Computing ψ-Caputo Fractional Derivative Values Using CUDA 10

Vsevolod Bohaienko[0000-0002-3317-9022]

V.M. Glushkov Institute of cybernetics of NAS of Ukraine, Kyiv, Ukraine
sevab@ukr.net

Abstract. The paper addresses the issues of efficient GPU-implementation of ψ-Caputo fractional derivative values computation on NVIDIA GPU's with compute capability 7.5 using CUDA 10 SDK on both CUDA and OpenCL languages. We consider a three-dimensional time-fractional diffusion equation solved by a locally one-dimensional finite difference scheme. To compute non-local part of the derivative a rectangle rule quadrature is used and a summation algorithm of linear computational complexity is considered along with a constant complexity order approximating algorithm based on integral kernel expansion into series. For the approximating algorithm we present a computational scheme that uses NVidia GPU's tensor cores. For both algorithms, we study the influence of the used scalar and vector data types on performance and accuracy. Studying the summation algorithm, comparing to the usage of 64-bit double-precision floating-point data type, the computations were ~2 times faster for 32-bit single-precision data type and ~3 times faster for 16-bit half-precision data type without significant loss of accuracy. For the approximated algorithm that was up to 5-times faster than the summation algorithm, the usage of low-precision data types slightly influence the performance reducing the accuracy during long-term simulations. The usage of vectorized operations in the approximation algorithm allowed up to 6-19% speed-up compared with non-vectorized implementations for a single-precision data type. The usage of tensor cores that operate with a half-precision data type allowed performing calculations 12% faster compared to the case when the same data type was used.

Keywords: GPU algorithms, finite-difference method, diffusion equation, ψ-Caputo fractional derivative, tensor cores, data types, CUDA, OpenCL.

1 Introduction

Memory effects in diffusion processes can be efficiently simulated using time-fractional differential equations [1-3].

Such equations contain the so-called fractional derivatives that are integral-differential operators.

The need to numerically calculate integrals while solving time-fractional differential equations increase the computational complexity order compared to the traditional differential equations.

Approaches to lower computational complexity include parallel computing techniques [4,5,6], particularly using GPUs [5,6], and approximation of integral kernels [7,8,9].

In this paper we consider a three-dimensional diffusion equation with the generalized ψ-Caputo derivative with functional parameter solved by a locally one-dimensional finite-difference scheme similarly to [10].

We focus on efficient GPU implementation of ψ-Caputo derivative's values computation, which is one the most time-consuming operation while performing simulations, on NVidia GPUs using CUDA 10 SDK.

We study GPU algorithms' performance when scalar and vector data types of different length and precision are used in CUDA and OpenCL algorithms' implementations along with the possibility to speed up computations making use of 16x16 matrix-matrix multiplication operations that are hardware-implemented in the so-called tensor cores of the NVidia GPUs with compute capabilities 7.x.

2 Problem statement and finite-difference scheme

We consider the following three-dimensional time-fractional diffusion equation [10]:

$$\sigma D_{t,g}^{(\beta)}C(x,y,z,t) = D\left(\frac{\partial^2 C(x,y,z,t)}{\partial x^2}+\frac{\partial^2 C(x,y,z,t)}{\partial y^2}+\frac{\partial^2 C(x,y,z,t)}{\partial z^2}\right)+ \\ + F(x,y,z,t), 0 \le x \le 1, 0 \le y \le 1, 0 \le z \le 1, t > 0, 0 < \beta \le 1 \quad (1)$$

where $C(x,y,z,t)$ is the diffusive substance concentration, σ is the porosity, D is the diffusion coefficient, F is the given source-term function, and $D_{t,g}^{(\beta)}$ is the generalized ψ-Caputo fractional derivative with respect to the time variable t with the functional parameter $g(t)$ that has the following form [11]:

$$D_{t,g}^{(\beta)}C(x,y,z,t) = \frac{1}{\Gamma(1-\beta)}\int_0^t \frac{\partial C(x,y,z,t)}{\partial t}(g(t)-g(\tau))^{-\beta}d\tau, \quad (2)$$

$$g(t) \in C^1[0,+\infty), \; g'(t) > 0 \; (t \ge 0), \; g(0) = 0 .$$

For the equation (1) we pose first-type initial and boundary conditions. The initial boundary value problem for the equation (1) is discretized using locally one-dimensional finite-difference scheme [10,12] on the uniform grid

$$\omega = \{(x_i, y_j, z_k, t_l): x_i = ih_1 \; (i=\overline{0,n_1+1}), \; y_j = jh_2 \; (j=\overline{0,n_2+1}), z_k = kh_3 \; (j=\overline{0,n_3+1}), \; t_l = l\tau$$

where h_1, h_2, h_3, τ are the steps with respect to the space and the time variables.

