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

Project Specification

project specification

Abstract

THIS IS MY ABSTRACT

Contents

List of Figures

List of Tables

Listings

1. From physics to... Physics

Modern high energy physics (HEP) requires experiments. To realize them several large scale objects are needed: from accelerators, detectors, data centers to the thousands of people involved to design and run the infrastructure.
Experiments are needed to trigger and observe physical processes. There are two kind of processes: interesting (**X**) and not interesting (**Y**) ones. The not interesting ones are processes that are already studied and measured, while the others are the objective of the research. Unfortunately, interesting processes are quite rare, thus a huge amount of trials before observing one of them is needed.
This creates a lot of difficult and interesting IT challenges because everything from data acquisition to data analysis is need to scale to match volume and time requirements.

Data generation The process starts with **particle collisions** this lead to particle interactions that create some **intermediate product**. This intermediate product can not be directly observed, bit it decays in **particles** that go through the detectors. The intermediate product is divided in **X** and **Y**, X is what physicists are trying to study while Y is something that produces similar particles as Y but is not what they are looking for.

Data aquisition Detectors are split in two levels: **physics detector level** and **electronics level**. The physics level takes in input sensor readouts and produces an analog signal. This analog signal is the digitalized by the electronics level. This level also decides if the event is interesting or not, based on this the event is discarded or recorded. The event that survive this ends up dumped into **disks** and send to the **High level trigger**.
It is important to notice that these decision are taken in real time, to meet the throughput and timing requirements everything is built with **asic**s and **fpga**s boards.

Data processing Data processing is performed both online and offline. The first step, called **local reconstruction**, in the offline data processing is to transform charges in physical quantities such as **energy**, **time**, and **position**. This operation is done channel by channel independently form each other. Then, information coming from different channel 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. The particle object forms the dataset used by theoretical physicists to do physics.

High level trigger (HLT) The **HLT** belongs both to data acquisition and data processing, since it would not be correct to include it in one of this categories, it deserves a separate discussion.
The HLT main purpose is to select interesting events before writing them to disk. In order to accomplish this, it runs the same code used for the offline processing, but **online** and with less features because of strict time constraints.
The main difference between **L1** and HLT is that L1 is implemented in hardware with no host while HLT is software based, it runs on hosts with accelerators used to speed-up the processing.

