Parallel Computation of Normalized Legendre Polynomials Using Graphics Processors

Konstantin Isupov(✉), Vladimir Knyazkov, Alexander Kuvaev, and Mikhail Popov

Department of Electronic Computing Machines,
Vyatka State University, Kirov 610000, Russia
{ks_isupov,knyazkov}@vyatsu.ru

Abstract. To carry out some calculations in physics and Earth sciences, for example, to determine spherical harmonics in geodesy or angular momentum in quantum mechanics, it is necessary to compute normalized Legendre polynomials. We consider the solution to this problem on modern graphics processing units, whose massively parallel architectures allow to perform calculations for many arguments, orders and degrees of polynomials simultaneously. For higher degrees of a polynomial, computations are characterized by a considerable spread in numerical values and lead to overflow and/or underflow problems. In order to avoid such problems, support for extended-range arithmetic has been implemented.

Keywords: Normalized Legendre polynomials · Extended-range arithmetic · GPU · CUDA

1 Introduction

Associated Legendre polynomials are solutions of the differential equation

$$(1-x^2)\frac{d^2}{dx^2}P_n^m(x) - 2x\frac{d}{dx}P_n^m(x) + \left[n(n+1) - \frac{m^2}{1-x^2}\right]P_n^m(x) = 0, \qquad (1)$$

where degree n and order m are integers satisfying $0 \le n$, $0 \le m \le n$, and x is a real variable in the interval $[-1, 1]$ which is usually expressed as $\cos\theta$, where θ represents the colatitude [1, Chap. 15].

These polynomials are important when defining geopotential of the Earth's surface [2,3], spherical functions in molecular dynamics [4], angular momentum in quantum mechanics [5], as well as in a number of other physical applications. The accuracy and scale of numerical simulations directly depend on the maximum degree of a polynomial which can be correctly computed. Modern applications typically operate upon first-kind ($m = 0$) polynomials at a degree of 10^3 or higher. The functions $P_n^m(x)$ grow combinatorially with n and can overflow for n larger than about 150. Therefore, for large n, instead of $P_n^m(x)$, *normalized* associated Legendre polynomials are computed. There are a number

V. Voevodin and S. Sobolev (Eds.): RuSCDays 2016, CCIS 687, pp. 172–184, 2016.
DOI: 10.1007/978-3-319-55669-7_14

of different normalization methods [6, Chap. 7]. We consider the computation of fully normalized polynomials

$$\bar{P}_n^m(x) = \sqrt{\frac{2n+1}{2}\frac{(n-m)!}{(n+m)!}}(1-x^2)^{m/2}\frac{\partial^m}{\partial x^m}P_n(x), \tag{2}$$

which satisfy the following equation:

$$\int_{-1}^{1}\{\bar{P}_n^m(x)\}^2 dx = 1. \tag{3}$$

Mathematical properties and numerical tables of $\bar{P}_n^m(x)$ are given in [7]. A number of recursive algorithms are suggested to evaluate $\bar{P}_n^m(x)$ [8–10]. One of the most common ones is based on the following relation [8]:

$$\bar{P}_n^{m-1}(x) = \frac{2mx}{\sqrt{(1-x^2)(n+m)(n-m+1)}}\bar{P}_n^m(x) - \sqrt{\frac{(n-m)(n+m+1)}{(n+m)(n-m+1)}}\bar{P}_n^{m+1}(x). \tag{4}$$

The starting points for recursion (4) are the values $\bar{P}_n^{n+1}(x) = 0$ and

$$\bar{P}_n^n(x) = \sqrt{\frac{1}{2}\frac{3\cdot 5\cdots(2n+1)}{2\cdot 4\cdots 2n}}(1-x^2)^{n/2}. \tag{5}$$

Equations (4) and (5) are asymptotically stable at any admissible parameters x, m and n, so if we consider them in terms of pure mathematics, they are appropriate for computing polynomials of an arbitrarily high degree. In practical computation, however, there are difficulties in computing (4) and (5) when n becomes large [3]. This is due to the following reasons:

- computations take an unacceptable long time;
- overflow or underflow exceptions may occur.