The finite-difference scheme, similarly to [10,12], is as follows:

$$C_{ijk}^{(l+1/3)} - \frac{\tau D}{k_1 h_1^2}(C_{i-1,j,k}^{(l+1/3)} - 2C_{ijk}^{(l+1/3)} + C_{i+1,j,k}^{(l+1/3)}) = C_{ijk}^{(l)} + \frac{\tau}{3k_1} F_{ijk}^{(l)}, \tag{3}$$

$$C_{ijk}^{(l+2/3)} - \frac{\tau D}{k_1 h_2^2}(C_{i,j-1,k}^{(l+2/3)} - 2C_{ijk}^{(l+2/3)} + C_{i,j+1,k}^{(l+2/3)}) = C_{ijk}^{(l+1/3)} + \frac{\tau}{3k_1} F_{ijk}^{(l)}, \tag{4}$$

$$C_{ijk}^{(l+1)} - \frac{\tau D}{k_1 h_3^2}(C_{i,j,k-1}^{(l+1)} - 2C_{ijk}^{(l+1)} + C_{i,j,k+1}^{(l+1)}) = C_{ijk}^{(l+2/3)} + \frac{\tau}{3k_1} F_{ijk}^{(l)}, \tag{5}$$

$$k_1 = \frac{b_l^{(l+1)}}{\Gamma(1-\beta)},$$

$$F_{ijk}^{(l)} = F_{ijk} + \frac{1}{\Gamma(1-\beta)} \sum_{s=0}^{l-2} b_s^{(l)} \frac{C_{ijk}^{(s+1)} - C_{ijk}^{(s)}}{\tau}, \tag{6}$$

$$b_s^{(i)} = \int_{t_s}^{t_{s+1}} (g(t_i) - g(\tau))^{-\beta} d\tau. \tag{7}$$

The equations (3)-(5) can be with the addition of discretized forms of boundary conditions represented as three-diagonal linear equation systems, which are solved by the Thomas algorithm [12].

The summation in the right-hand side of (6) is the first-order discretization of a non-local part of the fractional derivative obtained applying the rectangle quadrature to the integral in (2).

3 Approximating algorithm for computing ψ-Caputo fractional derivative values

Computation of non-local part of fractional derivative according to (6) has $O(l)$ computational complexity and requires storing all previously obtained solutions making it time and memory consuming while performing simulations over large time intervals. Hence, we consider the constant complexity order algorithm based on series expansion of the integrals (7) [13, 14] that requires storing solutions on only the two last time steps.

When $g(t)$ has an infinitely differentiated inverse function $f(t)$, $b_s^{(i)}$ can be represented in the form [13,14]

$$b_s^{(j)} = \sum_{n=0}^{\infty}\left((-1)^n \binom{-\beta}{n} g(t_j)^{-\beta-n} S_n\right),$$
$$S_n(t_s, t_{s+1}) = \sum_{m=0}^{\infty}\left[B_m \frac{f^{(m+1)}(g(t_{s+1}))}{m!}\right],$$

$$B_m = \int_{g(t_s)}^{g(t_{s+1})} x^n (x - g(t_{s+1}))^m dx,\, B_0 = \int_{g(t_s)}^{g(t_{s+1})} x^n dx = \frac{1}{n+1}(g(t_{s+1})^{n+1} - g(t_s)^{n+1}),$$
$$B_{i+1} = -\frac{n+i+2}{g(t_{s+1})(i+1)}\left(B_i - \frac{g(t_s)^{n+1}(g(t_s) - g(t_{s+1}))^{i+1}}{g(t_{s+1})(i+1)}\right),$$

