Verification of Kurepa's Left Factorial Conjecture for Primes up to 2^{31}

Ilijašević, Igor

***Abstract*—In this article we utilize Graphics Processing Units (GPUs) as a way to expedite massive computation needed to check Kurepa's hypothesis on left factorials for primes up to 2^{31}. We also give an estimate of computational efforts and an existing algorithm which can be used to extend the computations up to 2^{32}.**

***Index Terms*—CUDA, GPU, Kurepa, left factorial, primes.**

I. Introduction

KUREPA [1] defined left factorial as a sum

$$!n = \sum_{i=0}^{n-1} i!, \quad n \text{ integer} \tag{1}$$

and stated the conjecture (see [2, Problem B44] and the overview paper [3]) that

$$!n \not\equiv 0 \pmod{n}, \quad n > 2, \tag{2}$$

or equivalently $\gcd(!n, n!) = 2$. Let $r_p = !p \pmod{p}$ for an arbitrary prime p. An equivalent formulation of the conjecture is

$$r_p \neq 0 \quad \text{for all primes } p > 2. \tag{3}$$

The conjecture is previously verified by Mijajlović [4] for $p \leq 311009$, Živković [6] for $p < 2^{23}$, Gallot [7] for $p < 2^{26}$, Jobling [8] for $p < 144000000$ and Tatarević [9] for $p < 10^9$.

We extended the computation up to $p < 2^{31}$ using NVIDIA CUDA GTX 285 and GTX 780 GPUs. Given p, the value of r_p (3) is computed by obvious algorithm

Algorithm 1 Calculating left factorial modulus

```
1: procedure LeftFactorialMod(p)
2:     f = 1                                  ▷ 0! = 1
3:     s = 1                                  ▷ !0 = 1
4:     for i = 1 to p do
5:         f = f * i (mod p)                  ▷ f = i! (mod p)
6:         s = s + f (mod p)                  ▷ s = !i (mod p)
7:     end for
8:     return s                               ▷ s = !p (mod p)
9: end procedure
```

Given that $p < 2^{32}$, the values of all variables can be stored in a 64-bit representation. The calculation is not straightforward, because of the lack of efficient implementation of modular arithmetic on GPU.

Manuscript received January 27, 2014.

Ilijašević Igor is a PhD student with the Faculty of Mathematics, University of Belgrade, Belgrade, Serbia (e-mail: ilijasevic.igor@gmail.com).

Milovanović [5] introduced a generalization of left factorials, the numbers

$$K_m(n) = \sum_{i=0}^{n-1} \frac{(-1)^i}{i!} \sum_{v=i}^{n-1} v! \binom{m+n}{v+m+1}, \; K_m(0) = 0. \tag{4}$$

In terms of these numbers $n! = K_{-1}(n+1)$ and $!n = K_0(n)$. Let

$$A = \{13k + 9 | k \in \mathbb{N}\}$$

Milovanović formulated two new conjectures,

$$(\forall n \in \mathbb{N}\backslash\{5\}) \quad \gcd(K_0(n), K_1(n)) = 1, \tag{5}$$

($\gcd(K_0(5), K_1(5)) = \gcd(34, 51) = 17$) and

$$\gcd(K_1(n), K_2(n)) = \begin{cases} 1, & n \in \mathbb{N}\backslash A \\ 13, & n \in A \end{cases} \tag{6}$$

After a not too hard computer search we found counterexamples to these conjectures.

II. Overview of the Graphics Processing Units and CUDA framework

In order to extend the previous range of primes for which (3) is checked, the obvious choice is to do computations in parallel.

We decided to use nVidia hardware and framework. It should be noted that while a heterogeneous programming standard OpenCL exists and is supported by both nVidia and other vendors, due to the fact that we had access to nVidia hardware and the fact that nVidia OpenCL support is both behind in terms of performance and version support, we have chosen to use nVidia only supported options. However it is possible to modify programs to comply with OpenCL.

While evaluating different GPU architectures and CUDA versions one should be aware of their respective capabilities. Performance capabilities of various GPUs vary even in the same architecture class, for example GTX 780's 1/24 compared to GTX Titan's 1/3 double precision performance with respect to single precision floating point operations. Differences are even bigger for different architectures. For example, pre-Fermi class GPUs integer operations are internally realized using 24-bit arithmetic, compared to 32-bit arithmetic in latter devices; pre-Tesla cards lack the double precision arithmetic, etc.

