# Efficient reduction of Feynman integrals on supercomputers

Andrei V. Belitsky,[1,*] Anna A. Kokosinskaya,[2,**]
Alexander V. Smirnov,[2,3,***] Vadim V. Voevodin,[2,****] and Mao Zeng[4,*****]

(Submitted by Alexander Smirnov)

[1]Department of Physics, Arizona State University, Tempe, AZ 85287-1504, USA
[2]Moscow State University, Research Computing Center, 119992 Moscow, Russia
[3]Moscow Center for Fundamental and Applied Mathematics, 119992 Moscow, Russia
[4]Higgs Centre for Theoretical Physics, University of Edinburgh, Edinburgh, EH9 3FD, UK

Received

Abstract—Feynman integral reduction by means of integration-by-parts identities is a major power gadget in a theorist toolbox indispensable for calculation of multiloop quantum effects relevant for particle phenomenology and formal theory alike. An algorithmic approach consists of solving a large sparse non-square system of homogeneous linear equations with polynomial coefficients. While an analytical way of doing this is legitimate and was pursued for decades, it undoubtedly has its limitations when applied in complicated circumstances. Thus, a complementary framework based on modular arithmetic becomes critical on the way to conquer the current 'what is possible' frontier. This calls for use of supercomputers to address the reduction problem. In order to properly utilize these computational resources, one has to efficiently optimize the technique for this purpose. Presently, we discuss and implement various methods which allow us to significantly improve performance of Feynman integral reduction within the FIRE environment.

2010 Mathematical Subject Classification: `65Y04 Numerical algorithms for computer arithmetic, etc.; 68W10 Parallel algorithms in computer science`

Keywords and phrases: Feynman integrals, supercomputers, performance, optimization, modular arithmetic

* E-mail: andrei.belitsky@asu.edu
** E-mail: legko.zapomnit@mail.ru
*** E-mail: asmirnov80@gmail.com
**** E-mail: vadim_voevodin@mail.ru
***** E-mail: zengmao@hotmail.co.uk

## 1. INTRODUCTION

Successful verification of particle physics models demands accuracy of theoretical predictions on par with experimental measurements. This calls for calculations of high-order quantum effects in physical observables. These are readily encoded with Feynman Integrals (FIs)

$$G^{(L)}_{a_1,\ldots,a_N}(p_1,\ldots,p_E) = \int \frac{d^Dk_1}{i\pi^{D/2}} \ldots \frac{d^Dk_L}{i\pi^{D/2}} D_1^{-a_1} \ldots D_N^{-a_N} \tag{1}$$

in a $D = 4 - 2\varepsilon$ dimensionally regularized theory, where the number $L$ of $D$-fold integrations is determined by the perturbative order in coupling constants. The integrand is built out by a product Lorentz-invariant bilinear functions $D_i$ of loop $k_i = q_i$ $(i = 1,\ldots,L)$ and external $p_i = q_{i+L}$ $(i = 1,\ldots,E)$ particles' momenta raised to integer powers $a_i \in \mathbb{Z}$. For a given kinematical configuration, the complete basis of $D_i$ functions is spanned by $N = L(L+1)/2 + LE$ elements and the number of FIs thus grows as $\mathbb{Z}^N$.

To date, the most successful tool to find a minimal set, if it exists, of FIs for a given graph, is based on the integration-by-parts (IBP) identites [1],

$$\int \frac{d^Dk_1}{i\pi^{D/2}} \ldots \frac{d^Dk_L}{i\pi^{D/2}} \partial_i \cdot q_j D_1^{-a_1} \ldots D_N^{-a_N} = 0 \quad \text{for} \quad i = 1,\ldots,L\,;\ j = 1,\ldots,L+E\,, \tag{2}$$

which possess vanishing right-hand side only away from the four-dimensional space-time, i.e., $\varepsilon \neq 0$. This is the first reason to focus on $D \neq 4$. The second reason being that massless gauge theories develop infrared and (generally) ultraviolet divergences which require a regulator to make FIs well defined. As it is obvious from the representation (2), for a given point in the $\mathbb{Z}^N$ space of FIs there are $L(L+E)$ IBPs. The derivatives in the integrand shift the $a_i$ indices by $\sigma_i = \pm 1, 0$ and yield an underdetermined system of homogeneous linear equations for FIs. This system, once solved, allows one to deduce a minimal set of undetermined FIs known as the Master Integrals (MIs). A priori, it is completely unclear whether that number is even finite given there are infinitely many relations for infinitely many integrals as the integers $a_i$ are not restricted. However, this was affirmatively demonstrated in Ref. [2] using algebro-geometric techniques. Unfortunately, the proof is non-constructive and cannot be used either to help in solving the reduction problem or even to determine the number of MIs for a given family of FIs.

The procedure is thus to efficiently solve a huge sparse non-rectangular homogeneous system of linear equations

$$\sum_{\{\sigma_1,\ldots,\sigma_N\}} c_{\sigma_1,\ldots,\sigma_N} G^{(L)}_{a_1+\sigma_1,\ldots,a_N+\sigma_N} = 0 \tag{3}$$

with polynomial coefficients $c_{\sigma_1,\ldots,\sigma_N}$, being functions of the space-time dimension $D$ and kinematical invariants $s_{ij} \equiv (p_i + p_j)^2$. This is a classical problem in linear algebra and belongs to centuries-old books. However, a practical tool applicable to field-theoretical setups was suggested relatively recently starting with the work by Laporta [3]. It is based on a version of the well-known Gaussian elimination technique and takes into account a chosen priority of points in the $\mathbb{Z}^N$ space of FIs. It roughly consists in starting with a sector containing the smallest allowed number of $D_i$'s, solving for a subset of FIs and subsequently substituting these into FIs of the next level in complexity containing more $D_i$'s etc. etc. This is done analytically in $D$ and $s_{ij}$ without specifying their numerical values and relies on a heavy computer use. This technique is currently implemented in quite a number of public and private computer codes such as, but not limited to, `AIR` [4], `FIRE` [5], `LiteRed` [6] and `Kira` [7].