The first of these problems stems from the fact that during the numerical simulation it is required to compute many polynomials of different degrees at a fixed angle, or many fixed degree polynomials for a variety of angles. An effective solution to this problem has been made possible thanks to the intensive development of new generation massively parallel computing architectures, such as graphics processing units (GPUs).

The second problem is related to the limited dynamic range of real numbers which are represented in computers [11]. As a result, if x is about ± 1, the computation of the starting value $\bar{P}_n^n(x)$ leads to underflow, even though the desired value $\bar{P}_n^m(x)$ is within an acceptable dynamic range. For example, if $x = 0.984808$, which corresponds to angle $\theta \approx 10°$, then $\bar{P}_{5000}^{5000}(x) \approx 1.42 \times 10^{-3801}$ while $\bar{P}_{5000}^{0}(x) \approx 3.32 \times 10^{-1}$. The smallest normal value in IEEE-754 double-precision format is approximately equal to 10^{-308}. Thus, to evaluate $\bar{P}_{5000}^{5000}(x)$, it is necessary to extend the dynamic range by more than an order of magnitude. The value of $\bar{P}_{5000}^{5000}(x)$ is not of independent practical interest, however,

it is impossible to start recursion for calculating $\bar{P}_{5000}^{0}(x)$ without it being correctly computed, because if, due to underflow, $\bar{P}_{5000}^{5000}(x) = 0$, then all following values $\bar{P}_{5000}^{4999}(x)$, $\bar{P}_{5000}^{4998}(x)$, etc. will also become zero. On the other hand, calculating the fraction in (5) in a conventional way (first the numerator, and then the denominator) may lead to overflow exception. Some information about the range of angles and limitations to polynomial degrees at which calculations in IEEE-754 arithmetic do not result in exceptions is given in [2].

To avoid overflow or underflow problems, methods using global scaling coefficients are suggested [9]. However, as noted in [3], this solves the problem only for limited ranges of arguments and degrees. The general solution to the overflow and/or underflow problem when computing the normalized Legendre polynomials is suggested in [8] and involves the use of extended-range arithmetic.

In this paper we consider parallel computation of normalized polynomials $\bar{P}_n^m(x)$ of high degrees in extended-range arithmetic using CUDA-compatible GPUs. Due to a high level of task parallelism, the transfer of computations to the GPU has allowed to achieve significant performance improvement, as compared with the CPU implementation.

2 Extended-Range Arithmetic

2.1 Basic Algorithms

Currently, IEEE-754 standard is the main standard for binary floating-point arithmetic [12]. It defines two most widely used formats: a single-precision format (binary32) and a double-precision format (binary64). These formats are supported, to some extent, at both the hardware level and the level of programming language. In 2008, a revision to IEEE-754 standard was published, which further describes the quadruple-precision binary format—binary128, and two decimal formats—decimal64 and decimal128 [13]. However, support for these new formats is currently implemented in quite rare cases. The properties of single- and double-precision binary formats are presented in Table 1. In this table, the number of digits of the significand, p, defines precision of the format; integers $e_{\min}$ and $e_{\max}$ are the extremal exponents; $n_{\max}$ is the largest positive finite number, $n_{\min}$ is the smallest positive normal number, and $s_{\min}$ is the smallest positive subnormal number; the segment $[n_{\min}, n_{\max}]$ specifies the range of positive normal numbers, and the segment $[s_{\min}, n_{\max}]$ specifies the total range of positive finite numbers.

