DOI: 10.14529/jsfi250202

GPU Implementation of Zippel Method for Feynman Integral Reconstruction

Alexander V. Smirnov[1,2], *Boris I. Rozhnov*[1,2], *Vadim V. Voevodin*[1,2]

The Zippel algorithm performs a rational reconstruction of multivariate polynomials and aims specifically at the sparse case, where other approaches, such as iterative Newton and Thiele reconstructions, have a significantly higher complexity. It is applied in different fields of science, lately becoming an important step in Feynman integral reduction in elementary particle physics within the modular approach to reduction. For some cases with multiple variables the Zippel reconstruction might become a bottleneck for the whole evaluation so that different optimizations are required. In this paper, we describe how we ported the classical Zippel algorithm for polynomials together with its balanced version for rational functions to graphical processor units (GPUs), as well as carried out its performance evaluation on several types of GPUs. According to our information, this is the first publically available implementation of this algorithm on GPUs, and the results show speedup up to 14.5 times compared to CPU-based version.

Keywords: rational reconstruction, Zippel algorithm, Feynman integrals, GPU.

Introduction

The main motivation for this paper lies in the field of elementary particle physics, namely Feynman integral reduction [4], one of the key steps of evaluation of Feynman integrals. However, a Feynman integral reduction technically means solving a huge sparse linear system with coefficients being rational functions of multiple variables; hence, the scope of the paper and possible applications are much more broad. For example, the algorithm is used internally in computer algebra systems such as `Wolfram Mathematica`, `Maple`, `SageMath`.

The classical approach to Feynman integral reduction was to solve the system directly on a large enough server [1, 10–12, 14, 15, 17–19, 23–25], but with increasing complexity and the availability of supercomputer infrastructure, new methods were required. Therefore, an approach for a rational reconstruction of functions was proposed, which treats the unknown coefficients as black-box rational functions of multiple variables that need to be reconstructed afterwards [2, 5, 9, 13, 16, 20–22]. Within this "modular" approach the reduction is first performed multiple times with fixed values of variables in modular fields over large prime numbers (fitting into 2^{64} to utilize machine-size arithmetics), and then the functions are reconstructed. One should not confuse reconstruction and interpolation: with the values of a rational function fixed in multiple points, one surely has an unlimited number of rational functions with such properties, but the reconstruction methods aim to find the "true" function which is the most "simple" one satisfying such conditions and that *all* following probes (evaluations of the function at other points) should also match the guessed function.

The reconstruction method has a long history starting with Newton interpolation reconstructing a polynomial of one variable.

$$\begin{aligned} f_N(x) &= \mathrm{Newton}_x[f(x), N] \\ &\equiv a_1 + (x - x_1)\Big[a_2 + (x - x_2)\left[a_3 + (x - x_3)\left[a_4 + \ldots\right]\right]\Big]. \end{aligned} \tag{1}$$

[1] Lomonosov Moscow State University, Moscow, Russian Federation

[2] Moscow Center for Fundamental and Applied Mathematics, Moscow, Russian Federation

For multivariate polynomial reconstruction one can proceed variable by variable, i.e. when reconstructing from n variables to $n+1$ variables one takes a needed number of reconstructed functions $f(x_1, x_2, \ldots, x_n, x_{n+1,i})$ for different values of $x_{n+1,i}$ and runs the univariate Newton reconstruction.

For univariate rational functions there is the Thiele reconstruction

$$\begin{aligned} f_T(x) &= \mathrm{Thiele}_x[f(x), T] \\ &\equiv b_0 + (x - x_1)\left[b_1 + (x - x_2)\left[b_2 + (x - x_3)\left[b_4 + \ldots\right]^{-1}\right]^{-1}\right]^{-1}. \end{aligned} \tag{2}$$

The situation becomes more complicated with multivariate rational functions, since the recursive Thiele formula is a combination of continued fractions and is technically too complex to be evaluated in real examples. Therefore, for multivariate rational functions, other methods are used, one of those being the balanced Newton reconstruction proposed in [2] and the balanced Zippel reconstruction proposed in [26] for sparse multivariate functions.