While the traditional IBP approach described in the previous paragraph was proven to be extremely successful in practical solution of problems in high energy physics, in many circumstances it commands the use of computers with multiple CPUs and, what is worse, terabytes of RAM. Nowadays, multiple CPUs is an industry standard for personal computers and work stations, however, the need for high-memory nodes is a user-unfriendly constraint, either in their very availability even on superclusters or long queue waiting times for their access. And even if both of these conditions are fulfilled, an IBP reduction for a complex Feynman graph could take months of computer time without any warranty for its success in the end. The reason for this is clear from our

brief description of the Laporta algorithm alluded to above. Namely, at every step of the system-solving procedure, one needs to call for an external simplification library of rational fractions, which are generated in coefficients accompanying MIs, and bring the fraction $c/c'$ to a canonical form by getting rid of common factors in its numerator and denominator. This is a time-consuming process and turns out to be the bottleneck for the entire endeavor. A recent work [27] introduced efficient libraries to partially alleviate the problem. However, one definitely needs to look for alternative approaches which would bypass the need for excessive use of RAM required to handle swelling intermediate expressions.

A new framework is especially indespensable for physical observables depending on multiple scales such as high-multiplicity scattering amplitudes and form factors, see, e.g., recent [9, 10]. These involve multiple $s_{ij}$-invariants and, if one needs to consider off-shell kinematics, depend in addition on external particles' virtualities $p_i^2 \neq 0$. The presence of the latter makes the resulting IBP reduction practically insurmountable for direct analytical calculations. In fact, a technique which overcomes this predicament is known for almost a decade now [13–21] and is based on modular arithmetic. Its basic idea is quite simple: substitute numerical values for all variables involved and then solve the resulting system of IBP equations (3). This problem is naively much easier since the emerging coefficients are now numbers rather than ratios of lengthy polynomials. Still working with rational numbers may not be efficient enough for after the number coefficients can swell uncontrollably as well in the course of the Gaussian elimination and, therefore, not fit into the machine arithmetic format, either 64-bit or even 128-bit. To practically work with these numbers, one would need special libraries using long-integer arithmetic. This would make the reduction inefficient, defeating the purpose from the get-go. This impels one to switch to the modular arithmetic on the finite field $\mathbb{Z}_p$ by choosing the prime $p$ not exceeding $2^{64}$ instead of working in the field of rationals $\mathbb{Q}$. This is a standard practice in computer science as was reminded to the physics community in Ref. [11]. In this set-up, all the expansion coefficients $c_{\sigma_1,\ldots,\sigma_N}$ fit into the machine arithmetic and, as a consequence, an IBP reduction can be effectively run very fast.

A single point or a few (modular or not) in the $\boldsymbol{v} = (D, s_{ij})$ parameter space will not suffice to get the full parametric dependence of $c$'s. The question is then how many does one need to unambiguously reconstruct the polynomials with rational number coefficients without any prior knowledge of their order or the number of nonvanishing terms, and, in addition, accompanying expansion coefficients restored in turn from finite fields when these are used. This last step relies on classical ideas based on the well-known Chinese Remainder Theorem to combine short primes, which fit into a processor word, into large numbers making subsequent rational reconstruction from $\mathbb{Z}_{p_1\times\cdots\times p_n}$ by the Extended Euclidean Algorithm unambiguous [12]. The reconstruction from rational numbers to rational functions is far more complex and was addressed in Refs. [13–21] making use of a variety of methods. Some of these are more time consuming than the others. Thus the goal of an efficient reconstruction approach is then to require (i) the least possible number of blackbox samples (BBS) and (ii) a feasible restoration of mutivariate polynomials from these samples. Obviously, these two steps combined have to take significantly less cumulative time than the actual analytical reduction (if it is at all possible).

A very attractive feature of the modular approach is its amenability to parallelization: IBP reductions for various values of parameters over the finite field $\mathbb{Z}_p$ can be performed independently from one another. And while one might need thousands or even millions of reduction results, this approach ideally fits for supercomputer use, if properly organized. The IBP program `FIRE`, developed in Ref. [22], already had the modular approach built-in in the code starting from the version `6.0` [23]. However, its previous implementation was not sufficiently practical, in spite of being successfully used once in Ref. [24], due to unbefitting reconstruction methods. This last problem was resolved in Ref. [21] with the development of a novel technique for dense interpolation dubbed the balanced reconstruction, however, it was not coded or optimized for supercomputers. This drawback will be overcome in this paper. As a benchmark, we will use an example from a recent study of non-planar FIs contributing to a three-leg off-shell form factor [10], in particular, one of the most time-consuming IBP reductions of 38 level-7 FIs, arising in differential equations, down to the level 3 MI $G^{(2)}_{0,1,0,1,1,0,0,0,0}$. The standard analytical route was possible there but it was taking more than ten days and required extremely high memory use. Below we will report on the performance upgrade and improvements of the current version of `FIRE` which yielded a significant overall time reduction by almost an order of magnitude compared to the analytical approach.

Below to provide an in-depth analysis of various optimization techniques used to make the reconstruction of IBP identities more efficient, including the use of coding in C++, memory management and use of external libraries like Flint.

## 2. RECONSTRUCTION IN C++

Up to now, all public versions of `FIRE` capable of reconstruction, the initial [23] and the more recent [21], relied on codes written in `Wolfram Mathematica`. The latter is a commercial software which is typically not installed on supercomputers. Thus, in Ref. [24] alluded to above, proliferating back-and-forth transfers had to be done between a supercomputer, used to generate sampling points, and a local machine, used to reconstruct from these, creating a logistical burden. While it was borderline feasible for two-variable observables studied there, it would be highly impractical for more. In the current version of `FIRE`, used throughout this work, the balanced reconstruction [21] is coded in C++. A typical line in a script, which provides a brief glossary of options, reads[6]

```
mpirun -np NC0 FIRE6_MPI --calc ${CALC0} --reconstruct --variables v1_v2_v30 --newton N1_N2
                         -c config "v10:v1T;v20:v2T;v30:v30;P0"
```

This calls for the generation of BBS with `FIRE` for the $\boldsymbol{v}$-variables $v_1$, $v_2$ and $v_3$ in their Thiele ranges $[v_{1,0}, v_{1,\mathrm{T}}]$, $[v_{2,0}, v_{2,\mathrm{T}}]$ (the corresponding Newton ranges of the balanced reconstruction are defined by the first $N_1$ and $N_2$ elements in these) and the last variable $v_3$ in this example is fixed at $v_{3,0}$. The number of primes to be used in the rational reconstruction is set by the value $P_0$. Moreover, the code is organized in a manner that automatically shuts the production of sampling points down when their sufficient number is accumulated for a successful unambiguous reconstruction. This is activated by adding the flag `--abort`. For large-scale computations, the number of generated files could potentially exceed the storage capacity of Linux file systems and `FIRE` can be instructed to delete BBS no longer needed in the course of a computation with `--delete_tables`. Last but not least, external computer algebra systems used in intermediate simplifications is selected with the option `CALC0`.

