Mathematical Structures and Modeling
2018. N. 2(46). PP. 107–117

UDC 512:004.6
DOI: 10.25513/2222-8772.2018.2.107-117

HOW TO STORE TENSORS IN COMPUTER MEMORY: AN OBSERVATION

Martine Ceberio
Ph.D. (Phys.-Math.), Associate Professor, e-mail: mceberio@utep.edu
Vladik Kreinovich
Ph.D. (Phys.-Math.), Professor, e-mail: vladik@utep.edu

University of Texas at El Paso, El Paso, Texas 79968, USA

Abstract. In this paper, after explaining the need to use tensors in computing, we analyze the question of how to best store tensors in computer memory. Somewhat surprisingly, with respect to a natural optimality criterion, the standard way of storing tensors turns out to be one of the optimal ones.

Keywords: Tensors, computing, computer memory.

1. Why Tensors: A Reminder

Why tensors. One of the main problems of modern computing is that:

- we have to process large amounts of data;
- and therefore, long time is required to process this data.

A similar situation occurred in the 19 century physics:

- physicists had to process large amounts of data;
- and, because of the large amount of data, a long time is required to process this data.

We will recall that in the 19 century, the problem was solved by using tensors. It is therefore a natural idea to also use tensors to solve the problems with modern computing.

Tensors in physics: a brief reminder. Let us recall how tensors helped the 19 century physics; see, e.g., [6]. Physics starts with measuring and describing the values of different physical quantities. It goes on to equations which enable us to predict the values of these quantities.

A measuring instrument usually returns a single numerical value. For some physical quantities (like mass m), the single measured value is sufficient to describe the quantity. For other quantities, we need several values. For example, we need three components E_x, E_y, and E_z to describe the electric field at a given point. To describe the tension inside a solid body, we need even more values: we need 6 values $\sigma_{ij} = \sigma_{ji}$ corresponding to different values $1 \leqslant i, j \leqslant 3$: σ_{11}, σ_{22}, σ_{33}, σ_{12}, σ_{23}, and σ_{13}.

The problem was that in the 19 century, physicists used a separate equation for each component of the field. As a result, equations were cumbersome and difficult to solve.

The main idea of the tensor approach is to describe all the components of a physical field as a single mathematical object:

- a vector a_i,
- or, more generally, a tensor a_{ij}, a_{ijk}, ...

As a result, we got simplified equations — and faster computations.

It is worth mentioning that originally, mostly vectors (rank-1 tensors) were used. However, the 20 century physics has shown that higher-order matrices are also useful. For example:

- matrices (rank-2 tensors) are actively used in quantum physics,
- higher-order tensors such as the rank-4 curvature tensor R_{ijkl} are actively used in the General Relativity Theory.

From tensors in physics to computing with tensors. As we have mentioned earlier, 19 century physics encountered a problem of too much data. To solve this problem, tensors helped.

Modern computing suffers from a similar problem. A natural idea is that tensors can help. Two examples justify our optimism:

- modern algorithms for fast multiplication of large matrices; and
- quantum computing.

2. Modern algorithm for multiplying large matrices

In many data processing algorithms, we need to multiply large-size matrices:

$$\begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \ldots & \ldots & \ldots \\ a_{n1} & \ldots & a_{nn} \end{pmatrix} \begin{pmatrix} b_{11} & \ldots & b_{1n} \\ \ldots & \ldots & \ldots \\ b_{n1} & \ldots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & \ldots & c_{1n} \\ \ldots & \ldots & \ldots \\ c_{n1} & \ldots & c_{nn} \end{pmatrix}; \tag{1}$$

$$c_{ij} = a_{i1} \cdot b_{1j} + \ldots + a_{ik} \cdot b_{kj} + \ldots + a_{in} \cdot b_{nj}. \tag{2}$$

There exist many efficient algorithms for matrix multiplication.

The problem is that for large matrix size n, there is no space for both A and B in the fast (cache) memory. As a result, the existing algorithms require lots of time-consuming data transfers ("cache misses") between different parts of the memory.

