Employing HPC for Heterogeneous HEP Data Processing

AUGUST 2018

AUTHOR:

Marco
Barbone

CERN IT-DI-OPL
DEEP-EST

SUPERVISORS:

Viktor Khristenko
Felice Pantaleo
Maria Girone

CERN
openlab

Project Specification

The DEEP-EST Project aims to build a Modular Supercomputer Architecture (MSA) with the main focus on the usage of heterogenous computing resources. The majority of current HEP workflows are capable of utilizing exclusively Central Processing Units (CPU), which create inefficiency when trying to run on heterogenous systems equipped with additional hardware accelerators (GPUs, FPGAs).

Within the context of this project, we want to extend current CPU specific implementations of various algorithms (calorimeter reconstruction, track fitting, etc.) by utilizing OpenCL and CUDA language extensions in order to harness the computational power provided by various hardware accelerators.

Abstract

One of the most time consuming algorithms that is currently employed for the reconstruction of **High Energy Physics** (HEP) workflows is the local energy reconstruction. The time spent to execute this algorithm constitutes 24% of the total processing time, thus achieving substantial speedup by optimizing this problem will noticeably influence the total processing time. Even a speedup of a factor of 1.5 will shorten the processing time by about 5%. The purpose of this project is to dig deep into both algorithmic and architecture dependent optimizations, trying to best exploit heterogenous resources to maximize **High Performance Computing** (HPC) utilization.

Contents

List of Figures

List of Tables

1. From physics to... Physics

Modern **High Energy Physics** (HEP) experiments require complex multistage infrastructure. To realize them several large scale facilities are needed: from accelerators, detectors, data centers to the thousands of people involved to design and run the infrastructure.

The entry point of any **Large Hadron Collider** (LHC) experiment is the collision between proton-proton beams (although Pb beams are used) which triggers various physical processes. There are two kinds of processes, those that are not yet observed, called **Signal** (S) and the observed ones, called **Background** (B). The objective of LHC experiments is to expand our knowledge about elementary physical processes governing the universe by studying **S** through the already observed particles that it produces. Unfortunately, interesting processes are quite rare, thus a huge amount of collisions are needed to produce one of them.
This creates a lot of difficult and interesting IT challenges because everything from data acquisition to data analysis has to scale to meet throughput and timing requirements. The faster data are processed, more collisions can be triggered therefore, quicker the research can proceed.
As computer scientists, our purpose here at CERN is to accelerate data processing by best exploiting the resources at our disposal, achieving the maximum efficiency using smart and creative approaches, in particular targeting **High Performance Computing** (HPC) environment.
The LHC infrastructure is designed to process huge amounts of data (hundreds of Pb/s). The data flow architecture illustrated in figure 1.1 is organized in four main stages. Staring from physical processes to the physics carried out by analyzing the data, with data acquisition and processing in the middle. Each of these steps is complex and deserves a dedicated explanation.

Data Generation The process starts with beams of **protons** colliding every $25ns$ that leads to particle interactions that create some **intermediate products**. These intermediate products can not be directly observed, but they further decay into something observable by the detectors.

Data Acquisition Detectors are split in two levels: **physics detector level** and **electronics level**. The physics level takes as input sensor readouts and produces an analog signal. This analog signal is digitized by the electronics level. At this point there is too much data to send it directly to the next stage. Models are used to distinguish between which data that belongs to S and which one belongs to B. Since these models are complex and the timing constraints are very strict, only a simplified version of them is implemented directly in hardware, on top of **Application Specific Integrated Circuit**s (ASIC) and **Field Programmable Gate Array**s (FPGA) boards. Because models here are simplified there are a lot of B events labeled as S.
The events that survive this first stage are sent to the **High Level Trigger** (HLT).
The HLT has more relaxed time constraints since it receives less events and its job is to distinguish between S and B exploiting, an implementation of the same models used inside the previous stage, but with more features that allows more precise evaluations. Another important characteristic of the HLT is that it is implemented in software and executed on a CPU based cluster. Currently there is an attempt to integrate and exploit accelerators such as GPUs and FPGAs inside this cluster.

Data Processing At this point, events that pass previously described two level trigger system are used to generate a dataset used for analytics purposes. To build this dataset, the same models deployed inside the triggers are used, but with way more features. To extrapolate useful information and build the dataset **raw data** are sent through a complex multistage processing pipeline. This pipeline is offline so it does not share the timing constraints of the previous stage but, efficient exploitation of resources is needed because even after all the filtering data is still huge and models are very complex.

The first stage, called **local reconstruction**, transforms charges to physical quantities such as **energy**, **time**, and **position**. This operation is done channel by channel. These channels are independent form each other thus, this is an embarrassingly parallel problem. This parallelism is exploited though multi-threading by running this code on a cluster of multi-core machines, although other parallel architectures, such as GPUs, are currently tested to deploy them in future upgrades. Then, information coming from different channels of the same detector are combined by the **cross channel clustering** into **local detector cluster**s. The final step needed to obtain **particle object**s is the **cross detector clustering** in which data coming from different detector is combined. In the end, some particle objects are built. The set of all these particle objects form the dataset used to probe new horizons of fundamental physics.