## 3. OPTIMIZATION BY CHANGING THE ORDER OF RECONSTRUCTIONS

According to the general logic spelled out in the Introduction, and made explicit in its implementation in the sample script of the previous section, one starts the process by substituting integer values for all $\boldsymbol{v}$-variables in IBP identities and then proceeds with a choice of prime numbers for the modular arithmetic. So it only appears natural to perform the reconstruction in the opposite order: (let us dub it as the 'Beast' mode[7]) primes $\to$ rational coefficients $\to$ rational functions. This approach was originally implemented in `FIRE` with `Mathematica` [21] and its private C++ realization. This route was rather straightforward since it did not require employment of modular-polynomial arithmetic being that the reconstruction of rational numbers from primes was done first. Then an external library was called for operations on polynomial and rational function.

However, the reader can already anticipate a drawback intrinsic to the above approach. Namely, when reconstructing a rational number from its projections over modular fields, the number of primes required for its unique restoration heavily depends on the 'size' of the former, i.e., the maximum of absolute values of its numerator and denominator. The bigger the size is, the more modular values one needs and hence more IBP reductions to perform. So this quickly becomes an issue as the expansion coefficients $c$ in the IBPs (3) are polynomial in the $\boldsymbol{v}$-variables with integer-valued expansion coefficients $C_{i_1 i_2,\ldots}$, which are sought for,

$$c = \sum_{i_1,i_2,\ldots} C_{i_1,i_2,\ldots} v_1^{i_1} v_2^{i_2} \ldots \tag{4}$$

and after the variable substitutions and summation, the result is in general much bigger in size than the initial values of $C_{i_1,i_2,\ldots}$ that one started with.

A panacea to the above predicament was found in changing the order of the two steps by first reconstructing rational functions with their $C$-coefficients in a modular field and then restoring

[6] Depending on the `Slurm` scheduler, `mpirun` can be superseded by `srun` and the accompanying flag for the number of cores `NC0` used changes from `-np` to `-n`.

[7] Ugly but works.

| Method | Beast mode | | Beauty mode | |
|---|---|---|---|---|
| BBS / vars | # p's | # r's | # p's | # r's |
| $d$ | 11 | $13_{\mathrm{T}}$ | 11 | $13_{\mathrm{T}}$ |
| $w$ | 14 | $100_{\mathrm{T}}$ | 11 | $100_{\mathrm{T}}$ |
| $v$ | 15 | $106_{\mathrm{T}}$ | 11 | $106_{\mathrm{T}}$ |
| $u$ | 18 | $128_{\mathrm{T}}$ | 11 | $128_{\mathrm{T}}$ |
| $(d,w)$ | 15 | $13_{\mathrm{T}}\times51_{\mathrm{N}}$ | 11 | $13_{\mathrm{T}}\times51_{\mathrm{N}}$ |
| $(d,w,v)$ | 16 | $13_{\mathrm{T}}\times51_{\mathrm{N}}\times54_{\mathrm{N}}$ | 8 | $13_{\mathrm{T}}\times51_{\mathrm{N}}\times54_{\mathrm{N}}$ |
| $(d,w,v,u)$ | 19 | $13_{\mathrm{T}}\times51_{\mathrm{N}}\times54_{\mathrm{N}}\times65_{\mathrm{N}}$ | 7 | $13_{\mathrm{T}}\times51_{\mathrm{N}}\times54_{\mathrm{N}}\times65_{\mathrm{N}}$ |

Table 1. Comparison of the Beast (primes → rational coefficients → rationals functions) and Beauty modes (primes → rationals functions → rational coefficients) restorations. Labels on the number of rational numbers required for polynomial reconstruction stand for Thiele (T) and balanced Newton (N), respectively.

them from primes to rationals, i.e., (let us call it the 'Beauty' mode[8]) primes → rationals functions → rational coefficients. The first step here is done with the very same balanced method of Ref. [21], while in the second step, we parse the reconstructed rational functions over primes by analyzing their structure: We throw away exceptional cases where something was accidentally canceled out due to an unfortunate choice of variables and consider only rational functions possessing the same monomial structure both in their numerators and denominators but, obviously, different expansion coefficients over primes. Then for each set of these, accompanying a given monomial, we finally run the rational reconstruction.

Another feature of paramount importance which makes the Beauty preferable to the Beast mode is the fact that the number of prime values needed for an unambiguous reconstruction of rationals is not only smaller but stable as well, i.e., it does not depend on numerical values of variables used because the expansion coefficients $C_{i_1,i_2,\ldots}$ in the original function are constants to start with. This eliminates the guessing game from the robust estimate for seeding of BBS. For the Beast mode it is almost next to impossible to predict the number of necessary reductions in advance, only an upper limit estimate, which yields quite an overestimation, is possible. The new way allows us to minimize the number of IBP reductions. This will be discussed in the following section.

Before closing this section, let us demonstrate the realization of these improvements with our benchmark. The problem involves five variables $\boldsymbol{v} = (D, w, v, u, m)$, i.e., the space-time dimension $D$, three Madelstam invariants $u, v, w$ and a virtuality $m$. Since FIs are homogeneous functions of the $(w, v, u, m)$ variables, one can safely set one of them equal to one, say $m = 1$, and restore it at the very end from naive dimensional counting. To make a robust estimate for the number of sample points required in each of these, we ran trial IBP reductions (each takes about a minute or two on a small number of cores $\sim 50$), which are summarised in the first four rows of Table 1. We used there the following starting value for $\boldsymbol{v}_0 = (40, 30, 20, 10, 1)$ and relied on `CALCO=Flint` [25] as a simplifying external software. We immediately observe that while the number of necessary modular values in the finite field grows substantially with increasing initial values of variables for the Beast mode, it stays the same for the Beauty. The reason for this was already elucidated in the previous paragraph. As we enlarge the set of to-be-reconstructed variables, for the Beast mode we always have to employ the largest number of primes among those needed for individual restorations. Moreover, it was even necessary to add an extra one to warrant successful runs. On the contrary, in the Beauty mode, the number of primes decreases with the increasing number of variables subject to restoration. This trend is obvious from the right-hand side of Eq. (4) since, as we first perform

[8] Elegant and efficient.

the rational function reconstruction, the size of accompanying coefficients gets smaller[9]. This yields an e-fold, i.e., $19/7 \simeq 2.7$, reduction in the overall number of required sample points. In practical applications, it is advisable to increase the Newton's range for all variables by a few values to ensure successful reconstruction. The reason being that sometimes tables may arrive corrupted, or not at all, from some nodes and, therefore, unsuitable for further use. With this caveat in mind, for the full ($d$,$w$,$v$,$u$)-variable computation, performed on 1024 cores of the ASU's `Sol` supercomputer [26], we clocked the two reconstructions at 2-10:26 ($\sim$ 58 hours) and 1-04:22 ($\sim$ 28 hours), respectively. This is more than a factor of two speedup, making the Beauty mode copacetic. Further details will be provided in Section 5 below.

