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

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 big data analysis. To make it possible several large scale objects are needed: from accelerators, detectors, data centers to the thousands of people involved to design and run the infrastructure. Here at CERN everything starts with physics by colliding particles and everything ends with the physics performed by analyzing the data acquired. But, to produce the necessary data, in the right amount, a long and complicated process, showed in figure 1.1, is involved. It is worth to briefly describe it to understand the reason of our work and where is it place inside it.

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.

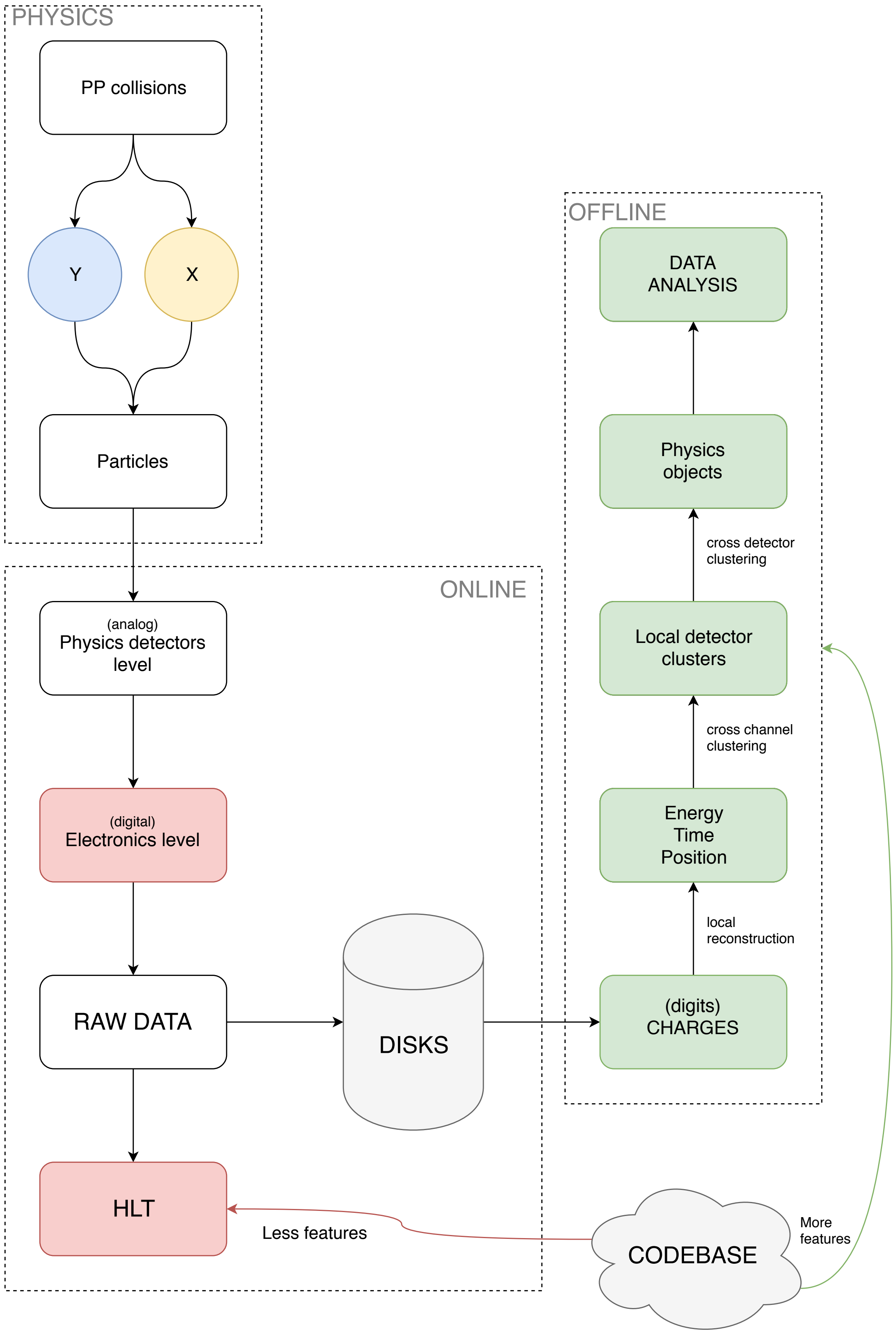
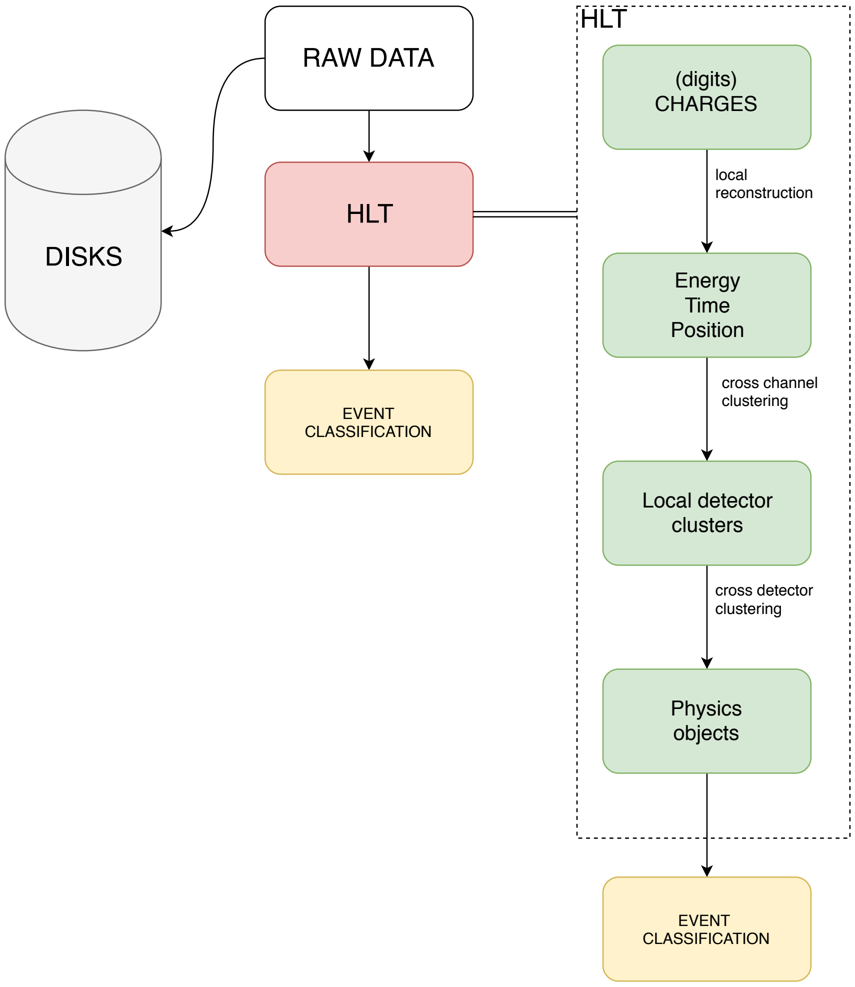