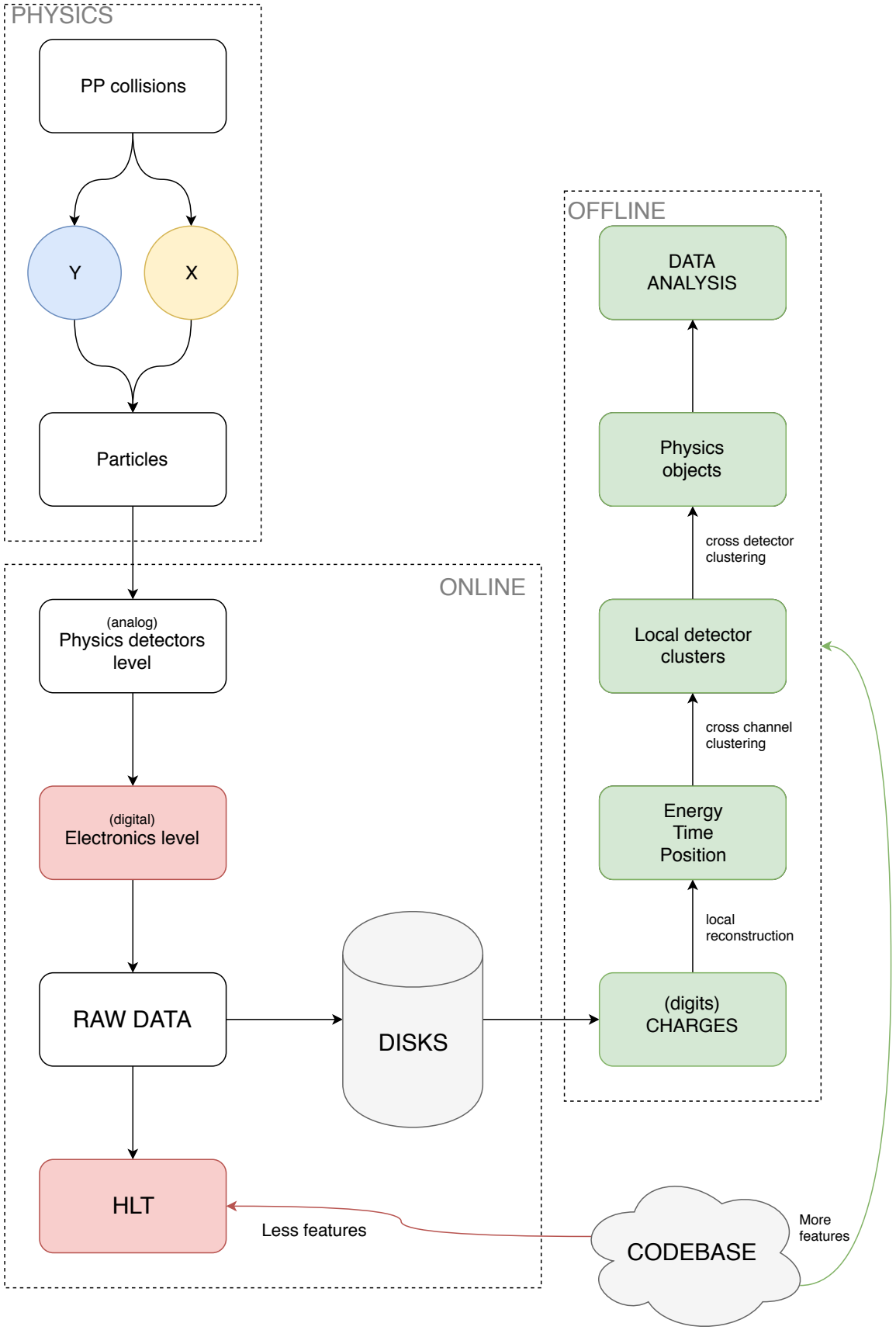
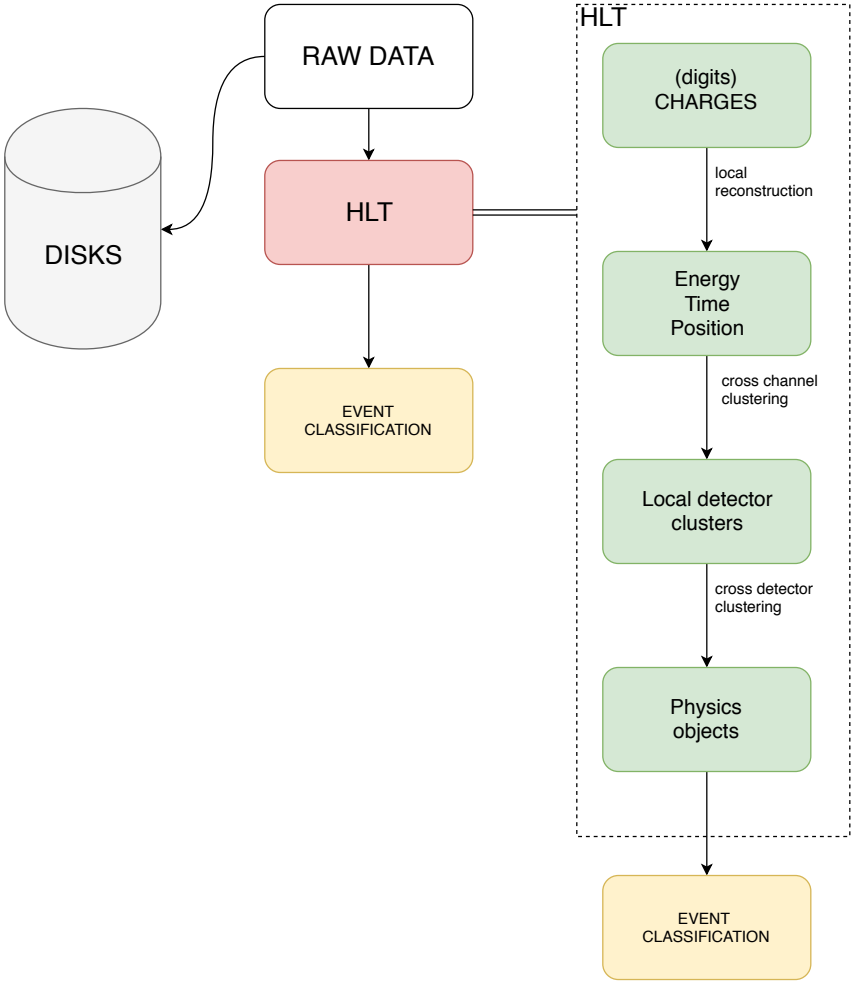