Our work was done primarily on GTX 285 with GTX 780 being available later merely as an additional device and as such no advanced Kepler capabilities have been utilized. Basic differences can be seen in Table I. An explanation of the

capabilities of devices with compute capability 1.3 and 3.5 can be found in [11, Chapter 2.5,Chapter 5.4.1,Appendix G.1.].

Here we present CUDA terminology used in our work.

Each CUDA capable device consists of several Streaming Multiprocessors (SM) which can vary in number.

- Each SM consists of a number of scalar in-order processors each executing same instruction at the same time using threads.
- Group of 32 threads is called a warp and is the minimum size of the data processed synchronously by SM. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path.
- Blocks present a group of threads which are defined by the programmers and variable in size, all threads in a block can coordinate with each other.
- Grid is a collection of blocks which are completely independent so there is no coordination and synchronization within a grid.

There are several memory hierarchies available in CUDA:

- Global memory is SDRAM memory available on the GPU. Each thread can read and write to this memory. It is the largest and the slowest memory available on the GPU.
- Texture cache is memory within each SM that can be used as read only cache. This memory is used if there exists a high degree of spatial locality in memory accesses.
- Constant memory is read only memory available for each SM.
- Shared memory is a small amount of memory available for each SM which can be both written and read by any thread in a block running on the same SM.
- Each SM contains a number of registers available for use by CUDA program threads which are shared resources allocated among the thread blocks executing on an SM.

Occupancy is the ratio of active warps to the maximum number of warps supported on an SM of the GPU. It is calculated in the form of

$$Occupancy \ = \ \frac{Active \ Warps}{Maximum \ Active \ Warps}.$$

Increase in usage of the following resources has a negative effect on occupancy:

- Register usage
- Shared memory usage
- Block size

We shall note the size of a block as SB and the number of threads per block as TPB.

III. Implementation of modular multiplication

Integer operations on GPU are very constrained. GPUs with Tesla architecture support only 24-bit integer multiplication natively. The following generations support 32-bit integer arithmetic fully in hardware.

Modulus operation on the GPU is extremely expensive. However if the modulus is known at compile time or if it is a constant, as in $\mathbf{LeftfactorialMod}(p)$, the remainder operation can be replaced using various methods, e.g. multiplication by precomputed divisor reciprocal. We have done our calculations using three such replacements, both for Tesla and Kepler architectures. All of the following algorithms use a precomputed constant (magic number) and seek to replace the expensive modulus operations with more suitable multiply, add and shift integer operations, or double precision floating point operations. Functions which are executed on the GPUs are called kernels.

TABLE I
Basic device information; for a more detailed device specification see [12], [13].

	GTX 285	GTX 780
Architecture	Tesla	Kepler
Compute capability	1.3	3.5
Streaming Multiprocessors (SM)	30	12
Maximum number of threads per block	512	1024

It should be noted that the following algorithms utilize the **umulhi** operation which gives us the 32 most significant bits (MSB) when multiplying two unsigned 32-bit numbers. We will note the least significant bits as LSB.

As the largest possible factorial we can keep in 64 bits is 20! and given that the modulus is ever changing, we have decided to run each calculation independently in parallel preventing us from using shared variables over the entire block. In simplest terms each thread is independent and calculates factorial and left factorial for a specific prime.

As we can see, in order to achieve the best results we have to take several things in consideration:

- We should strive for maximum occupancy which forces us to take special care of number of registers used by each thread which in turn forces us to seek more "lightweight" algorithms.
- Thread divergence is unfortunately very difficult to deal with the way we have decided to tackle the problem. As the distance between primes is not constant (minimum distance = 2, average = 22, maximum = 292) one of the possible ways to solve this problem is to group primes with similar distances together. Then after finishing the calculation for the first prime in a block reshuffle the prime execution on the threads and execute them in the same warps if possible. However this method requires the use of additional shared memory and in our tests it has not shown any additional performance improvements. Given that we operate with numbers over 10^9, we can see that in average case the divergent path, while considerable, still represents a small percentage of the whole execution:
$$\frac{p}{TPB * 22}, \quad 2^{31} > p > 10^9$$
- We should avoid using integer arithmetic on the GTX 285 and 64-bit integer arithmetic on the GTX 780. Unfortunately the double precision of the cards we have available rather limited compared to nVidia's commercial scientific cards, and as such results presented here are not representative of other cards.

