Procedia Computer Science
Volume 80, 2016, Pages 154–165
ICCS 2016. The International Conference on Computational Science

Multivariate Polynomial Multiplication on GPU

Diana Andreea Popescu[1]*and Rogelio Tomas Garcia[2]

[1] Computer Laboratory University of Cambridge
diana.popescu@cl.cam.ac.uk
[2] CERN European Organization for Nuclear Research
rogelio.tomas@cern.ch

Abstract
Multivariate polynomial multiplication is a fundamental operation which is used in many scientific domains, for example in the optics code for particle accelerator design at CERN. We present a novel and efficient multivariate polynomial multiplication algorithm for GPUs using floating-point double precision coefficients implemented using the CUDA parallel programming platform. We obtain very good speedups over another multivariate polynomial multiplication library for GPUs (up to 548x), and over the implementation of our algorithm for multi-core machines using OpenMP (up to 7.46x).

Keywords: computer algebra, multivariate polynomial multiplication, GPU, CUDA, particle accelerator design

1 Introduction

Symbolic computation is an area which deals with the study and development of algorithms and software for manipulating mathematical expressions that contain variables. Many scientific applications involve multivariate polynomial multiplications, especially in areas like particle accelerator design [1] and celestial mechanics [2].

We describe where multivariate polynomial multiplication appears in the optics code used for the design of particle accelerators at CERN. A magnetic lattice is an assembly of electromagnets arranged at specific longitudinal positions along the path of the charged particle beam. The magnetic lattice is necessary for focusing the particle beam. The particle trajectory along a path of an accelerator facility is referred to as a *beamline*. Each element on the beamline is defined by a transport matrix. A transport matrix contains various trigonometric functions that are represented using Taylor series expansion up to a set order to have sufficient accuracy of the computations. A *transport map* is a polynomial representation of the transformation done over the particle coordinates from one point to another along the beamline and can be obtained by

*Work partially done while the author was a student at the École Polytechnique Fédérale de Lausanne (EPFL) and an intern at CERN.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.306

multiplying the transport matrix with the vector containing the particle coordinates. *Transport map composition* is based on the mathematical operation of function composition. The functions are multivariate polynomials. Given a beamline with its elements in a sequence, the *transport map construction* refers to the process of applying the transport map composition operation with the first operand being the current element in the beamline and the second operand being the previous computed map. Due to the nature of the transport maps, the transport map composition results in multiplying multivariate polynomials.

In this context, we studied the operation of multivariate polynomial multiplication, for which we propose a novel and efficient algorithm for GPUs implemented using the CUDA parallel programming platform. We evaluate our implementation and compare it with other implementations, obtaining speedups of up to 548x compared with another multivariate polynomial multiplication implementation for GPUs and up to 7.46x compared with an implementation of our algorithm for multi-core machines using OpenMP [3].

This work is organized as follows. Section 2 introduces our multivariate polynomial representation in memory. Section 3 presents our algorithm, while section 4 presents its implementation using the CUDA programming platform. In section 5 we evaluate our implementation and compare it against other implementations. Section 6 describes the related work. We conclude in section 7.

2 Proposed multivariate polynomial representation in memory

An *exponent vector* $\mathbf{e} = (e_1, e_2, ..., e_m) \in \mathbf{N}^m$ defines a monomial $\mathbf{x}^{\mathbf{e}} = x_1^{e_1} \cdot x_2^{e_2} \cdots x_m^{e_m}$. An element in the exponent vector is an exponent of a variable. A *term* is the product of a non-zero coefficient $c \in \mathbf{R}\backslash\{0\}$ and a monomial, i.e. $c \cdot \mathbf{x}^e$. A *multivariate polynomial* f is a finite sum of terms, $f = c_1\mathbf{x}^{\mathbf{e}_1} + c_2\mathbf{x}^{\mathbf{e}_2} + ... + c_t\mathbf{x}^{\mathbf{e}_t}$, with each $c_i \in \mathbf{R}\backslash\{0\}$ and all the $\mathbf{e}_i$'s distinct elements of $\mathbf{N}^m$. We define the *order* of a polynomial as being given by the term or terms with the largest exponents' sum. For example, if we have the polynomial $x^2 \times y + x \times y$, its order is 3, given by the first term.