Truncating the infinite series, the summation in (6) using such representation can be organized [14] as follows:

$$\sum_{s=0}^{l-2} b_s^{(l)} \frac{C_{ijk}^{(s+1)} - C_{ijk}^{(s)}}{\tau} \approx \frac{1}{\tau}\sum_{n=0}^{K}\left((-1)^n \binom{-\beta}{n} g(t_l)^{-\beta-n} S_{n,l-1}\right),$$
$$S_{n,l} = S_{n,l-1} + (C^{(l)} - C^{(l-1)}) S_n(t_{l-1}, t_l),\, S_{n,0} = 0. \tag{8}$$

where K is the given number of terms in series.

So, to compute in a specific node (i, j, k) the value of the term $F_{ijk}^{(l)}$, which is the most time-consuming part when computing right-hand sides of three-diagonal linear equations systems (3)-(5), we need to store the values of $S_{n,\bullet}$ and on the time step l

- update $S_{n,l-1}$ into $S_{n,l}$ according to the second equation in (8);
- calculate the value of the sum according to the first equation in (8);
- substitute the obtained value of the sum into (6) getting the value of $F_{ijk}^{(l)}$.

4 Parallel implementation

We perform the solution of linear systems (3)-(5) using multithreaded OpenMP implementation with the calculation of a non-local part of the ψ-Caputo fractional derivative upon (6) or (8) performed for every node of the grid using GPU.

The calculations upon the series approximating algorithm (8) are organized following the scheme described in [15]:

- the values of $S_n(t_{l-1}, t_l)$ are calculated on CPU on the step $l-1$ in parallel with GPU computations upon (8) and uploaded on the step l into GPU memory along with the solution $C^{(l-1)}$;

- each GPU thread calculates the value of $F_{ijk}^{(l)}$ for a specific node (i, j, k) and, after GPU performed the computations, values of $F_{ijk}^{(l)}$ are copied back into CPU memory.

To use GPUs local memory [15], we form thread groups of a controllable size χ with computations organized in $\lceil l/\chi \rceil$ substeps. On the substep s, the values of $S_n(t_{l-1}, t_l), s\lceil l/\chi \rceil \le n < (s+1)\lceil l/\chi \rceil$ are loaded into local memory and a corresponding part of $S_{n,l}$ is updated by each thread. Further, the resulting K -terms sum in (8) is computed with $(-1)^n \begin{pmatrix} -\alpha \\ n \end{pmatrix} g(t_{l-2})^{-\alpha-n}$ in advance calculated in parallel and loaded into local memory.

To make use of vector data types present in CUDA and OpenCL languages, the summations in (8) are split into m-terms vector operations obtaining

$$\sum_{s=0}^{l-2} b_s^{(l)} \frac{C_{ijk}^{(s+1)} - C_{ijk}^{(s)}}{\tau} \approx \frac{1}{\tau} \sum_{n=0}^{\lceil K/m \rceil} \left(\vec{v}_{n,m}^{(1)} \left(S_{n\cdot m, l-1}, \ldots, S_{(n+1)\cdot m, l-1} \right) \right),$$

$$\vec{v}_{n,m}^{(1)} = \left((-1)^{n\cdot m} \begin{pmatrix} -\beta \\ n\cdot m \end{pmatrix} g(t_l)^{-\beta-n\cdot m}, \ldots, (-1)^{(n+1)\cdot m} \begin{pmatrix} -\beta \\ (n+1)\cdot m \end{pmatrix} g(t_l)^{-\beta-(n+1)\cdot m} \right)$$

$$\begin{gathered} \left(S_{n\cdot m, l}, \ldots, S_{(n+1)\cdot m, l} \right) = \left(S_{n\cdot m, l-1}, \ldots, S_{(n+1)\cdot m, l-1} \right) + \\ +(C^{(l)} - C^{(l-1)}) \left(S_{n\cdot m}(t_{l-1}, t_l), \ldots, S_{(n+1)\cdot m}(t_{l-1}, t_l) \right). \end{gathered} \tag{9}$$

The performance of the algorithm (9) was studied for 16-component double-precision vectors in [15].