An efficient solution to this problem is to represent each matrix as a matrix of blocks; see, e.g., [2, 10]:

$$A = \begin{pmatrix} A_{11} & \ldots & A_{1m} \\ \ldots & \ldots & \ldots \\ A_{m1} & \ldots & A_{mm} \end{pmatrix}, \tag{3}$$

then

$$C_{\alpha\beta} = A_{\alpha 1} \cdot B_{1\beta} + \ldots + A_{\alpha\gamma} \cdot B_{\gamma\beta} + \ldots + A_{\alpha m} \cdot B_{m\beta}. \tag{4}$$

Comment. For general arguments about the need to use non-trivial representations of 2-D (and multi-dimensional) objects in the computer memory, see, e.g., [21,22].

In the above idea,

- we start with a large matrix A of elements a_{ij};
- we represent it as a matrix consisting of block sub-matrices $A_{\alpha\beta}$.

This idea has a natural *tensor interpretation:*

- each element of the original matrix is now represented as
- an (x, y)-th element of a block $A_{\alpha\beta}$,
- i.e., as an element of a rank-4 tensor $(A_{\alpha\beta})_{xy}$.

So, in this case, an increase in tensor rank improves efficiency.

Comment. Examples when an increase in tensor rank is beneficial are well known in physics: e.g., a representation of a rank-1 vector as a rank-2 spinor works in relativistic quantum physics [6].

Quantum computing as computing with tensors. Classical computation is based on the idea of a *bit:* a system with two states 0 and 1. In quantum physics, due to the superposition principle, we can have states

$$c_0 \cdot |0\rangle + c_1 \cdot |1\rangle \tag{5}$$

with complex values c_0 and c_1; such states are called *quantum bits*, or *qubits*, for short.

The meaning of the coefficients c_0 and c_1 is that they describe the probabilities to measure 0 and 1 in the given state: $\mathrm{Prob}(0) = |c_0|^2$ and $\mathrm{Prob}(1) = |c_1|^2$. Because of this physical interpretations, the values c_1 and c_1 must satisfy the constraint $|c_0|^2 + |c_1|^2 = 1$.

For an n-(qu)bit system, a general state has the form

$$c_{0\ldots00} \cdot |0\ldots00\rangle + c_{0\ldots01} \cdot |0\ldots01\rangle + \ldots + c_{1\ldots11} \cdot |1\ldots11\rangle. \tag{6}$$

From this description, one can see that each quantum state of an n-bit system is, in effect, a *tensor* $c_{i_1\ldots i_n}$ of rank n.

In these terms, the main advantage of quantum computing is that it can enable us to store the entire tensor in only n (qu)bits. This advantage explains the known efficiency of quantum computing. For example:

- we can search in an unsorted list of n elements in time $\sqrt{n}$ — which is much faster than the time n which is needed on non-quantum computers [8, 9, 15];
- we can factor a large integer in time which does not exceed a polynomial of the length of this integer — and thus, we can break most existing cryptographic codes like widely used RSA codes which are based on the difficulty of such a factorization on non-quantum computers [15, 18, 19].

Tensors to describe constraints. A general constraint between n real-valued quantities is a subset $S \subseteq R^n$. A natural idea is to represent this subset block-by-block — by enumerating sub-blocks that contain elements of S.

Each block $b_{i_1\ldots i_n}$ can be described by n indices $i_1, \ldots, i_n$. Thus, we can describe a constraint by a boolean-valued tensor $t_{i_1\ldots i_n}$ for which:

- $t_{i_1\ldots i_n}$ ="true" if $b_{i_1\ldots i_n} \cap S \neq \emptyset$; and
- $t_{i_1\ldots i_n}$ ="false" if $b_{i_1\ldots i_n} \cap S = \emptyset$.

Processing such constraint-related sets can also be naturally described in tensor terms.

This representation speeds up computations; see, e.g., [3,4].

Computing with tensors can also help physics. So far, we have shown that tensors can help computing. It is possible that the relation between tensors and computing can also help physics.

As an example, let us consider Kaluza-Klein-type high-dimensional space-time models of modern physics; see, e.g., [7, 11–13, 16, 20]. Einstein's original idea [5] was to use "tensors" with integer or circular values to describe these models. From the mathematical viewpoint, such "tensors" are unusual. However, in computer terms, integer or circular data types are very natural: e.g., circular data type means fixed point numbers in which the overflow bits are ignored. Actually, from the computer viewpoint, integers and circular data are even more efficient to process than standard real numbers.