Figure 1.1: Cern data flow from collisions to analysis

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

Figure 2.1: HLT processing pipeline



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 of more than 10%. This is my focus in this project try to reduce it as much as possible.

Figure 2.2: Data processing time share

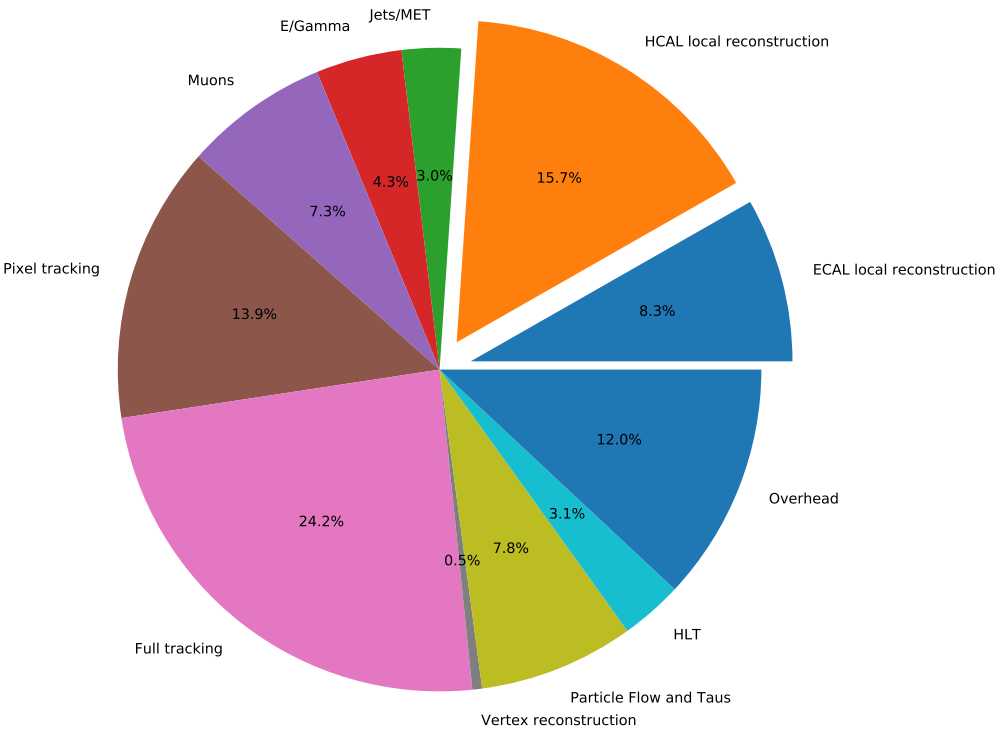


Table 2.1: Time spent into the various HLT reconstruction steps

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}.

$$\begin{aligned} &\min(\chi^2) = \arg min_x(\|Px - b\|^2) \\ &\forall i : x_i \geq 0 \end{aligned}$$

where:

$$\begin{aligned} &x = \text{energy vector} \\ &P : ENERGY \rightarrow CHARGE = \text{feature matrix} \\ &b = \text{charge vector} \\ &n = 10 \text{ (in this particular case)} \end{aligned} \tag{2.1}$$

Which is a standard minχ^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 exists, 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
ϵ the maximum accepted error
K the maximum number of iterations
Output:
x the solution vector

1: **function** NNLS(A, b, ϵ, 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 $i \in P : x_i \leftarrow 0$
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.
This improvements reduces the computational making the it to run faster.

