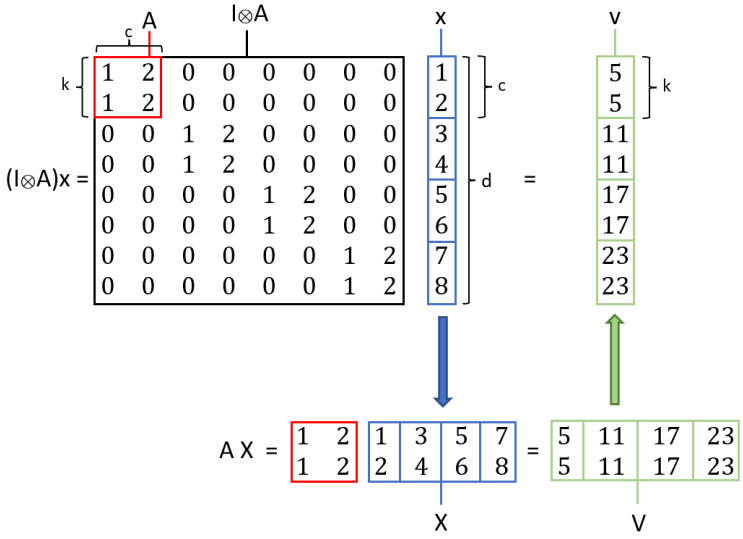

Figure 3: Simple numerical example of the JL transform operation described above.

To obtain the complete dimensionality reduction Tx, it is sufficient to compute the above product for each i (recall the definition of T), and concatenate the resulting vectors to obtain the final result.

The proposed transform is proven to compute the matrix-vector product Tx in time $O((dk + k^2\sqrt{m}\log^3 \frac{d}{k})\tau m^{-3/2})$, in the (m, τ)-TCU model. As previously stated, when using tensor cores, we have $\tau = O(m)$, so with this value the computation time can be rewritten as $O(dk/\sqrt{m} + k^2 \log^3 \frac{d}{k})$. This result can be simplified to $O(dk/\sqrt{m})$ in the case of the number of dimensions of the input set d being dominant over k^2, giving our method a speedup of $\sqrt{m}$ compared to traditional methods: as stated by [3], embedding a vector by performing a dense matrix-vector multiplication with a standard application of JL transforms takes time $O(dk)$.

4 Nvidia GPU microarchitecture

This study leverages the computing power of a Graphic Processing Unit, which provides much higher instruction throughput and memory bandwidth compared to a CPU. These two kinds of hardware components have different capabilities because they are designed with different goals in mind: CPUs excel at executing a single sequence of operations, called a *thread*, as fast as possible (and can execute a few tens of these in parallel), while GPUs are specialized in highly parallel computations (thousands of concurrent threads), and more transistors are dedicated to data processing rather than caching and flow control.

Thanks to the information provided by [8], [9] and [10], we will now take a look at the architecture and explain the internal structure of Nvidia GPUs, with a particular focus on the Tesla V100 GPU, which is the processor we used during our testing.

4.1 CUDA programming model

To better understand Nvidia GPU microarchitecture, first we need to briefly introduce CUDA. Released in November 2006, CUDA is a general purpose parallel computing platform and programming model used with Nvidia GPUs to solve many complex computational problems in a more efficient way. In the CUDA programming model, both the CPU (the *Host*) and GPU (the *Device*) can be freely used, and special care needs to be taken when thinking about memory allocation and access, because both the host and the device have their own memory spaces: making the most of GPU performance requires the data to be as close to the GPU as possible.

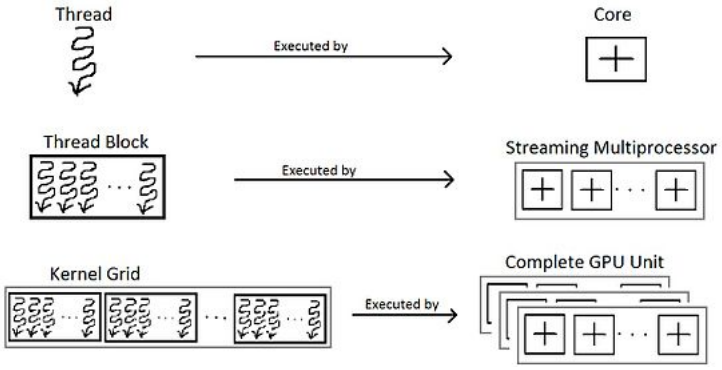

Figure 4: CUDA thread hierarchy

CUDA lets the programmer execute thousands of threads in parallel, all concurrently running the same function (known as the CUDA *kernel*). Threads can be organized in *blocks*: grouping of threads (up to 1024 in current GPUs) executing at the same time, working together on the same memory. Thread blocks are required to execute independently from one another, and can be further grouped to form *grids* (as seen in figure 5). This structure enables the programmer to easily write code that scales with

the number of cores, by setting the appropriate parameters when calling the kernel to decide the number of threads, blocks and grids.

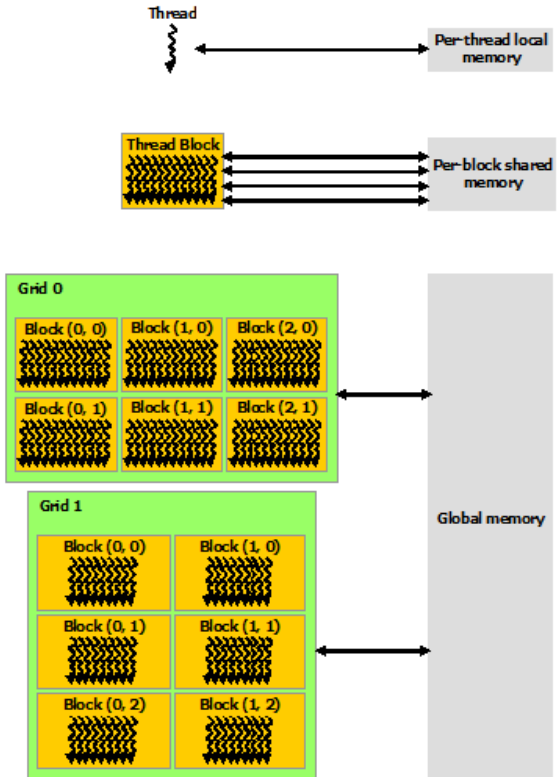

Figure 5: CUDA memory hierarchy

4.2 Streaming Multiprocessors

The Nvidia GPU architecture is built around the *Streaming Multiprocessor* (SM). SMs have their own control units, execution pipelines, caches and registers, and this is where all computations are actually performed.

In practice, when a CUDA program on the host CPU invokes a kernel grid, all the different blocks are distributed to available multiprocessors. One or multiple thread blocks can execute concurrently on a single SM, and as blocks terminate, new ones can be launched on the vacated multiprocessor. As such, SMs are designed to execute hundreds of threads at the same time, and do so by employing the SIMT (*Single-Instruction, Multiple-Thread*) architecture, thanks to which SMs can even take advantage of instruction-level parallelism.

The multiprocessor manages threads concurrently in groups of 32, called *warps*. Each thread of a given warp starts at the same initial state, but has its own instruction counter and register state. A warp executes one common instruction at a time, so if all 32 threads of the warp have the same execution path, the maximum efficiency is

achieved. However, threads are free to diverge from the others in case of data-dependent conditions, and in that case the warp starts to execute each branch taken, disabling threads outside that path.

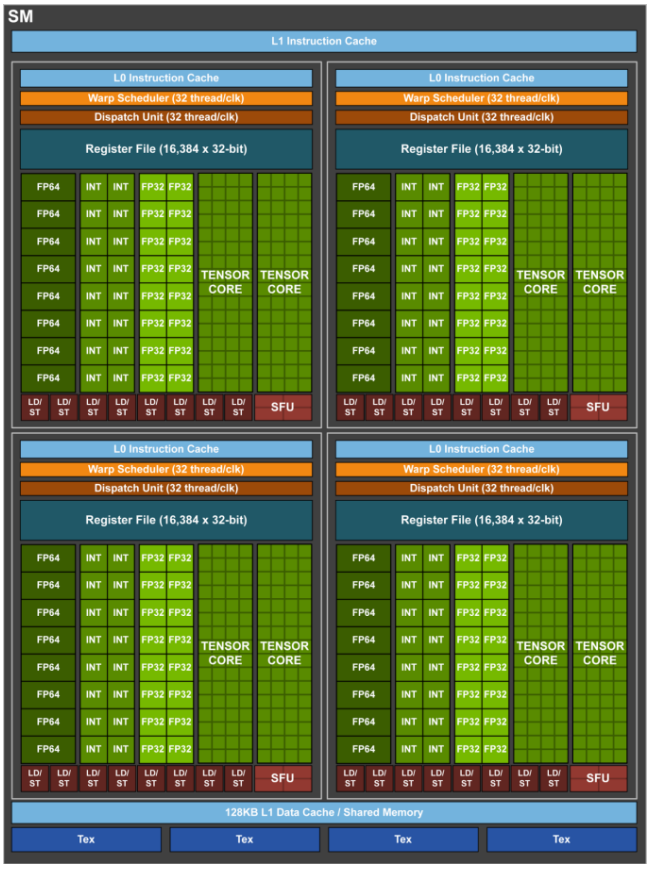

Figure 6: Volta GV100 Streaming Multiprocessor internal composition

The execution context for each warp processed by a SM is maintained on-chip for the whole duration of the execution, and, as such, switching from one context to another has no cost. For this task, the *Warp Scheduler* selects a warp with active threads (threads that are ready to be executed) and issues them with the correct instructions.

From the Hardware point of view, as we can see in figure 6, the physical implementation of a SM is partitioned into different processing blocks (they are 4 in the aforementioned image), each containing a number of FP32, FP64 and INT32 cores, an L0 instruction cache, a warp scheduler, a dispatch unit, and a Register file. The Volta architecture also includes new mixed-precision Tensor Cores, which we will later explain in detail.

4.3 Volta architecture specifics

The machine where we will perform our tests is equipped with a Tesla V100 GPU (also known as GV100), which belongs to the generation of Nvidia processors implementing the *Volta* architecture. This architecture brought a massive upgrade in both performance and energy efficiency, and represented the state-of-the-art for deep learning inference and HPC systems and applications, at the time of its 2017 release.

As we can see in figure 7, pairs of Streaming Multiprocessors are grouped into Texture Processing Clusters (TPC), and groups of TPCs form a GPU Processing Cluster (GPC).

Tesla contains multiple GPCs, as well as further caches and memory controllers. The full Volta GV100 GPU microarchitecture consists of:

- 6 GPCs, each having:
  - 7 TPCs (each including 2 SMs)
  - 14 SMs
- 8 512-bit memory controllers
- Each of the 84 Volta SMs is made up of:
  - 64 FP32 cores
  - 32 FP64 cores
  - 64 INT32 cores
  - 8 Tensor Cores

With 84 SMs, a full GV100 GPU has a total of 5376 FP32 cores, 5376 INT32 cores, 2688 FP64 cores, 672 Tensor Cores, and 336 texture units. It also includes a total of 6144 KB of L2 cache. figure 7 shows a full GV100 GPU with 84 SMs. In practice, the Tesla V100 accelerator only uses 80.

Volta's SMs are 50% more energy efficient compared to previous Nvidia architecture generations, enabling much greater performance in the same power range compared to previous architectures. Volta also introduced a new *independent thread scheduling* capability, which enables finer-grain synchronization and cooperation between parallel threads: it allows full concurrency between threads, regardless of warp. With independent thread scheduling, the GPU maintains execution state per thread, making better use of resources and even allowing threads to wait for data produced by others. A schedule optimizer determines how to group active threads from the same warp.

Thanks to its impressive raw computing power and efficiency, Tesla V100 is able to provide up to 7.8 TFLOPS of double precision floating point (FP64) performance, 15.7 TFLOPS of single precision (FP32) performance, and 125 Tensor TFLOPS.

Figure 7: Complete Volta GV100 GPU architecture

4.4 Volta Tensor Cores

As mentioned in the previous sections, Tesla V100 is equipped with Tensor Cores (TCUs), which were first introduced with the Volta microarchitecture.

Tensor cores were developed because of the need to manage and deal with the ever-increasing complexity and size of deep neural networks in machine learning applications: new networks with thousands of layers and millions of neurons demand higher performance and faster training times than ever. Tensor cores, designed specifically for deep learning, are the key element that enables the Volta GV100 architecture to deliver the necessary performance to train large neural networks.