## 4. ANALYZING AND OPTIMIZING APPLICATION PERFORMANCE

The methods discussed in the previous sections were intended to minimize the number of reductions needed to produce the reconstruction, so they belong to algorithmic improvements. In this section, we would like to focus on another important area of improvement – software optimizations based on application performance analysis.

The program analysis was performed on the Lomonosov-2 supercomputer. In most cases, the `pascal` partition was used, in which each node contains one 12-core Intel Xeon Gold 6126 CPU with 92 GB of RAM. The application ran on 210 processors, with Open MPI 1.8.4 being used. The average execution time of the analyzed application is 115 s.

First, an analysis of working with MPI was carried out. The program operates in a master-worker paradigm, where an MPI process with rank zero is assigned a role to distribute tasks and control the execution of other processes. Analysis of MPI usage in the application was performed using mpiP 3.5 profiler [31].

The results showed that MPI operations occupy only a small portion (<1%) of the overall execution time. It was observed that the master process spends more than 96% of the time waiting for the completion of tasks by other processes. But if we start distributing the payload to the master process as well, this does not lead to noticeable redistribution load, but may cause the master process to respond more slowly to worker process requests. Moreover, such a small fraction of the time spent on MPI operations suggests that working with MPI is clearly not a bottleneck in this program.

Next, a general performance analysis of individual processes was carried out using the Intel VTune Profiler tool, version 2019.5. Fig. 1 shows our analysis of the most computationally expensive fragments (functions, loops) used in the application. In each line, the leftmost column corresponds to the fragment name, the time for executing this fragment is indicated in "CPU Time" column, and on the right side the columns show the values of metrics from the Top-down approach [32]. A detailed description of Top-down metrics can be found in Intel documentation [33]. The rightmost column shows whether this fragment is executed in the program itself (the value "FLAME6") or whether it belongs to an external library that was called from the program.

Many of the most frequently used functions are Front-End Bound (corresponding cells are marked red), which means that the processor quite often does not manage to promptly preprocess instructions for their execution in functional units, leading to them being idle waiting. Next, one can notice that there are functions from the Linux kernel module `vmlinux` for paged memory organization (for example, `copy_page_rep`, `clear_page_c_e`), which also have a high Memory Bound indicator, i.e., during their execution the processor is often idle waiting for data from the memory. In addition, many external functions have a high Bad Speculation value. This indicates that the processor is often busy executing instructions that then turned out to be unnecessary (the most common reason is incorrect branch prediction). From the other side, the operations `add_to` (line 1), `mul_mod` (line 3) and `mul_inv` (line 6), which are the part of the analyzed program itself, have a high Core Bound value, which usually indicates insufficient loading of functional units (e.g., due to data dependency) or restrictions imposed by some complex arithmetic operations, such as division, which is quite widely used in this application. You can also highlight the constructors of the `point` class, which are also appear to be Memory Bound functions.

It is worth mentioning the vectorization aspect of the code. In this program, the main operations are performed on integers, not on floating-point operations, and VTune does not allow assessing the

[9] The $C$ coefficients are no longer multiplied by high powers of integers!

quality of vectorization of such operations (for this purpose, a tool like Intel Advisor can be used). However, it is known that the vectorization in this program can be improved, and this issue will be addressed in the future.

Grouping: Function / Call Stack

| Function / Call Stack | CPU Time ▾ | Clockticks | Instructions Retired | CPI Rate | Retiring | Front-End Bound | Bad Speculation | Back-End Bound | | A... | Module |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Memory Bound | Core Bound | F | |
| [Loop at line 3268 in add_to<__gnu_cxx::__no | 105.497s | 343,59... | 131,534... | 2.612 | 20.2% | 33.9% | 15.1% | 1.8% | 29.0% | ... | FLAME6 |
| copy_page_rep | 76.123s | 248,37... | 3,016,0... | 82.353 | 9.1% | 5.4% | 0.5% | 77.9% | 7.0% | ... | vmlinux |
| mul_mod | 45.046s | 146,41... | 113,321... | 1.292 | 32.5% | 35.7% | 1.1% | 3.7% | 26.9% | ... | FLAME6 |
| copy_user_enhanced_fast_string | 39.117s | 128,03... | 9,230,0... | 13.872 | 9.8% | 3.4% | 0.0% | 77.7% | 9.9% | ... | vmlinux |
| clear_page_c_e | 31.800s | 104,88... | 3,445,0... | 30.445 | 10.4% | 1.3% | 0.0% | 82.7% | 6.6% | ... | vmlinux |
| [Loop at line 755 in mul_inv] | 28.377s | 91,260... | 61,685,... | 1.479 | 34.2% | 36.2% | 2.3% | 0.7% | 26.7% | ... | FLAME6 |
| system_call_after_swapgs | 23.630s | 76,934... | 5,434,0... | 14.158 | 10.0% | 32.9% | 11.2% | 2.7% | 43.2% | ... | vmlinux |
| [panfs] | 23.565s | 79,131... | 46,254,... | 1.711 | 25.8% | 46.2% | 3.6% | 15.8% | 8.7% | ... | panfs |
| sysret_check | 23.515s | 77,259... | 6,565,0... | 11.768 | 12.7% | 33.6% | 9.2% | 2.6% | 41.9% | ... | vmlinux |
| __memcmp_sse4_1 | 22.458s | 74,672... | 104,767... | 0.713 | 32.2% | 35.8% | 18.9% | 7.5% | 5.6% | ... | libc-2.17.so |
| page_fault | 15.536s | 51,220... | 1,924,0... | 26.622 | 13.8% | 11.5% | 8.2% | 19.8% | 46.6% | ... | vmlinux |
| point::point | 14.835s | 50,804... | 78,793,... | 0.645 | 36.2% | 17.1% | 0.0% | 31.6% | 16.4% | ... | FLAME6 |
| [Loop@0x19cf0 in strcmp] | 13.692s | 43,043... | 8,047,0... | 5.349 | 5.3% | 7.9% | 13.3% | 63.6% | 9.9% | ... | ld-2.17.so |
| retint_userspace_restore_args | 13.497s | 43,784... | 5,109,0... | 8.570 | 10.1% | 25.1% | 4.3% | 2.8% | 57.7% | ... | vmlinux |
| error_swapgs | 12.950s | 44,915... | 2,769,0... | 16.221 | 8.1% | 23.2% | 8.5% | 2.3% | 57.9% | ... | vmlinux |
| check_match.9522 | 12.720s | 41,678... | 10,959,... | 3.803 | 8.6% | 9.3% | 17.8% | 53.8% | 10.3% | ... | ld-2.17.so |
| __memcpy_ssse3_back | 12.409s | 41,184... | 46,319,... | 0.889 | 33.2% | 35.7% | 8.1% | 14.9% | 8.2% | ... | libc-2.17.so |
| [Loop at line 291 in point::point] | 11.527s | 38,714... | 91,429,... | 0.423 | 53.4% | 31.8% | 0.0% | 1.6% | 16.6% | ... | FLAME6 |
| kyotocabinet::CacheDB::accept_impl | 11.276s | 36,621... | 19,370,... | 1.891 | 20.0% | 14.6% | 21.0% | 32.6% | 11.9% | ... | FLAME6 |
| [Loop@0x9660 in do_lookup_x] | 11.241s | 36,452... | 31,980,... | 1.140 | 28.1% | 20.8% | 20.0% | 19.5% | 11.6% | ... | ld-2.17.so |
| [Loop at line 3268 in add_to<std::_List_const_i | 10.926s | 36,062... | 15,860,... | 2.274 | 18.4% | 44.2% | 30.3% | 0.8% | 6.4% | ... | FLAME6 |
| _raw_spin_lock | 10.044s | 32,851... | 15,340,... | 2.142 | 14.3% | 30.7% | 4.8% | 41.5% | 8.7% | ... | vmlinux |
| tcmalloc::ThreadCache::FreeList::TryPop | 9.517s | 30,953... | 47,151,... | 0.656 | 49.7% | 35.8% | 18.8% | 0.0% | 0.0% | ... | FLAME6 |

