Arbitrary-Precision Arithmetics on the GPU

Bernhard Langer
Supervised by: Thomas Auzinger

Vienna University of Technology

Abstract

The majority of computer applications employ numerical data types with a fixed amount of precision for their computations. Their limited numerical range and precision are sufficient for most use cases. However, for some purposes, such as cryptography or geometrical computations, the required range and precision can become arbitrarily large. Numerical types that can handle such demands have higher memory requirements and are not natively supported by common hardware, which leads to increased computational complexity. In this paper, we examine how basic arithmetic operations on arbitrary-precision integers can be adapted to many-core architectures in the form of graphics processing units, which are widely available as commodity hardware. Apart from a detailed description of our method, we show superior performance characteristics of our implementation in comparison to state-of-the-art CPU libraries for high computational loads.

Keywords: gpu, cuda, integer arithmetics, parallel, arbitrary-precision

1 Introduction

Fixed-Precision Number Formats Arithmetic computations in most programs are performed using number formats with a fixed precision. These types allocate a constant amount of memory for each number to store its value and, therefore, only a limited amount of different values is available. A set of fixed precision formats is natively supported by common processing hardware, usually given by power-of-two binary lengths, e.g., 8-64 bits. Other arbitrary but fixed lengths have to be mapped to the hardware capabilities by software means.

Two main types of numbers can be differentiated: Fixed-precision *integers* are mostly used for counting or addressing purposes and are limited to a specific numerical range. Arithmetic operations on such numbers can result in an under- (resp. overflow), where the result of a computation is larger than the largest (resp. smaller than the smallest) possible value of the given data type. Fixed-precision *floating-point numbers* are used to represent approximations of real numbers and are limited both in precision and range. Consequently, rounding errors are a common downside, with implications depending on the application scenario.

Arbitrary-Precision Number Formats In some cases, fixed-length number types are not sufficient; for example, if the largest occurring value of an integer is not known prior to execution or if rounding errors of floating point arithmetics cannot be tolerated. In these cases, we can make use of arbitrary-precision number formats, for which the numeric range and precision are chosen dynamically. Arbitrary-precision arithmetics are essential to many applications, such as geometric algorithms or public-key cryptography [4].

Standard number formats are part of every major programming language, however only few of them provide arbitrary-precision number types (e.g., Lisp, Erlang, Java, Perl). For other languages, third party libraries have been developed to support such formats, such as the GNU Multi-Precision library (GMP) [8] or the Library for Efficient Data types and Algorithms (LEDA) [1]. Note that they explicitly target CPU hardware architectures.

Such computations are more complex when compared to regular hardware-supported 32/64-bit arithmetics. Basic addition/subtraction has a cost of $O(n)$ with n being the length in bits, while multiplication ranges between $O(n^2)$ and the conjectured optimum of $O(n \log(n))$ [5, 7, 10, 18].

To combat these performance issues, our overall goal in this paper is to leverage the capabilities of many-core hardware architectures to speed up arbitrary-precision computations. Specifically, we will make use of Graphics Processing Units (GPUs) by designing suitable data types and parallel algorithms. We present an implementation using a general and widely used GPU framework, the Compute Unified Architecture Framework (CUDA).

2 Related Work

Since General Purpose Computing on Graphics Processing Units (GPGPU) is a relatively new field, the majority of the work on arbitrary-precision arithmetics targets CPUs, which spawned several libraries. We already mentioned LEDA [14] and GMP [6] in the previous section and another established library is ARPREC [3] which itself is based on MPFUN [2], a multiple precision library for Fortran. Although many of them already provide a rich set of different data-types and operations, our goal is to accelerate the underlying computations for the use in time-critical applications. To our knowledge, there is no arbitrary-precision library for GPUs available and we

cover the existing works on fixed-precision arithmetics in the following.

GPU Multiple-Precision library (GPUMP) In 2010 Kaiyong Zhao and Xiaowen Chu created the GPUMP [23], a multiple-precision library for CUDA. GPUMP performs its operations on integer types with an arbitrary but fixed length. The functionality of GPUMP includes operations such as (modular) addition and subtraction, multiplication, division, Montgomery reduction/multiplication, exponentiation as well as comparators. GPUMP applies sequential arithmetic algorithms on pairs of numbers in parallel. It fails if the number grow beyond the predefined length limit and becomes inefficient for small numbers in terms of both computation time and memory usage. Since GPUMP is only applicable on integers with fixed length, its use in areas like geometry is very limited, whereas our work is based on arbitrary-precision integers.

