UNIVERSITÀ DI PISA

Corso di Laurea Magistrale in Scienze Fisiche
Curriculum 'Fisica delle Interazioni Fondamentali'

Tesi di Laurea

Development of a parallel trigger framework for rare decay searches

Candidato:
Felice Pantaleo

Relatore:
Prof. Marco Sozzi

Anno Accademico 2012-2013

To my sister Bea, the inventor of Pepepuz®.

To Isabel, "On ne voit bien qu'avec le cœur. L'essentiel est invisible pour les yeux".

To Marcello, my Pole Star.

Contents

Introduction

The NA62 Experiment represents the current program of testing of the Standard Model through the study of the K meson at CERN in Geneva. It offers a complementary approach with respect to the experiments on the Energy frontiers at the Large Hadron Collider.

The aim of the NA62 experiment is to measure the branching ratio of the process $K^+ \to \pi^+ \nu\bar{\nu}$ with a precision of 10%. Since the value predicted by the Standard Model is very precise, the measurement of this quantity represents an excellent way to investigate the existence of New Physics.

The simplicity of kaon decays (few decay channels, low final-state multiplicities) enable the possibility to reach an excellent sensitivity in the searches of lepton flavor violating decays. The experimental characteristics of decays like $K^+ \to \pi^- \mu^+ \mu^+$ are very clear and allow an efficient background rejection.

However, the measurement of this kind of events requires the production of a remarkable number of kaon decays. The bandwidth of tape recording system currently available does not allow the storage of all the produced events. A multi-stage selection of the potentially interesting events is required (trigger).

At NA62, a first selection is done in real-time (response time < 1 ms) by the *level 0 trigger*. The level 0 trigger is based on programmable logic (FPGA) that does not allow the same flexibility of the processors used for software programmable computers. The performance of parallel architectures like multi-cores CPUs and GPUs (Graphics Processing Units),

located on computers graphic card, is very promising with a view to their employment for more complex *pattern recognition* like Čerenkov light rings reconstruction, inside the NA62 RICH detector.

In the first chapter of this thesis I present the NA62 experiment and its experimental apparatus in the contest of which this work was carried out. In particular, the theoretical framework of the lepton flavor violating process $K^+ \to \pi^-\mu^+\mu^+$ is discussed with a brief summary of the previous searches of this decay.

Heterogeneous parallel computing is described in Chapter 2, focusing on the concepts used to develop a software trigger framework and parallel algorithms running on GPUs.

Chapter 3 focuses on the feasibility study on the possibility to use GPUs in a high-bandwidth and low-latency environment like a real-time trigger. At NA62, this study requested the development of various parallel algorithms for the determination of performance along with the bottlenecks detection.

In Chapter 4, I describe the development of a high performance software framework that uses multi-threaded programming techniques and fast network drivers for the transmission of trigger primitives from the front end electronics to the GPU memory for the computation and the events selection.

Finally, the use of a multi-ring reconstruction algorithm, which runs inside the developed framework, for the selection of $K^+ \to \pi^-\mu^+\mu^+$ events is described in Chapter 5. In order to determine the selection efficiency of the algorithm, I studied the background rejection efficiency and the acceptance for signal events as a function of some selection parameters, hence determining the advantages of this innovative approach.

Chapter 1

NA62 experiment

1.1 Introduction

The *NA62* (North Area 62) experiment at the CERN *Super Proton Synchrotron* (SPS), is designed for the study of rare kaon decays. It represents the current kaon physics program at CERN and offers a complementary approach, with respect to the Large Hadron Collider high energy frontier, to probe new physics at short distances, corresponding to energy scales up to ≈ 100 TeV.

The goal of the NA62 experiment [1] is to measure the branching ratio (BR) for the decay $K^{\pm} \to \pi^{\pm}\nu\bar{\nu}$ with a precision of $\sim$10%.

The Standard Model (SM) prediction $\mathrm{BR}(K^{\pm} \to \pi^{\pm}\nu\bar{\nu}) = (7.8 \pm 0.8) \times 10^{-11}$ [2] is unusually precise. This is motivated by the fact that the hadronic matrix element for the decay can be obtained from the experimentally determined rate for K_{e3} decays. The value of $\mathrm{BR}(K^{\pm} \to \pi^{\pm}\nu\bar{\nu})$ is a sensitive probe for physics beyond the SM. The exceptional theoretical cleanliness of this decay channel makes it extremely attractive to test the Standard Model predictions and test the existence of New Physics. However a considerable amount of kaons is required for its measurement, since this branching ratio is so small.

Next-generation experiments to measure $K^{+} \to \pi^{+}\nu\bar{\nu}$ will be well suited to carry out a rich search program for very rare or forbidden K^{+} decays, because intense K^{+} sources

and robust background rejection (through precise tracking, particle identification and hermetic coverage) are the defining features of such experiments. With $\sim 10^{13}$ K^+ and $\sim 2.5 \times 10^{12}$ π^0 decays in its fiducial volume after two years' worth of data taking, NA62 will have single-event sensitivities of $\sim 10^{-12}$ for a number of decays that violate lepton flavor conservation.

1.2 Theoretical framework

1.2.1 Cabibbo-Kobayashi-Maskawa matrix

The CKM framework provides an extension to the Cabibbo 2×2 matrix, which describes the couplings between u, c and d, s quarks, mediated by weak *Flavor-Changing Charged Currents* ($W^{\pm}$) [3]. The coupling is described by the introduction of states d' and s' obtained from the mass eigenstates d and s by means of a rotation by an angle θ_C, named Cabibbo angle:

$$\begin{pmatrix} d' \\ s' \end{pmatrix} = \begin{pmatrix} \cos\theta_C & \sin\theta_C \\ -\sin\theta_C & \cos\theta_C \end{pmatrix} \begin{pmatrix} d \\ s \end{pmatrix} \tag{1.1}$$

Experimental results have defined a value of $\theta_C \simeq 13^\circ$ [5].

The Cabibbo theory was extended to include the quark states belonging to the third generation as well. This generalization is provided by the Cabibbo-Kobayashi-Maskawa (CKM) matrix [4]:

$$\begin{pmatrix} d' \\ s' \\ b' \end{pmatrix} = \begin{pmatrix} V_{ud} & V_{us} & V_{ub} \\ V_{cd} & V_{cs} & V_{cb} \\ V_{td} & V_{ts} & V_{tb} \end{pmatrix} \begin{pmatrix} d \\ s \\ b \end{pmatrix} \tag{1.2}$$

Current experimental results yield the following values for the elements of the CKM

matrix [5]:

$$\begin{pmatrix} |V_{ud}| = 0.97425 \pm 0.00022 & |V_{us}| = 0.2252 \pm 0.0009 & |V_{ub}| = (4.15 \pm 0.49)\,10^{-3} \\ |V_{cd}| = 0.230 \pm 0.011 & |V_{cs}| = 1.006 \pm 0.023 & |V_{cb}| = (40.9 \pm 1.1)\,10^{-3} \\ |V_{td}| = (8.4 \pm 0.6)\,10^{-3} & |V_{ts}| = (42.9 \pm 2.6)\,10^{-3} & |V_{tb}| = 0.89 \pm 0.07 \end{pmatrix} \tag{1.3}$$

Transitions between quarks belonging to the same generation, i.e. $u \leftrightarrow d$, $c \leftrightarrow s$ and $t \leftrightarrow b$, are seen to be favored, whereas those between quarks from different families are suppressed.

The value for $|V_{us}|$ is determined from the measurement of $\mathrm{BR}(K^+ \to \mu^+\nu(\gamma))$ [6] and a combined result of $K_L^0 \to \pi e\nu$, $K_L^0 \to \pi\mu\nu$, $K^\pm \to \pi^0 e^\pm \nu$, $K^\pm \to \pi^0 \mu^\pm \nu$ and $K_S^0 \to \pi e\nu$ decays [5].

Unlike the Cabibbo matrix, the CKM matrix does not represent a pure rotation, as it also includes complex parameters. Using the Wolfenstein parametrization the CKM matrix can be written [7]:

$$V_{CKM} = \begin{pmatrix} 1 - \lambda^2/2 & \lambda & A\lambda^3(\rho - i\eta) \\ -\lambda & 1 - \lambda^2/2 & A\lambda^2 \\ A\lambda^3(1 - \rho - i\eta) & -A\lambda^2 & 1 \end{pmatrix} + \mathcal{O}(\lambda^4) \tag{1.4}$$

where

$$A, \lambda > 0 \tag{1.5}$$

$$\lambda = \sin\theta_{12} = \sin\theta_C \tag{1.6}$$

$$A\lambda^2 = \sin\theta_{23} \tag{1.7}$$

$$A\lambda^3(\rho - i\eta) = \sin\theta_{13} e^{-i\varphi} \tag{1.8}$$

θ_{ij} are three real Cabibbo-like angles, while $e^{-i\varphi}$ is a complex phase term that carries CP violation.

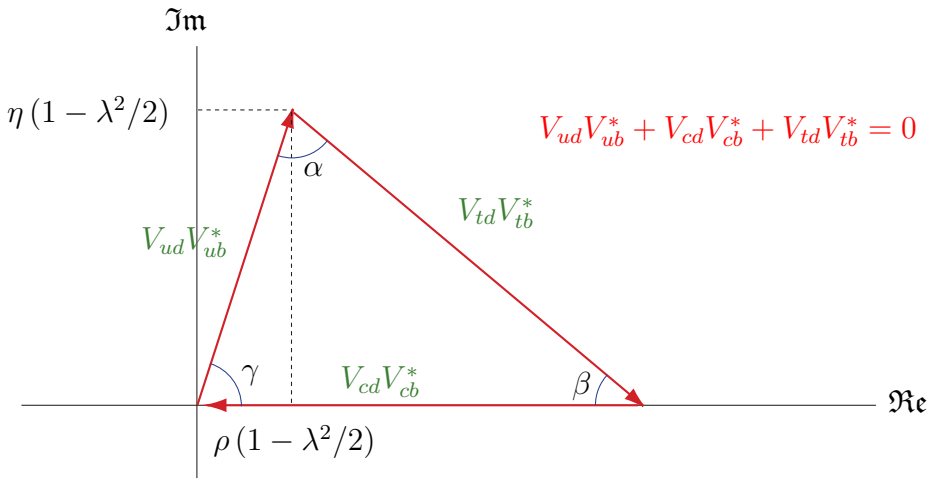


Figure 1.1: Unitarity triangle in the complex plane.

Unitarity of the CKM matrix implies:

$$\sum_{i=u,c,t} V_{ik}^* V_{ij} = \delta_{jk} \qquad j,k = d,s,b \tag{1.9}$$

$$\sum_{j=d,s,b} V_{jk}^* V_{ij} = \delta_{ik} \qquad i = u,c,t; \quad k = d,s,b \tag{1.10}$$

The six vanishing combinations can be represented as triangles in the complex plane (e.g. for $V_{ud}V_{ub}^* + V_{cd}V_{cb}^* + V_{td}V_{tb}^* = 0$ see Figure 1.1).

Flavor Changing Neutral Current (FCNC) processes, relate either up-type or down-type flavors but not both, and either charged lepton or neutrino flavors but not both. Within the Standard Model, these processes do not occur at tree level, and are therefore suppressed. For this reason the $K^+ \to \pi^+ \nu\bar{\nu}$ transition is strongly suppressed within the Standard Model and its contributions arise from the one-loop contributions shown in Figure 1.2.

Separating the contributions of the quarks u, c and t, appearing as internal lines, at the leading non-trivial order the amplitude of the $s \to d\nu\bar{\nu}$ process may be expressed as

$$A\,(s \to d\nu\bar{\nu}) = \sum_{q=u,c,t} V_{qs}^* V_{qd}\, A_q \qquad \text{with} \tag{1.11}$$

$$A_q \sim \left(\frac{m_q^2}{m_W^2}\right)^{\delta} \qquad (\delta > 0) \qquad q = u,c,t \tag{1.12}$$

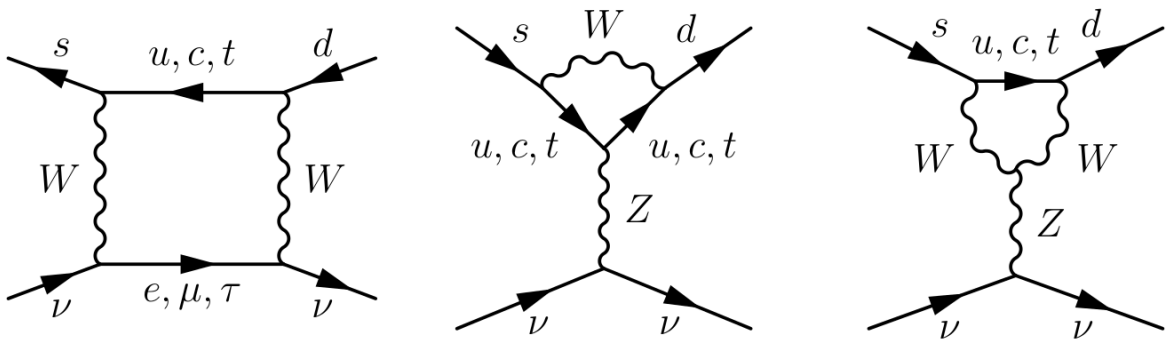


Figure 1.2: A $W-$box and two $Z-$penguin diagrams. These are the one-loop Feynman diagrams contributing to the $s \to d\nu\bar{\nu}$ process.

The top quark term dominates, due to its higher mass. As a consequence this process can be well described by short-distance dynamics, with the effective Hamiltonian

$$H_{\text{eff}} = \frac{\alpha G_F}{2\sqrt{2}\pi \sin^2\theta_W} \sum_{l=e,\mu,\tau} \left(V_{cs}^* V_{cd} X_l + V_{ts}^* V_{td} \Upsilon_t(x_t) \right) (\bar{s}d)\,(\bar{\nu}_l \nu_l) \tag{1.13}$$

where G_F, α and θ_W are the Fermi and fine-structure constants and the Weinberg angle, respectively. Being $x_t = m_t^2/M_W^2$, $\Upsilon_t(x_t)$ is a function representing the dominant top quark contribution, whose associated uncertainty is very small and mainly due to the experimental error on the top quark mass [8]:

$$\Upsilon_t = 1.469 \pm 0.017 \pm 0.002 \tag{1.14}$$

where the two uncertainties correspond to QCD next-to-leading order and two-loops EW corrections respectively. The $\{X_l\}_{l=e,\mu,\tau}$ functions encode instead the charm quark contributions and can be computed at the next-to-next-to-leading order with an error lower than 4% [9]. Next, u and $\bar{s}$ are the quark states embedded into the K^+ meson. Finally, the terms $(\bar{s}d)$ and $(\bar{\nu}_l\nu_l)$ represent $V-A$ neutral weak currents.

Since the coupling amplitude depends on the semi-leptonic operator $(\bar{s}d)\,(\bar{\nu}_l\nu_l)$, the hadronic part of the amplitude of the studied process can be determined from that of

the decay $K^+ \to \pi^0 e^+ \nu_e$ by means of isospin symmetry, leading to [10]

$$\frac{BR(K^+ \to \pi^+ \nu\bar{\nu})}{BR(K^+ \to \pi^0 e^+ \nu_e)} = \frac{r_K}{\lambda^2} \left\{ \left[\Im(V_{ts}^* V_{td}) \right]^2 \Upsilon_t^2 + \left[\lambda^4 \Re(V_{cs}^* V_{cd}) P_0 + \Re(V_{ts}^* V_{td}) \Upsilon_t \right]^2 \right\} \tag{1.15}$$

In this equation, the total charm-quark term is conveniently described by the parameter P_0

$$P_0 = \frac{1}{\lambda^4} \left(\frac{2}{3} X_e + \frac{1}{3} X_\tau \right) = 0.42 \pm 0.06 \tag{1.16}$$

where $X_\mu = X_e$ and X_τ are all known at NLO [10], and $r_K = 0.901$ provides the necessary isospin-breaking corrections to be applied in order to relate $BR(K^+ \to \pi^+ \nu\bar{\nu})$ to $BR(K^+ \to \pi^0 e^+ \nu_e)$.

The theoretical expectation is then

$$BR(K^+ \to \pi^+ \nu\bar{\nu}) = \left(7.81^{+0.80}_{-0.71} \pm 0.29\right) \cdot 10^{-11} \tag{1.17}$$

where the uncertainties were separated in order to highlight the smallness of the second contribution, which is the purely theoretical error. The first error is instead due to the input CKM parameters [2].

In the Wolfenstein parametrization at leading order $V_{ts} \simeq -V_{cb}$, and the experimental knowledge of $|V_{cb}|$, $|V_{cs}|$ and $|V_{cd}|$ is quite accurate. In this parametrization the only free parameter is V_{td}, to be experimentally determined with an uncertainty theoretically as low as 5 – 7% [2].

Our current knowledge of $|V_{td}|$ mainly derives from the analysis of the neutral strange B meson system. The most accurate measurement of the mass difference $\Delta m_s = m(B_s^0) - m(\bar{B}_s^0)$ is obtained averaging CDF [11] and LHCb [12] results, to give

$$\Delta m_s = (17.719 \pm 0.043)\ \text{ps}^{-1}. \tag{1.18}$$

from which [13]:

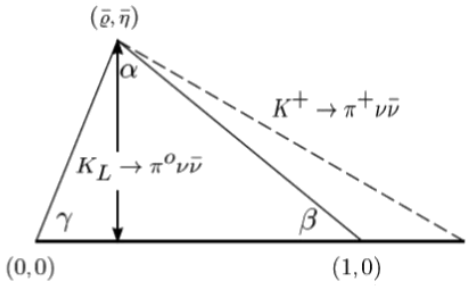


Figure 1.3: Unitarity triangle corresponding to the orthogonality condition 1.22.

$$|V_{td}| = (8.4 \pm 0.6) \cdot 10^{-3} \tag{1.19}$$

$$|V_{ts}| = (42.9 \pm 2.6) \cdot 10^{-3} \tag{1.20}$$

$$|V_{td}/V_{ts}| = 0.211 \pm 0.001 \pm 0.006 \tag{1.21}$$

$|V_{td}|$ can also be investigated independently by measuring the branching ratio of the $K^+ \to \pi^+\nu\bar{\nu}$ process.

The orthogonality condition written using the first two columns of the CKM matrix is:

$$V_{us}^* V_{ud} + V_{cs}^* V_{cd} + V_{ts}^* V_{td} = 0. \tag{1.22}$$

Defining $\lambda_t = V_{td}V_{ts}^*$, it is possible to observe that, for the corresponding unitarity triangle in the complex plane:

- The length of the vector $V_{us}^* V_{ud}$ is determined from the K_{e3} decay.
- The length of the vector $V_{ts}^* V_{td}$, namely $\Im(\lambda_t)\Re(\lambda_t)$ can be measured by $K^+ \to \pi^+\nu\bar{\nu}$ (eq.1.15).
- The height of the unitarity triangle, $\Im(\lambda_t)$, can be measured from the $K_L \to \pi^0\nu\bar{\nu}$ [14].

Therefore the branching ratio measurements of the $K^+ \to \pi^+\nu\bar{\nu}$ process will determine this unitarity triangle. Referring to figure 1.3, the $K^+ \to \pi^+\nu\bar{\nu}$ branching ratio defines the length of the dashed side, while the displacement of the right-down vertex is due to the charm-quark contribution.

1.3 Lepton flavor violation in kaon decays

Within the Standard Model, the strong, weak and electromagnetic interactions are respectively connected to $SU(3)_C$, $SU(2)_L$ and $U(1)_{EM}$ gauge groups. The symmetry to which an interaction is related explains its characteristics.

Neutrinos are fermions that don't have either strong or electromagnetic interactions, i.e. they are singlets of $SU(3)_C \times U(1)_{EM}$. Neutrinos that have weak interactions are referred as *active neutrinos*. Conversely *sterile neutrinos* don't have any SM gauge interactions, i.e. they are singlets of the full SM gauge group.

1.3.1 Massless neutrinos in the Standard Model

The matter content of the Standard Model (SM) consists of three families of chiral quarks and leptons. Each family has five different gauge representations:

$$L_L(1,2,-\frac{1}{2}),\ \ Q_L(3,2,\frac{1}{6}),\ \ E_R(1,1,1),\ \ U_R(3,1,\frac{2}{3}),\ \ D_R(3,1,-\frac{1}{3}). \tag{1.23}$$

The Standard Model has three active neutrinos accompanying the charged lepton mass eigenstates, e, μ and τ, thus there are weak charged current (CC) interactions between the neutrinos and their corresponding charged leptons given by:

$$-\mathcal{L}_{CC} = \frac{g}{\sqrt{2}} \sum_l \bar{\nu}_{Ll} \gamma^\mu l_L^- W_\mu^+ + \text{h.c.} \tag{1.24}$$

where ν, l and W^μ indicate respectively the neutrino, the charged lepton and the W-boson fields. Furthermore, the SM neutrinos have also neutral current (NC) interactions:

$$-\mathcal{L}_{NC} = \frac{g}{2\cos\theta_W} \sum_l \bar{\nu}_{Ll} \gamma^\mu \nu_{Ll} Z_\mu^0. \tag{1.25}$$

where Z_μ^0 indicates the Z-boson field. The Standard Model does not contain sterile neutrinos.

One can determine the decay width of the Z_0 boson into neutrinos by using equation 1.25. This width is proportional to the number of light left-handed neutrinos. The measurement

of the invisible Z width gives $N_\nu = 2.9840 \pm 0.0082$[1]. This value implies that any extension of the SM must contain exactly three light active neutrinos.

Furthermore the Standard Model, with its gauge symmetry presents an additional global symmetry:

$$G_{SM}^{global} = U(1)_B \times U(1)_{L_e} \times U(1)_{L_\mu} \times U(1)_{L_\tau} \tag{1.26}$$

where $U(1)_B$ is the baryon number symmetry and $U(1)_{L_{e,\mu,\tau}}$ are the three lepton flavor symmetries. The total lepton number is defined as:

$$L = L_e + L_\mu + L_\tau \tag{1.27}$$

In the SM, fermions masses arise from the Yukawa interactions which couple a right-handed fermion with its left-handed doublet and the Higgs field ϕ:

$$-\mathcal{L}_{\text{Yukawa}} = Y_{ij}^d \bar{Q}_{Li} \phi D_{Rj} + Y_{ij}^u \bar{Q}_{Li} \tilde{\phi} U_{Rj} + Y_{ij}^l \bar{L}_{Li} \phi E_{Rj} + \text{h.c.}, \tag{1.28}$$

where Y^d is the fermion coupling with the Higgs field.

After spontaneous symmetry breaking lead to charged fermion masses:

$$m_{ij}^f = Y_{ij}^f \frac{v}{\sqrt{2}}. \tag{1.29}$$

where v is the vacuum expectation value of the Higgs field. Since no right-handed neutrinos exist in the Standard Model, the Yukawa interactions (eq. 1.28) leave the neutrinos massless.

1.3.2 Massive neutrinos

More than ten years have passed since the experimental discovery of neutrino oscillations and hence non-zero neutrino masses and lepton mixing from the solar neutrino deficiency and atmospheric neutrino anomaly. The existence of lepton mixing means that lepton family number is not a valid symmetry.

[1] Combined fit from ALEPH, DELPHI, L3 and OPAL Experiments.