The two commonly used representations for polynomials in computer algebra systems are the distributed format and the recursive format [4]. In the **distributed format**, polynomials are represented as a list of terms (($\mathbf{e}_1$, c_1), ($\mathbf{e}_2$, c_2),...,($\mathbf{e}_t$, c_t)), each term being represented as a pair of its exponent vector and its coefficient. This representation can be implemented as a list of terms or as a hash table whose elements are terms, the key being the exponent vector and the value being the coefficient. In the case of a GPU implementation using CUDA, the list of terms would be simple arrays, as more complex data structures like hash tables or self balancing binary search trees are usually not implemented on GPUs. A hash table or search tree on GPU would not be very efficient due to the non-linear memory access patterns. In the **recursive format**, a polynomial is considered as an univariate polynomial whose coefficients are polynomials in the remaining variables. This recursive representation could use as a data structure a recursive list or a recursive vector. Due to the memory access patterns for this representation, it is not suited for the GPU memory which requires coalescing memory accesses.

The polynomial representation influences the performance of the polynomial multiplication [4, 5]. We exemplify our proposed representation in memory used for a multivariate polynomial on GPU in Figure 1 for two polynomials, $A = x^2 + y^2 + 2xyz$ and $B = 3z^3 + x^2 + y^2$. We have an array with integer elements for the exponents and an array with double elements for the coefficients. We could have used an array of terms (structures), where a term would have been

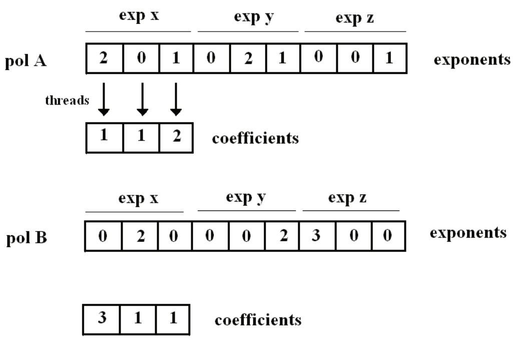


Figure 1: Multivariate polynomial representation for GPU

a structure comprising the exponents and the coefficient. We do not use an array of structures due to the fact that it would have led to uncoalesced memory accesses, as explained in [6]. Uncoalesced memory accesses determine more memory transactions, which in turn affect the performance of the application. Instead we use the structure of arrays approach described in [6]. If we have $NVARS$ number of variables (in Figure 1, $NVARS = 3$, with x, y, z as variables), the array of exponents can be seen as $NVARS$ arrays of exponents, one for each variable. Hence, the structure for representing a polynomial has $NVARS$ integer arrays of exponents and a double array for coefficients. Each term's exponents and coefficient can be found in the two arrays. A warp of threads will access consecutive array elements when reading data from global memory, either the coefficients array or the exponents arrays, as shown in Figure 1.

3 Our Algorithm

Let $A(\mathbf{x}) = \sum_{i=1}^{n_a} a_i \mathbf{x}^{\alpha_i}$ and $B(\mathbf{x}) = \sum_{j=1}^{n_b} b_j \mathbf{x}^{\beta_j}$, where a_i and b_j are numerical coefficients, $\mathbf{x}$ denotes the variables $x_1, x_2, ..., x_m$ and the m-dimensional integer vectors α_i and β_j are the exponents. The product P of A and B is as follows:

$$P(\mathbf{x}) = \sum_{i=1}^{n_a} \sum_{j=1}^{n_b} P_{ij}(\mathbf{x}),$$

where

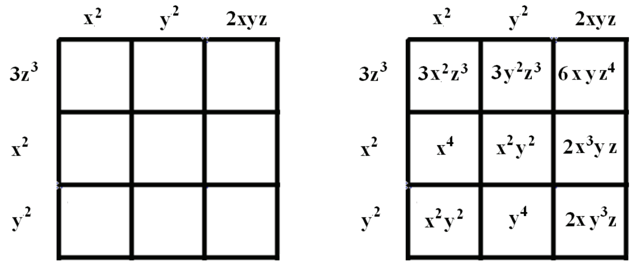
$$P_{ij}(\mathbf{x}) = a_i b_j \mathbf{x}^{\gamma_{ij}}, \gamma_{ij} = \alpha_i + \beta_j,$$

for $i = 1, 2, ..., n_a$, and $j = 1, 2, ..., n_b$.