Multi-Precision Floating-Point on GPUs A multiple-precision library for floating-point number types, the CUDA Multi-Precision library (CUMP), was presented by Takatoshi Nakayama and Daisuke Takahashi in 2011 [15]. Additional work has been done by Andrew Thall [22], Mian Lu et al. [13], as well as Mioara Joldes et al. [11].

2.1 Algorithms

In this section, we shortly review relevant parallel algorithms for our setting.

Parallel Algorithms in CUDA The use of parallel primitives on graphics hardware architectures was pioneered by Mark Harris and colleagues. We build on these concepts and refer to them [9, 19] for a detailed overview on the necessary considerations for algorithms to map well to GPUs and CUDA in particular, such as the usage of shared memory buffers, optimal memory access schemes and issues with code path divergence.

Integer Multiplication While integer addition is rather straightforward, their optimal multiplication is still an open problem. We use the standard school method with complexity $O(n^2)$. More advanced approaches, such as the divide-and-conquer approach by Anatolii Karatsuba [12] have lower complexity of $O(3n^{\log_2 3})$, the conjectured optimum of $O(n \log(n))$ is most closely reached by methods employing the Fast Fourier Transformation (FFT) [7, 18]. Such methods either show their advantage only for huge numbers ($>10^3$ decimal digits) or they are hard to efficiently map to graphics hardware. We show that our simple approach still runs significantly faster than current state-of-the-art CPU implementations.

3 Methodology

Arbitrary-precision arithmetics can be performed on various different number types such as integers, rationals or algebraic. The fundamental number type is the unsigned integer type, additional signs can be handled separately. As arbitrary-precision rational numbers are usually composed of a sign and two integers, we target the unsigned integer type in this paper. Our arbitrary-length integer representation format is based on an array of unsigned sub-integers of fixed length, which we denote as *words*. The word length should be chosen to map well to the underlying arithmetic hardware and in the following, we assume that common arithmetic operations (e.g., $+, -, \times$) are natively supported on words. Furthermore, we expect such operations to 'wrap around' in case of an overflow, i.e., all operations are computed modulo the largest representable value of a word plus one. Note that this is the standard behavior for unsigned integers in all common languages. While theoretically unbounded, the amount of available memory will limit the number of words that can be stored and will act as a practical limit on the maximal size of our arbitrary-precision integers, which we will simply denote as *numbers*.

According to standard literature, the many-core processing hardware (i.e., the graphics card in our implementation) is referred to as *device*, while *host* refers to the CPU (plus the standard system memory). Furthermore, the part of the program executed on the device will be referred to as *kernel* [17].

3.1 Parallelization Strategy

While we target graphics hardware in particular, our work generalizes to most common many-core architectures (e.g. Intel Xeon Phi), which exhibit Single Instruction, Multiple Data (SIMD) computation units as their atomic elements. Each unit computes ℓ_{SIMD}-many data elements in parallel in each execution cycle. Our algorithmic design is strongly motivated by the observation that instruction divergences are costly if they happen inside a SIMD unit but incur no additional cost when different SIMD units follow diverging code paths.

A key assumption is that we expect the word count of the numbers, which we operate on, to be larger than the SIMD units' length. For smaller numbers, the parallel extensions of CPUs (e.g., SSE or AVX) can be efficiently used. In our case, we employ a two-level parallelization strategy. First, we distribute each issued computation to one SIMD unit and compute them independently and in parallel. As different computations generally handle input numbers of different lengths and employ different operators ($+, -, \times$), significant instruction divergence is expected between them. All associated downsides are negated since SIMD units act independently from each other on our targeted architectures. Each SIMD unit itself operates on ℓ_{SIMD}-many words in parallel and we rely the provided intra-SIMD synchronization capabilities to handle sequential sections of the algorithms. After a computation is finished, the responsible SIMD unit fetches the next item from the computation pool until its depletion. Note that this approach is only efficient on large data set, where more computations than SIMD