The operation at the core of the training and inferencing process is Matrix-Matrix multiplication (General Matrix Multiply, or GEMM), which is performed to multitply large matrices of input data with weights in the (many) connected layers of the network. TCUs are programmable matrix-multiply-and-accumulate units specifically designed to solve this operation in the fastest and most efficient way possible.

Each core provides a 4×4×4 matrix processing array (which can be seen in figure 8), and operates on 4x4 matrices, performing the following operation:

$$\mathbf{D} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$$

where A,B,C,D are in fact 4×4 matrices. Nvidia refers to TCUs as performing *mixed precision* math, because the input matrices A and B are in half precision (meaning

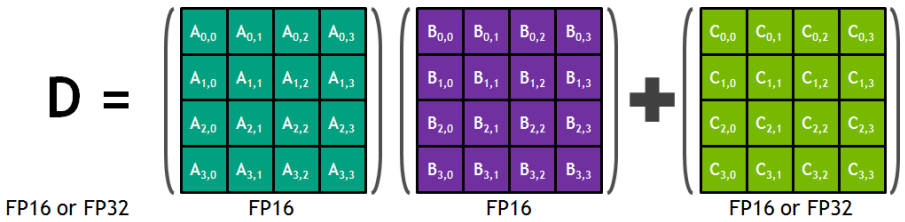

Figure 8: Tensor Core 4×4×4 matrix multiply and accumulate

the elements they are made of are 16-bit floating point numbers, or FP16) but the accumulation matrices C and D can be in *full precision* (32-bit floats, or FP32). As described in figure 9, the FP16 multiplication results in a full precision product that is then accumulated using FP32 addition with the other intermediate products for a 4×4×4 matrix multiply, requiring a total of 64 operations to generate the 4×4 output matrix.

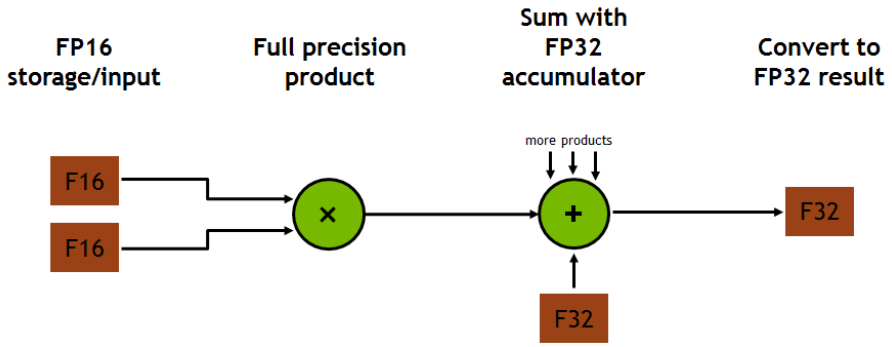

Figure 9: Flow of multiply-accumulate operations inside tensor cores

Of course, in practice, GEMM operations are performed on arbitrarily big matrices that are first decomposed into smaller elements, in order for tensor cores to be exploited.

In our case, we are not interested in training neural networks, but we can still take advantage of TCUs and perform fast matrix multiplication thanks to cuBLAS, CUDA's Basic Linear Algebra Subroutines library.

As a matter of fact, a result from [6] states that the (m, τ)-TCU model defines the asymptotic computation time for computing the product between two rectangular matrices:

Theorem 3 (*Rectangular matrices product*): Let A and B be $p \times r$ and $r \times q$ matrices respectively, with $p, r, q \geqslant \sqrt{m}$. Then, there exists an algorithm that computes the product $A \cdot B$ in time $O(prqm^{-3/2}\tau)$ on the (m, τ)-TCU model.

4.5 Example of a Gemm algorithm implementation with Tensor Cores

Even though we don't have access to the exact details of cuBLAS' Gemm algorithms we will be using in the code, thanks to the CUDA documentation [11], as well as one of the examples made available by Nvidia [12], we can still get a good idea of how they work.

First of all, a *Gemm* (General Matrix Multiplication) operation is defined as the following matrix multiplication and accumulation: $D = \alpha AB + \beta C$, with A, B and C input matrices, and α, β scalar inputs. In the simple matrix product that we are interested in, the gemm operation would be performed with $\alpha = 1$ and $\beta = 0$.

Sticking to the nomenclature used in the documentation, A has size $M \times K$, B is a $K \times N$ matrix, and, of course, C and D have size $M \times N$. In practice, C and D are phisically the same matrix, with C being overwritten with the results of the products and additions.

The basic idea of the implementation of a Gemm operation on a GPU is that the output matrix needs to be split into multiple *tiles* of size $Mtiles \times Ntiles$, which are then assigned to different thread *blocks* (refer to section 4.1 for their definition).

Taking figure 10 as reference, each thread block computes its output by advancing through the K dimension of its matrix in tiles, multiplying the loaded values from A and B and accumulating the result into the output matrix.

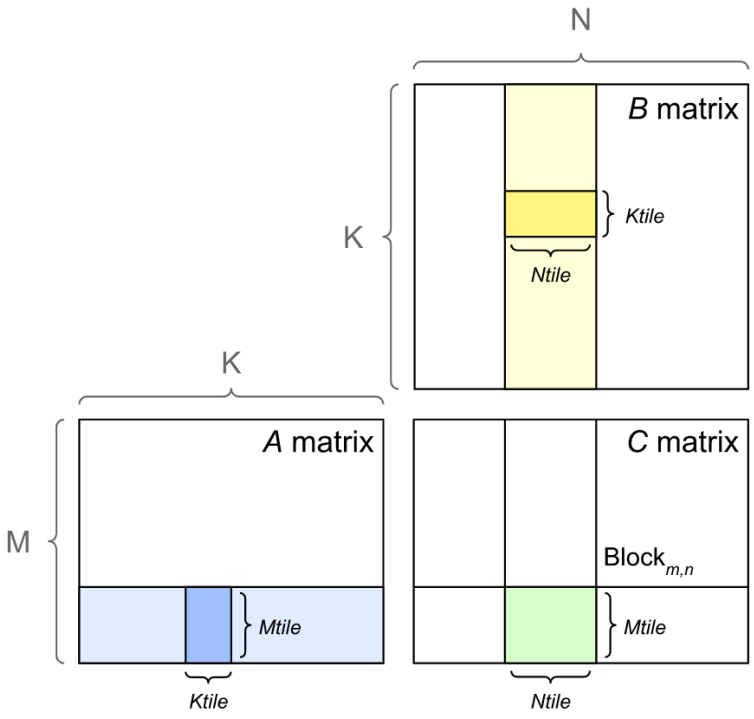

Figure 10: Tiled outer product approach to GEMMs

Figure 11 represents a very simple numerical example of the idea described above. In this case, each block is given a 2×2 tile of the output matrix. As shown by the blue squares, each portion of the output is computed in $K/Ktile = 2$ steps; the first one partially computes the result of the tile by performing the product between the two smaller matices taken from the inputs, and in the second one, new input tiles are selected, advancing in the K dimention, and their product is added to the output (in the example, since matrix B is the 4×4 identity and has tiles wholly composed of zeroes, some of these products don't add anything to the result).

For simplicity's sake, what's shown in the example is a sequential representation of the operation, where each output tile is processed one after the other. In practice, every tile is managed in parallel by a different thread block, and the figure can also be interpreted in that way: we can view each row of matrix triplets as the steps performed by one block concurrently with respect to the others.

Algorithm 1 Algorithm of a single Gemm operation with Tensor Cores

$WMMA\_K \leftarrow 16$ ▷ each warp advances through the k dimension of A and B in steps of 16

procedure SIMPLE_WMMA_GEMM($A, B, C, D, m, n, k, \alpha, \beta$)
    $a\_frag, b\_frag, c\_frag, acc\_frag$ ▷ declare fragments

    select output matrix tile to compute, based on the block and thread IDs

    for $i \leftarrow 0$ to k; i+=WMMA_K **do** ▷ advance through k dimension
        fill a_frag and b_frag with the correct data from each input matrix (synchronous step)
        ▷ perform product with TCUs and accumulate results in acc_frag
        $wmma :: mma\_sync(a\_frag, b\_frag, acc\_frag)$
    end for

    load the current value of c_frag (synchronous step)
    ▷ scale it by beta, and add to it acc_frag scaled by alpha
    for $i \leftarrow 0$ to $c\_frag.num\_elements$ **do**
        $c\_frag[i] \leftarrow \alpha * acc\_frag[i] + \beta * c\_frag[i]$
    end for
    store the output of the computed tile (synchronous step)
end procedure

Let's now take a look at the algorithm found at [12], of which the pseudocode is found above: I will explain the `simple_wmma_gemm()` CUDA kernel found inside file `cudaTensorCoreGemm.cu` of the GitHub page. This algorithm uses the *wmma* API to perform MMAs (Matrix-Multiplication and Accumulation), which uses instances of the `fragment` class to handle every tile of the matrix. A fragment represents a section of

a matrix distributed across threads in the warp, and so, different pieces of the tile are handled by different threads of the same warp (opaquely to the programmer).

Each warp handles a 16×16 tile of the output. First of all, the fragments are initialized and, as always, the operations require the matrices to be uploaded to the GPU, (in this specific case, to the single threads) through the wmma API. Note that, since a tile isn't contiguous in the memory where the matrix is originally stored, the inputs of the wmma function need to be set to also account for the stride between consecutive columns. To avoid conflicts, all the steps that require access to the memory (be it read or write operations) happen synchronously among the various threads and warps.

After the initialization, the algorithm starts a loop that, within each cycle, selects and uploads the correct pieces of matrices A and B to the thread, performs their multiplication through `wmma::mma_sync()`, and moves along the K dimension of the matrices by 16 rows or columns. The wmma function always adds the multiplication result to the same accumulation fragment, so that at the end of the loop, it will contain the intermediate result.

Next, the algorithm simply starts a new loop to account for the values of α,β, and the previous value held by the C matrix in that tile. After the tile from C is transfered to a fragment, one by one, every element is multiplied by β, and the corresponding element of the accumulation tile, multiplied by α, is added to it. At the end of this process, the values are transfered back to retrieve the tile of the result matrix D.

As we have seen, there are a lot of performance gains to be found in details such as warp management, and the reference code also contains an even more optimized version of the algorithm that employs a fine-tuned management of global and shared memory of blocks (memory hierarchy in CUDA is shown in figure 5).

Details such as those are probably the reason why the specifics of the algorithms used in the cuBLAS library are not provided by Nvidia.