In order to introduce a neutrino mass one could add an arbitrary number m of sterile neutrinos, i.e. electroweak singlet neutrinos $\nu_{si}(1,1,0)$, to the particle content of the SM. Therefore, as described in [21], it is possible to build two kinds of mass terms, that arise from gauge invariant renormalizable operators:

$$-\mathcal{L}_{M_\nu} = M_{Dij}\bar{\nu}_{si}\nu_{Lj} + \frac{1}{2}M_{Nij}\bar{\nu}_{si}\nu^c_{sj} + \text{h.c.} \tag{1.30}$$

where ν^c indicates a charge conjugated field, $\nu^c = C\bar{\nu}^T$ and C is the charge conjugation matrix, M_D is a complex $m \times 3$ matrix and M_N is a symmetric $m \times m$ matrix.

The first term is generated after spontaneous electroweak symmetry breaking from Yukawa interactions:

$$Y^\nu_{ij}\bar{\nu}_{si}\tilde{\phi}^\dagger L_{Lj} \Rightarrow M_{Dij} = Y^\nu_{ij}\frac{v}{\sqrt{2}}. \tag{1.31}$$

It is a Dirac mass term and it conserves the total lepton number while breaking the lepton flavor number symmetries.

The second term in Eq. 1.30 is a Majorana mass term. It is a singlet of the SM gauge group and since it involves two neutrino fields, it changes lepton number by two units, $|\Delta L| = 2$.

Equation 1.30 could be rewritten as:

$$-\mathcal{L}_{M_\nu} = \frac{1}{2}\overline{\overrightarrow{\nu}^c}M_\nu\overrightarrow{\nu} + \text{h.c.}, \tag{1.32}$$

where

$$M_\nu = \begin{pmatrix} 0 & M_D^T \\ M_D & M_N \end{pmatrix} \tag{1.33}$$

and $\overrightarrow{\nu} = (\overrightarrow{\nu}_L, \overrightarrow{\overline{\nu}^c}_s)^T$ is a $(3+m)$-dimensional vector. The matrix M_ν is complex and symmetric and can be diagonalized by a unitary matrix V^ν of dimension $(3+m)$, so that,

in terms of the $(3+m)$ mass eigenstates $\overrightarrow{\nu}_{mass} = (V^{\nu})^{\dagger}\overrightarrow{\nu}$, eq. 1.32 can be rewritten as:

$$-\mathcal{L}_{M\nu} = \frac{1}{2}\sum_{k=1}^{3+m} m_k \bar{\nu}_{Mk}\nu_{Mk} \tag{1.34}$$

where $\nu_{Mk} = \nu_{mass,k} + \nu^c_{mass,k}$.

Since $\nu_M = \nu^c_M$, these neutrinos take the name of *Majorana neutrinos*. Majorana neutrinos correspond to non-degenerate eigenvalues that naturally form a set of 3 light masses for the three known neutrinos (mostly left-handed), and m very large masses (mostly right-handed neutrinos). This mechanism is called *see-saw mechanism*, and can provide a plausible explanation for why the known neutrinos are so light, making the SM an effective low energy theory. Although one knows from the invisible Z width that there are three light SU(2)-doublet neutrinos, one does not know the number m of electroweak-singlet neutrinos.

1.3.3 $K^+ \to \pi^-\mu^+\mu^+$ decay

Many extensions of the Standard Model introduce new interactions which produce violation of lepton flavor in specific processes. Supersymmetric unified theories with soft supersymmetry breaking terms generated at the Planck scale [22], or mechanism for dynamical electroweak symmetry breaking with strong coupling such as extended technicolor (ETC) [23], Little Higgs models [24], or models that introduce heavy neutrinos into the SM [25], predict lepton flavor violating processes.

Searches for *Lepton Flavor Violation* (LFV) in kaon decays have placed tight constrains on the parameter space for some of these models.

Among searches for the violation of the total lepton number are searches for $|\Delta L| = 2$ processes:

- neutrino-less double beta decay of nuclei [26]:

$$(A, Z) \to (A, Z+2) + e^- + e^-. \tag{1.35}$$

Mode	UL at 90% CL	Experiment	Ref.
$K^+ \to \pi^+\mu^+e^-$	1.3×10^{-11}	BNL 777/865	[15]
$K^+ \to \pi^+\mu^-e^+$	5.2×10^{-10}	BNL 865	[16]
$K^+ \to \pi^-\mu^+e^+$	5.0×10^{-10}	BNL 865	[16]
$K^+ \to \pi^-e^+e^+$	6.4×10^{-10}	BNL 865	[16]
$K^+ \to \pi^-\mu^+\mu^+$	1.1×10^{-9}	NA48/2	[17]
$K^+ \to \mu^-\nu e^+e^+$	2.0×10^{-8}	Geneva-Saclay	[18]
$K^+ \to e^-\nu\mu^+\mu^+$	no data		
$K_L \to \mu e$	4.7×10^{-12}	BNL 871	[19]
$K_L \to \pi^0\mu e$	7.6×10^{-11}	KTeV	[20]

Table 1.1: Current status of searches for selected LFV and Lepton Number Violation $K^\pm$ decays for which limits might potentially be improved by NA62. Two of the best results obtained with K_L decays are also listed for comparison.

- $\mu^- \to e^+$ in the field of a nucleus [27].
- $K^+ \to \pi^- l^+ l'^+$ with $l^+l'^+ = e^+e^+, e^+\mu^+, \mu^+\mu^+$.

  In particular, the $K^+ \to \pi^-\mu^+\mu^+$ transition could be possible if mediated by a virtual Majorana neutrino, i.e. one which is its own antiparticle (Fig. 1.5).

The simplicity of kaon decays (few decay channels, low final-state multiplicities), together with the clear experimental signatures for LFV decays, enable efficient background rejection.

Kaon decay experiments have reached sensitivities to branching ratios of the order of 10^{-12}, that can provide access to mass scales up to 100TeV in the search for new physics at tree level [28].

The experiment E865 at BNL, which searched for the $K^+ \to \pi^+\mu^+e^-$ decay, set the 90% CL upper limit [16]:

$$BR(K^+ \to \pi^-\mu^+\mu^+) < 3.0 \times 10^{-9} \tag{1.36}$$

from the analysis of the invariant-mass distribution with assigned particle identification $M(\pi^-\mu^+\mu^-)$ (Fig. 1.4).

A similar search was performed by the NA48/2 experiment at CERN (Fig. 1.6), which took data during 2003 and 2004. Its primary purpose was to study direct CP violation in the K^+/K^- system.

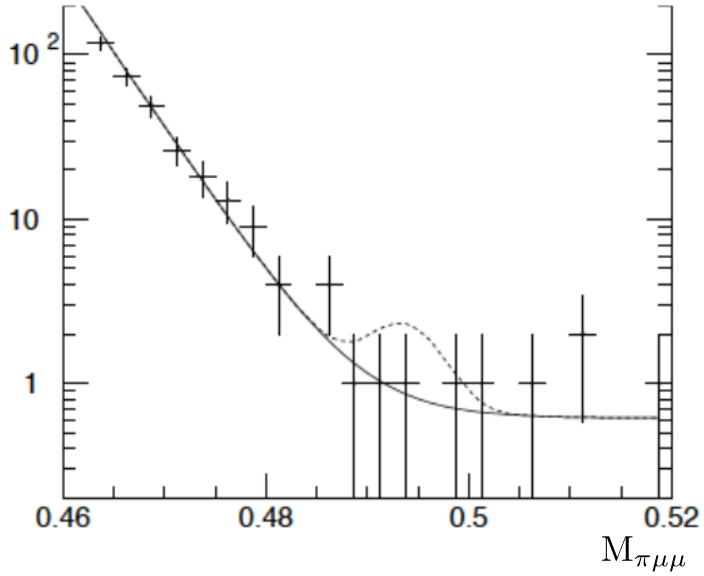


Figure 1.4: Invariant-mass distribution of $\pi^-\mu^+\mu^-$ event candidates from the experiment E865 at BNL [16]. The points with error bars are data, the solid line is a fit to an empirical function, and the dashed line is a fit that includes a signal at the 90% C.L. upper limit.

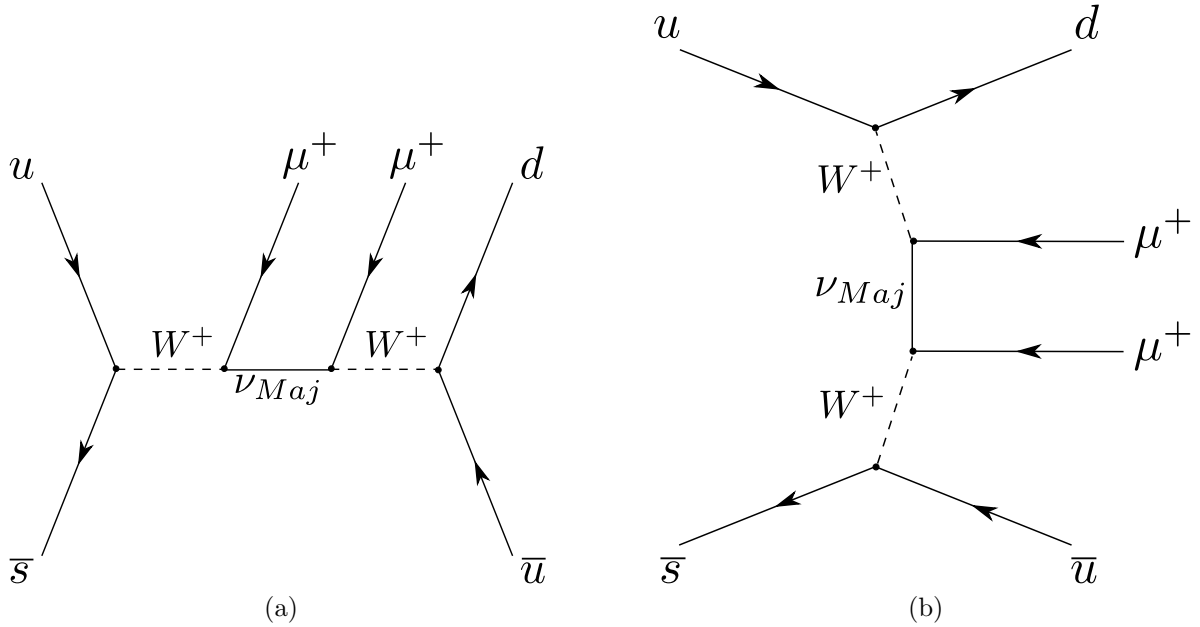


Figure 1.5: Lowest-order diagrams contributing to $K^+ \to \pi^-\mu^+\mu^+$. The decay can proceed if the neutrino is its own antiparticle, i.e. it is a Majorana fermion.

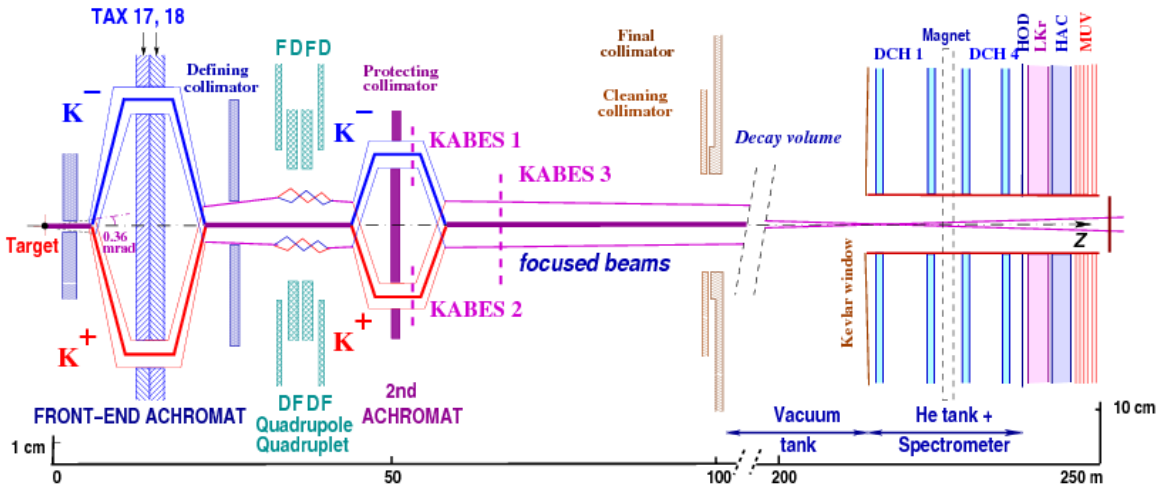


Figure 1.6: Schematic diagram of the NA48/2 experiment: two simultaneous 60GeV K^+ and K^- beams entered a 114m long vacuum decay tank, downstream of which, a 23m long, helium-filled spectrometer, and an analyzing magnet with a momentum kick of 120 MeV were used to track and analyze charged secondaries. Downstream of the spectrometer were located a scintillator hodoscope (Hod), the liquid-krypton electromagnetic calorimeter (LKr), an iron/scintillator hadronic calorimeter (HAC), and a stack of muon-veto detector (MUV).

The $K \to \pi\mu\mu$ (signal) and $K \to \pi\pi\pi$ (background) samples were collected with the same two-level trigger for three-tracks decays. Furthermore two tracks were required to be identified as muons by hits in the Muon Veto, in order to identify an event as signal.

Figure 1.7 shows the invariant-mass distributions for $\pi^\mp\mu^\pm\mu^\mp$ events (left) and $\pi^\mp\mu^\pm\mu^\pm$ events (right). NA48/2 identified ≈ 3000 signal candidates forming the peak at $m_{K^\pm}$ for the opposite-sign muons. Concerning the same-sign μ case, there are 52 signal candidates in the region near $M(\pi\mu\mu) = m_{K^\pm}$ and $52.6 \pm 19.8_{syst}$ background events expected from the Monte Carlo simulation, resulting in a threefold improvement of the 90% CL upper limit [31] with respect to the E865 result:

$$BR(K^+ \to \pi^-\mu^+\mu^+) < 1.1 \times 10^{-9}. \tag{1.37}$$

This analysis, however, was not fully optimized for the purposes of rejecting $K_{\pi 3}$ background. Subsequent analysis demonstrated that the $K_{\pi 3}$ events with $M(\pi\mu\mu) \approx m_{K^\pm}$ are all of the type where two of the pions decay to muons, and one of the $\pi \to \mu$ decays

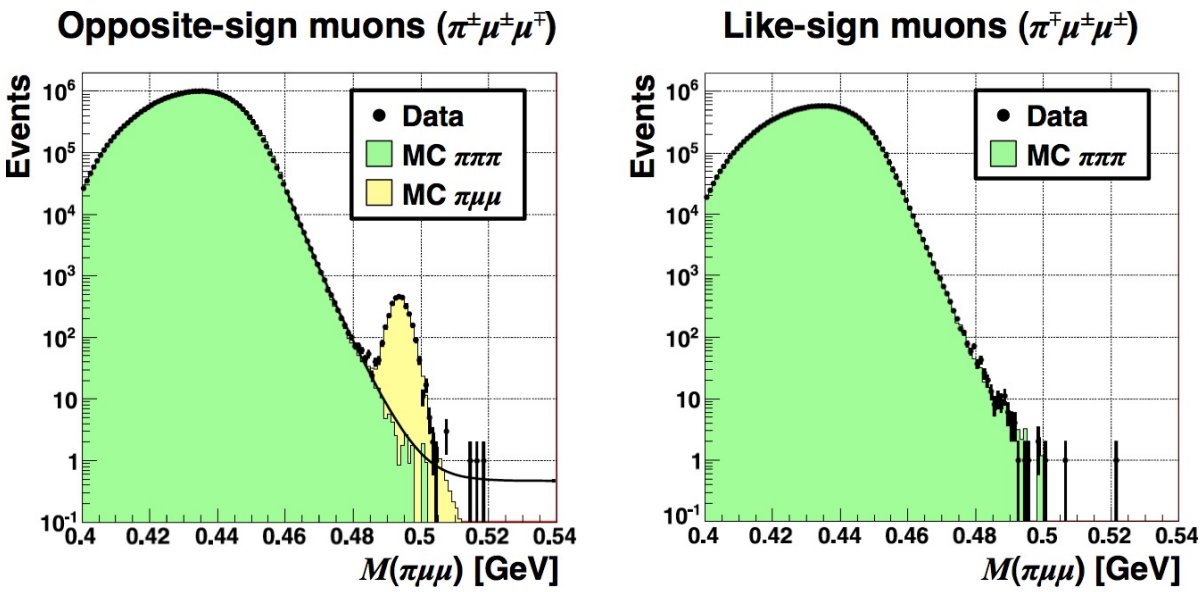


Figure 1.7: NA48/2 invariant-mass distributions for $K \to \pi\mu\mu$ candidates [31].

occurs downstream of the spectrometer magnet and upstream of the last drift chamber. In this topology, the latter track (which is identified as a muon) is misreconstructed; this can increase the apparent value of $M(\pi\mu\mu)$. Requiring that the pion track have $p > 20$GeV decreases the acceptance for signal events by 50%, but also eliminates all $K_{\pi 3}$ with $M(\pi\mu\mu) \approx m_{K^{\pm}}$. With the background reduced by at least an order of magnitude, the NA48/2 sensitivity is increased to $\sim 10^{-10}$.

1.4 NA62 Experimental apparatus

The 270m long experimental apparatus of NA62 extends from a K^+ production beryllium target to the beam dump (Fig. 1.8).

Beam elements and two detectors (KTAG/CEDAR and GTK), whose function is to measure the K^+ beam, cover the first 100m.

Subsequently, the beamline opens into the vacuum tank about 100 m downstream of the target. The vacuum tank is about 110 m long and fully encloses the four tracking stations of the magnetic spectrometer; the pressure inside which is kept at a level of 10^{-6} mbar.

Finally, the decay products are measured by the main detectors that extend downstream of the fiducial region.

Part of the experimental apparatus was commissioned during a technical run in 2012; installation continues and data taking is expected to begin in late 2014.

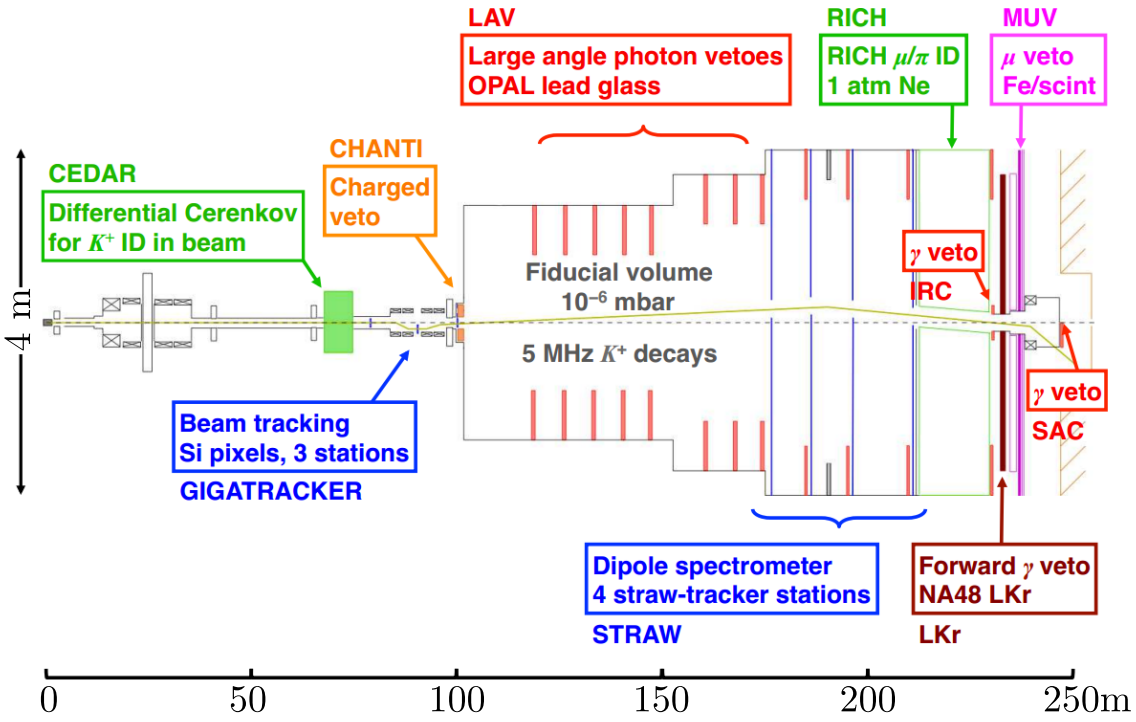


Figure 1.8: The NA62 beam line and detector.

1.4.1 The K12 beam

The experiment makes use of a 400 GeV primary proton beam from the SPS with 310^{12} protons per pulse and a duty factor of about 0.3.

After the extraction, these protons are focused on a beryllium target (2 mm in diameter and 400 mm in length), so as to produce other particles like protons, neutrons, photons, pions, hyperon, kaons and electrons. An absorber, along the proton beam direction, is used to dump all the neutral particles produced in the collision with the target.

Table 1.2: NA62 Beam characteristics.

400GeV/c protons on target /s	1.1×10^{12}
75GeV/c hadrons on target /s (6% Kaons)	7.5×10^{8}
75GeV/c K^+ decays /s	4.5×10^{6}

Downstream of the beryllium target, a system of dipole magnets deflects the secondary charged particles, and a copper/iron beam dump with a built-in slit allows the selection of positively charged particles in a narrow band of momenta 75GeV/c $\pm$ 1%.

This beam consists of about 525 MHz of π^+, 170 MHz of p, and 45 MHz of K^+, for a total rate of 750 MHz. The secondary hadron beam subsequently passes through acceptance-defining and cleaning collimators and a set of four quadrupoles of alternating polarities. In order to keep the beam within the beam pipe while traversing the apparatus, the beam is steered by a dipole (TRIM3) located upstream of the entrance of the decay volume to compensate for the deflections by the downstream spectrometer magnet.

1.4.2 KTAG/CEDAR detector

The KTAG is responsible for the identification of the K^+ component of the beam. In order to achieve this goal it employs a differential Čerenkov counter called CEDAR, based on the CERN CEDAR-W design [32].

The main feature of the KTAG is the very accurate timing (σ_t = 100ps). The beam identification from the KTAG/CEDAR is fundamental to the suppression of background from beam-gas interactions. Without it, the vacuum in the decay tank would have to be kept at the level of 10^{-8}mbar. The original CEDAR detector is a 4.35m long radiator vessel filled with gas at a controlled pressure (Fig. 1.9). Čerenkov light is reflected by a mirror located at the end of the vessel, onto a circular diaphragm with adjustable aperture width. In the original version of the detector, the Čerenkov light hits eight photomultipliers at the upstream end. In the KTAG, one of the CEDAR-W detectors has been refurbished to run with H_2 at 3.85 bar and outfitted with a new, high- segmentation readout.