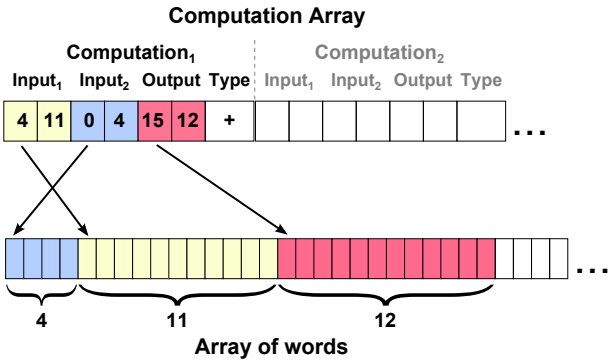

Figure 1: Memory layout of our numbers showing one example computation with two input and one output numbers as well as the operation type $(+,-,\times)$ to be performed. The computation array holds the offset and size of the corresponding words in the global array of words.

units are issued.

3.2 Memory Management

A global array stores all words and the numbers can be identified by their offset into this array and their size (see Figure 1). A separate array holds all the computations that are issued. For each computation, the offset and length are stored for two input and one result number as well as the type of operation to be performed. As the offset and size of the results is generally not known in beforehand, we supply the functionality to reserve additional space in the global array of words. We keep a sophisticated memory allocator [20] as future work and just point to the first free memory location. Note that if host and device manage separate memory spaces (as is the case with graphic cards), data transfers have to be issued.

3.3 Addition

Sequential Addition Before moving to parallel algorithms, we first take a look on how two numbers X and Y are added by a sequential method. As already mentioned, the numbers are composed of several words, which we enumerate as $x_0,\ldots,x_m$ and $y_0,\ldots,y_n$ with x_0 and y_0 holding the Least-Significant-Bits (LSBs). We assume without loss of generality that $n \leq m$ holds for the word counts of the two numbers.

As addition is natively supported on words, we still have to account for potential carries. We will add each word-pair x_i, y_i sequentially and in case of an overflow due to the finite range of the numeric type of the words, we pass a carry c_i to the next addition. This can be achieved with a simple loop over all words. In each iteration, we compute the sum $s_i = x_i + y_i + c_{i-1}$. In case of an overflow, we rely on the wrapping behavior for s_i and issue a carry $c_i = 1$ (instead of $c_i = 0$) for the next addition. Note that for iterations $i > n$, we set $x_i = 0$ and terminate with the last iteration $i = m+1$, where $x_{m+1} = y_{m+1} = 0$.

Parallel Addition with Word Counts $< \ell_{\text{SIMD}}$ While one can trivially add the corresponding words (i.e., each $x_i + y_i$) in a parallel manner, synchronization issues arise from the carry propagation due to its sequential nature. In this section and the next, we describe the addition of numbers whose word count is smaller than the width ℓ_{SIMD} of the SIMD units. We generalize for numbers of arbitrary length afterwards. All $m < \ell_{\text{SIMD}}$ additions of the terms are executed in parallel by a SIMD unit. Each addition potentially issues a carry that has to be propagated to the more significant words.

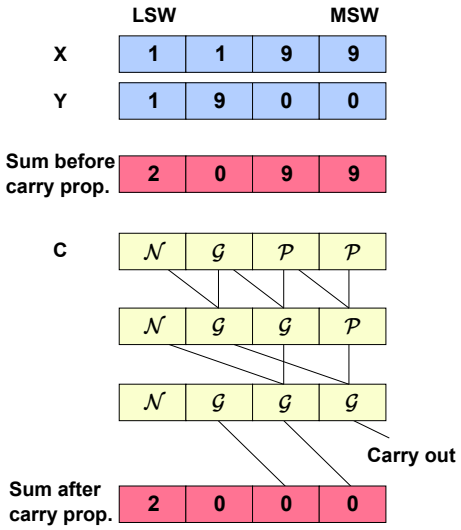

Figure 2: Illustration of the carry propagation with decimal number type as words: We compute the sum of numbers X and Y with lengths $n = 4$ for each word separately and perform the parallel carry propagation using the additional array C with values $\mathcal{G}$ for generation, $\mathcal{P}$ for propagation and $\mathcal{N}$ for no carry. The numbers are ordered from Least-Significant-Word (LSW) on the left to Most-Significant-Word (MSW) on the right. The carries are propagated in $log(n) = 2$ many steps and added to the sums to obtain the final result (bottom).