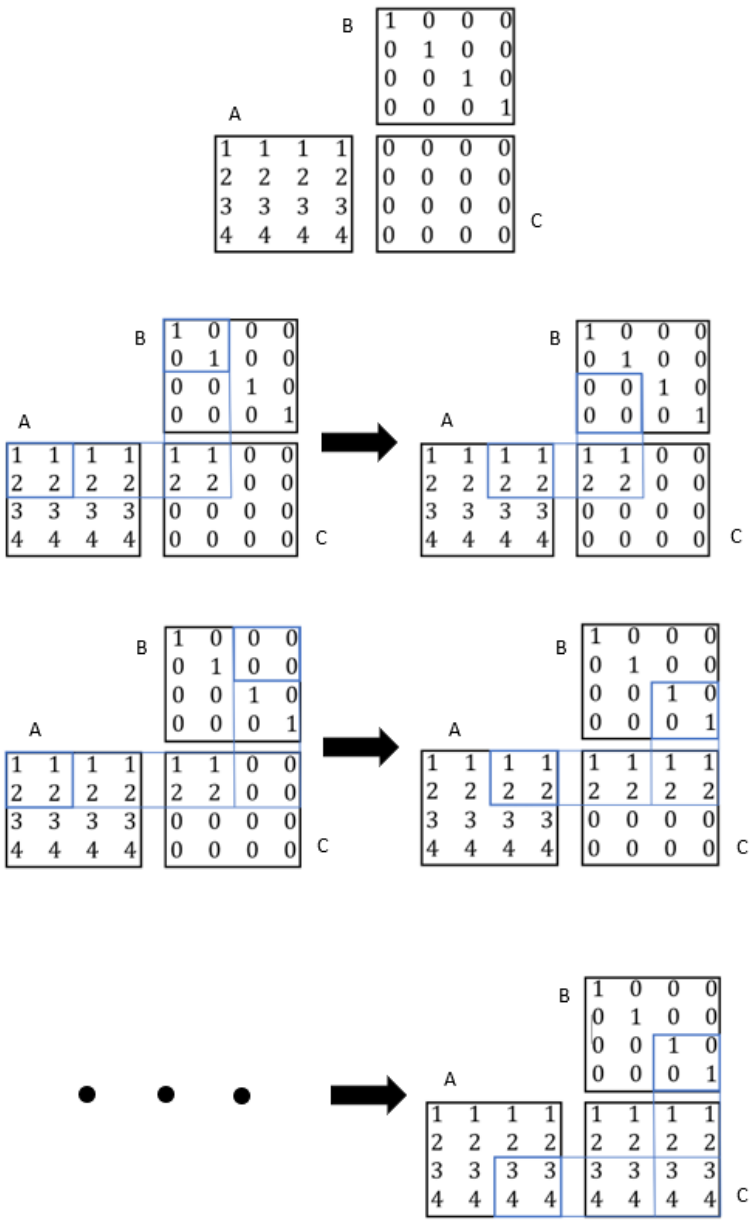

Figure 11: Simple numerical example of a Gemm operation using the tiled approach

5 Algorithm description

As we know, the algorithms in question will be run on an Nvidia Tesla V100 GPU and, as such, will make use of Nvidia's CUDA API with which we can write code that runs on a GPU, using the processor for general purpose computing. In particular, we will be exploiting the *cuBLAS* (CUDA Basic Linear Algebra Subroutine) library, which allows us to activate tensor cores when performing matrix multiplication. As stated by the documentation [9], in order to use this API, the application needs to allocate the resources and fill them with data in the GPU memory space, and after performing the desired operations, the results can then be uploaded to the CPU space. It is also required to manually initialize the library by creating a cuBLAS handle, which holds the library context, via the `cublasCreate()` method, which allocates the necessary hardware resources on both host and device. Once we have the handle, the math mode can be set to `CUBLAS_TENSOR_OP_MATH`, which allows the library to perform tensor core operations whenever possible.

CUDA also offers a feature called *Unified Memory*, thanks to which memory mamagement is streamlined into one common virtual memory, which can be accessed by either of the two kinds of processing units. This, however, only makes the programmer's job easier, but the data transfers in the correct memory spaces still need to happen, and relying on the page-faulting mechanism to move data is inefficient and slow, compared to a bulk copy. For thihs reason, we will not be using unified memory in our algorithms.

In this section, we will describe four different algorithms developed to perform the dimensionality reduction of either a single or multiple vectors, with and without TCUs. The first algorithm is the simplest and just performs a standard matrix-vector multiplication with cuBLAS and parallelization, without any specific optimization and, more importantly, without the aid of TCUs. It will be used as the baseline of performance in the "worst-case scenario", just in order to appreciate how much faster the other algorithms are. The second algorithm is the first one employing TCUs, and it does so by computing the product between the dimenionality reduction transform and a batch of vectors forming the second matrix (remember, tensor cores require matrix-matrix multiplication to be activated). The third algorithm is the most important one, being the implementation of the method proposed by Ahle and Silvestri and described in section 3. The final algorithm we will study is a variant (or more precisely, an extension) of the last one. Like the second algorithm, it computes the dimensionality reduction of a batch of vectors with tensor cores, while also exploiting the JL transform and its performance advantage over regular methods.

5.1 Standard matrix-vector multiplication

This is the first and simplest algorithm that we will consider. It doesn't make use of Tensor Cores and, by itself, it's not very interesting. It's simply used for us to have the baseline of performance that we can achieve for the problem of dimensionality

reduction.

This algorithm simply performs a basic multiplication between an $m \times n$ matrix A and a single vector v of length n, computing the m-long result vector r by implementing the standard matrix-vector product definition:

$$r = \begin{bmatrix} a_{11} & a_{12} & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & a_{2n} \\ . & . & . & . & . \\ \\ a_{m1} & a_{m2} & . & . & a_{mn} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ \\ v_n \end{bmatrix} = \begin{bmatrix} a_{11}v_1 + a_{12}v_2 + ... + a_{1n}v_n \\ a_{21}v_1 + a_{22}v_2 + ... + a_{2n}v_n \\ . \\ . \\ \\ a_{m1}v_1 + a_{m2}v_2 + ... + a_{mn}v_n \end{bmatrix}$$

The program first allocates memory for both the matrix and the vector, and, after loading the necessary data by filling them with random numbers, performs the multiplication parallelly through cuBLAS.

Algorithm 2 Standard matrix-vector multiplication

1: $n \leftarrow vector\ length$
2: $m \leftarrow matrix\ rows$
3:
4: $v \leftarrow malloc(n)$
5: $A \leftarrow malloc(m * n)$
6: $r \leftarrow malloc(m)$
7: $generateUnitVector(v)$ ▷ fill v and A with random numbers
8: $generateJLTransform(A)$
9: $ADevice \leftarrow cudaMalloc(m * n)$
10: $vDevice \leftarrow cudaMalloc(n)$
11: $rDevice \leftarrow cudaMalloc(m)$
12: ... ▷ initiate cuBLAS library handle and set math mode
13: $ADevice \leftarrow cublasSetMatrix(A)$ ▷ transfer data to device
14: $vDevice \leftarrow cublasSetVector(v)$
15: $cublasSgemV(ADevice, vDevice, rDevice)$ ▷ perform matrix-vector product
16: $r \leftarrow cublasGetVector(rDevice)$ ▷ retrieve result

This method uses a parallel algorithm to perform the multiplication, but, as we will see during the performance analysis, it is completely outclassed by the next ones we will study. Without exploiting tensor cores, it would be very difficult to find faster or more efficient methods to implement a basic matrix-vector product, since this program uses state-of-the-art algorithms developed by Nvidia that perform matrix-vector product with parallelization. This is why we chose to keep this algorithm as the baseline of performance.

5.2 Multiplication between a matrix and a batch of vectors

This algorithm performs the multiplication between a given matrix M and a batch of multiple vectors, making use of TCUs to achieve a faster execution time.

If we take a look at the definition of matrix multiplication between two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times l}$, we can easily observe how the i-th column of the result matrix corresponds to the result of the matrix-vector multiplication between A and the i-th column of B (which is itself, of course, a vector).

$$A \times B = \begin{bmatrix} a_{11} & a_{12} & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & a_{2n} \\ . & . & . & . & . \\ \\ a_{m1} & a_{m2} & . & . & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & . & . & b_{1l} \\ b_{21} & b_{22} & . & . & b_{2l} \\ . & . & . & . & . \\ \\ b_{n1} & a_{n2} & . & . & b_{nl} \end{bmatrix} =$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + ... + a_{1n}b_{n1} & . & . & . & a_{11}b_{1l} + a_{12}b_{2l} + ... + a_{1n}b_{nl} \\ a_{21}b_{11} + a_{22}b_{21} + ... + a_{2n}b_{n1} & . & . & . & a_{21}b_{1l} + a_{22}b_{2l} + ... + a_{2n}b_{nl} \\ . & & . & . & . \\ . & & . & . & . \\ \\ a_{m1}b_{11} + a_{m2}b_{21} + ... + a_{mn}b_{n1} & . & . & . & a_{m1}b_{1l} + a_{m2}b_{2l} + ... + a_{mn}b_{nl} \end{bmatrix}$$

Given the fact that, by using TCUs, we have an incredibly fast way of performing matrix multiplication, we can exploit the previous observation and group all vectors of the batch inside of a matrix. This allows us to obtain the dimensionality reduction of multiple vectors at the same time. The columns of the result matrix indeed represent the reduction of each vector of the batch.

In the same manner as the previous algorithm, the program first prepares all the data in host memory with the two input matrices: the dimensionality reduction transform T, as well as B, which represents the whole batch of vectors. Now, the data can be transfered to the GPU space. To this end, cuBLAS offers the function `cublasSetMatrix()`, which copies a tile of a specified number of elements from a matrix M1 in host memory space to a matrix M2 in device space. The memory used for the matrices in the device's space, as previously stated, first needs to be allocated, and this is done by simply calling `cudaMalloc()` for each of them.

Once everything is ready, the algorithm can compute the matrix product through the `cublasGemmEx()` function (where *Gemm* means General Matrix Multiplication), and the resulting matrix is subsequently extracted and saved back on the host.

The documentation states that there are multiple gemm algorithms that can be selected when calling the function, though their specifics and/or differences are not explained anywhere. The general funcitoning, however whould be very similar to what was explained in section 4.5, though it's not perfectly clear how the matrices are decomposed in order to be reduced to a size that's appropriate for the TCUs.

Algorithm 3 Matrix-Matrix multiplication with Tensor Cores

1: $m \leftarrow matrixrows$
2: $n \leftarrow lengthofvectorsinthebatch$
3: $l \leftarrow numberofvectorsinthebatch$
4: $A \leftarrow malloc(m * n)$ ▷ allocate matrices in host memory
5: $B \leftarrow malloc(n * l)$
6: $R \leftarrow malloc(m * l)$
7: $generateJLtransform(A)$ ▷ fill inputs
8: $unnitVectors(B)$
9: $Adevice \leftarrow cudaMalloc(m * n)$ ▷ allocate matrices in device (GPU) memory
10: $Bdevice \leftarrow cudaMalloc(n * l)$
11: $Rdevice \leftarrow cudaMalloc(m * l)$
12:
13: ... ▷ initiate cuBLAS library handle and set math mode
14:
15: $Adevice \leftarrow cublasSetMatrix(A)$ ▷ transfer data to device memory
16: $Bdevice \leftarrow cublasSetMatrix(B)$
17:
18: $Rdevice \leftarrow cublasGemmEx(Adevice, Bdevice)$ ▷ perform product with TCUs
19:
20: $R \leftarrow cublasGetMatrix(Rdevice)$ ▷ transfer multiplication result back to the host

Because of this, in practice, when calling `cublasGemmEx()` inside the program, the parameter specifying the algorithm was set as `CUBLAS_GEMM_DEFAULT_TENSOR_OP`, which applies heuristics to select the gemm algorithm (while allowing the use of Tensor Cores) autonomously.

5.3 Johnson-Lindenstrauss transform

In the following section, we will see an implementation of the Johnson-Lindenstrauss dimensionality reduction, as it is presented in section 3. As a refresher, the reference paper [1] describes a way to construct a matrix $M \in \mathbb{R}^{k \times d}$ having the (ϵ, δ)-JL property, and for which there is an efficient algorithm to compute the product Mx on a TCU. At a high level, the JL property of the matrix means that the result of the product preserves many features of the original vector x, such as, for example, its norm.

In terms of the practical implementation of the algorithm, since we ultimately just aim at performing matrix multiplication with TCUs, the program is structurally and functionally quite similar to the previous one described in section 5.2.

We start with the assumption that A is already given as input, so the program doesn't have to spend time extracting the matrix from the block identity $(I_r \otimes A)$.

A very important detail is that, thanks to the functioning of the cuBLAS methods used to transfer the data between host and device (`cublasSetMatrix` and

`cublasGetMatrix`), both the transformations $x \to X$ and $V \to v$ (see figure 3) happen without any computation: it's just a matter of re-arranging the data during its upload and download between memory spaces by properly setting the input parameters of the cuBLAS functions.

The first transformation happens when calling `cublasSetMatrix()` [9]: this function copies a tile of `rows` $\times$ `cols` elements from a matrix `A` in host memory space to a matrix `B` in GPU space. The leading dimensions (the number of rows) for source matrix A and destination matrix B are given as input with parameters `lda` and `ldb`, respectively. So, when transferring vector x to the GPU, we can set its host pointer as the source matrix, and by specifying `rows=c` and `cols=d/c`, with `lda=c` and `ldb=c`, we are creating the X matrix in the device with the correct number of rows and columns. This whole process is made very convenient by the fact that matrices in cuBLAS are stored in column-major format. As seen in figure 12, this means that every contiguous chunk of memory taken from x will be interpreted as a column of the matrix, and not as a row (which is the most common practice).