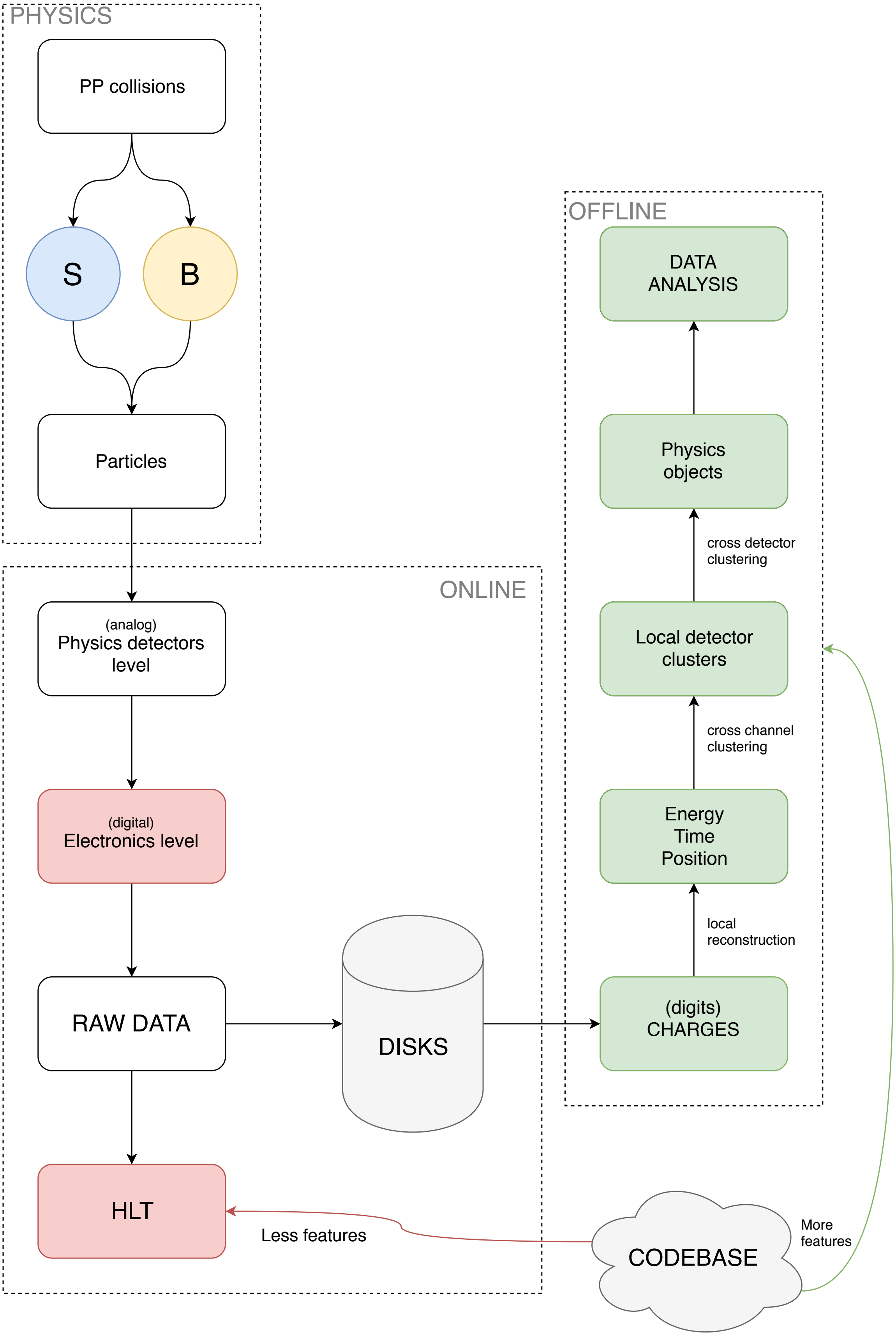


Figure 1.1: Cern data flow from collisions to analysis

2. Local energy reconstruction

The HLT shares the same code used for offline data processing, thus the processing pipeline is more or less the same. The fundamental difference is that the output is used to perform event classification instead of data analysis. The table 2.1 shows how much time is spent in every single step of the reconstruction process. Most of it is spent into tracking but after it, the second more time consuming step is HCAL+ECAL local reconstruction that takes $113ms$ corresponding to 24% of the total time. Given the time needed to perform this reconstruction even achieving a speedup of two would reduce the total processing time by more than 10%. This is the focus of this project: try to reduce it as much as possible by both optimizing it for CPU and exploiting GPU resources.

Figure 2.1: HLT processing pipeline

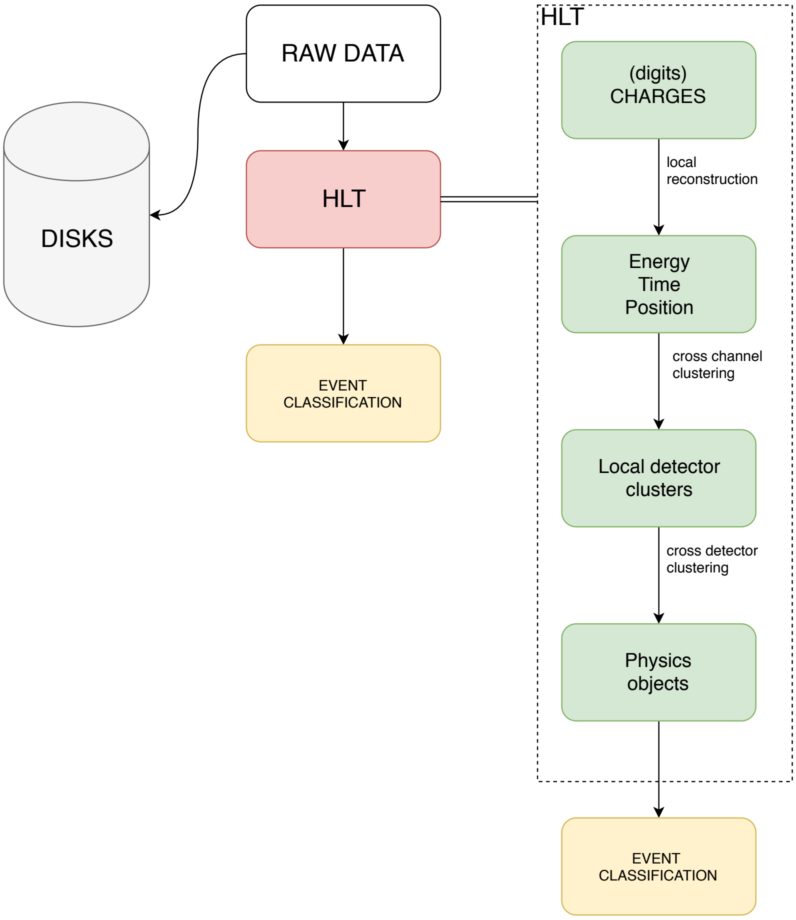


Figure 2.2: Data processing time share

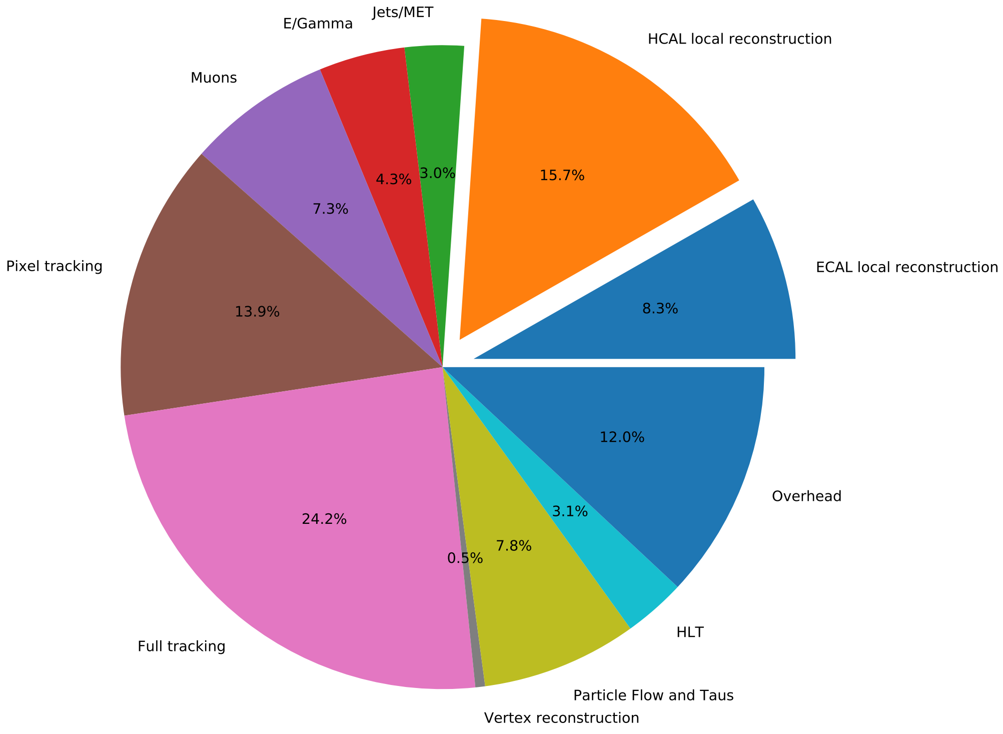


Table 2.1: Time spent into the various HLT reconstruction steps

Step	Real-Time	Percentage
ECAL local reconstruction	38.9 ms	8.25%
HCAL local reconstruction	73.9 ms	15.67%
Jets/MET	14 ms	2.97%
E/Gamma	20.4 ms	4.33%
Muons	34.2 ms	7.25%
Pixel tracking	65.7 ms	13.93%
Full tracking	114.2 ms	24.22%
Vertex reconstruction	2.3 ms	0.49%
Particle Flow and Taus	36.8 ms	7.8%
HLT	14.7 ms	3.12%
Overhead	56.4 ms	11.96%
Total	471.5 ms	100%