The balanced Zippel reconstruction of rational functions is the main subject of this paper. It is based on the Zippel reconstruction of polynomials suggested initially in [28] and developed in [3, 8]. The Zippel method is an approach with the lowest complexity for the sparse polynomial reconstruction. It is already widely applied in programs for Feynman integral reduction using the modular approach followed by reconstruction. However, in some cases, the Zippel reconstruction itself might become a bottleneck for the whole approach due to its quadratic complexity by the number of terms in the skeleton polynomial (see next section for details) which might be measured in millions. Hence, any possible optimization of the Zippel algorithm is needed.

There is a number of public Zippel algorithm implementations, mostly in some open-source repositories. Such programs as `Kira` or `FIRE` use the Zippel algorithm as a step in the reduction of Feynman integrals.

The main contribution of this paper is an implementation of the Zippel algorithm on GPUs (Graphical Processing Units). According to our information, this is the first such implementation of this algorithm on GPUs. We currently publish a description and evaluation results, while the code is in a private repository related to Feynman integrals, but we intend to make it public as a part of the new `FIRE` version this year. We are also considering a possible separation of the modular GPU and reconstruction part providing it as a separate library.

The rest of the paper is organized as follows. Section 1 describes the original Zippel algorithm. Section 2 is devoted to our implementation of this algorithm on GPUs. Section 3 provides evaluation results and benchmarks.

1. Description of Zippel Method

The Zippel method is a reconstruction procedure from a polynomial with $k-1$ variables to a polynomial with k variables. The traditional approach for dense polynomials (meaning that if one considers all possible monomials of a given degree most of the coefficients are non-zero) is to proceed with a Newton reconstruction formula. However, for sparse polynomials, this is quite inefficient because it requires a large number of probes for the reconstruction.

Let us remind the method taking a few formulas from our paper [26]. Let us suppose that we have already reconstructed a polynomial $f(x_1, \ldots, x_{k-1}, c_0)$, where c_0 is some constant value of x_k. There might be more variables, but they should be fixed at this point. We aim to run the

Newton reconstruction for x_k, so we need similar reconstructed polynomials for other values of x_k.

Let us suppose we take another value c_i of x_k. We consider the existing polynomial $f(x_1, \ldots, x_{k-1}, c_0, \ldots)$ as a skeleton, take all its non-zero monomials and assume that the set of nonzero monomials will remain the same for the yet unknown $f(x_1, \ldots, x_{k-1}, c_i, \ldots)$ so that it can have the following form:

$$f(x_1, \ldots, x_{k-1}, c_i, \ldots) = a_1 \cdot x_1^{p_{1,1}} \ldots {x_{k-1}}^{p_{k-1,1}} + \ldots + a_t \cdot x_1^{p_{1,t}} \ldots {x_{k-1}}^{p_{k-1,t}} \tag{3}$$

for some t, where a_i are unknown constant coefficients and p are exponents taken from the skeleton.

This is a linear system for a_i and hence knowing the values of $f(x_1, \ldots, x_{k-1}, c_i, \ldots)$ for t different sets of $\{x_1, \ldots, x_{k-1}\}$ (sampling points or probes) we can solve the system. This, however, has a complexity $O(t^3)$, thus the Zippel algorithm for polynomials suggests a specific set of sampling points, i.e. $y_1, ...y_{k-1}, y_1^2, ...y_{k-1}^2, \ldots, y_1^t, ...y_{k-1}^t$. In this case, the system turns into a Vandermonde system which can be solved with complexity $O(t^2)$. The Zippel method consists of solving this system leading to the knowledge of $f(x_1, \ldots, x_{k-1}, c_i, \ldots)$ for different i, followed by univariate Newton reconstruction in x_k to obtain $f(x_1, \ldots, x_{k-1}, x_k, \ldots)$.

For the tasks described one needs to reconstruct rational functions of multiple variables, but it is possible to adapt the Zippel approach, for example, as the balanced Zippel method described in [26].