Table 1. The properties of IEEE-754 single-precision and double-precision formats

	p	$e_{\min}$	$e_{\max}$	$n_{\min}$	$n_{\max}$	$s_{\min}$
binary32	24	-126	$+127$	2^{-126}	$(2-2^{-23}) \times 2^{127}$	2^{-149}
binary64	53	-1022	$+1023$	2^{-1022}	$(2-2^{-52}) \times 2^{1023}$	2^{-1074}

The situation when the intermediate result of an arithmetic operation or function exceeds in magnitude the largest finite floating-point number $n_{\max} = (2 - 2^{1-p}) \times 2^{e_{\max}}$ in IEEE-754 standard is defined as *overflow*. When there is overflow, the result, depending on the used rounding mode, is replaced with infinity ($\pm\infty$) or the largest finite floating-point number. The situation when the intermediate result of an arithmetic operation is too close to zero, i.e. in magnitude it is strictly less than the smallest positive normal number $n_{\min} = 2^{e_{\min}}$ is defined as *underflow* [13,14]. When there is underflow, the result is replaced with zero, subnormal number, or the smallest positive normal number. In all cases, the sign of the rounded result coincides with the sign of the intermediate result. The exceptions examined are presented in Fig. 1.

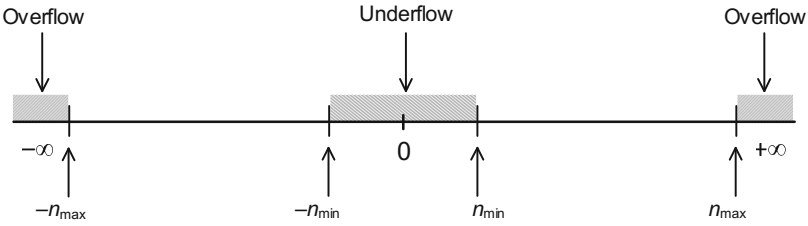

Fig. 1. Overflow and underflow in floating-point arithmetic

One of the ways to eliminate overflow or underflow is scaling. This method requires estimating the source operands and multiplying them by factor K chosen so that all intermediate results are within the normal range. After the computation of the final result, scaling is carried out by dividing it by K [14]. In terms of computing speed, this technique is evidently the best one. However, it requires a detailed analysis of the whole computing process and is not applicable in many cases. A more common approach is emulation of extended-range arithmetic. To do this, the integer e is paired with a conventional floating-point number f, and this pair is considered as a number

$$f \times B^e, \tag{6}$$

where B is a predetermined constant that is a power of the floating-point base [8, 11]. Significand f can take values in the interval $(1/B, B)$. Given this, B must be such as for any arithmetic operation performed with f, no underflow or overflow occurs. It is advisable that B is a natural power of two (for a binary computer).

For instance, if

- f is a double-precision number (binary64),
- e is a 32-bit signed integer ($-2147483648 \le e \le 2147483647$) and
- $B = 2^{256}$,

then the range of the represented numbers will exceed $10^{\pm 165492990270}$.

The algorithms for basic extended-range operations are considered in [8,11], and therefore, we will focus only on some of them. In the following, we will assume that the base of exponent B is uniquely determined and the extended-range number is represented by a pair (f, e). Algorithm 1 performs the "adjustment" of the number. It is one of the basic extended-range arithmetic algorithms. It provides control of the value range of significand f, as well as its correction in case the input is incorrect encoding of zero.

Algorithm 1. Adjustment of the extended-range representation

```
1: procedure Adjust(f, e)
2:     if f = 0 then
3:         return (0, 0)
4:     else if |f| ≥ B then
5:         f ← f/2^(log2 B)                ▷ Subtracting log2 B from exponent of f
6:         e ← e + 1
7:     else if |f| ≤ 1/B then
8:         f ← f × 2^(log2 B)              ▷ Adding log2 B to exponent of f
9:         e ← e − 1
10:    end if
11:    return (f, e)
12: end procedure
```

It is important to note that it is not always enough to carry out the Adjust procedure only one time. This can take place at least in the following two cases: (a) when the number is converted from the machine format or a format with different from the current exponent base; (b) when subtraction of almost equal numbers (or addition with different signs) is performed. In any of these cases, it is possible that, after the Adjust procedure has been performed, significand f is less than $1/B$. If it is ignored and the computation process is continued, gradual "zeroing" of the result is likely to take place. To avoid this, it is possible to use a cyclic adjustment procedure which is implemented by Algorithm 2.