Continuing this work, we consider the calculation upon (9) and the saving of $S_{n,l}$ using floating-point data types of different precision (16, 32, and 64 bits) and different vector sizes (2, 4, 8, 16).

In all cases, we save the values of C and S_n in a double-precision floating-point format in both CPU and GPU memories. The summation upon (6) can also be represented in vectorized form, and we consider it with the same data types as the algorithm (9).

As the newest NVidia GPUs implement 16x16 matrix-matrix (tensor) multiplication operations in hardware in the so-called tensor cores, we transform the summation in (8) to make use of them.

The considered tensor operation $T_{ma}(A, B, C)$ multiplies two 16x16 half-precision matrices A, B and adds a 16x16 single-precision matrix C to the result of multiplication. The result of this operation also has a single-precision data type. Considering the vector of sums $S^{(16)}$ for 16 grid cells $C_{(1)}, \ldots, C_{(16)}$, the summation part of (9) for $K = 16$ can be performed as follows:

$$S^{(16)} \approx \frac{1}{\tau}(T_{ma}(S,V,A_0))_1,\ V = (\vec{v}_{n,16}^{(1)\ T}, v_0^T, ..., v_0^T),\ S = \begin{pmatrix} S_{1,l-1,(0)} & ... & S_{16,l-1,(0)} \\ ... & ... & ... \\ S_{1,l-1,(16)} & ... & S_{16,l-1,(16)} \end{pmatrix} \quad (10)$$

where A_0 is the 16x16 zero matrix, v_0 is the 16-component zero vector, $S_{n,l,(i)}$ is the $S_{n,l}$ coefficient value for the grid cell i, and $(\cdot)_1$ denotes the first column of the matrix.

The algorithm (10) makes use only of one matrix-vector multiplication of 16 performed in T_{ma} operation.

5 Numerical experiments

We consider the test problem for the equation (1) with $g(t) = t^{\gamma}$ and

$$F(x,y,z,t) = \frac{\Gamma(1+2/\gamma)}{\Gamma(1-\beta+2/\gamma)} t^{2-\beta\gamma} - 2D(x^2y^2 + y^2z^2 + x^2z^2).$$

Then, the solution of (1) has the form $C(x,y,z,t) = C_0(x,y,z,t) = x^2y^2z^2 + t^2$ for the case of $\sigma = 1$ and the first-type initial and boundary conditions derived from $C(x,y,z,t)$.

The system (3)-(5) was solved on 200 time steps of size $\tau = 0.0025$, $\gamma = 0.8, \beta = 0.8$, grid size of $N \times N \times N, N = 40,...,80$, and $K = 16,...,64$ on a single node of SCIT-4 cluster of VM Glushkov Institute of Cybernetics of NAS of Ukraine. The used node contains two Intel(R) Xeon(R) Bronze 3104 CPUs with total of 12 cores and NVidia RTX 2080 Ti GPU.

The algorithms were implemented in both CUDA and OpenCL languages and the corresponding source code can be accessed through https://github.com/sevaboh/FPDE_HPC.

5.1. Performance for a fixed N

The available set of data types for CUDA implementation was *double*, *float*, *half*, *double[2,4]*, *float[2,4]*, and *half2*.

In the case of OpenCL we considered the implemented in CUDA10 *double*, *float*, *double[2,4,8,16]*, and *float[2,4,8,16]* data types. The times spent on computation upon (6), (9), and (10) on 200 iterations for $N = 40, K = 16$ and all the considered combinations of languages and data types are given in Table.1. In all experiments thread block size χ was equal to K.

The times spent on a single iteration are near to equal for the approximating algorithm (9) and tend to increase linearly for the summation upon (6). This linear increase is non-monotone (Fig.1) due to incomplete GPU resources usage when $t\%K \neq 0$ where t is the time step number.

Doing calculations upon (6) in CUDA implementation, comparing to the usage of a double-precision data type, the algorithms that were implemented using data types of lower precision performed the computations ~2 times faster for 32-bit single-precision data type and ~3 times faster for 16-bit half-precision data type.

The usage of the approximating algorithm (9) up to ~5 times decreased the computation time compared to the summation upon (6).