The algorithm that we propose starts from the naive polynomial multiplication algorithm, which computes all the $n_a n_b$ terms of the product. Due to the fact that the polynomial multiplication exhibits strong data parallelism, it is well adapted for the GPU parallel programming paradigm. The Algorithm 1 has several steps, implemented as distinct kernels and is explained in the next paragraph.

```
Algorithm 1: Algorithm for multivariate polynomial multiplication
  Input: $polynomial_A, polynomial_B$
  Output: $polynomial_{result}$
  $Kernel_1$ (computeResultTerms):
    • each thread computes the product of two terms (one from $polynomial_A$ and one from
      $polynomial_B$), called the result term ;
    • each thread computes a key based on the exponents for its result term;
  $Kernel_2$ (sortByKey): sort the result terms by the computed key;
  $Kernel_3$ (reduceByKey): reduce the terms which have the same key (which means
  adding the terms which have the same exponents for all variables);
  $Kernel_4$ (computeExponents): recover the exponents of the result terms;
  return $polynomial_{result}$
```

The first kernel computes all the terms of the product. The result can be thought of as a matrix of terms as seen in Figure 2. Each thread reads the coefficients and exponents of two terms, computes the product of the coefficients and adds the exponents for each variable. We notice that we can obtain result terms that have the same exponents for all variables. These terms' coefficients have to be added together, but since we use simple arrays for representation, we cannot easily identify them, unless if we perform repeated linear searches on the arrays, and this would be expensive. To solve this issue, each thread also computes a key based on the exponents of the resulted term as follows: for the term x^3yz^2, with variables x, y, z considered in this order, if the exponents have a maximum value of 99, the key will have the value $3 \times 100^2 + 1 \times 100 + 2 = 30102$. The way in which we compute the key is like representing the number 312 (3 is x's exponent, 1 is y's exponent, 2 is z's exponent) in base 100. Thus, in order to compute the key, we first need to choose a maximum exponent, which is similar to choosing a base in which you represent a number. This way of condensing the exponents into a single number, the key, is described in [5]. The second kernel will sort the result terms based on the computed keys, and as such the terms with the same exponents will be placed consecutively in the result array. Using the sorted result, the third kernel will reduce the terms with the same key by adding their coefficients. In the end, in the fourth kernel we recover the exponents from the key in order to provide the multiplication's result to the user in the same format as the input polynomials.



(a) Multiplication Grid (b) Multiplication Result Terms

Figure 2: Multivariate Polynomial Multiplication - Example

The maximum number of variables that an input polynomial can have and the maximum value of an exponent will be limited by the number of bits used to store the key. If the key is stored in a 64-bit integer and the maximum value of an exponent is 999, then a polynomial can have a maximum of 6 variables without the key's value overflowing. Similarly, if the key is stored in a 64-bit integer and the maximum value of an exponent is 99, then a polynomial can have a maximum of 10 variables. Given that we multiply two polynomials, the maximum exponent of the polynomial result will be the sum of the maximum exponents of the two input polynomials. Thus, the maximum exponent of an input polynomial can be 499 and 49 for the cases we presented, in order to avoid the polynomial result having an exponent bigger than the maximum exponent allowed.

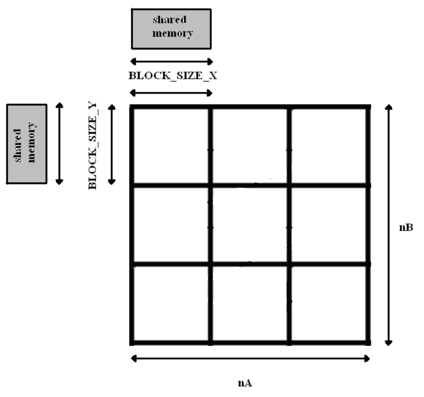
4 Algorithm Implementation

The implementation is done using the CUDA parallel computing platform. The polynomials are represented in memory using the representation described in Section 2. The first kernel maps a thread to a product of two terms, one from polynomial A and one from polynomial B, using a Cartesian `(x,y)` mapping to the automatic variables in CUDA C: `threadIdx.x` is horizontal (it indexes polynomial A arrays) and `threadIdx.y` is vertical (it indexes polynomial B arrays). In Figure 3a we can see the computational grid which has thread blocks of size `BLOCK_SIZE_X x BLOCK_SIZE_Y`. Each thread block copies from the global memory in its shared memory `BLOCK_SIZE_X` elements from the arrays of polynomial A and `BLOCK_SIZE_Y` elements from the arrays of polynomial B. Using shared memory assures much faster access to the data than global memory access, because it is located on chip. Due to the polynomial representation in memory, each thread warp reads contiguous data from global memory, achieving memory coalescing. The data is used to compute the coefficients, the exponents and the key based on exponents for the new terms. All these computations are done using shared memory and registers. The results are written afterwards to global memory.