Furthermore, the KTAG implements an additional external optics that projects the light spots radially, so that the light can be distributed over 48 PMTs in each spot

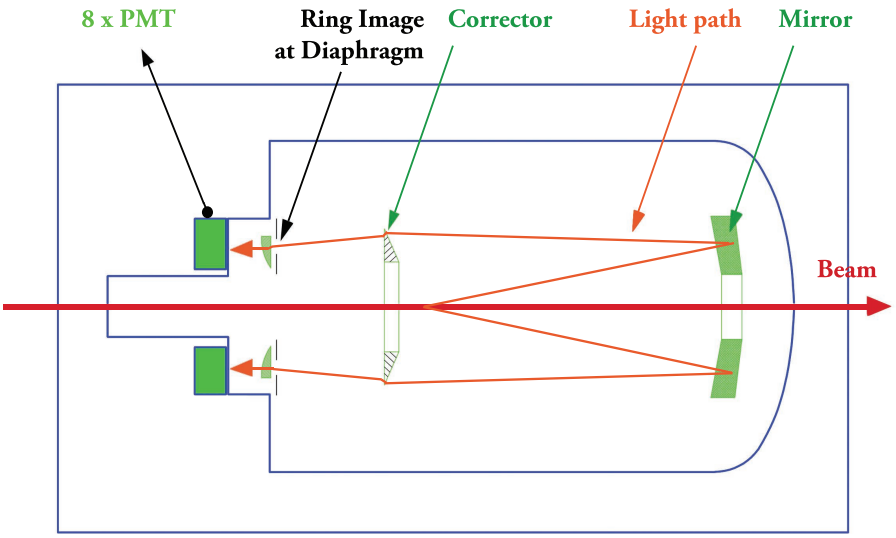


Figure 1.9: Sketch of the original CEDAR.

(Fig. 1.10).

Thanks to the photo detectors, the front end electronics and the readout system (TEL62, see section 1.5), the KTAG has high efficiency, excellent time resolution, and ability to work at high rate. With the help from the CEDAR/KTAG, the probability of mismatching the pri- mary and secondary tracks is held below 1%. Nevertheless, events with mismatched tracks still account for half of the events not rejected by kinematics.

1.4.3 GTK

The *Gigatracker* (GTK) is the beam spectrometer. It consists of three silicon pixel detectors. These detectors are crossed by the full beam intensity (Fig. 1.11) and provide a time resolution of 150ps.

An achromatic (no net bending) system of four magnets is used as a spectrometer for particles of any momentum (Fig. 1.12). The momentum can be measured with a resolution of 0.2% RMS, with the angular resolution being 15μrad on an event-by-event basis.

Since the full intensity beam passes through the GTK, this detector has strict requirements on the material budget, that is of 0.5% of X_0. Accordingly, the sensors are 200μm thick

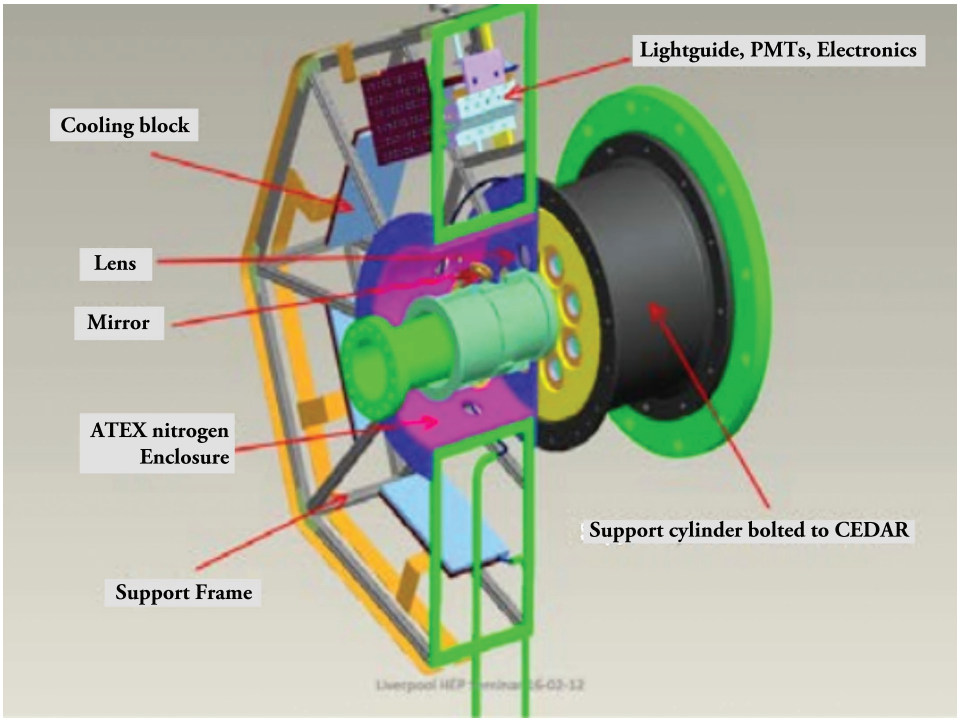


Figure 1.10: Optics of the KTAG.

flip-chip bonded to ten 100μm thick readout chips, which are cooled by means of a micro-channel cooling plate. The total thickness of one GTK station is less than 500μm.

1.4.4 CHANTI

The CHANTI is made of six guard-ring counters following the last GTK station. These counters are helpful to reduce critical background originating from inelastic interactions of the beam with the collimator and the Gigatracker (GTK) stations. The most critical events are the ones in which the inelastic interaction takes place in the last GTK station (GTK-3). In such cases, pions or other particles produced in the interaction, if emitted at low angle, can reach the straw tracker and mimic a K decay in the fiducial region.

Each guard-ring is composed of four layers of triangular plastic scintillator bars read out by wave length shifting (WLS) fibers and silicon PMTs (Fig. 1.13).

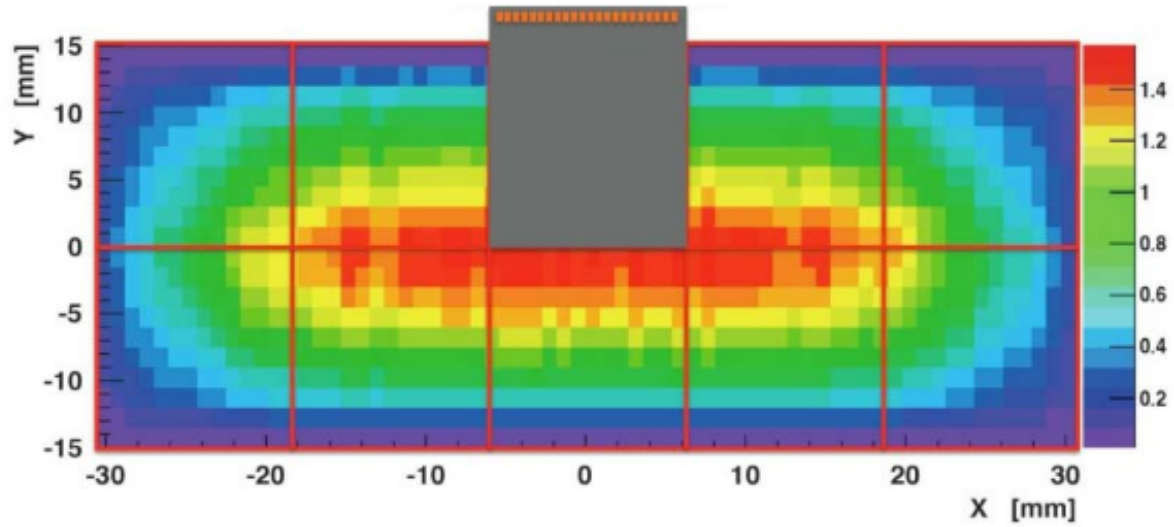


Figure 1.11: Beam intensity distribution expressed in MHz/mm^2, in the GTK detector.

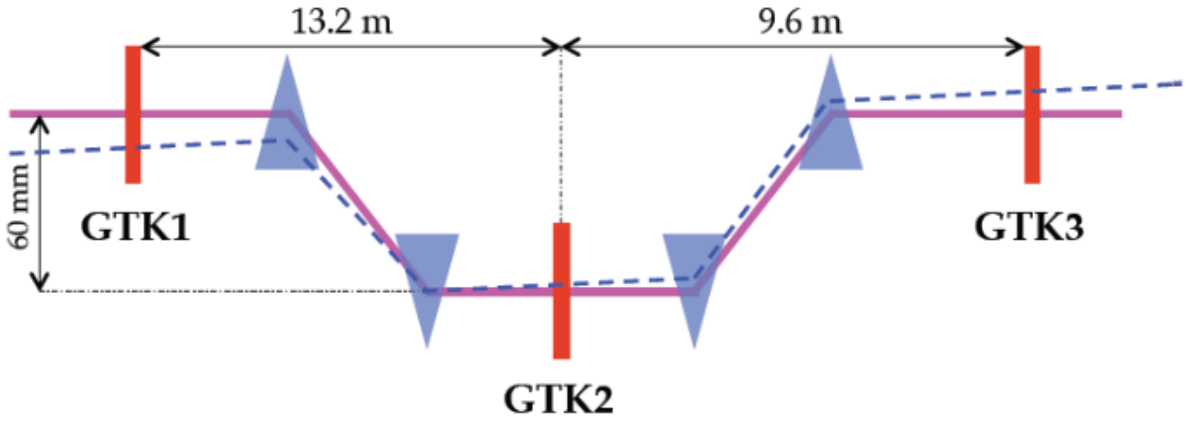


Figure 1.12: The secondary beam line passes through an achromatic system of four dipole magnets. This allows the GTK to measure time, direction and momentum of all the particles in the beam.

1.4.5 Photon Veto system

The design of the photon veto system is oriented to the suppression of the $K^+ \to \pi^+\pi^0$ background. The required inefficiency is 10^{-8} on the detection of the π^0. Since the upper cut on the π^+ momentum will be 35 GeV/c, the π^0 momentum amounts to at least 40 GeV/c. Consequently the photon veto system needs very high detection efficiency for high energy photons. Hence the inefficiency must be less than 10^{-5} for photons whose energy is above 10 GeV, and below 10^{-3} for photons above 1 GeV.

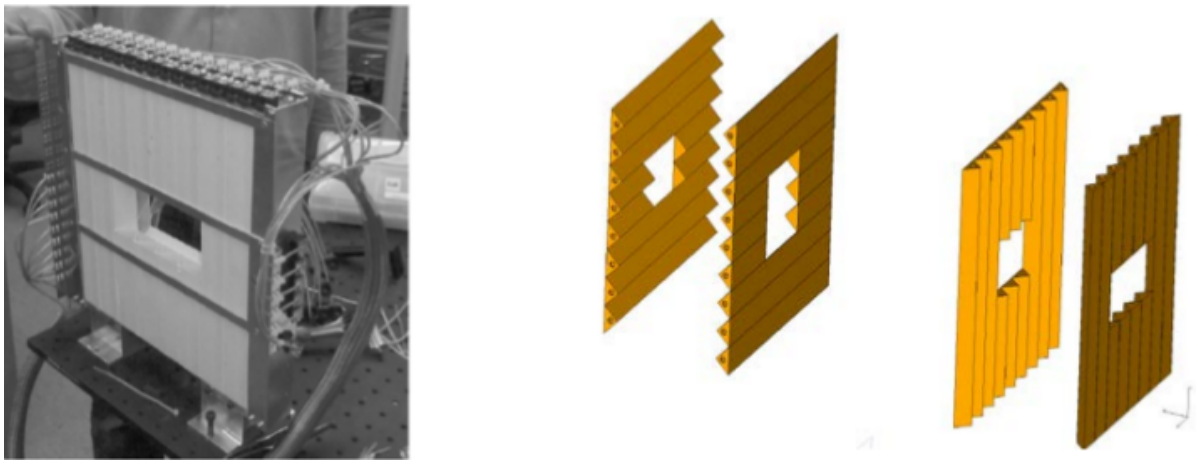


Figure 1.13: One CHANTI station is made of four layers of triangle-shaped scintillators each with a Wave Length Shifter fiber readout.

The photon veto system consists of four separate subdetector systems:

- Large-angle vetoes;
- The Liquid Krypton calorimeter.
- IRC;
- SAC;

LAV

The ring-shaped *large-angle photon vetoes* (LAVs) are placed at 12 stations spaced by 6 m along the vacuum volume in the upstream region and by up to 20 m for the most downstream station (Fig. 1.14). They provide coverage for decay photons between 8.5 mrad and 50 mrad.

The Liquid Krypton calorimeter

The NA48 *Liquid Krypton calorimeter* (LKr) is a detector built for the study of CP violation in the neutral kaon system performed at CERN in the NA48 experiment [17] (Fig. 1.15). The calorimeter is a quasi-homogeneous ionization chamber; the Kr radiation length is short ($X_0 = 4.7$cm). This makes possible to measure high energy (> 10 GeV)

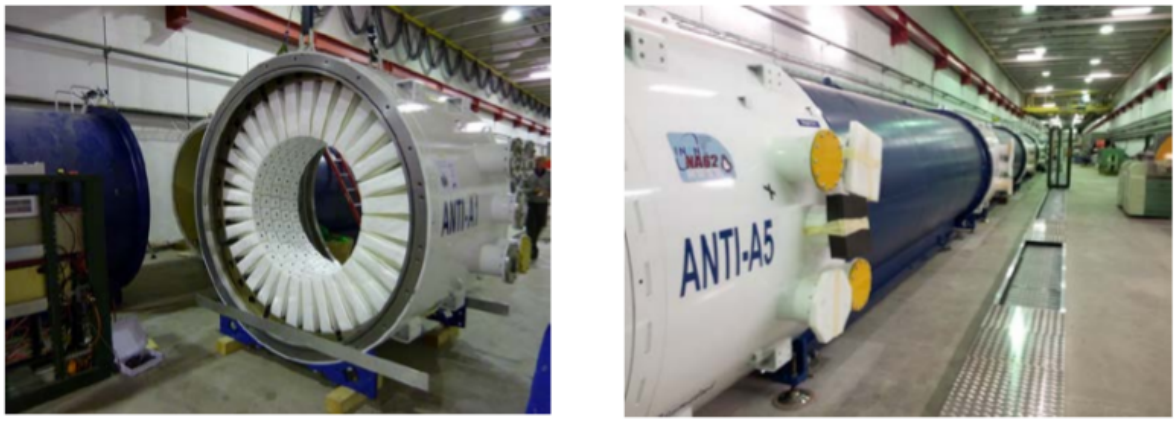


Figure 1.14: One of the LAV stations.

electromagnetic showers in a compact volume. The LKr calorimeter is octagonal, and contains a cylinder of 128cm in radius, 127cm thick along the beam direction, that corresponds to 27 radiation lengths. It contains a 9cm radius hole in the center for the beam pipe.

The total active volume is 7m^3 and is divided into 13248 cells (towers) of $2 \times 2\text{cm}^2$ cross-section. The anodes and cathodes are thin copper–beryllium ribbons running almost parallel to the beam. The central anode is kept at high voltage while the cathodes on either side are grounded, forming a drift gap of approximately 1 cm.

The energy resolution of the calorimeter, measured using an electron beam of different momenta (15, 25, 50, 100 GeV/c) and 0.1% momentum spread, can be parametrized as (Fig. 1.16):

$$\frac{\sigma(E)}{E} = \frac{3.2\%}{\sqrt{E(\text{GeV})}} \oplus \frac{9\%}{E(\text{GeV})} \oplus 0.42\% \tag{1.38}$$

The first term is due to stochastic sampling fluctuations, the second one to electronic noise and the last one to inhomogeneities of the material in front of the calorimeter and the non perfect inter-calibration of the cells.

In order to satisfy the NA62 rate requirements the LKr will be equipped with new *Calorimeter Readout Modules* (CREAM) shown in Figure 1.17, that provide Flash ADC

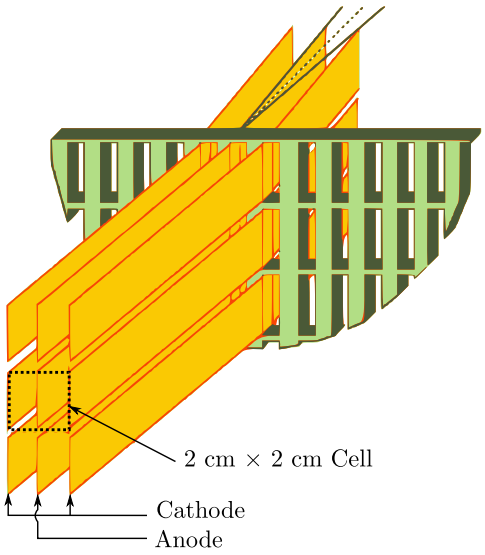


Figure 1.15: Detail of the LKr cell structure showing the "accordion geometry" structure of the ribbons.

readout with a sampling rate of 40 MHz. The digitized inputs are loaded into a circular buffer that can store raw data for 10 ms.

IRC and SAC

In order to detect and veto photons at very small angles ($\theta < 1$mrad), NA62 employs two Small Angle Calorimeters (Fig. 1.18):

- the *IRC* that covers the annular region around the inner radius of the Liquid Krypton Calorimeter,
- *small-angle shashlyk calorimeter* (SAC) around which the beam is deflected completes the coverage for very-small-angle photons that would otherwise escape via the beam pipe.

1.4.6 Straw Tracker

The *Straw Tracker* task is to measure the spatial coordinates and momentum of secondary charged particles originating in the decay region (Fig. 1.19). It is composed of four

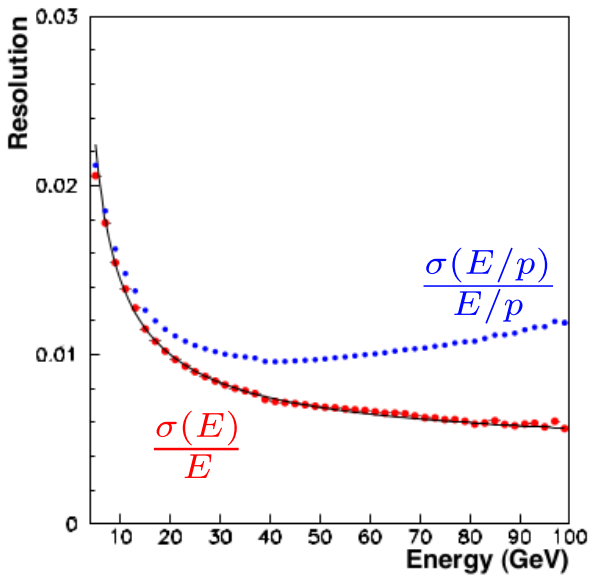


Figure 1.16: Energy resolution of the LKr calorimeter, measured using an electron beam, as a function of the energy. The resolution of the E/p ratio between the energy, as measured by the calorimeter, and the momentum of the electron, as measured by the magnetic spectrometer, is shown as well.

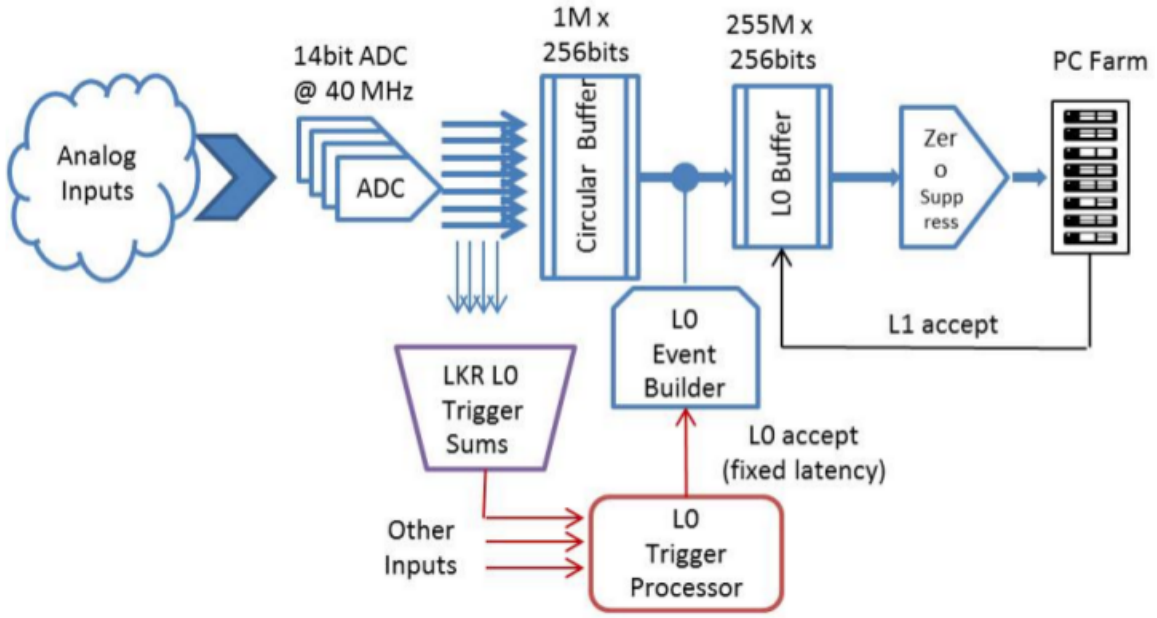


Figure 1.17: Scheme of the CREAM readout.

stations and between the second and third Straw chambers there is a large aperture

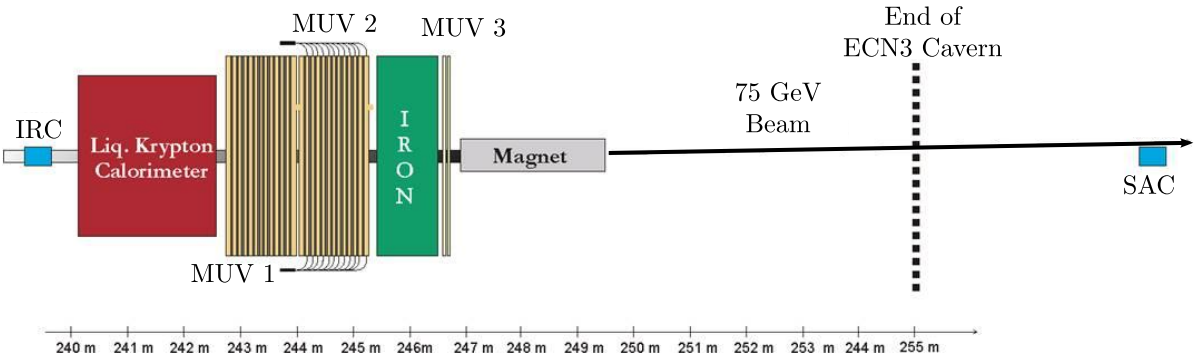


Figure 1.18: The IRC and the SAC (at the end of the beam line), are used to veto photons at very small angles.

dipole magnet that generates a vertical B-field of 0.36T.

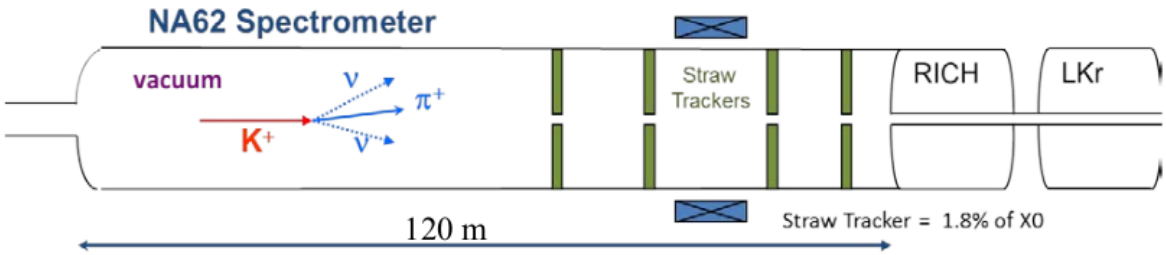


Figure 1.19: Four straw chambers are located inside the vacuum tank, with a dipole magnet in the middle.