Here the usage of a single-precision data type decreased the time only by <12% when the algorithm was implemented in CUDA.

This can be explained by a high amount of data type conversion operations in the implementations of (9) due to the storage of the solutions C and the coefficients S_n using 64-bit double-precision data type.

The usage of a half-precision data type in CUDA implementation slowed the computations compared to the usage of a single-precision data type.

However, the half-precision implementation that uses tensor cores was the fastest of all the considered variants. The usage of a single-precision data type in OpenCL implementation slowed down the computations compared to the usage of a double-precision data type.

Table 1. Times spent on computation upon (6), (9), and (10) for $N = 40, K = 16$

Data type	CUDA, summation upon (6)	CUDA, approximating algorithm (9)	OpenCL, summation upon (6)	OpenCL, approximating algorithm (9)
double	344,18	78,37	286,86	81,14
float	204,03	74,86	88,27	83,97
half	105,77	77,32		
double2	350,24	81,41	287,11	81,90
float2	191,38	73,02	91,08	84,97
half2	98,14	74,70		
double4	345,18	79,52	287,57	81,56
float4	182,00	70,19	90,22	84,87
double8			292,92	80,81
float8			90,50	85,18
double16			292,41	80,99
float16			90,76	84,38
Usage of tensor cores		69,49		

For $K = 16$, vectorization accelerated computations only for the case of a single-precision floating-point data type in CUDA implementation.

To explain the observed behavior, we compare PTX assembly sources generated by CUDA and OpenCL compilers with the generation of FMA (floating-point multiply-add) instruction enabled.

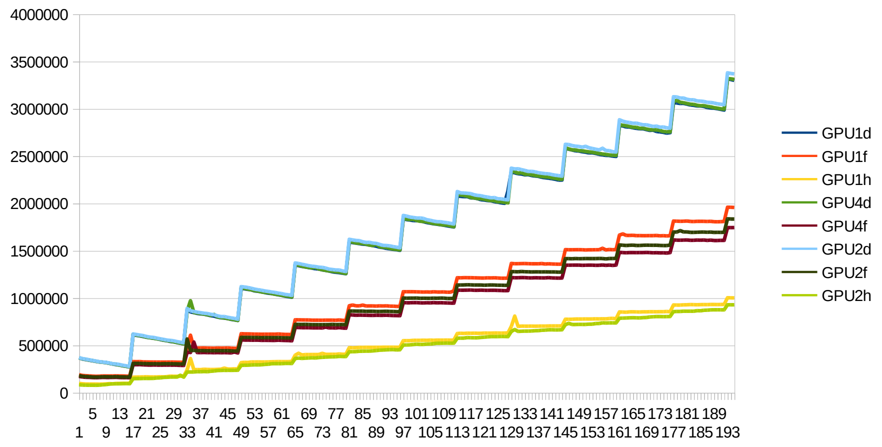

Fig. 1. CUDA kernel execution time, *ns*, for the total summation upon (6), $N = 40, K = 16$

The following conclusions can be made from the performed comparison:

- both compilers perform automatic unwinding of 16 loop iterations;
- both compilers do not generate vector arithmetic instructions when vector data types are used;
- both compilers generate vectorized 128-bit load/store instructions *[ld,st].[shared,global].v2.f64* and *[ld,st].[shared,global].v4.f32* and this is their usage that accelerated computations when vector data types are used;
- when both compilers perform loop unwinding, memory is addressed through fixed offsets from the base for both shared and global memory (e.g. *ld.global.f64 %fd77, [%rd23+40]*). For the vectorized data types, offsets to global memory are computed for every memory access to the corresponding vector (e.g. *mul.wide.s32 %rd28, %r67, 128; add.s64 %rd29, %rd3, %rd28; ld.global.v2.f64 {%fd172, %fd173}, [%rd29+112]; ld.global.v2.f64 {%fd176, %fd177}, [%rd29+96];...*). For both algorithms (6) and (8), such additional operations slowed down the computations. However, the increase of K leads to the decrease of the number of auxiliary operations allowing obtaining up to 15% speed-up for *double16* data type usage in OpenCL;
- CUDA compiler automatically vectorizes the reading of 32-bit floating-point data stored in the local memory using 64-bit instruction *ld.shared.v2.f32* making the calculation upon (9) faster than in the case of double-precision data type usage. The absence of such automatic vectorization in OpenCL makes the single-precision OpenCL implementation to perform slower than in the case of a double-precision data type. The same applies to half-precision data type usage in CUDA that yields slower code than for the case of a single-precision data type;

- While performing calculation upon (6), CUDA compiler implements division operation in *div.rn* instruction, while OpenCL compiler uses approximated *div.full* instruction making OpenCL code faster.