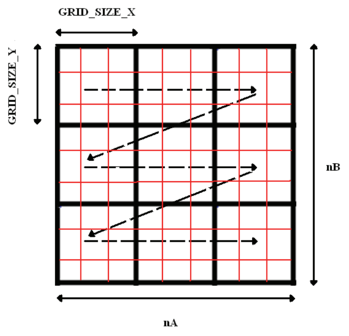
The implementation can multiply polynomials of arbitrary sizes. As we can see from Figure 3b, in case we have a grid smaller than the size of the input polynomials, each thread will compute more result terms: the computational grid moves in a zig-zag pattern over the polynomial result matrix. Each thread keeps two indices, `tbx` and `tby`, which are initialized to `threadIdx.x` and `threadIdx.y`, respectively. In order to move the computational grid, we add to the `tbx` index the size of the computational grid on the `x` coordinate. When we reach the limit given by the size of polynomial A, we reset it to `threadIdx.x`, but we increase the `tby` index by adding the size of the computational grid on the `y` coordinate, after which we will again increase the `tbx` index. The `tby` index increases until we reach the size of the polynomial B. In this way, we obtain the zig-zag pattern used to compute all the terms of the multiplication result.

In order to deal with input polynomials sizes which are not divisible by the thread block sizes, we use boundary checks with if-statements (the statement is in fact a while statement, because a single thread computes several result terms if the grid size is insufficient to cover the input polynomials). The hardware is optimized for the case when all threads in a warp take the same branch of an if-statement [7]. For the thread blocks which cover the margins, some threads will be deactivated by the checks, and only here we can have control flow divergence for some warps.

The second and third kernel are the functions *sort_by_key* and *reduce_by_key* provided by the Thrust parallel template library [8, 9] from NVIDIA. These functions perform the sorting and reducing operations on the GPU. The sort algorithm used by the library is radix sort for



(a) Computational Grid



(b) Arbitrary long polynomials with
nA = size of polynomial A
nB = size of polynomial B

Figure 3: Multivariate Polynomial Multiplication on GPU

GPU architectures and it is described in [10]. The reduce implementation is based on the implementation provided in the CUDA SDK Samples [11].

The fourth kernel performs the inverse transformation on the computed key, obtaining a vector of exponents from a key. The mapping for a thread is done on the `x` coordinate: each thread `threadIdx.x` reads a key from the array and it extracts from it a vector of exponents. The kernel works for arbitrary sized arrays of keys. In the same way as the first kernel, but here we have only one dimension, we add to the index `tbx` (which was initialized to `threadIdx.x`) the grid size, until we reach the end of the input array. In this way, we move the computational grid along the input array of keys.

For the input and output data on CPU, we allocate pinned (non-pageable) memory on the host for a faster data transfer between host and device.

The polynomials used in the particle accelerator software had to be truncated to a certain order. To achieve this, we modified our algorithm in the following way. The first kernel computes an additional array which stores the differences *order - exponents_sum* for each result term. The second kernel is now represented by the kernel that removes the result terms which have order bigger than the required one. We use the *remove_if* function from the Thrust library [8]. The rest of the algorithm is the same as Algorithm 1.

If the user wants to multiply several polynomials, this can be done sequentially. The first two polynomials will be transferred to the GPU and they will be multiplied using our algorithm. However, instead of transferring the result to the CPU, we would transfer the next polynomial in the product to the GPU. Next, we multiply the result with the transferred polynomial. This operations can be repeated for any number of polynomials, as long as there is still global memory available on the GPU.

5 Experimental Evaluation

Experimental Setup. We evaluate our algorithm and other algorithms on an NVIDIA Tesla K20c with CUDA Driver Version 7.5, CUDA Runtime Version 7.0 and CUDA Capability 3.5.