Each Straw Tracker is made of 1792 straws, positioned in four directions (views) in order to measure spatial coordinates (Fig. 1.21).

Each straw is made of 36μm thin foil of polyethylene terephthalate (PET) welded to form a 2.1m long and 9.8mm diameter tube. Electrical conductance is guaranteed by the inner copper/gold coating. It provides precise tracks information with a resolution $< 120\mu$m.

1.4.7 RICH

I will describe in some more detail the RICH because it is the detector on which my work focused.

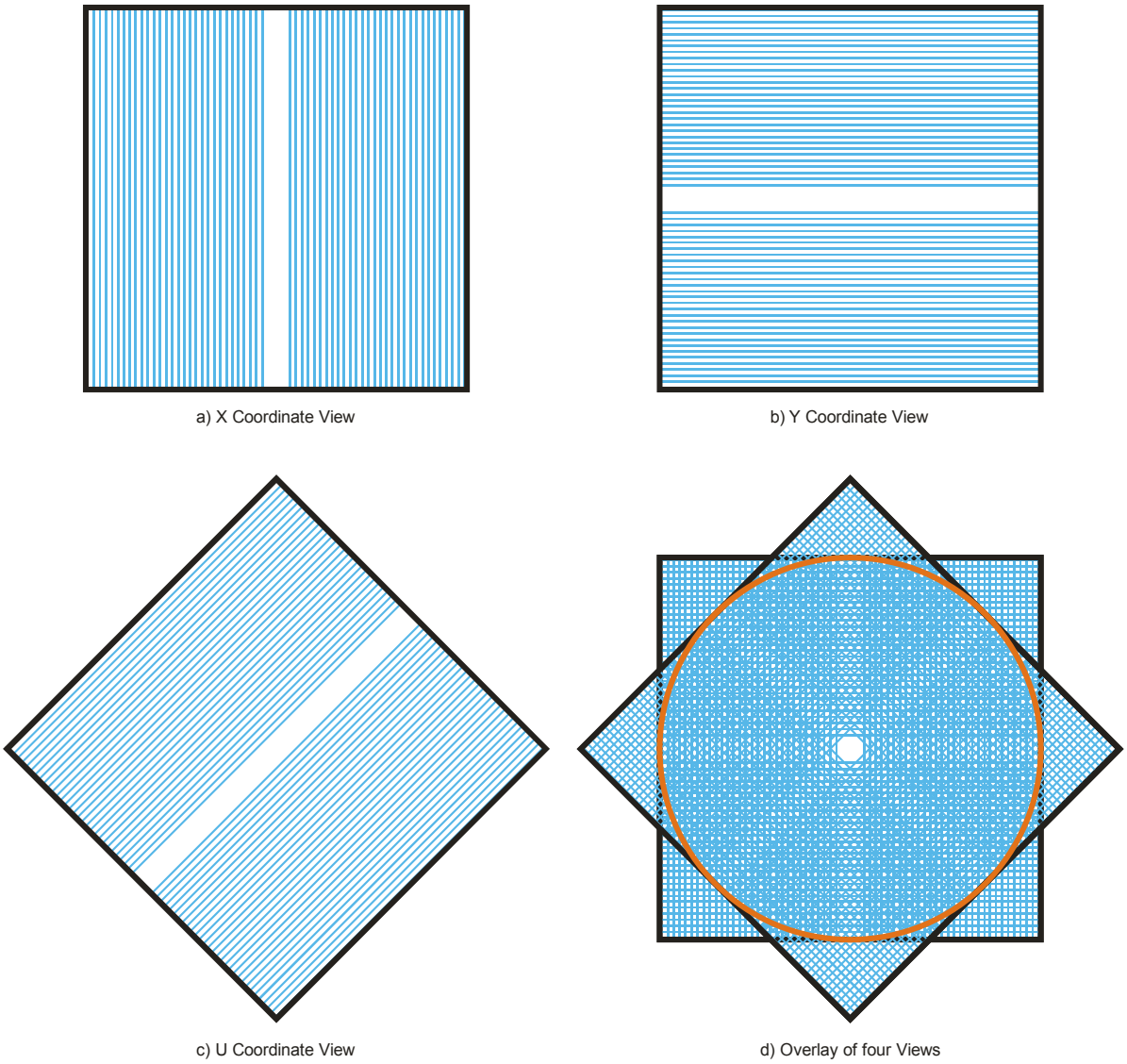


Figure 1.20: The four views measured by a single Straw Tracker.

A *RICH* (Ring Imaging Čerenkov) detector is based on the Čerenkov effect. A spherical mirror, placed at one end of the vessel, reflects the produced cone of Čerenkov light on its focal plane where photon detectors are located.

For a particle traveling along a direction perpendicular to the focal plane, the radius r_{ring} of the Čerenkov ring that is projected on the detector is given by:

$$r_{ring} = f \tan \vartheta_c \tag{1.39}$$

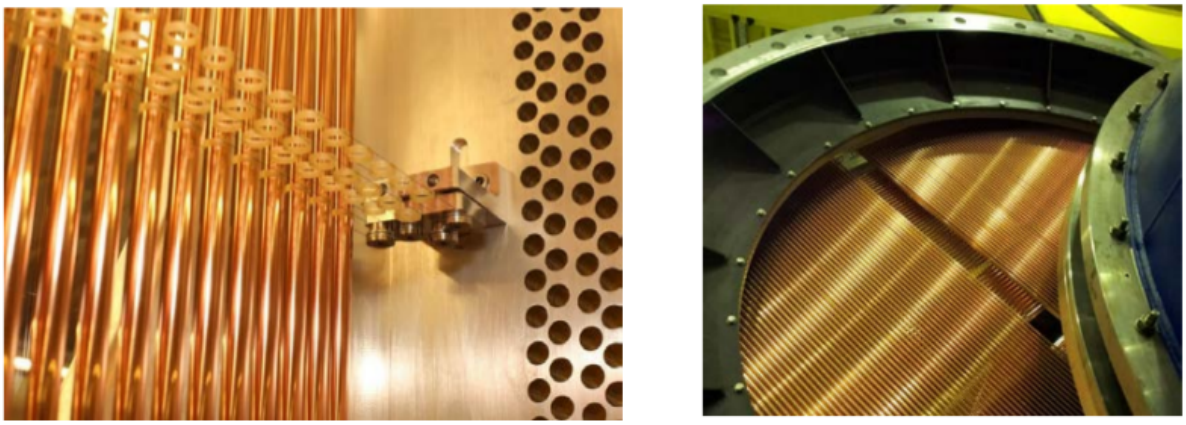


Figure 1.21: Photo of a Straw Tracker chamber installed during the 2012 Technical Run.

where f is the mirror focal length, that is equal to a one-half of the radius for a spherical mirror, and ϑ_c is the angle with respect to the direction of the particle at which Čerenkov photons are emitted. More in general, if α is the angle that the direction of the particle forms with the focal plane, the ring will be an ellipse with the major and minor semi-axes given by:

$$r_{min} = f \tan \vartheta_c; \qquad r_{max} = f \frac{\tan \vartheta_c}{\cos \alpha} \tag{1.40}$$

Consequently, it is possible to obtain the Čerenkov angle from the measurement of radius of the ring and eventually, using information about the particle momentum from the Straw spectrometer, to determine the mass of the particle.

Furthermore, the direction (θ_x, θ_y) of the particle with respect to the z-axis is related to the coordinates (x_c, y_c) of the center of the ring by the following relations:

$$\theta_x = \tan^{-1}\left(\frac{x_c}{f}\right), \quad \theta_y = \tan^{-1}\left(\frac{y_c}{f}\right). \tag{1.41}$$

NA62 RICH detector design

The main purpose of the NA62 RICH detector is to improve the muon identification: it must be capable to distinguish pions from muons in the range $15 \div 35$ GeV/c.

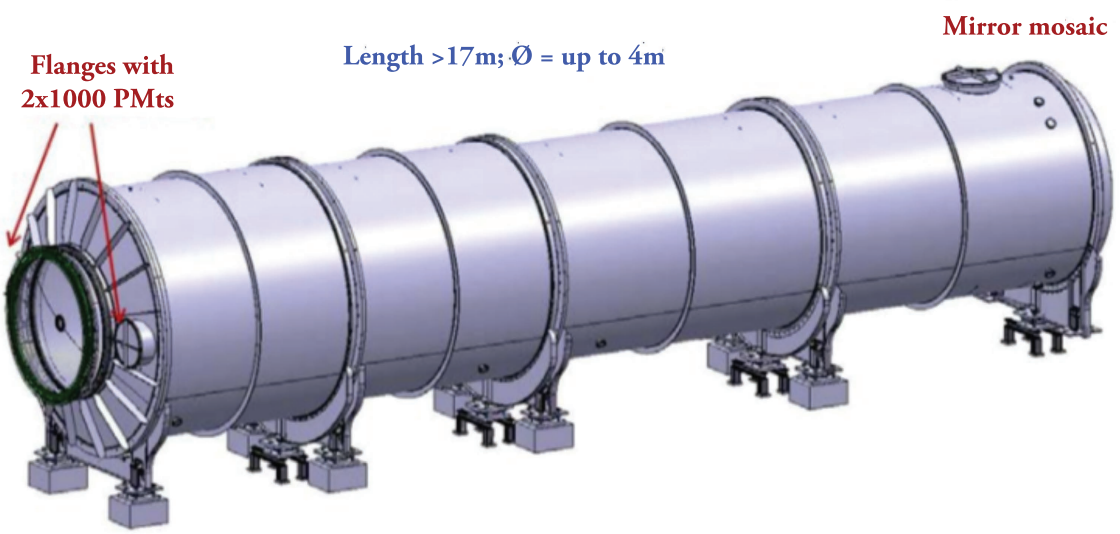


Figure 1.22: Sketch of the NA62 RICH detector layout. The beam direction is left to right.

The threshold momentum below which no Čerenkov radiation is emitted is:

$$p_t = \frac{m}{\sqrt{n^2 - 1}} \tag{1.42}$$

where n is the refractive index and m the mass of the charged particle. However, full efficiency is achieved at momenta about 20% higher than the threshold. For the above reason the Čerenkov threshold should be 12.5GeV/c for a pion.

Setting a threshold of $p_t = 12.5$ GeV/c, gives a requirement on the refractive index $(n-1) = 62 \times 10^{-6}$ that is close to the value of the refractive index for Neon at normal temperature and pressure. The choice of filling the vessel with Neon at room temperature is also supported by its other properties:

- Good light transparency in the visible and near-UV regions;
- Low chromatic dispersion;
- Low atomic weight, to minimize radiation length;
- Non-flammable.

A mosaic of spherical mirrors is used to image the Čerenkov light cone into a ring on the focal plane of the mirrors. In order to avoid the absorption of the reflected light due to

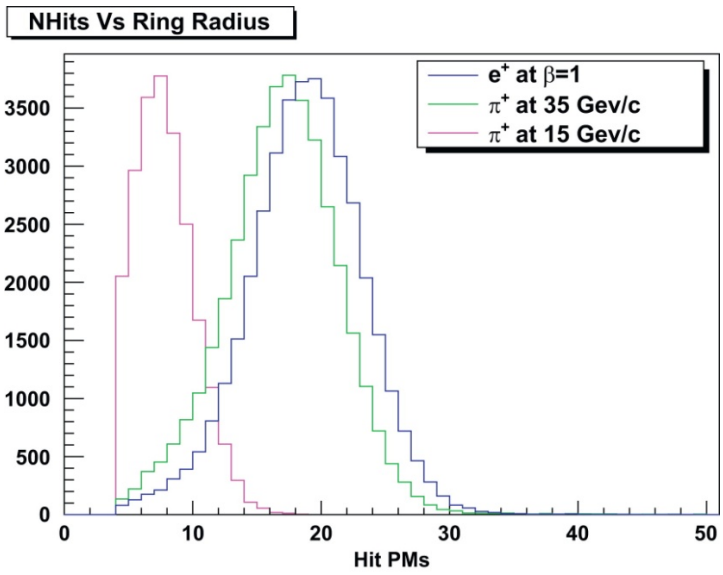


Figure 1.23: Number of hit PMTs for different charged particles with different momentum.

the beam pipe, the mirrors are divided into two different spherical surfaces, one with the center of curvature on the left-hand side and the other on the right-hand side of the beam pipe. The total reflective surface exceeds 6 m^2 and required the use of a matrix of 20 mirrors.

The spherical mirrors employed have a nominal curvature radius of 34 m and hence a focal length of 17 m. The 25 mm thick glass that forms the mirror is coated with aluminum and with a dielectric film in order to protect the mirrors themselves.

In order to maximize the angular resolution of the detector, the granularity of the photon detection system has to be as small as possible. On the other hand, a large number of *photomultiplier tubes* (PMTs) directly impacts on the cost of the RICH apparatus.

Accordingly, a reasonable compromise was found with a number of PMTs of 1952.

1.4.8 CHOD

The existing NA48 *charged hodoscope* (CHOD) is located downstream the RICH detector and soon upstream the LKr calorimeter.

It provides a fast trigger for charged decay products with a time resolution of 150 ps.

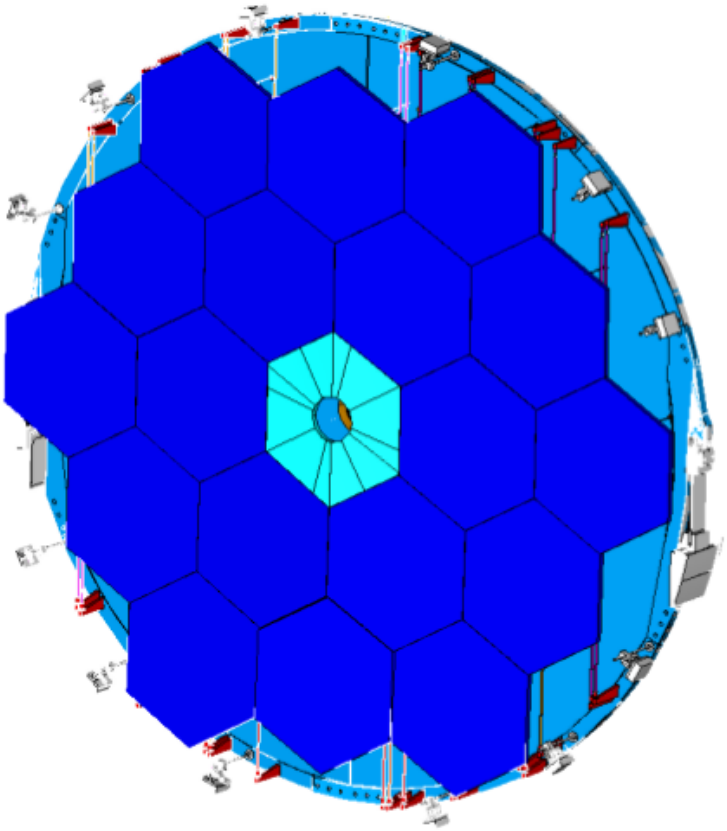


Figure 1.24: The NA62 RICH detector mirror system.

The detector consists of 128 plastic scintillator slabs arranged in two planes, separated by 75 cm, of 64 horizontal and 64 vertical scintillators (Fig. 1.25). The thickness of each slab is 2 cm (0.05 radiation lengths), whereas the length varies from 60 cm up to 121 cm and the width from 6.5 cm (in the region close to the beam pipe) to 9.9 cm. The CHOD has a central hole of 108 mm in radius for the beam pipe. Each CHOD plane is divided into four quadrants (made of 16 slabs). The scintillation light produced at the passage of a charged particle is collected at one end of a counter by a fish-tail shaped light guide connected to a photo-multiplier.

1.4.9 Muon Veto

The *Muon Veto* system (MUV) is composed by three detectors, called MUV1, MUV2, MUV3.

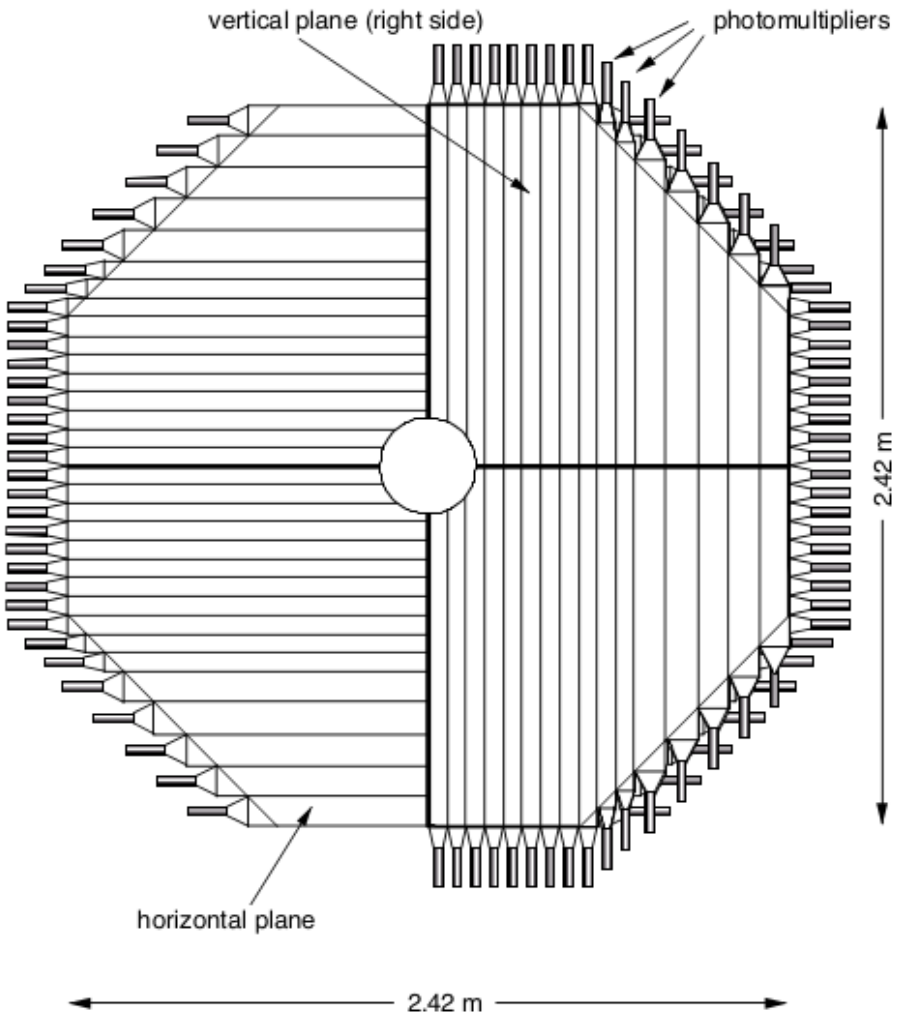


Figure 1.25: The NA48 charged hodoscope.

MUV1 and MUV2 are iron-scintillator sandwich calorimeters with respectively 24 and 22 layers of scintillator strips, alternating in horizontal and vertical directions. MUV1 is a newly constructed detector, whereas MUV2 was the former NA48 hadron calorimeter front module.

The MUV3 detector is located downstream an additional 80cm thick iron wall and is made of a matrix of 5 cm thick scintillator tiles each with its PMT readout. Its task is to quickly veto muons in the trigger.

1.5 Trigger and Data Acquisition

The *Trigger and Data Acquisition* (TDAQ) system at NA62, provides essential preliminary online analysis of the raw data for the purpose of filtering potentially interesting events into the data storage system. At the design input rate of 10MHz this is a formidable task, which requires real-time handling of raw data.

To keep the overall data rate within the current capability of the archival storage system of few kHz, the trigger system must achieve a rejection ratio for background events of at least 1,000:1. However a loss-less data acquisition system is mandatory to avoid adding artificial detector inefficiencies when vetoing background particles. Accordingly, the detector *readout* and the DAQ systems are unified in a single digital system.

The TDAQ system is structured in three levels.

1.5.1 Level 0

The standard first level of trigger will be completely *hardware based*, and in charge of reducing the 10MHz input rate to at least 1MHz. It is completely *synchronous*, and trigger decisions must be taken within *few milliseconds*. This time depends on the size of the buffers and on the trigger rate, and could be increased in the future. Since the response time is critical, the Level 0 exploits only the positive information from the fastest detectors, such as RICH and CHOD, combined with the veto information from LKr, MUV and LAV.

The L0 trigger primitives are evaluated on the same board (TEL62) in which the data are stored waiting for the trigger decision. The TEL62 board is a general purpose board (an upgraded version of the TELL1 board developed by EPFL for the LHCb experiment [46]) with 5 FPGA and large buffers to store the data, waiting for the trigger decision delivered to the board through the CERN standard *Timing, Trigger and Control* (TTC) interface [47]. The trigger primitives produced in the TEL62 are sent to the *L0 trigger processor* (L0TP) as UDP datagrams using dedicated Gigabit Ethernet connections (Fig. 1.26).

The events passing L0 trigger selection will go into the L1 trigger via Ethernet connections.

Sub-detector	Stations	Channels per station	Tot. channels	Hit rate (MHz)	Raw data rate (GB/s)
CEDAR	1	240	240	50	0.3
GTK	3	18000	54000	2700	2.25
LAV	12	320–512	4992	11	0.3
CHANTI	1	276	276	2	0.04
STRAW	4	1792	7168	240	2.4
RICH	1	1912	1912	11	0.8
CHOD	1	128	128	12	0.1
IRC	1	20	20	4.2	0.04
LKr	1	13248	13248	40	22
MUV	3	176–256	432	30	0.6
SAC	1	4	4	2.3	0.02

Table 1.3: Payload net rates for the NA62 experiment sub-detectors.

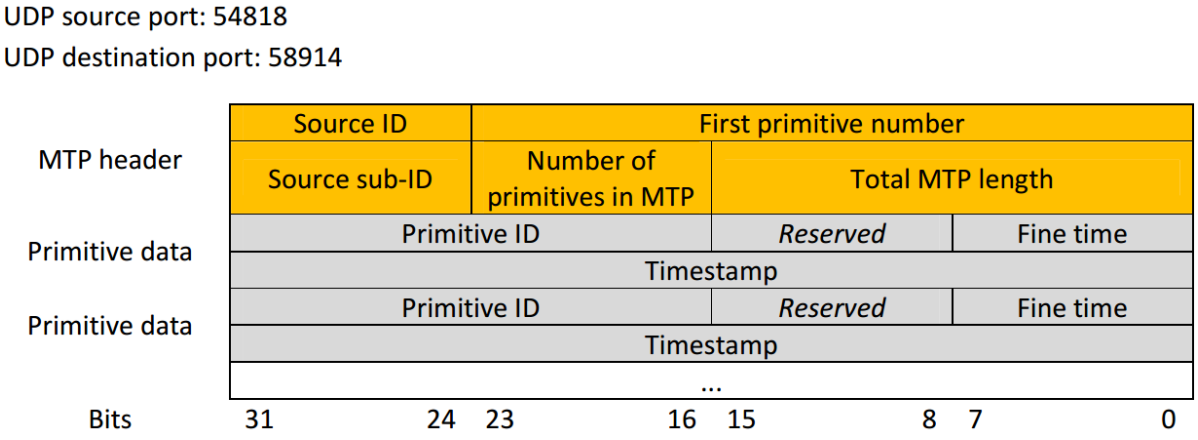


Figure 1.26: The NA62 trigger primitives data format.