To check how the tendencies of vectorization efficiency observed for $K = 16$ changes for its bigger values, the series of computational experiments were conducted for $N = 40$ and $K = 16,...,64$ for the case of the approximating algorithm (9).

Here, automatic loop unwinding leads to the decrease of the number of auxiliary operations with the increase of K. The computation time also decreases by up to 15% for *double16* data type usage in OpenCL, up to 5.3% for *double8* data type usage in OpenCL, up to 2.6% for *float16* data type usage in OpenCL, and up to 19% for *float4* data type usage in CUDA. The usage of *double[2,4]* and *float[2,8]* data types in OpenCL slowed down the computations.

Thus, while OpenCL language supports larger vector data types than CUDA language (16 elements compared to 4 elements), its inefficient compilation by NVidia compiler leads to the situation when OpenCL vectorized implementations of the algorithm (9) allow accelerating computations only for large K and the largest vector data types.

5.2 Performance in the case of variable grid size

The influence of the time of fractional derivative values calculation on the total time spent on simulation along with the speed-ups of different GPU implementations are given in Table 2 for the summation upon (6) and in Table 3 for the approximating algorithm (9) for $N = 40, \dots, 80$.

Table 2. Algorithms' relative characteristics for the summation upon (6), $K = 16$, 200 iterations

	N	40	60	80
CPU	Percentage of summation to total time	31,93%	36,64%	36,41%
CUDA *float* to CPU	Total time	40,74%	43,58%	51,15%
	Summation time	939,46%	1374,70%	1568,14%
CUDA *float*	Percentage of summation to total time	4,32%	3,57%	3,30%
	Percentage of auxiliary host-GPU operations	28,62%	22,25%	22,22%
CUDA *float4* to CUDA *float*	Summation time	7,52%	5,40%	8,80%
OpenCL *float* to CUDA *float*	Summation time	68,19%	104,21%	95,05%

In the algorithm (6), time spent on fractional derivative values computation (summation time) comprises up to 36% of total computation time on CPU. This value lowers to ~4% on GPU when this operation became 9-30 times faster and the total computation time became ~50% less. As can be seen from the Table 2, GPU algorithms' efficiency increases with the increase of N while the efficiency of vectorization remains on the same level. For the approximating algorithm (9) and $K = 16$, fractional derivative values computation comprises up to 40% of the total computation time while executing on CPU with slight decrease of this percentage with the increase of N. GPUs efficiency here is close to the case of the summation upon (6): 15-27 times acceleration of the calculations upon the algorithm (9), 51-63% decrease of the total computation time compared to CPU, ~3% of time spent by GPU algorithms for fractional derivative values computation. Comparing to the slower algorithm (6), the time spent on auxiliary CPU-GPU memory copying operations while calling GPU kernels is here bigger: ~60% compared to ~25%. Efficiency of vectorization do not depend on N with the usage of tensor cores yielding the highest 12% increase compared with the usage of a half-precision data type and 8% increase compared to the usage of a single-precision data type. Efficiency of the usage of tensor operations compared to the fastest case of vectorized *float4* implementation however is only 2%.

