Performance optimization for LHCb experiment software

Florian Lemaitre

August 31, 2015

Company :	CERN
Dates :	from april to august 2015
Supervisor :	Benjamin Couturier

Acknowledgments

I would like to take the opportunity to thank all people who have contributed in some way to this internship, especially Pr Lionel Lacassagne who presented this project to me and was always here when needed.

I also want to thank Mr Benjamin Couturier who supervised my work. His advice was really useful and appreciated.

My thanks also go to Dr Yasmine Amhis and Mr Renato Quagliano with whom I spent a lot of time discussing about the studied algorithm.

I would like to thank Dr Manuel Schiller who took from his time to explain the algorithm among other things to me.

I also thank all the member of PH-LBC department who welcomed me within, as well as CERN for allowing me to work here for five months.

Finally, I would like to thank Pr Fabrice Popineau who took time to check up on me.

Table of Contents

List of Tables

List of charts

List of Figures

Listings

Introduction

CERN, European Organization for Nuclear Research, is a high energy physics laboratory where experiments are run to study the properties of the sub-atomic world. The LHC, Large Hadron Collider, is the biggest particle accelerator in the world. It allowed CERN to discover the Brout-Henglert-Higgs Boson in 2012.

Colossal amounts of data are recorded by the LHC experiments (O(petabytes)), and analyzing this data is very computationally intensive: thousands of computing nodes are required to filter and reconstruct the particle tracks. For this reason, optimized code that takes full advantage of the hardware is critical and allows maximizing the physics that can be computed with the infrastructure available.

This project focused on the profiling and optimization of a specific algorithm used in the reconstruction of data recorded by the SciFi tracker planned for the LHCb upgrade.

The idea for optimizing the code that way is to write plain C or C++ in such a way that the compiler is able to do all the low level optimizations, especially the vectorization. Vectorization is a really important optimization because it allows to do upto eight times the same operation on contiguous data for current processors, and even sixteen times on some processors.

The study is split in three parts:

- The first step is to rewrite the actual algorithm with C++ optimizations and to change the algorithm workflow in order to easily introduce high level transformations.
- The second step is the analysis of high level transformations and their performance behavior. Especially, some transformations help the vectorization, some allow better cache use:.
- The last step is the numerical stability analysis of a part of the algorithm. This step allows to validate some transformations resulting in less accurate results.

I Presentation

I.1 CERN

CERN, the European Organization for Nuclear Research, has been probing the fundamental structure of the universe for 61 years. Established in 1954 on Franco-Swiss border and with 22 member states, CERN has become the largest particle physics laboratory in the world. Now, about 10 000 physicists and engineers from the whole world work together to study sub-atomic physics.

An interesting fact about CERN is that it gave birth to the Wold Wide Web in the 90's, thanks to a project based on the concept of hypertext. The first website was activated in 1991 and on April 30th, CERN announced that the World Wide Web would be free to anyone.

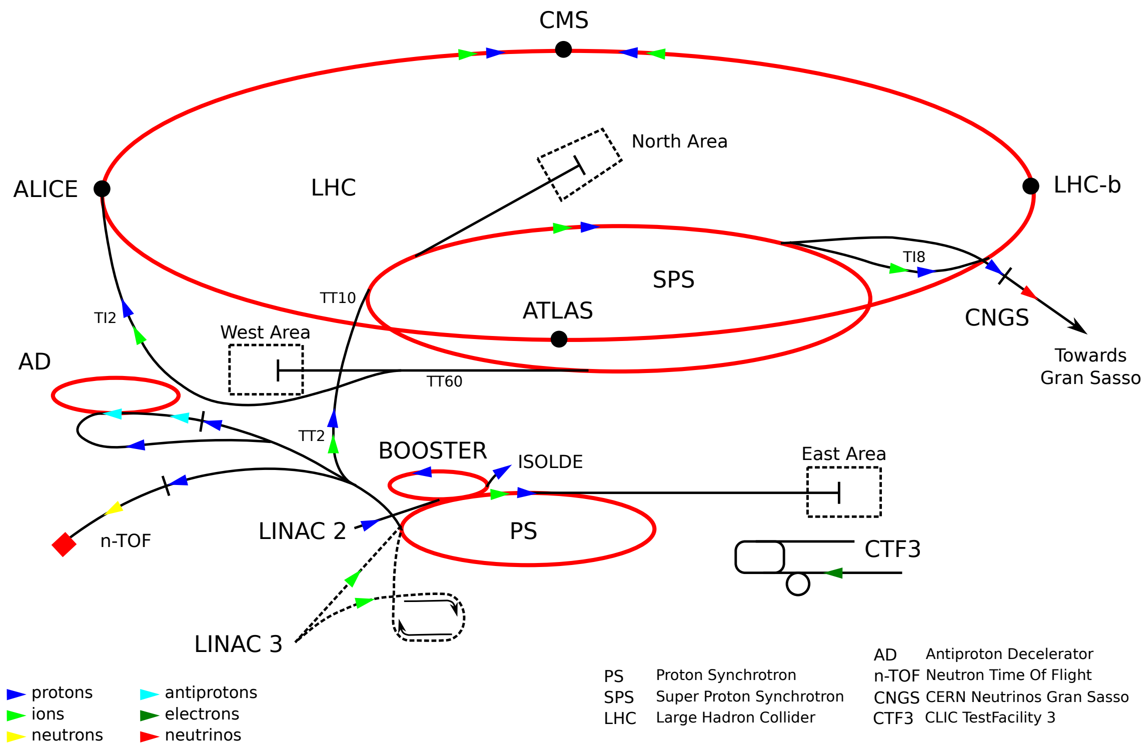

Figure 1: Map of the CERN accelerator complex

I.1.a LHC

In order to study the fundamental particles, what the universe is made of, CERN has created the most complex scientific instrument: the LHC. The LHC, the Large Hadron Collider, is the biggest and the most powerful particle accelerator. Built between 1998 and 2008, it lies in a 27 kilometer long tunnel, as deep as 175 meters underground. It allowed CERN to make majors discoveries in high energy physics, like the discovery of the last part of the Standard Model in 2012: the Brout-Henglert-Higgs Boson. The LHC is split onto 4 main experiments: Alice (A Large Ion Collider Experiment), Atlas (A Toroidal LHC Apparatus), CMS (Compact Muon Solenoid) and LHCb (LHC-Beauty). Each experiment consists of a collision point between the two beams of the LHC, a detector which detects the particles generated by the collisions, and a computer farm (thousands of machines) to filter and analyze the output data from the detector.

I.1.b LHCb

LHCb is one of the 4 main experiments of the LHC and is designed to investigate the difference between the matter and the anti-matter. It especially tries to answer the question of the missing anti-matter in the universe by analyzing the fundamental properties of the anti-matter.

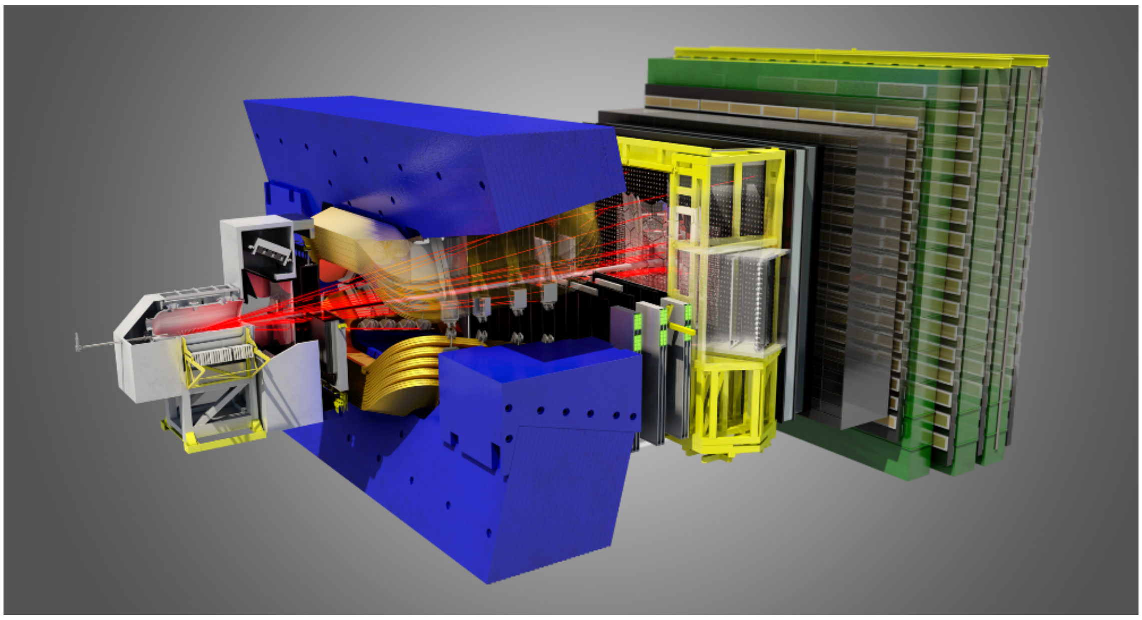

Figure 2: LHCb detector

The detector consists of a big magnet and a bunch of sub-detectors:

VeLo: The Vertex Locator is a tiny but very accurate particle detector. It's as near as possible from the collision.

RICH1: The first Ring Imaging Cherenkov detector is just after the VeLo and used to identify low-momentum particles

Tracker Turicensis: A tracker, ie a track detector before the magnet. It uses 2 stations.

Main Tracker: Another tracker with 3 stations just after the magnet. Composed with a straw-tube based detector and a silicon strip detector in the center.

RICH2: The second Ring Imaging Cherenkov detector used to measure momemtum of the high-momentum tracks.

Calorimeters: The calorimeters are after the main Tracker and RICH2. It allows to measure the energy of high-momentum tracks. It is also used as a trigger for high-momentum particles.

Muon Chamber: The muon detector is used to trigger muon events.

The LHCb experiment detects 40 000 000 collisions per second. This produces tremendous amount of data, and cannot be analyzed or even stored so fast. So just at the output of the detector, there is the Low Level Trigger (LLT) composed of about 350 FPGAs. Then, filtered data are forwarded to the High Level Trigger (HLT) composed of about 2000 servers distributed among 6 farms. The software of the HLT is split into two steps (HLT1 and HLT2). The first step is about reconstructing particles tracks while the second one is about merging together all the information in order to reconstruct the whole event. The incoming data are stored in a buffer before being computed. So considering the input event rate and the pauses between collisions batches, a single thread of the HLT1 needs to perform its computation in about 30ms. These three triggers filter the event and keep only interesting events according to physical criteria they have. This part is called Online and is done in real time underground, close to the detector.

The data is then send to the surface where it is stored on magnetic tape and then computed by the Grid: a global network of computing nodes which analyze events in more detail. This part is called Offline.

The scientific code is written in C++, and an internal framework called Gaudi is used to write algorithm.

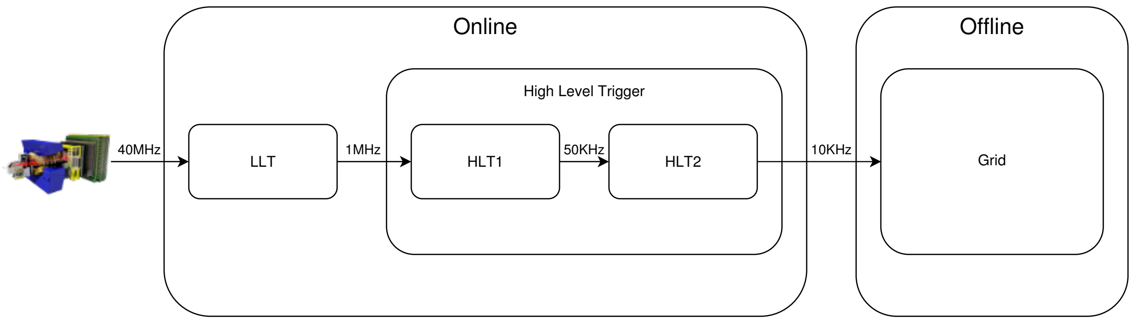

Figure 3: Current LHCb Dataflow

For the next run of the LHC, it is planned to upgrade the LHCb detector, especially the main Tracker system. The new Tracker is called SciFi (for Scintillating Fiber Tracker). The upgrade will also change the Online dataflow. Indeed, it is planned to remove the electronic LLT, and only have the HLT. This imply that the first step of the HLT needs to compute about 40 times more events per second. So the performance of this first part is really crucial.

I.2 Objectives

The main goal of the project is to improve the performance of the Tracking algorithm for the SciFi tracker (HLT1). The idea is not so much to reduce the computation time of one single event, but more to increase the number of events computed per seconds: get a higher total throughput.

Thereby, some optimizations are less interesting here than in High Performance Computing like multi-threading the code with OpenMP for example. Indeed, it is easy to run several instances of the same program to compute different events with almost the same improvement than with the multi-threaded version of the program.

However, this is not the case for the vectorization because it uses a parallelism we cannot achieve otherwise. In fact, running a vectorized thread does not slow down the other threads and processes running on the same machine. Moreover, vectorization allows to get a high parallelism, thereby a much higher throughput. With a suitable algorithm, we can envision to achieve a speed-up close to ×4 with *SSE* and even close to ×8 with *AVX*. This is the reason why the project mainly considers the vectorization optimization.

The final goal of this project is not quite just improve the performance of this algorithm but find a way to write simple code that can benefit from these improvements. In other words: writing almost plain C or C++, and letting the compiler doing all the optimizations for us, especially the vectorization. So taking the current code, making the required high level transformations and letting the compiler do the low level optimizations. High level transformations are transforms beyond the scope of an optimizing compiler for better general purpose processor (GPP) effciency. To address intrinsic processor parallelism (*SIMD* and multicore) and specificity (memory cache hierarchy).

II Algorithm Transformations

The first part of my internship was to rewrite the implementation of the Gaudi algorithm in order to identify the bottlenecks. It first appeared that the code spent a lot of time with I/O. So I removed all the messages printing from the code. It then appeared that the code spend a lot of time iterating and not computing the loops body.

After that, it was decided to write the algorithm so that the future modifications, especially the high level transformations, would be much simpler to do. The main idea was to split the nested loops in multiple almost flat loops, putting the intermediate results in arrays. With this, it would be simpler for the compiler to vectorize loops as they become straight-forward, and with inner-most computation.

This part could not be continued without analyzing high level transformations. That is why I did not go further on rewriting the algorithm.

III High level transformations comparison

The problem considered is the parabola solving algorithm using the Cramer's rule (cf: Listing 7, `solveParabola` function):

- input: 3 points from the plane (XZ) (6 numbers)
- output: 3 parabola parameters (a, b, c)

Beside it is not a big part of the tracking algorithm, it is simple (it is just calculus) and is vectorization friendly.

In fact, the algorithm is not exactly a parabola solver but a cubic solver with an already fixed parameter. The fixed parameter is the cubic term and is called `dRatio`. If `dRatio` equals 0, we are in the parabolic case, but in this study, `dRatio` is a constant and is not equal to zero.

This algorithm needs at least (cf: Listing 8, `solveParabola` function (no operation duplication)):

- add/sub: 26
- multiplication: 36
- division: 1
- absolute value: 1
- comparison: 1
- branch: 1

- load: 6
- store: 3

We have an arithmetic intensity pretty high: $AI = \frac{nb\ operations}{nb\ loads{+}stores} = 7.22$. With an AI of 7.22, the problem may be memory bound. But it is still an arithmetic intensive problem, so we can envision that the I/Os do not slow-down too much the computation. Vectorization can thereby bring a really interesting point to our project.

This function is called on each element of an array of 3 points (input) and 3 parabolic parameters (output). The input values are directly extracted from Gaudi in order to have real data for the test. The input data are stored in one big array and not 3 little arrays because the Gaudi algorithm already does this part and computes some search windows in order to decrease the number of parabolas to create. So the big input array has less elements than the array resulting from the Cartesian product of the 3 initial arrays.

Typically, the big array contains about 30 000 elements. It is pretty good because it can be entirely stored in the Ł3 cache of the processor. Even if the computation is straightforward, it is relevant to consider it because the data shall already be in cache because they are computed just before.

III.1 Measuring time

In order to have meaningful measures, it is important to have the distribution of the time spent by the function. All the measurements have been done with the *icc* primitive `_rdtsc()`.

When looking at the performance distributions of our function, it is interesting to notice that the distributions have not just one peak but several. Indeed, with $N = 10\,593$, we can see a big peak, and a little one further, for example. At this point, it is really important to understand what happens.

We can see that the performance is almost constant in time, but a bit slower about every 10 iterations (1.4% slower), and even slower about every 66 iterations (11.7% slower). The reason is as simple as inconvenient: when an interruption occurs, it temporary freezes the execution of our program, but still continues to increment the cycles counter of the core. Thereby, every time an interruption occurs, even hardware interruptions, we lose a high number of cycles. These interruptions can be, for example, the task scheduler, the kernel timer, disk activity, network activity... In fact, the little peaks are spaced by about 1 ms, which is exactly the kernel timer latency on a common x86-64 system.