1.5.2 Level 1 and Level 2

L1 and L2 are implemented in the readout PC farm. The L1 will apply, in software, a further reduction factor of 10. The algorithms at this level analyze the data coming from a single detector. At the L2 the full event is reconstructed and a more complete event selection is applied in order to reduce the rate to tens of kHz. The latency in the software levels is not fixed, but all the events have to be processed before the next accelerator burst (order of 20-30 s).

1.5.3 Parallel computing at NA62

The quest for reducing significantly the 10 MHz data rate delivered to the detectors, together with the retention of physics signal potentially interesting for searches of new physics phenomena, led to the evaluation of modern multi-core and many-core architectures for the enhancement of the foreseen TDAQ system. The primary goal of the NA62 TDAQ is to apply a specific set of physics selection algorithms on the events read out and reject the events without any interesting physics content.

By its very nature, the TDAQ relies on technologies that have evolved extremely rapidly but that cannot rely anymore on an exponential growth of frequency guaranteed by the manufacturers. The performance of modern processors is now increasing through a growing parallelism in their architecture. Graphics Processing Units (GPUs) are massively parallel architectures that can be programmed using extensions to the standard C and C++ languages. In a synchronous system they are proven to be highly reliable and show a deterministic time response. These two features allow them to be well suited to run pattern recognition algorithms on detector data in a trigger environment.

These algorithms are often parallel by their very nature, and the data produced by a particle physics detector reduces the pattern recognition task to its purest form. From the physics perspective, such an enhancement of the trigger capabilities, already at the Level 0 of the TDAQ chain, would allow inclusion of new triggers and the selection of events that are currently not recorded efficiently.

1.6 LFV searches at NA62 experiment

In the NA62 fiducial volume there will be about 10^{13} K^+ decays in two years of data taking (see Table 1.2), corresponding to a 60-fold increase in statistics with respect to NA48/2.

With respect to the earlier NA48 setup, in NA62 the GTK detector will improve the kinematic reconstruction for charged tracks by means of a fast and precise tracking of individual beam particles and the four new Straw Tracker spectrometer will improve tracking on secondary particles; the spectrometer magnet will produce a higher kick

(270MeV), resulting in an improved invariant-mass resolution for three-track vertices (from $2-4$ MeV/c^2 to $1-2$ MeV/c^2).

Finally, the redundant particle identification system that includes the RICH and the MUV system will be included in the trigger. Hit multiplicity from the RICH might give a further contribution to the reduction of the background rates due to charged particles, while the MUV3 can contribute to the identification of muons in the final state.

Another contribution to the improvement of the kinematic reconstruction could originate from the inclusion of the *pattern recognition* of the Čerenkov rings in the trigger. This is currently not possible using standard trigger electronics that use FPGA. The use of Graphics Processing Units (GPUs) for this purpose has been explored and more details will follow in the next chapters.

Simulations show that all the upgrades discussed above contribute to eliminate the tail in the invariant-mass distribution from $K_{\pi 3}$ in Figure 1.7 making it possible for NA62 to reach a single-event sensitivity for $K^+ \to \pi^- \mu^+ \mu^+$, of 10^{-12} [30].

Chapter 2

Parallel computing

The intent of this chapter is to cover the concepts and the ideas that motivated the design of the parallel trigger framework and the concurrent algorithms developed.

In particular it goes through:

- The transformation of computing systems from flexible, centrally managed and powerful architectures (*homogeneous computing system*) to systems that use a variety of different types of specialized computational units (*heterogeneous computing system*);

- Data parallel execution models and parallel programming techniques. Since almost ten years, transistor frequency growth has stopped, and parallel techniques are one of the best tools that users have to keep improving the performance of their software.

- A comparison between Central Processing Units (CPUs) and Graphic Processing Units (GPUs) designs. They are representatives of different design philosophies, and being able to assign tasks to the best suited architecture leads to improved performance and lower power consumption.

- A brief description of the NVIDIA CUDA® API and how it can be used to compute and communicate with a GPU device.

2.1 Heterogeneous Parallel Computing

The whole industry, from mobile to supercomputing, is moving into heterogeneous parallel computing. This is motivated by the fact that heterogeneous parallel computing systems can achieve much better energy efficiency and higher performance than the traditional computing systems.

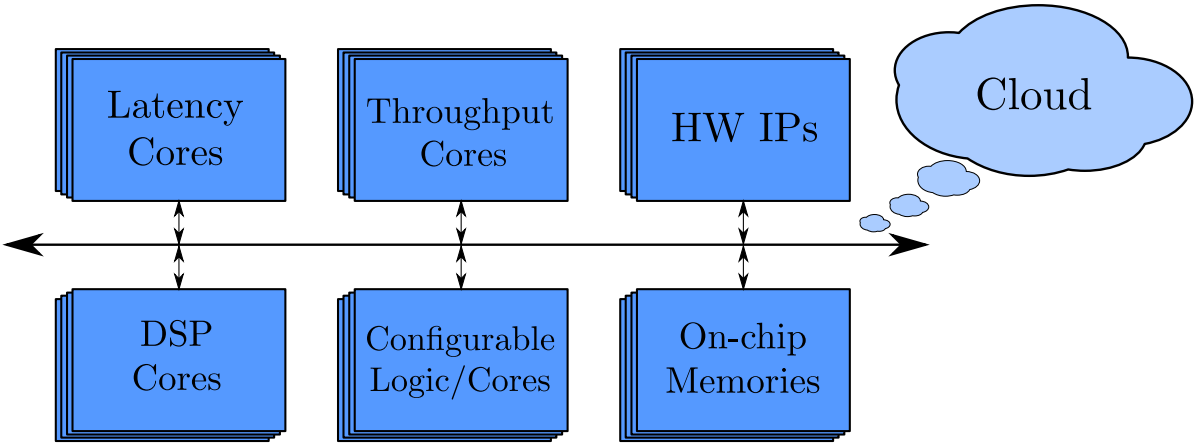


Figure 2.1: A typical SoC design for a mobile application.

The idea behind heterogeneous parallel computing systems is to use the best match for the job. Figure 2.1 shows a typical design for a *System on a Chip* (SoC) for a mobile application: it is a whole mixture of different kind of specialized cores. In a truly heterogeneous parallel computing environment, services offered *in the cloud* [33] can also be instrumental, especially in mobile computing systems.

2.1.1 Supercomputers

Heterogeneous parallel computing is not just used in mobile systems. On the other side of the spectrum there are *supercomputing systems* that are using heterogeneous computing in a big way. Supercomputers are designed to achieve high computing power and are usually dedicated to execute heavy computations. These systems belong to big companies or research institutions and usually their employees/researchers share the computing power available by making use of batch queue systems.

To ease the choice of the best architecture for a job, the definition of the performance of a computer must be independent from the architecture. The LINPACK Benchmark [34] was introduced by Jack Dongarra during the 1970s, and consists in solving a dense system of linear equations (i.e. the matrix associated with the system contains mostly non-zero values). It is possible to scale the size of the problem and to optimize the software in order to achieve the best performance possible on that system. No single number can ever reflect the overall performance of a system, but the measured performance of a system for solving a dense system of linear equations gives a good approximation of the peak performance of that system. Since 1993, the TOP500 project [35] lists every six months the world 500 fastest supercomputers, ranked according to LINPACK benchmark results.

The performance of a computer is measured in *FLOating-point operations Per Second* (FLOPS) that is the number of operations executed on 64-bit (double precision) data in the unit time.

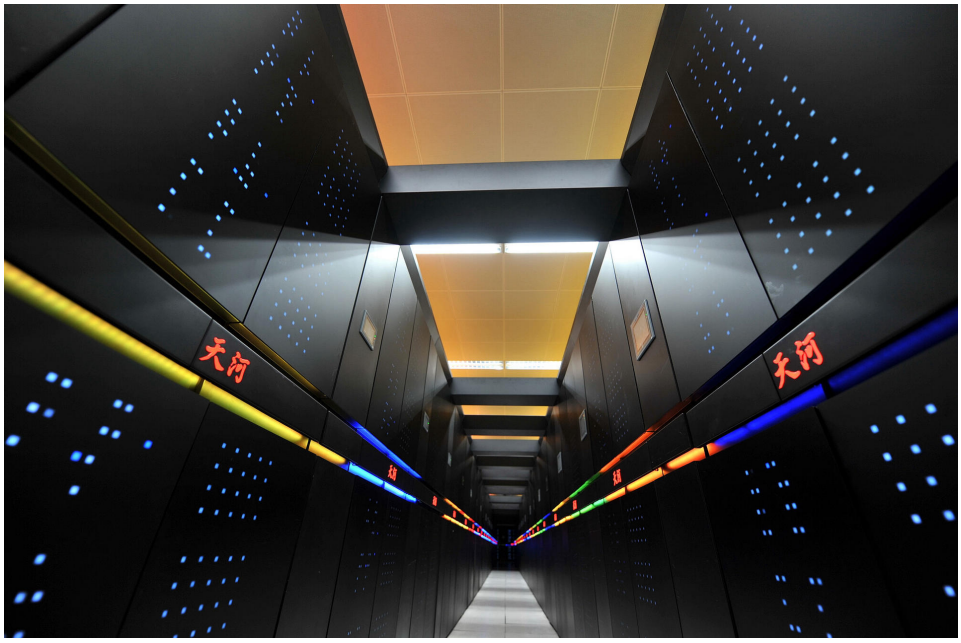


Figure 2.2: Tianhe-2, a supercomputer developed by China's National University of Defense Technology.

Tianhe-2 supercomputer, located in Guangzhou, China, (Fig. 2.2) is currently the fastest supercomputer in the TOP500 (November 2013) and can execute up to 33.86 PFLOPS

thanks to the heterogeneous use of *Central Processing Units* (CPUs) and Intel® Xeon Phi™ coprocessors.

2.2 Principles of parallel computing

Software has been traditionally written for *serial* computation (Fig. 2.3):

- the sequence of instructions that forms the problem is executed by one Processing Unit;
- every instruction has to wait for the previous one to be completed before its execution can start;
- at any moment in time, only one instruction may execute.

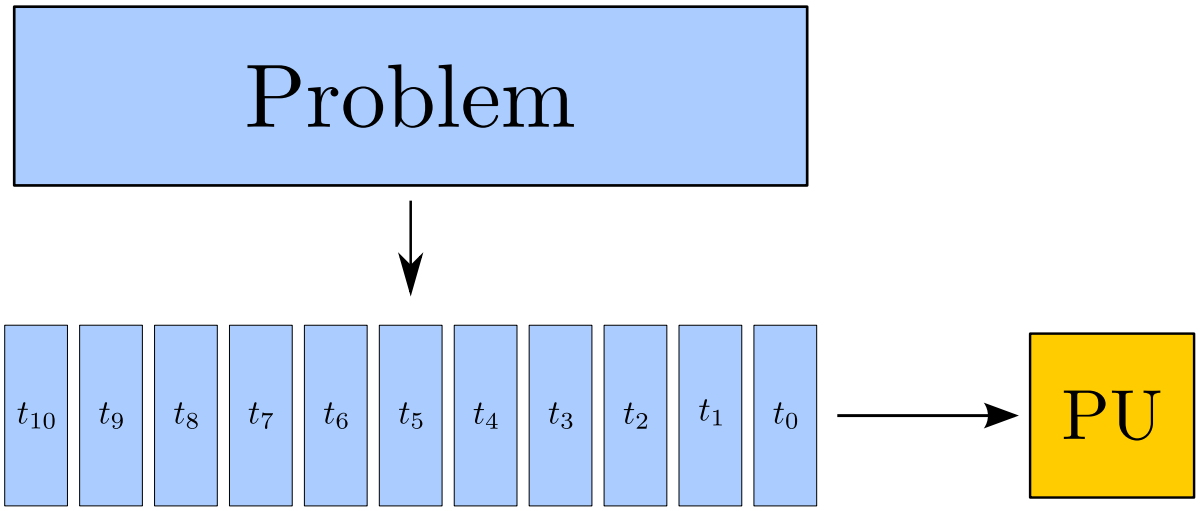


Figure 2.3: Serial computing: Every chunk of the problem is executed *serially* by a single *Processing Unit* (PU).

2.2.1 Moore's Law

In 1965 Gordon Moore, one of the co-founders of Intel, hypothesized that the performance of microprocessors and the number of their transistors would have doubled every 12 months [36]. This hypothesis, that takes the name of *Moore's Law* is still valid and it

was elaborated in its final form, stating that microprocessor performance doubles every 18 months (Fig. 2.4).

In 1997 Intel launched a CPU called *Pentium II* that had a frequency of 300 MHz with 7.5 millions transistors. Then, in late 2000, Intel launched the *Pentium 4* CPU with 42 millions transistors running at 1.5 GHz. In 42 months the performance of microprocessors increased by a factor 5 as predicted by Moore's Law.

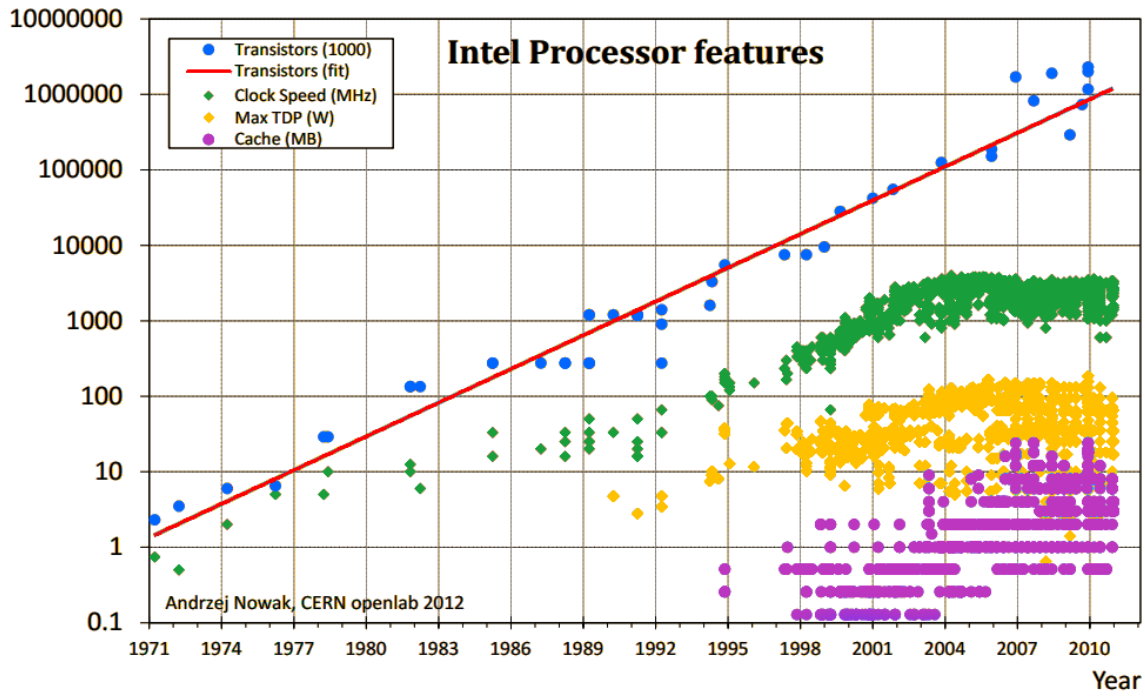


Figure 2.4: Moore's Law. After 2004 the power consumption and the frequency remained uniform due to the Power Wall.

At this stage, engineers figured out that they could speed up computation by increasing the core clock speed and thus started the march towards higher clock frequencies. However, this progress hit a brick wall commonly known as *Power Wall*, since increased clock rates resulted in too much wasted energy in the form of heat [37].

To face this paradigm change, Moore's law has been reinterpreted:

- The number of cores per chip will double every two years;
- Instruction parallelization increases (Fig. 2.5).

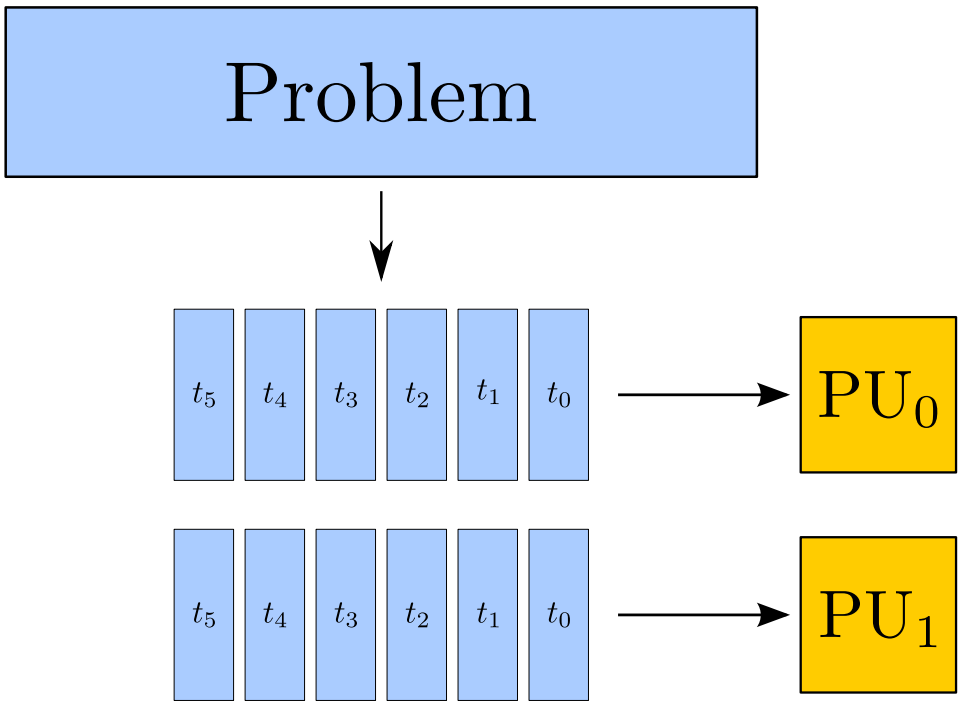


Figure 2.5: Problem is solved by more than one Processing Unit *concurrently*.

The idea of parallelism is not new: in 1842 Luigi Federico Menabrea, while describing the Analytical Engine by English mathematician Charles Babbage, highlighted the possibility that different components belonging to the same engine concurrently evaluate independent elements of a table [38]. During the 1940s, von Neumann proposed methods for solving differential equations on a discrete grid, with all the elements of the grid being updated in parallel by determining how the neighbors influence each other. Parallel architectures became a real work tool only in the 1960s.

2.2.2 Flynn's Taxonomy

A classification of computers called Flynn's taxonomy [39] describes four computer classes in both a serial and parallel context. The four classes are:

- **SISD** - *Single Instruction stream - Single Data stream.* A single processor computer that executes one stream of instructions on one set of data. Single-core processors belong to this class.

- **SIMD** - *Single Instruction Stream - Multiple Data stream.* A multiprocessor where each processing unit executes the same instruction stream as the others on its own set of data. In other words, a set of processors shares the same control unit, and their execution differs only by the different data elements each processor operates on.

- **MISD** - *Multiple Instruction stream - Single Data stream.* Each processing element of the multiprocessor executes its own instructions, but operates on a shared data set.

- **MIMD** - *Multiple Instruction stream - Multiple Data stream.* Each processing element executes its own instruction stream on its own set of data.

Most of the work described in this thesis is centered on SIMD and MIMD classes.

2.2.3 Strong scaling

When programming a multiprocessor machine, one tries to exploit every core in order to achieve the highest speedup possible. In an 8 core machine, one could expect to get a factor 8 in performance speedup after optimizing an application to run on all the machine cores. This is normally not true because the maximum theoretical speedup that an application can reach after parallelization is limited by what is called *Amdahl's Law*[1] [40], related to the fact that a program contains a serial part that can be executed by only one processing unit.

Let T_s be the execution time of a serial program and T_p its execution time after parallelization when using a number p of processing units. The speedup obtained is then defined as the ratio:

$$S(p) = \frac{T_s}{T_p}. \tag{2.1}$$

[1] Amdahl's law is not the only reason that limits the gain in performance. Parallelization often requires synchronization between concurrent threads. It could also requires to go down in the memory hierarchy and often lead to loss of performance.

Now assume that the parallel program contains a serial part, which could not be parallelized. Calling f the fraction of the program that runs serially, the parallel execution time will then be given by $fT_s + (1-f)T_p = fT_s + \frac{(1-f)T_s}{p}$.

This leads to the following relation for the speedup:

$$S(p, f) = \frac{T_s}{fT_s + \frac{(1-f)T_s}{p}} \tag{2.2}$$

We are interested to the maximum speedup possible when a parallel program contains a fraction f that is serial. Equation 2.2 becomes:

$$S_{max} \equiv \lim_{p\to\infty} S(p, f) = \frac{T_s}{f \cdot T_s} = \frac{1}{f} \tag{2.3}$$

The trend of Amdahl's Law for different values of f and for a variable number of Processing Units is shown in Fig. 2.6. If the parallel part of the program is the 95% of the whole program, the maximum speedup one can obtain is $S_{max} = \frac{1}{1-0.95} = 20$.

Amdahl's Law shows how well an application scales by increasing the number of processing elements when the problem size is fixed, and its study takes the name of *strong scaling*.

The study of strong scaling can give an estimate of the time interval between the submission of a task and the return of the result (*latency-oriented*).

2.2.4 Weak scaling

One of the hypotheses underlying Amdahl's Law is that the size of the problem is fixed. When the problem size increases, the sequential part often does not change. Therefore, as the problem size n increases, so do the opportunities for parallelization. Let $f(n)$ be the sequential code fraction of the program, diminishing as the size of the problem n approaches infinity.