Parallel Carry Propagation To perform the carry propagation in parallel, we will use the prefix scan algorithm illustrated in Figure 2. Since we perform the addition of X and Y in parallel, we will store these carries in a temporary array C, where each addition $x_i + y_i$ can produce a carry that is stored in c_i. A carry c_{i-1} can only be propagated if $x_i + y_i$ is the maximum value $v_{\max}$ of a single word. A little convenient detail is that carry propagation and generation can not occur at the same time. Even if x_i and y_i are at the highest value, we have $v_{\max} + v_{\max} \mod (v_{\max}+1) = v_{\max} - 1$. Thus it is possible to store both cases (propagation and generation) in the same carry array C. We denote the three distinctive values in this array with $\mathcal{G}$ for generation, $\mathcal{P}$ for propagation and $\mathcal{N}$ for no carry.

Now we have to find a generalized associative operation $\otimes$ that can perform this propagation. Given a pair of

C_i	C_{i-1}	$C_i \otimes C_{i-1}$	C_i	C_{i-1}	$C_i \otimes C_{i-1}$
$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{G}$	$\mathcal{P}$	$\mathcal{G}$
$\mathcal{N}$	$\mathcal{G}$	$\mathcal{N}$	$\mathcal{P}$	$\mathcal{N}$	$\mathcal{N}$
$\mathcal{N}$	$\mathcal{P}$	$\mathcal{N}$	$\mathcal{P}$	$\mathcal{G}$	$\mathcal{G}$
$\mathcal{G}$	$\mathcal{N}$	$\mathcal{G}$	$\mathcal{P}$	$\mathcal{P}$	$\mathcal{P}$
$\mathcal{G}$	$\mathcal{G}$	$\mathcal{G}$			

Table 1: Results for carry propagation function $\otimes$. $\mathcal{G}$ for carry generation, $\mathcal{N}$ for no generation and $\mathcal{P}$ for possible carry propagation

carry values, it computes the resulting behavior and when iteratively applied to all pairs, it correctly adds carries to the relevant values. We list all possible combinations in Table 1. In the case that c_i is already set to $\mathcal{N}$ or $\mathcal{G}$ it does not matter which value c_{i-1} has as the initial value remains the same. In the last three cases, where c_i is set to $\mathcal{P}$, it will inherit the value from c_{i-1}, therefore $\mathcal{P}$ is our identity element.

Parallel Addition with Word Counts $\geq \ell_{\text{SIMD}}$ For longer numbers, we can use a simple loop as described above for a sequential addition algorithm. The only difference is, that we do not iterate over every single word but instead over chunks of ℓ_{SIMD}-many words. Thus, each SIMD unit performs chunk-wise addition sequentially. For additional work parallel approaches are suggested.

3.4 Parallel Multiplication

We employ a parallel version of the school algorithm to multiply two numbers. Again, we assume two numbers X and Y, each composed of m and n words $x_0, \ldots, x_m$ and $y_0, \ldots, y_n$. In contrast to addition, where the upper bound on the length of the result is $\max(m,n)+1$, for multiplication it is $m+n$ and we thus store the result in an array P of words $p_0, \ldots, p_{m+n}$.

Multiplication with Word Counts $\leq \ell_{\text{SIMD}}$ For simplicity, we will first take a look at multiplication of two numbers with a maximum length of l_{SIMD} words each. Again, a single SIMD unit performs this computation, with all others running in parallel. The basic idea is to compute each line of the example on the right sequentially, while the workload of a single line is distributed across the processing elements of a SIMD unit. Note that in this example one word corresponds to one decimal place with 10 possible values. We start by multiplying $x_0, ..., x_m$ with y_0 and writing the result in $p_0, ..., p_{m+1}$. Keep in mind that the first sub-product is of the length $\leq m+1$. Then we compute the second sub-product of $x_0, ..., x_m$ with y_1, which is added to the previous result but shifted by one word to the left, i.e., we add it to $p_1, ..., p_{m+2}$ using our addition algorithm from before. We will continue this until the last sub-product of $x_0, ..., x_m$ with y_n that will reside in the result array in $p_{n-1}, ..., p_{m+n}$.

$$\begin{array}{r} 5\,1\,2 \\ \times\ 1\,2\,8 \\ \hline 4\,0\,9\,6 \\ 1\,0\,2\,4 \\ 5\,1\,2 \\ \hline 6\,5\,5\,3\,6 \end{array}$$

Sub-Products We now take a look at how to perform the j^{th} line of the example. Within the SIMD unit, each element i will perform the multiplication $x_i y_j$. Although we assume that the multiplication of two single words is supported, we cannot directly apply it, as the result will be two words long.