The bigger the data is, the longer the execution is, and the more probable an interruption occurs during the execution of the function. Thereby, the slow-down is more frequent, so, the second peak is higher. As the interruptions give constant penalties, the bigger the data is, the more negligible the slow down is. Consequently, the second peak goes closer to the first.

After a while, the first peak disappears and lets the place for the second which becomes first. That is exactly what we can see on the previous distributions:

- between 51 093 and 75 717, the second peak becomes bigger than the first one: the associated interruption occurs more than half time.
- between 92 175 and 112 210, the first peak disappears: the associated interruption now occurs every time.

Another interesting point is that the bigger the data is, the thicker the peaks are. So the deviance decreases when the data size increases because the stochastic events occurring during the measurement and slowing down the function are almost constant, so their impact is becoming negligible when the data is big: the time spent in the function is divided by the data size.

At this point, the best way to measure would be to take the most probable value: the maximum of the distribution. Doing like that allows to just ignore extra peaks and just keep one: the biggest. However, this can behave badly when two or more peaks are about the same size. Indeed, in such a situation, the measure may not be predictable and may randomly oscillate between several values. This can be solved by taking the maximum of the first peak. The measure is now reproducible and is still meaningful because it corresponds on the most probable value if the probable slowing down event does not occur.

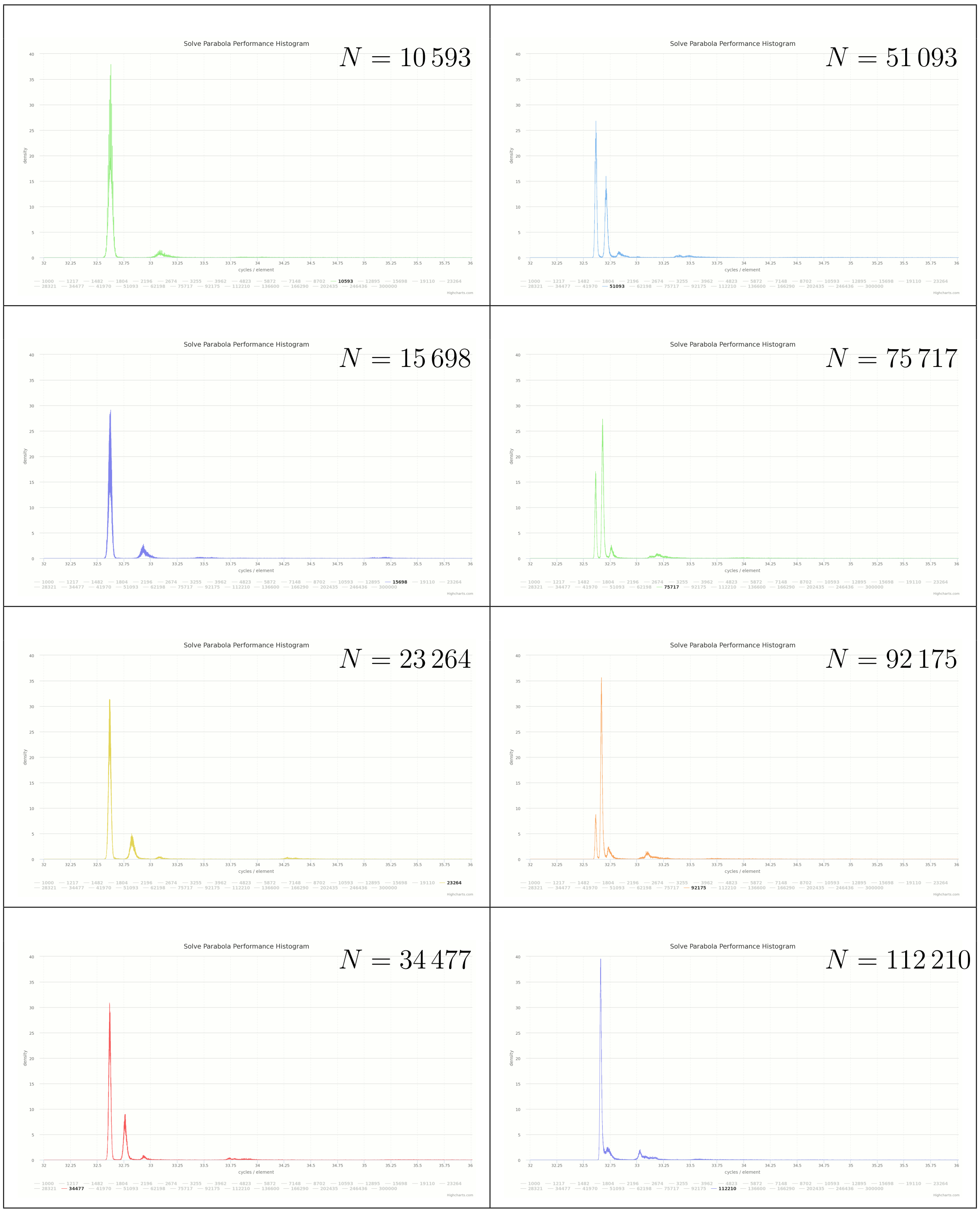

Chart 1: Performance distribution of `solveParabola` for *SoA* data layout depending on data size

The main problem here is the time needed to construct a distribution accurate enough to a maximum. For instance, every plot has been done executing 100 000 times the function. Just to find accurately the maximum, we do not need 100 000 execution, but we need far more than only few dozens.

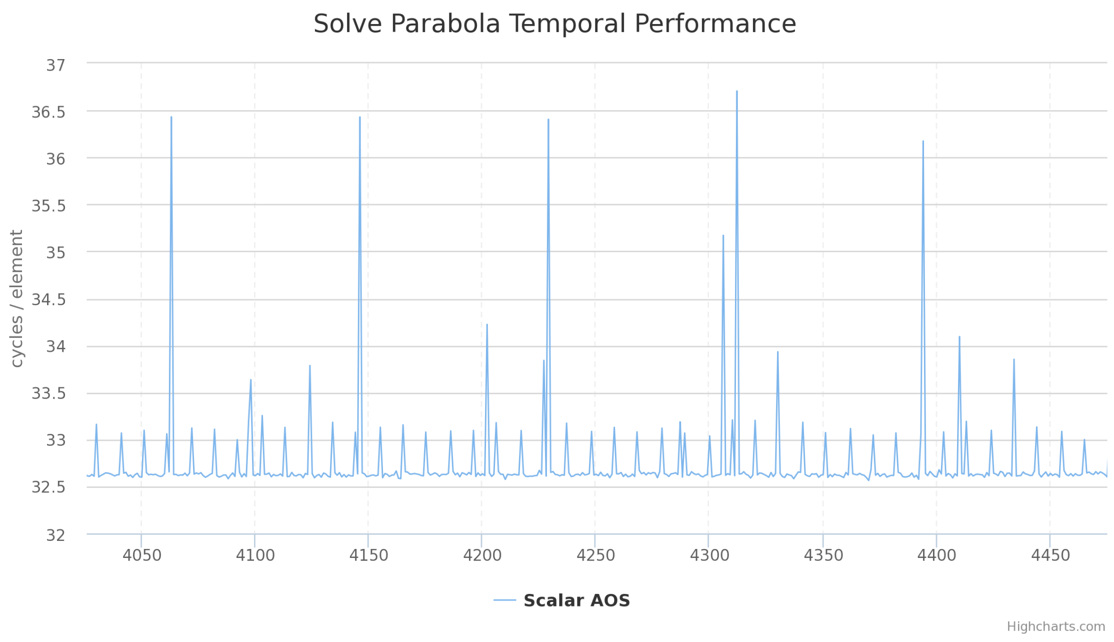

Chart 2: Performance of `solveParabola` depending on time ($N = 10\,593$)

Consequently, we chose to take the minimum of all the executions because it is quick to compute and it also ignores the extra peaks like the previous method, but requires only few executions. For the rest of the study, we mainly do 32 executions which gives very accurate results, especially for big enough data. In fact, the minimum converges pretty quickly and with less than 10 executions, we are able to reach close results. But the more, the safer.

The minimum gives good results especially because the deviance of the peaks is pretty low. Moreover, this gives to us reliable and reproducible results. The mean and the median are not reliable values because they consider the extra peaks which are not dependent on the studied function but only dependent on the machine and the operating system.

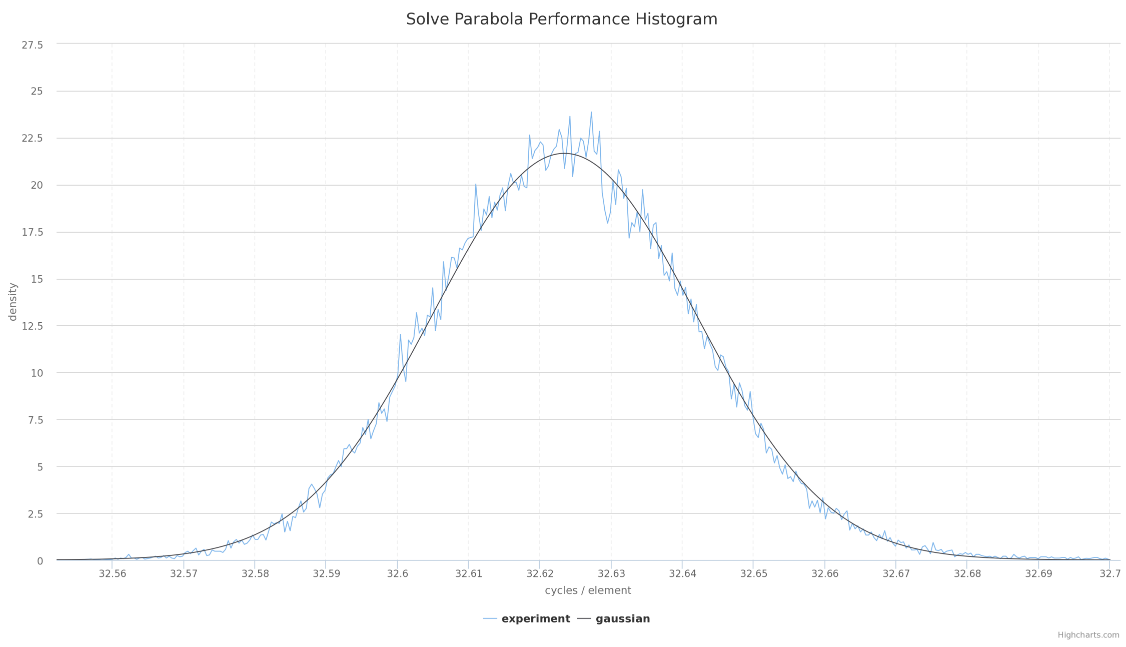

Chart 3: Isolated Gaussian peak from the performance distribution ($N = 10\,593$)

III.2 Memory Layouts definition

III.2..i Array of Structures (*AoS*)

The *AoS* layout is the simplest to implement in C. It is indeed the most natural way. This memory layout has the advantage to only need one single pointer. It allows to reduce systematic cache eviction (when try to caching addresses with the same last bits).

```c
struct Hit  { float x, z; };
struct Param{ float a, b; };
typedef struct {
  struct Hit hit;
  struct Param param;
} AOS[N];
```

Listing 1: example of *AoS* structure

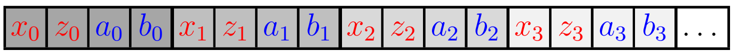
x_0 z_0 a_0 b_0 x_1 z_1 a_1 b_1 x_2 z_2 a_2 b_2 x_3 z_3 a_3 b_3 ...

Figure 4: *AoS* memory layout

III.2..ii Structure of Array (*SoA*)

The *SoA* layout consists in storing each attribute in a separate array. In fact, it is the default memory layout in Fortran 77. It helps the vectorization because we can load a bunch of the same parameter in a contiguous way. It may sometimes decrease the number of cache misses if the structure is heavy and not entirely used in loops (for example: a structure with about 100 attributes, but only 3 are used in loop). However, this memory layout usually increases systematic cache eviction.

```c
typedef struct {
  float x[N], z[N];
  float a[N], b[N];
} SOA;
```

Listing 2: example of *SoA* structure

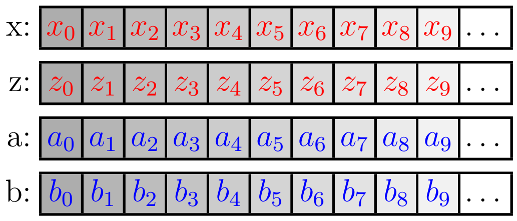
x: x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 ...
z: z_0 z_1 z_2 z_3 z_4 z_5 z_6 z_7 z_8 z_9 ...
a: a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 ...
b: b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 ...

Figure 5: *SoA* memory layout

III.2..iii Array of Structures of Arrays (*AoSoA*)

The *AoSoA* layout is a hybrid between *AoS* and *SoA*: it stores the attributes in little arrays and packs these arrays in a larger array. With this layout, we can load attributes by pack, so if the size of the pack is correct, we can vectorize the code exactly as with the *SoA* layout as explained in [6], even hiding the *SIMD* part from the programer with subtile C++ [2] However, we keep the contiguous memory we have with the *AoS*. So like *AoS*, this memory layout helps reducing systematic cache eviction, but not as well as *AoS*. A good size for the little arrays is the size of the *SIMD* registers (4 in *SSE*, 8 in *AVX*).

```c
typedef struct {
  float x[k], z[k];
  float a[k], b[k];
} AOSOA[N/k];
```

Listing 3: example of *AoSoA* structure

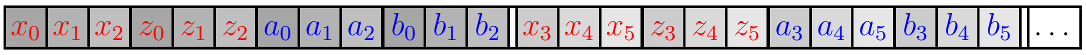
x_0 x_1 x_2 z_0 z_1 z_2 a_0 a_1 a_2 b_0 b_1 b_2 x_3 x_4 x_5 z_3 z_4 z_5 a_3 a_4 a_5 b_3 b_4 b_5 ...

Figure 6: *AoSoA* memory layout ($k = 3$)

III.2..iv Structure of Arrays of Structures (*SoAoS*)

The idea is to have every structures of the main structure inside different arrays. It does not really help the compiler to vectorize the code, but may have an impact on the cache in the same way as *SoA* in a smaller way. This memory layout has been tried only to have exhaustive memory layout set for the analysis.

```c
struct Hit  { float x, z; };
struct Param{ float a, b; };
typedef struct {
  struct Hit hit[N];
  struct Param param[N];
} SOAOS;
```

Listing 4: example of *SoAoS* structure

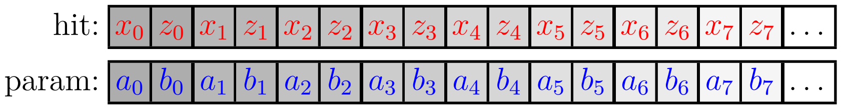
hit:
x_0 z_0 x_1 z_1 x_2 z_2 x_3 z_3 x_4 z_4 x_5 z_5 x_6 z_6 x_7 z_7 ...
param:
a_0 b_0 a_1 b_1 a_2 b_2 a_3 b_3 a_4 b_4 a_5 b_5 a_6 b_6 a_7 b_7 ...

Figure 7: *SoAoS* memory layout

III.3 Memory Layouts comparison

III.3.a Fixed size Results

Scalar				Vector				Intrinsics
AoS	*SoAoS*	*SoA*	*AoSoA*	*AoS*	*SoAoS*	*SoA*	*AoSoA*	*SoA*
32.59	33.06	30.96	30.74	18.95	20.02	5.67	5.95	4.51

Table 1: Performance of `solveParabola` in cycles/element ($N = 50\,000$)

Looking at the speed-ups table, we can see that there is almost no difference between *AoS* and *SoAoS* and also between *SoA* and *AoSoA*. Even better, for the vectorized version, we can see: the simpler, the better. We can also see that all the scalar versions are almost as fast as the others. Thus, it is meaningful to compare our results to the *AoS* scalar version of the algorithm.

About the vectorization speedup, we can see a 50% gain for *AoS* vectorized version over the scalar one. This is pretty bad as we use *AVX* with its 8 float-wide registers, but it is completely normal: the *AoS* memory layout prevents packed loads and stores. We need thereby 8 times more of loads and stores which slows down the program. But the other part of the computation is straightforward and easily vectorizable. This is why we can observe a speedup despite the high number of memory access.

Speed-ups		**Scalar**				**Vector**				**Intrinsics**
		AoS	*SoAoS*	*SoA*	*AoSoA*	*AoS*	*SoAoS*	*SoA*	*AoSoA*	*SoA*
Scalar	*AoS*	1	1.01	0.95	0.94	0.58	0.61	0.17	0.18	0.14
	SoAoS	0.99	1	0.94	0.93	0.57	0.61	0.17	0.18	0.14
	SoA	1.05	1.07	1	0.99	0.61	0.65	0.18	0.19	0.15
	AoSoA	1.06	1.08	1.01	1	0.62	0.65	0.18	0.19	0.15
Vector	*AoS*	1.72	1.74	1.63	1.62	1	1.06	0.30	0.31	0.24
	SoAoS	1.63	1.65	1.55	1.54	0.95	1	0.28	0.30	0.23
	SoA	5.75	5.83	5.46	5.42	3.34	3.53	1	1.05	0.79
	AoSoA	5.48	5.56	5.21	5.17	3.19	3.37	0.95	1	0.76
SIMD	*SoA*	7.23	7.33	6.87	6.82	4.21	4.44	1.26	1.32	1

Table 2: Speed-ups for `solveParabola` ($N = 50\,000$)

Regarding the *SoA* version, we can fully benefit from vectorization. However, while we don't have penalties with the loads and stores, we can see that we don't reach the expected speedup: with the intrinsics version, we have a speedup of 7.2 instead of having a speedup of 8 (*AVX* has 8 float-wide registers).