Figure 1.1: Cern data flow from collisions to analysis

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

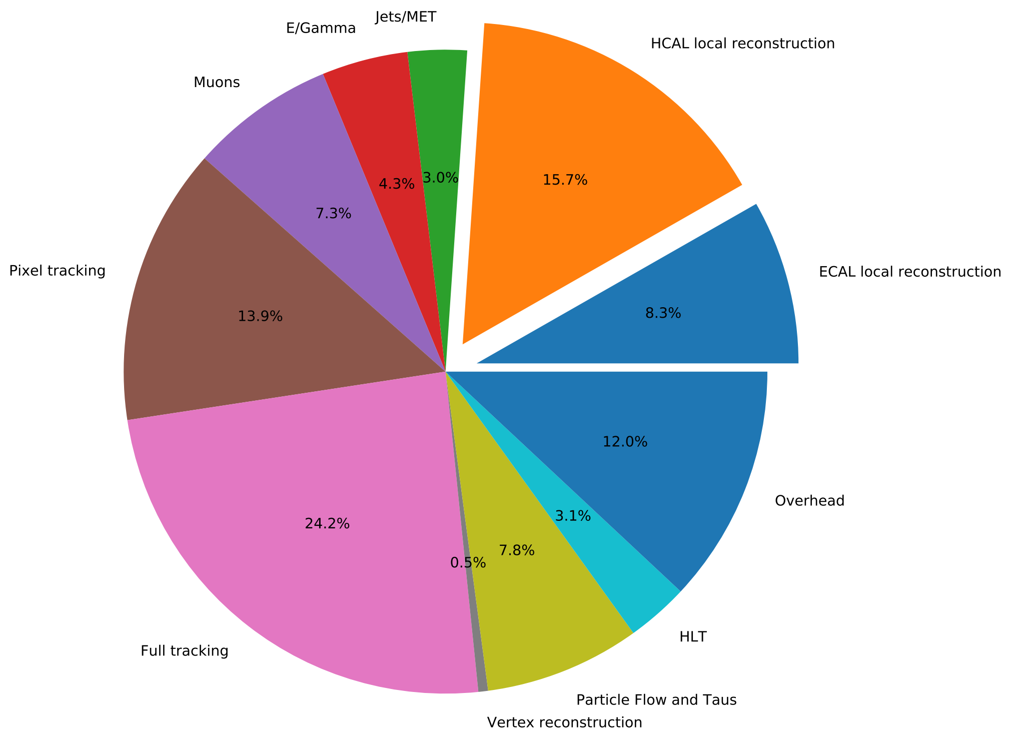


Table 2.1: Time spent into the various HLT reconstruction steps

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

As the name says the goal of this step is:

Claim: For each channel $\{$given n charge readouts $\rightarrow$ reconstruct the energy$\}$.
Where in this case n is fixed to 10.
Translated into mathematical terms this means:

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

Which is a 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.