It is also important for the proper reconstruction order to perform it with the use of modular arithmetic over large prime numbers. This prevents coefficient growth and decreases the number of needed sampling points, and the final reconstruction to rational numbers is performed as a final step. Thus, to proceed with the algorithms, one needs an efficient library working with modular polynomials of multiple variables. There are few such libraries, with `FLINT` [6, 7] being one of the most efficient.

2. Proposed GPU Implementation of the Zippel Method

Most nodes of modern supercomputers and clusters are now equipped with dedicated GPUs that enable parallel execution of mathematical operations on large datasets in a multithreaded mode. When implemented correctly, they can significantly reduce the computation time during the program execution.

Modern GPUs are capable of handling thousands of threads simultaneously, offering excellent horizontal scalability, and their architecture is optimized for the simultaneous execution of uniform operations – a critical feature for the scientific computing tasks performed by the `FIRE` program.

For the case of multiple variables the Zippel reconstruction step sometimes becomes a bottleneck for the whole reduction process, hence implementing it on a GPU seemed an important task.

2.1. Modular Integer Operations on GPUs

There are not many ready-to-use solutions for modular arithmetic on GPU. Only after implementing our solution we found a `CUMODP` (CUDA Modular Polynomial Library) library.

A further investigation showed that this library uses 32-bit modular integers so is not directly ready for our case, which requires 64-bit operations.

Thus, during the development of the GPU migration solution we tested several approaches:

- standard modular arithmetic operators provided by the `nvcc` compiler using the `uint128_t` data type;
- an algorithm leveraging properties of modular arithmetic with the `uint64_t` data type with special GPU intrinsics;
- a ported version from `CUMODP` to the 64-bit case;
- and GPU-ported basic functions from the `FLINT` library (which is used in the original `FIRE` application).

Let us describe those variants in detail and compare their efficiency on a multiplication modular some large prime n fitting into `uint64_t`.

Variant 1 is the most straightforward using the `uint128_t` type, here we completely rely on the compiler:

```
// return (a * b) mod n
__inline__ __device__
unsigned long long mul_mod(
    unsigned long long a,
    unsigned long long b,
    unsigned long long n)
 {
    return (static_cast<__uint128_t>(a) * b ) % n;
 }
```

A better approach (number 2) is to avoid the `uint128_t` type stating in `uint64_t` and using special CUDA intrinsic, `__umul64hi(a, b)` which takes two 64-bit numbers and returns the high part of its multiplication as a result. This approach lets one avoid using the `uint128_t` type. The code can be implemented in the following way:

```
__inline__ __device__
unsigned long long mul_mod(
    unsigned long long a,
    unsigned long long b,
    unsigned long long n)
{
    // Calculate the most significant 64 bits
    // of the product of the two 64 unsigned bit integers
    unsigned long long hi = __umul64hi(a, b);
    unsigned long long lo = a * b;
    hi %= n;
    lo %= n;
    // pow = 2^64 mod n
    unsigned long long pow = (1ULL << 63) % n;
    pow = ( pow << 1 ) % n;
    hi = ( hi * pow ) % n;

    return (hi + lo) % n;
}
```

Variant 3 is based on CUMODP and uses a fast modular multiplication algorithm based on floating-point arithmetic. It has several significant limitations, the main one being its dependence on floating-point precision – the mantissa must be large enough to hold the full range of values without rounding errors. In this example, it has been extended to support larger numbers by using quadruple-precision floating-point numbers. The code looks the following way:

```
__inline__ __device__
int64_t mul_mod1(int64_t a, int64_t b, int64_t n)
{
    long double ninv = 1.0L / (long double) n;
    int64_t q = (int64_t)((((long double)a) * ((long double)b)) * ninv);
    int64_t res = a * b - q * n;
    res += (res >> 63) & n;
    res -= n;
    res += (res >> 63) & n;

    return res;
}
```