The NVIDIA card has 13 streaming multiprocessors with 192 CUDA cores per multiprocessor and 5 GB of global memory and is installed on an Intel Sandy Bridge Xeon CPU E5-2670, 2.60GHz, 64GB of RAM, with 8 cores, for which we enabled hyper-threading.

Performance. We evaluate our implementation by measuring the execution times of multiplying two multivariate polynomials of different sizes, from 100 terms to 9000 terms for each polynomial. We also vary the number of variables that each polynomial has, from 1 to 10 variables. The polynomials' exponents have a value of maximum 49. We ran the multiplication 5 times for each test on randomly generated polynomials and we report the average of the obtained execution times. The reported execution times include the input and output data transfer time to/from the host to the GPU. We used a thread block size of 16. The results can be seen in Figure 4. It can be seen that our algorithm is efficient, the execution time when multiplying polynomials of sizes 8000 and 9000 with 10 variables being less than 625 milliseconds. We observe that the execution time increases almost linearly with the number of terms of the polynomials that are multiplied. The execution time also depends on the number of variables the polynomials have, increasing with the number of variables. This is due to several reasons: increased data transfer times to/from the GPU due to a larger array of exponents, more computational operations done to compute the key from the exponents (in kernel 1) and for extracting the exponents from the key (in kernel 4). We also analysed the individual running times of the 4 kernels. The most time is spent in the second kernel (the sorting phase), followed by the third kernel (the reduction phase), the fourth kernel and the first kernel.

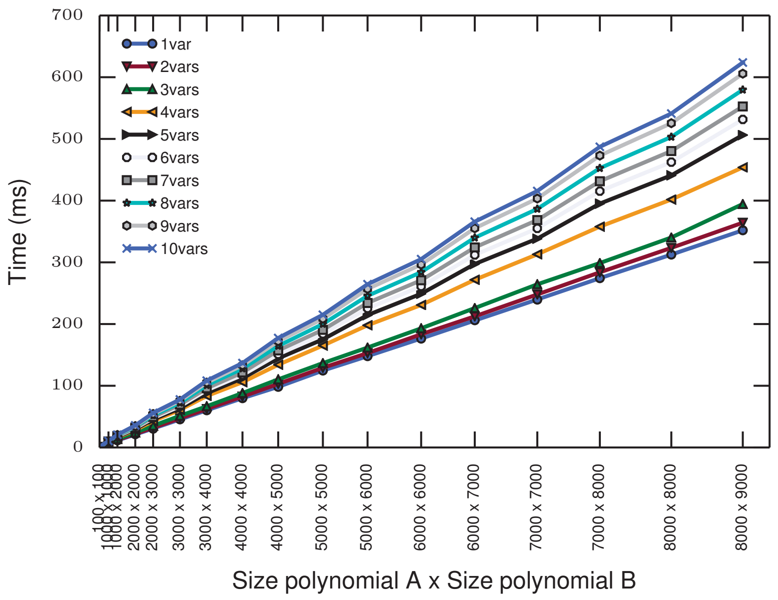
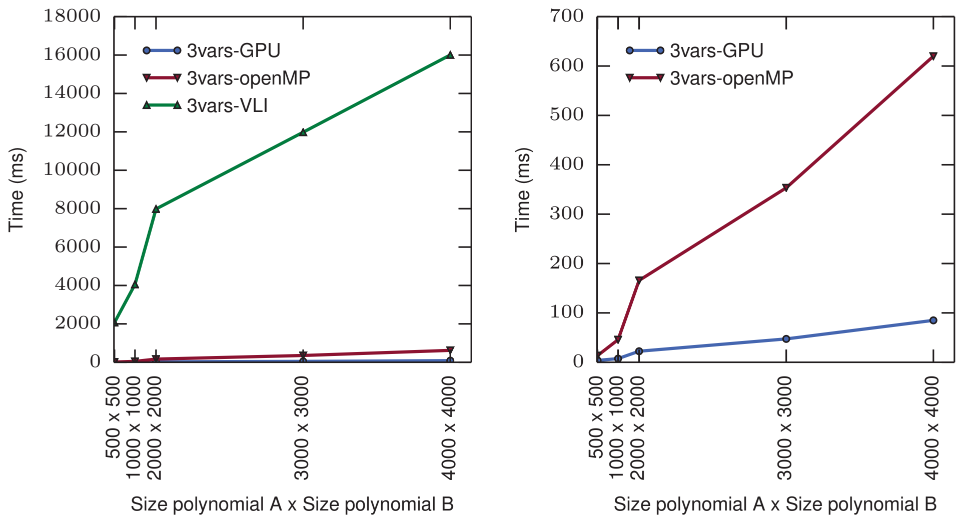