Figure 1. The most computationally expensive program fragments and their performance metrics, analysis performed using Intel VTune Profiler

Based on the performed analysis, the following conclusions can be drawn regarding the possibility of program optimization:

1. MPI operations occupy only a small part of the program, and optimizing this aspect of the application will not bring a significant speedup.

2. In addition to application functions, Linux system calls for working with paged memory (for example, `copy_page_rep`, `clear_page_c_e`) take up significant time, and these calls show rather low performance. Thus, a possible optimization option is to modify the program to reduce the number of such calls, or to find more optimized implementations of these libraries.

3. A number of program functions (like `mul_inv`, `mul_mod`) can leave CPU underutilized due to the use of operations with large delays (such as the division operation). Modern compilers can optimize division when using 32-bit numbers, but for 64-bit and 128-bit operands DIV, IDIV operations are generated, and the delay in this case can exceed 70 processor cycles, as well as the readiness for the next such operation. In some situations, it may be possible to change the code to use multiplication instead of division (if overflow problems are not expected), or to use approximate division implementations.

4. Some functions (for example, `add_to`) have a high Bad Speculation score, which is usually due to the Branch Prediction. In the case of the specified `add_to` function, this can happen both due to the presence of a loop with a break condition or the presence of branching operators in the loop. Modifying this fragment can also help speed up the execution of the program.

5. The program is not fully vectorized, and a more detailed study of this issue is also an option for further optimization.

6. Some constructors of the point class are memory-bound functions – they frequently access the memory of different arrays. For them, it is worth considering memory optimization (for example, improving data locality).

After performance analysis, an optimized version of the program was developed. In the second version of the application, two modifications are introduced: 1) the constructor of the point class now uses bit storage to optimize memory management; 2) an approximate implementation of the division operation is used (the division accuracy is sufficient in this case, and the execution time of such an implementation of the operation is reduced). Other optimization options are expected to be tested in the future.

The average execution time of the optimized program decreased by 5 seconds, which is 4.3% of the total execution time of the initial version.

The performance of the optimized version was analyzed using VTune and compared with the initial version. The analysis was carried out in the same conditions specified at the beginning of this section.

Let us briefly describe the main conclusions. Although the speedup is not so notable, the efficiency metrics (useful utilization of CPU, as well as the average number of clock cycles per instruction) improved comparing to the initial program. Further, the list of the most computationally expensive program fragments (see Fig. 2) changed significantly. Now it is mainly occupied by functions from the `glibc` library for allocating and freeing memory, such as `malloc` and `free`. Moreover, the fragments from the program itself (i.e., not external functions) first occur only at the 12th place in Fig. 2, i.e., the top 11 most computationally expensive fragments are executed outside the body of the program itself. This indicates that for further optimization it is necessary to either use more optimized implementations of external functions (i.e. use other libraries) or reduce the number of external calls in the program, primarily related to memory allocation/freeing and working with memory pages. This will be done in the future.