2.1 Problem statement

For each channel $\{$given n charge readouts $\rightarrow$ reconstruct the energy$\}$.

$$\begin{aligned}
&\min(\chi^2) = \arg min_x(\|Px - b\|^2) \\
&\forall i : x_i \geq 0 \\
&\text{where:} \\
&x = \text{energy vector} \\
&P : ENERGY \rightarrow CHARGE = \text{feature matrix} \\
&b = \text{charge vector} \\
&n = 10 \text{ (in this particular case)}
\end{aligned} \tag{2.1}$$

Which is a standard χ^2 problem with additional positivity constraints. This constraint is present because physically speaking negative energy does not make sense.
As shown in [1] the statement above is incomplete. A perfect mapping from charges to energy does not exist because signal from the shower does not dissipate within one time slice ($25ns$). Adding the correlation term this problem becomes:

$$\begin{aligned}
&\arg min_x(\left\|(Px - b)^T \Sigma(x)^{-1}(Px - b)\right\|^2) \\
&\forall x : x \geq 0
\end{aligned} \tag{2.2}$$

It is worth pointing out that Σ depends on x, meaning also Σ is unknown.
To compute Σ an iterative procedure is used:

1. Compute Σ.
2. Minimize χ^2.
3. If not convergence goto 1.

More precisely Σ is the covariance matrix representing the noise correlation between time samples i and j, obtained from data where no signal is present, and the single sample noise.
To solve the problem stated in 2.1 several algorithms exist, for example **lsqnonneg** illustrated in [4] and the **ffnls** illustrated in [2]. The one implemented is fnnls since as measured in [3] it is faster.
The problem presented in 2.2 is not a χ^2 problem but to solve it with nnls needs to be reduced into the canonical form. The redution exploits the Cholesky decomposition and is illustrated in 2.3.

$$\begin{aligned}
&(Px - b)^T \Sigma(x)^{-}1(Px - b) \\
\equiv\quad & \Sigma = LL^T, (AB)^{-1} = B^{-1}A^{-1} \\
&(Px - b)^T L^{-T} L^{-1}(Px - b) \\
\equiv\quad & (AB)^T = B^T A^T \\
&(L^{-1}Px - L^{-1}b)^T L^{-1}(Px - b) \\
\equiv\quad & \\
&(L^{-1}Px - L^{-1}b)^T (L^{-1}Px - L^{-1}b) \\
\equiv\quad & L^{-1}P = P', L^{-1}b = b' \\
&(P'x - b')^T (P'x - b')
\end{aligned} \tag{2.3}$$

2.2 Fast non negative least square algorithm (FNNLS)

The nnls is an *active set iterative algorithm*. It uses two sets:

- **Passive set (P)**: the constraint is ”passive”, meaning that it is not satisfied.

- **Active set (R)**: the constraint is ”active”, meaning that it is satisfied.

The pseudo-code presented in algorithm 1, starts with a feasible solution (line 2), then checks for the positivity constraint. If there are some negative components it finds a non negative one that minimize the error, exploiting a gradient (line 5). More details can be found in [4]

Algorithm 1 NNLS

```
Input:
    A real valued matrix of dimension $m \times n$
    b real valued vector of dimension $m$
    $\epsilon$ the maximum accepted error
    K the maximum number of iterations
Output:
    $x$ the solution vector

 1: function NNLS($A$, $b$, $\epsilon$, $K$)
 2:     $x \leftarrow 0$
 3:     $P = \varnothing$
 4:     $R = \{1, 2, ..., m\}$
 5:     $w = A^T(b - Ax)$                                   ▷ compute the gradient
 6:     while $R \neq \varnothing \wedge max(w) < \epsilon \wedge k < K$ do
 7:         $j \leftarrow max(w^P)$                             ▷ $w^P \leftarrow \{w_j : j \in P\}$
 8:         Add $j$ to $P$
 9:         Remove $j$ from $R$
10:         $A^P \leftarrow \{a_{ij} \in A : i \in P \wedge j \in P\}$
11:         $s \leftarrow ((A^P)^T A^P)^{-1} A^P b^P$         ▷ s is a vector of the same dimension of P
12:         while $min(s) \leq 0$ do
13:             $\alpha = min_i\{\frac{x_i}{x_i - s_i} : i \in P \wedge s_i \leq 0\}$
14:             $\forall i \in P : x_i \leftarrow x_i + \alpha(s_i - x_i)$
15:             move to $R$ all $i \in P : x_i = 0$
16:             $s \leftarrow ((A^P)^T A^P)^{-1} A^P b^P$     ▷ recompute s for the next iteration
17:         $\forall i \in P : x_i = s_i$
18:         $w \leftarrow A^T(b - Ax)$
19:         $k \leftarrow k + 1$
```

This algorithm is slow because at each iteration it requires to calculate the pseudo-inverse (line 11). FNNLS, showed in algorithm 2, is faster because it reduces the computational burden of this operation. The idea is simple: instead of projecting the matrix A over P and then performing the transposition and multiplication, it saves $A^T A$ and performs the projection over P. Another operation avoided is the multiplication between A and b, also in this case the multiplication is performed in the preprocessing phase. At runtime, only the projection over P is performed.
These improvements reduces the computational making the it to run faster.

Algorithm 2 FNNLS

```
$w \leftarrow (A^T b)(A^T A)x$
$s \leftarrow ((A^T A)^P)^{-1}(A^T b)^P$
```

2.3 Implementation details

This algorithm has several numerical issues coming from the pseudo-inverse computation (line 11 of pseudo-code 1 and line 2 of pseudo-code 2).
The original definition of the matrix A^P is $A^P = \{A.col(i) : \forall i \in P\} \cup \{0 : \forall i \in R\}$. While the definition of the vector b^P is $b^P = \{b_i : \forall i \in P\} \cup \{0 : \forall i \in R\}$. This results in a $m \times n$ matrix and a m dimensional vector. Calculating the pseudo-inverse with this matrix generates numerical issues, the result is a matrix containing some $-nan$.
One way to solve this problem is to reduce the number of operations needed to perform the computation. To achieve this there are three ways:

1. Invert the matrix through some decomposition.
2. Changing the definition of A^P to reduce the size of the matrix and the vector.
3. Combining both 1 and 2.

Decompositions From the decompositions present in this page [7], **Cholesky** and **HouseHolder** with and without pivoting have been tested. These decompositions have been chosen following the advices present on this page of Eigen documentation. As a result all of them except the HouseHolder with pivoting were numerically unstable. But, the HouseHolder with pivoting is not fast enough to meet the time constraints so, other approaches has been tested.