Table 3. Algorithms' relative characteristics for the approximating algorithm (9), $K = 16$, average on 200 iterations

	N	40	60	80
CPU	Percentage of summation to total time	39,85%	38,04%	36,85%
CUDA *float* to CPU	Total time	61,19%	53,05%	55,29%
	Summation time	1717,88%	2536,28%	2735,08%
CUDA *float*	Percentage of summation to total time	3,53%	2,21%	2,02%
	Percentage of auxiliary host-GPU operations	67,30%	61,63%	59,96%
CUDA *float4* to CUDA *float*	Kernel execution time	6,43%	6,40%	6,60%
OpenCL *float* to CUDA *float*	Kernel execution time	-10,84%	-9,83%	-9,72%
CUDA *half* to CUDA *float*	Kernel execution time	-3,27%	-3,31%	-3,44%
CUDA *half2* to CUDA *float*	Kernel execution time	0,21%	0,68%	0,69%
CUDA *half*, tensor cores, to CUDA *float*	Kernel execution time	7,53%	8,07%	8,24%
CUDA *half*, tensor cores to CUDA *half*	Kernel execution time	11,16%	11,77%	12,09%

5.3 Errors of solution

Changes in maximal square absolute error on the iterations while using summation upon (6) and series expansion upon (9) for different data types are given in Fig. 2. Lowering of data type precision has a little influence on the quality of solution for the summation upon (6) due to monotone increase of $b_s^{(i)}$ when $i \to s$ that minimizes summation errors. The error for the algorithm (6) decreases with the increase of time step number.

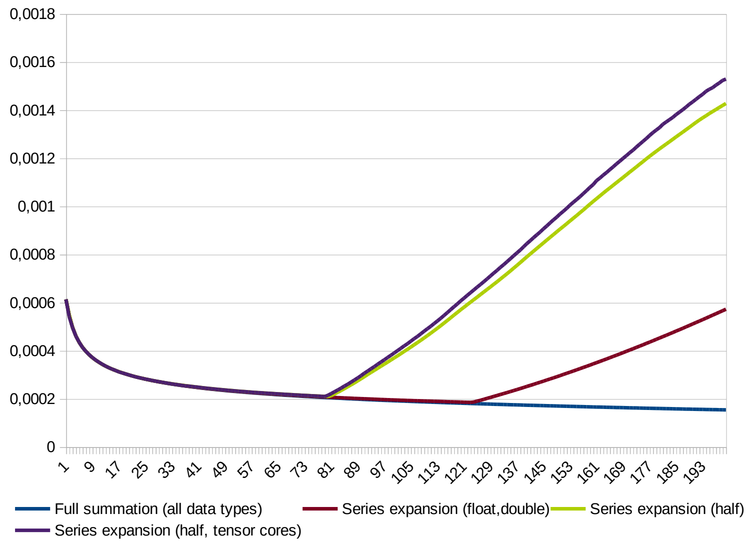

Fig. 2. Maximal square absolute error on time steps while solving the problem using different data types

Close values of errors between the summation upon (6) and the approximating algorithm (9) with $K = 16$ on the initial time steps show the adequacy of a chosen value of K. However, for the algorithm (9), the recurrently changing values of $S_{n,l}$ overflow the data type values range starting from some time step leading to linear increase of error. In the conducted experiments such behavior was observed for a half- and single-precision data types.

6 Conclusions

Calculation of time-fractional ψ-Caputo derivatives' values significantly influence time needed to solve fractional derivative equations that contain them. For the considered case of the diffusion equation solved using a finite-difference scheme, this operation comprises up to 40% of time spent while performing calculation on CPU. Its parallelization on GPU is efficient because the computations on the cells of the grid are independent.

We studied additional possibilities to speedup GPU implementations considering the summation algorithm of linear computational complexity and the approximating recurrent algorithm of constant complexity. Considering the usage of different precision floating-point data types, their vectors, and matrix-matrix multiplication operations implemented in tensor cores of the latest NVidia GPUs the following conclusion can be made:

- for the summation algorithm (6), comparing to the usage of a double-precision data type, the computations were ~2 times faster using 32-bit single-precision data type and ~3 times faster using 16-bit half-precision data type without significant loss of accuracy;
- for the approximating algorithm (9) that was up to 5-times faster than the summation algorithm, the usage of different data types slightly influence the performance due to the high amount of data type conversion operations. However, the usage of low-precision data types here leads to linear increase of solution error starting from some time step due to data type overflow;
- Inefficient compilation of vectorized memory access in OpenCL code leads to the situation when OpenCL vectorized implementations of the algorithm (9) allow accelerating computations only for large number K of terms in truncated series and the largest vector data types. Thus, the usage of CUDA is preferable for the approximating algorithm (9). However, for the summation algorithm (6), OpenCL implementations were faster due to the differences in compiling the division arithmetic operation;
- GPU implementations 9-30 times accelerate fractional derivative values computation with the efficiency that increases with the increase of grid size;
- the usage of tensor cores that operate with 16-bit half-precision floating-point data type allowed performing calculations 12% faster than the usage of a half-precision data type without vectorization. However, comparing to the efficiency of the fastest vectorized implementation that uses *float4* data type, only 2% acceleration was obtained.