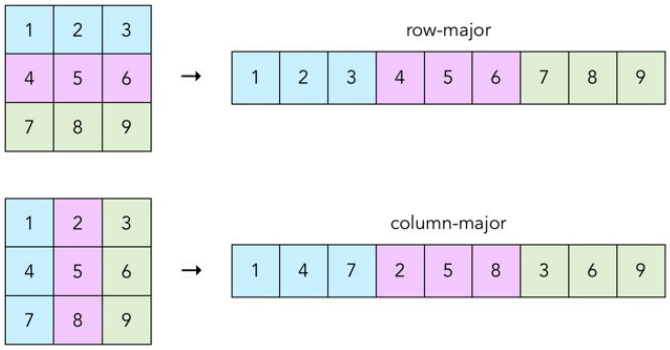

Figure 12: Row-major and Column-major allocation examples. Since, as seen in Figure 3, the chunks of vector x form the columns of matrix X, the column-major format is very convenient for our needs.

For a similar reason, when extracting the result V matrix from the device's memory with `cublasGetMatrix`, we can directly save it, "as is", into a single block of host memory; there is no need to transform the saved matrix because columns of V will already be concatenated to one another in the host memory's registers, forming the result vector v.

5.4 Batch JL Algorithm

The previous algorithm can be easily expanded to perform multiplication between the block identity matrix $(I_r \otimes A)$ and a batch of b vectors $x \in \mathbb{R}^d$, using TCUs.

Indeed, in this case, the block identity matrix is multiplied with a new matrix

$B \in \mathbb{R}^{d \times b}$, which represents the batch of vectors. As in the batch construction of section 5.2, every vector forms a column of B.

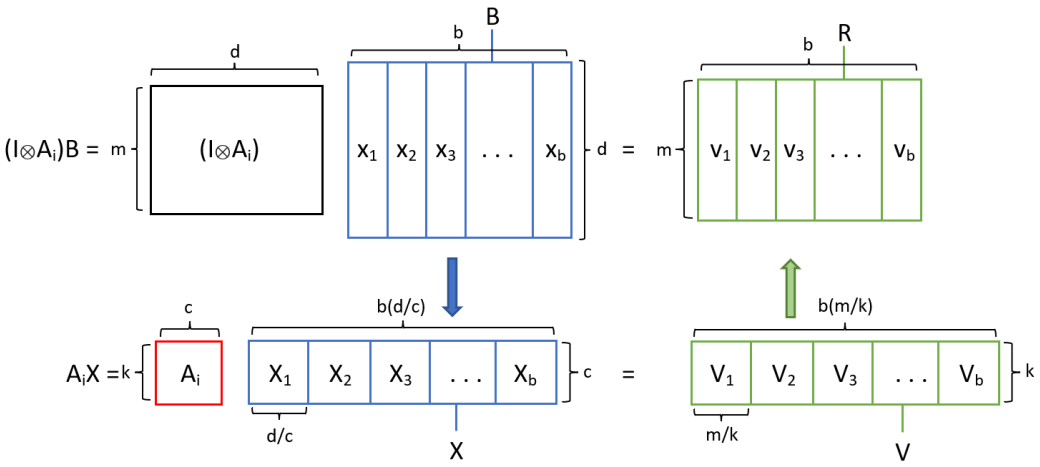

Figure 13: Visualization of the batch JL multiplication described in this section.

If we consider any individual vector x_i from the batch, the observation made in section 3 is still true: it can be split and transformed into a matrix X_i like previously described. X_i can once again be multiplied with A to obtain the matrix V_i, the columns of which, once concatenated, form the vector v_i, which equals to the result of the original product $(I_r \otimes A)x_i$.

Since every matrix X_i needs to be multiplied with A, the size of the chunks in which vectors are split during the transformation is always c. Thus, as shown in figure 13, we can simply concatenate the obtained $X_i \in \mathbb{R}^{c \times (d/c)}$ matrices into a "longer" matrix with c rows and $b(d/c)$ columns, which we will call X. The transformation from x_i to X_i happens during the upload to device memory, as described for the previous algorithm in 5.3.

After this step, the product AX can be computed using Tensor Cores, giving the $k \times b(m/k)$ V matrix as output. This matrix is formed by the concatenation of the single $V_i \in \mathbb{R}^{k \times (m/k)}$ matrices representing the result of the product AX_i for every vector of the batch (each is obtained like matrix V from figure 3).

Again, thanks to the column-major storage format of cuBLAS, we can transfer the whole result matrix back to the host, saving them into the $m \times b$ result matrix R. The individual v_i result vectors are contiguous in memory, and can be easily read or extracted from R by offsetting the pointer to the matrix by the appropriate $i \cdot m$ amount (where m is the length of the result vectors and columns of R).

As we explained in the previous algorithm, both of the transformations represented by the blue and green arrows in figure 13 happen naturally during the data transfer, without further computational effort.

6 Experimantal results

In this section, we will analyze the practical behaviour and effectiveness of the above described dimensionality reduction algorithms on the Nvidia Tesla V100 GPU. In particular, we will test each one on a defined set of input parameters and measure their performance, both in terms of execution time and quality of the results. Section 6.1 will provide the results of our computation time measurements, which will be performed on all four algorithms introduced in chapter 5, while section 6.3 will examine the quality of the output produced by the JL transform algorithm (from section 5.3) with different input parameters, as well as comparing its results with a standard execution of the Achlioptas method described in [5]. In this chapter, we will display numerous plots in order to describe and visualize the results; their representation will often include the variance of the data, displayed with coloured squares in the figures.

6.1 Time performance

The goal of this section is to compute and analyze the amount of time needed to perform a single dimensionality reduction on a vector of length N by multipliyng it with an $M \times N$ matrix. For a more convenient interpretation of the data, the only parameter that will change is N, while the dimensionality of the reduced vectors $M = 1024$ will stay the same during all tests. Each algorithm has been tested on the following values of N:

- $2^{10} = 1024$
- $2^{12} = 4096$
- $2^{14} = 16384$
- $2^{16} = 65536$
- $2^{18} = 262144$
- $2^{20} = 1048576$

Time measurements were performed with `cudaEvents`. Running the code with many cudaEvents calls obviously slows down the programs to some degree, but since they are all tested with the same method and the same number of library calls, it's fair to say that we can still consider the measurements and the comparisons between algorithms as valid. This is also the performance measurement method suggested by Nvidia [13].

Since in a real-case scenario the input vectors and matrices would still need to be uploaded to the GPU memory space before performing the products, I opted to include the data transfers of the inputs from host to device, and of the results from device to host, in the time measurements. I also calculated the percentage of total execution time spent during the data transfers. On the note of input datasets, since for now we don't

care about qualitative results, it's sufficient to generate a dense random matrix for the dimensionality reduction transform, and random vectors for the inputs.

The programs repeat the multiplication a certain number of times n and store the execution time of each iteration. From these informations, we can later calculate the average value of the execution time $\bar{t}$, as well as its sample variance $\sigma^2 = \frac{\Sigma(t_i - \bar{t})^2}{n-1}$.

Obviously, practical measurements are always going to be affected by some amount of variance: while testing the code, we noticed that, more often than not, computing one multiplication with a given algorithm consistently resulted in a certain time measurement t, but, when repeating it n times, the average time did not even closely amount to nt, especially when talking about the algorithm where tensor cores are activated. This suggests that there is some kind of execution overhead during the first executions that we must account for. For this reason, in order to more precisely represent the real value, the multiplication was always performed 100 times for each value of N, with the execution time of each iteration being stored. From those results, we could later calculate the average value of the execution time $\bar{t}$, as well as its sample variance $\sigma^2 = \frac{\Sigma(t_i - \bar{t})^2}{n-1}$.

This repetition in the measurement is implemented in all algorithms.

6.1.1 Baseline algorithm

Let's start with the baseline algorithm: times are almost identical between iterations, and, as seen in figure 14, the results follow a perfectly linear pattern with respect to the size of N. Looking at this algorithm alone, a $\sim 400ms$ execution time for the reduction of a vector made of a million elements might seem impressive, but, as we will soon discover, TCUs really make an exceptional difference in computation times.

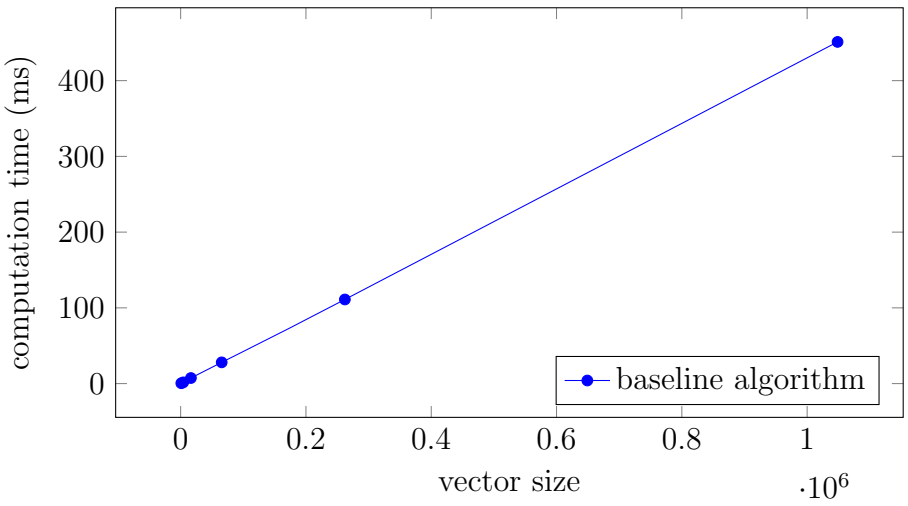

Figure 14: Standard matrix-vector multiplication algorithm performance

Looking at the data transfer times, we can clearly observe that the percentage of time spent on read/write operations is very high, with the conclusion that the performance of the algorithm is heavily reliant on the memory i/o capacity, and not limited by the computational capabilities of the device.

vector size	time (ms)	variance	data transfer(%)
1,048,576	451.149	3.34	94.9
262,144	111.072	0.0595	95.2
65,536	28.05	0.0057	96.4
16,384	7.35	0.004	97.1
4,096	1.625	0.003	96.5
1,024	0.488	0.0002	96.6

6.1.2 Batch algorithm

Moving on to the standard batch algorithm: as previously described, the program does not actually perform a single matrix-vector multiplication, but, given the i-th vector of the batch, the i-th column of the result is still its dimensionality reduction. Keeping this in mind, we can still measure the time of a single dimensionality reduction by simply dividing the time of a batch multiplication by b, the number of vectors in the batch. In this case, $b = 256$ was kept constant throughout all values of N that were tested.

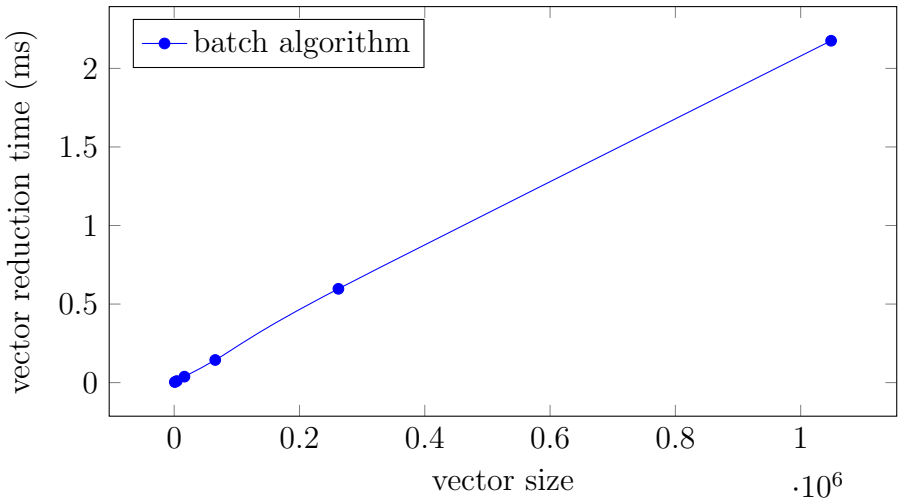

Figure 15: Plot of batch algorithm: the values shown represent the dimensionality reduction time for a single vector of the batch.