Algorithm 2 FNNLS

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

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 results 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 The decompositions tested are **Cholesky** and **HouseHolder** with and without pivoting. These decompositions has 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 I noticed that a lot of useless operations containing 0 were present. Not only this slows down this is overhead but also increases the algorithmic error. Changing the definition to the one provided into the pseudo-code allowed also plain inverse and Cholesky 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 version 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 it to be studied.

2.5 Optimizations

There are two kind 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.

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

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 respect to the unroll. Also the compiler unroll vectorized loops combining the best of both approaches.
The unroll value has been determined empirically 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 GPU version achieves a speedup of 2.6. From 3.1 can be noted that the numerical optimizations allows to achieve a factor

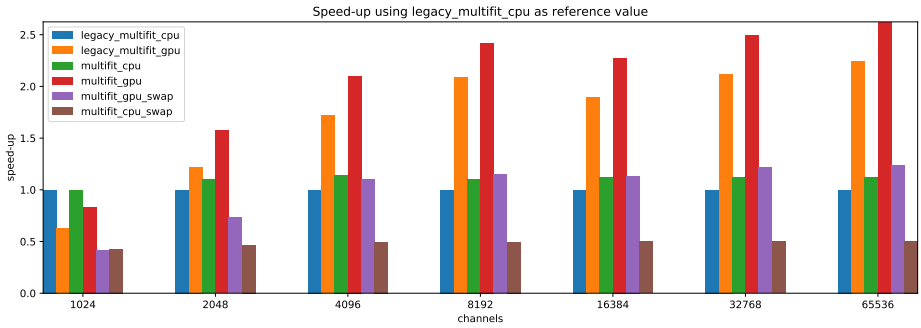


Figure 3.1: Speedup achieved with 10 iterations, log channel scale, higher is better

of 2 in speedup. The GPU performs twice as fast with respect to the CPU. Other optimizations like loop unrolling and branch reduction give a performance gain about 20% in CPU and 30% on GPU. From the plot present in 3.2 can be noted that, on GPU, the time increases faster at the beginning and around 15000 channels there is a change of slope and the increase becomes sub-linear. On the CPU, the increase is linear with respect to the channels. The plot in 3.3 makes more evident the difference between the different version of the algorithm. The GPU outperforms

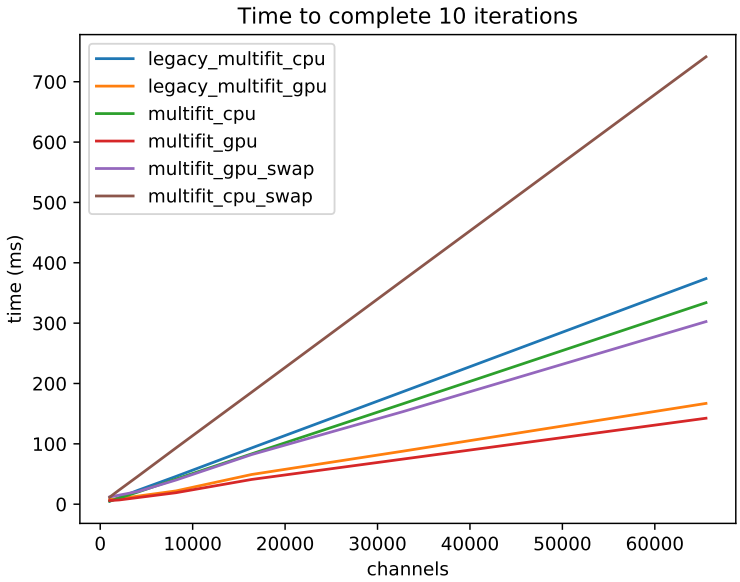


Figure 3.2: Time needed to complete 10 iterations, linear channel scale, lower is better

the CPU and it will be interesting to study what will happen if the number of channels increases even more.