The speedup is given by *Gustafson's Law* [41]:

$$S(n) = f(n) + p[1 - f(n)]. \tag{2.4}$$

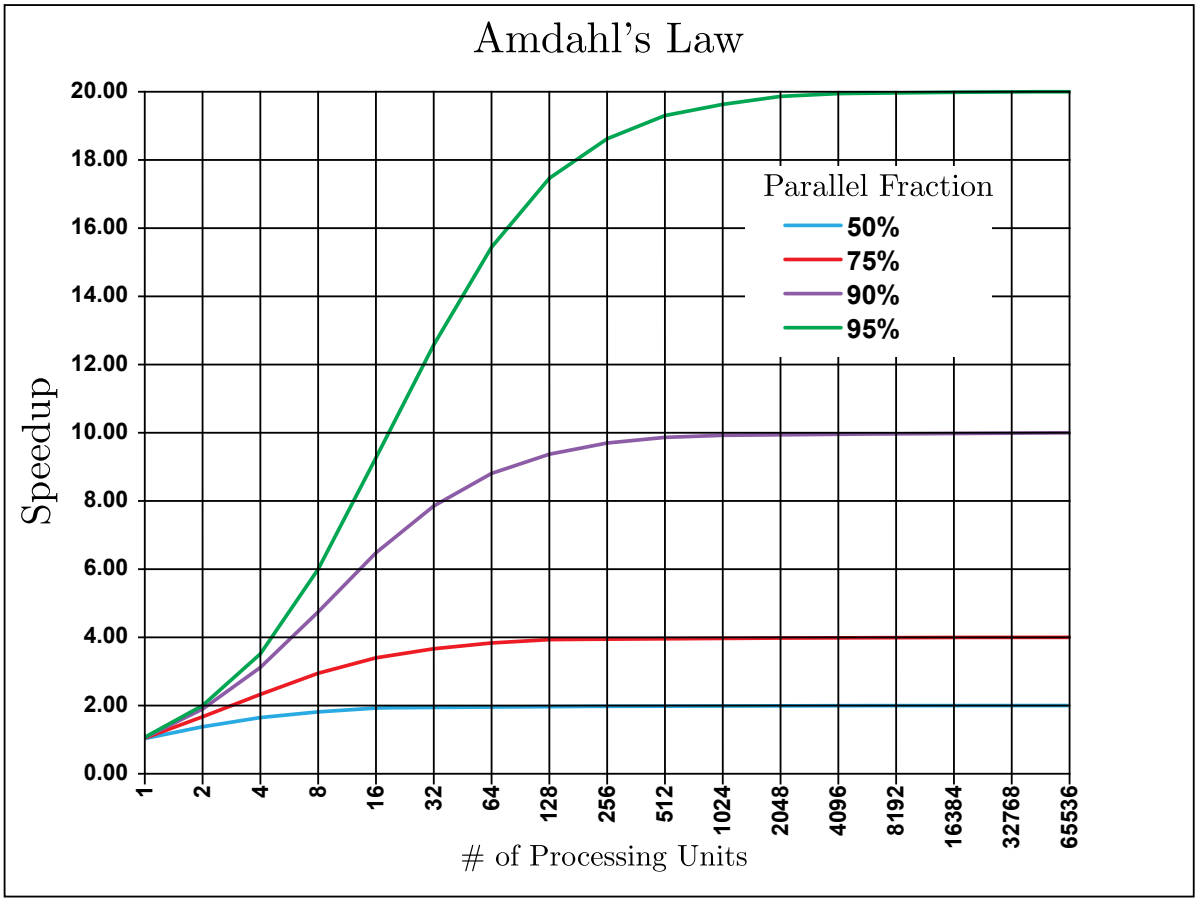


Figure 2.6: Amdahl's law for different fractions of parallel code.

Often, as the size of the problem increases, $f(n)$ decreases to zero. The maximum speedup is then given by:

$$S_{max} \equiv \lim_{n \to \infty} S(n) = p \tag{2.5}$$

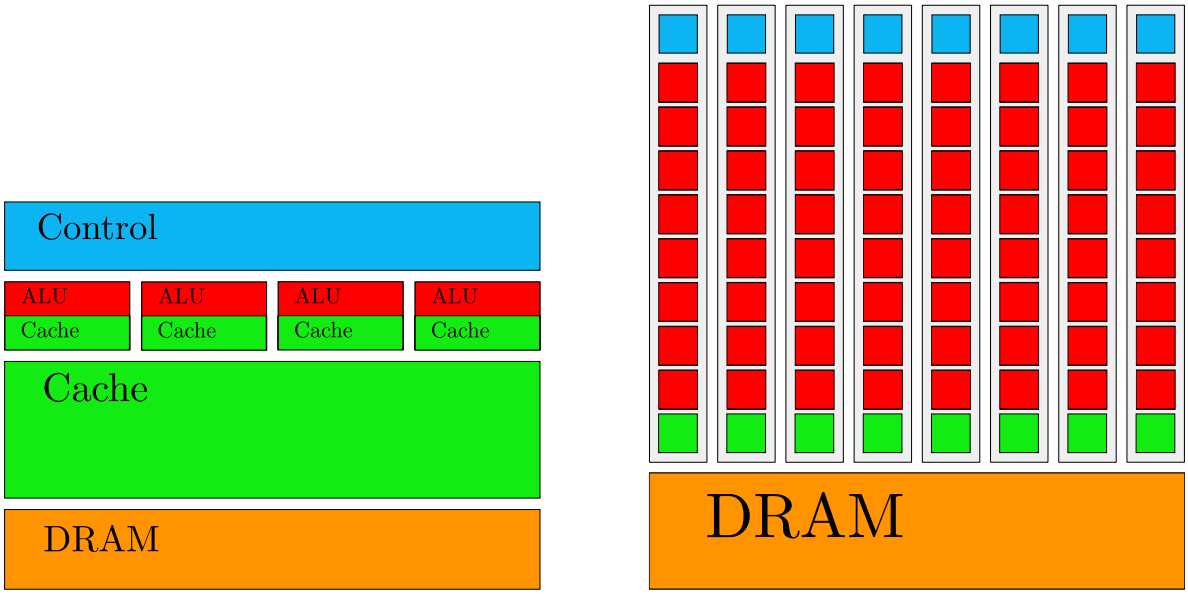
which means that computations involving arbitrarily large data sets can be efficiently parallelized.

Gustafson's Law addresses the shortcomings of Amdahl's Law, which does not fully exploit the computing power that becomes available as the number of processing units increases. Gustafson's Law instead proposes that programmers adjust the size of problems

to use the available equipment to solve problems within a practical fixed time. The scaling with respect to the problem size is called *weak scaling*. If more parallel equipment becomes available, larger problems can be solved in the same amount of time.

Gustafson's Law study is oriented to get the most done in a certain amount of time, and therefore it is said to be *throughput-oriented*.

2.3 Architectures



(a) CPU (b) GPU

Figure 2.7: Simplified architecture of a traditional CPU and GPU. GPUs tend to allocate more transistors to execution units, whereas in CPUs more space resources are dedicated to caches and control units.

CPUs and GPUs are based on very different design approaches.

2.3.1 CPU

CPUs are designed with optimized instruction execution latency.

There are several important design techniques for today's CPUs (Fig. 2.7a):

- **Large cache memories** - They are designed to convert long latency memory accesses to short latency cache accesses. The goal is to keep as many of the data elements as possible in the cache so often when the CPU needs data it will find it in the cache instead of accessing the DRAM, which takes a much longer time.

- There are two control mechanisms:

  - **Branch prediction** - Branches correspond to decisions in the source code (e.g. if-then-else structures). The control unit tries to guess which way a branch will take before this is known. The purpose of the branch prediction is to predict which path will be taken, therefore reducing the latency.

  - **Data Forwarding** - A result of an instruction is immediately forwarded to the next instruction, making CPU able to use it in the clock cycle right after the evaluation.

- **Powerful Arithmetic Logic Units (ALUs)** - More transistors assigned to execution can produce a result in a smaller time.

Use of CPUs in the trigger

For the above reasons, in a low-latency trigger environment, CPUs are best suited for executing instructions on a limited amount of events per time, producing the result of the computation in a small amount of time. For example, CPUs could be employed to do fast processing on each incoming trigger primitive on the fly, without having to wait for some primitives to be accumulated.

2.3.2 GPU

GPUs, on the other hand, are representative instances of the so called *throughput oriented* design (Fig. 2.7b). Their design techniques pursue the goal of maximizing the throughput of instructions that the processor can achieve by executing as many instructions as possible at the same time.

The approaches behind a throughput-oriented design can be summarized as:

- **Small caches** - Caches are used just as staging and consolidation memory to reduce the traffic to the DRAM

- **Simple control** - There is no branch prediction and no data forwarding. Branches are predicated: all the possible code branches are executed before the branch condition is proved. (Fig. 2.9).

- **Many energy efficient ALUs** - Since these modules have fairly long latency for producing results, the best way to achieve performance is by having a massive number of simultaneous execution threads to tolerate latencies.

In recent times GPUs are also used for *General Purpose computing* (GPGPU), and not only for graphics applications. In the High Performance Computing sector there are two major GPU vendors: AMD and NVIDIA. In the following, I will discuss only the NVIDIA solutions because they can be programmed at lower level allowing better control of the hardware through the software. GPU parallelism, on Graphic cards produced by NVIDIA, is exposed for general-purpose computing thanks to an architecture called *Compute Unified Device Architecture* (CUDA).

The CUDA architecture is built around a scalable array of multi-threaded Streaming Multiprocessors (SMs). The design of the Streaming Multiprocessors has been evolving rapidly since the introduction of the first CUDA-capable hardware in 2006.

A SM is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called *SIMT* (Single-Instruction, Multiple-Thread), that is an extension of SIMD architecture. A SM contains thousands of registers that can be partitioned among threads of execution and execution cores for integer and floating-point operations. The reason there are many registers, and the reason the hardware can context switch between threads efficiently, is to maximize throughput of the hardware.

The instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism through simultaneous hardware multi-threading. Unlike the case of CPU cores, instructions are issued in order.with

Figure 2.8: A NVIDIA Kepler Streaming Multiprocessor: 192 single-precision CUDA cores, 64 double-precision (DP) units, 32 special function units (SFU), and 32 load/store units (LD/ST).

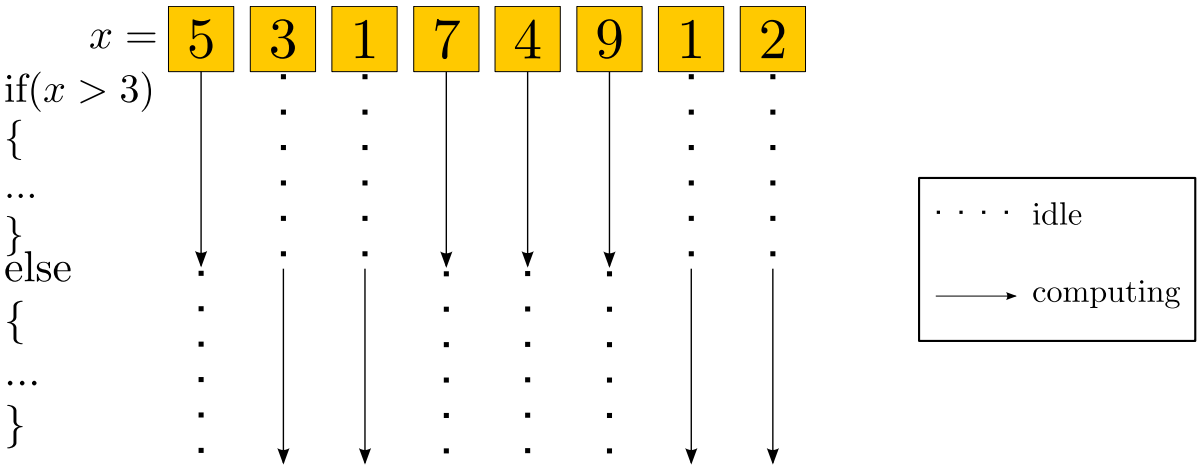


Figure 2.9: Branch predication: different parallel threads executing the same branch check which of the two cases they are going to follow. At first only the threads following the first path execute, whereas the others are in idle state. After the execution of the first case code has completed, the threads following the second path can run the second block of code, while the other threads are not being used.

2.3.3 Host-Device connection

In a heterogeneous system, the CPU and its memory space are referred as *host*, while the GPU and its memory space are referred as *device*.

The host and the device are usually connected through a high speed bus named *PCI Express* (PCIe). A PCIe connection consists of one or more data-transmission lanes, connected serially. Each lane consists of two pairs of wires, one for receiving and one for transmitting. There can be one, four, eight, or sixteen lanes in a single consumer PCIe slot, denoted as ×1, ×4, ×8, or ×16. Each lane is an independent connection between the PCIe controller and the expansion card, and bandwidth scales linearly, so an eight-lane connection will have twice the bandwidth of a four-lane connection. This helps to avoid bottlenecks between the CPU and the graphics card.

A single PCIe gen. 1.1 lane can carry up to 2.5 *Gigatransfers* per second (GT/s) in each direction simultaneously.

For PCIe gen. 2.0, introduced in later years, that increases to 5 GT/s, while now a single PCIe gen. 3.0 lane can carry 8 GT/s.

All PCI Express generations lose some of their theoretical maximum throughput due to

the physical overhead associated with electronic transmissions. Gigatransfers per second include the bits required by the interface overhead.

PCIe gen. 1.1 and 2.0 use *8b/10b encoding*, i.e. each 8 bits of data cost 10 bits to transmit, so 20 percent of the theoretical bandwidth is lost to overhead.

Accounting for overhead overhead, the maximum per-lane data rate of PCIe gen.1.0 is 80% of 2.5 GT/s, i.e. 2 Gbit/s per second, or 250 MB/s. Since the PCIe interface is bidirectional, the bandwidth is 250 MB/s in each direction, per lane.

PCIe gen. 2.0 doubles the per-lane throughput to 5 GT/s, resulting in 500 MB/s of actual data transfer per lane.

PCIe gen. 3.0 achieves twice the speed of PCIe gen. 2.0, despite having a per-lane throughput only 60% higher than a PCIe connection. That's because PCIe 3.0 and above use a more efficient encoding scheme called *128b/130b*, so the overhead is only 1.54%. That means that a single PCIe gen. 3.0 lane, at 8 GT/s, can send 985 MB/s.

Use of GPUs in the trigger

All the above reasons make GPUs well suited to execute not too complex functions on many trigger primitives at the same time. In order to achieve high throughput and fully exploit the GPU architecture, trigger primitives must be accumulated before computation can start.

Furthermore, branch predication makes the time spent in executing an if-then-else structure perfectly predictable.

2.4 Parallel programming techniques

A *thread* is a flow of control that can execute in parallel with others in the same process. Unlike processes, all the threads[2] within a program share the same address space, but not their execution stack.

[2]I will always refer to threads in the same program when speaking of multiple threads.

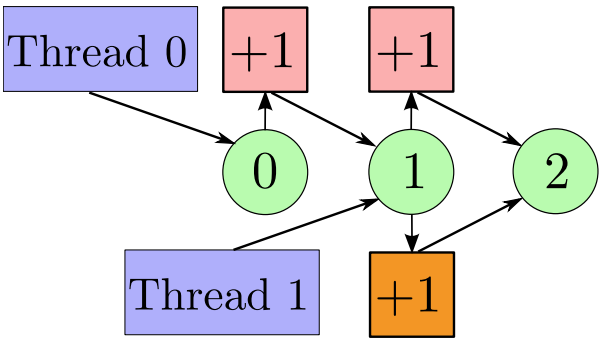


Figure 2.10: An example of race condition: there are two concurrent threads incrementing the same variable a total of three times. If no synchronization happens, *Thread 1* could read the shared value while *Thread 2* is incrementing it, leading to a wrong result.

The common address space allows threads to communicate directly among themselves using shared memory. The fact that two threads can operate on the same memory location can be very useful, but it can also lead to *data hazards* if they are not synchronized. Data hazards usually occur when threads modify data in different points in the instruction pipeline and the order of reading and writing operation matters (data dependence). Data hazards can occur in three circumstances:

- read-after-write (RAW),
- write-after-read (WAR),
- write-after-write (WAW).

Overlooking data hazards can lead to *race conditions*, i.e. a corruption of the shared state (Fig. 2.10). Race conditions can be tricky to debug since the result depends on the timing between concurrent threads and therefore it is unpredictable. When a piece of code is clean of data hazards, it is said to be *thread-safe*. The code that must be executed by a single thread at a time to exclude data hazards is called *critical section*.

The easiest ways to avoid conflicts in such sections and grant access one thread at a time (mutual exclusion, *mutex*) to the critical section consists in using and mixing:

- **Locks** - Synchronization is achieved by associating a lock with each shared section. For a thread to work on a shared section, it must have control over the lock associated with it, it must *hold* the lock. Only one thread can hold a lock at a

time. If a thread cannot acquire the lock it can either wait for it to be free again or leave the critical section.

Spinning occurs when a thread that wants a specific lock continuously checks that lock to see if it is still taken, instead of yielding CPU-time to another thread. Therefore locks can introduce a large overhead and reduce the scalability of the program, and on modern systems they can be avoided by using *Transactional Memory* [42] and other lock-free techniques.

- **Condition variables** - A condition is associated to each thread. More threads associated to the same condition are enqueued into a wait-queue and the lock is released. When the condition becomes true, the first thread in queue is notified and resumed and it acquires the lock. Condition variables are useful to avoid lock contention, that could happen when multiple threads are requesting to acquire the lock in a tiny time interval. The time interval between the notification signal is received and the time at which the thread is awake and ready to acquire the lock is *self-adaptive* and depends on the rate history immediately before the notification signal. It can range from few microseconds to a hundred microseconds.

2.5 CUDA C

CUDA® C is a heterogeneous parallel programming interface that enables an exploitation of data parallelism on a heterogeneous parallel system which uses NVIDIA GPUs.

The CUDA programming model assumes that both the *host* (CPU) and the *device* (GPU) maintain their own separate memory spaces in *DRAM* (Dynamic Random Access Memory), referred to as *host memory* and *device memory*, respectively.

The way to conceptualize the execution of a CUDA program is that it starts with a serial code running on the host, then it enters a section where we have abundant parallelism, where we launch a parallel function called *kernel*. The execution of the SIMT kernel C code will create a large number of parallel threads. These threads are going to be executed on different parts of the data structure, in a SIMT fashion. After the execution of the kernel function control goes back to the serial code (Fig. 2.11).

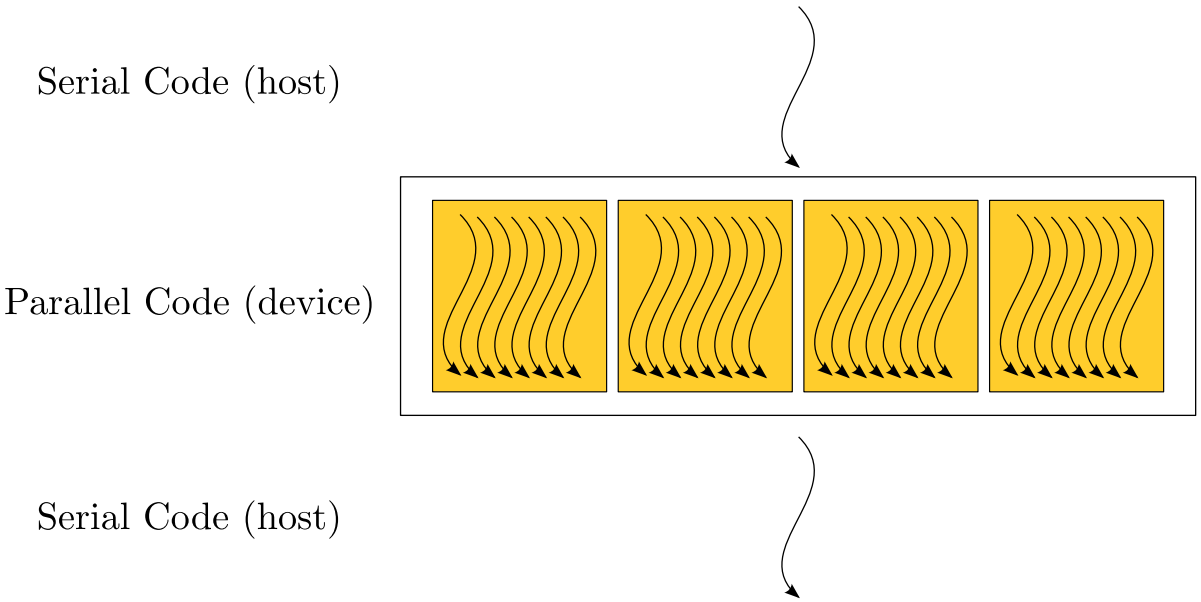


Figure 2.11: Execution of a CUDA parallel program: CUDA parallel threads, which are divided in blocks, are used to offload the host parallel code.

2.5.1 Threads and Memory Hierarchy

The key to the parallel execution of a CUDA program is the generation and the coordination of a large array of parallel threads. A CUDA kernel is executed by a *grid* of threads that run the same kernel code.

CUDA organizes the threads in a two-level hierarchy: groups of threads that can cooperate are called *blocks*, and a group of blocks forms the *grid*. Every block has three block indexes, `x,y,z`, contained in a preinitialized variable called `blockIdx`. Within the block, each thread has three block indexes, `x,y,z`, contained in the variable `threadIdx`.

Another built-in variable called `blockDim` indicates the number of threads in a block: every block in the grid will have the same size and therefore the same `blockDim`. This variable is initialized when the kernel is launched: every thread grid can be launched with a different configuration. The maximum value that `blockDim.x` can have depends on the architecture of the GPU device used.

`threadIdx`, `blockIdx` and `blockDim` allow all the threads to distinguish among themselves and to compute memory addresses and make control decisions.

Table 2.1: Visibility, lifetime, access time and typical size of different kinds of GPU memories.

Memory	**Visibility**	**Lifetime**	**Access time**	**Size**
Registers	Thread	Thread	Very fast	4bytes
Shared mem.	Block	Block	Fast	48kBytes/SM
Global mem.	Grid	Application	up to 150× slower than SM	> 1GB

A simple example of a possible mapping between threads and biunivocal indexes could be: `int index = blockIdx.x * blockDim.x + threadIdx.x`.

When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are numbered and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the free multiprocessors.

2.5.2 Data allocation and movement API functions

The *Application Programming Interfaces* (APIs) are typically sets of functions that the programming environment provides to help the users to perform some functionalities.

Since the CUDA programming model assumes that the host and device memory spaces are separated, in order to to start a computation on some data that is located in the host memory space, an allocation on the device, and a memory transfer from the host to the device global memory are needed.

The CUDA API makes available the following functions for the memory management and memory movement:

- **`cudaMalloc()`** - It is modeled after the C `malloc()` function. It allocates objects in the device global memory. It takes two parameters:
  - the address of the pointer to the allocated object;
  - the size of allocated object in terms of bytes.
- **`cudaMallocHost()`** - It has the same interface as `cudaMalloc()`, but it allocates objects in the host pinned memory to allow streaming (section 2.5.5).

- **`cudaFree()`** - It frees objects from the device global memory. Whenever the data that we allocated in the device memory is not needed anymore, a call to `cudaFree()` function with a pointer to the object will free the memory.
- **`cudaFreeHost()`** - Same as `cudaFree()`, but for objects allocated using `cudaMallocHost()`.
- **`cudaMemcpy()`** - It is a function that is fashioned after che C `memcpy()` function. It performs a memory copy between the host and device. A call to `cudaMemcpy()` is *synchronous*, i.e. it does not return the control to the CPU until the operation has been completed. It requires four parameters:
  - The pointer to the destination location of the copy;
  - The pointer to the source of the copy;
  - The number of bytes to copy;
  - The direction of the transfer (four directions possible: Host → Device, Device → Host, Host → Host, Device → Device);

When allocating host memory using the CUDA API functions, one can choose between pinned and non-pinned memory. Non-pinned memory can be swapped, and this means that the CPU is asked to provide information about the location of the data. Furthermore, the driver needs to access every single page of the non-locked memory, copy it into pinned buffer and pass it to the Direct Memory Access(DMA).

On the other side, pinned memory is physical RAM that cannot be swapped. The driver only needs to know which pages in RAM that have to be transferred and the copy is done directly via Direct Memory Access.

For the above reasons, the usage of pinned memory guarantees faster memory transfer through the PCI Express bus.

2.5.3 CUDA Runtime

As mentioned before, the CUDA parallel execution is based on kernels. A kernel function triggers the parallel execution of a large number of threads that will execute the same

code. These threads are assigned to execution resources on a block-by-block basis. At runtime, the number of blocks assigned to each Streaming Multiprocessor is reduced until resource usage is under limit.

Once a block is assigned to a Streaming Multiprocessor it is divided into units of thread scheduling called *warps*. Each warp is executed in a SIMD fashion: at any given time, all the threads within a warp will be executing the same instruction. When a branch is diverging inside a warp, its predication (Fig.2.9) can lead to a consistent loss in performance.