The FLINT library is open-source, written in C, and optimized for 32/64-bit arithmetic, which allows its code to be ported to CUDA with minimal modifications. However, the implementation is large enough due to many functions that are required. The top-layer function for multiplication looks as follows:

```
__device__
mp_limb_t nmod_mul_d(mp_limb_t a, mp_limb_t b, nmod_t mod)
{
   b <<= mod.norm;
   mp_limb_t res;
   do {
       mp_limb_t q0xx, q1xx, rxx, p_hixx, p_loxx;
       mp_limb_t nxx, ninvxx;
       unsigned int normxx;
       ninvxx = (mod).ninv;
       normxx = (mod).norm;
       nxx = (mod).n << normxx;
       umul_ppmm_d(p_hixx, p_loxx, (a), (b));
       umul_ppmm_d(q1xx, q0xx, ninvxx, p_hixx);
       add_ssaaaa_d(q1xx, q0xx, q1xx, q0xx, p_hixx, p_loxx);
       rxx = (p_loxx - (q1xx + 1) * nxx);
       if (rxx > q0xx) rxx += nxx;
       rxx = (rxx < nxx ? rxx : rxx - nxx) >> normxx;
       (res) = rxx;
   } while (0);

   return res;
}
```

We ran a benchmark for all variants and got the following results. The multiplication was performed modulo two numbers located in global memory. The tests were performed on Radeon RX 560X (163 GFlop/s). The test involved 1 core and 1 thread.

Table 1. Results of additional fuzzing tests
for modular multiplication (10 000 000 iterations)

Algorithm	Time (sec)
`uint128_t`	140.13
`uint64_t`	24.67
`CUDMODP*`	9.96
`FLINT`	1.48

As we can see from Tab. 1, both native implementations are quite inefficient. The reason is that the GPUs themselves are 32-bit machines, so even the 64-bit operations are supported but are not natural for the cores implemented as consequent 32-bit operations. The `CUMODP` implementation also has problems due to a division in the code. The `FLINT` version is an obvious winner.

Thus, we decided to port the `FLINT` multiplication to GPUs using all its powerful features including the precalculated inversion letting one to avoid the division operations. The following functions were ported for modular addition, multiplication, exponentiation, inversion, and remainder operations for 128- and 192-bit composite integers: `nmod_mul`, `add_ssaaaa`, `umul_ppmm`, `NMOD_RED2`, `NMOD_RED3`, `n_gcdinv`, `nmod_pow`, `n_invmod`, `nmod_inv`. We are considering to later provide the ported code as a standalone library.

2.2. Zippel Algorithm Implementation Details

The next step was to benchmark the CPU implementation of the Zippel algorithm in order to determine the algorithm parts requiring optimization. This was performed both with profilers and by manual code analysis searching for parts with quadratic complexity by the number of terms t in the skeleton monomial.

The most resource-intensive parts of the algorithm are two functions: `ZippelMultiplePrime()`, which performs a Zippel reconstruction of several expressions of various degrees in the simple case, and `BalancedZippel()`, which prepares a balanced numerator and denominator for a subsequent call to `ZippelMultiplePrime()`.

The `ZippelMultiplePrime()` function computes values of expressions at specified points using Lagrange interpolation. The estimated theoretical complexity for a single thread is high; however, with efficient parallelization, the complexity can be reduced to $O((n^2 + nm)/k)$, where k is the number of threads. The main steps of the Zippel algorithm can be illustrated as follows:

- Polynomial values at specified points are computed using Horner's scheme.
- During reconstruction, a vector of powers is generated in parallel.
- This vector of powers is multiplied by the vector of polynomial values.
- The resulting vector is scalar-multiplied by the modular inverse of the polynomial value computed via Horner's method.
- The final result vector is written to memory.

Several optimizations were introduced during GPU porting.

First, we used aligned memory, which plays a crucial role in working with two-dimensional data arrays on GPUs. The data is arranged in memory such that the addresses are powers-of-two aligned, with the alignment degree depending on the GPU architecture. This approach minimizes memory access latency and increases bandwidth during parallel access to array elements.