There are two problems: we use too many registers and there are a lot of dependencies between operations (cf: Chart 13, `solveParabola` dependency flow chart). Thus, the computation of one iteration cannot be shorter because of the dependencies, and we can't unroll the loop in order to overlap iterations because we run out of registers. In fact, we use so many registers that we can't store every intermediate computation in registers. We need to spill some variables: put some variables in memory instead of in registers. The compiler chooses to spill the constants "variables": it reduces the chance to have a cache miss as they have constant address.

An important point has to be highlighted: there is speedup of 25% for the intrinsics code over the auto-vectorized code (*SoA* version). After investigations, we can see 3 reasons:

- Branch management
- Slower division
- Variable Spilling

III.3.a.i Branch Management

The main difference in the branching between the intrinsics code and the auto-vectorized code is the following.

The intrinsics code does not implement a branch, but just compute the value, and mask the result assignment. This way to do is really fast if the branch is almost never taken. And that is precisely the case with the given data.

On the other hand, the auto-vectorized version implements a real branch. But the branching in *SIMD* is quite tricky to do, so there is a quite important overhead doing this branching mechanism. But this version is faster when the branch is always taken.

Branching	Auto-vectorization	intrinsics	speed-up	reciprocal speed-up
no branch	$4.41\,c/el$	$4.34\,c/el$	1.8%	−1.8%
never taken	$5.64\,c/el$	$4.53\,c/el$	24.3%	−19.5%
always taken	$3.40\,c/el$	$4.54\,c/el$	−25.2%	33.6%

Table 3: Branching influence on `solveParabola` performance ($N = 50\,000$)

III.3.a.ii Division

When I wrote my intrinsics version of the code, I firstly used the `div` function in order to compute the reciprocal of `det`. When analyzing the auto-vectorized code produced by *icc*, I saw that *icc* uses the `rcp` function (fast computes an approximation of the reciprocal). I tried to use this function which gave me faster results, but with a greater Root Mean Square error.

Surprisingly, the auto-vectorized version does not suffer of this great error: *icc* uses a trick to compensate the error due to the fast computation. This trick allows to compute the reciprocal with a good accuracy faster than with the instruction `divps`:

- `divps`: 21 cycles latency (1 division unit)
- `rcpps`: 7 cycles latency (1 division unit)
- corrected `rcpps`: 17 cycles latency (1 division unit, 1 addition unit, 1 multiplication unit, 1 *Fused Multiply-Add (FMA)* unit)

```
__m256 reciprocal_ps(__m256 x) { // 17 cycles
  __m256 rcp_approx, one_comp,  rcp_approx2;

  // Computes the reciprocal approximation (7 cycles)
  rcp_approx = _mm256_rcp_ps(x);

  // computes the error in order to compensate it (5 cycles)
  one_comp = _mm256_mul_ps(x, rcp_approx); // 5c
  rcp_approx2 = _mm256_add_ps(rcp_approx, rcp_approx); // 3c
  // The addition and the multiplication are computed
  // in parallel because there is no dependence between them

  // Compensates the error (5 cycles)
  return _mm256_fnmadd_ps(one_comp, rcp_approx, rcp_approx2);
}
```

Listing 5: Fast and corrected reciprocal

However, using this trick slows down the program: this is because it uses some addition and multiplication units that we need for other computation we can do in parallel. Thus, using the `divps` instruction allows to do more computation in parallel of the division that masks quite well the division latency (not entirely because using only `rcpps` is still faster).

III.3.a.iii Variable Spilling

The last difference I could see between the intrinsics version and the auto-vectorized one is about variable spilling. Variable spilling consists in storing some register variables in memory to free some registers and then load the variable from memory when needed.

When looking at the assembly (cf: Appendix D, `solveParabola` loop assembly dumps), we can see that the auto-vectorized version spills more variables than the intrinsics one and this mechanism takes time: load and store can be done in 4 cycles, which is slower than already having the value in a register.

# Variables spilled	Auto-Vectorization	Intrinsics
Constants	7	5
Variables	8	1

Table 4: Number of spilled variables

However, this points seems to impact the performance by less than 3%.

III.3.b Size Dependency

III.3.b.i Register Size Influence

For small data, we can see a regular saw-tooth pattern. The period of the pattern is 8, the size of the *AVX* registers. In fact, it is completely normal: when the size cannot be divided by 8, you must compute the last elements in another way. The simplest way is to have another non-vectorized loop after the main and vectorized loop. With this trick, it is easy to compute the remaining elements.

It is the way chosen by *icc* to write the remainder loop. Thus, when we add one element, it takes one more non-vectorized iteration. When we reach 8 non-vectorized iterations, we can pack those into one vectorized iteration, and the time taken by one element is thereby decreased.

However, it is not the only way to do it. What I've used to do the remainder "loop" is to copy the vectorized loop body, but instead of storing a whole *SIMD* register, I store only the extra values. It is pretty simple to do, thanks to the `maskmovps` instruction. With this technique, we don't have the overhead of a scalar loop, but we do have a much smaller overhead due to the mask computation. The time does not increase if we add an element and if there is enough place in the remaining loop iteration, and the time per element decreases.

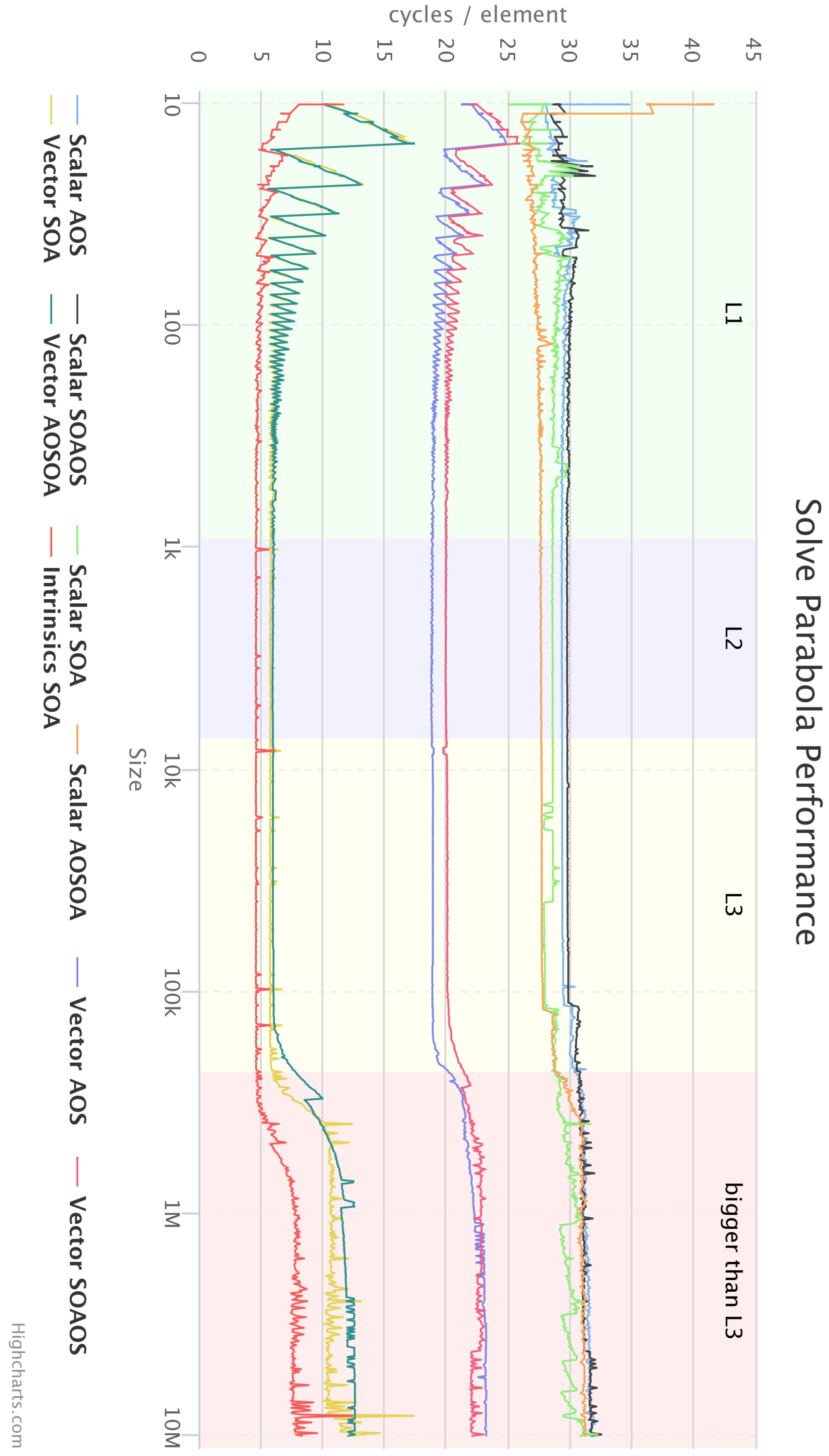

Chart 4: Performance of `solveParabola` depending on data size

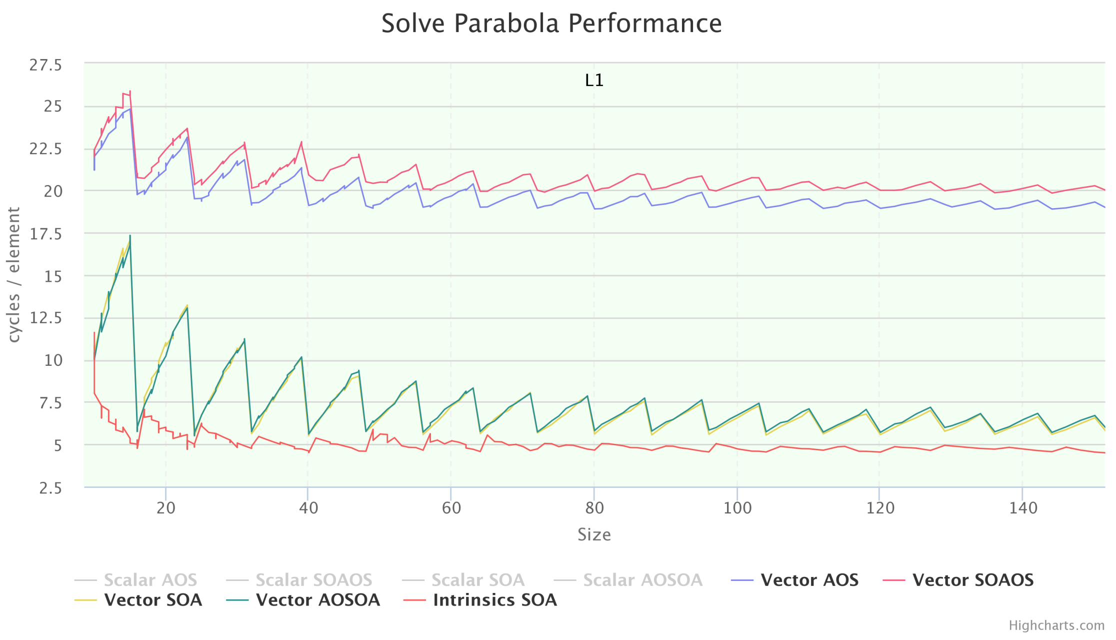

Chart 5: Performance of `solveParabola` depending on data size (small data)

The bigger the data is, the lower the tooth are. This is because the overhead of the remainder loop is amortized by the time the vectorized loop takes, that becomes much more than the remainder. In other words, the main loop takes a linear time while the remainder takes almost "constant" time.

III.3.b.ii Cache Influence

We can clearly see the point when the cache is not sufficient anymore. However, we can also see there is no penalty to have data larger than the *L1* or *L2* cache size as long as the data is smaller than *L3* cache. This could be explained by the fact that the problem is compute bound, we do many more operations than load and store. But it is not sufficient to be masked when data does not fit within the *L3* cache.

When the data is too big for *L3* cache, we can see a drop in the speed by almost a factor 2 on the *SoA* vectorized versions. But on the scalar version, the drop is hardly noticeable because of the nature of this very problem ($IA = 7.2$). Thus, cache misses are hidden by the computation.

III.3.b.iii Power of 2 Alignment

For the *SoA* versions of the code (both auto-vectorized and intrinsics), we can see a strange but well known behavior: when the data size is a power of 2 (or a multiple of a great power of 2), we can see performance drops.

The problem comes from the cache behavior. Indeed, the cache looks at the *Least Significant Bit (LSB)* to store the value in the cache. But if two addresses have the same *LSB*, there is a conflict and even if there is enough place in the cache, you can't have the

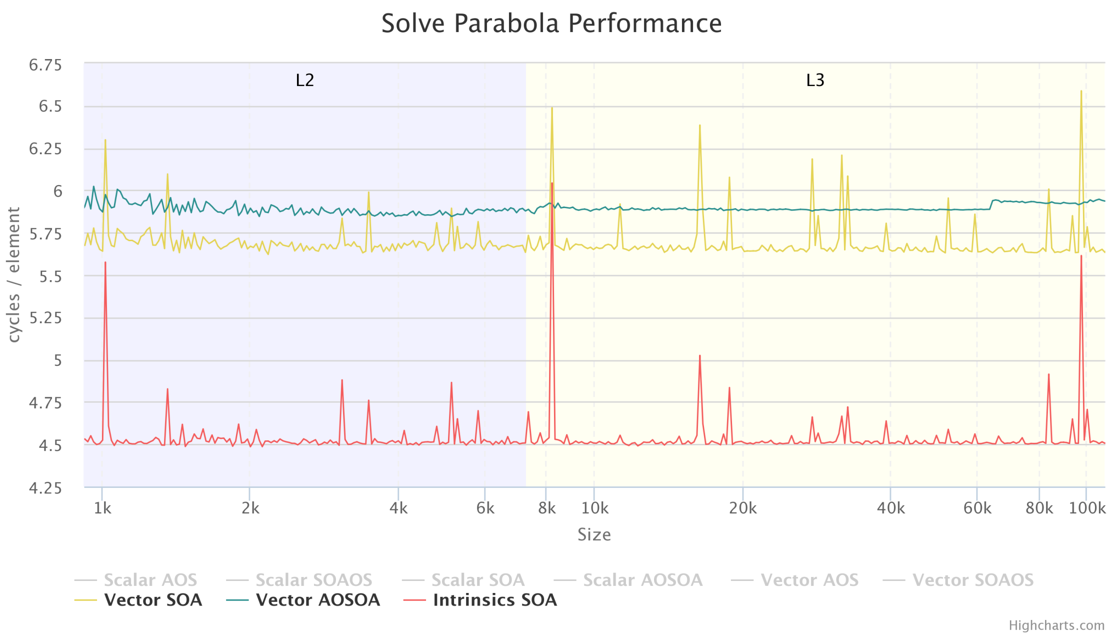

Chart 6: Performance of `solveParabola` depending on data size (power of 2)

2 values in the cache at the same time. Current caches can handle better those cases by allowing to store for examples 16 values with the same *LSB*, but the problem remains if we have a lot of values whose addresses have the same *LSB*.

When you have multiple arrays aligned with a great power of 2, the addresses will have the same *LSB* pretty often. This is called systematic cache eviction. Thus, we will have cache conflict often enough to have an impact on the performance.

With the *SoA* memory layout, we have several arrays, and when we allocate them, the arrays may be following the others in memory. So if the data size is a great power of 2, all arrays may have the same alignment on that power of 2, and cache conflicts may occur between all the arrays.

An interesting point is that we don't have the problem with the *AoSoA* memory layout because this memory layout have only one big array, so the addresses of the attributes don't share their *LSB*. So we can avoid cache conflict using different memory layout.

III.4 Thread influence

III.4.a Fixed size results

$N = 50\,000$		computation time (c/el)			speed-up to ref		speed-up to 1 core		Per core speed-up to 1 core	
		1 core	4 cores	8 threads	1 core	best	4 cores	8 threads	4 cores	8 threads
Scalar	*AoS*	32.59	8.21	7.52	1	4.33	3.97	4.33	99.2%	108.3%
	SoAoS	33.06	8.34	7.59	0.99	4.29	3.97	4.36	99.1%	108.9%
	SoA	30.96	7.78	7.06	1.05	4.62	3.98	4.38	99.5%	109.6%
	AoSoA	30.74	7.74	7.02	1.06	4.65	3.97	4.38	99.2%	109.5%
Vector	*AoS*	18.95	5.17	4.55	1.72	7.16	3.66	4.17	91.6%	104.2%
	SoAoS	20.02	5.09	4.58	1.63	7.12	3.93	4.38	98.4%	109.4%
	SoA	5.67	1.48	1.38	5.75	23.66	3.83	4.12	95.7%	102.9%
	AoSoA	5.95	1.60	1.42	5.48	22.92	3.72	4.18	93.0%	104.5%
SIMD SoA		4.51	1.21	1.11	7.23	29.28	3.74	4.05	93.5%	101.3%

Table 5: Speed-ups for `solveParabola` with OpenMP ($N = 50\,000$)

Looking at multi-threaded `solveParabola` speed-ups, we can see 2 important results. The first one is the speedup to one core: it is almost always close to 4, especially for scalar versions. It means that the threaded version is quite good because our machine has exactly 4 cores. The vectorized versions seem to gain a bit less than the scalar ones, but this can be easily explained by the higher bandwidth: indeed, these versions are very fast, and it is mainly limited by the global bandwidth.