$\boldsymbol{A}^P$ Observing the computations performed utilizing the original definition can be noticed that a lot of useless operations containing 0 were present. Not only this is overhead but also increases the algorithmic error. Changing the definition to the one provided into the pseudo-code allowed Cholesky and also the plain inverse to work without issues.

In the end the Cholesky without pivoting, applied on the modified A^P matrix, was chosen because it is proven in [5] to perform best.

2.4 GPU porting

Until now there are two versions ported on GPU, the one taken from *cms-sw* codebase and the one implemented from scratch.
To exploit the parallelism provided by these devices a channel level parallelization is performed but, a dynamic parallelism inside the channel is to be studied.

2.5 Optimizations

There are two kinds of optimizations performed: numerical and algorithmic. In the first case some mathematical properties are exploited to reduce the number of operations, in the other case some architecture level knowledge is used to speedup the computation.

2.5.1 Numerical

The optimization performed here is to avoid using swap matrices and the P and R vectors. Studying the algorithm it is possible to notice that the P, R partitioning can be obtained in-place by permuting the matrix A.

$$\begin{array}{cc} & \xrightarrow{\text{\# Passive}} \\ \text{\# Passive}\ \Bigg\downarrow & \left[\begin{array}{cc|cc} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \hline \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{array}\right] \end{array} \tag{2.4}$$

As shown in 2.4 the passive set grows from the top left corner. The size of this block is the same as P. This way, instead of a vector, a counter is enough to save the active set. Every time a variable enters in the active set the corresponding column and row are swapped accordingly and the $nActive$ counter is incremented.

Both the vector b and x are permuted, because otherwise they would not be aligned with the matrix.

A permutation matrix is used to keep track of all the swapping and the solution is reordered before being returned.

This optimization allows to reduce both memory allocation/de-allocation and cache faults resulting in a performance improvement of a factor of 2, as showed in 3.1.

2.5.2 Algorithmic

The optimization performed here is exploiting some pragmas to vectorise fixed iteration loops and to unroll the others, which have non constant number of iterations.

From the profiling performed can be noted that in case of fixed iteration loops the vectorization gives better performance results with respect to the unroll. Also the compiler unroll vectorized loops combining the best of both approaches.

The unroll value has been determined empirically using the tool called **cachegrind**, using the method illustrated in this page. Exploiting measurements like branch misprediction and cache faults for each value and used the one that minimize them.

3. Results

3.1 Test01: GPU vs CPU

The results of this test are used as baseline for further optimizations.
Test configuration:

- **CPU**: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz
- **GPU**: NVIDIA Tesla K40c

Implementations tested:

- **legacy_multifit_cpu**: plain *cms-sw* cpu code.
- **legacy_multifit_gpu**: plain gpu porting *cms-sw* cpu code.
- **multifit_cpu**: inplace fnnls cpu implementation.
- **multifit_gpu**: inplace fnnls gpu implementation.
- **multifit_cpu_swap**: fnnls cpu implementation with swapping matrices.
- **multifit_gpu_swap**: fnnls gpu implementation with swapping matrices.

All the cpu implementations are single-threaded. With 64k channels the optimized GPU version achieves a speedup of 2.67. Since the corresponding CPU version achieves a speedup of 1.10, the GPU implementation runs more than twice as fast as the corresponding CPU one.

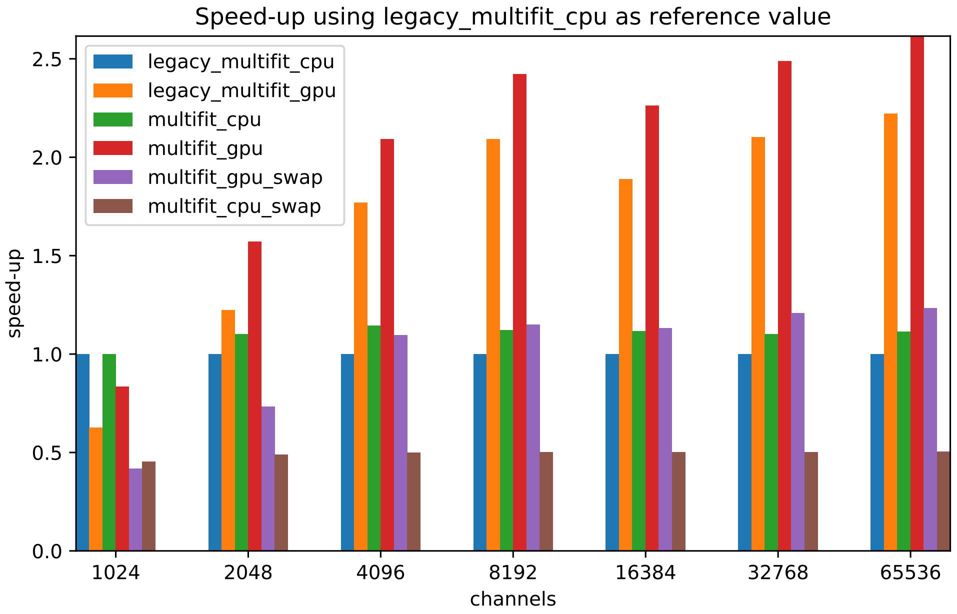


Figure 3.1: Speedup achieved with 10 iterations, higher is better

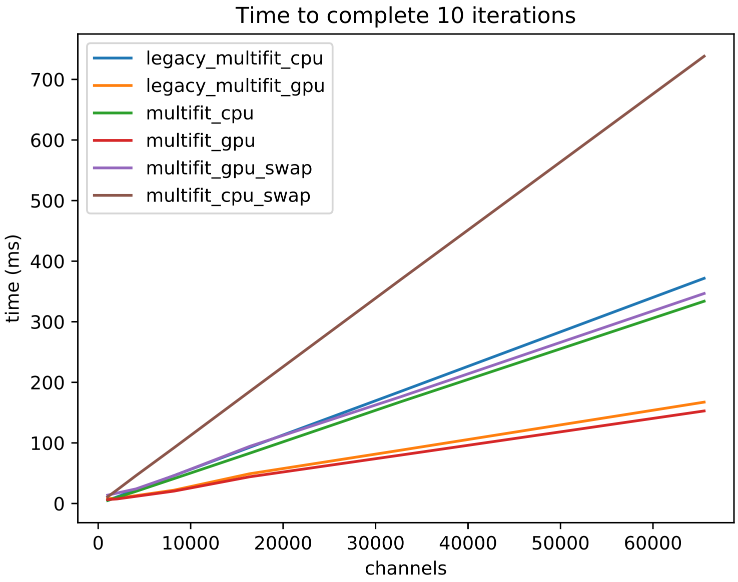


Figure 3.2: Time needed to complete 10 iterations, linear channel scale, lower is better

From 3.1 that the GPU performs almost six times as fast with respect to the CPU. Other optimizations like loop unrolling and branch reduction give a performance gain about 20% on CPU.