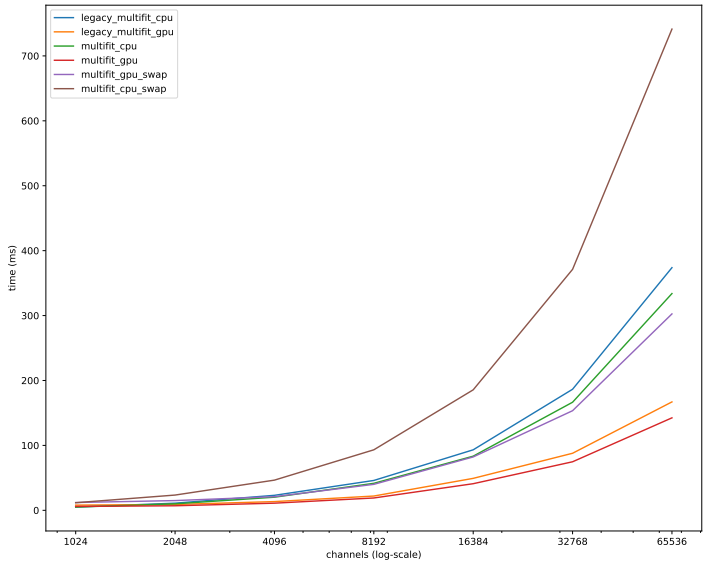


Figure 3.3: Time needed to complete 10 iterations, log channel scale, lower is better

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.

```
48  #endif
49  void inplace_fnnls(const FixedMatrix& A,
50                                      const FixedVector& b,
51                                      FixedVector& x,
52                                      const double eps,
53                                      const unsigned int max_iterations) {          0.0% 0.
54    // Fast NNLS (fnnls) algorithm as per
55    // http://users.wfu.edu/plemmons/papers/Chennnonneg.pdf
56    // page 8
57
58    // FNNLS memorizes the A^T * A and A^T * b to reduce the computation.
59    // The pseudo-inverse obtained has the same numerical problems so
60    // I keep the same decomposition utilized for NNLS.
61
62    // pseudoinverse (A^T * A)^-1 * A^T
63    // this pseudo-inverse has numerical issues
64    // in order to avoid that I substituted the pseudoinverse whit the QR
65    // decomposition
66
67    // I'm substituting the vectors P and R that represents active and passive set
68    // with a boolean vector: if active_set[i] the i belogns to R else to P
69
70    // bool active_set[VECTOR_SIZE];
71    // memset(active_set, true, VECTOR_SIZE * sizeof(bool));
72
73
74    auto nPassive = 0;
75
76    // #ifdef __CUDA_ARCH__
77    // FixedMatrix AtA = transpose_wrapper(A);
78    // #endif
79    // #ifndef __CUDA_ARCH__
80    // FixedMatrix AtA = transpose_multiply(A);
81    // #endif
82    FixedMatrix AtA = A.transpose() * A;                                            37.6% 0
83    // assert(AtA == A.transpose() * A);
84    FixedVector Atb = A.transpose() *;                                              3.7% 0.
85
86    FixedVector s;                                                                  0.1% 0.
87    FixedVector w;                                                                  0.0%
```