Figure 4: Execution time of our algorithm on the NVIDIA Tesla K20c

Comparison with other algorithms. We compare our algorithm with the VLI Library for High Precision Integer and Polynomial Arithmetic [12] and with our algorithm implemented for multi-core machines. We ran the multiplication 5 times for each test and we report the average of the obtained execution times. The reported execution times include the input and output data transfer time to/from the host to the GPU. VLI implements the multivariate polynomial multiplication for GPUs and can multiply multivariate polynomials that have up to 4 variables. When running the VLI polynomial multiplication, we used integer coefficients of 128 bits, because we did not manage to use coefficients with a smaller number of bits due to the library's implementation. We measured the running time of the polynomial multiplication function of VLI. The VLI's multivariate polynomial multiplication algorithm first determines the number of terms of the product polynomial that will contribute to the coefficient of a result term of the final product. A thread will be responsible for computing the coefficient of a result term. The coefficients are assembled in groups of 32 and are scheduled on warps according to an execution plan depending on the number of terms that contribute to the coefficients. In contrast, our algorithm does not split the work according to a criterion, and a thread performs a single multiplication of the coefficients of two input terms. Also, our implementation can multiply polynomials that have more than 4 variables and that have floating-point coefficients.

We also implemented our algorithm for multi-core machines using OpenMP [3] for parallelization. The first and fourth kernel can be easily transposed to loops parallelized with OpenMP. The second and third kernel use the same Thrust library [9] functions *sort_by_key* and *reduce_by_key*, because the Thrust library also supports other parallel backends. In our case we use OpenMP as the backend and we set the number of threads that these functions will use to 16 by using the OMP_NUM_THREADS environment variable.



(a) Execution times for all 3 algorithms

(b) Execution times for our algorithm for GPUs and on multi-cores (zoomed view of Figure 5a)

Figure 5: Comparison of our algorithm for GPUs (3vars-GPU), the VLI library's multivariate implementation (3vars-VLI) and our algorithm implemented for multi-core machines (3vars-OpenMP) using polynomials with 3 variables and maximum exponent 14.

We compare the 3 implementations using randomly generated polynomials of different sizes with 3 variables and each variable has a maximum exponent of 14, similarly to the characteristics of the "dense" polynomials used in evaluating VLI [12]. We report speedups between 146x and 548x in comparison to the VLI polynomial multiplication and between 2.3x and 7.46x in comparison to the multi-core implementation of our algorithm using 16 threads depending on the polynomials' sizes, as can be seen from Figure 5a and Figure 5b.

We also evaluate the 3 algorithms using polynomials of 4096 terms with 3 variables, but this time we vary the maximum exponent of the polynomials, as in [12]. Since the maximum exponent of the variables is between 2 and 14, the polynomials have a large number of terms that share the same exponents for their variables and could be added together. Because of this, the VLI algorithm has small running times when the exponents are small, since it forms a small number of coefficient groups. However, as the maximum exponent increases, the number of distinct terms in the input polynomial also increases, which leads the VLI algorithm to form a larger number of coefficient groups, influencing the overall execution time.

Our algorithm's running time is not influenced by the maximum exponent of the polynomials as is the VLI algorithm, and as such the execution time on these tests ranges from 83.7 ms for maximum exponent 2 to 87.2 ms for maximum exponent 14. This is partly due to the fact that, when the maximum exponent is bigger, there are more distinct result terms in the polynomial result for which the fourth kernel extracts the exponents from the key. The situation is similar for the OpenMP implementation, with execution times of 702 ms for maximum exponent 2 to 811.2 ms for maximum exponent 14.

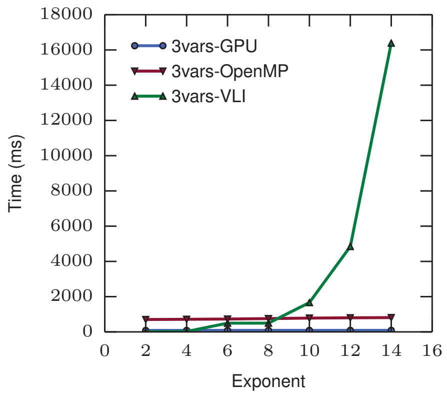


Figure 6: Comparison of our algorithm for GPUs (3vars-GPU), the VLI library's multivariate polynomial multiplication (3vars-VLI) and our algorithm implemented for multi-core machines (3vars-OpenMP) using polynomials of 4096 terms with 3 variables.