Remaining open problem. One area where tensors naturally appear is an efficient Taylor series approach to uncertainty propagation; see, e.g., [1,14,17]. Specifically, the dependence of the result y on the inputs $x_1, \ldots, x_n$ is approximated by the Taylor series:

$$y = c_0 + \sum_{i=1}^{n} c_i \cdot x_i + \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij} \cdot x_i \cdot x_j + \ldots \tag{7}$$

The resulting tensors $c_{i_1\ldots i_r}$ are symmetric:

$$c_{i_1\ldots i_r} = c_{\pi(i_1)\ldots\pi(i_r)} \tag{8}$$

for each permutation π. As a result, the standard computer representation leads to a $r!$ duplication. An important problem is how to decrease this duplication.

3. How to Store Tensors in Computer Memory

Need to store values in computer memory. The computer memory is 1-D, so whatever multi-dimensional object we describe, its components are stored sequentially. What is the best way to arrange 2-D and higher-dimensional data in a computer memory?

Storing 2-D values in computer memory: towards formalization of the problem. Let us describe this problem in precise terms. We will start this description with the simplest case of 2-D objects.

Storing 2-D object, with components a_{ij}, $1 \leqslant i, j \leqslant n$, means assigning, to each pair (i, j), the cell number $f(i, j)$ in such a way that different pairs (i, j) correspond to different cell numbers $f(i, j)$.

So, to describe a storing arrangement, we must describe a function

$$f : \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\} \to N \tag{9}$$

that maps each pair of integers $i, j \in \{1, 2, \ldots, n\}$ into a natural number.

How to gauge the quality of a memory arrangement? motivations. It is desirable to arrange the storage in such a way that neighboring elements of a 2-D object are located in the memory as close to each other as possible. Neighboring elements are elements (i, j) and (i', j') for which $|i - i'| \leqslant 1$ and $|j - j'| \leqslant 1$. Thus, we can gauge the quality of the memory arrangement by the largest distance between the locations of neighboring points.

As a result, we arrive at the following numerical characteristics of the quality of different memory arrangements f.

How to gauge the quality of a memory arrangement? a formula. The quality of a memory arrangement f is described by the value

$$C(f) \stackrel{\text{def}}{=} \max\{|f(i, j) - f(i', j')| : |i - i'| \leqslant 1, |j - j'| \leqslant 1\}. \tag{10}$$

The smaller this value, the better. Thus, we are interested in finding the arrangement with the smallest possible value of the quantity $C(f)$.

Standard memory arrangement. Before we start analyzing possible memory arrangements, let us recall the standard one. In the standard programming arrangement of a 2-D array, the values are stored row by row:

- first, we have elements of the first row,

$$f(1, 1) = 1, f(1, 2) = 2, \ldots, f(1, n) = n; \tag{11}$$

- then, we have elements of the second row,

$$f(2, 1) = n + 1, f(2, 2) = n + 2, \ldots, f(2, n) = n + n = 2n; \tag{12}$$

- ...
- the elements of the k-th row are store at

$$f(k, 1) = (k - 1) \cdot n + 1, f(k, 2) = (k - 1) \cdot n + 2, \ldots,$$

$$f(k, n) = (k - 1) \cdot n + n = k \cdot n; \tag{13}$$

- ...
- finally, the elements of the last (n-th) row are stored at locations

$$f(n, 1) = (n - 1) \cdot n + 1, f(n, 2) = (n - 1) \cdot n + 2, \ldots,$$

$$f(n, n) = (n - 1) \cdot n + n = n^2. \tag{14}$$

Quality of the standard memory arrangement. What is the value of the quantity $C(f)$ for the standard memory arrangement f? In other words, how far away from each other can neighboring elements (i, j) and (i', j') be located in the computer memory?