Alternatively, we will split the numbers x_i and y_j in two words of half size each, with x_{high} being the most significant bits and x_{low} being the least significant bits of x_i. Then we will perform four multiplications $x_{\text{high}}y_{\text{high}}$, $x_{\text{high}}y_{\text{low}}$, $x_{\text{low}}y_{\text{high}}$, $x_{\text{low}}y_{\text{low}}$ and store the (shifted) results in the two words p_{high} and p_{low}. Each processing element i adds its result p_{low} to the result array at p_{i+j} in parallel. After that, we perform a carry propagation. Finally, each element adds its result p_{high} to the result array at p_{i+j+1} where another carry propagation is performed.

Multiplication with Word Counts $> \ell_{\text{SIMD}}$ If we only increase the length of Y, the algorithm works just fine, since the limitation given by the SIMD length ℓ_{SIMD} only concerns the length of X. For longer X, we split it into chunks of ℓ_{SIMD} words each and process them sequentially. For the k-th chunk of X and the j-th word of Y, for example, the SIMD unit would compute the product $(x_{k\ell_{\text{SIMD}}}, ..., x_{(k+1)\ell_{\text{SIMD}}-1})\, y_j$.

4 Implementation in CUDA

In our implementation we mapped our parallel algorithms to CUDA [16]. The natively supported integer data type has 32 bits while the length of a SIMD unit – called *warp* – is also 32. Thus, both our word length (in bits) and SIMD length ℓ_{SIMD} are set to 32, making a SIMD-sized chunk 1024 bits long. Carry propagation was performed with intra-warp prefix scans using shared memory, while result space reservation employed global and shared memory atomics. At kernel start, we spawned as many warps as the device supported in blocks of integer size and terminated them only after the computation pool was depleted. No intra-block synchronization primitives were used as we rely on the implicit intra-warp synchronization.

5 Results

For different test cases, we compare the timings of code execution on the CPU with the timings on the GPU on a test system with an Intel Core i7 4700MQ CPU and a nVidia K2100M GPU. We organize the test cases according to computation type $(+, -, \times)$ and the amount of computations that are issued. The length of our numbers are randomly sampled from a normal distribution with mean μ

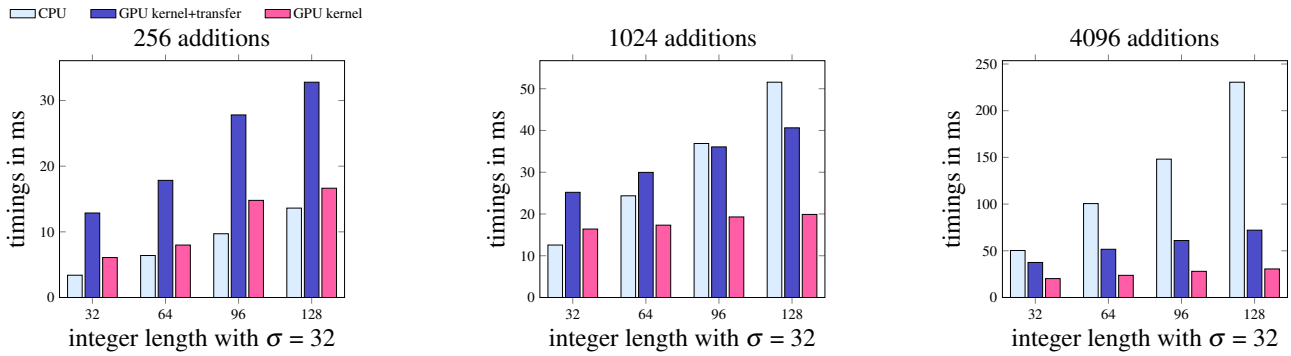

Figure 3: Addition benchmarks. Integer lengths in multiples of 1024 bits with deviation σ are shown on the x-axis. The timing of our GPU implementation is shown with and without data transfer to and from the device on the y-axis. For small computational loads, the GPU is not sufficiently occupied, while for a sufficiently large amount of computations, superior performance is obtained.