When writing the code we have used the tools made available by nVidia's CUDA Toolkit [10]. Some of the tools we used:

- Nsight monitor is a debugging tool invaluable in debugging the code and in bug detection regarding memory accesses.
- CUDA GPU Occupancy Calculator is used to determine the ideal theoretical block size as it contains the information regarding the various compute capabilities. It does so by giving us the theoretical occupancy of a kernel based on the number of parameters. Occupancy is given in percentages and it gives us an idea of the GPU pipeline utilization.
- Visual Profiler is a profiler valuable in showing us the various usages regarding the kernels running on the GPU.

Implementation 1 [14] is the algorithm employed by the nVidia CUDA Compiler (nvcc) driver. The basis of the algorithm requires us to calculate:

- the magic number
- required number of right shifts
  - is it a special case requiring an additional add operation on the CPU
- send the data to the GPU

However this algorithm requires that both numerator and denominator be represented as same width integers (in our case 64-bit). This presents a problem since GPUs map 64-bit instructions to multiple native 24-bit or 32-bit instructions depending on their compute capabilities. Also the algorithm has two distinct ways of operation depending on whether the divisor and its magic number require an additional add operation which complicates the code and require either an if statement in the kernel code or two separate kernels.

Algorithm 2 First implementation of modular multiplication

1: $mag, shift, add$ ▷ calculate magic number, shift amount and if an addition is required before sending the prime to the GPU
2: **procedure** NVCC_MUL_MOD($a, b, p, mag, shift, add$)
3: $lo = a * b$ ▷ 32 LSB
4: $hi =$ umulhi(lo, mag) ▷ 32 MSB
5: **if** add **then**
6: $sub = lo - hi$
7: $shr = sub \gg 1$
8: $addition = shr + hi$
9: $shr = addition \gg shift$
10: $mul = shr * p$
11: **else**
12: $shr = hi \gg shift$
13: $mul = shr * p$
14: **end if**
15: **return** $(lo - mul)$ ▷ remainder
16: **end procedure**

Implementation 2 [15] is a much more promising algorithm which utilizes the double precision floating point operations (available in devices with compute capability 1.3 and up) as a way to speed up the modulus calculation. One should note that this algorithm allows only up to full-precision 31-bit modular arithmetic. The algorithm also utilizes the magic number 3^{51} [16] as a replacement for a truncate function. When compiled, it uses 14 registers for both GTX 285 and GTX 780.

As an example, we give a simplified implementation of modular multiplication for Kepler (Fermi) architecture from [15].

Algorithm 3 Second implementation of modular multiplication

1: $inv = (\text{double})(1 \ll 30)/p$ ▷ calculate inv before sending the prime to the GPU
2: **procedure** MUL_MOD(a, b, p, inv)
3: $hi =$ umulhi$(a * 2, b * 2)$ ▷ 32 MSB
4: $rf =$ truncate$(hi * inv)$ ▷ $rf = \lfloor r/p \rfloor$
5: $r = a * b - rf * p$ ▷ partial residue
6: **return** $(r < 0\ ?\ r + p : r)$ ▷ adjust by p if negative
7: **end procedure**

Implementation 3 [18] is an improvement of [17]. This algorithm allows us to expand the modulus operation from 2^{31} to 2^{32} and as such it is a supplement to the previous algorithm if one wishes to test up to 2^{32}. This algorithm also seeks to minimize the use of 64-bit arithmetic in favor of 32-bit arithmetic. One of the differences compared to the original algorithm is the fact that we have removed the calculation of the quotient since it is not required in our case. Using **unsigned** variables gives us automatic $(\mathrm{mod}\ 2^{32})$ required by some parts of the algorithm. As some parts of the algorithm need 64-bit variables, we have used lines 4 and 3 to note the required type and size of the variables. When compiled it uses 17 registers for both GTX 285 and GTX 780. We would have tested the mentioned range using this algorithm, however we were time limited in regards to the GPU hardware access.

IV. Results

Using nVidia CUDA GPU Occupancy Calculator found in [10] we can see that register usage for both Implementations 2 and 3 gives us 100% occupancy for the following block sizes:

- GTX 285 - 192, 256, 384, 512
- GTX 780 - 128, 256, 512, 1024.

After testing we have found the differences between the block sizes to be under 1232 milliseconds with the larger block sizes being faster. Given the previous results we have decided on the following: results r_p are received in blocks; the size of a block is obtained by multiplying the number of Streaming Multiprocessors that are used by the number of threads per block, which is chosen to be equal to maximum number of threads per block. The size of a block is chosen to be $SB_1 = 30 \cdot 512 = 15360$ and $SB_2 = 12 \cdot 1024 = 12288$ for GTX 285 and GTX 780, respectively.