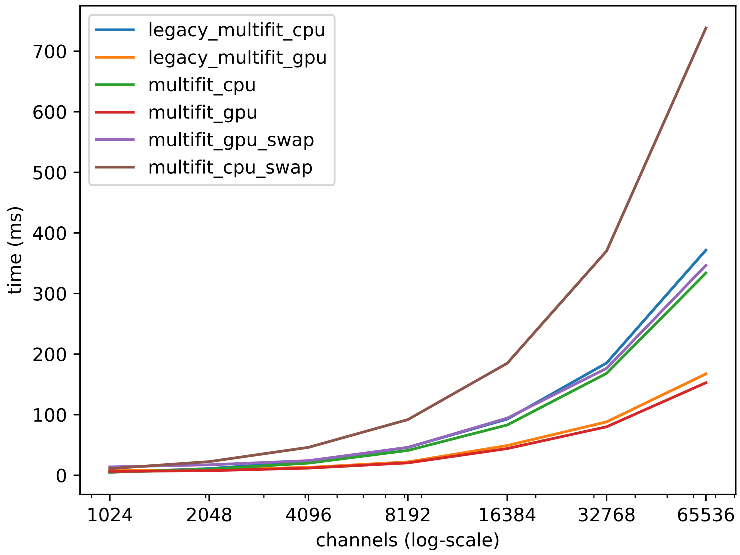


Figure 3.3: Time needed to complete 10 iterations, log channel scale, lower is better

From the plot present in 3.2 can be noted that in the GPU version there is a change of slope around 15k channels, after this number the growth slows-down. The plot in 3.3 makes more evident the difference between the different version of the algorithm. The GPU outperforms the CPU and since the growth is sub-linear, it will be interesting to study what will happen if the number of channels increases even more.

3.2 Test02: GPU vs CPU, Optimized Matrix multiplication

All the cpu implementations are single-threaded. With 64k channels the GPU version achieves a speedup of 2.67 that it the same as the previous test.

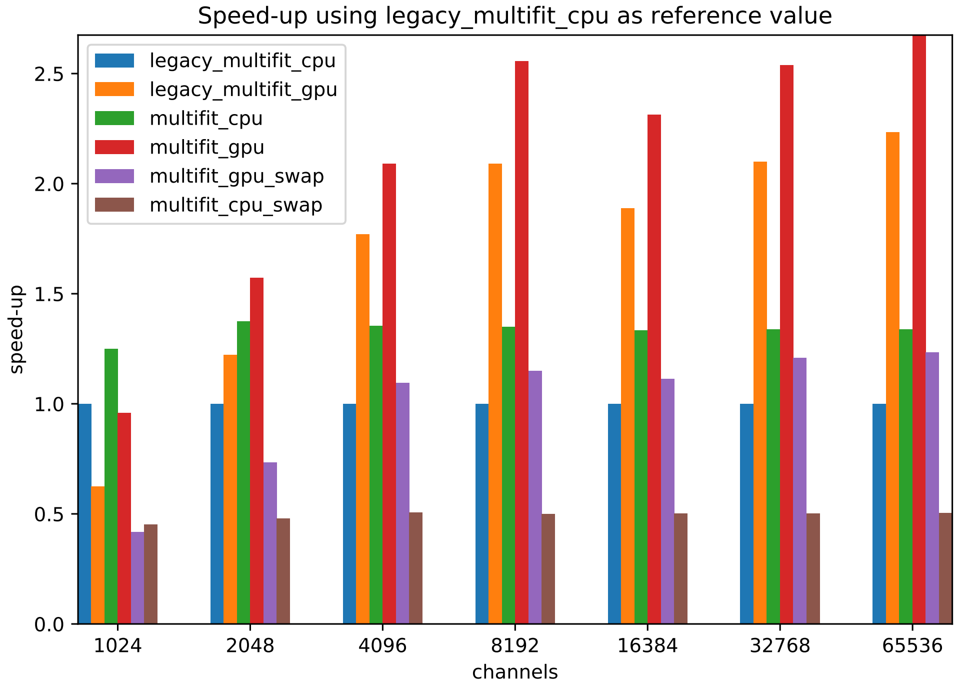


Figure 3.4: Speedup achieved with 10 iterations, higher is better

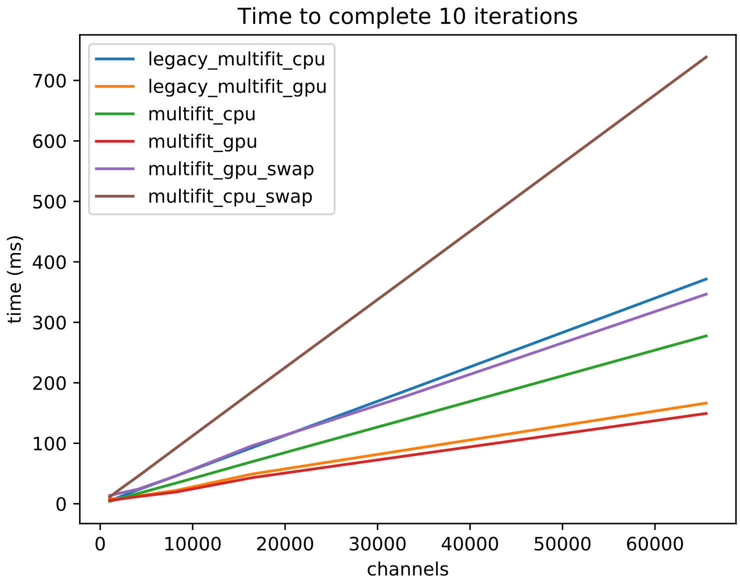


Figure 3.5: Time needed to complete 10 iterations, linear channel scale, lower is better

From 3.1 can be noted that even with all the optimizations the GPU speedup it does not increase. This is due to the fact that the GPU is underutilized so even if the implementation requires less resources it can not be noted in this test, and the data transfer respect to the computation is very big. The optimized CPU implementation in this case performs 35% better than the legacy one.