Algorithm 2. Cyclic adjustment of the extended-range representation. Procedure should be used in conversion, signed addition and subtraction algorithms.

```
1: procedure CyclicAdjust(f, e)
2:     (f1, e1) ← Adjust(f, e)
3:     while e ≠ e1 or f ≠ f1 do
4:         (f, e) ← (f1, e1)
5:         (f1, e1) ← Adjust(f, e)
6:     end while
7:     return (f, e)
8: end procedure
```

Algorithm 3 performs addition of extended-range representations. Algorithms for subtraction and comparison are quite similar to the addition algorithm, so their description seems to be unnecessary.

Algorithm 3. Adding extended-range numbers, $(f_z, e_z) \leftarrow (f_x, e_x) + (f_y, e_y)$

procedure ADD(f_x, e_x, f_y, e_y)
  if $f_x = 0$ **and** $e_x = 0$ **then return** (f_y, e_y)
  else if $f_y = 0$ **and** $e_y = 0$ **then return** (f_x, e_x)
  end if
  $\Delta e = |e_x - e_y|$
  if $e_x > e_y$ **then**
    $f_z \leftarrow f_x + f_y \times 2^{-\Delta e \times \log_2 B}$
    $e_z \leftarrow e_x$
  else if $e_y > e_x$ **then**
    $f_z \leftarrow f_y + f_x \times 2^{-\Delta e \times \log_2 B}$
    $e_z \leftarrow e_y$
  else if $e_y = e_x$ **then**
    $f_z \leftarrow f_x + f_y$
    $e_z \leftarrow e_x$
  end if
  return CYCLICADJUST(f_z, e_z)
end procedure

2.2 Implementation of Extended-Range Arithmetic

We have implemented all basic algorithms of extended-range arithmetic, and a number of mathematical functions for CPUs and NVIDIA CUDA-compatible GPUs. Do to it, data types shown in Fig. 2 were declared.

```
typedef struct {
  er_frac_t frac;    //significand
  er_exp_t exp;      //exponent
} __extended_range_struct;

//single number:
typedef __extended_range_struct *er_t;
//arrays:
typedef __extended_range_struct *er_arr_t;
//for device side code:
typedef __extended_range_struct er_static_t;
```

Fig. 2. Extended-range data types: `er_frac_t`—standard floating-point number (`double` by default), `er_exp_t`—machine integer (`int64_t` by default)

The list of implemented CPU- and CUDA-functions includes the following:

- memory management and constants initialization;
- addition, subtraction, multiplication and division, supporting four IEEE-754 rounding modes, as well as comparison functions;
- integer floor and ceiling functions, computation of the fractional part;
- functions of converting numbers from the double-precision IEEE-754 data type to extended-range data type, and vice versa;
- factorial, power, square root, and a number of other mathematical functions.

The exponent base B is defined in parameters. By default $B = 2$. It is quite enough for the computation carried out. The declaration of CPU- and GPU-functions is identical (`cuda` namespace is used for GPU-functions). Pointers are used for effective passing of parameters. All functions are thread-safe.

Efficiency of extended-range arithmetic is largely determined by the speed of converting numbers from the machine floating-point representation to extended-range representation, and vice versa. To implement these procedures, we used bit-wise operations. In particular, Fig. 3 shows the subroutine `er_set_d` that converts a conventional IEEE-754 double-precision number into the extended-range representation. This subroutine uses the `DoubleIntUnion` structure, which allows storing double and integer data types in the same memory location.

```
union DoubleIntUnion {
    double   dvalue;
    uint64_t ivalue;
}

void er_set_d(er_t res, const double x) {
    DoubleIntUnion u;
    if (x == 0) {
        res->exp = res->frac = 0;
        return;
    }
    u.dvalue = x;
    uint8_t sign = (uint8_t) (u.ivalue >> SIGN_OFFSET);
    res->exp = ((u.ivalue & ~((uint64_t) 1 << SIGN_OFFSET))
             >> EXP_OFFSET) - EXP_BIAS;
    u.ivalue = u.ivalue & (((uint64_t) 1 << EXP_OFFSET) - 1)
             | ((uint64_t) EXP_BIAS << EXP_OFFSET);
    res->frac = u.dvalue;
    if (sign)
        res->frac = -res->frac;
    cyclic_adjust(res);
}
```

Fig. 3. Conversion of a double-precision floating-point number into the extended-range representation. For the double data type, `SIGN_OFFSET` = 63, `EXP_OFFSET` = 52 and `EXP_BIAS` = 1023.