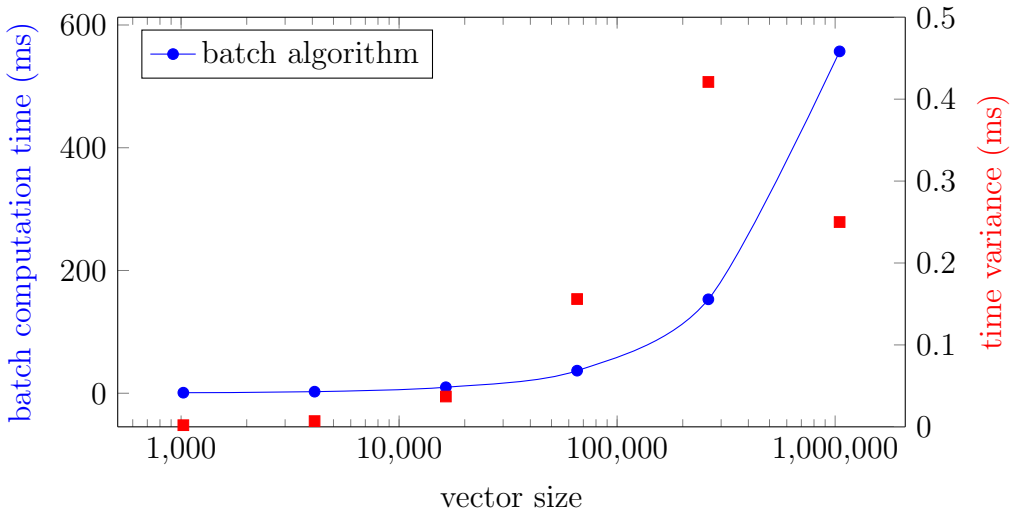

Figure 16: Logarithmic plot of batch algorithm. Times refer to the execution time and variance of the multiplication of an entire batch.

The measured values follow a mostly linear pattern once again, and the data transfer time percentage is quite high this time as well: it's at a value of around $90 - 95\%$ of the total computation time, making this algorithm memory limited.

As previously anticipated, compared to the baseline algorithm, we can already see a massive improvement in execution time (by multiple orders of magnitude!), obviously thanks to the usage of TCUs.

vector size	batch (ms)	batch variance	single vector (ms)	data transfer %
1,048,576	557.02	0.25	2.176	94.6
262,144	153.05	0.421	0.597	94.7
65,536	36.78	0.156	0.144	95.1
16,384	9.799	0.037	0.038	93.8
4,096	2.556	0.007	0.0099	89.9
1,024	0.891	0.002	0.0035	92.3

6.1.3 Single Johnson-Lindenstrauss algorithm

Next, we have the algorithm using the JL transform described in section 3: remember that, in this algorithm, we want to compute the matrix-vector multiplication $(I_{r_i} \otimes A_i)x$ by transforming x into a matrix X and performing the matrix-matrix multiplication AX. As such, the size of matrices A and X taking part in the actual calculations with tensor cores are a lot smaller than the M and N inputs (figure 3 gives a rough idea of this: $A \in \mathbb{R}^{k \times c}$ is a lot smaller than $(I_r \otimes A) \in \mathbb{R}^{M \times N}$) . When preparing these tests this was something to keep in mind.

To be clearer, the program was set so that $(I_r \otimes A)$ would be the input matrix of size $M \times N$; these are, in practice, the same parameters described at the beginning of the chapter, set so as to allow comparisons with the other algorithms. A is a $k \times c$ matrix, but these two values have to be chosen according to a given r (which represents the number of blocks in which the input vector is split), so that $M = k * r$ and $N = c * r$. For the purpose of this performance evaluation, I decided to try different values of $r = 32, r = 64, r = 128$.

This also means that the other matrix, X, is of size $c \times r$, and, as such, depends on the value of r. As expected, the performance of the algorithm significantly changes depending on the chosen value of the parameter r.

This algorithm is extremely fast, and the percentage of time spent during data transfers is also less than previous algorithms, though it still takes up the majority of execution time, especially at the largest values of N. This happens because the transform matrix $(I_r \otimes A)$ of size $M \times N$ is not transferred wholly, but only its submatrix A (of size $k \times c$) is used in the multiplication, so not only does this dimensionality reduction method provide a theoretical speedup, it also has the side effect of making the whole operation even more optimized by requiring less data to be transferred from host to device memory.

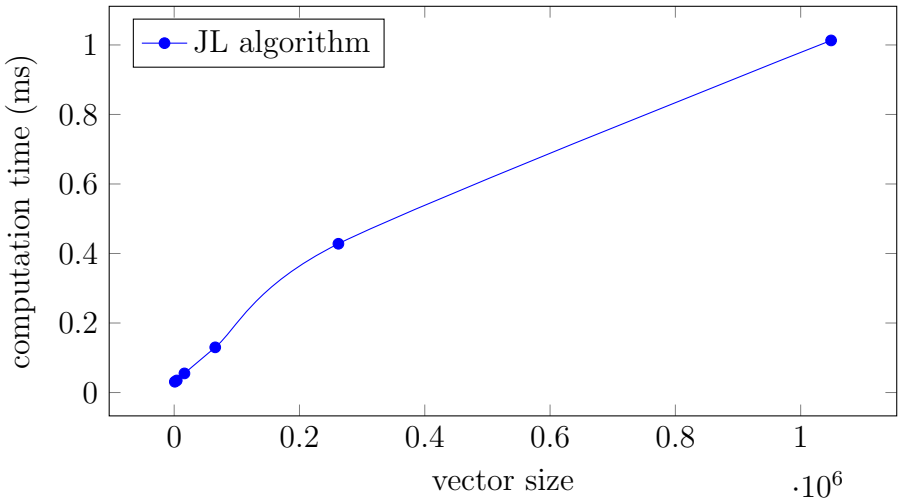

Figure 17: Plot of the single-JL algorithm's performance with $r = 32$.

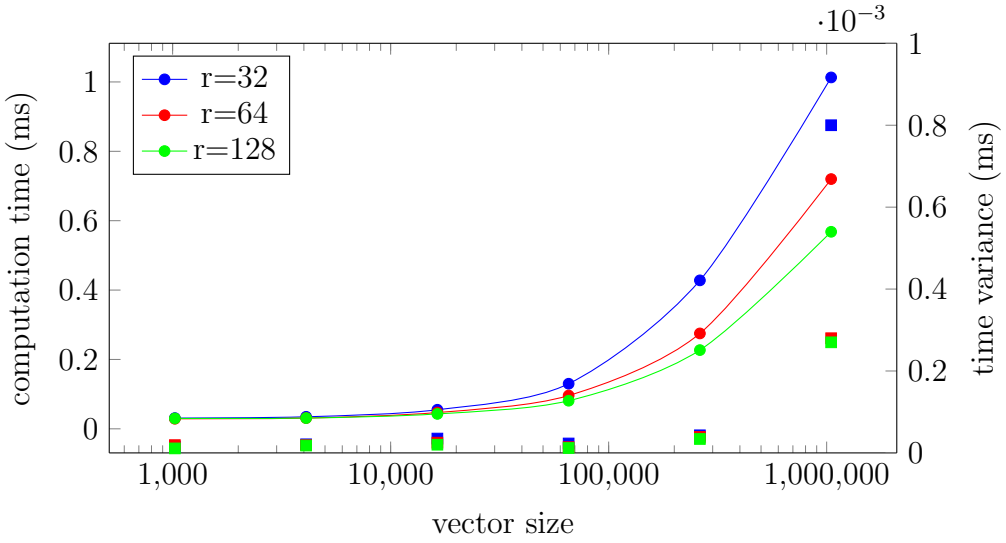

Figure 18: JL algorithm times calculated with different values of r (log scale).

We can also notice from the plot in figure 17 that the execution times are no longer perfectly linear, and it becomes increasingly advantageous to use this algorithm over the previous ones as the number dimensions of the input vectors increase.

Figure 18 clearly shows that tweaking the value of r significantly changes execution times. In this case, a higher number of blocks, leads to faster execution times. This behaviour could be explained by looking at the characteristics of the Gemm algorithm and at the structure of matrix X we obtain from the initial vector: x is split into r blocks, each making up a column of X. Thus, if the number of blocks increases, the number of columns of the matrix will decrease, and the Gemm algorithm executed during the multiplication is more efficiently parallelized, with each result tile having

vector size	r$\bar{3}$2 execution time (ms)	variance	data transfer(%)
1,048,576	1.013	0.0008	88.5
262,144	0.428	0.00004	86.5
65,536	0.13	0.00002	72.9
16,384	0.055	0.00004	71.4
4,096	0.0347	0.00002	69.2
1,024	0.031	0.00002	71.4

vector size	r$\bar{6}$4 execution time (ms)	variance	data transfer(%)
1,048,576	0.72	0.00028	88.1
262,144	0.275	0.00004	84.2
65,536	0.096	0.00001	72.3
16,384	0.047	0.00003	71.9
4,096	0.031	0.00002	70.5
1,024	0.029	0.00002	72.5

vector size	r$\bar{1}$28 execution time (ms)	variance	data transfer(%)
1,048,576	0.568	0.00027	88.6
262,144	0.227	0.00003	86
65,536	0.081	0.00001	72.2
16,384	0.043	0.00002	75.1
4,096	0.031	0.00002	73.3
1,024	0.03	0.00001	73.9

shorter sections of the input matrices to do calculations on.

When analyzing the program execution and library calls with the Nvidia profiler, one can notice how, given the same size of the inputs M and N, both the multiplication and data transfer times decrease as r increases. So it's probably a safe assumption to say that cuBLAS is able to more efficiently execute tasks when dealing with matrices that are closer to a square matrix, in terms of proportions between the number of rows and columns.

6.1.4 Batch JL algorithm

Finally, we will examine the computational performance of the batch-JL algorithm, described in section 5.4: what was stated above about the relationship between M, N, c and k is also applied here, and the value of r is also set to the three different values tested on the previous algorithm. Another thing worth mentioning is that, as with the previous standard batch algorithm (6.1.2), the batch size is set to a fixed $b = 256$.

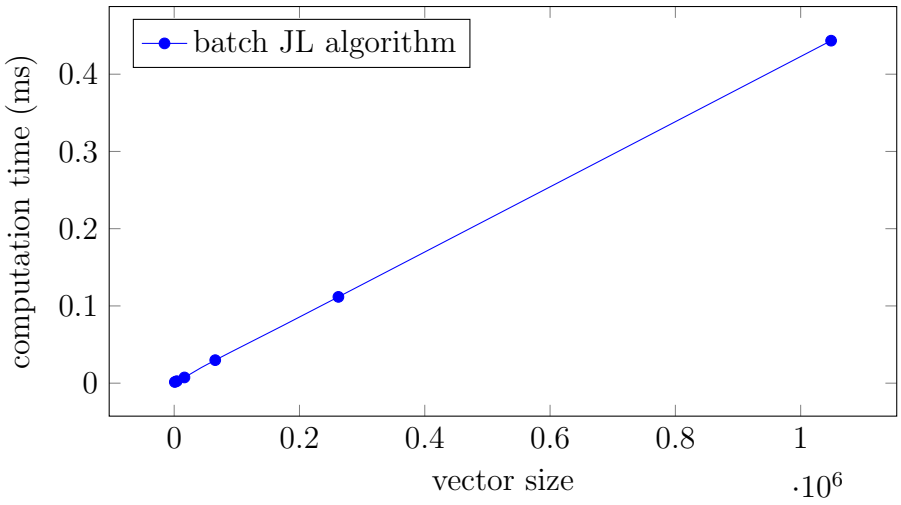

Figure 19: Plot of batch JL algorithm: the values shown represent the dimensionality reduction time for a single vector of the batch ($r = 32$).

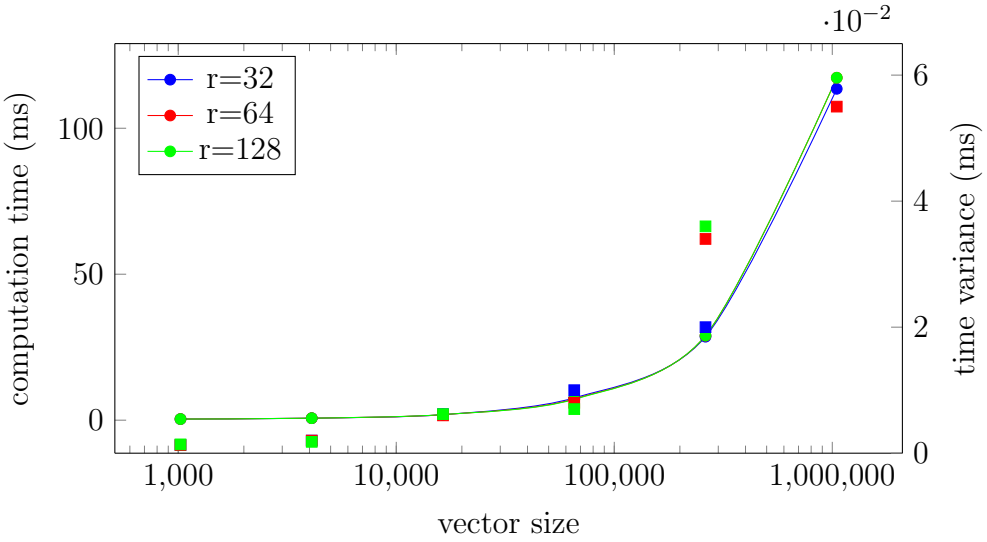

Figure 20: Logarithmic plot of batch JL algorithm. Times are relative to the execution time of the entire batch (not divided by 256). We can clearly observe how, in this case, changing the value of r does not produce significant differences in execution times.