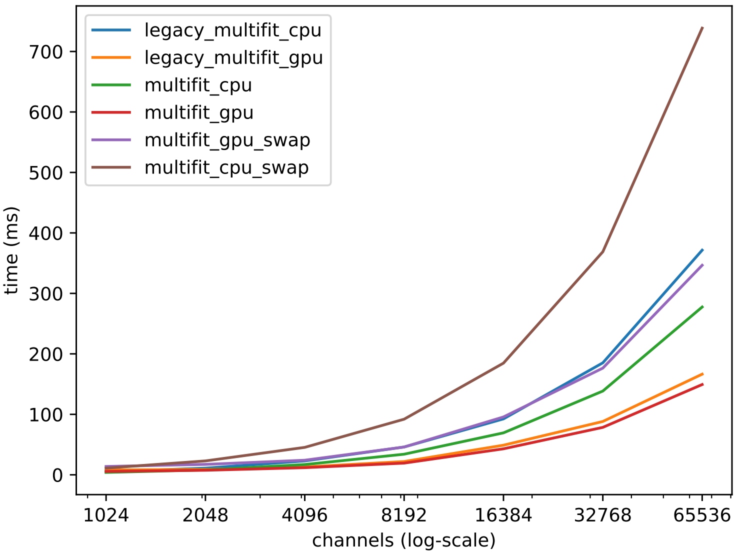


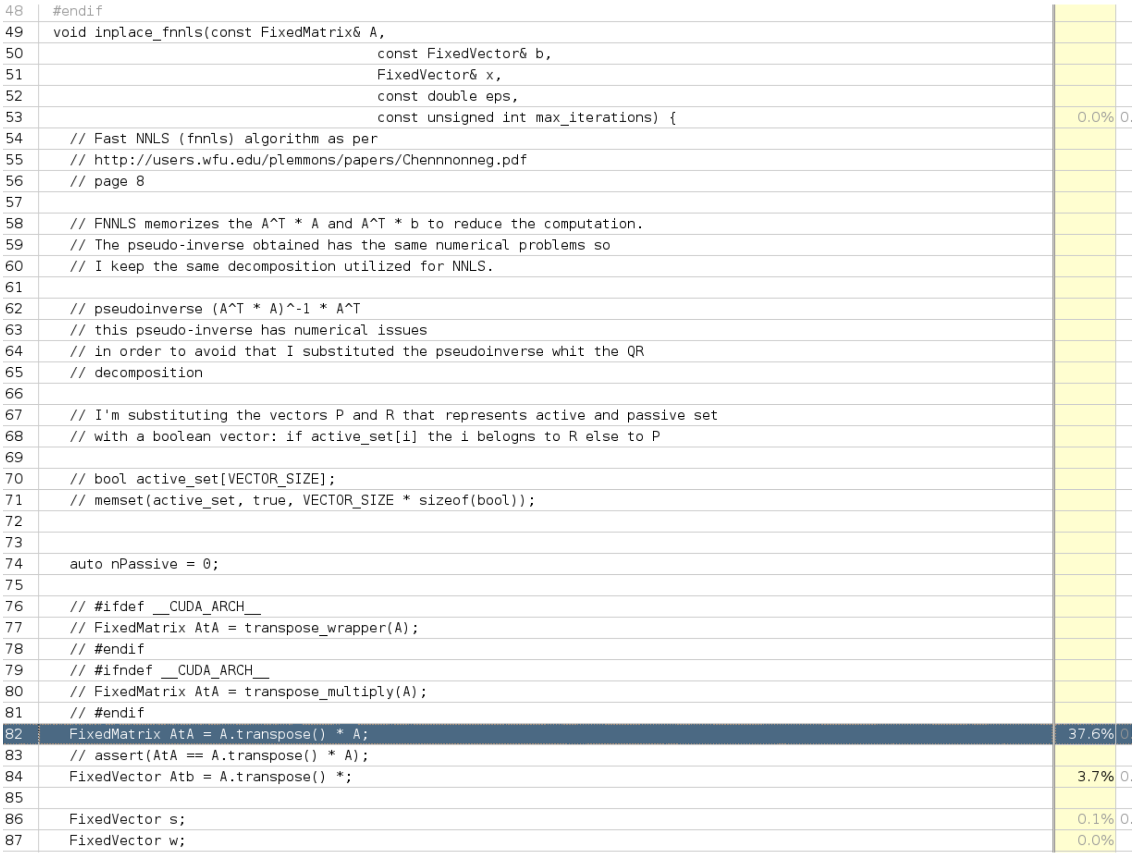
Figure 3.6: Time needed to complete 10 iterations, log channel scale, lower is better

4. Profiling and further optimizations

In order to further optimize the code some profiling is needed to identify the bottleneck and solve them... If possible.