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

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 is not satisfied.
- **Active set (R)**: the constraint is ”active”, meaning that 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 $i \in P : x_i \leftarrow 0$
16:             $s \leftarrow ((A^P)^T A^P)^{-1} A^P b^P$      ▷ recompute s for the next iteration
17:         $\forall i \in P : x_i \leftarrow s_i$
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 performs best.

2.4 GPU porting

Until now there are two version ported on GPU, the one taken from *cms-sw* codebase and the one that I implemented.
To exploit the parallelism provided by these devices a channel level parallelization is performed. More precisely all the readout from the channel are buffered and then there is a kernel invocation on all of them.

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. Simulating the algorithm it is possible to notice that the P, R partitioning can be obtained in-place by permuting the matrix A.

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

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.

2.5.2 Algorithmic

The optimization performed here is exploiting some pragmas to vectorise fixed iteration loops and to unroll the others.
From the profiling performed I noticed 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. I measured branch misprediction and cache faults for each value and used the one that minimize them.

3. Results

Test01: GPU vs CPU 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.

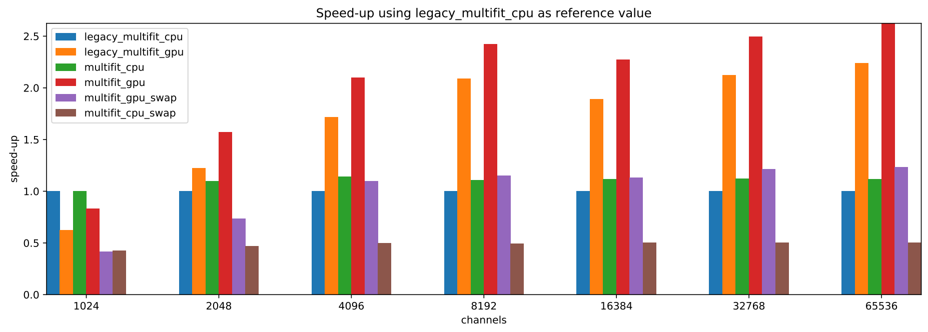


Figure 3.1: Speedup achieved with 10 iterations, log channel scale, higher is better

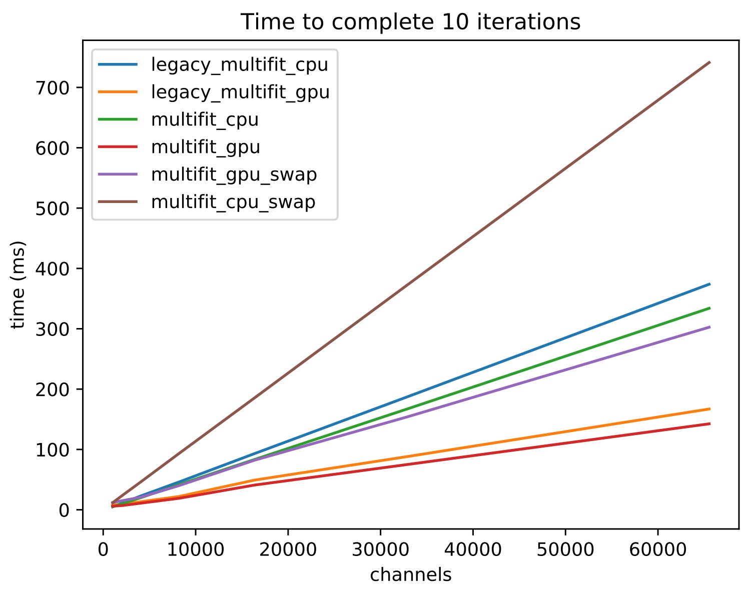


Figure 3.2: Time needed to complete 10 iterations, linear channel scale, lower is better

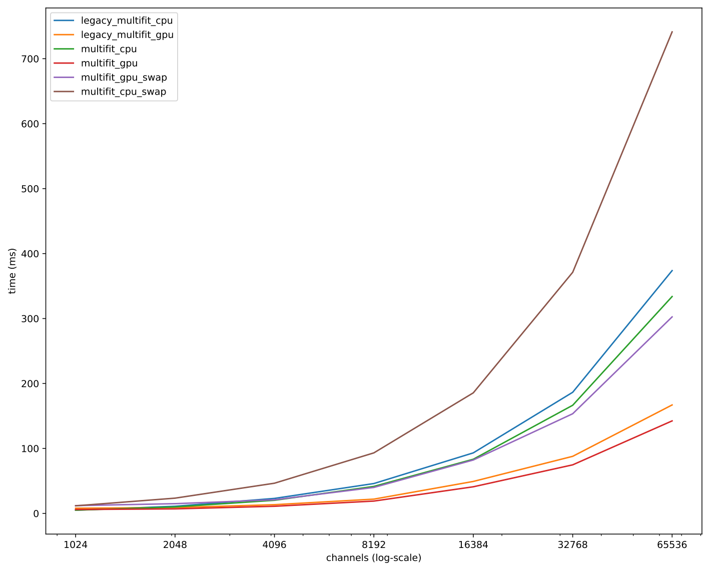


Figure 3.3: Time needed to complete 10 iterations, log channel scale, lower is better

4. 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

205 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.