If these two elements are in the same row, i.e., if $i = i'$, then these neighboring elements (i, j) and (i, j'), with $|j - j'| = 1$, are neighbors in the memory as well:

$$|f(i, j) - f(i, j')| = |j - j'| = 1. \tag{15}$$

If these two elements are in the neighboring rows, $|i - i'| = 1$ and $|j - j'| \leqslant 1$, then we get

$$f(i, j) - f(i', j') = ((i - 1) \cdot n + j) - ((i' - 1) \cdot n + j') =$$
$$= (i - i') \cdot n + (j - j'). \tag{16}$$

Here,

$$|f(i, j) - f(i', j')| = |(i - i') \cdot n + (j - j')| =$$
$$= |n + (j - j')| \leqslant n + |j - j'| \leqslant n + 1. \tag{17}$$

Thus, for the standard memory arrangement f, the largest distance $C(f)$ between the memory locations of neighboring values cannot exceed $n + 1$. The distance between the locations of the neighboring values can be actually equal to $n + 1$: e.g., for values $(1, 1)$ and $(2, 2)$. Thus, for the standard memory arrangement, we have $C(f) = n + 1$.

A surprising result: the standard memory arrangement is optimal. Based on the fact that other memory arrangements of 2-D objects are often beneficial, one would expect these other memory arrangements be better than the standard one in the sense of our criterion $C(f)$. Surprisingly, this is not the case: it turns out that the standard memory arrangement is optimal.

To be more precise, we will prove that for every possible memory arrangement F, we have $C(f) \geqslant n+1$. Thus, the standard arrangement, for which $C(f) = n+1$, is indeed optimal.

Proof. Let us prove the inequality $C(f) \geqslant n + 1$. Let f be an arbitrary memory arrangement. This arrangement results in n^2 locations $f(i, j)$ corresponding to n^2 different pairs (i, j).

Let us denote the smallest of these n^2 values by $\underline{f}$, and the largest of these values by $\overline{f}$:

$$\underline{f} \stackrel{\text{def}}{=} \min\{f(i, j) : 1 \leqslant i, j \leqslant n\}, \tag{18}$$

$$\overline{f} \stackrel{\text{def}}{=} \max\{f(i, j) : 1 \leqslant i, j \leqslant n\}. \tag{19}$$

Between $\underline{f}$ and $\overline{f}$ (including both), there are n^2 different integers. For every $a < b$, the list $a, a+1, \ldots, b$ contains $b-a+1$ integers. Thus, we must have $\overline{f} - \underline{f} + 1 \geqslant n^2$, hence

$$\overline{f} - \underline{f} \geqslant n^2 - 1. \tag{20}$$

Let $(\underline{i}, \underline{j})$ denote the pair for which $f(\underline{i}, \underline{j}) = \underline{f}$, and let $(\overline{i}, \overline{j})$ denote the pair for which $f(\overline{i}, \overline{j}) = \overline{f}$. We can now design a sequence of pairs (i_k, j_k) going from $(i_0, j_0) = (\underline{i}, \underline{j})$ to $(i_N, j_N) = (\overline{i}, \overline{j})$ in such a way that for every k, the pairs (i_k, j_k) and (i_{k+1}, j_{k+1}) are neighbors.

Indeed, if $\underline{i} < \overline{i}$, we start with $i_0 = \underline{i}$, and then take $i_1 = i_0 + 1$, $i_2 = i_0 + 2$, etc., until we reach $\overline{i}$ — after this, we continue to take $i_k = \overline{i}$.

If $\underline{i} > \overline{i}$, we start with $i_0 = \underline{i}$, and then take $i_1 = i_0 - 1$, $i_2 = i_0 - 2$, etc., until we reach $\overline{i}$ — after this, we continue to take $i_k = \overline{i}$.

If $\underline{i} = \overline{i}$, then we simply take $i_k = \underline{i}$ for all k.

Similarly, if $\underline{j} < \overline{j}$, we start with $j_0 = \underline{j}$, and then take $j_1 = j_0 + 1$, $j_2 = j_0 + 2$, etc., until we reach $\overline{j}$ — after this, we continue to take $j_k = \overline{j}$.

If $\underline{j} > \overline{j}$, we start with $j_0 = \underline{j}$, and then take $j_1 = j_0 - 1$, $j_2 = j_0 - 2$, etc., until we reach $\overline{j}$ — after this, we continue to take $j_k = \overline{j}$.