3 Computation of Normalized Legendre Polynomials on CPU and GPU

3.1 Computation of Starting Point of Recursion

Our implementation of normalized Legendre polynomials computation is based on the recursion (4), which, in turn, requires computation of relation (5). In case of high degree n of polynomial, direct computation of (5) is time-consuming since it requires computing two double factorials in the extended-range arithmetic, $(2n+1)!! = 3 \cdot 5 \cdots (2n+1)$ and $(2n)!! = 2 \cdot 4 \cdots 2n$. When one polynomial is computed, it is not critical. However, the problem becomes urgent when many polynomials of various degrees are computed sequentially. In addition, in case of direct computation of factorials in the machine-precision floating-point arithmetic, significant rounding errors can accumulate. To partially solve the problem, the ROM lookup table (LUT) can be used which stores values $\frac{(i \cdot 2h+1)!!}{(i \cdot 2h)!!}$ for $i = 1, 2, \ldots, N$, where h and N are some integers. Then, for computing $\frac{(2n+1)!!}{(2n)!!}$, where n is the polynomial degree, one has to take the $\lfloor n/h \rfloor$-th value from LUT, and compute $\prod_{i=1}^{n-q} \frac{2(q+i)+1}{2(q+i)}$, where $q = h\lfloor n/h \rfloor$, and multiply these two values. The size of LUT is determined by step h and the maximum degree of polynomial n we want to compute. For instance, if $n = 50000$ and $h = 100$, LUT will contain $N = 500$ values. LUT content is computed in advance with high precision, after which it is converted into the extended-range format.

3.2 Developed Software for Computing Legendre Polynomials

Based on the implemented extended-range arithmetic functions (Subsect. 2.2), CPU- and CUDA-subroutines were developed, which allow computing $\bar{P}_n^m(x)$ for large $n \geq 0$ and at any m, $0 \leq m \leq n$. They are shown in Table 2.

For implementation on the GPU, the direct paralleling scheme was chosen, according to which i-th GPU thread computes a polynomial of degree $n[i]$ and order $m[i]$ for argument $x[i]$. The result is written with a corresponding offset to `res` array. The number of the required thread blocks is defined by the following:

$$N = \left\lceil \frac{vector\_size}{max\_threads\_per\_block} \right\rceil, \tag{7}$$

where $vector\_size$ is the size of the input vectors, $max\_threads\_per\_block$ is the maximum number of threads in a block. If N does not exceed the maximum number of blocks for the device, fully parallel computation of all polynomials is possible. Otherwise, some threads compute more than one polynomial. Listing of CUDA kernel `legendre_lst` is given in Fig. 4.

4 Experimental Results

The evaluation of correctness and efficiency of the developed subroutines was carried out within HP SL390+NVIDIA Tesla M2090 stand of UniHUB.ru platform at the Institute for System Programming of the Russian Academy of Sciences [15]. Three software implementations of the recursive algorithm (4) have

Table 2. Subroutines to compute normalized Legendre polynomials

Subroutine	Parameters	Description

`legendre_eqls`	`er_t res`
	`er_t x`
	`uint32_t const n`
		Computation of $\bar{P}_n^n(x)$ in accordance with (5) with optimization from Subsect. 3.1. The result is a pointer `res`.

`legendre_recur`	`er_t res`
	`er_t x`
	`er_t p1`
	`er_t p2`
	`uint32_t const n`
	`uint32_t const m`
		One iteration of recursion (4). For the given $\bar{P}_n^m(x)$ (parameter `p1`) and $\bar{P}_n^{m+1}(x)$ (parameter `p2`) $\bar{P}_n^{m-1}(x)$ is computed. The result is a pointer `res`.