This is the algorithm that yields the best results, although the percentage of time spent during data transfers still dominates execution time, independently of N. This tells us that having more data (we are dealing with $b = 256$ vectors instead of just one) does not increase the time spent to compute the products as much as it increases the time spent during data upload/download. Obviously, since many vectors need to be transferred between host and device, the advantage of only having to copy matrix A that we had in the single JL algorithm is greatly reduced, if not completely lost.

Looking at the Nvidia profiler, we can notice how, compared to the previous JL algorithm that only deals with a single vector, the average time for both Gemm operations and data transfers is not proportionate to the 256 multiplicative factor given by the number of inputs in the batch. This suggests that time is gained because it's a lot more efficient to start a single cuBLAS operation for a large amount of data than to instance multiple smaller operations to achieve the same result. It's also worth noting that the Gemm multiplication library call differs between the two algorithms: the single JL algorithm uses `volta_sgemm_32x32_sliced1x4_nn`, while the batch one calls `turing_s884gemm_128x128_ldg8_f2f_nn` (though I could not find anything more specific about these two library functions).

Again, looking at the plots of figures 19-20, execution times are almost perfectly linear. This can be attributed to the linearly increasing data size and the fact that the algorithm, like the previous ones, is memory limited, always spending more than 95% of execution time on i/o operations between device and host memory.

It's also quite interesting to note how, in this case, increasing r provides almost no improvement in the performance at all. My guess is this happens because we are dealing with a batch containing a lot of vectors, so the matrix X involved in the operations (see figure 13 is too unbalanced along one of its dimensions in a way that changing the number r of blocks in which vectors are split does not make enough of a difference in the parallelization and execution efficiency of the cuBLAS library.

vector size	r$\bar{3}$2 batch time (ms)	variance	vector ex.time (ms)	data transfer(%)
1,048,576	113.5	0.08	0.4434	96.6
262,144	28.59	0.02	0.1117	96.3
65,536	7.646	0.01	0.0298	96.7
16,384	1.893	0.0062	0.0074	95.3
4,096	0.678	0.002	0.0026	94.1
1,024	0.383	0.0013	0.0015	95.34

In the previous section we hypothesized that having a balanced proportion between rows and columns might be an important factor in increasing the efficiency of cuBLAS operations and thus decreasing execution times. We can observe that the number of rows of matrix A depends both by r and M ($k = M/r$). Let's now see what happens when changing the value of this parameter.

In this part of the experiment, we will fix the values of $N = 2^{20}$ and $r = 128$, while

vector size	r̄64 batch time (ms)	variance	vector ex.time (ms)	data transfer(%)
1,048,576	117.3	0.055	0.458	96.8
262,144	28.94	0.034	0.113	97.2
65,536	7.286	0.008	0.0285	96.4
16,384	1.904	0.006	0.0074	94.8
4,096	0.688	0.002	0.0027	93.5
1,024	0.391	0.0013	0.0015	94.7

vector size	r̄128 batch time (ms)	variance	vector ex.time (ms)	data transfer(%)
1,048,576	117.3	0.08	0.458	96.8
262,144	28.98	0.036	0.1132	97.2
65,536	7.147	0.007	0.0279	96.9
16,384	1.923	0.0062	0.0075	94.8
4,096	0.684	0.0018	0.0027	92.5
1,024	0.385	0.0014	0.0015	93.2

M will go from 2^{10} to 2^{20}.

Here are the results:

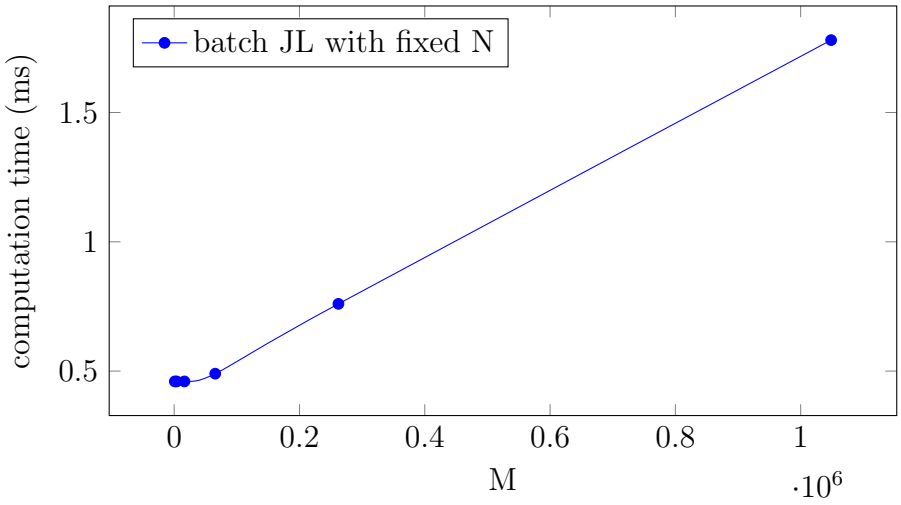

Figure 21: Plot of batch JL algorithm with different values of M, $N = 2^{20}$ and $r = 128$. The values shown represent the dimensionality reduction time for a single vector of the batch.

We can conclude a few things from the plots 21 and 22: first of all, we can see that once again, a higher value of r leads to a faster execution time. So we can conclude that, with the right proportions of the matrices, increasing r improves the performance quite consistently.

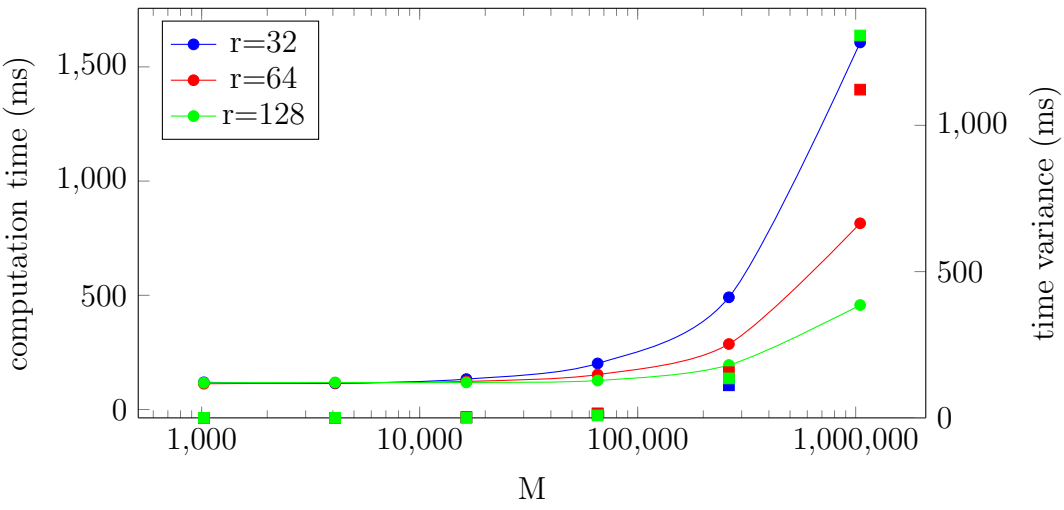

Figure 22: Plot of batch JL algorithm with differing values of M and r.

The other observation we can make is that, no matter the value of r, up to a certain point, we can increase the value of M without having any impact on the performance of the algorithm: the execution time stays very stable, up to $M = 2^{16}$, even though we are dealing with an ever-increasing amount of data. Remember, matrix A is a $k \times c$ matrix, where $k = M/r$ and $c = N/c$, and M is increased compared to the previous algorithms, while N is set at the highest value ever set in the previous tests. Even when looking at the output of the Nvidia profiler, it's difficult to precisely tell why this occurs, other than, again, thanks to an optimized parallelization by cuBLAS made possible by the different size of the inputs. Obviously, when talking about dimensionality reduction, our aim would be to decrease the value M of the number of output dimensions. Nevertheless, this unexpected behaviour was worth highlighting, if only to appreciate the exceptional performance of tensor core units.

Looking at the results for the highest values of M, we can notice how the data size is so huge that the multiplication time percentage increases up to the point of taking up most of the execution time (data transfer time percentage decreases significantly). This also happens at the same threshold of input size where execution time begins to exponentially increase. Clearly, data transfer time must increase linearly with the input size, but the multiplication time increases exponentially in k (which is also supported by the theory [1]).

M	r$\bar{3}$2 batch time (ms)	variance	vector ex.time (ms)	data transfer(%)
1,048,576	1,607	1,479	6.28	40.2
262,144	491.37	111.4	1.91	51.2
65,536	201.8	14.2	0.79	72.7
16,384	133.2	2.4	0.52	91
4,096	114.4	0.14	0.45	96.5
1,024	119.2	0.06	0.46	96.8

M	r$\bar{6}$4 batch time (ms)	variance	vector ex.time (ms)	data transfer(%)
1,048,576	815.3	1,122	3.18	40.2
262,144	286.8	156	1.12	58.6
65,536	152.9	15.3	0.59	83.2
16,384	123.1	1.13	0.48	94.6
4,096	116.7	0.11	0.46	96.7
1,024	112.9	0.09	0.44	96.6

M	r$\bar{1}$28 batch time (ms)	variance	vector ex.time (ms)	data transfer(%)
1,048,576	457.5	1,307	1.78	50.08
262,144	194.9	135	0.76	71.5
65,536	127.09	8.3	0.49	90.6
16,384	118.64	0.56	0.46	96.7
4,096	118.72	0.084	0.46	96.8
1,024	117.22	0.086	0.46	96.9

6.2 Algorithm comparison

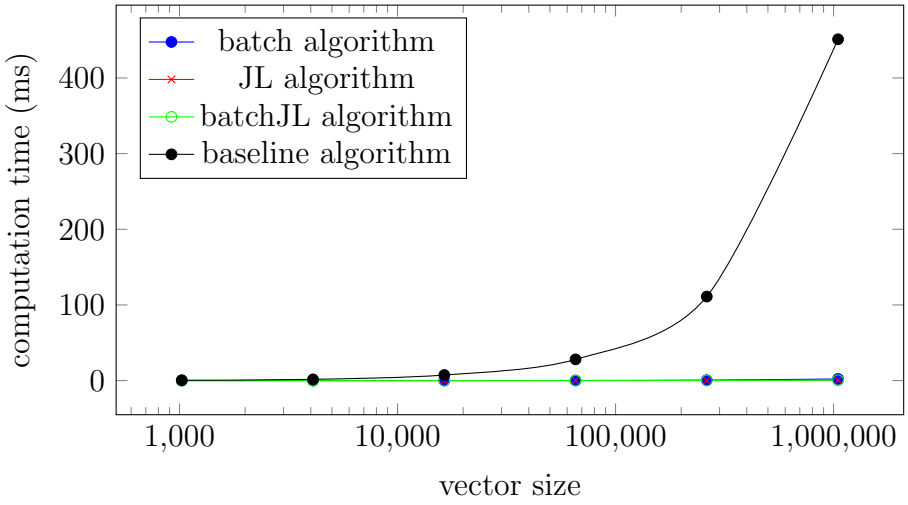

Figure 23: Time performance comparison between all four tested algorithms.

In this section, we will compare the time performance of the different algorithms we tested. The above figure clearly shows that the three algorithms that use TCUs are so fast that a proper comparison with the baseline is not really possible or meaningful. This plot suggests that the faster algorithms do not carry meaningful differences, but obviously this is just a consequence of the three operating at a completely difference timescale from the baseline.