References

1. Gorenflo, R., Mainardi, F.: Fractional calculus: integral and differential equations of fractional order. In: Fractals and Fractional Calculus in Continuum Mechanics, pp.223–276, Springer Verlag, Wien, Austria (1997)
2. Podlubny, I.: Fractional differential equations, Academic Press, New York (1999)
3. Bulavatsky, V.M.: Mathematical modeling of dynamics of the process of filtration convection diffusion under the condition of time nonlocality. Journal of Automation and Information Science **44(4)**, 13–22 (2012). doi: 10.1615/JAutomatInfScien.v44.i4.20
4. Bonchis, C., Kaslik, E., Rosu, F.: HPC optimal parallel communication algorithm for the simulation of fractional-order systems. J Supercomput **75**, 1014–1025 (2019). doi: 10.1007/s11227-018-2267-z

5. Liu, J., Gong, C., Bao, W., Tang, G., Jiang, Y.: Solving the Caputo Fractional Reaction-Diffusion Equation on GPU. Discrete Dynamics in Nature and Society **2014**, 820162 (2014). doi: 10.1155/2014/820162
6. Golev, A., Penev, A., Stefanova, K., Hristova, S.: Using GPU to speed up calculation of some approximate methods for fractional differential equations. International Journal of Pure and Applied Mathematics **119(3)**, 391-401 (2018). doi: 10.12732/ijpam.v119i3.1
7. Gong, C., Bao, W., Liu, J.: A piecewise memory principle for fractional derivatives. Fract. Calc. Appl. Anal. **20(4)**, 1010–1022 (2017). doi: 10.1515/fca-2017-0052
8. Ford, N.J., Simpson, A.C.: The numerical solution of fractional differential equations: Speed versus accuracy. Numerical Algorithms, **26(4)**, 333–346 (2001). doi: 10.1023/A:1016601312158
9. Baffet, D., Hesthaven, J.S.: A kernel compression scheme for fractional differential equations. Siam J. Numer. Anal **55(2)**, 496–520 (2017). doi: 10.1137/15M1043960
10. Bogaenko, V.A., Bulavatsky V.M.: Computer modeling of the dynamics of migration processes of soluble substances in the case of groundwater filtration with free surface on the base of the fractional derivative approach (in Russian). Dopov. Nac. Akad. Nauk Ukr. **12**, 21–29 (2018). doi: 10.15407/dopovidi2018.12.021
11. Almeida, R.: A Caputo fractional derivative of a function with respect to another function. Communications in Nonlinear Science and Numerical Simulation **44**, 460–481 (2017). doi: 10.1016/j.cnsns.2016.09.006
12. Samarskii A.: The Theory of Difference Schemes. CRC Press, New York (2001).
13. Bohaienko, V.O.: A fast finite-difference algorithm for solving space-fractional filtration equation with a generalised Caputo derivative. Computational and Applied Mathematics **38(3)**, 105 (2019). doi: 10.1007/s40314-019-0878-5
14. Bohaienko V.O.: Efficient computation schemes for generalized two-dimensional time-fractional diffusion equation. In: Papers of the International scientific and practical conference ITCM – 2019, Ivano-Frankivsk, Ukraine, pp. 238–241 (2019).
15. Bohaienko, V.O.: Performance of vectorized GPU-algorithm for computing ψ-Caputo derivative values. In: Hu Z., Petoukhov S., Dychka I., He M. (eds) Advances in Computer Science for Engineering and Education III. ICCSEEA 2020. Advances in Intelligent Systems and Computing, vol 1247. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-55506-1_24