`legendre`	`er_t res`
	`double const x`
	`uint32_t n`
	`uint32_t m`
		Computation of normalized Legendre polynomial $\bar{P}_n^m(x)$ of degree `n` and order `m`. The result is a pointer `res`.

`legendre_lst`	`er_arr_t res`
	`double const *x`
	`uint32_t const *n`
	`uint32_t const *m`
	`uint32_t size`
		Computation of the vector of normalized Legendre polynomials for given vector of arguments `x`, vector of degrees `n`, and vector of orders `m`. The result is a pointer `res` to an array.

been examined: CPU- and GPU-implementations based on extended-range arithmetic, and calculations using the GNU MPFR Library.

In the first experiment, we examined dependence of computation time for the first-kind polynomial on n. The value $\cos(179^\circ) \approx -0.999848$ was taken as an argument. The degree n varied in the range of 100 to 53200, and it was doubled at each stage of the testing procedure. The results are presented in Fig. 5(a).

In the second experiment, vectors of polynomials were calculated at fixed $m = 0$ and $n = 20000$, whose size ranged from 32 to 8192. The arguments were defined by the formula $x_i = \cos\left(i \times \frac{180}{vector\_size}\right)$, which allowed calculations for each vector in the angular range $[0^\circ, 180^\circ]$, having a uniform step determined by the size of the vector ($vector\_size$). The experiment results are shown in Fig. 5(b). Longer GPU computation time, observed at the vector size greater than 2048, is explained by the limited resources of the used device.

It is worth noting that the subroutines for computation of normalized and non-normalized associated Legendre polynomials are implemented in a number of well-known software packages, such as The GNU Scientific Library, Boost, ALGLIB. However, they allow calculations only for rather small degrees (up to several thousand). Therefore, these implementations were not analysed in the experiments.

```
__global__ void legendre_lst(er_arr_t res,
                             double const *x,
                             uint32_t const *n,
                             uint32_t const *m,
                             uint32_t size){
const uint32_t id = threadIdx.x + blockIdx.x * blockDim.x;
  if (id < size) {
    uint32_t thread_n = n[id];
    uint32_t thread_m = m[id];
    er_static_t thread_x;
    cuda::er_set_d(&thread_x, x[id]);
    legendre_eqls(&res[id], &thread_x, thread_n);
    if (thread_n > thread_m) {
      er_static_t p0, p1, p2;
      cuda::er_set(&p1, &res[id]);
      cuda::er_set_d(&p2, 0.0);
      uint32_t current_m = thread_n;
      legendre_recur(&p0,&thread_x,&p1,&p2,thread_n,current_m);
      uint32_t iter_n = thread_n - thread_m - 1;
      for (uint32_t i = 0; i < iter_n; i++) {
        current_m = current_m - 1;
        cuda::er_set(&p2, &p1);
        cuda::er_set(&p1, &p0);
       legendre_recur(&p0,&thread_x,&p1,&p2,thread_n,current_m);
      }
      cuda::er_set(&res[id], &p0);
    }
  }
}
```

Fig. 4. CUDA kernel for computing normalized Legendre polynomials

The computed polynomials $\bar{P}_n^m(\cos\theta)$ for $n = 1000, 5000, 15000, 20000$, $m = 0$ with intervals of θ equal to 1° are shown in Fig. 6, and the logarithms of the starting values (5) for recursion (4) are shown in Fig. 7.