Grouping: Function / Call Stack

| Function / Call Stack | CPU Time ▾ | Cloc... | Instru... Retired | CPI Rate | Retiring | Front-End Bound | Bad Speculation | Back-End Bound | | Average CPU Frequency | Module |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | Memory Bound | Core Bound | | |
| libc_malloc | 95.739s | 317,... | 738,... | 0.430 | 69.9% | 18.5% | 2.8% | 2.3% | 6.6% | 3.3 GHz | libc-2.17.so |
| _int_free | 69.207s | 227,... | 518,... | 0.439 | 58.7% | 16.7% | 0.0% | 14.0% | 18.1% | 3.3 GHz | libc-2.17.so |
| _int_malloc | 64.481s | 210,... | 260,... | 0.810 | 37.1% | 22.4% | 17.7% | 11.0% | 11.9% | 3.3 GHz | libc-2.17.so |
| copy_user_enhanced_fast_string | 34.887s | 116,... | 8,76... | 13.255 | 9.3% | 3.0% | 0.0% | 77.3% | 10.8% | 3.3 GHz | vmlinux |
| [Loop@0x81040 in _int_free] | 32.847s | 109,... | 96,2... | 1.134 | 10.4% | 17.6% | 24.8% | 17.7% | 29.5% | 3.3 GHz | libc-2.17.so |
| sysret_check | 26.211s | 85,8... | 6,48... | 13.228 | 10.7% | 35.5% | 9.6% | 2.1% | 42.1% | 3.3 GHz | vmlinux |
| system_call_after_swapgs | 25.234s | 83,6... | 5,77... | 14.500 | 10.2% | 29.2% | 10.0% | 2.8% | 47.8% | 3.3 GHz | vmlinux |
| __memcmp_sse4_1 | 23.886s | 77,6... | 105,... | 0.738 | 35.2% | 45.8% | 23.6% | 0.0% | 0.0% | 3.3 GHz | libc-2.17.so |
| [panfs] | 22.934s | 75,3... | 48,8... | 1.542 | 26.2% | 42.8% | 2.4% | 18.5% | 10.1% | 3.3 GHz | panfs |
| [Loop@0x80570 in malloc_consolidate] | 16.814s | 57,7... | 98,4... | 0.587 | 45.5% | 12.5% | 20.1% | 12.6% | 9.3% | 3.4 GHz | libc-2.17.so |
| [Loop@0x82340 in _int_malloc] | 16.594s | 51,9... | 127,... | 0.407 | 46.1% | 6.6% | 0.0% | 17.5% | 34.4% | 3.1 GHz | libc-2.17.so |
| point::point | 15.637s | 48,7... | 62,4... | 0.781 | 42.7% | 17.0% | 13.2% | 17.3% | 9.8% | 3.1 GHz | FLAME6 |
| page_fault | 14.634s | 47,5... | 1,82... | 26.121 | 11.5% | 11.4% | 10.0% | 22.8% | 44.3% | 3.2 GHz | vmlinux |
| error_swapgs | 12.309s | 39,7... | 3,10... | 12.791 | 10.8% | 27.4% | 10.3% | 3.3% | 48.2% | 3.2 GHz | vmlinux |
| retint_userspace_restore_args | 12.224s | 41,8... | 5,35... | 7.818 | 11.6% | 30.2% | 4.9% | 1.7% | 51.6% | 3.4 GHz | vmlinux |
| [Loop at line 69 in n_gcdinv] | 11.627s | 38,8... | 48,5... | 0.800 | 50.3% | 40.3% | 7.9% | 0.1% | 1.5% | 3.3 GHz | libflint.so.18.0.0 |
| __memcpy_ssse3_back | 11.151s | 37,1... | 43,9... | 0.845 | 35.4% | 45.0% | 18.9% | 0.5% | 0.2% | 3.3 GHz | libc-2.17.so |
| [Loop at line 293 in point::point] | 10.981s | 36,6... | 109,... | 0.335 | 66.3% | 35.3% | 0.0% | 0.6% | 3.7% | 3.3 GHz | FLAME6 |
| free | 10.434s | 35,1... | 17,6... | 1.989 | 15.2% | 32.6% | 46.4% | 5.0% | 0.9% | 3.4 GHz | libc-2.17.so |
| nmod_mul | 10.029s | 31,3... | 81,5... | 0.384 | 60.6% | 5.0% | 15.8% | 2.3% | 16.3% | 3.1 GHz | FLAME6 |
| kyotocabinet::CacheDB::accept_impl | 9.688s | 31,9... | 23,8... | 1.338 | 29.8% | 14.6% | 10.4% | 29.8% | 15.4% | 3.3 GHz | FLAME6 |
| nmod_mul | 9.427s | 31,0... | 41,8... | 0.742 | 24.6% | 11.9% | 26.2% | 16.0% | 21.2% | 3.3 GHz | FLAME6 |
| point::operator= | 8.866s | 30,0... | 54,5... | 0.551 | 41.5% | 17.5% | 3.4% | 8.2% | 29.5% | 3.4 GHz | FLAME6 |
| _raw_spin_lock | 8.796s | 29,8... | 15,5... | 1.925 | 17.8% | 38.6% | 3.7% | 30.9% | 9.1% | 3.4 GHz | vmlinux |
| check_match.9522 | 8.079s | 24,6... | 10,8... | 2.285 | 14.9% | 12.2% | 17.4% | 43.5% | 12.0% | 3.1 GHz | ld-2.17.so |
| [Loop@0x9660 in do_lookup_x] | 7.658s | 24,9... | 25,8... | 0.967 | 31.9% | 25.0% | 22.1% | 10.9% | 10.2% | 3.3 GHz | ld-2.17.so |
| [Loop@0x19cf0 in strcmp] | 7.437s | 24,3... | 7,64... | 3.187 | 8.9% | 12.9% | 16.6% | 46.3% | 15.3% | 3.3 GHz | ld-2.17.so |
| [Loop at line 3275 in add_to<__gnu_cxx::__normal_ite | 6.961s | 22,5... | 53,0... | 0.425 | 63.3% | 5.3% | 24.1% | 1.0% | 6.4% | 3.2 GHz | FLAME6 |
| [Loop at line 3565 in pass_back(std::set<point, std::gr | 6.766s | 22,5... | 13,3... | 1.685 | 12.4% | 3.4% | 5.9% | 63.6% | 14.7% | 3.3 GHz | FLAME6 |
| mem_cgroup_charge_common | 6.530s | 19,6... | 2,40... | 8.157 | 2.9% | 2.5% | 9.2% | 75.1% | 10.3% | 3.0 GHz | vmlinux |

Figure 2. The most computationally expensive program fragments in the optimized version of the application

## 5. OPTIMIZATION WITH THE USE OF FLINT LIBRARY

The use of the FLINT library [25] in FIRE was already implemented in Ref. [5] through the library FUEL [27]. Initially, there was no support for rational functions with modular arithmetic since these were not used within the Beast mode. But after changing the reconstruction order, as was discussed in Section 3, there arose an urgent need to efficiently perform such operations in the Beauty mode.

Since FLINT does not (yet) have an in-house implementation of modular rational functions, we wrote our own version on top of FLINT's routines for polynomials over prime fields. The FLINT

routines include binary arithmetic operations (additions etc.) for such polynomials, polynomial GCD computations and exact division of polynomials when there is no remainder. We translated a similar implementation of modular rational functions from the Julia package Nemo [28] to C++. As an example, let us spell out the algorithm, translated from Nemo, for computing and simplifying the sum of two rational functions, $n_1/d_1 + n_2/d_2$, where $n_1, n_2, d_1, d_2$ are polynomials (over prime fields). We assume the two rational functions to have been simplified previously, which means $n_1$ and $d_1$ have no nontrivial polynomial GCD, and similarly for $n_2$ and $d_2$. Various special cases, corresponding to `if-else` statements in the actual code, are treated separately to ensure efficiency on a computer. We summarize the algorithm below.

- Case 1: $d_1 = d_2$.

  The sum is equal to the new rational function $(n_1 + n_2)/d_1$. If $d_1 = 1$, this is the returned result. Otherwise, we compute the polynomial GCD of $(n_1 + n_2)$ and $d_1$, and divide the numerator and denominator by the GCD if the GCD is not equal to one, before returning the rational function.