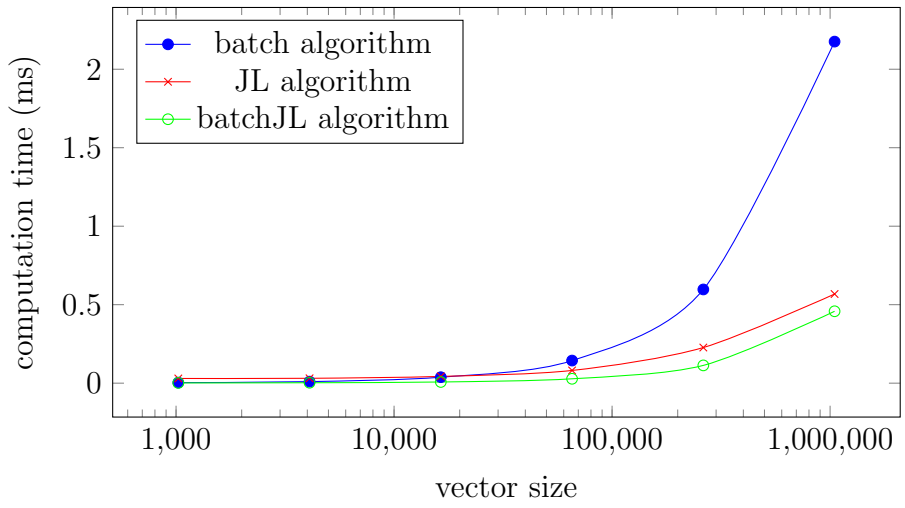

Figure 24: Comparison of the performance of the three fastest algorithms. A logarithmic scale is applied to the x axis in order to better show values on the lower end of the vector size.

As shown by figure 24, the time spent by the three algorithms to compute the

dimensionality reduction of a single vector is basically identical when considering low values of N. As the size of the inputs increases, the batch JL algorithm becomes more and more advantageous, though not by a huge margin when compared to the single JL version. Since multiplication times take a very little percentage of the total, we can conclude that transferring all the data in one go (the whole batch) and then performing the multiplication is a better strategy than alternating between data transfer and multiplication for each vector we need to reduce. The slower performance of the standard batch algorithm can be explained by simply observing that the size of the matrices involved is a lot higher ($M \times N$ and $N \times b$) compared to the JL algorithms ($k \times c$ and $c \times b(N/c)$).

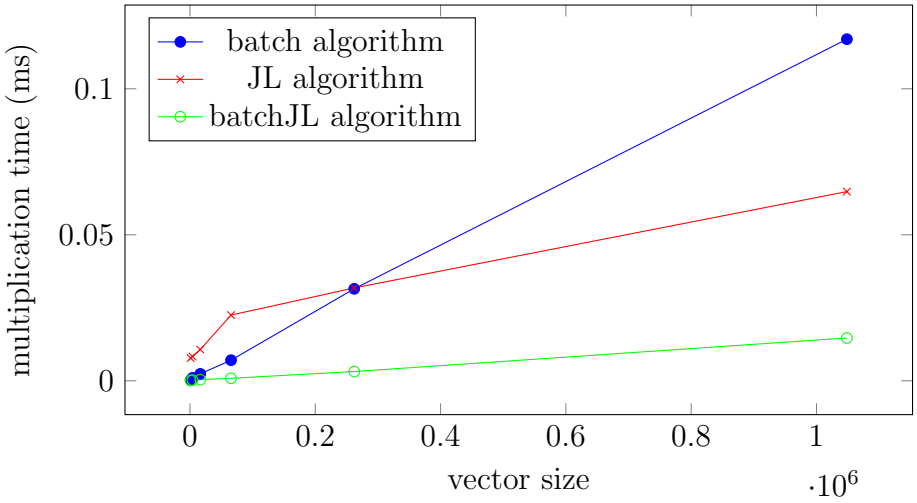

Figure 25: Plot of the three main algorithms' execution time, considering only the time spent on multiplications without the data transfer percentage.

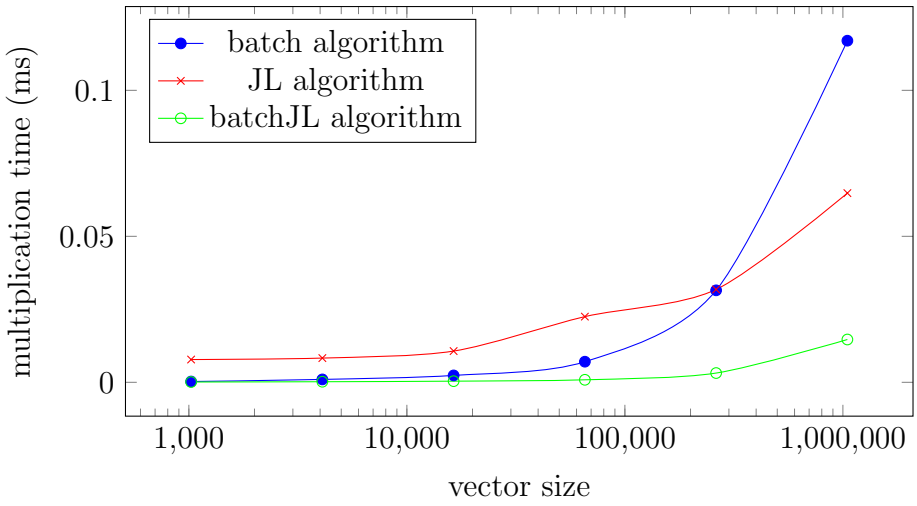

Figure 26: Log version of the previous plot.

N	batch (ms)	single JL (ms)	batchJL (ms)
1,048,576	0.117	0.0648	0.01466
262,144	0.0315	0.0318	0.00317
65,536	0.00706	0.0225	0.00086
16,384	0.00236	0.0107	0.00039
4,096	0.00099	0.0083	0.0002
1,024	0.00027	0.0078	0.0001

6.3 Quality performance measurements

In this section, we will analyze the quality of the results of our algorithms. This can be done by computing the norm of the reduced vectors and comparing it with the norm of their original respective. In particular, all tests were performed on unit vectors (vectors of norm equal to 1), so the values relative to the norm difference which are displayed in the plots of this chapter are relative to the reference value of 1.

The quality measurements will be performed by using the batch JL algorithm 5.4. In particular, we are interested in studying the relationships between certain parameters relative to the construction of the block identity matrix $(I_r \otimes A)$: we will investigate how the quality of the results changes depending on the values of the input and output dimensions of the vectors, as well as depending on the size of matrix A representing the JL transform. Since unit vectors will be used as inputs throughout all the tests, the norms of the result will need to be as close as possible to 1 in order for the results to be satisfactory.

We will also compare the results obtained by our proposed JL method to the ones measured from an implementation of the standard Achlioptas method, which simply requires to generate the transform matrix with the probability distributions described by Theorem 1 (from chapter 3) as the whole $M \times N$ input matrix. The difference between the two methods is that our JL algorithm generates the $k \times c$ matrix A and performs the multiplication with this smaller matrix, while Achlioptas uses the whole $M \times N$ array of values.

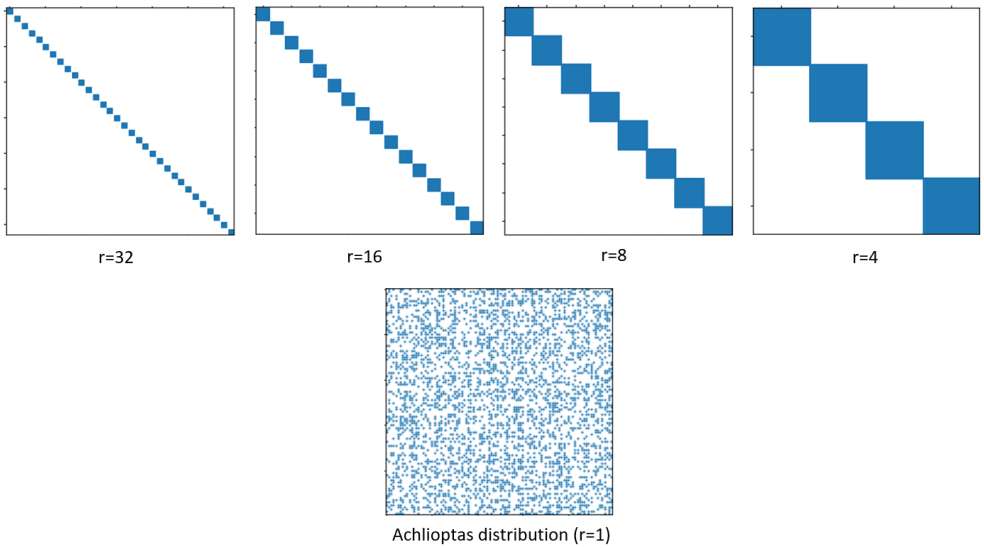

Figure 27: JL transform matrix sparsity visualization with different values of r. As the value decreases, the non-zero portion of the matrix increases. The blue squares are not actually fully dense, but feature the sparsity of the Achlioptas distribution (the one at (4), since the other one does not have non-zero items), also shown at the bottom of the image.

Considering the new JL method, it's important to observe that by tweaking the number of blocks r, the values of $k = M/r$ and $c = N/r$ also change. When considering a fixed value of M and N, as the value of r decreases, the two parameters specifying the dimensions of A increase, and the matrix $(I_r \otimes A)$ is composed by a decreasing number of larger A matrices, becoming less sparse (figure 27): as k and c increase, our transform and an Achlioptas transform with the same values of M, N become increasingly similar in terms of sparseness (when $r = 1$ they are identical). We will see whether or not these traits have an impact in the quality.

First, let's analyze the JL algorithm by itself: we will see how the quality changes when keeping $M = 2^{10}$ and $N = 2^{20}$ fixed, while tuning the value of k. As we know, changing this value implies a change in the value of r, and since r must be integer and M, N are powers of 2, we will use values of k from 2^3 to 2^{10}. Both of the probability distributions described in Theorem 1 will be tested.

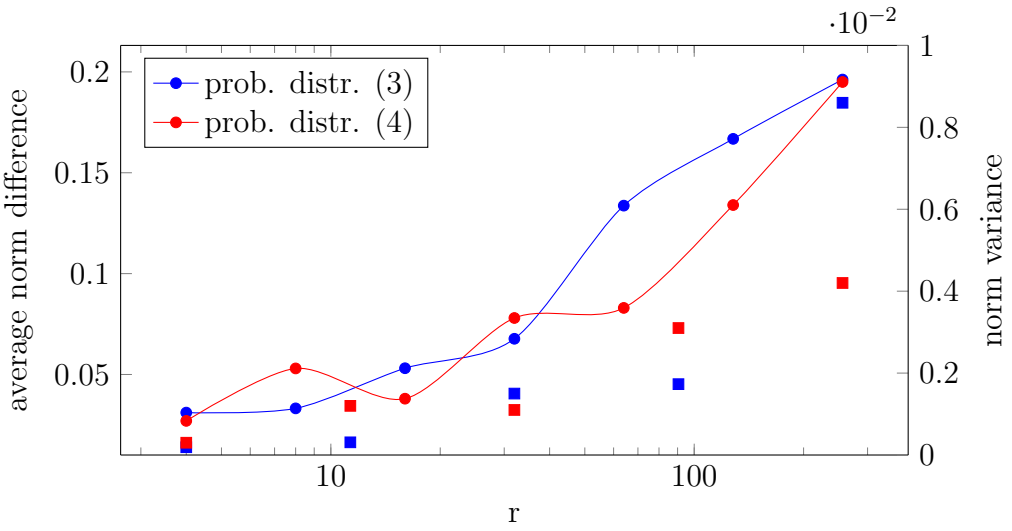

Figure 28: Quality plot with fixed M, N dimensions representing results from both JL embedding methods described in the paper.

As we can see from figure 28, there does not seem to be a particularly significant difference between the two methods, though distribution (4) seems to achieve slightly better results, on average. What does make a difference is the value of r: as the blocks increase and the size of matrix A decreases, the norm difference between inputs and outputs gets noticeably higher. Remembering that we are dealing with unit vectors, a difference of 0.2 represents a 20% delta between the two norms, which is too high of an error margin. At the lower end of the value of r, we have a more satisfying quality of results: the algorithm is able to produce output vectors that have less than a 5% difference from their corresponding inputs while reducing their dimensions by 10 orders of magnitude.