If $\underline{j} = \overline{j}$, then we simply take $j_k = \underline{j}$ for all k.

At each step, each of the coordinates is changed by at most 1, so the pairs (i_k, j_k) and (i_{k+1}, j_{k+1}) are indeed neighbors.

We need $|\underline{i} - \overline{i}| + 1$ steps to reach from $\underline{i}$ to $\overline{i}$, and we need $|\underline{j} - \overline{j}|$ steps to reach from $\underline{j}$ to $\overline{j}$. Thus, overall, we need

$$N = \max(|\underline{i} - \overline{i}|, |\underline{j} - \overline{j}|) + 1 \tag{21}$$

steps. For values from 1 to n, the largest possible difference $|\underline{j} - \overline{j}|$ is equal to $n - 1$, hence $N \leqslant n$.

Now, we have

$$f(\underline{i}, \underline{j}) - f(\overline{i}, \overline{j}) = f(i_0, j_0) - f(i_N, j_N) =$$

$$= (f(i_0, j_0) - f(i_1, j_1)) + (f(i_1, j_1) - f(i_2, j_2)) + \ldots +$$

$$+ (f(i_{N-1}, j_{N-1}) - f(i_N, j_N)). \tag{22}$$

Thus,

$$|f(\underline{i}, \underline{j}) - f(\overline{i}, \overline{j})| \leqslant$$

$$\leqslant |f(i_0, j_0) - f(i_1, j_1)| + |f(i_1, j_1) - f(i_2, j_2)| + \ldots +$$

$$+ |f(i_{N-1}, j_{N-1}) - f(i_N, j_N)|. \tag{23}$$

Since for each k, the pairs (i_k, j_k) and (i_{k+1}, j_{k+1}) are neighbors, we have

$$|f((i_k, j_k) - f(i_{k+1}, j_{k+1})| \leqslant C(f).$$

So, from (23), we conclude that

$$|f(\underline{i}, \underline{j}) - f(\overline{i}, \overline{j})| \leqslant (N - 1) \cdot C(f). \tag{24}$$

Since $N \leqslant n$, we thus have

$$|\underline{f} - \overline{f}| = |f(\underline{i}, \underline{j}) - f(\overline{i}, \overline{j})| \leqslant (n - 1) \cdot C(f). \tag{25}$$

On the other hand, we know that $n^2 - 1 \leqslant |\underline{f} - \overline{f}|$. Thus, we conclude that

$$n^2 - 1 \leqslant (n - 1) \cdot C(f), \tag{26}$$

and therefore, that

$$C(f) \geqslant \frac{n^2-1}{n-1} = n+1. \tag{27}$$

The statement is proven.

The standard memory arrangement is not the only optimal one. The fact that the standard memory arrangement turned out to have the optimal (smallest possible) value of $C(f)$ may not sound so surprising if we realize that several different memory arrangements have the exact same optimal value of $C(f)$.

One such arrangement is clear: instead of storing the values row by row, we can store them column by column:

- first, we have elements of the first column,

$$f(1,1) = 1, f(2,1) = 2, \ldots, f(n,1) = n; \tag{28}$$

- then, we have elements of the second column,

$$f(1,2) = n+1, f(2,2) = n+2, \ldots, f(n,2) = 2n; \tag{29}$$

- ...
- the elements of the k-th column are store at

$$f(1,k) = (k-1)\cdot n + 1, f(2,k) = (k-1)\cdot n + 2, \ldots,$$
$$f(n,k) = (k-1)\cdot n + n = k\cdot n; \tag{30}$$

- ...
- finally, the elements of the last (n-th) column are stored at locations

$$f(1,n) = (n-1)\cdot n + 1, f(2,n) = (n-1)\cdot n + 2, \ldots,$$
$$f(n,n) = (n-1)\cdot n + n = n^2. \tag{31}$$

There are other examples as well: e.g., elements of a 2×2 matrix can be stored in the order $(1,1)$, $(1,2)$, $(2,2)$, $(2,1)$ with the same value $C(f) = n+1 = 3$ as for row-by-row or column-by-column memory arrangements.