The other important result is about the hyper-threading. We can see an increase in the performance about 10%. It is pretty low compared to some algorithms with which we can reach a 50% improvement. This can be explain by both the limited bandwidth and the lack of unused executive units. Indeed, the algorithm uses a lot of execution units, so the hyper-threaded thread have not enough units to run fast.

III.4.b Size dependency

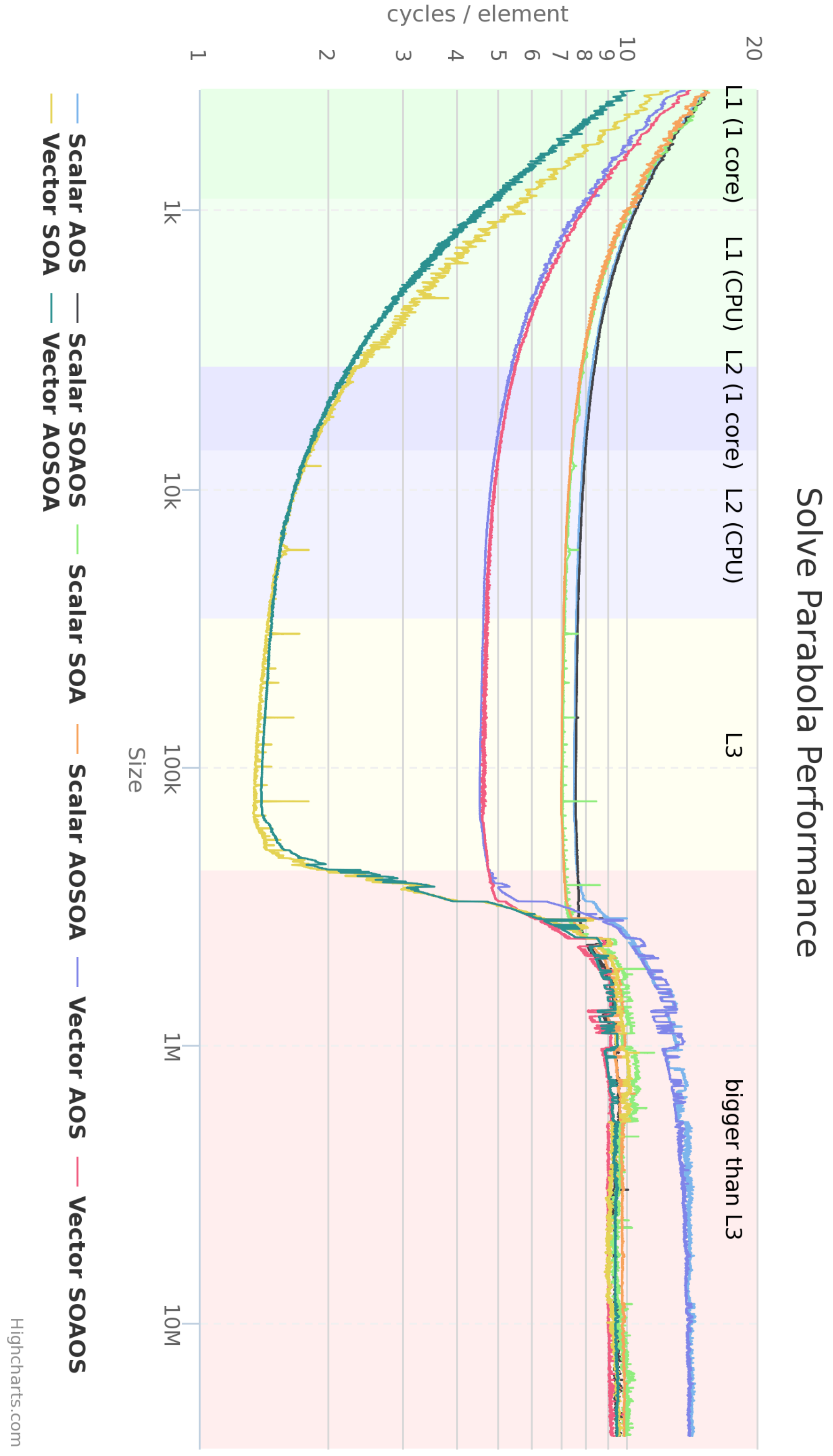

Chart 7: Performance of `solveParabola` depending on data size (OpenMP)

III.4.b.i OpenMP overhead

The main difference between the OpenMP version and the mono-threaded one is the presence of a constant overhead due to OpenMP. This overhead is due to the thread management including their creation, their feeding and their synchronization. As this overhead is almost constant, it is non-negligible only for "small" data. For big data, the increase in performance completely masks the overhead.

We can clearly see this at beginning of the previous chart: computation time is very high for small data and quickly drops and reaches its nominal speed.

However, we can see that we do not really reach the nominal speed for several functions like Vector *SoA*: the data does not fit in the *L3* cache before. Indeed, the computation time per element is still decreasing a bit before. We are close to it, but we don't reach it.

Another relevant point is the following: OpenMP becomes interesting very late and is really not interesting for small data. Indeed, the OpenMP is faster than the mono-threaded version only for data bigger than 1000 elements. Before this point, the overhead is too big.

III.4.b.ii Cache influence and power of 2 alignment

We can clearly see exactly the same behavior about the cache than in the mono-threaded version. No surprise here.

However, what we can see is that for the fast codes, the speed when data is too big is almost the same than in the mono-threaded version. That means we reach the memory bandwidth between the *L3* cache and the memory (RAM). Consequently, the slower versions, even the scalar ones, are almost as fast as the fastest versions outside the *L3* cache. Only the *AoS* versions, scalar and vector, are actually slower.

III.5 `float` performance vs `double` performance

III.5.a Comparing best performances

c/el		Scalar				Vector				Intrinsics
		AoS	*SoAoS*	*SoA*	*AoSoA*	*AoS*	*SoAoS*	*SoA*	*AoSoA*	*SoA*
1 core	`float`	31.81	33.33	30.52	29.66	14.44	13.61	5.75	6.17	5.33
	`double`	35.80	36.79	34.19	32.80	23.63	20.84	13.75	14.06	11.92
	ratio	1.13	1.10	1.12	1.11	1.64	1.53	2.39	2.28	2.24
OpenMP	`float`	2.32	2.38	2.20	2.18	1.04	0.98	0.43	0.46	0.40
	`double`	2.56	2.67	2.45	2.45	1.73	1.91	1.02	1.08	0.87
	ratio	1.11	1.12	1.11	1.12	1.67	1.94	2.40	2.36	2.18

Table 6: Speed-ups `float` computation vs `double` computation from best performances for each one

When we compare the best performance difference between `float` computation and `double` computation, we can see that `float` is obviously faster than `double`. In fact, `float` is twice faster than `double`. This is simple: *SIMD* registers have a fixed size, and `doubles` are twice bigger than `floats`. So, with one single vector instruction, we can compute only half `doubles` than `floats`.

However, if we look closer, we can see that the speed-ups is not 2, but is actually bigger about 2.4 in some cases. This can be explained by the fact that some instructions have not the same latency in both cases. For example, the division suffers from this phenomenon. Indeed, `float` division with 256-bit wide registers has a latency of 21 while the `double` one has a latency of 20.

register size	`float`	`double`
128 bits	13 / 5	20 / 12
256 bits	21 / 13	35 / 25

Table 7: Latencies of the div instruction on Haswell (latency / reciprocal throughput in cycles)

III.5.b Comparing at same size

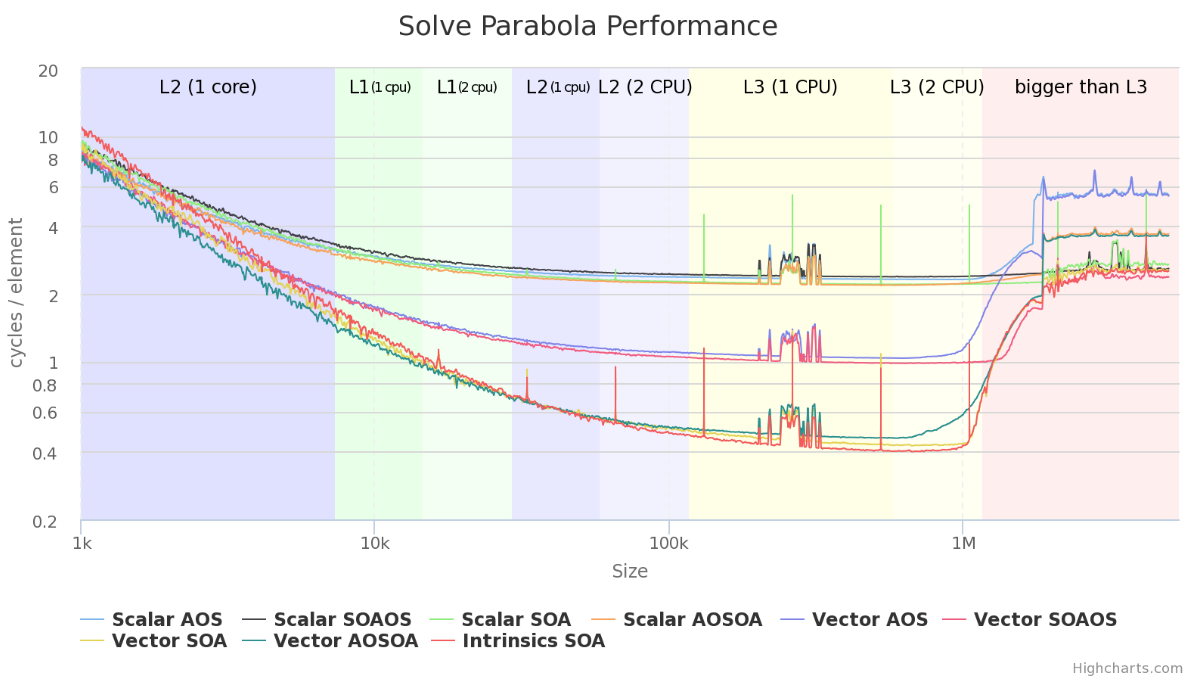

Chart 8: Performance of `solveParabola` using `float` and OpenMP

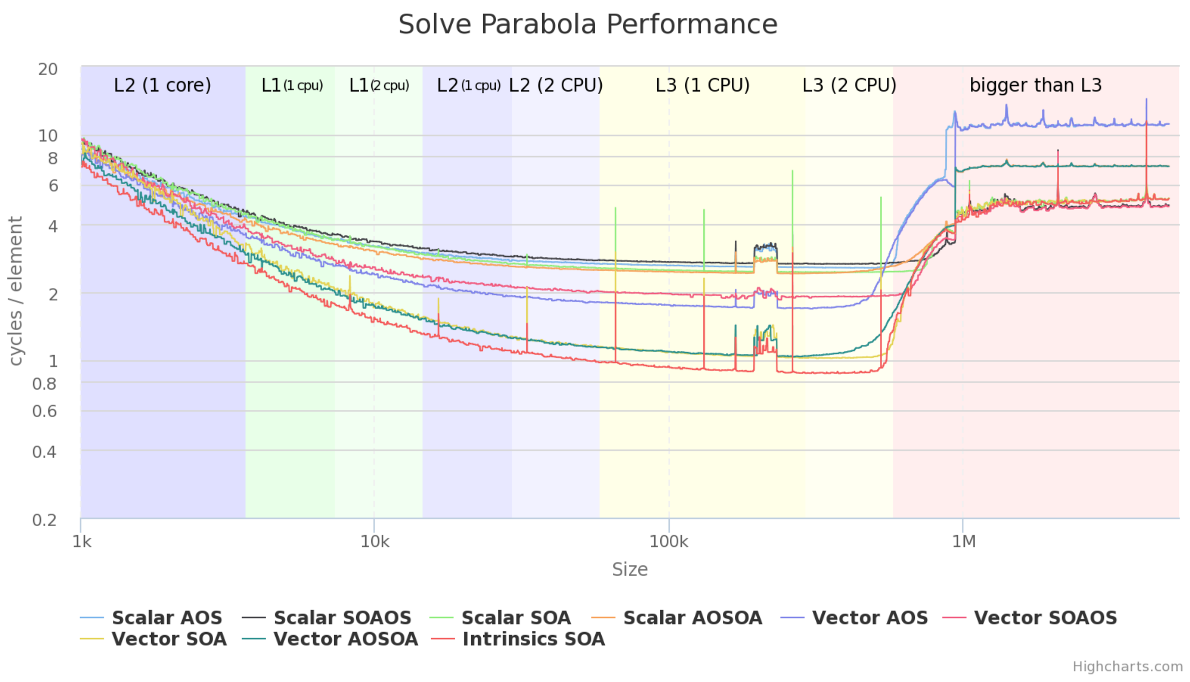

Chart 9: Performance of `solveParabola` using `double` and OpenMP

However, this is not the only effect of using `double` instead of `float`. Indeed, on the performance chart of `float` computation and `double` computation, we can clearly see that with `double`, cache can only contains half data than in `float` (about 1 000 000 elements in `float` and 500 000 elements in `double`). The explication is still simple: cache has a fixed size, but `double`s are twice bigger than `float`s. So we run out of cache twice faster with `double`.

This implies a bigger problem that affects performance much more than previously explained. Indeed, considering we want to compute about 1 000 000 elements with *SoA* memory layout and OpenMP. Using `float`s, we have a performance about $0.41c/el$. But with `double`s, the data does not fit within the cache *L3*, and we reach a performance about $4.5\,c/el$. So the `float` speed-up here is not $\times 2.40$ as expected but raises to $\frac{4}{0.4} = \times 11$. So the `double` computation is here more than four times slower than previously expected.

This is the reason why it is advantageous to do the whole computation with `float`s when this is possible. However, this is not always the case. Sometimes, `float` is not precise enough to give good results.

IV Numerical Stability

In scientific computation, we need to manipulate real numbers. However, it is impossible for a computer to handle reals. So we need to approximate reals with a subset $\mathbb{F}$ of reals. The idea is to approximates numbers by other numbers which can easily be stored and manipulated by computers [3].

The idea is simple: we have $\mathbb{F} \subset \mathbb{R}$ and $\#\mathbb{F}$ is finite. We want to store $x \in \mathbb{R}$. For this purpose, we find an approximation $\hat{x} \in \mathbb{F}$ such as $\forall \hat{y} \in \mathbb{F} \setminus \{\hat{x}\}, |\hat{x} - \hat{y}| > |x - \hat{x}|$. For every function $f : \mathbb{R} \to \mathbb{R}$, we also define its approximation $\hat{f} : \mathbb{F} \to \mathbb{F}$ such as $\forall \hat{a} \in \mathbb{F}, \hat{f}(\hat{a}) = \widehat{f(\hat{a})}$.

The problem with such an approximation is that we cannot assure $\hat{f}(\hat{x}) = \widehat{f(x)}$. In other words, we can assure that the approximate function gives the better result possible for its entry, but only if the entry is fully accurate. Otherwise, we cannot assure the output accuracy. All our problems come from this phenomenon.

Here comes the floating point representation normalized by IEEE 754. It normalizes how to store and how to manipulate such numbers. The norm does not fit exactly the previous rule because $\{-\infty, +\infty, \text{NaN}\} \in \mathbb{F}$ but it is not really a problem here.

Considering this, it is important to check the accuracy of the output. Indeed, if we have a quantity about a meter and we need a precision about a micrometer, we need an accuracy about 10^{-6}. But if the computation gives an output with an accuracy of 10^{-3}, we have not enough precision and the maximum precision we have is about one millimeter instead of one micrometer as expected.

Precision		Half	Single	Double	Extended	Quad
Usual C type			`float`	`double`	`long double`	
Precision	(bits)	11	24	53	64	113
	(digits)	3.3	7.2	15.9	19.3	34.0
Max Exponent	(bits)	15	127	1023	16383	16383
	(digits)	4.5	38.2	307.9	4931.7	4931.7

Table 8: Floating point precision

The problem is the following: the output accuracy depends on the input accuracy, but also on the floating point representation. Indeed, the floating point representation can only handle a finite and fixed precision depending on the floating point representation used.

It is crucial to distinguish two quantities:

precision: Represents the number of bits in the mantissa of the binary floating point representation

accuracy: Represents the number of *correct* bits in the mantissa.