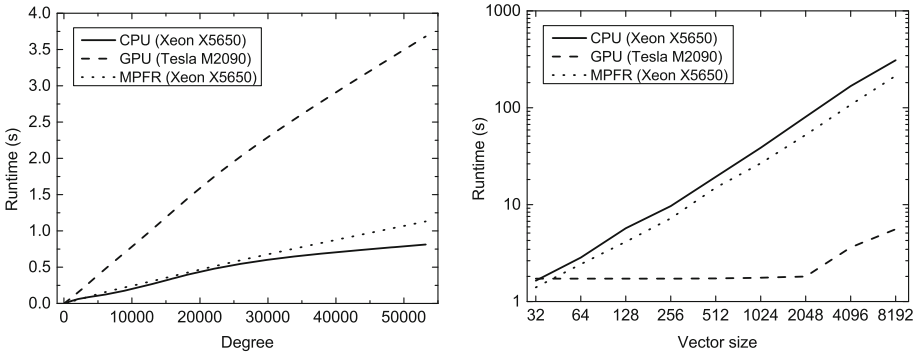

Fig. 5. Experimental results: computation time of $\bar{P}_n^m(x)$ at fixed $m = 0$ and $x = \cos(179°)$ versus degree n (a); computation time of the vector of $\bar{P}_n^m(x)$ at fixed $m = 0$ and $n = 20000$ versus the vector size (b)

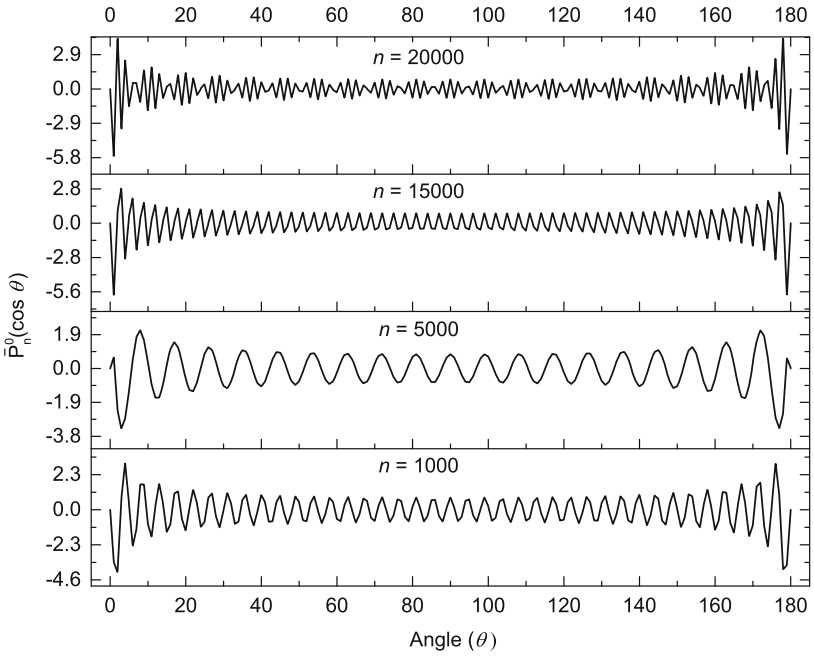

Fig. 6. Normalized associated Legendre polynomials

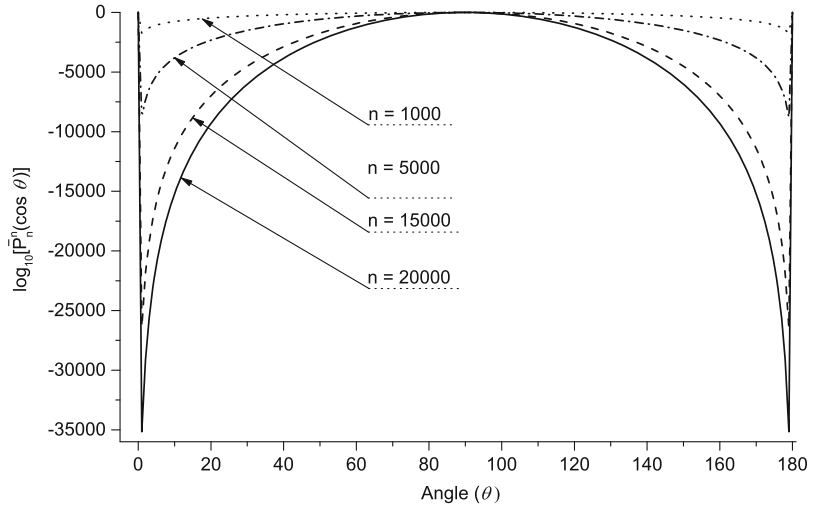

Fig. 7. Logarithms of the starting values for recursive computation of normalized associated Legendre polynomials