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

```
123     // swap AtA to avoid copy
124     AtA.col(nPassive).swap(AtA.col(w_max_idx));                                   1.7%
125     AtA.row(nPassive).swap(AtA.row(w_max_idx));                                   1.8%
126     // swap Atb to match with AtA
127     Eigen::numext::swap(Atb.coeffRef(nPassive), Atb.coeffRef(w_max_idx));         0.1%
128     Eigen::numext::swap(x.coeffRef(nPassive), x.coeffRef(w_max_idx));             0.1%
129     // swap the permutation matrix to reorder the solution in the end
130     Eigen::numext::swap(permutation.indices()[nPassive],                          0.3%
131                         permutation.indices()[w_max_idx]);
132
133     ++nPassive;
134
135 #ifdef DEBUG_FNNLS_CPU
136     cout << "max index " << w_max_idx << endl;
137     std::cout << "n_active " << nActive << std::endl;
138 #endif
139
140 // inner loop
141 #pragma unroll VECTOR_SIZE
142     while (nPassive > 0) {
143       s.head(nPassive) =                                                          34.5%
144           AtA.topLeftCorner(nPassive, nPassive).llt().solve(Atb.head(nPassive));
145
146       if (s.head(nPassive).minCoeff() > 0.) {                                     1.0%
147         x.head(nPassive) = s.head(nPassive);                                      1.3%
148         break;                                                                    0.0%
149       }
150
151 #ifdef DEBUG_FNNLS_CPU
152       cout << "s" << endl << s.head(nPassive) << endl;
153 #endif
154
155       auto alpha = std::numeric_limits<double>::max();
156       Index alpha_idx = 0;
157
158 #pragma unroll VECTOR_SIZE
159       for (auto i = 0; i < nPassive; ++i) {
160         if (s[i] <= 0.) {                                                         0.0%
161           auto const ratio = x[i] / (x[i] - s[i]);                                0.0%
162           if (ratio < alpha) {
163             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 his transpose, if the storage order is column major perform a column$\times$column dot product or a row$\times$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 $|columns| \times |rows|$

```
59    // FNNLS memorizes the A^T * A and A^T * b to reduce the computation.
60    // The pseudo-inverse obtained has the same numerical problems so
61    // I keep the same decomposition utilized for NNLS.
62
63    // pseudoinverse (A^T * A)^-1 * A^T
64    // this pseudo-inverse has numerical issues
65    // in order to avoid that I substituted the pseudoinverse whit the QR
66    // decomposition
67
68    // I'm substituting the vectors P and R that represents active and passive set
69    // with a boolean vector: if active_set[i] the i belogns to R else to P
70
71    // bool active_set[VECTOR_SIZE];
72    // memset(active_set, true, VECTOR_SIZE * sizeof(bool));
73
74
75    auto nPassive = 0;
76
77    // #ifdef __CUDA_ARCH__
78    // FixedMatrix AtA = transpose_wrapper(A);
79    // #endif
80    // #ifndef __CUDA_ARCH__
81    FixedMatrix AtA = transpose_multiply(A);                    16.1%
82    // #endif
83    // FixedMatrix AtA = A.transpose() * A;
84    // assert(AtA == A.transpose() * A);
85    FixedVector Atb = A.transpose() *b;                         5.3%  0.
86
87    FixedVector s;                                              0.0%
88    FixedVector w;                                              0.0%
89
90    Eigen::PermutationMatrix<VECTOR_SIZE> permutation;          0.1%
91    permutation.setIdentity();                                  0.3%
92
93  // main loop
94  #pragma unroll VECTOR_SIZE
95    for (auto iter = 0; iter < max_iterations; ++iter) {
96      const auto nActive = VECTOR_SIZE - nPassive;              0.0%  0.
97
98  #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, is it 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 $m \times m$

```
1: function TRANSPOSE_MULTIPLY(A)
2:     m × m matrix B
3:     for i ← 0; i < m; ++i do
4:         for j ← i; j < m; ++j do                ▷ A^T A is symmetric, compute half of it
5:             B_ji ← 0
6:             for k ← 0; k < m; ++j do
7:                 B_ji += A_ik * A_jk
8:             B_ij ← B_ji                          ▷ Copy the result on the other half
       return B
```

5. Conclusions

Write your conclusions here.

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. 5

[2] R. Bro and S. d. Jong. A fast non-negativity- constrained least squares algorithm. *Journal of Chemometrics*, 11:393–401, 1997. 5

[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. 5

[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. 5, 6

[5] Do Q Lee. Numerically efficient methods for solving least squares problems. 7

[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]. 14

A. My First Appendix

In this file (appendices/main.tex) you can add appendix chapters, just as you did in the thesis.tex file for the 'normal' chapters. You can also choose to include everything in this single file, whatever you prefer.