If a warp is waiting for a long-latency operation to complete, the scheduler won't select it for execution. This mechanism goes under the name of *latency hiding* and this is the main reason why GPUs do not dedicate much chip area to cache memory and branch prediction control mechanisms.

2.5.4 CUDA kernel

Kernel functions that can be specified in a CUDA program are very similar to C functions except for a few subtle differences:

- the kernel function definition has to be preceded by `__global__`
- the kernel launch in the host code has a CUDA specific syntax to specify the configuration parameters: `kernel<<<# of blocks, # of threads>>>(args);`
- the kernel launch is asynchronous;
- a kernel function must return `void`

There are other keywords to specialize function declaration in CUDA, summarized in the table below.

Table 2.2: Specialized function declaration indicates the subject that will launch the function and the device that will execute it.

	Executed on	**Callable by**
`__device__ double deviceFunc()`	device	device
`__global__ void kernelFunc()`	device	host
`__host__ double hostFunc()`	host	host

2.5.5 Streams

CUDA supports concurrent execution of kernels and `cudaMemcpy()` by using *streams*, that are queues of tasks. By having multiple streams, it is possible to execute tasks in different streams in parallel.

A host thread inserts API functions into the stream, that is really a queue. This task queue is read and processed asynchronously by the driver and device. The driver ensures that tasks in the queue are processed in sequence (Fig.2.12). In order to benefit from task parallelism, host memory must be allocated using `cudaMallocHost()`, and copies must be called using `cudaMemcpyAsync()` with the stream identifier as an additional parameter.

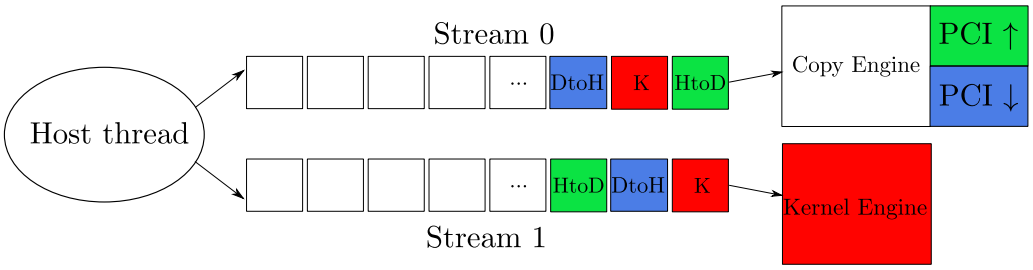


Figure 2.12: CUDA Streams.

2.6 Conclusion

In this chapter I limited the description only to the parallel techniques used.

Many are the paradigms and the architectures that were not covered by the discussion but that have been examined (e.g. OpenCL, OpenMP [44], Threading Building Blocks [45]).

An accurate selection was carried on and led to the choice of NVIDIA devices programmed using CUDA C, and multi-core CPUs programmed using POSIX threads.

Chapter 3

Feasibility study

I am going to describe a pilot project for the use of GPUs in a real-time triggering application in the early trigger stages at the CERN NA62 experiment, and the results of the first field tests together with a data acquisition (DAQ) system. This pilot project within NA62 aims at integrating GPUs into the central L0 trigger processor, and also to use them as fast online processors for computing trigger primitives. The requirements that would make the integration of a software system based on GPUs are given by the RICH detector bandwidth ($\sim$ 800 MB/s) and by the NA62 L0 latency deadline of $\sim$ 1 ms.

Several TDC-equipped sub-detectors with sub-nanosecond time resolution will participate in the first-level NA62 trigger (L0), fully integrated with the data-acquisition system, to reduce the readout rate of all sub-detectors to 1 MHz, using multiplicity information asynchronously computed over time frames of a few ns, both for providing positive trigger requirements and for vetoing conditions. The online use of GPUs would allow the computation of more complex trigger primitives already at this first trigger level.

The architectures of the proposed systems will be described, focusing on measuring the performance and the efficiency of various approaches meant to solve these high energy physics problems.

3.1 Algorithms

In order to develop a framework that can run on off-the-shelf hardware a feasibility study is required. This study is oriented to define a set of rules that the framework has to abide by.

Generic algorithms of increasing complexity were developed to find and remove the bottlenecks that could affect the performance of the final product. All the applications developed use data which is assumed to be already available in the host PC.

3.1.1 Hit Counting

The first test was made on a generic hit counting problem.

Consider a typical detector that measures the energy released by particles in one event in each of its channels (Listing 3.1), with:

- **`NCHANS`** representing the number of channels in the detector,
- **`int id`** identifying the single channel,
- **`double threshold[id]`** being the energy threshold above which channel **`id`** is activated,
- **`Event`** being a structure containing:
  - **`int id[i]`**, an array containing the identifiers of the channels hit by the particles forming the event,
  - **`double adc[i]`**, an array containing the energy released in the channel **`id[i]`**,
  - **`int length`**, a number indicating the length of the arrays **`id`** and **`adc`**.

A channel is activated when one of the particles in the event deposits an amount of energy in a channel that is above the threshold of that channel, i.e.:

```
ok[i] = Event.adc[i] > threshold[Event.id[i]].          (3.1)
```

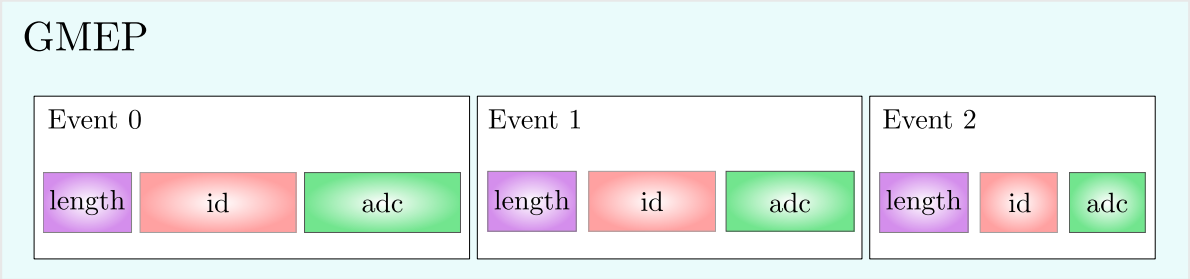


Figure 3.1: Array of structures.

An event can pass the trigger only if the sum of the elements of the array **`ok`**, exceeds a second threshold **`TH`**.

Before an event is generated, how many and which channels are hit and how much energy is released in that channel is unknown. The minimal output from the GPU must be the identification of the event, in case it passed the trigger requirements, or nothing if it did not.

There is no advantage in using a massively parallel architecture like a GPU to determine whether just one event will return a positive trigger response; one rather wants the GPU to compute many of these events concurrently. Many events can be contained in a *GPU Multi Event Packet* (GMEP). A GMEP is a generic container data format used in NA62 for passing data to GPUs.

The size of the GMEP implemented for the hit counting problem is:

- 6 bytes for each contained event;
- 4 bytes for each hit inside each event.

Parallel Friendly data-structures

The first optimization, crucial for the final framework implementation, comes naturally when defining the structure of a GMEP package, and concerns whether this GMEP is an *Array of Structures* (Fig. 3.1) or a *Structure of Arrays* (Fig. 3.2). An event comes as an array of structures, that for their sparse access pattern to the memory, do not represent the best structure the GPU can compute on.

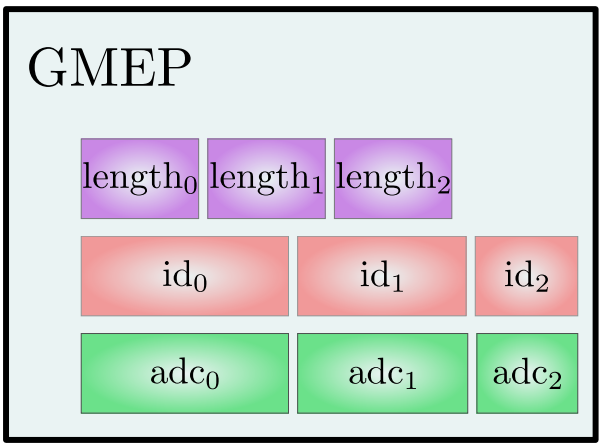


Figure 3.2: Structure of arrays.

These arrays of structures, by mean of a simple transposition, can be transformed in structures of arrays, in order to achieve higher memory bandwidth and throughput. Structure of arrays has the advantage of keeping the values of the same variables in adjacent memory locations.

Parallel reduction

A coarse grained, event-level parallel implementation is straightforward. However event parallelism is not the best solution to exploit all the computational potential of a massively parallel architecture like a GPU.

A reduction kernel was developed. Given an array `a[2N]`, the reduction sum `Sum` of `a` is the sum of all its elements:

$$\texttt{Sum=a[0]+a[1]+...a[2N-1]}. \tag{3.2}$$

This kernel splits the original array `a[2N]` in two sub-arrays `b[N]` and `c[N]`, and evaluates in parallel in shared memory the partial sums `b[i]+c[i]` for all the elements in the sub-arrays. This procedure is repeated until the results array contains only one element that is the total sum of the elements of the original array (Fig. 3.3). If the original array does not contain a number of elements that is a power of 2, `c[i]` will be outside the limits of the array. In this case the thread will copy in the result the value of `b[i]` without performing the sum.

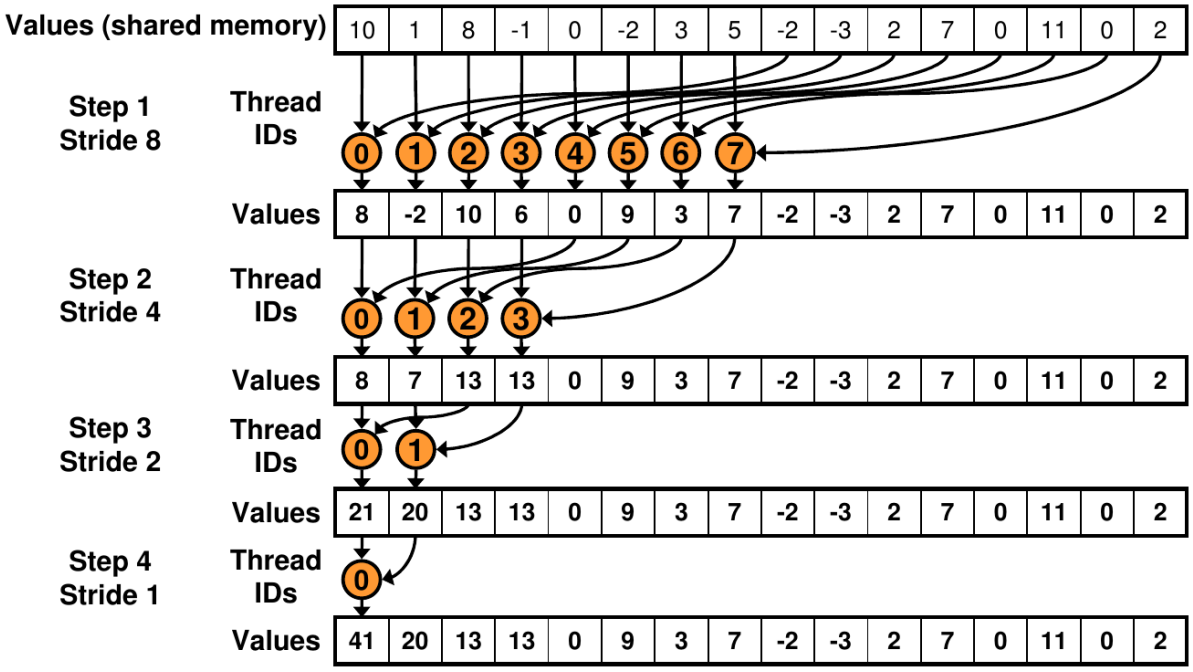


Figure 3.3: Reduction kernel using sequential addressing.

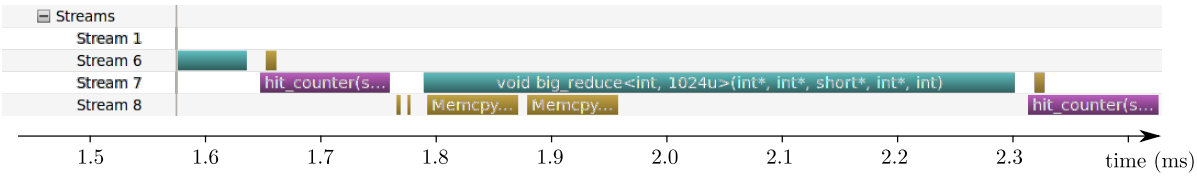


Figure 3.4: Profiling a CUDA application can give useful hints on what it is actually needed to speed-up performance. It gives information about the time spent, the memory consumption and memory bandwidth usage in each operation.

The sequential cost is $O(N)$. $O(N/logN)$ threads were spawned, each performing $O(logN)$ sequential operations to avoid to have too many threads idling. Since the time complexity is $O(logN)$, all the threads will cooperate for $O(logN)$ steps.

Every known implementation of a reduction kernel in CUDA [49] assumes that the function is `__global__` (see section 2.5.4) and that the result will be next used by the host. Since the NVIDIA CUDA profiler (Fig. 3.4) shows that the reduction kernel can take $\approx 20\mu$s before starting its computation, and in a low latency environment this represent a good fraction of the maximum latency (of 1 ms for NA62 Level 0), it is convenient to launch one single kernel, `big_reduce()`, that spawns all the single event reductions as `__device__` functions.

Listing 3.1: Simplified code for the hit-counting algorithm.

```
__global__
void hit_counter(const GMEP* GMEP_GPU, int* ok, const int total_N_hit)
{
   int index = threadIdx.x + blockDim.x*blockIdx.x;
   if(index < total_N_hit)
      GMEP_GPU->ok[index] = GMEP_GPU->adc[index] > TH[GMEP_GPU->id[index]];

}

template <class T, unsigned int blockSize>
__global__
void big_reduce(const T* idata_gpu, T* odata_gpu, const short int* length, const int* offset, const int nEvents)
{
   int eventIdx = blockIdx.x;
   if(eventIdx < nEvents)
      reduce_kernel<T,blockSize>(&idata_gpu[offset[eventIdx]], &odata_gpu[eventIdx], length[eventIdx]);
}


int main()
{
   AoStoSoA(Event, &GMEP[i]);
   ...
   cudaMemcpyAsync(&GMEP_gpu[i],&GMEP_host[i], gmep_size, cudaMemcpyHostToDevice, stream[0]);
   hit_counter<<<blocksPerGrid, threadsPerBlock, 0, stream[0]>>>(&GMEP_gpu[i], ok, total_lenght);
   big_reduce<<<dimGrid, dimBlock, smemSize, stream >>>(ok, Result_GPU, length, offset, nEvents);
   cudaMemcpyAsync(&Result_host[i], &Result_GPU[i], Result_size, cudaMemcpyDeviceToHost, stream[0]);
   ...

}
```

Data Flow

Since the size of the events is variable and only known at run time, one could be tempted to dynamically allocate memory on the fly for each GMEP. Again, the profiler comes to aid, showing that a call to the CUDA memory allocation API function `cudaMalloc()` can cost hundreds of μs. This issue can be solved by allocating a circular queue during initialization, whose elements are as large as the maximum possible size of a GMEP, and reuse the same memory locations in a round-robin fashion. This solution is convenient when the time spent in I/O operations is comparable with the time spent in computation (or longer), but this is not the case for a low-latency trigger software.

The data flow of the application is then the following (Fig. 3.1):

1. An array of events is converted to a structure of arrays, keeping the value of the length of each event in an array `length` (line 22).

2. The structure is copied inside the GPU global memory (line 24).

3. The hit-counting kernel is launched on the whole GMEP. It evaluates for each hit, if the energy released is above the channel threshold, and puts the result (1 or 0) in the corresponding element of a utility array, `ok[i]` (line 25).

4. The `big_reduce()` templated function spawns `reduce_kernel()` functions as many times as the number of events in the GMEP. These reduction functions count the number of 1s in the `ok` array, for each event in parallel (line 26).

5. If the number of 1s for an event exceeds the threshold `TH`, the event identifier is saved and returned to the host as output (line 27).

Streams

The use of streams, described in section 2.5.5, can increase the maximum throughput of a CUDA application, by hiding the latency. This is because they allow independent kernels and memory transfers to overlap in time creating a pipeline (Fig. 3.4), making use of separate engines. While on a code running offline one might create independent input and output data and manually interlace the function calls that run using different streams, the use of streams is not straightforward in an online application. One example is given by the code in listing 3.2:

- line 4: the memory transfer of the results of the kernel on the i-th GMEP takes place. The evaluation of the kernel on this GMEP has already finished here.

- line 5-6: the kernels are launched on the (i+1)-th GMEP.

- line 7: the events contained in the (i+2)-th GMEP are sent to the GPU for evaluation.

Listing 3.2: CUDA API functions calls manually interlaced using CUDA streams.

```
int main()
{
   ...
   cudaMemcpyAsync(&Result_host[i], &Result_GPU[i], Result_size, ↩
       cudaMemcpyDeviceToHost, stream[i]);
   hit_counter<<<blocksPerGrid, threadsPerBlock, 0, stream[i+1]>>>(&GMEP_gpu[i+1], &ok↩
       [i+1], total_lenght[i+1]);
   big_reduce<<<dimGrid, dimBlock, smemSize, stream[i+1] >>>(&ok[i+1], &Result_GPU[i↩
       +1], &length[i+1], &offset[i+1], nEvents[i+1]);
   cudaMemcpyAsync(&GMEP_gpu[i+2],&GMEP_host[i+2], gmep_size, cudaMemcpyHostToDevice, ↩
       stream[i+2]);
   ...

}
```

Tests

Tests on the above hit counting problem have been carried out using the following hardware:

- Host CPU: Intel Xeon E5630
- Host RAM: 12GB DDR2
- GPU device: NVIDIA Tesla Fermi C2050
- Communication bus: PCI Express Gen. 2 (Theoretical peak bandwidth: 8GB/s)

The NVIDIA Tesla C2050 card features clock speeds of 1.15 GHz core clock and 1.5 GHz on the 3GB of GDDR5 memory that runs on a 384-bit memory interface. Each SM, in turn, contains 32 CUDA processing cores for a total number of CUDA cores of 448.

The time measurements have been carried out using the dedicated CUDA API functions. Since on a CPU one does not separate the time interval spent in computation from the time interval for fetching the data from the caches or main memory (unless it is explicitly specified) the measurements of GPU performance always include computation and memory transfer in both directions.

The number of hits inside each event was generated at run time according to a Poisson distribution with mean 200. Figure 3.5 shows the data processing throughput for a varying number of events inside a GMEP. One can understand the behavior shown in figure 3.5 from the architecture of a GPU. As discussed in section 2.3, a GPU is made of

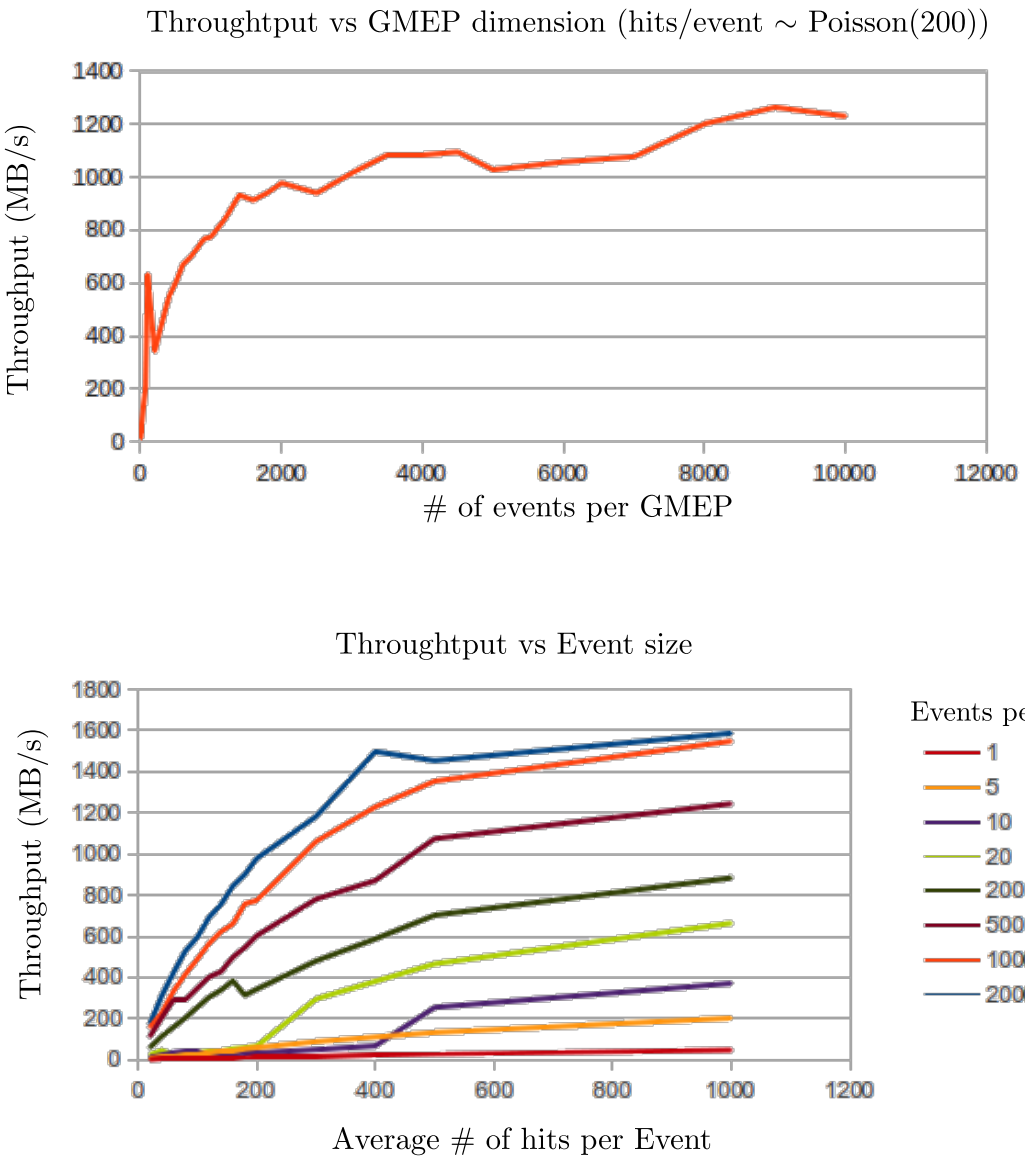


Figure 3.5: Above: Measurements of the throughput for a varying number of events inside one GMEP. Below: Measurements of the throughput for different GMEP sizes, with a varying number of hits inside each event. Throughput is the amount of data fully processed in the unit time.

Streaming Multiprocessors that run scheduled warps on demand. When a kernel does not need many warps to run, the number of active Streaming Multiprocessors that work concurrently is small. When the number of events inside a GMEP or the average size of the events are increased, the demand for Streaming Multiprocessors grows as well.

Consequently, a growing number of active Streaming Multiprocessors means that more operations are executed on a larger dataset in the same time interval (weak scaling).