Multi-dimensional case. In the k-dimensional case, we need to assign location $f(i_1,\ldots,i_k)$ to tuples $(i_1,\ldots,i_k)$. It is also natural to gauge the quality of the memory arrangement by the largest distance between the locations of neighboring values, i.e., tuples $(i_1,\ldots,i_k)$ and $(i'_1,\ldots,i'_k)$ for which $|i_j - i'_j| \leqslant 1$ for all j. The quality of a memory arrangement f can be thus naturally described by the value

$$C(f) \stackrel{\text{def}}{=} \max\{|f(i_1,\ldots,i_k) - f(i'_1,\ldots,i'_k)| :$$
$$|i_j - i'_j| \leqslant 1 \text{ for all } j = 1,\ldots,k\}. \tag{32}$$

In the standard computer arrangement, we store elements in lexicographic order: i.e., $(i_1,\ldots,i_k)$ is placed before $(i'_1,\ldots,i'_k)$ if for the first differing coordinate $i_j \neq i'_j$, we have $i_j < i'_j$. In other words, we first store values

$$(1,\ldots,1),\ldots,(1,\ldots,n), \tag{33}$$

then values

$$(1, \ldots, 2, 1), \ldots, (1, \ldots, 2, n), \tag{34}$$

etc. In this arrangement,

- the difference in the last coordinate $i_k - i'_k = 1$ leads to a difference of 1 in memory locations;
- the difference in the next to last coordinate $i_{k-1} - i'_{k-1} = 1$ leads to a difference of n in memory locations,
- $\ldots$,
- the difference in the first coordinate $i_1 - i'_1$ leads to a difference of n^{k-1} in memory locations.

Thus, the difference in location of neighboring tuples cannot exceed

$$n^{k-1} + n^{k-2} + \ldots + n + 1.$$

This distance is attained, e.g., for the points $(1, \ldots, 1)$ and $(2, \ldots, 2)$. Thus, for the standard memory arrangement f, we have

$$C(f) = n^{k-1} + n^{k-2} + \ldots + n + 1. \tag{35}$$

Similarly to the 2-D case, we can prove that this memory arrangement is optimal. Indeed, in this case, for the difference between the values

$$\underline{f} \stackrel{\text{def}}{=} \min\{f(i_1, \ldots, i_k) : 1 \leqslant i_j \leqslant n\}, \tag{36}$$

$$\overline{f} \stackrel{\text{def}}{=} \max\{f(i_1, \ldots, i_k) : 1 \leqslant i_j \leqslant n\}, \tag{37}$$

we have $\overline{f} - \underline{f} \geqslant n^k - 1$. We can still move from the tuple $(\underline{i}_1, \ldots, \underline{i}_k)$ at which the smallest value $\underline{f}$ is attained to the tuple $(\overline{i}_1, \ldots, \overline{i}_k)$ at which the largest value $\overline{f}$ is attained in $\leqslant n - 1$ transitions from a tuple to a neighboring one. Thus, we can conclude that

$$n^k - 1 \leqslant (n - 1) \cdot C(f), \tag{38}$$

and therefore, that

$$C(f) \geqslant \frac{n^k - 1}{n - 1} = n^{k-1} + n^{k-2} + \ldots + n + 1. \tag{39}$$

The optimality is proven.

Acknowledgment

This work was supported in part by the US National Science Foundation grant HRD-1242122 (Cyber-ShARE Center of Excellence).

The authors are thankful to Fred G. Gustavson and Lenore Mullin for her encouragement.