For example, we want to compute $f(x)$ using `floats` (24 bits precision). So we store x into a `float`: we get an approximation $\hat{x}$ of x. Considering the binary scientific notation,

value	scientific notation	precision	accuracy
x	$1.0101\underline{001}_b \cdot {10_b}^{101_b}$	∞	∞
$\hat{x}$	$1.0101_b \cdot {10_b}^{101_b}$	5	5
$f(x)$	$1.1110\underline{101}_b \cdot {10_b}^{-11_b}$	∞	∞
$\widehat{f(\hat{x})}$	$1.1011_b \cdot {10_b}^{-11_b}$	5	2

Table 9: Precision vs Accuracy: floating point example (5 bits)

we have the 23 firsts bits (binary digits) of both x and $\hat{x}$ equal and the 24th bit of $\hat{x}$ is rounded (according to the current rounding mode). $\hat{x}$ has no more bits in the mantissa. So the precision is the number of bits used to represent $\hat{x}$, the approximation of x (here 24).

Now, we compute $\widehat{f(\hat{x})}$ an approximation of $f(x)$. We compare $\widehat{f(\hat{x})}$ and $f(x)$: we count the number of equal bits in the binary scientific notation until the first different bit. This number is the accuracy of $\widehat{f(\hat{x})}$.

If the chosen floating point representation cannot handle the requested precision plus the error introduced by the computation, we need to chose a more precise floating point representation. This can be a problem if we need to go fast because `double` computation cannot be optimized as well as `float` computation. This is even worse with `long double` or with other floating point representations because those cannot be vectorized as processors are not conceived to vectorize them for now.

This also pose the problem of the reproducibility of the computation as exposed in [7] and [4] So it is important to check the numerical options of the compiler. Indeed, current compilers have several modes for numerical optimizations. If we want our program to be fast, it can be interesting to relax the numerical constraint, sacrificing the reproducibility and letting the compiler to do numerically unsafe transformations like reorganize the operations order. For the whole project, the *icc* option `-fp-model fast`.

This is important to understand how the accuracy of the output behaves depending on the operations done during the computation and on the floating point representation chosen. The idea is to run a function with several floating point precisions and compare results with the more precise floating point representation used. To do so, I used an arbitrary precision library: MPFR. It implements a IEEE 754 compliant floating point arithmetic except the exception support and can be expected to have the same behavior than common floating point representation implemented in processors. Using MPFR, it is pretty easy to change the precision of the floating point representation and it is easy to compute the same function with different precision.

In what follows, results are compared with the results obtained with a 1024 bits precise floating point representation and accuracy is computed with the same extremely precise floating point representation. It is computed with the following formula:

$$accuracy = -log_2 \left| \frac{\hat{x} - x}{x} \right| \qquad \text{where} \left| \begin{array}{l} x \text{ is the precise value} \\ \hat{x} \text{ is the approximation} \end{array} \right.$$

With this formula, it is possible to get a negative accuracy or greater than the precision. Getting an accuracy greater than the precision has no meaning. However, a negative accuracy can be meaningful. Indeed, the negative part can be interpreted like the error on the exponent.

IV.1 Theoretical algorithms

IV.1..i Multiplicative Reduction

```
mpreal multiply(mpreal* a, int n) {
  int i;
  mpreal s = 0;
  for (i = 0; i < n; ++i) {
    s *= a[i];
  }
  return s;
}
```

Listing 6: Multiplicative reduction

When analyzing the accuracy of a really simple algorithm like a multiplicative reduction (cf: Listing 6, Multiplicative reduction), it appears naturally that the accuracy decreases when the number of multiplication increases. In fact, we can see that, for a multiplicative reduction, the accuracy decreases logarithmically on the number of multiplication (cf: Chart 10, Floating point accuracy of a multiplicative reduction).

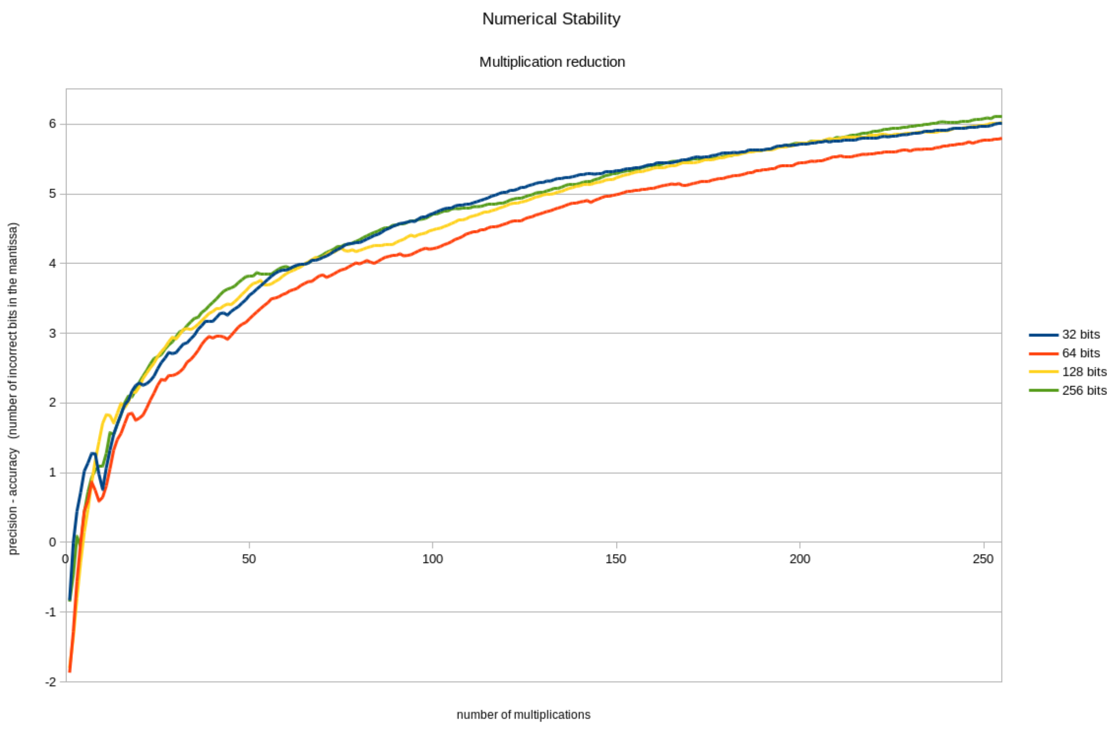

Chart 10: Floating point accuracy of a multiplicative reduction

Another interesting point is that the number of incorrect bits in the mantissa (ie: *precision*−*accuracy*) is independent of the precision of the floating point representation used. This result will be more visible on a real algorithm (cf: IV.2, Actual algorithm). It is surprising at the beginning, but when you think about it, it is quite logical. Indeed, when doing a floating operation according to IEEE 754, for example a multiplication, every bits in the mantissa is exact, except the last one which is rounded. So the error is propagating from the last bit in the mantissa. The error increases when doing multiple operations because for next operations, the input is not exact. During this process, the first bits in the mantissa are not involved, so it does not depends on the number of bits in the whole mantissa (except when the error reaches the first bit).

IV.1..ii Additive Reduction

However, the main source of numerical instability is not rounding errors in multiplication, but loss of significance with addition and subtraction. Loss of significance occurs when an addition or a subtraction cannot be computed without losing accurate bits. There two cases of Loss of significance:

Absorption: When adding (or subtracting) a big number and a tiny one. Here is an example with 5 digits numbers: $1234.5 + 0.12345 \to 1234.6$ instead of 1234.62345. Absorption seems pretty inoffensive, but repeated, it can bring much bigger error.

Cancellation: When subtracting two near equal numbers. Here is an example with 5 digits numbers: $1234.6 - 1234.5 \to 0.1$. The result here is only accurate on the first digit instead of 5 digits.

If the number 1234.6 is an approximation of $1234.5 + 0.12345$ we get the following result: $((1234.5 + 0.12345) - 1234.5) \to 0.1$ instead of 0.12345. When a cancellation is important (a few bits remaining), it is called catastrophic cancellation.

However, it is possible to reduce the effect of Loss of significance with appropriate algorithms. To demonstrate this point, I analyzed the accuracy of several summation algorithms:

- The naive summation algorithm
- The naive summation algorithm with data sorted in descending order
- An accurate summation algorithm (always add the two smaller elements)

```
input  : $A$ (Array to sum)
output: Sum of $A$
$s \leftarrow 0$
for $x \in A$ do
    $s \leftarrow s + x$
end
return $s$
```

Algorithm 1: Naive summation (C implementation in appendix)

```
input  : $A$ (Array to sum)
output: Sum of $A$
if $A$ is empty then
    return 0
end
while $A$ has more than one element do
    $x \leftarrow \min(A)$
    remove $x$ from $A$
    $y \leftarrow \min(A)$
    remove $y$ from $A$
    put $(x + y)$ into $A$
end
return the only element of $A$
```

Algorithm 2: Accurate summation (C implementation in appendix)

When executing these algorithms on 2048 random shuffled elements between 1 and 10, we can see that the accurate summation algorithm gives a full accuracy (accuracy greater than precision). But the naive algorithm gives not fully accurate results: $precision - accuracy \in [0.8, 2.5]$. When data are sorted in descending order, the naive algorithm is even worse: $precision - accuracy \in [2.3, 3.4]$.

$precision - accuracy$	32 bits	64 bits	128 bits	256 bits
Naive summation	0.85	1.46	1.99	2.42
Anti-sorted naive summation	2.37	2.35	2.89	3.39
Accurate summation	-1.46	-1.38	-1.66	-1.48

Table 10: Accuracy dependency on algorithm used ($N = 2048$)

IV.2 Actual algorithm

`fitParabola` (cf: Listing 15, `fitParabola` function) consists of generating 3 by 3 symmetric matrix with some simple algebraic expressions and then solving the associated system with the Cholesky decomposition upto three times (cf: Listing 16, Cholesky solver). The end of the algorithm is just computing some quantities about the found solution involving for example sums and maximums.

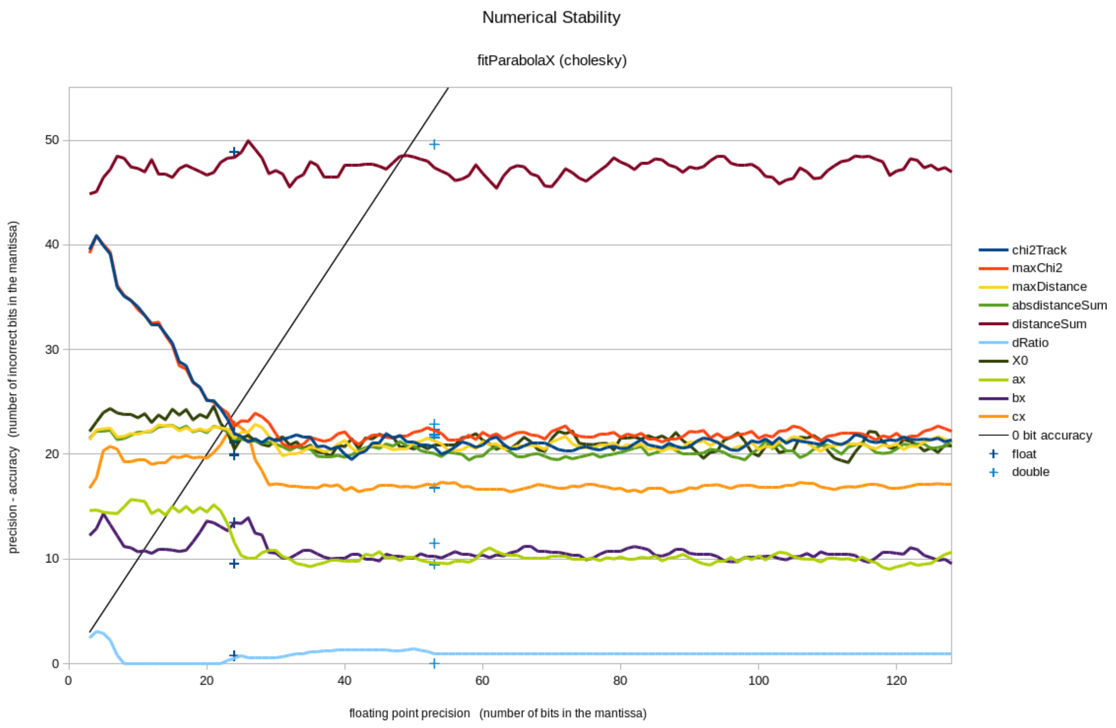

Chart 11: `fitParabola` variables accuracy (max error)

When looking the variables accuracy, something is clear: the number of incorrect bits in the mantissa ($precision - accuracy$) seems to be constant. As explained before, it is due to the fact that the error propagates from the last and least significant bits in the mantissa. Consequently, if you need 3 accurate bits more on the result, you need to do the computation with a floating point representation more precise by 3 bits too. However, this is valid only if the accuracy is positive. Indeed, if is accuracy is negative, our variables have not the right exponent and their values are completely meaningless.

Besides that, a lot of variables seems to have the expression $precision - accuracy$ constant even with negative accuracy. Only two variables have this expression dropping for small precision (`chi2Track` and `maxChi2`). This behavior could be explained by full absorption and full cancellation ($(1 + 1e100) - 1e100 \rightarrow 0$ instead of 1). With more time, it would be possible to analyze this hypothesis, but that was not possible.

About the algorithm on its own, we seem to have a pretty bad worst accuracy (accuracy in the worst empirical case). Indeed, most variables have a number of incorrect

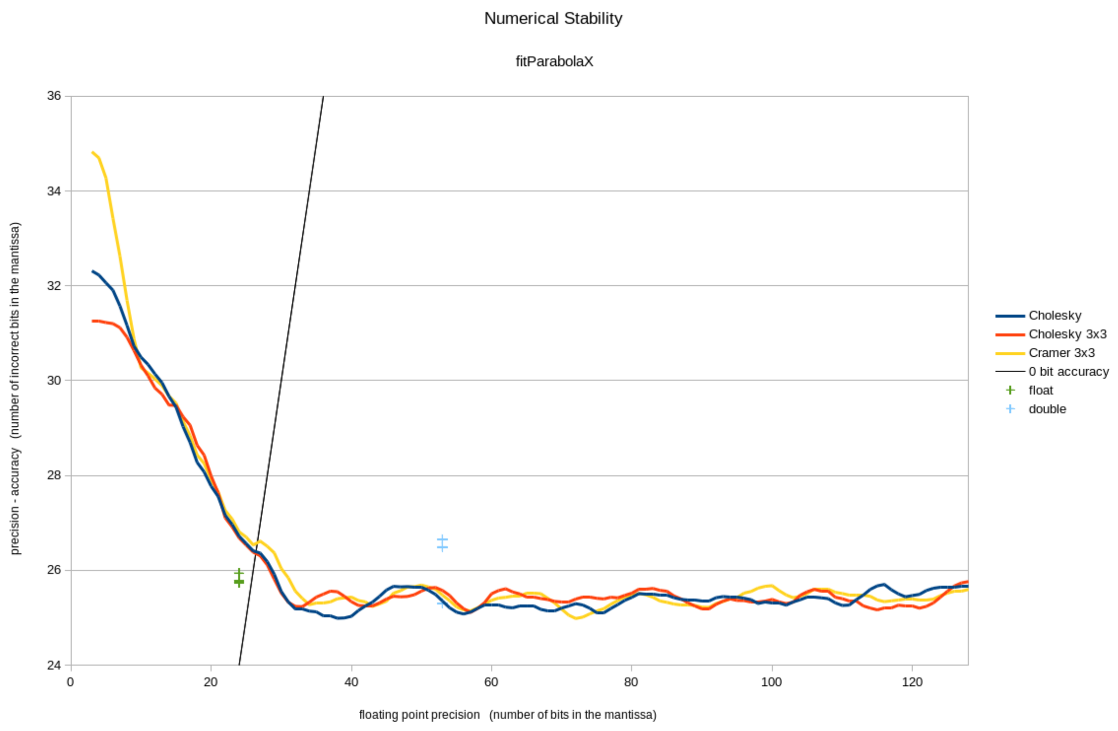

Chart 12: `fitParabola` algorithms accuracy (max error, average on variables accuracy)

bits about 21 which is quite bad, especially in `float`: in this case, we have only 3 correct bits, which means only one single decimal digit correct. The worst case is the variable `distanceSum` which have a very high number of incorrect bits (between 45 and 50). This means that the `float` computation of this variable is so erroneous that the value is meaningless. With `double` computation, the accuracy is at least positive, but is about 4 bits: a little better than one correct digit. However, it is important to remember that is the maximum error and the average case is much more accurate.

As we saw earlier, it is sometimes possible to increase accuracy with other algorithms. So I tried to use another solving algorithm because that is the less accurate part of the algorithm (solving matrices is always a problem). When solving with the explicit equations of the Cramer's rule (cf: Listing 17, Cramer's rule solver) instead of using the Cholesky decomposition, it seems to have the same accuracy which is surprising. But this is possibly due to the origin of the error. Indeed, when solving a matrix system, it is important to know the condition number of the matrix and how it influences the accuracy.

The condition number is the ratio between the greatest eigenvalue the smallest one. So the condition number of a matrix represents how the matrix stretches dimensions when multiply a vector. By solving a matrix system, the error on the result is typically the error on the input multiplied by the condition number of the matrix. In other words: $accuracy(output) = accuracy(input) - log_2(condition\_number)$. Which means that the main source of accuracy loss is the conditioning of matrices and not the algorithm used.