Second, the power vector in this algorithm is temporary and can be replaced with an accumulated sum, which length does not exceed t. Since access to this accumulator is frequent, placing it in shared memory provides the best data access performance.

Third, the modular addition operation which normally requires a modulo reduction at each step can be replaced by a standard summation with overflow control (emulating a 192-bit integer). This allows deferring a costly modulo operation until the final data processing stage.

$$(...(((a_1 \cdot b_1) \bmod N + (a_2 \cdot b_2) \bmod N) \bmod N + \dots) \bmod N) \equiv ((a_1 \cdot b_1) + (a_2 \cdot b_2) + \dots) \bmod N$$

The schematic representation of summing a large number of 64-bit values followed by modulo reduction is as follows:

```
...
ulong tempAccumLow = 0, TempAccumMid = 0, TempAccumHi = 0;
...
for (...) {
    ...
    // Multiply two 64-bit integers, result being split
    // into high (p1) and low (p0) parts
    umul_ppmm(p1, p0, term, value);

    // Add p1 and p0 to the accumulator
    add_ssaaaa(tempAccumMid, tempAccumLow, tempAccumMid, tempAccumLow, p1, p0);

    // Overflow control
    if (tempAccumMid < p1) tempAccumHi++;
    ...
}
...
// Equivalent to: ((a << 128) | (b << 64) | c) % N
NMOD_RED3(res, TempAccumHi, tempAccumMid, tempAccumLow, N);
...
```

Replacing the temporary power vector in the `ZippelMultiplePrime()` function also significantly reduces the amount of memory consumed on the GPU, which is critical when processing data in a large number of parallel threads. For example, processing a polynomial of approximately 5 million terms would require an additional 0.3 GB of memory per thread to store intermediate results.

Another block ported to the GPU is the preparation of balanced coefficients. The balancing approach involves constructing expressions by simultaneously multiplying both the numerator and denominator by additional numerical factor. This allows the numerator and denominator to be independently reconstructed using the Zippel algorithm (for details see [26]).

To construct such expressions, it is necessary to raise base variable values to various powers and substitute them into the polynomial on the GPU.

For this purpose, a fast exponentiation algorithm is used. A two-dimensional array of coefficients is generated by multiplying a base value by a precomputed k-th power of the same value, where k is the width of the array. This approach allows for quickly obtaining a coefficient array with a step of k:

$$\begin{aligned}
x_1^n, x_1^{n-1}, \ldots, x_1^{n-k} &= x_1^k \cdot x_1^{n-k}, x_1^{n-k-1}, \ldots, x_1^{n-2k} \\
x_2^n, x_2^{n-1}, \ldots, x_2^{n-k} &= x_2^k \cdot x_2^{n-k}, x_2^{n-k-1}, \ldots, x_2^{n-2k} \\
&\vdots \\
x_m^n, x_m^{n-1}, \ldots, x_m^{n-k} &= x_m^k \cdot x_m^{n-k}, x_m^{n-k-1}, \ldots, x_m^{n-2k}
\end{aligned}$$

Each GPU thread (kernel) processes its own value independently of others. Second-level loops are parallelized inside the kernel using individual threads. Each thread computes a portion of the polynomial. The final polynomial values are then summed together.

GPUs offer flexible mechanisms for inter-thread communication within the same warp, which provides significantly faster data exchange compared to global or shared memory access. In this case, inter-thread communication is implemented using the built-in function `__shfl_down_sync()`, which allows threads within a warp to exchange data in a downward direction.

Below is an example of a `warpReduceSum()` function, which sums all values across threads within a warp without relying on global or shared memory:

```
__inline__ __device__
mp_limb_t warpReduceSum(ulong val, nmod_t flint_mod)
{
    for (int shift = warpSize/2; shift > 0; shift /= 2)
    {
        val = nmod_add_d(val,
                        __shfl_down_sync(warpSize - 1, val, shift),
                        flint_mod);
    }

    return val;
}
```

To conclude, the implemented algorithm enables offloading the most resource-intensive part – coefficient reconstruction – to the GPU, thereby providing a flexible horizontal scalability. It also means that this implementation can be easily adapted for running on multiple devices or cluster nodes since in real physical examples there are multiple coefficients which have to be reconstructed, so that they can be distributed among different GPUs and even supercomputer nodes. Additionally, this approach frees up CPU resources, which can then be used for other tasks. The resulting performance will be discussed in the next section.