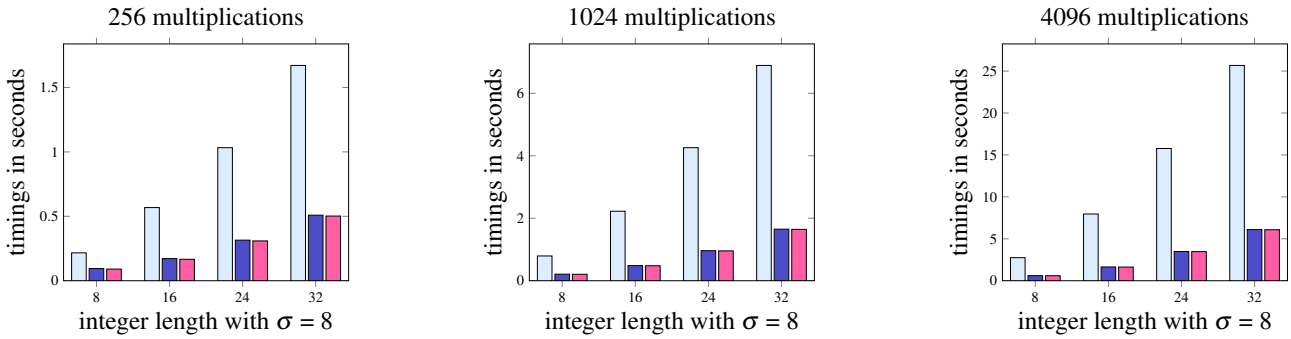

Figure 4: Multiplication benchmarks. Integer lengths in multiples of 1024 bits with deviation σ are shown on the x-axis. Due to the higher complexity of the multiplication, the GPU outperforms the CPU already at 256 multiplications and the advantage grows slightly as we raise the computation count.

and standard deviation σ to reveal possible code divergence issues.

Benchmarks We compute our benchmarks for the operations $+$, $-$ and $\times$. For each of these operations we instruct the GPU and CPU to perform a predefined number of operations. We create a pool of 1024 randomly generated large integers sampled from the standard distribution with μ and σ and each operation is performed on two randomly chosen numbers from this pool. The benchmark results shown in the plots are the averaged timings of 32 executions. The GPU benchmarks are computed with our own framework, while we generate the CPU benchmarks with the state-of-the-art LEDA library [14] as an objective reference and leave optimizations on the CPU as future work.

Addition and Subtraction Comparison The operation types addition and subtraction perform almost identically, since they use the same algorithms and the conclusions in this section hold for both operations. In the first test case with only 256 operations (see Figure 3), the CPU computations are still performed faster compared to our own framework or almost equal if we do not take the data transfers into account, since the computational load is too small and the GPU not fully occupied. As we increase the amount of operations to be performed, we can see an advantage of the GPU – data transfer taken into account – at around 1024 operations. The advantage of the GPU and the performance gap between CPU and GPU increases with every raise of the operation count as we saturate the full compute capabilities of the graphics hardware.

Multiplication Comparison Due to the higher computational complexity of multiplications, we already see the performance advantages of the GPU compared to the CPU at the first test case with 256 operations, although the lengths of the integers are only a fourth of the lengths in the additions and subtractions benchmark. Due to the shorter numbers used for the multiplication benchmarks, the data transfer happens relatively fast, and the two cases with and without transfer behave the same. Already at 256 operations, the GPU performs around three times as fast as the CPU and this performance gap almost remains throughout our test cases, as shown in Figure 4. At 4096 operations the GPU performs about four times as fast as the CPU.

6 Limitations and Future Work

As our work is a pioneering effort into arbitrary-precision integer arithmetics on graphics hardware, there are multiple venues for future work:

Faster Multiplication One could replace our school-method multiplication with the Karatsuba Algorithm [12] for a better computational complexity. Even better performance on longer numbers can be achieved with the use of FFT based methods [21].

Parallelization Strategies For small numbers, a per-thread parallelization can yield better device occupancy for a smaller amount of computations. For huge numbers, a per-block or per-device parallelization can lead to better occupancy as well.