Conclusion

Throughout this project, we have been able to better understand the way optimizing compilers behave depending on the shape of the code and how change this shape in order to let them taking full advantage of the hardware and fully optimizing the code, especially vectorizing it.

First of all, we have seen that the time measurement is not simple and deserves its own study. The small conclusion for this part is the following: in order to have reliable measures, taking the minimum time of several executions is a good way to do it. It is quite fast and reliable enough.

Then, the most important factor for optimizing the code is doing high level transformations. Indeed, that is precisely what a compiler is not authorized to do, so this task remains to the human. The fact is, when suitable high level transformations have been done, compilers are able to optimize the code very well without so much effort (mostly add compiler directives like `pragmas` and `restrict` keyword).

Among these transformations, the most important one is about the memory layout. Indeed, it is mainly the change of the memory layout which enables or not the vectorization by the compiler. It is easy for them to vectorize *SoA* or *AoSoA* and the results are near from the manual *SIMD* version. But *AoSoA* is quite difficult to implement, so the best memory layout to use in general might be *SoA*. Besides this, *AoSoA* seems to have a better cache behavior avoiding systematic cache eviction.

OpenMP does great about the performance speed-up. Indeed, the code actually needs only a few changes and it gives a very high efficiency: speed-ups are about the number of cores used by OpenMP. However, the OpenMP overhead can be problematic if data is too small.

Then, using `floats` instead of `doubles` is really interesting concerning the performances. Indeed, the performance ratio is greater than 2, and it is even more when `double` data is too big to fits within the cache, but `float` data is small enough as it is twice smaller.

However, using `floats` can bring numerical stability problems. Indeed, by its nature itself, the floating point computation is not infinitely precise. And the accuracy of the result strongly depends on the floating point precision. In some cases, `floats` are just not suitable for the required calculation. In these cases, it is better to use `doubles`.

All these points would be studied in much more details and could have their own study.

References

[1] Allen, R. and Kennedy, K., editors (2002). *Optimizing compilers for modern architectures: a dependence-based approach*, chapter 8,9,11. Morgan Kaufmann.

[2] Ewart, T., Delalondre, F., and Schürmann, F. (2014). Cyme: A library maximizing simd computation on user-defined containers. In *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ISC 2014, pages 440–449, New York, NY, USA. Springer-Verlag New York, Inc.

[3] Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48.

[4] Jézéquel, F., Langlois, P., and Revol, N. (2014). First steps towards more numerical reproducibility. *ESAIM: Proceedings*, 45:229–238.

[5] Lacassagne, L., Etiemble, D., Hassan-Zahraee, A., Dominguez, A., and Vezolle, P. (2014). High level transforms for simd and low-level computer vision algorithms. In *ACM Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, pages 49–56.

[6] Leißa, R., Haffner, I., and Hack, S. (2014). Sierra: A simd extension for c++. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 17–24, New York, NY, USA. ACM.

[7] Revol, N. and Théveny, P. (2013). Numerical reproducibility and parallel computations: Issues for interval algorithms. *CoRR*, abs/1312.3300.

Glossary

AoS Array of Structures. 5, 16–19, 25, 27, 28

AoSoA Array of Structures of Arrays. 5, 17–19, 24, 25, 28, 39

AVX Advanced Vector Extensions. 10, 17–19, 21, 43

FMA Fused Multiply-Add. 20, 43

Gaudi Framework written in C++ used within LHCb to write scientific code. 9, 11, 12

IEEE 754 Floating point normalization. 31, 32, 35

LSB Least Significant Bit. 23, 24

MPFR Multi-precision floating point library written in C. 32

OpenMP Multi-processing library for C, C++, Fortran. 4, 10, 25–30, 39

SIMD Simple Instruction Multiple Data. 5, 10, 17, 19–21, 25, 28, 39, 43, 51

SoA Array of Structures. 5, 14, 16–19, 23–25, 27, 28, 30, 39, 42, 48–51

SoAoS Structure of Arrays of Structures. 5, 18, 19, 25, 28

SSE Streaming SIMD Extensions. 10, 17

Appendix

A Haswell architecture used

The Haswell machine used embeds the following CPU:

- Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz

The code for this machine have been compiled for the *AVX2* with *FMA* instructions set.

Cache level	**Size**	**Latency**	**Associativeness**
L1	32 KB (per core)	4 cycles	8-way
L2	256 KB (per core)	11 cycles	8-way
L3	8192 KB (shared)	36 cycles	16-way

Table 11: Haswell cache properties

Port	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
int	*ALU* shift	*ALU* *LEA*				*ALU* shift	*ALU* shift	
SIMD int	mul	*ALU*				*ALU*		
SIMD float	***FMA*** divide	***FMA*** divide						
SIMD misc	logic shift	logic				logic		
Misc	branch		load store	load store	store data		branch	store

Table 12: Haswell execution units

Operation	**Execution unit**	**Latency**	**Reciprocal Throughput** (per unit)
FMA/mul	FMA	5	1
add/sub	add	3	1
div	divide	21	13
rcp (fast reciprocal)	divide	7	1
logical operations	logic	1	1
load	load	$\geqslant 4$	1
store	store	$\geqslant 4$	1

Table 13: Haswell floating point operations characteristics

B solveParabola codes

```c
void solve(float zRef, float dRatio,
    float h1x, float h1z,
    float h2x, float h2z,
    float h3x, float h3z,
    float* restrict a, float* restrict b, float* restrict c
) {
  float z1 = h1z - zRef;
  float z2 = h2z - zRef;
  float z3 = h3z - zRef;
  float x1 = h1x;
  float x2 = h2x;
  float x3 = h3x;

  float corrZ1 = (float) 1.0 + dRatio*z1;
  float corrZ2 = (float) 1.0 + dRatio*z2;
  float corrZ3 = (float) 1.0 + dRatio*z3;

  float det = z1*z1*corrZ1*z2 + z1*z3*z3*corrZ3 + z2*z2*corrZ2*z3 -
              z2*z3*z3*corrZ3 - z1*z2*z2*corrZ2 - z3*z1*z1*corrZ1;

  if (fabsf(det) < (float) 1e-8) {
    *a = (float) 0.0;
    *b = (float) 0.0;
    *c = (float) 0.0;
    return;
  }

  float detA = (x1)*z2 + z1*(x3) + (x2)*z3 - z2*(x3) - z1*(x2) - z3*(x1);

  float detB = z1*z1*corrZ1*x2 + x1*z3*z3*corrZ3 + z2*z2*corrZ2*x3 -
               x2*z3*z3*corrZ3 - x1*z2*z2*corrZ2 - x3*z1*z1*corrZ1;

  float detC = z1*z1*corrZ1*z2*x3 + z1*z3*z3*corrZ3*x2 + z2*z2*corrZ2*z3*x1 -
               z2*z3*z3*corrZ3*x1 - z1*z2*z2*corrZ2*x3 - z3*z1*z1*corrZ1*x2;

  *a = detA / det;
  *b = detB / det;
  *c = detC / det;
}
```

Listing 7: solveParabola function

B.1 No operation duplication

```
void solve(float zRef, float dRatio,
    float h1x, float h1z,
    float h2x, float h2z,
    float h3x, float h3z,
    float* restrict a, float* restrict b, float* restrict c
) {
  float z1 = h1z - zRef; // uses 1 add unit
  float z2 = h2z - zRef;
  float z3 = h3z - zRef;
  float x1 = h1x; // no unit used (it is just an alias)
  float x2 = h2x;
  float x3 = h3x;

  float corrZ1 = (float) 1.0 + dRatio*z1; // uses 1 FMA unit
  float corrZ2 = (float) 1.0 + dRatio*z2;
  float corrZ3 = (float) 1.0 + dRatio*z3;

  float z1z1corrZ1 = z1*z1*corrZ1; // uses 2 FMA units
  float z2z2corrZ2 = z2*z2*corrZ2;
  float z3z3corrZ3 = z3*z3*corrZ3;

  float det = z1z1corrZ1*z2 + z1*z3z3corrZ3 + z2z2corrZ2*z3 -
              z2*z3z3corrZ3 - z1*z2z2corrZ2 - z3*z1z1corrZ1;

  if (fabsf(det) < (float) 1e-8) {
    *a = (float) 0.0;
    *b = (float) 0.0;
    *c = (float) 0.0;
    return;
  }

  float x1z2 = x1*z2;
  float x1z3 = x1*z3;
  float x2z1 = x2*z1;
  float x2z3 = x2*z3;
  float x3z1 = x3*z1;
  float x3z2 = x3*z2;

  float detA = x1z2 + x3z1 + x2z3 - x3z2 - x2z1 - x1z3;

  float detB = z1z1corrZ1*x2 + x1*z3z3corrZ3 + z2z2corrZ2*x3 -
               x2*z3z3corrZ3 - x1*z2z2corrZ2 - x3*z1z1corrZ1;

  float detC = z1z1corrZ1*x3z2 + z3z3corrZ3*x2z1 + z2z2corrZ2*x1z3 -
               z3z3corrZ3*x1z2 - z2z2corrZ2*x3z1 - z1z1corrZ1*x2z3;

  float oneOverDet = (float) 1.0 / det;

  *a = detA * oneOverDet;
  *b = detB * oneOverDet;
  *c = detC * oneOverDet;
}
```

Listing 8: `solveParabola` function (no operation duplication)

B.2 Intrinsics version

```
void solveIntrinsics(float zRef, float dRatio,
    __m256 x1, __m256 z1,
    __m256 x2, __m256 z2,
    __m256 x3, __m256 z3,
    __m256 *a, __m256 *b, __m256 *c
) {
  // Constants
  __m256 zRefV   = _mm256_set1_ps((float) zRef);
  __m256 dRatioV = _mm256_set1_ps((float) dRatio);
  __m256 oneV    = _mm256_set1_ps((float) 1.0);

  // Input
  __m256 z1V = _mm256_sub_ps(z1, zRefV);
  __m256 z2V = _mm256_sub_ps(z2, zRefV);
  __m256 z3V = _mm256_sub_ps(z3, zRefV);
  __m256 x1V = x1;
  __m256 x2V = x2;
  __m256 x3V = x3;

  // Intermediate computation
  __m256 corrZ1 = _mm256_fmadd_ps(dRatioV, z1V, oneV);
  __m256 corrZ2 = _mm256_fmadd_ps(dRatioV, z2V, oneV);
  __m256 corrZ3 = _mm256_fmadd_ps(dRatioV, z3V, oneV);

  __m256 z1_2 = _mm256_mul_ps(z1V, z1V);
  __m256 z2_2 = _mm256_mul_ps(z2V, z2V);
  __m256 z3_2 = _mm256_mul_ps(z3V, z3V);

  __m256 z1z1corrZ1 = _mm256_mul_ps(z1_2, corrZ1);
  __m256 z2z2corrZ2 = _mm256_mul_ps(z2_2, corrZ2);
  __m256 z3z3corrZ3 = _mm256_mul_ps(z3_2, corrZ3);

  // det
  __m256 det = _mm256_add_ps(_mm256_add_ps(
      _mm256_fmadd_ps (z2V, z1z1corrZ1, _mm256_mul_ps(z1V, z3z3corrZ3)),
      _mm256_fmsub_ps (z3V, z2z2corrZ2, _mm256_mul_ps(z2V, z3z3corrZ3))),
      _mm256_fnmsub_ps(z1V, z2z2corrZ2, _mm256_mul_ps(z3V, z1z1corrZ1))
  );
  __m256 oneOverDet = _mm256_div_ps(oneV, det);

  // detC
  __m256 detC = _mm256_add_ps(_mm256_add_ps(
      _mm256_fmadd_ps (
                      _mm256_mul_ps(x3V, z2V), z1z1corrZ1,
        _mm256_mul_ps(_mm256_mul_ps(x2V, z1V), z3z3corrZ3)),
      _mm256_fmsub_ps (
                      _mm256_mul_ps(x1V, z3V), z2z2corrZ2,
        _mm256_mul_ps(_mm256_mul_ps(x1V, z2V), z3z3corrZ3))),
      _mm256_fnmsub_ps(
                      _mm256_mul_ps(x3V, z1V), z2z2corrZ2,
        _mm256_mul_ps(_mm256_mul_ps(x2V, z3V), z1z1corrZ1))
  );

  // detB
  __m256 detB = _mm256_add_ps(_mm256_add_ps(
      _mm256_fmadd_ps (x2V, z1z1corrZ1, _mm256_mul_ps(x1V, z3z3corrZ3)),
      _mm256_fmsub_ps (x3V, z2z2corrZ2, _mm256_mul_ps(x2V, z3z3corrZ3))),
      _mm256_fnmsub_ps(x1V, z2z2corrZ2, _mm256_mul_ps(x3V, z1z1corrZ1))
  );

  // detA
  __m256 detA = _mm256_add_ps(_mm256_add_ps(
      _mm256_fmadd_ps (x1V, z2V, _mm256_mul_ps(x3V, z1V)),
      _mm256_fmsub_ps (x2V, z3V, _mm256_mul_ps(x3V, z2V))),
      _mm256_fnmsub_ps(x2V, z1V, _mm256_mul_ps(x1V, z3V))
  );
  __m256 aV, bV, cV;
  aV = _mm256_mul_ps(detA, oneOverDet);
  bV = _mm256_mul_ps(detB, oneOverDet);
  cV = _mm256_mul_ps(detC, oneOverDet);

  // computes abs(det), then computes condition
  __m256 sign_mask = _mm256_set1_ps(-0.0); // -0.0 = 1 << 31
  __m256 abs = _mm256_andnot_ps(sign_mask, det);
  __m256 cond = _mm256_cmp_ps(abs, _mm256_set1_ps((float) 1e-8), _CMP_LT_OS);

  aV = _mm256_andnot_ps(cond, aV);
  bV = _mm256_andnot_ps(cond, bV);
  cV = _mm256_andnot_ps(cond, cV);

  *a = aV;
  *b = bV;
  *c = cV;

}
```

Listing 9: `solveParabola` function (intrinsics version)