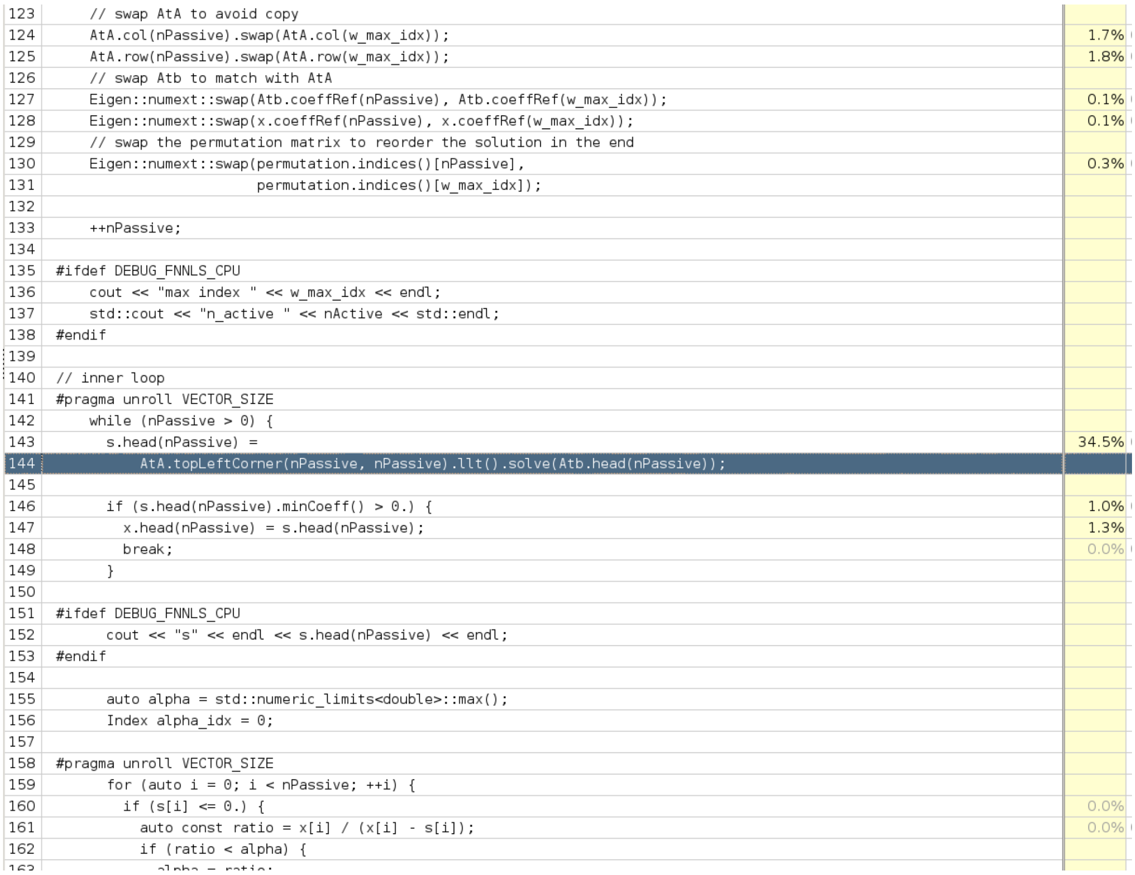
4.1 Finding CPU hotspots

The methodology followed is to use **Intel VTune** on the entire regression. Profiling only the nnls portion of the code did not give reliable results because terminates too fast. Profiling the entire regression instead, gave interesting results; As showed in 4.1 and 4.2 there are two hotspots in the code.



```
#endif
void inplace_fnnls(const FixedMatrix& A,
                                       const FixedVector& b,
                                       FixedVector& x,
                                       const double eps,
                                       const unsigned int max_iterations) {      0.0% 0.
  // Fast NNLS (fnnls) algorithm as per
  // http://users.wfu.edu/plemmons/papers/Chennnonneg.pdf
  // page 8

  // FNNLS memorizes the A^T * A and A^T * b to reduce the computation.
  // The pseudo-inverse obtained has the same numerical problems so
  // I keep the same decomposition utilized for NNLS.

  // pseudoinverse (A^T * A)^-1 * A^T
  // this pseudo-inverse has numerical issues
  // in order to avoid that I substituted the pseudoinverse whit the QR
  // decomposition

  // I'm substituting the vectors P and R that represents active and passive set
  // with a boolean vector: if active_set[i] the i belogns to R else to P

  // bool active_set[VECTOR_SIZE];
  // memset(active_set, true, VECTOR_SIZE * sizeof(bool));


  auto nPassive = 0;

  // #ifdef __CUDA_ARCH__
  // FixedMatrix AtA = transpose_wrapper(A);
  // #endif
  // #ifndef __CUDA_ARCH__
  // FixedMatrix AtA = transpose_multiply(A);
  // #endif
  FixedMatrix AtA = A.transpose() * A;                                          37.6% 0
  // assert(AtA == A.transpose() * A);
  FixedVector Atb = A.transpose() *;                                             3.7% 0.

  FixedVector s;                                                                 0.1% 0.
  FixedVector w;                                                                 0.0%
```

Figure 4.1: VTune profiling of multifit_cpu. 37% of the total time is spent performing $A^T A$.



```
    // swap AtA to avoid copy
    AtA.col(nPassive).swap(AtA.col(w_max_idx));                                 1.7%
    AtA.row(nPassive).swap(AtA.row(w_max_idx));                                 1.8%
    // swap Atb to match with AtA
    Eigen::numext::swap(Atb.coeffRef(nPassive), Atb.coeffRef(w_max_idx));       0.1%
    Eigen::numext::swap(x.coeffRef(nPassive), x.coeffRef(w_max_idx));           0.1%
    // swap the permutation matrix to reorder the solution in the end
    Eigen::numext::swap(permutation.indices()[nPassive],                        0.3%
                        permutation.indices()[w_max_idx]);

    ++nPassive;

#ifdef DEBUG_FNNLS_CPU
    cout << "max index " << w_max_idx << endl;
    std::cout << "n_active " << nActive << std::endl;
#endif

// inner loop
#pragma unroll VECTOR_SIZE
    while (nPassive > 0) {
      s.head(nPassive) =                                                        34.5%
          AtA.topLeftCorner(nPassive, nPassive).llt().solve(Atb.head(nPassive));

      if (s.head(nPassive).minCoeff() > 0.) {                                   1.0%
        x.head(nPassive) = s.head(nPassive);                                    1.3%
        break;                                                                  0.0%
      }

#ifdef DEBUG_FNNLS_CPU
      cout << "s" << endl << s.head(nPassive) << endl;
#endif

      auto alpha = std::numeric_limits<double>::max();
      Index alpha_idx = 0;

#pragma unroll VECTOR_SIZE
      for (auto i = 0; i < nPassive; ++i) {
        if (s[i] <= 0.) {                                                       0.0%
          auto const ratio = x[i] / (x[i] - s[i]);                              0.0%
          if (ratio < alpha) {
            alpha = ratio;
```

Figure 4.2: Second bottleneck found using VTune. 34% of the time is spent calculating the Cholesky decomposition.

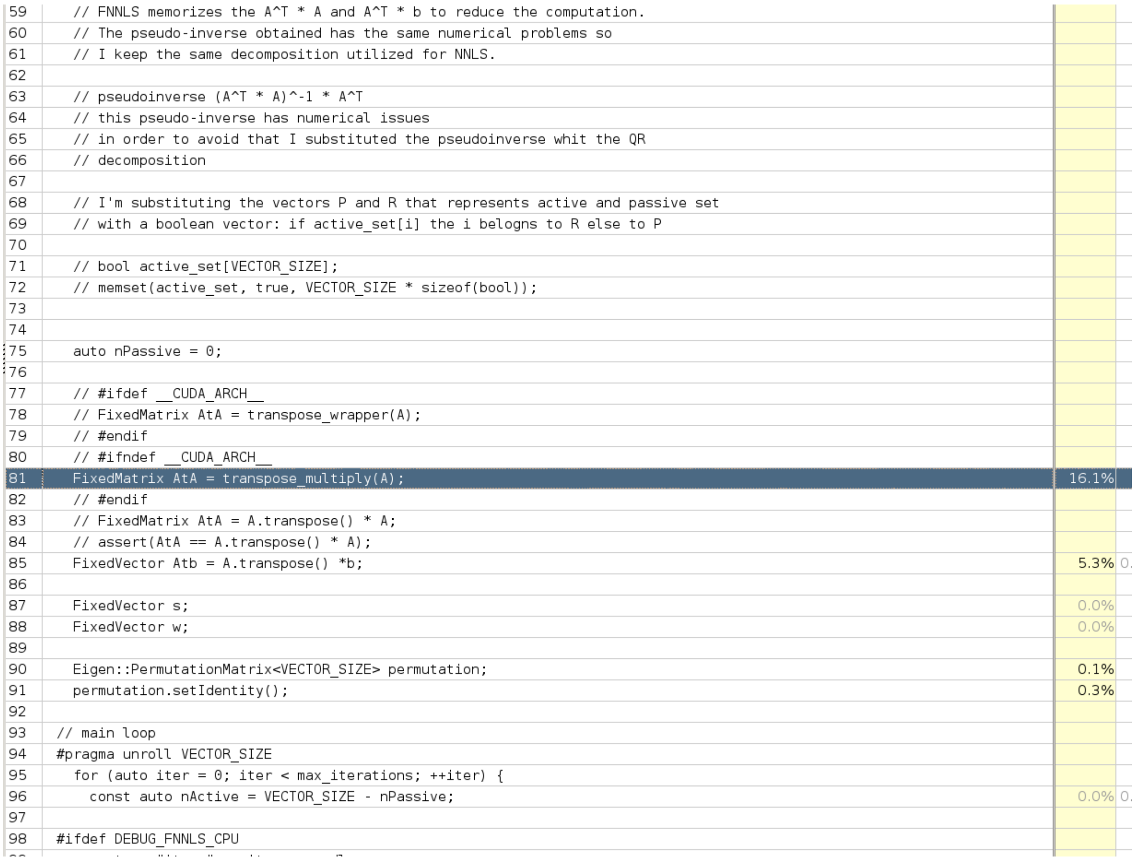
4.1.1 Matrix multiplication

The first hotspot is the computation of A^TA that requires 37% of the total execution time. To solve this hotspot it is possible by exploiting the observations:

- The matrix A^TA is symmetric so it is enough calculate the one triangular portion and the copy the results back.
- The matrix is 10×10, it fits in L1 cache, thus cache efficiency is more important than algorithmic complexity.

Exploiting the hypothesis stated above the implementation illustraded in pseudo-code 3, specific for this case, has been provided.