Let p_i denote the i-th prime, $p_1 = 2$, and let $\pi(x) = \max\{i \mid p_i \leq x\}$. Let $a = \pi(10^9) + 1 = 50847535$, $b = \pi(2^{31}) + 1 = 105097565$ and $c = \pi(2^{32}) = 203280221$. Then $p_a = 10^9 + 7$, $p_b = 2^{31} + 11$ and $p_c = 2^{32} - 5$. Let $P(u, S) = \{p_u, p_{u+1}, \ldots, p_{u+S-1}\}$ denote the set of S primes starting with p_u. Let $SP_{j,k}$ denote sets used for performance

Algorithm 4 Third implementation of modular multiplication

```
rec   ▷ calculate reciprocal before sending the prime to the GPU
procedure MUL_MOD(a, b, p, rec)
    unsigned 32-bit q_1, q_0, r, u_1, u_0          ▷ 32-bit vars
    unsigned 64-bit ab, q_1q_0                     ▷ 64-bit vars
    ab = a * b                                     ▷ 64-bit product
    u_1 = ab ≫ 32                                  ▷ 32 MSB
    u_0 = ab                                       ▷ 32 LSB
    q_1q_0 = rec * u_1                             ▷ 64-bit product
    q_1q_0 = q_1q_0 + ab
    q_1 = q_1q_0 ≫ 32                              ▷ 32 MSB
    q_0 = q_1q_0                                   ▷ 32 LSB
    q_1 = q_1 + 1          ▷ unsigned gives us (mod 2^32)
    r = u_0 − (q_1 * p)    ▷ unsigned gives us (mod 2^32)
    if r > q_0 then                                ▷ unpredictable condition
        r = r + p          ▷ unsigned gives us (mod 2^32)
    end if
    if r ≥ p then                                  ▷ unlikly condition
        r = r − p
    end if
    return r                                       ▷ remainder
end procedure
```

TABLE II
AVERAGE RUNTIME IN SECONDS OVER 20 RUNS.

	GTX 285				GTX 780			
	$SP_{1,1}$	$SP_{1,2}$	$SP_{1,3}$	$SP_{1,4}$	$SP_{2,1}$	$SP_{2,2}$	$SP_{2,3}$	$SP_{2,4}$
I1	24829	53303	53328	106640	717	1531	1566	3203
I2	1891	4126	-	-	428	911	-	-
I3	-	-	5584	12970	-	-	964	2081

TABLE III
AVERAGE SPEEDUP.

	GTX 285				GTX 780			
	$SP_{1,1}$	$SP_{1,2}$	$SP_{1,3}$	$SP_{1,4}$	$SP_{2,1}$	$SP_{2,2}$	$SP_{2,3}$	$SP_{2,4}$
I1	1x	1x	1x	1x	1x	1x	1x	1x
I2	13.13x	12.92x	-	-	1.68x	1.72x	-	-
I3	-	-	9.54x	8.22x	-	-	1.59x	1.54x

comparison of various implementations: $SP_{j,1} = P(a, SB_j)$, $SP_{j,2} = P(b - SB_j + 1, SB_j)$, $SP_{j,3} = P(b, SB_j)$, $SP_{j,4} = P(c - SB_j + 1, SB_j)$, where $j = 1, 2$ for GTX 285 and GTX 780 respectively.

The results are given in Table II (average runtime in seconds over 20 runs over blocks), Table III (relative speedups of Implementations 2 and 3, compared to Implementation 1), and Table IV (average time to calculate r_p for a single prime p, calculated as $time/SB$ in ms). The rows labeled I1, I2, I3 correspond to the three previously described implementations.

As can be seen from Table II because GTX 285 lacks proper 32-bit arithmetic Implementation 1 is not ideal, while GTX 780 gives respectable performance for such implementation. Table III clearly shows that while using different implementations gives massive performance improvements for GTX 285, improvements are more modest for GTX 780. Table IV merely states that due to the way GPUs operate, a single threaded program would have to process $\mathbf{LeftfactorialMod}(p)$ for one prime in a given time frame in order to be competitive.

TABLE IV
AVERAGE TIME TO CALCULATE r_p FOR A SINGLE PRIME p, CALCULATED AS $time/SB$ IN MS.