3. Performance Evaluation and Analysis

After developing the proposed GPU solution, it was necessary to evaluate its performance and speedup in comparison with the original CPU version (which is available as a private version of `FIRE` but with possible pre-publication access by request).

3.1. Experimental Conditions

Experiments were carried out on three GPUs: NVIDIA V100, P100 and A100, and Intel Xeon Gold 6126 2.6 GHz (12 cores, 178 Gop/s peak for INT64 operations) was used for comparing with the original CPU version. The main characteristics of used GPUs are shown in Tab. 2. The main type of data used is int64, so the theoretical peak performance was calculated for this operation type. Experiments were conducted on the equipment of the Supercomputer Center of Lomonosov Moscow State University: "pascal" partition of the Lomonosov-2 supercomputer [27] was used for tests on P100; "volta1" Lomonosov-2 partition – for tests on V100 and Intel Xeon 6126; calculations on A100 were performed on a standalone server.

Table 2. Characteristics of GPUs used for performance evaluation

GPU name	P100	V100	A100
INT64 peak performance, Top/s	2.38	3.53	4.87
Number of CUDA cores	3584	5120	6912
Theoretical peak memory bandwidth, GB/s	720	900	1555
Memory size, GB	16	32	40

Four different input data sizes were considered, all related to the final reconstruction over the sixth variable (d, the space-time dimension) using the Zippel algorithm. The datasets correspond to the same physical problem and represent reductions to master integrals of varying level. As a result, they differ in the number of monomials in the skeleton polynomial. The selected examples include the following numbers of monomials: 125 thousand ("125k"), 300 thousand ("300k"), 750 thousand ("750k"), and 4.8 million ("4.8m"). The computational complexity (and therefore the calculation time) does not depend much on the structure of the datasets, only on their size, so there was no need to consider different dataset variants.

As it was mentioned earlier, the complexity of the Zippel algorithm grows quadratically with the number of monomials, which makes this dataset sufficiently representative. In particular, the largest example was barely within the time constraints when computed on a CPU on the Lomonosov-2 supercomputer.

Each test was repeated 10 times in order to collect enough statistics on statistical significance between execution time of different experiments (the only exception is that for "4.8m" size the number of launches on CPU was reduced to 5, as they require a lot of time to run). The differences in execution times between identical runs were minimal and there were no outliers, so only average values are reported below.

3.2. Evaluations Results

Table 3 shows the results of execution time comparison on four platforms and for four input data sizes as described above. The top row shows the execution time in seconds for CPU version on Intel Xeon, while the rows below show the speedup on GPUs compared to Intel Xeon result.

We can see that execution time on Intel CPU varies from less than 1 minute up to 20+ hours, showing a wide duration spectrum. On "125k" input size, GPU speedup is up to 4, and it starts to increase as the input size gets bigger, leading up to 14.5x speedup on A100 on "4.8m" input size. It is interesting to note that on "125k" input size the speedup for V100 is shown to be bigger than the speedup for A100, although the latter GPU is more modern and powerful, but

Table 3. Comparison of execution time for different platforms

	Input data size			
Platform	**125k**	**300k**	**750k**	**4.8m**
Xeon 6126, time	44 s	255 s	1550 s	74431 s
P100, speedup	2.69	3.23	3.55	3.84
V100, speedup	3.87	5.36	7.69	8.85
A100, speedup	3.81	7.46	10.76	14.57

this difference is actually statistically insignificant (confidence intervals are overlapping). The difference between all other results is statistically significant.