Additional Formats A support for rational numbers as quotients of two integers would add implicit division capabilities. However, an efficient method to compute the greatest common divisor would be needed to reduce the memory requirements. Furthermore, the framework can be extended to support algebraic number formats, which is highly non-trivial due to the conceptual and algorithmic complexities involved.

Additional Operations Although the framework is a proof of concept, it can be extended to make it practically usable. For that it needs to support more mathematical operations than simple arithmetics, such as exponential functions, least common multiple, greatest common divisor, min/max functions and comparators.

7 Conclusion

We presented a method to perform arbitrary-precision integer arithmetics on massively parallel hardware in the form of graphic cards. By employing a two-level parallelization scheme, we ensure minimal code divergence within the SIMD units while still providing effective load balancing across all units. By employing parallel prefix sum computations we allow for an efficient carry propagation and dynamic computation of memory offsets to both read and write integers of arbitrary length. Our CUDA implementation was compared to a state-of-the-art CPU-based libraries and with several benchmarks we showed that method is several times faster for large computation loads.

References

[1] Algorithmic Solutions. *The LEDA User Manual*, version 6.4 edition, July 2012.

[2] David H. Bailey. MPFUN: A portable high performance multiprecision package. Technical report, NASA Ames Research Center, December 1990.

[3] David H. Bailey, Hida Yozo, Xiaoye S. Li, and Brandon Thompson. ARPREC: An arbitrary precision computation package. *Lawrence Berkeley National Laboratory*, 2002.

[4] Joshua Davies. *Implementing SSL/TLS using cryptography and PKI*. John Wiley and Sons, 2011.

[5] Anindya De, Piyush P. Kurur, Chandan Saha, and Ramprasad Saptharishi. Fast integer multiplication using modular arithmetic. *SIAM Journal on Computing*, 42(2):685–699, 2013.

[6] Laurent Fousse, Guillaume Hanrot, Vincent Lefevre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13, 2007.

[7] Martin Fürer. Faster integer multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009.

[8] Torbjörn Granlund. *The GNU Multiple Precision Arithmetic Library*. GMP development team, 6.0.0 edition, March 2014.

[9] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[10] David Harvey, Joris Van Der Hoeven, and Grégoire Lecerf. Even faster integer multiplication. *arXiv preprint arXiv:1407.3360*, 2014.

[11] Mioara Joldes, Valentina Popescu, and Warwick Tucker. Searching for sinks of Henon map using a multiple-precision GPU arithmetic library. *HAL archives*, November 2013. 6p. <hal-00957438>.

[12] Anatolii Alexeevich Karatsuba. The complexity of computations. In *Proceedings of the Steklov Institute of Mathematics – Interperiodica Translation*, volume 211, pages 169–183. Providence, RI: American Mathematical Society, 1995.

[13] Mian Lu, Bingsheng He, and Qiong Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 19–26, New York, NY, USA, 2010. ACM.

[14] Kurt Mehlhorn. *LEDA: a platform for combinatorial and geometric computing*. Cambridge University Press, 1999.

[15] Takatoshi Nakayama and Daisuke Takahashi. Implementation of multipleprecision floating-point arithmetic library for GPU computing. In *Proceedings of the 23rd Int. Conf. on Parallel and Distributed Computing and Systems*, IASTED '11, pages 343–349, 2011.

[16] NVIDIA. CUDA, 2014. `http://www.nvidia.com/cuda/`.

[17] nVidia. CUDA toolkit documentation v6.5. `http://docs.nvidia.com/cuda/index.html`, 2014.

[18] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3-4):281–292, 1971.

[19] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[20] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012.

[21] Daisuke Takahashi. Parallel implementation of multiple-precision arithmetic and 2,576,980,370,000 decimal digits of π calculation. *Parallel Comput.*, 36(8):439–448, August 2010.

[22] Andrew Thall. Extended-precision floating-point numbers for GPU computation. In *ACM SIGGRAPH 2006 Research posters*, page 52. ACM, 2006.

[23] K. Zhao and X. Chu. GPUMP: A multiple-precision integer library for GPUs. In *10th International Conference on Computer and Information Technology*, CIT '10, pages 1164–1168. IEEE, 2010.