This code is very simple but, in cache simple things are needed. The goal is to avoid useless computations, cache faults, and bubbles generated by branches. The main idea behind it is given that A^TA is symmetric compute only half of it and copy the values to the other part. Moreover, knowing that the multiplication is not between two random matrices but, between a matrix and tis transpose, if the storage order is column major perform a column×column dot product or a row×row one in the other case. Since the storage order is column major there is another minor optimization that can be performed to gain the last drop of performance. This optimization is about the loop order. A careful choice minimizes jumps, for example in this particular case iterate over column generates no jumps neither cache misses, while iterating over rows generate a jump. A loop order of $row \rightarrow column \rightarrow dot\ product$ means that for each row there is one and only one jump. Instead, a loop order like $column \rightarrow row \rightarrow dot\ product$ generates $|columns| \times |rows|$



```
  // FNNLS memorizes the A^T * A and A^T * b to reduce the computation.
  // The pseudo-inverse obtained has the same numerical problems so
  // I keep the same decomposition utilized for NNLS.

  // pseudoinverse (A^T * A)^-1 * A^T
  // this pseudo-inverse has numerical issues
  // in order to avoid that I substituted the pseudoinverse whit the QR
  // decomposition

  // I'm substituting the vectors P and R that represents active and passive set
  // with a boolean vector: if active_set[i] the i belogns to R else to P

  // bool active_set[VECTOR_SIZE];
  // memset(active_set, true, VECTOR_SIZE * sizeof(bool));


  auto nPassive = 0;

  // #ifdef __CUDA_ARCH__
  // FixedMatrix AtA = transpose_wrapper(A);
  // #endif
  // #ifndef __CUDA_ARCH__
  FixedMatrix AtA = transpose_multiply(A);                    16.1%
  // #endif
  // FixedMatrix AtA = A.transpose() * A;
  // assert(AtA == A.transpose() * A);
  FixedVector Atb = A.transpose() *b;                          5.3% 0.

  FixedVector s;                                               0.0%
  FixedVector w;                                               0.0%

  Eigen::PermutationMatrix<VECTOR_SIZE> permutation;           0.1%
  permutation.setIdentity();                                   0.3%

// main loop
#pragma unroll VECTOR_SIZE
  for (auto iter = 0; iter < max_iterations; ++iter) {
    const auto nActive = VECTOR_SIZE - nPassive;               0.0% 0.

#ifdef DEBUG_FNNLS_CPU
```

Figure 4.3: Cache efficient 10×10 matrix multiplication. The time needed to perform it is only 16.1% respect to the 37.6% spent by eigen implementation.

jumps, which is bad for performance.
This optimized version of the matrix multiplication, as showed in 4.3 takes only 16.1% of the total time, while the reference one takes 37.6% of it, giving a speedup of 2.33.
On the GPU side instead, it is possible to optimize this further by parallelizing this product with a kernel invocation.

4.1.2 Updating the Cholesky

The second bottleneck can be solved using the closed formula present in [6] to update the Cholesky without recomputing it in case of adding/removing one column and one row.
Given that nnls updates a components of the solution vector at the time, to reduce the number of iterations performed inside the regression it is possible to directly exploit this information to update the Σ matrix. That way the whole regression, that is intrinsically iterative might be executed in fewer iterations.

Algorithm 3 Cache efficient matrix transposition and multiplication. A column major storage order is assumed otherwise the index needs to be reversed.

Input:

A real valued matrix of dimension $m \times n$

Output:

$\boldsymbol{A^T A}$ Real valued matrix of size $m \times m$

```
function TRANSPOSE_MULTIPLY($A$)
    $m \times m$ matrix $B$
    for $i \leftarrow 0; i < m; ++i$ do
        for $j \leftarrow i; j < m; ++j$ do        ▷ $A^T A$ is symmetric, compute half of it
            $B_{ji} \leftarrow 0$
            for $k \leftarrow 0; k < m; ++j$ do
                $B_{ji} += A_{ik} * A_{jk}$
            $B_{ij} \leftarrow B_{ji}$        ▷ Copy the result on the other half
    return $B$
```

5. Conclusions

The results of all the tests performed are summarized in the table 5.1 below. There are two important things to notice: first, with improvements applied to the CPU version a speedup of additional $\mathbf{38\%}$ was achieved. In case of CPU, increasing the number of channels does not affect the speedup, because all the versions are single threaded, therefore an approximately uniform improvement is observed. Second of all, by porting to the GPU a speedup of a factor of $\mathbf{2.67}$ was achieved for the case of $\boldsymbol{65K}$ channels (with trivial parallelization across all the channels). Important to note that for the GPU, a different trend is observed (increase up to $\approx 8K$ and then a slow down, and an increase again), as expected, due to the parallel nature of the architecture.

Table 5.1: Speedup achieved after applying all the optimizations.

channels	1024	2048	4096	8192	16384	32768	65536
legacy_multifit_cpu	1.00	1.00	1.00	1.00	1.00	1.00	1.00
legacy_multifit_gpu	0.62	1.22	1.77	2.09	1.89	2.10	2.23
multifit_cpu	1.25	1.38	1.35	1.35	1.33	1.34	1.34
multifit_gpu	0.96	1.57	2.09	2.56	2.31	2.54	2.67
multifit_gpu_swap	0.42	0.73	1.10	1.15	1.11	1.21	1.23
multifit_cpu_swap	0.45	0.48	0.51	0.50	0.50	0.50	0.50

There are few things to further develop, some are architecture dependent and some are algorithmic dependent:

- **GPU**: Dynamic parallelism to better exploit the available resources.
- **GPU**: Parallelise matrix operations further, however that needs to be properly measured as matrices being used are quite small.
- **Algorithmic**: Update the Cholesky instead of recomputing it.

Bibliography

[1] P Adzic, R Alemany-Fernandez, Carlos Almeida, N Almeida, Georgios Anagnostou, M.G. Anfreville, Ivan Anicin, Zeljko Antunovic, Etiennette Auffray, S Baccaro, S Baffioni, D Barney, L.M. Barone, Pierre Barrillon, Alessandro Bartoloni, S Beauceron, F Beaudette, K.W. Bell, R Benetta, and The CMS Electromagnetic Calorimeter Group. Reconstruction of the signal amplitude of the cms electromagnetic calorimeter. 46:23–35, 07 2006. 6

[2] R. Bro and S. d. Jong. A fast non-negativity- constrained least squares algorithm. *Journal of Chemometrics*, 11:393–401, 1997. 6

[3] Donghui Chen and Robert J. Plemmons. Nonnegativity constraints in numerical analysis. In *in A. Bultheel and R. Cools (Eds.), Symposium on the Birth of Numerical Analysis, World Scientific*. Press, 2009. 6

[4] Charles L Lawson, 1938 Hanson, Richard J., Society for Industrial, and Applied Mathematics. *Solving least squares problems*. Philadelphia : SIAM, [rev. ed.] edition, 1995. ”This SIAM edition is an unabridged, revised republication of the work first published by Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1974”–T.p. verso. 6, 7

[5] Do Q Lee. Numerically efficient methods for solving least squares problems. 8

[6] Wikipedia. Cholesky decomposition — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Cholesky%20decomposition&oldid=856784026, 2018. [Online; accessed 28-August-2018]. 17

[7] Wikipedia. Matrix decomposition — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Matrix%20decomposition&oldid=856974701, 2018. [Online; accessed 30-August-2018]. 8