In Tab. 4, the same results for GPUs are presented, but the speedup on the V100 and A100 platforms relative to the P100 is shown, which allows for an easier comparison of execution time difference between three GPUs. We can see that on "125k" input size, the speedup is not so big – less than 1.5 times both for V100 and A100, but on "4.8m" input size it rises up to 2.3 for V100 and 3.8 for A100 GPU. Thus, for small task sizes, the usage of more powerful graphics accelerators does not provide a significant benefit, but as the size increases, the advantage of more powerful GPUs becomes more and more noticeable.

Table 4. Comparison of execution time between different GPUs

	Input data size			
Platform	**125k**	**300k**	**750k**	**4.8m**
Speedup of V100 compared to P100	1.44	1.66	2.17	2.31
Speedup of A100 compared to P100	1.42	2.31	3.03	3.80

A few words about the efficiency of the proposed implementation should be said. For its rough estimation, the GPU load metric was used, which shows the percent of time during which one or more kernels was active (executing on the GPU). According to preliminary information (obtained not for all types of experiments), the efficiency grows with the increase of the task size, and for the case of "4.8m" it reaches values above 90% on all three types of GPU. For the size "700k" this value is on average not lower than 80%. A more detailed analysis is planned in the future, but it can already be concluded that the efficiency of the obtained implementation is high. It is also worth noting that the utilization within the CPU version is also high – a commonly used CPU load metric (percentage of time spent in user program) shows on average a value above 95%.

Conclusion

In this paper, a GPU implementation of the Zippel method based on porting and adapting `FLINT` functions is proposed. According to our information, this is the first implementation of this algorithm on GPUs. It provides a significant speedup when compared to a CPU variant. We intend to make the code public this year as part of the new `FIRE` version and expect this implementation to be used for different reduction tasks in elementary particle physics, both as part of the `FIRE` package and without it, since the reconstruction utilities can be used stand-alone. It is in our plans to speed up the reconstruction even more with the use of texture memory.

Acknowledgements

The work was supported by the Ministry of Education and Science of the Russian Federation as part of the program of the Moscow Center for Fundamental and Applied Mathematics under Agreement No. 075-15-2025-345. The research was carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University [27].

References

1. Anastasiou, C., Lazopoulos, A.: Automatic integral reduction for higher order perturbative calculations. Journal of High Energy Physics 07, 046 (2004). https://doi.org/10.1088/1126-6708/2004/07/046

2. Belitsky, A.V., Smirnov, A.V., Yakovlev, R.V.: Balancing act: Multivariate rational reconstruction for IBP. Nucl. Phys. B 993, 116253 (2023). https://doi.org/10.1016/j.nuclphysb.2023.116253

3. Ben-Or, M., Tiwari, P.: A deterministic algorithm for sparse multivariate polynomial interpolation. In: Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing. p. 301–309. STOC '88, Association for Computing Machinery, New York, NY, USA (1988). https://doi.org/10.1145/62212.62241

4. Chetyrkin, K.G., Tkachov, F.V.: Integration by Parts: The Algorithm to Calculate beta Functions in 4 Loops. Nucl. Phys. B 192, 159–204 (1981). https://doi.org/10.1016/0550-3213(81)90199-1

5. De Laurentis, G., Page, B.: Ansätze for scattering amplitudes from p-adic numbers and algebraic geometry. Journal of High Energy Physics 12, 140 (2022). https://doi.org/10.1007/JHEP12(2022)140

6. Hart, W.: Fast Library for Number Theory: An introduction. In: Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.) Mathematical Software – ICMS 2010. Lecture Notes in Computer Science, vol. 6327, pp. 88–91. Springer (2010). https://doi.org/10.1007/978-3-642-15582-6_15

7. Hart, W., Johansson, F., Schultz, D., *et al.*: FLINT: Fast Library for Number Theory. http://www.flintlib.org/, version 2.9 (or your version), accessed: 2025-05-01

8. Kaltofen, E., Lee, W.s., Lobo, A.A.: Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. In: Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation. p. 192–201. ISSAC '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/345542.345629

9. Klappert, J., Lange, F.: Reconstructing rational functions with FireFly. Comput. Phys. Commun. 247, 106951 (2020). https://doi.org/10.1016/j.cpc.2019.106951