- Case 2: $d_1 = 1$.

  The sum is equal to the new rational function $(n_2 + d_2)/d_2$. Since $n_2$ and $d_2$ have no nontrivial polynomial GCD, $(n_2 + d_2)$ and $d_2$ also have no nontrivial polynomial GCD, and the above rational function is returned.

- Case 3: $d_2 = 1$.

  The treatment is similar to case 2.

- Case 4: all other cases.

  We compute the polynomial GCD of $d_1$ and $d_2$ and let the result be $g$.

  - Case 4(a): $g$ is equal to 1.
    The sum is $(n_1 d_2 + n_2 d_1)/(d_1 d_2)$. This result is returned, since it is easy to prove that the numerator and denominator cannot have a nontrivial GCD, under the starting assumption that $n_1/d_1$ and $n_2/d_2$ are previously simplified rational functions.
  - Case 4(b): $g$ is a nontrivial polynomial.
    We let $q_1 = d_1/g$ and $q_2 = d_2/g$. The sum is $(n_1 q_2 + n_2 q_1)/(q_1 q_2)$, and we further simplify this sum if there is a nontrivial GCD between the numerator and denominator to be canceled, before we return the result.

Another performance improvement we have implemented is related to the communication between FIRE and the simplification library, in this case FLINT. All communications are via strings: FIRE sends strings of rational functions expressions to FLINT, while FLINT (with the help of our parser code) reads the strings and sends back the simplified rational functions, again as strings. Since simplified rational functions will be used as sub-expressions in subsequent calculations, long expressions may be re-sent to the simplification library, causing a re-evaluation before further calculations. This overhead is avoided if the expression is stored in memory in an internal format native to the simplification library. To accomplish this without a dramatic rewrite of the FIRE database, we retained the use of strings as the bidirectional communication format, while adding special syntax to the strings, support4ed by our parser, to refer to expressions that are stored in memory. Specifically, the expressions are stored in a hash map that maps its numerical label (1, 2, 3, etc.) to the actual expression in a native (i.e. non-string) format. Imagine two previously simplified rational functions, say, $(1 - x)/(1 + x^2)$ and $x/(1 - x^2)$, have to be added in a subsequent calculation, previous versions of FIRE will send the string $(1 - x)/(1 + x^2) + x/(1 - x^2)$ to the simplifier, while the current version of FIRE can optionally send a string like {1}+{3} to mean adding the 1st and 3rd expressions stored in memory.[10] This has resulted in dramatic performance improvements. In our four-variable benchmark example described in the last paragraph of Section 3, the measurement of times needed for rational function reconstruction of one-to-four variables in the Beauty mode is shown in Table 2.

[10] The actual implementation requires some memory management such as periodic cleaning of unneeded stored expressions to prevent memory usage from growing out of control.

| vars/reconstruction | Old Way | New Way | With storing |
|---|---|---|---|
| $d$/Thiele | 0.06 | 0.06 | 0.06 |
| $(d, w)$/Balanced Newton | 2.49 | 1.77 | 0.6 |
| $(d, w, v)$/Balanced Newton | 85 | 39.6 | 6.28 |
| $(d, w, v, u)$/Balanced Newton | 5213 | 2346 | 402 |

Table 2. Runtimes (in seconds) for rational reconstructions. The 'Old Way' corresponds to the initial implementation of communications with FLINT in FUEL — non-modular functions were called first and then projection of coefficients to the proper modular field were taken. The 'New Way' is the current approach. The use of storing speeds it up things even more.

## 6. CONCLUSIONS

In this paper, we devised a number of various techniques for improving performance of Feynman integral reduction on supercomputers. Cumulatively, with all methods applied, we estimate the resource economy more that twice compared with the version without their implementation. In IBP reductions started from scratch, initial optimization can be achieved by a proper of choice of the order of seeding variables to be restored, as was already discussed in Ref. [21]: the rule of thumb being to starting from a variable which requires less Thiele/balanced Newton BBSs and gradually proceeding to the ones which need more. This is highly problem-specific indeed. In this paper, all runs for our benchmark example were done for the optimal order of variable reconstruction $d \to w \to v \to u$ from the get-go. Future improvements include developing a hybrid technique of balancing and Zippel [34, 35] reconstruction methods, which are particularly suited for sparse polynomials. The achieved performance gains open up possibilities to apply `FIRE` on supercomputers to a number of cutting-edge physical problems including minimal and four-leg form factors of the stress-tensor multiplet, and five- and six-leg scattering amplitudes on the Coulomb branch of the maximally supersymmetric Yang-Mills theory at two loops, just to name a few.

Acknowledgments. A.B. is grateful to Ayush Saurabh and Gil Speyer for the initial introduction to ASU's `Sol` supercomputer and Vladimir A. Smirnov for useful discussions. The work of A.B. was supported by the U.S. National Science Foundation under the grant No. PHY-2207138. The work of A.S., in part of developing improved reconstruction algorithms, 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. 075-15-2022-284. The work of A.S. and V.V., in part of optimizing supercomputer algorithms, was supported by the Russian Science Foundation under Agreement No. 21-71-30003.

The research was carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University [30]. Performance benchmarking was also performed at the Polus supercomputer [29].

## REFERENCES

[1] K. G. Chetyrkin and F. V. Tkachov, "Integration by parts: The algorithm to calculate $\beta$-functions in 4 loops," Nucl. Phys. B 192 (1981), 159-204 doi:10.1016/0550-3213(81)90199-1.

[2] A. V. Smirnov and A. V. Petukhov, "The Number of Master Integrals is Finite," Lett. Math. Phys. 97 (2011), 37-44 doi:10.1007/s11005-010-0450-0 [arXiv:1004.4199 [hep-th]].

[3] S. Laporta, "High precision calculation of multiloop Feynman integrals by difference equations," Int. J. Mod. Phys. A 15 (2000), 5087-5159 doi:10.1142/S0217751X00002159 [arXiv:hep-ph/0102033 [hep-ph]].

[4] C. Anastasiou and A. Lazopoulos, "Automatic integral reduction for higher order perturbative calculations," JHEP 07 (2004), 046 doi:10.1088/1126-6708/2004/07/046 [arXiv:hep-ph/0404258 [hep-ph]].

[5] A. V. Smirnov and M. Zeng, "FIRE 6.5: Feynman Integral Reduction with New Simplification Library," [arXiv:2311.02370 [hep-ph]].

[6] R. N. Lee, "Presenting LiteRed: a tool for the Loop InTEgrals REDuction," [arXiv:1212.2685 [hep-ph]].