5 Conclusion

The paper considers the problem of GPU-based calculation of normalized associated Legendre polynomials. At high degrees n and orders m, this polynomials are characterized by a large spread of numerical values, which greatly limits the possibility of their correct computation in IEEE-754 arithmetic. In particular, when $n = 20000$, obtaining correct results in the double-precision format is only possible for angles ranging from 75° to 105°. Calculations for angles beyond this range result in underflow. To overcome this limitation, extended-range arithmetic is implemented on GPU. The experimental evaluation shows that, thanks to the natural task parallelism and a simple computation scheme, the use of GPU is effective even with small length vectors. With increasing problem size the speedup becomes more significant.

When parallel computation of polynomials of the same degree for a set of different arguments is made, the computation scheme is balanced, since time complexity of the extended-range arithmetic operations does not depend significantly on the magnitude of the arguments. If vectors of polynomials of different (high) degrees are computed, then, to improve the GPU-implementation performance, a more complicated computation scheme involving load balancing can be applied.

Acknowledgement. This work was supported by the Russian Foundation for Basic Research, project No. 16-37-60003 mol_a_dk.

References

1. Arfken, G.B., Weber, H.J., Harris, F.E.: Mathematical Methods for Physicists, 7th edn. Academic Press, Boston (2013)
2. Wittwer, T., Klees, R., Seitz, K., Heck, B.: Ultra-high degree spherical harmonic analysis and synthesis using extended-range arithmetic. J. Geodesy **82**(4), 223–229 (2008). doi:10.1007/s00190-007-0172-y
3. Fukushima, T.: Numerical computation of spherical harmonics of arbitrary degree and order by extending exponent of floating point numbers. J. Geodesy **86**(4), 271–285 (2012). doi:10.1007/s00190-011-0519-2
4. Morris, R.J., Najmanovich, R.J., Kahraman, A., Thornton, J.M.: Real spherical harmonic expansion coefficients as 3D shape descriptors for protein binding pocket and ligand comparisons. Bioinformatics **21**(10), 2347–2355 (2005). doi:10.1093/bioinformatics/bti337
5. Rose, M.: Elementary Theory of Angular Momentum. Wiley, New York (1957)
6. Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., Booth, M., Rossi, F., Ulerich, R.: GNU Scientific Library (2016). https://www.gnu.org/software/gsl/manual/gsl-ref.pdf
7. Belousov, S.: Tables of Normalized Associated Legendre Polynomials. Pergamon Press, New York (1962)
8. Smith, J.M., Olver, F.W.J., Lozier, D.W.: Extended-range arithmetic and normalized Legendre polynomials. ACM Trans. Math. Softw. **7**(1), 93–105 (1981). doi:10.1145/355934.355940

9. Wenzel, G.: Ultra-high degree geopotential models GPM98A, B, and C to degree 1800. In: Joint Meeting of the International Gravity Commission and International Geoid Commission, Trieste (1998)
10. Holmes, S.A., Featherstone, W.E.: A unified approach to the Clenshaw summation and the recursive computation of very high degree and order normalised associated Legendre functions. J. Geodesy **76**(5), 279–299 (2002). doi:10.1007/s00190-002-0216-2
11. Hauser, J.R.: Handling floating-point exceptions in numeric programs. ACM Trans. Program. Lang. Syst. **18**(2), 139–174 (1996). doi:10.1145/227699.227701
12. IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985, pp. 1–20 (1985). doi:10.1109/IEEESTD.1985.82928
13. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008, pp. 1–70 (2008). doi:10.1109/IEEESTD.2008.4610935
14. Muller, J.M., Brisebarre, N., de Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser Boston, New York (2010)
15. The "University cluster" program's technological platform. https://unihub.ru/resources/sl390m2090