10. Klappert, J., Lange, F., Maierhöfer, P., *et al.*: Integral reduction with Kira 2.0 and finite field methods. Comput. Phys. Commun. 266, 108024 (2021). https://doi.org/10.1016/j.cpc.2021.108024

11. Lange, F., Usovitsch, J., Wu, Z.: Kira 3: integral reduction with efficient seeding and optimized equation selection (2025), https://arxiv.org/abs/2505.20197

12. Laporta, S.: High precision calculation of multiloop Feynman integrals by difference equations. Int. J. Mod. Phys. A 15, 5087–5159 (2000). https://doi.org/10.1142/S0217751X00002159

13. Laurentis, G., Maître, D.: Extracting analytical one-loop amplitudes from numerical evaluations. Journal of High Energy Physics 07, 123 (2019). https://doi.org/10.1007/JHEP07(2019)123

14. Lee, R.N.: Presenting LiteRed: a tool for the Loop InTEgrals REDuction (12 2012)

15. Lee, R.N.: LiteRed 1.4: a powerful tool for reduction of multiloop integrals. J. Phys. Conf. Ser. 523, 012059 (2014). https://doi.org/10.1088/1742-6596/523/1/012059

16. Magerya, V.: Rational Tracer: a Tool for Faster Rational Function Reconstruction (11 2022)

17. Maierhöfer, P., Usovitsch, J.: Kira 1.2 Release Notes (12 2018)

18. Maierhöfer, P., Usovitsch, J., Uwer, P.: Kira—A Feynman integral reduction program. Comput. Phys. Commun. 230, 99–112 (2018). https://doi.org/10.1016/j.cpc.2018.04.012

19. von Manteuffel, A., Studerus, C.: Reduze 2 – Distributed Feynman Integral Reduction (2012), https://arxiv.org/abs/1201.4330

20. von Manteuffel, A., Schabinger, R.M.: A novel approach to integration by parts reduction. Phys. Lett. B 744, 101–104 (2015). https://doi.org/10.1016/j.physletb.2015.03.029

21. Peraro, T.: Scattering amplitudes over finite fields and multivariate functional reconstruction. Journal of High Energy Physics 12, 030 (2016). https://doi.org/10.1007/JHEP12(2016)030

22. Peraro, T.: FiniteFlow: multivariate functional reconstruction using finite fields and dataflow graphs. Journal of High Energy Physics 07, 031 (2019). https://doi.org/10.1007/JHEP07(2019)031

23. Smirnov, A.V., Chuharev, F.S.: FIRE6: Feynman Integral REduction with Modular Arithmetic. Comput. Phys. Commun. 247, 106877 (2020). https://doi.org/10.1016/j.cpc.2019.106877

24. Smirnov, A.V., Smirnov, V.A.: FIRE4, LiteRed and accompanying tools to solve integration by parts relations. Comput. Phys. Commun. 184, 2820–2827 (2013). https://doi.org/10.1016/j.cpc.2013.06.016

25. Smirnov, A.V.: FIRE5: a C++ implementation of Feynman Integral REduction. Comput. Phys. Commun. 189, 182–191 (2015). https://doi.org/10.1016/j.cpc.2014.11.024

26. Smirnov, A.V., Zeng, M.: Feynman integral reduction: balanced reconstruction of sparse rational functions and implementation on supercomputers in a co-design approach. Numerical Methods and Programming 25(Special issue), 30–45 (2024). https://doi.org/10.26089/NumMet.2024s03

27. Voevodin, V., Antonov, A., Nikitenko, D., *et al.*: Supercomputer Lomonosov-2: Large Scale, Deep Monitoring and Fine Analytics for the User Community. Supercomputing Frontiers and Innovations 6(2), 4–11 (2019). https://doi.org/10.14529/jsfi190201

28. Zippel, R.: Probabilistic algorithms for sparse polynomials. In: Ng, E.W. (ed.) Symbolic and Algebraic Computation. pp. 216–226. Springer Berlin Heidelberg, Berlin, Heidelberg (1979)