6 Related work

The Karatsuba algorithm [13] for multiplication of polynomials with n terms (where n is a power of 2) has a lower complexity, $O(n^{\log_2 3})$, than the naive algorithm, $O(n^2)$. This algorithm is better suited for dense polynomials multiplication. Generalizations of the Karatsuba algorithm for polynomials with arbitrary number of terms are presented in [14].

Several works study the **univariate polynomial multiplication** on multi-cores and GPUs [15], [16], [17], [18]. The algorithm from [15] can multiply univariate polynomials with integer coefficients on GPU and it uses a generalization of discrete Fourier transform to finite fields. Maza *et al.* [16] implemented a FFT based univariate polynomial multiplication over finite fields for dense univariate polynomials on GPU. The algorithm in [17] is for dense univariate polynomials of sizes up to 2^{12} and it has 2 phases. In the first phase, all the multiplications of coefficients are done in parallel on the GPU. The second phase is represented by the addition of the terms which have the same exponent in parallel. Due to the fact that the algorithm deals with the simple case of univariate dense polynomials and due to the distribution of the terms to the threads, the addition can be performed without prior sorting, since the terms that have the same exponent are already grouped together. In contrast, our work deals with multivariate polynomials. Monagan *et al.* [18] present an algorithm for multiplying sparse distributed polynomials on multiprocessors. In the algorithm, each core has a heap of pointers which is used to multiply terms from the polynomials, but these local heaps will have to be merged in order to get the final result. To solve this, a global heap is used and, periodically, each thread accesses this global heap (if it is available and not locked by another thread) in order to merge its local terms with the global heap.

Multivariate polynomial multiplication has been studied in [19], [20], [21], [22] and [12] (which was described in Section 5). These works focus on multi-core machines, only [22] and [12] mentioning algorithms for GPU. In [19], the multiplication of sparse multivariate polynomials using the recursive representation is studied. The use of recursive format has drawbacks, amongst which are the potential load imbalance and the allocation of many small objects. Thus, their implementation uses a work-stealing technique to solve the load imbalance and a custom memory allocator. High-performance techniques for dense multivariate polynomial multiplication targeting multi-cores are presented in [20]. Dense representations of polynomials allow the use of multiplication algorithms based on Fast Fourier Transform (FFT). Methods to reduce multivariate multiplication to balanced bivariate multiplication based on 2-D FFT are provided. The authors introduce two fundamental techniques, called contraction and extension. The first one allows to turn a n-variate multiplication (for $n > 2$) into a bivariate multiplication. The second technique, the extension, transforms the univariate multiplication to bivariate multiplication. Biscani [21] focuses on the design of a cache-friendly hash table implementation that stores polynomial terms for a polynomial in distributed format.

The algorithm in [22] for multiplying sparse multivariate polynomials in distributed format is designed to work on shared memory multi-core computers, computer clusters and GPUs. The first step of the algorithm splits the work between threads to avoid any communication between them during the computational task. Since the possible integer vectors exponents of polynomials are in a vector interval, this interval may be split into subintervals which are processed by the different threads. On GPUs, each interval is processed by a thread block. The second step is computing the resulting terms using a parallel loop over all the subintervals. Because the processing of the intervals can take different times, the work can be balanced between the cores by using a number of intervals greater than the number of cores. Each thread computes the summation of its own terms. The third step is represented by a concatenation of the results from each thread to obtain the form of the polynomial product. In contrast, our algorithm sorts the resulting terms to find the terms that should be added and performs a reduction on the intermediate result terms.

7 Conclusion and future work

We presented a novel and efficient multivariate polynomial multiplication algorithm for GPUs using floating-point double precision coefficients. We implemented our algorithm using the CUDA parallel programming platform. We obtain very good speedups over the VLI multivariate polynomial multiplication library for GPUs (up to 548x) and over the implementation of our algorithm for multi-core machines using OpenMP (up to 7.46x). As future work, we would like to explore the possibility of computing the key in a different way in order to allow the multiplication of polynomials with bigger exponents and more variables. Alternatively, we want to explore using very large integers for storing the key (the VLI Library for Integer Arithmetic [12] supports integers with 512 bits). Another direction for future work is extending the algorithm to accommodate polynomials that do not fit in the global memory. Also, we would like to extend the algorithm to run on multiple GPUs.