[7] J. Klappert, F. Lange, P. Maierhöfer and J. Usovitsch, "Integral reduction with Kira 2.0 and finite field methods," Comput. Phys. Commun. 266 (2021), 108024 doi:10.1016/j.cpc.2021.108024 [arXiv:2008.06494 [hep-ph]].

[8] K. Mokrov, A. Smirnov and M. Zeng, "Rational Function Simplification for Integration-by-Parts Reduction and Beyond," doi:10.26089/NumMet.v24r425 [arXiv:2304.13418 [hep-ph]].
[9] S. He, Z. Li, R. Ma, Z. Wu, Q. Yang and Y. Zhang, "A study of Feynman integrals with uniform transcendental weights and their symbology," JHEP 10 (2022), 165 doi:10.1007/JHEP10(2022)165 [arXiv:2206.04609 [hep-th]].
[10] A. V. Belitsky and V. A. Smirnov, "Near mass-shell double boxes," [arXiv:2312.00641 [hep-th]].
[11] M. Kauers, "Fast solvers for dense linear systems," Nucl. Phys. B Proc. Suppl. 183 (2008), 245-250 doi:10.1016/j.nuclphysbps.2008.09.111.
[12] J. von zur Gathen and J. Gerhard, "Modern Computer Algebra, Modern Computer Algebra", Cambridge University Press (2013).
[13] A. von Manteuffel and R. M. Schabinger, "A novel approach to integration by parts reduction," Phys. Lett. B 744 (2015), 101-104 doi:10.1016/j.physletb.2015.03.029 [arXiv:1406.4513 [hep-ph]].
[14] T. Peraro, "Scattering amplitudes over finite fields and multivariate functional reconstruction," JHEP 12 (2016), 030 doi:10.1007/JHEP12(2016)030 [arXiv:1608.01902 [hep-ph]].
[15] T. Peraro, "FiniteFlow: multivariate functional reconstruction using finite fields and dataflow graphs," JHEP 07 (2019), 031 doi:10.1007/JHEP07(2019)031 [arXiv:1905.08019 [hep-ph]].
[16] J. Klappert and F. Lange, "Reconstructing rational functions with FireFly," Comput. Phys. Commun. 247 (2020), 106951 doi:10.1016/j.cpc.2019.106951 [arXiv:1904.00009 [cs.SC]].
[17] J. Klappert, S. Y. Klein and F. Lange, "Interpolation of dense and sparse rational functions and other improvements in FireFly," Comput. Phys. Commun. 264 (2021), 107968 doi:10.1016/j.cpc.2021.107968 [arXiv:2004.01463 [cs.MS]].
[18] G. Laurentis and D. Maître, "Extracting analytical one-loop amplitudes from numerical evaluations," JHEP 07 (2019), 123 doi:10.1007/JHEP07(2019)123 [arXiv:1904.04067 [hep-ph]].
[19] G. De Laurentis and B. Page, "Ansätze for scattering amplitudes from p-adic numbers and algebraic geometry," JHEP 12 (2022), 140 doi:10.1007/JHEP12(2022)140 [arXiv:2203.04269 [hep-th]].
[20] V. Magerya, "Rational Tracer: a Tool for Faster Rational Function Reconstruction," [arXiv:2211.03572 [physics.data-an]].
[21] A. V. Belitsky, A. V. Smirnov and R. V. Yakovlev, "Balancing act: Multivariate rational reconstruction for IBP," Nucl. Phys. B 993 (2023), 116253 doi:10.1016/j.nuclphysb.2023.116253 [arXiv:2303.02511 [hep-ph]].
[22] A. V. Smirnov, "Algorithm FIRE – Feynman Integral REduction," JHEP 10 (2008), 107 doi:10.1088/1126-6708/2008/10/107 [arXiv:0807.3243 [hep-ph]].
[23] A. V. Smirnov and F. S. Chuharev, "FIRE6: Feynman Integral REduction with Modular Arithmetic," Comput. Phys. Commun. 247 (2020), 106877 doi:10.1016/j.cpc.2019.106877 [arXiv:1901.07808 [hep-ph]].
[24] R. N. Lee, A. V. Smirnov, V. A. Smirnov and M. Steinhauser, "Four-loop quark form factor with quartic fundamental colour factor,"https://www.overleaf.com/project/6568475d383964d33b25dfd3 JHEP 02 (2019), 172 doi:10.1007/JHEP02(2019)172 [arXiv:1901.02898 [hep-ph]].
[25] `Flint`, `https://flintlib.org`.
[26] `Sol`, `https://asurc.atlassian.net/wiki/spaces/RC/pages/1640103978/Sol+Supercomputer`.
[27] K. Mokrov, A. Smirnov and M. Zeng, "Rational Function Simplification for Integration-by-Parts Reduction and Beyond," doi:10.26089/NumMet.v24r425 [arXiv:2304.13418 [hep-ph]].
[28] Claus Fieker, William Hart, Tommy Hofmann, and Fredrik Johansson. Nemo/hecke: computer algebra and number theory packages for the julia programming language. In Proceedings of the 2017 acm on international symposium on symbolic and algebraic computation, pages 157–164, 2017.
[29] `Polus`, `http://hpc.cmc.msu.ru/polus`.
[30] Vl. Voevodin, A. Antonov, D. Nikitenko, P. Shvets, S. Sobolev, I. Sidorov, K. Stefanov, Vad. Voevodin, S. Zhumatiy: Supercomputer `Lomonosov-2`: Large Scale, Deep Monitoring and Fine Analytics for the User Community. In Journal: Supercomputing Frontiers and Innovations, Vol.6, No.2 (2019). pp.4вЂ“11. DOI:10.14529/jsfi190201.
[31] J. Vetter and C. Chambreau, "mpip: Lightweight, scalable mpi profiling", 2005, `http://gec.di.uminho.pt/Discip/MInf/cpd1415/PCP/MPI/mpiP_Lightweight,ScalableMPIProfiling.pdf`.
[32] A. Yasin, "A Top-Down method for performance analysis and counters architecture," ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software. 2014, 35-44. doi:10.1109/ISPASS.2014.6844459.
[33] `Top-down Microarchitecture Analysis Method`, `https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookb`

[34] R. Zippel, “Probabilistic algorithms for sparse polynomials,” in: Ng, E.W. (eds) Symbolic and Algebraic Computation. EUROSAM 1979. Lecture Notes in Computer Science, vol. 72, Springer. https://doi.org/10.1007/3-540-09519-5_73.

[35] R. Zippel, "Interpolating polynomials from their values," Journal of Symbolic Computation 9 (1990), 375-403.