It is also worth recalling chapter 3: the requirements given by Theorem 1 state

r	k	norm diff.	variance
4	256	0.0311	0.00019
8	128	0.0332	0.00031
16	64	0.0531	0.0015
32	32	0.0677	0.00173
64	16	0.1337	0.0086
128	8	0.1668	0.01166
256	4	0.1962	0.01155

(a) probability distribution (3)

r	k	norm diff.	variance
4	256	0.027	0.0003
8	128	0.053	0.0012
16	64	0.038	0.0011
32	32	0.078	0.0031
64	16	0.083	0.0042
128	8	0.134	0.0142
256	4	0.195	0.0124

(b) probability distribution (4)

that the quality of the result is directly correlated to the output dimensions $M \geqslant \frac{4+2\beta}{\epsilon^2/2-\epsilon^3/3}\log n$. Consider that when choosing, for example, $n = 256$ (the batch size), $\beta = 0.9$ and $\varepsilon = 0.2$, which are not very strict parameters, we get $M \geqslant 1855$. We can clearly see that having $M = 1024$ in the previous test does not leave a lot of room for choosing big values of k, but we still managed to obtain a decent quality.

Another thing to note is that, at the end of the previous section we concluded that having a bigger value of r (small k) improves execution time. This means that choosing the values for these parameters requires to make a tradeoff between execution time and quality of the results.

The next plots will show how the quality changes when the values of the input matrices and vectors are modified. We are interested in calculating how the performance changes depending on the difference between the input and output dimension. Intuitively, having result vectors that are more similar in length to their corresponding originals should yield better results, and thus, a smaller difference between the vectors' norms.

From figure 29 it's clear that, as Achlioptas demonstrated, when considering norm differences, higher output dimensions produce the best results, regardless of the difference from the input, and the quality only depends on the number of output dimensions, so our intuition has misled us in this instance.

Testing while changing parameters M and N also enables us to compare the results of our JL method to the original Achlioptas method. This is because the only values that we can change in the Achlioptas method are the input and output dimensions of the vectors to reduce. Since we just observed that changing the dimensions of input vectors does not impact the results (figure 29), we can limit our testing to different values of output dimensions M.

We will evaluate the quality of the practical results given by both algorithms after the reduction of input unit vectors having $N = 65536$ entries to a number of dimension ranging from $M = 1024$ to $M = 8192$. Regarding the measurements on our JL algorithm, we will perform multiple tests with different numbers of blocks $r = 64$, $r = 32$

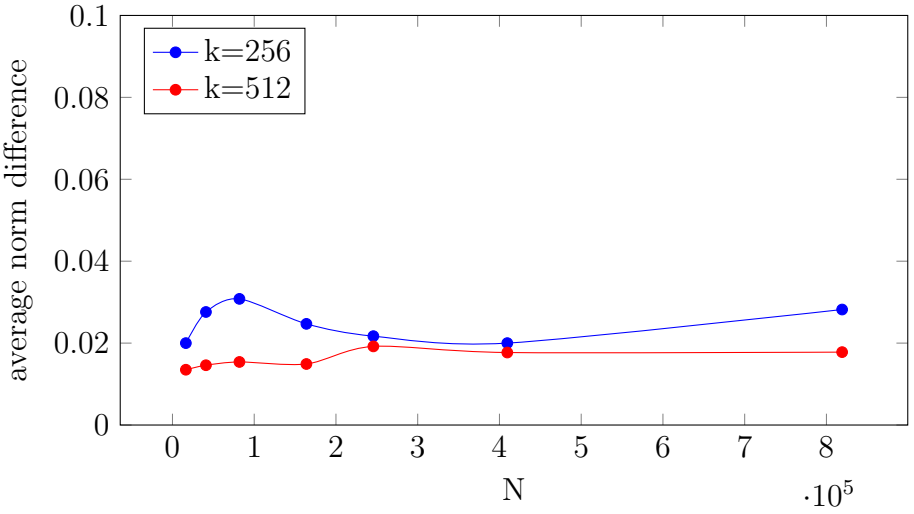

Figure 29: Plot of algorithm quality with differing values for the input vector's dimension. M is fixed at a value of 2^{10}, and k is at 512.

N	norm difference	variance
16,384	0.02	0.0001
40,960	0.0276	0.0002
81,920	0.0308	0.0007
163,840	0.0247	0.0002
245,760	0.0217	0.0003
409,600	0.02	0.0002
819,200	0.0282	0.0004

(a) k=256

N	norm difference	variance
16,384	0.0135	0.0001
40,960	0.0146	0.0001
81,920	0.0154	0.0001
163,840	0.0149	0.0001
245,760	0.0192	0.0001
409,600	0.0177	0.0001
819,200	0.0178	0.0001

(b) k=512

and $r = 16$.

Given our previous observations, the expected results are that the Achlioptas method, though slower in terms of execution time, should provide the best results from a qualitative standpoint, while the norm differences between inputs and outputs of our JL transform should decrease with the decrease of the number of blocks r. Also, the overall quality should increase as stated by the theory (in Theorem 1, increasing k means using more strict ε and β quality parameters).

The results of figure 30 seem to support all of our previous claims: norm differences decrease as the number of blocks in the block identity matrix decreases, and the quality of results also increases with the value of M. Achlioptas' method gives the best quality by an almost constant factor (given a fixed value of r), but the quality achieved by the proposed JL transform is still satisfactory, save for the case of values of M being at their lowest while the values of r at their highest.

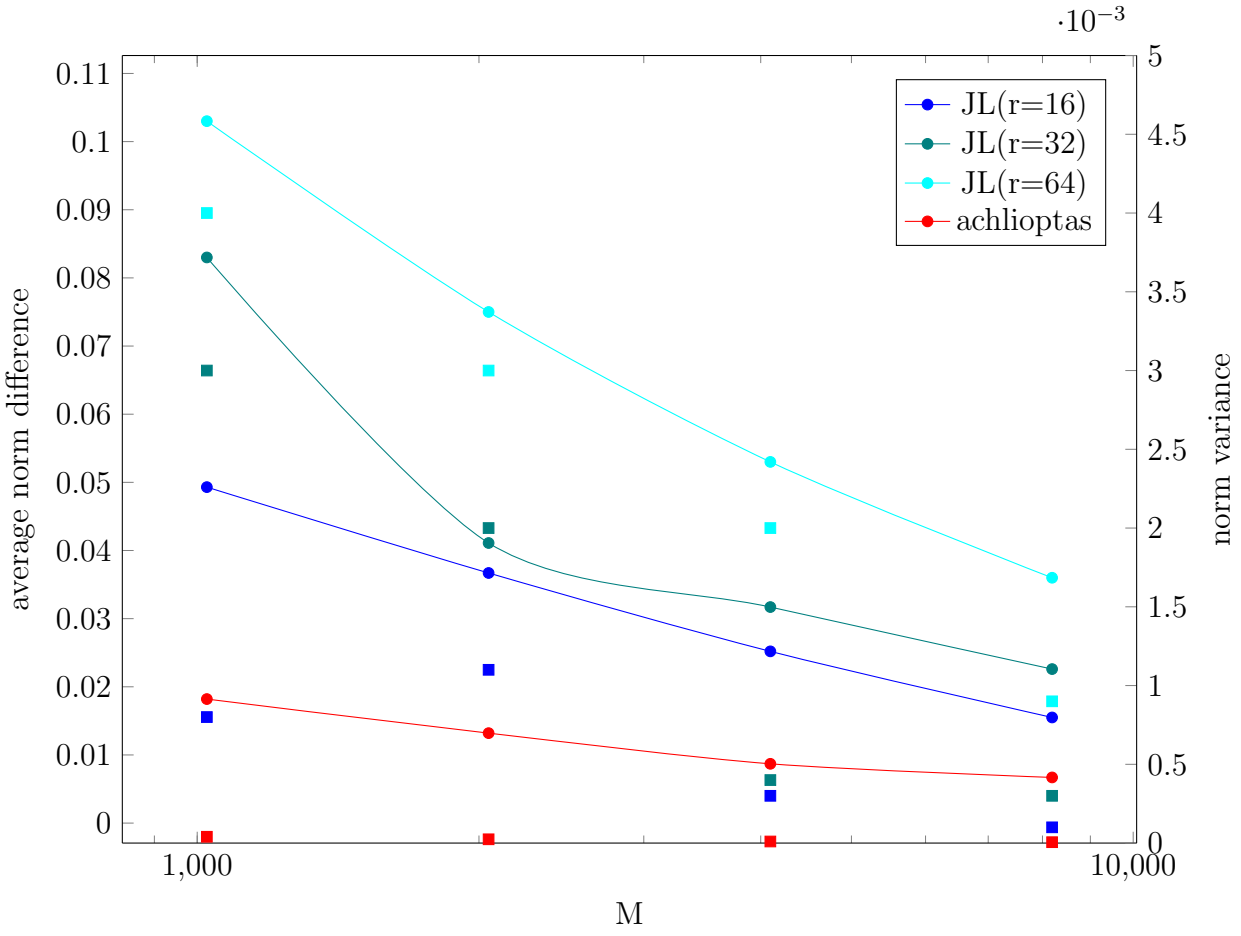

Figure 30: Comparison between the quality of Achlioptas' method and our proposed JL transform, with fixed input dimensions $N = 65536 = 2^{16}$.

7 Conclusion

In this thesis, we have seen how tensor core units can be used in a field different from the one they were originally designed for. We successfully developed several algorithms to speed up dimensionality reduction using these hardware accelerators, showing the effectiveness of TCUs compared to standard parallel strategies.

In particular, we studied the features and effectiveness of the JL dimensionality reduction method proposed by Ahle and Silvestri in [1], which showed promising results, while also expanding and improving its time performance with the batched version of the algorithm.

Crucially, the proposed method also yields good results when considering the quality of the embeddings, although they are generally less accurate than other methods that use the same probability distributions (Achlioptas method [5]), but not by a huge margin. It is also worth noting that the other methods are worse than ours when talking about execution time performance. As shown in the previous chapter, there is a tradeoff to be found between execution time and quality of the results, which is governed by various parameters; this feature could make our JL algorithms more versatile to be used in practice, allowing developers to tweak these parameters depending on the needs of the application in question.

An aspect of the performance which was not explored in this thesis is the measurement of quality with respect to the pairwise distance between points. In general, we would like Euclidean distances to be preserved as much as possible after the reduction of the inputs. The two lemmas of Johnson and Lindenstrauss equivalently define the JL property of probability distributions both in terms of preservation of pairwise distance between points and preservation of the norms of the single vectors. This suggests that the JL matrices used in our testing should, in theory, maintain both aspects of the quality, though further tests would be required to confirm this behaviour in our proposed method.

It would also have been interesting to explore the execution and quality performance of the algorithms when working on real datasets instead of only testing them on random unit vectors.

References

[1] T. D. Ahle and F. Silvestri, "Similarity search with tensor core units," 2020.

[2] L. Van Der Maaten, E. Postma, J. Van den Herik, *et al.*, "Dimensionality reduction: a comparative," *J Mach Learn Res*, vol. 10, no. 66-71, p. 13, 2009.

[3] C. B. Freksen, "An introduction to johnson-lindenstrauss transforms," *CoRR*, vol. abs/2103.00564, 2021.

[4] P. Frankl and H. Maehara, "The johnson-lindenstrauss lemma and the sphericity of some graphs," *Journal of Combinatorial Theory, Series B*, vol. 44, no. 3, pp. 355–362, 1988.

[5] D. Achlioptas, "Database-friendly random projections: Johnson-lindenstrauss with binary coins," *Journal of Computer and System Sciences*, vol. 66, no. 4, pp. 671–687, 2003. Special Issue on PODS 2001.

[6] R. Chowdhury, F. Silvestri, and F. Vella, *A Computational Model for Tensor Core Units*, p. 519–521. New York, NY, USA: Association for Computing Machinery, 2020.

[7] D. M. Kane and J. Nelson, "Sparser johnson-lindenstrauss transforms," *J. ACM*, vol. 61, jan 2014.

[8] "Nvidia tesla v100 gpu architecture," Aug 2017.

[9] Nvidia, "Programming guide :: Cuda toolkit documentation." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[10] J. Appleyard and S. Yokim, "Programming tensor cores in cuda 9 — nvidia developer blog." https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/, Oct 2017.

[11] Nvidia, "Matrix multiplication background user guide." https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

[12] Nvidia, "Cuda-samples/samples/cudatensorcoregemm at master · nvidia/cuda-samples." https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3_CUDA_Features/cudaTensorCoreGemm.

[13] M. Harris, "How to implement performance metrics in cuda c/c++." https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/.