Acknowledgments

The authors would like to thank Roger Hersch for helpful discussions. This work was supported by an EPFL Excellence Scholarship, a CERN Technical Student studentship and by the EU FP7 Marie Curie ITN METRICS project (grant agreement nr. 607728).

References

[1] D. A. Popescu. Parallel Computing Methods For Particle Accelerator Design. Master's thesis, Ecole Polytechnique Fédérale de Lausanne and CERN, July 2013. Report CERN-THESIS-2013-121, https://cds.cern.ch/record/1598904.

[2] M. Gastineau and J. Laskar. *Computational Science – ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006. Proceedings, Part II*, chapter Development of TRIP: Fast Sparse Multivariate Polynomial Multiplication Using Burst Tries, pages 446–453. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[3] The OpenMP Board. The OpenMP API specification for parallel programming. http://openmp.org/, 2016.

[4] M. Monagan and R. Pearce. Sparse Polynomial Multiplication and Division in Maple 14. *ACM Commun. Comput. Algebra*, 44(3/4):205–209, January 2011.

[5] R. Fateman. Comparing the speed of programs for sparse polynomial multiplication. *SIGSAM Bulletin*, 37(1):4–15, March 2003.

[6] J. Luitjens. Global Memory Usage and Strategy. Technical report, NVIDIA Corporation, 2011.

[7] J. Balfour. CUDA Threads and Atomics. Technical report, NVIDIA Research, April 2011.

[8] N. Bell and J. Hoberock. *GPU Computing Gems Jade Edition*, chapter Thrust - A Productivity-Oriented Library for CUDA, pages 359–373. Morgan Kaufmann, June 2011.

[9] N. Bell and J. Hoberock. Thrust - a parallel algorithms library. http://thrust.github.io/, January 2016.

[10] D. G. Merrill and A. S. Grimshaw. Revisiting sorting for GPGPU stream architectures. In *PACT'10 Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 545–546, 2010.

[11] M. Harris. Optimizing parallel reduction in CUDA. Technical report, NVIDIA Corporation, 2007.

[12] T. Ewart, A. Hehn, and M. Troyer. *Supercomputing: 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings*, chapter VLI – A Library for High Precision Integer and Polynomial Arithmetic, pages 267–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[13] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, 7:595, January 1963.

[14] A. Weimerskirch and C. Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations. Cryptology ePrint Archive, Report 2006/224, 2006. http://eprint.iacr.org/.

[15] P. Emeliyanenko. Efficient multiplication of polynomials on graphics hardware. In *Advanced Parallel Processing Technologies, 8th International Symposium, Proceedings*, volume 5737 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2009.

[16] M. M. Maza and W. Pan. Fast polynomial multiplication on a GPU. *Journal of Physics: Conference Series*, 256(1):012009, 2010.

[17] S. A. Haque and M. M. Maza. Plain polynomial arithmetic on GPU. *Journal of Physics: Conference Series*, 385(9):385–394, 2012.

[18] M. Monagan and R. Pearce. Parallel sparse polynomial multiplication using heaps. In *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, ISSAC 2009, pages 263–270, New York, USA, 2009. ACM.

[19] M. Gastineau. Parallel operations of sparse polynomials on multicores: I. multiplication and poisson bracket. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 44–52, New York, NY, USA, 2010. ACM.

[20] M. M. Maza and Y. Xie. Balanced dense polynomial multiplication on multi-cores. In *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, pages 1–9, Dec 2009.

[21] F. Biscani. Parallel sparse polynomial multiplication on modern hardware architectures. In *Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation*, ISSAC 2012, pages 83–90, New York, NY, USA, 2012. ACM.

[22] M. Gastineau and J. Laskar. Highly Scalable Multiplication for Distributed Sparse Multivariate Polynomials on Many-Core Systems. In *Computer Algebra in Scientific Computing*, volume 8136 of *Lecture Notes in Computer Science*, pages 100–115. Springer International Publishing, 2013.