References

1. Berz M., Hoffstätter G. Computation and application of taylor polynomials with interval remainder bounds // Reliable Computing. 1998. No. 4(1). P. 83–97.
2. Bryant R.E., O'Hallaron D.R. Computer Systems: A Programmer's Perspective. Upper Saddle River : Prentice Hall, 2003.
3. Ceberio M., Ferson S., Kreinovich V., Chopra S., Xiang G., Murguia A., Santillan J. How to take into account dependence between the inputs: from interval computations to constraint-related set computations, with potential applications to nuclear safety, bio- and geosciences // Journal of Uncertain Systems. 2007. No. 1(1). P. 11–34.
4. Ceberio M., Kreinovich V., Pownuk A., Bede B. From interval computations to constraint-related set computations: towards faster estimation of statistics and odes under interval, p-box, and fuzzy uncertainty // Foundations of Fuzzy Logic and Soft Computing / P. Melin, O. Castillo, L.T. Aguilar, J. Kacprzyk, W. Pedrycz (eds.). Proceedings of the World Congress of the International Fuzzy Systems Association IFSA'2007. Cancun, Mexico, June 18–21, 2007. Springer Lecture Notes on Artificial Intelligence. 2007. No. 4529. P. 33–42.
5. Einstein A., Bergmann P. On the generalization of Kaluza's theory of electricity // Ann. Phys. 1938. No. 39. P. 683–701.
6. Feynman R., Leighton R., Sands M. The Feynman Lectures on Physics. Boston : Addison Wesley, 2005.
7. Green M.B., Schwarz J.H., Witten E. Superstring Theory. Vol. 1–2. Cambridge University Press, 1988.
8. Grover L.K. A fast quantum mechanical algorithm for database search // Proceedings of the 28th Annual ACM Symposium on the Theory of Computing. May 1996. P. 212-ff.
9. Grover L.K. From Schrödinger's equation to quantum search algorithm // American Journal of Physics. 2001. No. 69(7). P. 769–777.
10. Gustavson F.G. The relevance of new data structure approaches for dense linear algebra in the new multi-core/many core environments // Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics PPAM'2007. Gdansk, Poland, September 9–12, 2007. Springer Lecture Notes in Computer Science. 2008. No. 4967. P. 618–621.
11. Kaluza Th. Sitzungsberichte der K. Prussischen Akademie der Wiseenschaften zu Berlin. 1921. P. 966 (in German); Engl. translation: On the unification problem in physics [13, p. 1–9].
12. Klein O. Zeitschrift für Physik. 1926. Vol. 37. P. 895 (in German); Engl. translation: Quantum theory and five-dimensional relativity [13, p. 10–23].
13. Lee H.C. (ed.). An Introduction to Kaluza-Klein Theories. Singapore : World Scientific, 1984.
14. Neumaier A. Taylor forms // Reliable Computing. 2002. No. 9. P. 43–79.
15. Nielsen M., Chuang I. Quantum Computation and Quantum Information. Cambridge : Cambridge University Press, 2000.
16. Polchinski J. String Theory. V. 1–2. Cambridge University Press, 1998.
17. Revol N., Makino K., Berz M. Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY // J. Log. Algebr. Program. 2005. No. 64(1). P. 135–154.

18. Shor P. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer // Proceedings of the 35th Annual Symposium on Foundations of Computer Science. Santa Fe, NM, Nov. 20–22, 1994.
19. Shor P. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer // SIAM J. Sci. Statist. Comput. 1997. V. 26. P. 1484-ff.
20. Starks S.A., Kosheleva O., Kreinovich V. Kaluza-Klein 5D ideas made fully geometric // International Journal of Theoretical Physics. 2006. No. 45(3). P. 589–601.
21. Tietze H. Famous Problems of Mathematics: Solved and Unsolved Mathematical Problems, from Antiquity to Modern Times. New York : Graylock Press, 1965.
22. Zaniolo C., Ceri S., Faloutsos C., Snodgrass R.T., Subrahmanian V.S., Zicari R. Advanced Database Systems. Morgan Kaufmann, 1997.

КАК ХРАНИТЬ ТЕНЗОРЫ В ПАМЯТИ КОМПЬЮТЕРА: ОБЗОР

М. Себерио
к.ф.-м.н., доцент, e-mail: mceberio@utep.edu
В. Крейнович
к.ф.-м.н., профессор, e-mail: vladik@utep.edu

Техасский университет в Эль Пасо, США

Аннотация. В этой статье, объяснив необходимость использования тензоров в вычислениях, мы анализируем вопрос о том, как лучше хранить тензоры в памяти компьютера. Оказывается, что относительно естественного критерия оптимальности стандартный способ хранения тензоров оказывается одним из оптимальных.

Ключевые слова: Тензоры, вычислительная техника, память компьютера.

Дата поступления в редакцию: 02.01.2018