C `solveParabola` dependency flow chart

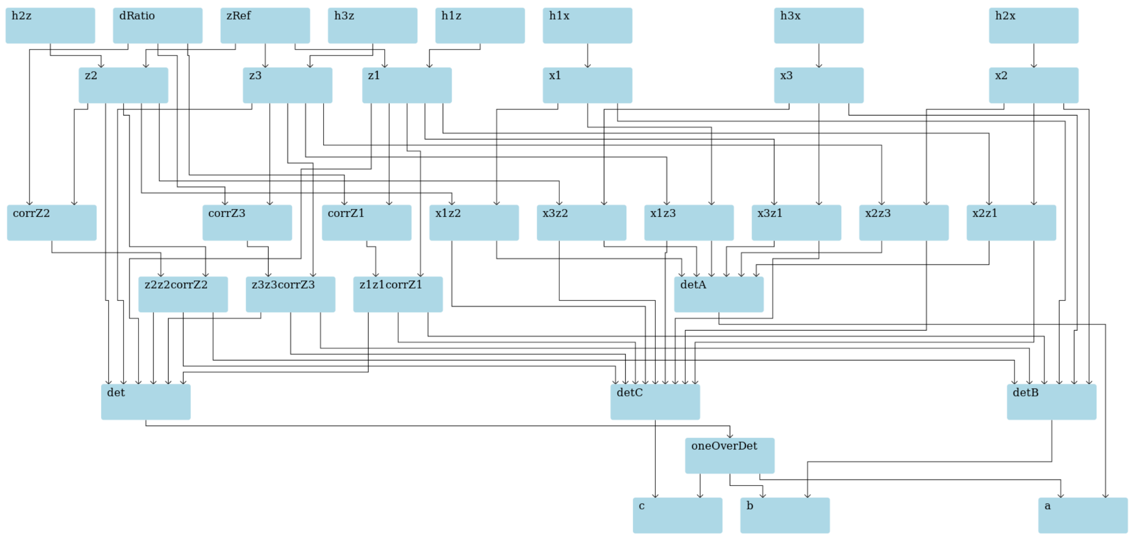

Chart 13: `solveParabola` dependency flow chart

D solveParabola loop assembly dumps

D.1 Scalar *SoA*

```
404fec:   c4 41 7a 10 1c 96        vmovss (%r14,%rdx,4),%xmm11
404ff2:   c4 41 7a 10 14 94        vmovss (%r12,%rdx,4),%xmm10
404ff8:   c5 7a 10 44 24 f8        vmovss -0x8(%rsp),%xmm8
404ffe:   c4 41 7a 10 0c 90        vmovss (%r8,%rdx,4),%xmm9
405004:   c4 41 22 5c d8           vsubss %xmm8,%xmm11,%xmm11
405009:   c4 41 2a 5c d0           vsubss %xmm8,%xmm10,%xmm10
40500e:   c4 41 32 5c c8           vsubss %xmm8,%xmm9,%xmm9
405013:   c4 c1 22 59 fb           vmulss %xmm11,%xmm11,%xmm7
405018:   c4 c1 2a 59 f2           vmulss %xmm10,%xmm10,%xmm6
40501d:   c4 c1 32 59 d9           vmulss %xmm9,%xmm9,%xmm3
405022:   c5 fa 10 05 5a 8a 00     vmovss 0x8a5a(%rip),%xmm0
40502a:   c5 f8 28 e8              vmovaps %xmm0,%xmm5
40502e:   c5 f8 28 e0              vmovaps %xmm0,%xmm4
405032:   c4 c2 71 b9 eb           vfmadd231ss %xmm11,%xmm1,%xmm5
405037:   c4 e2 29 b9 e1           vfmadd231ss %xmm1,%xmm10,%xmm4
40503c:   c4 e2 31 b9 c1           vfmadd231ss %xmm1,%xmm9,%xmm0
405041:   c4 41 7a 10 24 97        vmovss (%r15,%rdx,4),%xmm12
405047:   c4 41 7a 10 6c 95 00     vmovss 0x0(%r13,%rdx,4),%xmm13
40504e:   c4 41 7a 10 34 91        vmovss (%r9,%rdx,4),%xmm14
405054:   c5 42 59 c5              vmulss %xmm5,%xmm7,%xmm8
405058:   c5 ca 59 fc              vmulss %xmm4,%xmm6,%xmm7
40505c:   c5 e2 59 e8              vmulss %xmm0,%xmm3,%xmm5
405060:   c5 40 57 3d 08 8a 00     vxorps 0x8a08(%rip),%xmm7,%xmm15
405068:   c5 ba 5c f5              vsubss %xmm5,%xmm8,%xmm6
40506c:   c4 c1 52 58 df           vaddss %xmm15,%xmm5,%xmm3
405071:   c5 b8 57 15 f7 89 00     vxorps 0x89f7(%rip),%xmm8,%xmm2
405079:   c5 a2 59 c3              vmulss %xmm3,%xmm11,%xmm0
40507d:   c5 c2 58 e2              vaddss %xmm2,%xmm7,%xmm4
405081:   c4 c2 59 b9 c1           vfmadd231ss %xmm9,%xmm4,%xmm0
405086:   c5 fa 10 15 f2 89 00     vmovss 0x89f2(%rip),%xmm2
40508e:   c4 c2 49 b9 c2           vfmadd231ss %xmm10,%xmm6,%xmm0
405093:   c5 78 54 3d c5 89 00     vandps 0x89c5(%rip),%xmm0,%xmm15
40509b:   c4 c1 78 2f d7           vcomiss %xmm15,%xmm2
4050a0:   76 0f                    jbe    4050b1 <solveParabolaSOA+0x121>
4050a2:   89 04 96                 mov    %eax,(%rsi,%rdx,4)
4050a5:   89 44 95 00              mov    %eax,0x0(%rbp,%rdx,4)
4050a9:   89 04 93                 mov    %eax,(%rbx,%rdx,4)
4050ac:   e9 83 00 00 00           jmpq   405134 <solveParabolaSOA+0x1a4>
4050b1:   c5 7a 10 3d cb 89 00     vmovss 0x89cb(%rip),%xmm15
4050b9:   c4 c1 2a 5c d1           vsubss %xmm9,%xmm10,%xmm2
4050be:   c5 82 5e c0              vdivss %xmm0,%xmm15,%xmm0
4050c2:   c5 8a 59 e4              vmulss %xmm4,%xmm14,%xmm4
4050c6:   c4 41 22 5c f9           vsubss %xmm9,%xmm11,%xmm15
4050cb:   c4 c2 61 b9 e4           vfmadd231ss %xmm12,%xmm3,%xmm4
4050d0:   c4 41 02 59 fd           vmulss %xmm13,%xmm15,%xmm15
4050d5:   c5 fa 59 de              vmulss %xmm6,%xmm0,%xmm3
4050d9:   c4 c2 19 ab d7           vfmsub213ss %xmm15,%xmm12,%xmm2
4050de:   c4 41 22 5c fa           vsubss %xmm10,%xmm11,%xmm15
4050e3:   c4 c1 62 59 f5           vmulss %xmm13,%xmm3,%xmm6
4050e8:   c4 62 09 a9 fa           vfmadd213ss %xmm2,%xmm14,%xmm15
4050ed:   c4 e2 79 a9 e6           vfmadd213ss %xmm6,%xmm0,%xmm4
4050f2:   c5 fa 11 64 95 00        vmovss %xmm4,0x0(%rbp,%rdx,4)
4050f8:   c5 82 59 d0              vmulss %xmm0,%xmm15,%xmm2
4050fc:   c5 fa 11 14 96           vmovss %xmm2,(%rsi,%rdx,4)
405101:   c5 a2 59 d7              vmulss %xmm7,%xmm11,%xmm2
405105:   c5 22 59 dd              vmulss %xmm5,%xmm11,%xmm11
405109:   c5 aa 59 ed              vmulss %xmm5,%xmm10,%xmm5
40510d:   c4 c2 39 bb d2           vfmsub231ss %xmm10,%xmm8,%xmm2
405112:   c4 42 39 bd d9           vfnmadd231ss %xmm9,%xmm8,%xmm11
405117:   c4 62 41 ab cd           vfmsub213ss %xmm5,%xmm7,%xmm9
40511c:   c4 c1 32 59 fc           vmulss %xmm12,%xmm9,%xmm7
405121:   c4 62 11 a9 df           vfmadd213ss %xmm7,%xmm13,%xmm11
405126:   c4 c2 09 a9 d3           vfmadd213ss %xmm11,%xmm14,%xmm2
40512b:   c5 6a 59 c0              vmulss %xmm0,%xmm2,%xmm8
40512f:   c5 7a 11 04 93           vmovss %xmm8,(%rbx,%rdx,4)
405134:   48 ff c2                 inc    %rdx
405137:   48 3b d7                 cmp    %rdi,%rdx
40513a:   0f 8c ac fe ff ff        jl     404fec <solveParabolaSOA+0x5c>
```

Listing 10: solveParabola loop assembly (scalar *SoA*)

D.2 Vector *SoA*

```
406838:   c4 c1 7c 10 04 86        vmovups (%r14,%rax,4),%ymm0
40683e:   c4 41 7c 10 24 87        vmovups (%r15,%rax,4),%ymm12
406844:   c5 fc 10 0c 83           vmovups (%rbx,%rax,4),%ymm1
406849:   c5 fc 10 5c 24 20        vmovups 0x20(%rsp),%ymm3
40684f:   c5 7c 10 05 09 98 00     vmovups 0x9809(%rip),%ymm8
406857:   c5 7c 10 0c 24           vmovups (%rsp),%ymm9
40685c:   c5 7c 11 a4 24 e0 00     vmovups %ymm12,0xe0(%rsp)
406865:   c5 7c 10 1c 86           vmovups (%rsi,%rax,4),%ymm11
40686a:   c4 c1 7c 10 14 80        vmovups (%r8,%rax,4),%ymm2
406870:   c5 7c 10 2c 81           vmovups (%rcx,%rax,4),%ymm13
406875:   c5 7c 5c d3              vsubps %ymm3,%ymm0,%ymm10
406879:   c5 74 5c e3              vsubps %ymm3,%ymm1,%ymm12
40687d:   c5 ec 5c cb              vsubps %ymm3,%ymm2,%ymm1
406881:   c4 c1 2c 59 e2           vmulps %ymm10,%ymm10,%ymm4
406886:   c4 c1 1c 59 f4           vmulps %ymm12,%ymm12,%ymm6
40688b:   c5 7c 11 9c 24 20 01     vmovups %ymm11,0x120(%rsp)
406894:   c5 74 59 f1              vmulps %ymm1,%ymm1,%ymm14
406898:   c5 7c 11 94 24 00 01     vmovups %ymm10,0x100(%rsp)
4068a1:   c4 c1 7c 28 e8           vmovaps %ymm8,%ymm5
4068a6:   c4 c1 7c 28 f8           vmovaps %ymm8,%ymm7
4068ab:   c4 c2 35 b8 ea           vfmadd231ps %ymm10,%ymm9,%ymm5
4068b0:   c4 c2 35 b8 fc           vfmadd231ps %ymm12,%ymm9,%ymm7
4068b5:   c4 62 35 b8 c1           vfmadd231ps %ymm1,%ymm9,%ymm8
4068ba:   c5 5c 59 dd              vmulps %ymm5,%ymm4,%ymm11
4068be:   c5 cc 59 e7              vmulps %ymm7,%ymm6,%ymm4
4068c2:   c4 41 0c 59 f8           vmulps %ymm8,%ymm14,%ymm15
4068c7:   c5 f4 59 c4              vmulps %ymm4,%ymm1,%ymm0
4068cb:   c5 fc 11 a4 24 40 01     vmovups %ymm4,0x140(%rsp)
4068d4:   c5 7c 11 bc 24 c0 00     vmovups %ymm15,0xc0(%rsp)
4068dd:   c4 c2 25 b8 c4           vfmadd231ps %ymm12,%ymm11,%ymm0
4068e2:   c4 c2 05 b8 c2           vfmadd231ps %ymm10,%ymm15,%ymm0
4068e7:   c4 c2 05 bc c4           vfnmadd231ps %ymm12,%ymm15,%ymm0
4068ec:   c4 c2 5d bc c2           vfnmadd231ps %ymm10,%ymm4,%ymm0
4068f1:   c4 62 7d 18 15 ce 98     vbroadcastss 0x98ce(%rip),%ymm10
4068fa:   c4 e2 25 bc c1           vfnmadd231ps %ymm1,%ymm11,%ymm0
4068ff:   c4 c1 7c 54 d2           vandps %ymm10,%ymm0,%ymm2
406904:   c5 ec c2 3d 73 97 00     vcmpltps 0x9773(%rip),%ymm2,%ymm7
40690d:   c5 44 57 15 8b 97 00     vxorps 0x978b(%rip),%ymm7,%ymm10
406915:   c4 41 3d 76              (bad)
406919:   c0 c4 c1                 rol    $0xc1,%ah
40691c:   44 55                    rex.R push %rbp
40691e:   1c 81                    sbb    $0x81,%al
406920:   c4 c1 44 55 2c 84        vandnps (%r12,%rax,4),%ymm7,%ymm5
406926:   c4 c1 44 55 74 85 00     vandnps 0x0(%r13,%rax,4),%ymm7,%ymm6
40692d:   c5 fc 11 9c 24 a0 00     vmovups %ymm3,0xa0(%rsp)
406936:   c5 fc 11 6c 24 60        vmovups %ymm5,0x60(%rsp)
40693c:   c5 fc 11 b4 24 80 00     vmovups %ymm6,0x80(%rsp)
406945:   c4 42 7d 17 d0           vptest %ymm8,%ymm10
40694a:   0f 84 af 02 00 00        je     406bff
406950:   c5 7c 53 f0              vrcpps %ymm0,%ymm14
406954:   c5 fc 10 bc 24 20 01     vmovups 0x120(%rsp),%ymm7
40695d:   c5 fc 10 ac 24 e0 00     vmovups 0xe0(%rsp),%ymm5
406966:   c5 fc 28 d9              vmovaps %ymm1,%ymm3
40696a:   c5 fc 10 8c 24 00 01     vmovups 0x100(%rsp),%ymm1
406973:   c5 0c 59 c8              vmulps %ymm0,%ymm14,%ymm9
406977:   c4 c1 1c 59 c5           vmulps %ymm13,%ymm12,%ymm0
40697c:   c5 9c 59 f7              vmulps %ymm7,%ymm12,%ymm6
406980:   c5 64 59 c5              vmulps %ymm5,%ymm3,%ymm8
406984:   c5 f4 59 d7              vmulps %ymm7,%ymm1,%ymm2
406988:   c5 f4 59 e5              vmulps %ymm5,%ymm1,%ymm4
40698c:   c4 e2 1d ba c7           vfmsub231ps %ymm7,%ymm12,%ymm0
406991:   c4 41 0c 58 fe           vaddps %ymm14,%ymm14,%ymm15
406996:   c4 c2 65 ba d5           vfmsub231ps %ymm13,%ymm3,%ymm2
40699b:   c4 62 15 aa e4           vfmsub213ps %ymm4,%ymm13,%ymm12
4069a0:   c4 42 0d ac cf           vfnmadd213ps %ymm15,%ymm14,%ymm9
4069a5:   c5 14 5c f5              vsubps %ymm5,%ymm13,%ymm14
4069a9:   c5 44 5c fd              vsubps %ymm5,%ymm7,%ymm15
4069ad:   c4 62 65 a8 f0           vfmadd213ps %ymm0,%ymm3,%ymm14
4069b2:   c5 fc 10 84 24 a0 00     vmovups 0xa0(%rsp),%ymm0
4069bb:   c4 42 75 aa fe           vfmsub213ps %ymm14,%ymm1,%ymm15
4069c0:   c4 c1 04 59 c9           vmulps %ymm9,%ymm15,%ymm1
4069c5:   c4 63 7d 4a f1 a0        vblendvps %ymm10,%ymm1,%ymm0,%ymm14
4069cb:   c5 fc 10 8c 24 40 01     vmovups 0x140(%rsp),%ymm1
4069d4:   c5 fc 10 84 24 c0 00     vmovups 0xc0(%rsp),%ymm0
4069dd:   c4 41 7c 11 34 81        vmovups %ymm14,(%r9,%rax,4)
4069e3:   c5 74 59 ff              vmulps %ymm7,%ymm1,%ymm15
4069e7:   c4 42 55 b8 fb           vfmadd231ps %ymm11,%ymm5,%ymm15
4069ec:   c4 62 15 b8 f8           vfmadd231ps %ymm0,%ymm13,%ymm15
```

```
4069f1:   c4 62 55 bc f8          vfnmadd231ps %ymm0,%ymm5,%ymm15
4069f6:   c5 fc 10 6c 24 60       vmovups 0x60(%rsp),%ymm5
4069fc:   c4 62 15 bc f9          vfnmadd231ps %ymm1,%ymm13,%ymm15
406a01:   c4 41 4c 59 eb          vmulps %ymm11,%ymm6,%ymm13
406a06:   c4 42 45 bc fb          vfnmadd231ps %ymm11,%ymm7,%ymm15
406a0b:   c4 42 7d aa e5          vfmsub213ps %ymm13,%ymm0,%ymm12
406a10:   c4 c1 34 59 ff          vmulps %ymm15,%ymm9,%ymm7
406a15:   c4 c2 75 aa d4          vfmsub213ps %ymm12,%ymm1,%ymm2
406a1a:   c4 e3 55 4a ef a0       vblendvps %ymm10,%ymm7,%ymm5,%ymm5
406a20:   c4 62 3d ac da          vfnmadd213ps %ymm2,%ymm8,%ymm11
406a25:   c4 c1 7c 11 2c 84       vmovups %ymm5,(%r12,%rax,4)
406a2b:   c4 41 34 59 e3          vmulps %ymm11,%ymm9,%ymm12
406a30:   c5 7c 10 9c 24 80 00    vmovups 0x80(%rsp),%ymm11
406a39:   c4 43 25 4a d4 a0       vblendvps %ymm10,%ymm12,%ymm11,%ymm10
406a3f:   c4 41 7c 11 54 85 00    vmovups %ymm10,0x0(%r13,%rax,4)
406a46:   48 83 c0 08             add     $0x8,%rax
406a4a:   48 3b c7                cmp     %rdi,%rax
406a4d:   0f 82 e5 fd ff ff       jb      406838
```

Listing 11: `solveParabola` loop assembly (vectorized *SoA*)