	GTX 285				GTX 780			
	$SP_{1,1}$	$SP_{1,2}$	$SP_{1,3}$	$SP_{1,4}$	$SP_{2,1}$	$SP_{2,2}$	$SP_{2,3}$	$SP_{2,4}$
I1	1616	3470	3471	6942	58	124	127	260
I2	123	268	-	-	34	74	-	-
I3	-	-	363	844	-	-	78	169

TABLE V
THE VALUES OF $!p$ FROM 10^9 UP TO 2^{31} CLOSE TO 0 OR p.

p	r_p	p	$p - r_p$
1023141859	96	1278568703	6
1167637147	67	1330433659	75
1250341679	15	1867557269	42
1283842181	80		

The results are given in Table V, where the values of r_p and $p - r_p$ satisfying $r_p \leq 100$ or $r_p \geq p - 100$, respectively, are listed, accompanied by the corresponding primes p.

These values were checked using Wolfram Mathematica 9.0.1.0 and are correct according to the implementation of the $\mathbf{LeftfactorialMod}(p)$ given in algorithm 1.

It should be noted that the calculations were done on commercial hardware which lacks Error-correcting code (ECC) memory and zero error tolerance stress testing present in enterprise class hardware. ECC memory is capable of detecting and correcting data corruptions due to internal (magnetic or electrical interference) or external factors (background radiation). ECC memory can correct single bit errors using Hamming codes or triple modular redundancy.

Given the lack of ECC we have decided to validate our results by statistical sample using the formula given by [19] in order to reach the following values: Population size = 54250031, Confidence = 99.9%, Margin of error = 0.5%, Sample size = 108060.

After completion we have randomly chosen 108060 elements and calculated their values on two different central processing units (CPUs). In our validation we have found no errors while comparing the random subset and this works results.

V. COUNTEREXAMPLES TO CONJECTURES ON GENERALIZED LEFT FACTORIALS

It is easily proved that the numbers $K_m(n)$ (4) satisfy the recurrent relation

$$K_{m-1}(n) + K_m(n-1) = K_m(n)$$

implying

$$\sum_{i=0}^{n-1} K_{m-1}(i) = K_m(n)$$

and

$$K_m(n) = \sum_{i=0}^{n-1} \binom{n-i+m-1}{m} i!$$

Consider first the conjecture (5). Suppose $p|\gcd(K_0(n), K_1(n))$ for some prime $p > 5$ and for some n. There are two cases:

- If $n \geq p$ then $K_0(n) \equiv \sum_{i=0}^{p-1} i! \equiv r_p \pmod p$, and $K_1(n) \equiv \sum_{i=0}^{p-1}(n-i)i! = \sum_{i=0}^{p-1}(n-(i+1)+1)i! = n\sum_{i=0}^{p-1} i! - p! + 1 \equiv nr_p + 1 \pmod p$. Consequently, if $K_0(n) = 0 \pmod p$ then $K_1(n) \equiv 1 \neq 0 \pmod p$, and $K_0(n) \equiv K_1(n) \equiv 0 \pmod p$ has no solutions.
- If $n < p$ then $K_0(n) = !n \pmod p$, $K_1(n) = n\cdot!n - n! + 1 \pmod p$ and the system $K_0(n) \equiv K_1(n) \equiv 0 \pmod p$ is equivalent to the system $!n = 0 \pmod p$, $n! = 1 \pmod p$. Searching for pairs (n, p) satisfying this system is similar to Algorithm 1: for fixed prime p one has to check for all $n < p$ if $!n = 0 \pmod p$, $n! = 1 \pmod p$ is true. The first such pair is $(n, p) = (372085, 425701)$, i.e. $372085! = 1 \pmod{425701}, !372085 = 0 \pmod{425701}$.

Hence, the smallest counterexample is $425701|\gcd(K_0(372085), K_1(372085)$.

Consider now the conjecture (6). Suppose $p|\gcd(K_1(n), K_2(n))$ for some prime $p > 5$ and for some $n \geq p$. Then

$$K_1(n) \equiv \sum_{i=0}^{p-1}(n-i)i! \pmod p$$

is a linear polynomial in n and

$$K_2(n) \equiv \sum_{i=0}^{p-1} \binom{n-i+1}{2} i! \pmod p$$

is a quadratic polynomial in n. Hence, it is straightforward to check if $K_1(n) \equiv K_2(n) \equiv 0 \pmod p$ has some solution n. The first p leading to such an counterexample is $p = 859$; then

$$K_1(n) = 1 + 261n \pmod{859},$$

$$K_2(n) = 300 + 561n + 560n^2 \pmod{859}$$

and the system $K_1(n) \equiv K_2(n) \equiv 0 \pmod{859}$ has the solutions $n = 260 + 859k, \quad k \in \mathbb{N}$. The condition $n \not\equiv 9 \pmod{13}$ is equivalent to $k \not\equiv 9 \pmod{13}$. Hence there are many counterexamples to (6), and the smallest is obtained for $n = 260 + 859 = 1119$: $859|\gcd(K_1(1119), K_2(1119))$.