D.3 Intrinsics *SoA*

```
406f44:   c4 21 7c 10 3c be      vmovups (%rsi,%r15,4),%ymm15
406f4a:   c4 81 7c 10 0c b8      vmovups (%r8,%r15,4),%ymm1
406f50:   c5 fc 10 7c 24 20      vmovups 0x20(%rsp),%ymm7
406f56:   c5 fc 10 2d 62 91 00   vmovups 0x9162(%rip),%ymm5
406f5e:   c5 fc 10 54 24 40      vmovups 0x40(%rsp),%ymm2
406f64:   c4 81 7c 10 1c b9      vmovups (%r9,%r15,4),%ymm3
406f6a:   c4 21 7c 10 34 ba      vmovups (%rdx,%r15,4),%ymm14
406f70:   c4 21 7c 10 2c bb      vmovups (%rbx,%r15,4),%ymm13
406f76:   c4 21 7c 10 24 b8      vmovups (%rax,%r15,4),%ymm12
406f7c:   c5 04 5c df            vsubps %ymm7,%ymm15,%ymm11
406f80:   c5 74 5c d7            vsubps %ymm7,%ymm1,%ymm10
406f84:   c5 64 5c c7            vsubps %ymm7,%ymm3,%ymm8
406f88:   c4 c1 24 59 f3         vmulps %ymm11,%ymm11,%ymm6
406f8d:   c4 c1 2c 59 e2         vmulps %ymm10,%ymm10,%ymm4
406f92:   c4 41 3c 59 f8         vmulps %ymm8,%ymm8,%ymm15
406f97:   c5 7c 28 cd            vmovaps %ymm5,%ymm9
406f9b:   c5 fc 28 c5            vmovaps %ymm5,%ymm0
406f9f:   c4 62 25 b8 ca         vfmadd231ps %ymm2,%ymm11,%ymm9
406fa4:   c4 e2 2d b8 c2         vfmadd231ps %ymm2,%ymm10,%ymm0
406fa9:   c4 c1 4c 59 f9         vmulps %ymm9,%ymm6,%ymm7
406fae:   c5 dc 59 f0            vmulps %ymm0,%ymm4,%ymm6
406fb2:   c4 c1 2c 59 c5         vmulps %ymm13,%ymm10,%ymm0
406fb7:   c5 bc 59 de            vmulps %ymm6,%ymm8,%ymm3
406fbb:   c5 fc 28 cd            vmovaps %ymm5,%ymm1
406fbf:   c4 e2 3d b8 ca         vfmadd231ps %ymm2,%ymm8,%ymm1
406fc4:   c4 c1 2c 59 d1         vmulps %ymm9,%ymm10,%ymm2
406fc9:   c5 84 59 e1            vmulps %ymm1,%ymm15,%ymm4
406fcd:   c4 c1 24 59 ce         vmulps %ymm14,%ymm11,%ymm1
406fd2:   c4 e2 25 a8 d4         vfmadd213ps %ymm4,%ymm11,%ymm2
406fd7:   c4 c2 5d ba da         vfmsub231ps %ymm10,%ymm4,%ymm3
406fdc:   c4 c2 1d ba ca         vfmsub231ps %ymm10,%ymm12,%ymm1
406fe1:   c4 e2 25 aa d3         vfmsub213ps %ymm3,%ymm11,%ymm2
406fe6:   c4 c2 45 ba d0         vfmsub231ps %ymm8,%ymm7,%ymm2
406feb:   c4 c2 4d be d3         vfnmsub231ps %ymm11,%ymm6,%ymm2
406ff0:   c5 d4 5e da            vdivps %ymm2,%ymm5,%ymm3
406ff4:   c4 c1 3c 59 ee         vmulps %ymm14,%ymm8,%ymm5
406ff9:   c5 fc 11 54 24 60      vmovups %ymm2,0x60(%rsp)
406fff:   c4 c1 24 59 d5         vmulps %ymm13,%ymm11,%ymm2
407004:   c5 44 59 fd            vmulps %ymm5,%ymm7,%ymm15
407008:   c4 c1 7c 28 e8         vmovaps %ymm8,%ymm5
40700d:   c4 e2 1d aa ea         vfmsub213ps %ymm2,%ymm12,%ymm5
407012:   c4 62 7d ba ff         vfmsub231ps %ymm7,%ymm0,%ymm15
407017:   c4 c1 24 59 c1         vmulps %ymm9,%ymm11,%ymm0
40701c:   c4 41 4c 59 cd         vmulps %ymm13,%ymm6,%ymm9
407021:   c4 c2 4d a8 ef         vfmadd213ps %ymm15,%ymm6,%ymm5
407026:   c4 e2 25 aa c4         vfmsub213ps %ymm4,%ymm11,%ymm0
40702b:   c4 62 0d aa da         vfmsub213ps %ymm2,%ymm14,%ymm11
407030:   c4 e2 5d ac cd         vfnmadd213ps %ymm5,%ymm4,%ymm1
407035:   c4 c2 1d a8 e1         vfmadd213ps %ymm9,%ymm12,%ymm4
40703a:   c4 e2 0d a8 c4         vfmadd213ps %ymm4,%ymm14,%ymm0
40703f:   c4 e2 1d aa f0         vfmsub213ps %ymm0,%ymm12,%ymm6
407044:   c5 e4 59 c1            vmulps %ymm1,%ymm3,%ymm0
407048:   c4 e2 15 ae fe         vfnmsub213ps %ymm6,%ymm13,%ymm7
40704d:   c4 41 14 5c ec         vsubps %ymm12,%ymm13,%ymm13
407052:   c4 41 0c 5c e4         vsubps %ymm12,%ymm14,%ymm12
407057:   c5 64 59 f7            vmulps %ymm7,%ymm3,%ymm14
40705b:   c4 42 3d aa e3         vfmsub213ps %ymm11,%ymm8,%ymm12
407060:   c5 7c 10 05 78 90 00   vmovups 0x9078(%rip),%ymm8
407068:   c4 42 2d ac ec         vfnmadd213ps %ymm12,%ymm10,%ymm13
40706d:   c5 14 59 db            vmulps %ymm3,%ymm13,%ymm11
407071:   c5 3c 55 54 24 60      vandnps 0x60(%rsp),%ymm8,%ymm10
407077:   c5 2c c2 05 80 90 00   vcmpltps 0x9080(%rip),%ymm10,%ymm8
407080:   c4 c1 3c 55 cb         vandnps %ymm11,%ymm8,%ymm1
407085:   c4 c1 3c 55 d6         vandnps %ymm14,%ymm8,%ymm2
40708a:   c5 bc 55 d8            vandnps %ymm0,%ymm8,%ymm3
40708e:   c4 81 7c 2b 0c ba      vmovntps %ymm1,(%r10,%r15,4)
407094:   c4 81 7c 2b 14 be      vmovntps %ymm2,(%r14,%r15,4)
40709a:   c4 81 7c 2b 5c bd 00   vmovntps %ymm3,0x0(%r13,%r15,4)
4070a1:   49 83 c7 08            add    $0x8,%r15
4070a5:   4d 3b fb               cmp    %r11,%r15
4070a8:   0f 8c 96 fe ff ff      jl     406f44
```

Listing 12: `solveParabola` loop assembly (*SIMD SoA*)

E Summation functions

E.1 Naive summation

```
mpreal add(mpreal* a, int n) {
  int i;
  mpreal s = 0;
  for (i = 0; i < n; ++i) {
    s += a[i];
  }
  return s;
}
```

Listing 13: Naive summation

E.2 Accurate summation

```
mpreal add2(mpreal* a, int n) {
  int i, j;
  mpreal temp;
  for (i = 0; i < n-1; ++i) {
    // find the smallest element
    for (j = i+1; j < n; ++j) {
      if (abs(a[j]) < abs(a[i])) {
        // swap(a[i], a[j])
        temp = a[j]; a[j] = a[i]; a[i] = temp;
      }
    }
    // find the second smallest element
    for (j = i+2; j < n; ++j) {
      if (abs(a[j]) < abs(a[i+1])) {
        // swap(a[i+1], a[j])
        temp = a[j]; a[j] = a[i+1]; a[i+1] = temp;
      }
    }
    // put the result in the array
    a[i+1] += a[i];
  }
  return a[n-1];
}
```

Listing 14: Accurate summation

F fitParabola code

```
void fitParabola(const TrackAOS* track, resultAOS* result) {
  mpreal mat[6], rhs[3], ay, by;
  int loop, i;
  result->chi2Track = result->maxChi2 = result->maxDistance
    = result->absdistanceSum = result->distanceSum
    = result->dRatio = result->X0 = 0.0/0.0;

  result->dRatio = track->dRatio;
  result->ax = track->ax;
  result->bx = track->bx;
  result->cx = track->cx;
  ay = track->ay;
  by = track->by;

  if (track->nbHits < global->minXplanes) return;

  for (loop = 0; loop < 3; ++loop) {
    mat[0] = 0;
    mat[1] = 0; mat[2] = 0;
    mat[3] = 0; mat[4] = 0; mat[5] = 0;
    rhs[0] = 0; rhs[1] = 0; rhs[2] = 0;
    for (i = 0; i < track->nbHits; ++i) {
      mpreal w = track->hits[i].w;
      mpreal dz = track->hits[i].z0 - global->zRef;
      mpreal deta;
      if (global->useCubic) {
        deta = dz*dz*(1+result->dRatio*dz);
      } else {
        deta = dz*dz;
      }
      mpreal dist = distance_Track(
          track->hits[i].x0, track->hits[i].z0,
          track->hits[i].dxdy, track->hits[i].dzdy,
          result->ax, result->bx, result->cx, ay, by,
          result->dRatio, track->zRef
      );

      mat[0] += w;
      mat[1] += w*dz;   mat[2] += w*dz*dz;
      mat[3] += w*deta; mat[4] += w*dz*deta; mat[5] += w*deta*deta;
      //right hand side
      rhs[0] += w*dist;
      rhs[1] += w*dist*dz;
      rhs[2] += w*dist*deta;
    }
    if (!CholeskySolve(mat, rhs, 3)) {
      return;
    }
    if (abd(rhs[0]) > 1e4 || abs(rhs[1]) > 5. || abs(rhs[2]) > 1e-3) {
      return;
    }
    result->ax += rhs[0];
    result->bx += rhs[1];
    result->cx += rhs[2];

    result->dRatio =    global->dRatio[0]
                      + global->dRatio[1]*rhs[0]
                      + global->dRatio[2]*rhs[0]*rhs[0];
    if (loop > 0 && abs(rhs[0]) < 5e-3 &&
        abs(rhs[1]) < 5e-6 && abs(rhs[2]) < 5e-9) {
      break;
    }
  }
  //Fit is done
  //track.assoc is set here

  //Compute some values on the track
  result->chi2Track = 0.;
  result->maxChi2 = 0.;
  result->maxDistance = 0.;
  result->absdistanceSum = 0.;
  result->distanceSum =0.;
  for (i = 0; i < track->nbHits; ++i) {
    mpreal dist = distance_Track(
        track->hits[i].x0, track->hits[i].z0,
        track->hits[i].dxdy, track->hits[i].dzdy,
        result->ax, result->bx, result->cx, ay, by,
        result->dRatio, track->zRef
    );
    mpreal chi2_onHit = dist*dist * track->hits[i].w;
    result->absdistanceSum += abs(dist);
    result->distanceSum += dist;
    result->maxDistance = max(result->maxDistance, abs(dist));
    result->chi2Track += chi2_onHit;
    result->maxChi2 = max(result->maxChi2, chi2_onHit);
  }
  result->distanceSum /= track->nbHits;
  result->absdistanceSum /= track->nbHits;
  result->X0 = result->ax - result->bx * global->zRef +
               result->cx * global->constC;
}
```

Listing 15: `fitParabola` function

G `fitParabola` accuracy with abacus

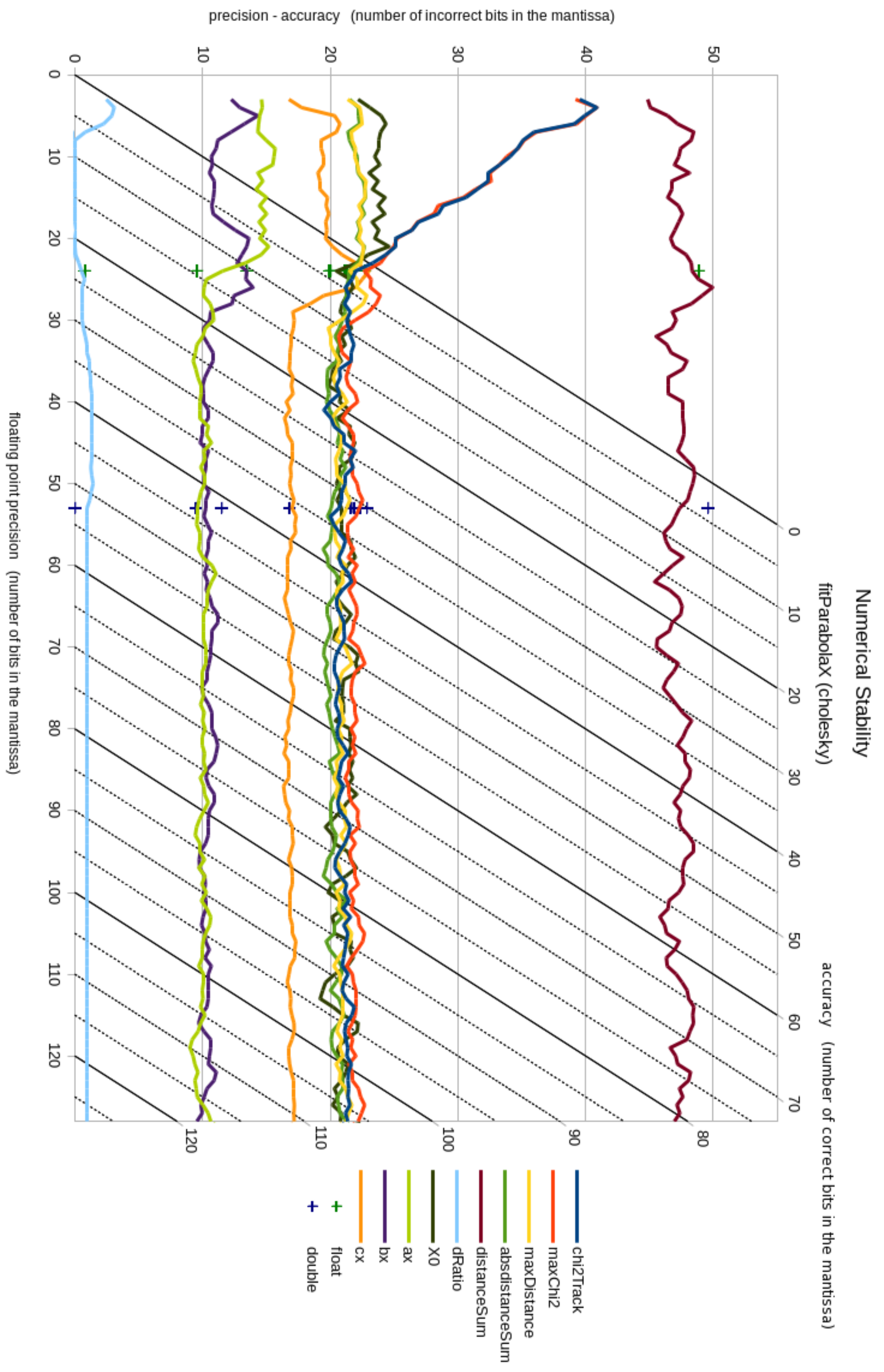

Chart 14: `fitParabola` variables accuracy with abacus (max error)

H Linear system solvers used in fitParabola

H.1 Cholesky

```
// mat is stored in a triangular way (so mat(i, j) = mat[i*(i+1)/2+j])
int solveCholeskyN(const float* mat, float* rhs, int n) {
  int i, j, k;
  float s, *L = new float[n*n];

  // decompose mat into L
  for (i = 0; i < n; ++i) {
    for (j = 0; j <= i; ++j) {
      s = 0;
      for (k = 0; k < j; ++k) {
        s += L[i * n + k] * L[j * n + k];
      }
      if (i == j) {
        L[i * n + j] = sqrt(mat[i*(i+1)/2+i] - s);
      } else {
        L[i * n + j] = (1.0 / L[j*n + j] * (mat[i*(i+1)/2+j] - s));
      }
    }
  }

  // Check results
  for (i = 0; i < n; ++i) {
    if (L[i*n + i] == 0) {
      delete[] L;
      return 0;
    }
  }

  // solving LY = rhs
  for (i = 0; i < n; ++i) {
    s = rhs[i];
    for (j = 0; j < i; ++j) {
      s -= L[i*n + j] * rhs[j];
    }
    rhs[i] = s / L[i*n + i];
  }
  // solving Ltx = Y
  for (i = n-1; i >= 0; --i) {
    s = rhs[i];
    for (j = n-1; j > i; --j) {
      s -= L[j*n + i] * rhs[j];
    }
    rhs[i] = s / L[i*n + i];
  }
  delete[] L;
  return 1;
}
```

Listing 16: Cholesky solver

H.2 Cramer's Rules

```
// mat is stored in a triangular way (so mat(i, j) = mat[i*(i+1)/2+j])
int solveCramer3(const float* mat, float* rhs) {
  float det, det0, det1, det2;

  // Compute det(mat)
  det =  mat[0]*mat[2]*mat[5] + mat[3]*mat[1]*mat[4]
       + mat[1]*mat[4]*mat[3] - mat[3]*mat[2]*mat[3]
       - mat[0]*mat[4]*mat[4] - mat[1]*mat[1]*mat[5];

  // Compute det(rhs|mat(1:2,:))
  det0 = rhs[0]*mat[2]*mat[5] + rhs[2]*mat[1]*mat[4]
       + rhs[1]*mat[4]*mat[3] - rhs[2]*mat[2]*mat[3]
       - rhs[0]*mat[4]*mat[4] - rhs[1]*mat[1]*mat[5];

  // Compute det(mat(1,:)rhs|mat(2,:))
  det1 = mat[0]*rhs[1]*mat[5] + mat[3]*rhs[0]*mat[4]
       + mat[1]*rhs[2]*mat[3] - mat[3]*rhs[1]*mat[3]
       - mat[0]*rhs[2]*mat[4] - mat[1]*rhs[0]*mat[5];

  // Compute det(mat(0:1,:)|rhs)
  det2 = mat[0]*mat[2]*rhs[2] + mat[3]*mat[1]*rhs[1]
       + mat[1]*mat[4]*rhs[0] - mat[3]*mat[2]*rhs[0]
       - mat[0]*mat[4]*rhs[1] - mat[1]*mat[1]*rhs[2];

  // Cramer's rules
  rhs[0] = det0 / det;
  rhs[1] = det1 / det;
  rhs[2] = det2 / det;
  return 1;
}
```

Listing 17: Cramer's rule solver