VI. Conclusion

In this work we used several implementations of modular multiplication on GPU in order to check Kurepa's left factorial conjecture for primes up to 2^{31}.

However we have also shown that while it is possible to achieve an improvement, the success is dependent on the GPU architectures. While even earlier devices with compute capability 1.3 are usable for this sort of calculation, the lack of actual 32-bit arithmetic is a major problem.

Our long term goal is to resolve the range of primes up to 2^{32}. Should we use only one GTX 780 we can see that we have to solve 7991 blocks rising to an approximate runtime of 140 days. However we should also strive to potentially change the approach and algorithm used to a more suitable one (utilizing shared memory, minimizing thread divergence and seeking to replace multiplication dependent algorithms with a combination of multiplication and addition algorithms.

While researching Kurepa's left factorial we have found several counterexamples with regards to conjectures on generalized Kurepa's left factorials.

Acknowledgment

We would like to thank Prof. Miodrag Živković who helped devise the premise of this work and for his helpful comments and suggestions and Prof. Gradimir V. Milovanović for informing us about generalization of Kurepa's left factorial and his conjectures.

We would also like to thank various other persons who have been gracious enough to grant us the use of the hardware we utilized in our work.

References

[1] Kurepa Đ., "On the left factorial function," *Math. Balkanica*, 1971, pp. 147-153

[2] Guy R., "Unsolved problems in number theory, 3rd edition," *Springer-Verlag*, 2004, pp. 175

[3] Ivić A., Mijajlović Ž., "On Kurepa's problems in number theory," *Publ. Inst. Math. (Beograd)*, 1995, pp. 19–28

[4] Mijajlović Ž., "On some formulas involving !n and the verification of the !n hypothesis by use of computers," *Publ. Inst. Math. (Beograd)*, 47(61), 1990

[5] Milovanović G. V., "A sequence of Kurepa's functions," *Scientific Review*, 19-20, 1996, pp. 137-146

[6] Živković M., "Massive computation as a problem solving tool," Proceedings of the 10th Congress of Yugoslav Mathematicians (Belgrade, 2001), *Univ. Belgrade Fac. Math.*, Belgrade, 2001

[7] Gallot Y., "Is the number of primes $\frac{1}{2}\sum_{i=0}^{n-1} i!$ finite ?," http://yves.gallot.pagesperso-orange.fr/papers/lfact.html, 20. July. 2000

[8] Jobling P., "A couple of searches," http://groups.yahoo.com/neo/groups/primeform/conversations/topics/5095, 2. December. 2004

[9] Tatarević M., "Searching for a counterexample to the Kurepa's left factorial hypothesis (p < 10^9)," http://mtatar.wordpress.com/2011/07/30/kurepa/, 30. July. 2011

[10] -, "CUDA Toolkit | NVIDIA Developer Zone," *NVIDIA Corporation*, https://developer.nvidia.com/cuda-toolkit

[11] -, "CUDA C Programming Guide 5.5," *NVIDIA Corporation*, 2013

[12] -, "GeForce GTX 285 | GeForce," *NVIDIA Corporation*, http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-285

[13] -, "GeForce GTX 780 Graphics Card with Kepler Technology | GeForce | GeForce," *NVIDIA Corporation*, http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-780

[14] Henry S. Warren Jr., "Hacker's Delight, Second Edition," *Addison-Wesley Professional*, 2012, pp. 227-234

[15] Emeliyanenko P., "Computing resultants on Graphics Processing Units: Towards GPU-accelerated computer algebra," Journal of Parallel and Distributed Computing, 2013, pp. 1494-1505

[16] Hecker C., "Let's get to the (floating) point," Game Developer Magazine, 1996, pp. 19-24

[17] Granlund T., P. L. Montgomery, "Division by invariant integers using multiplication," Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, 1994, pp. 61-72

[18] Möller N., Granlund T., "Improved Division by Invariant Integers," IEEE Transactions on Computers, 2011, pp. 165-175

[19] Krejcie R. V., Morgan D. W. "Determining Sample Size for Research Activities," Educational and Psychological Measurement 30, 1970, pp. 607-610