In view of these facts, the plots in figure 3.5 highlights four crucial aspects:

- For relatively small dataset dimensions, it takes an almost constant time to process a varying number of events. What varies is the time spent in memory transfers.
- When all the available Streaming Multiprocessors are activated (for longer datasets) there is not much space for increasing the throughput, and a saturation plateau shows up.
- There is a step behavior due to the fact that the GPU has a discrete number of active SMs.
- The plateau has a throughput of the same order of magnitude of the PCI Express gen.2 bandwidth. This fact proves that this application is I/O limited and consequently can benefit a lot from the upgrade of the communication bus to the third generation PCI Express.

For the definition of an *operating point*, i.e. the ideal size of a GMEP, one should avoid to choose a small size to prevent unstable and low throughput: the choice of the operating point for a trigger application is not unique, since the throughput is not the only requirement. Furthermore, the main requirement for using a GPU in the NA62 L0 trigger is that latency must be limited and stable in time. Figures 3.6 and 3.7 show that, although a larger number of events guarantees a better performance and lower overhead, there is also an upper limit to the acceptable number of events due to by the maximum latency threshold.

The fact that both throughput and latency can possibly be within their limits is good news but it is not sufficient to say that GPUs are ready to work in a real-time environment: the stability of the latency is an extremely important factor to consider for this kind of application. GPUs don't run an operative system that can delay the execution of

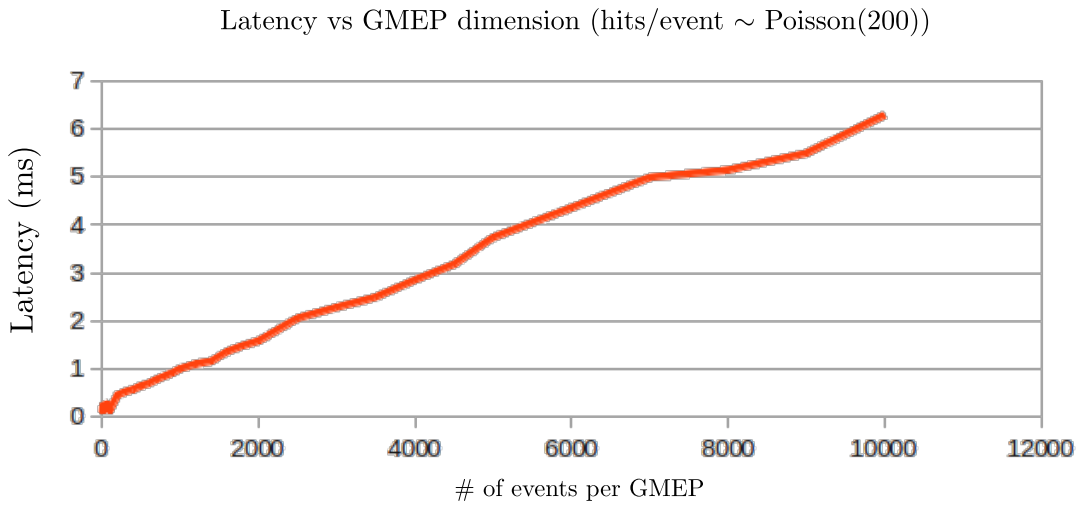


Figure 3.6: Latency trend with respect to a varying size of the GMEP.

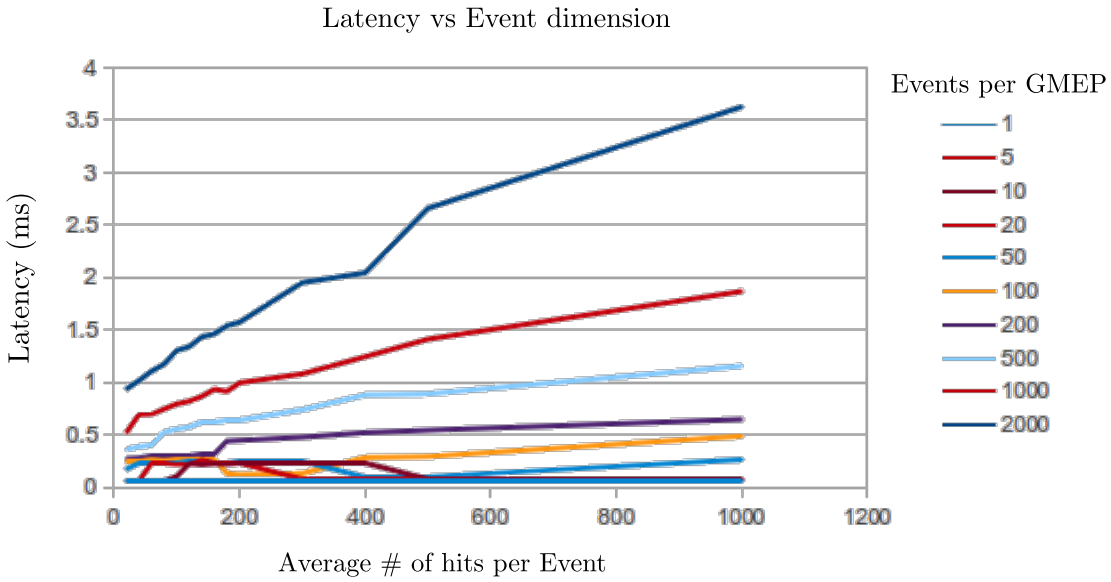


Figure 3.7: Latency trend with respect to a varying average size of the events for a varying number of events inside each GMEP.

operations, but there can be other factors (e.g. temperature) that influence the latency stability. I ran the code in a loop for more than six millions times (Fig. 3.8) and the test showed that latency is actually quite stable with a maximum value of 649 μs.

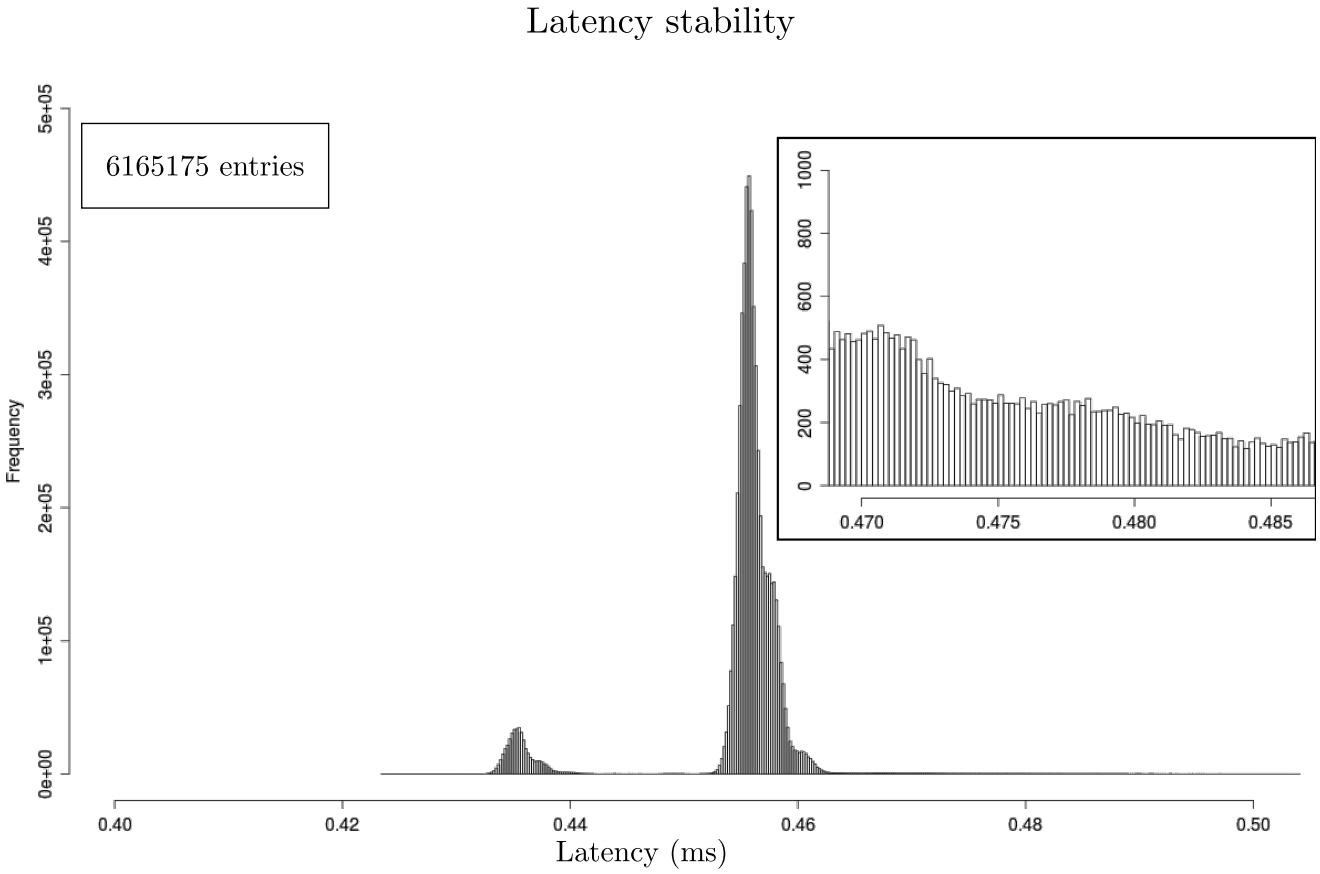


Figure 3.8: Latency stability for the hit counting problem: the time between a GMEP is being transferred to the GPU global memory and the result is returned to the host memory. The GMEP considered contained 1000 events with 100 hits on average.

3.1.2 Lookup Table

A second possible application of a GPU in a trigger system is given by its use for evaluating conditions on input data based on the use of a *lookup table* (LUT). A LUT is an initialized array or matrix in memory that contains precalculated information representing e.g. the output trigger response for a given input data pattern.

If the number of possible outcomes is limited and the direct calculation is too expensive, it can be convenient to store all the possible results in a LUT and, by means of a simple indexing operation, one can retrieve the information needed in constant time.

The advantage of using a GPU for storing and accessing a LUT lies in the possibility to parallelize its access and to handle multiple queries at the same time.

LUT location

There are several places where one could choose to store a LUT on a GPU:

- Global memory: it provides slow but cached access to memory and is very large, in the GB range;
- Texture memory: originally conceived for use in graphics rendering pipelines. It is large, in the GB range, and cached on chip, so in some situations it will provide higher effective bandwidth by reducing memory requests to off-chip DRAM.

  Texture caches are specifically designed for graphics applications where memory access patterns exhibit a great deal of spatial locality (Fig. 3.9).
- Constant memory: like the global memory, reading from constant memory can be slow if data is not cached. Its size is in the order of ten kilobytes.
- Shared memory: if the LUT is not too big (less than 48kB) it can fit into the shared memory, that is on chip and for this reason has a short access time.

Time resolution in CHOD detector

In order to measure the latency and its stability, I found a possible application of the LUT approach in the correction of the time measured by Photo-multipliers (PMTs) at one end of the slabs that constitute the NA48 CHOD detector used by NA62 in its initial phase (Fig. 1.25), to compensate for the light travel time depending on the hit position along the slab. This application can potentially increase the detector time resolution and help in solving ambiguity problems when two tracks hit the detector at the same time, generating four possible matching candidates between the two hodoscope planes.

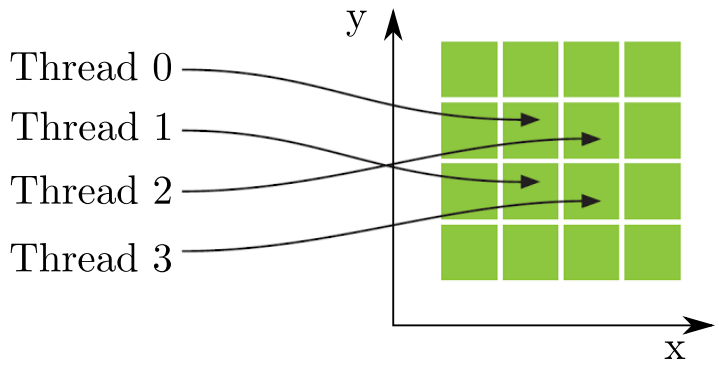


Figure 3.9: Spatial locality in parallel texture memory fetches: when thread 0 accesses a memory location, the surrounding memory location are cached, hence shortening the other threads access time.

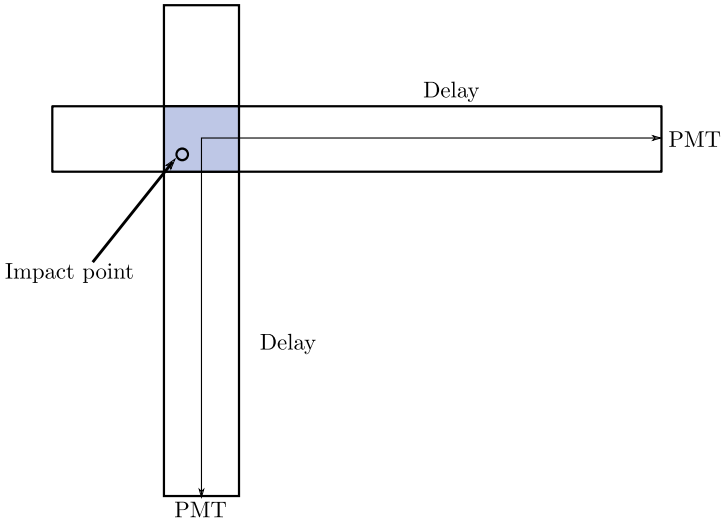


Figure 3.10: Time correction in the NA48 CHOD detector.

Since one plane of the CHOD consists of 128 plastic scintillator slabs, of 64 horizontal and 64 vertical scintillators I stored a 64×64 large LUT inside the global, shared, constant and texture memories. This LUT contained, for each cell, the precalculated values (2 double precision floating point variables) of the time corrections to be applied to the hit time measured at the PMTs. For each type of memory, I implemented a kernel in charge of correcting the hit times, which were generated by making use of a Monte Carlo simulation (Listing 3.3).

Listing 3.3: Simplified kernels to fetch data from LUT stored in different memories.

```
  __constant__ int dA[TEXTURE_SIZE];
__global__ void shiftConstant(GMEP* element)
{
  int xid = blockIdx.x * blockDim.x + threadIdx.x;

  __syncthreads();
  if(xid < element->length)
      element->timestamp[xid] += dA[element->xHit[xid] + 64*element->yHit[xid]];
}

...

__global__ void shiftGlobal(GMEP* element, int* shifts)
{
  int xid = blockIdx.x * blockDim.x + threadIdx.x;
  if(xid < element->length)
      element->timestamp[xid] += shifts[element->xHit[xid] + 64*element->yHit[xid]];
}

...

__global__ void shiftShared(GMEP* element, int* shifts)
{
  int xid = blockIdx.x * blockDim.x + threadIdx.x;
  extern __shared__ int  shared_shifts[];
  shared_shifts[threadIdx.x] = shifts[threadIdx.x];
  shared_shifts[1024+threadIdx.x] = shifts[1024+threadIdx.x];
  shared_shifts[2048+threadIdx.x] = shifts[2048+threadIdx.x];
  shared_shifts[3072+threadIdx.x] = shifts[3072+threadIdx.x];
  __syncthreads();
  if(xid < element->length)
      element->timestamp[xid] += shared_shifts[element->xHit[xid] + 64*element->yHit[↩
          xid]];
}

...

texture<float, 1, cudaReadModeElementType> texRef;
__global__ void shiftTexture(GMEP* element)
{
  int xid = blockIdx.x * blockDim.x + threadIdx.x;
  if(xid < element->length)
      element->timestamp[xid] += 1+tex1D(texRef, element->xHit[xid] + 64*element->yHit↩
          [xid]);
}
```

Tests

Tests were carried out using the following hardware:

- Host CPU: Intel Xeon E5630
- Host RAM: 12GB DDR2
- GPU device: NVIDIA Tesla Fermi C2050
- Communication bus: PCI Express Gen. 2 (Theoretical peak bandwidth: 8GB/s)

The latency stability test handled ten millions hits, and showed that the latency of the texture memory is far more stable than the others (Fig. 3.11). The fact that the texture memory latency shows two close peaks is due to the fact that this memory is cached on chip, so threads of the same warp that read texture addresses that are close together will achieve best performance (Fig. 3.9).

Furthermore, Figure 3.12 proves that the LUT problem is completely bound by the I/O bandwidth, since the saturation value of the throughput close to the maximum bus bandwidth.

3.2 Conclusion

In this chapter I described the feasibility study carried so as to prove that the latency and throughput ranges in which optimized GPU algorithms work. This performance assessment showed that, if properly optimized, a GPU application can work as a real-time trigger with latencies smaller than ~ 1 ms and throughput higher than ~ 800 MB/s.

This study shows that it is possible to increment the complexity of the algorithms, thus developing a pattern recognition algorithm, while staying within the latency and throughput requirements of the NA62 RICH detector.

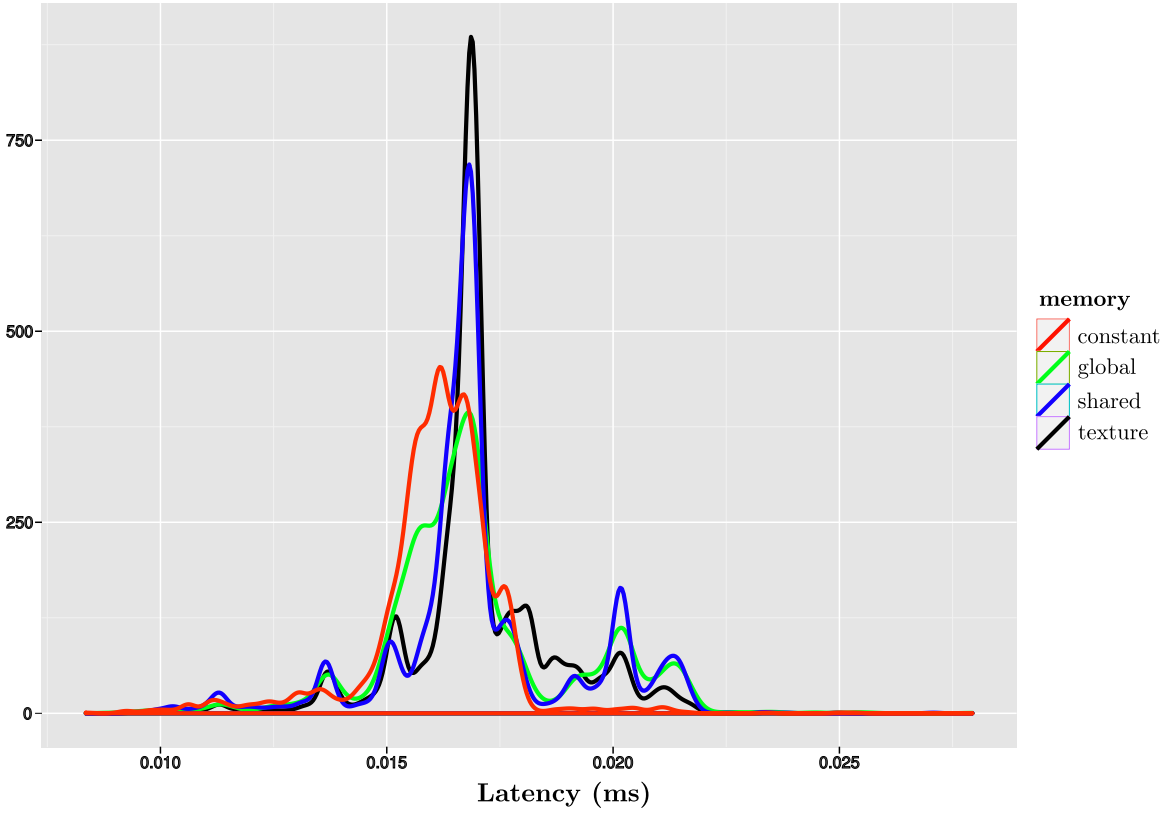


Figure 3.11: Distributions of the latency stabilities of LUT fetches from different memories over 10 millions trials.

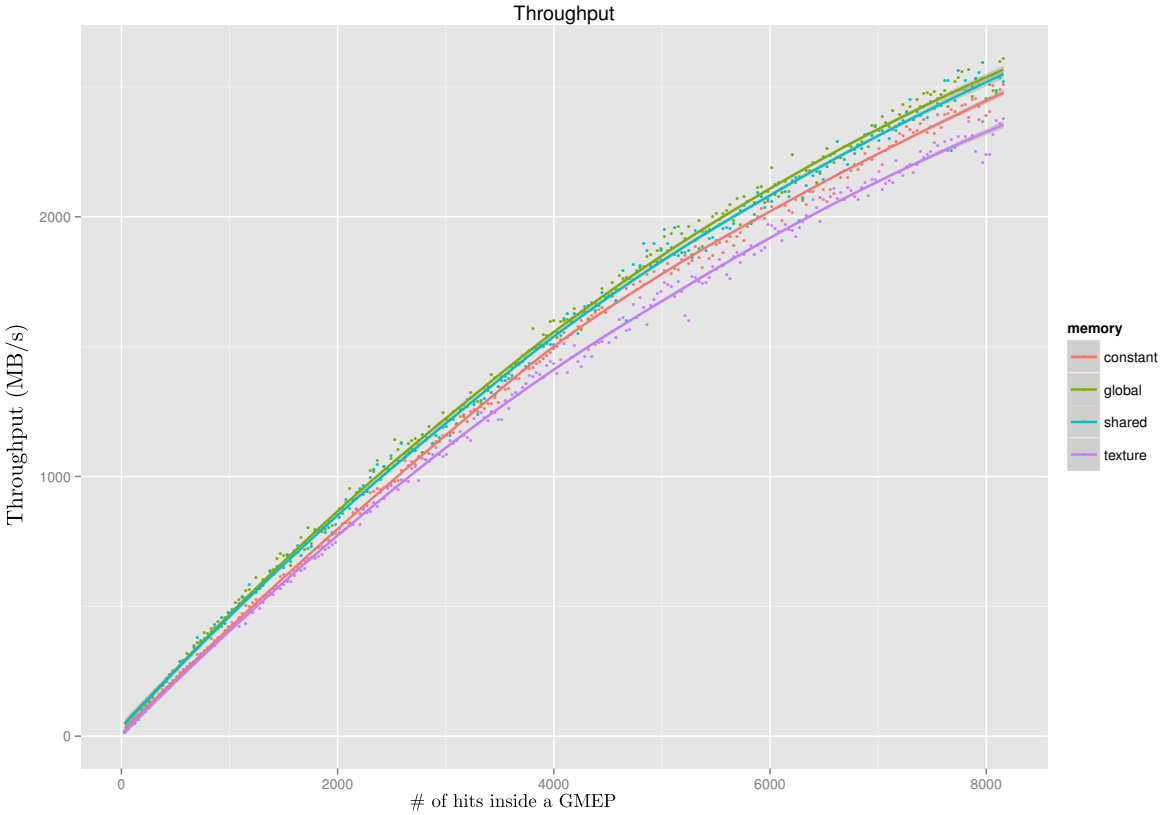


Figure 3.12: Throughput trend of the time correction LUT kernel when changing the number of tracks inside one GMEP, for different GPU memories.

Chapter 4

Ring reconstruction using GPUs

As a first real application of the use of GPUs in the NA62 trigger system, I studied the possibility to reconstruct rings produced by Čerenkov light in the RICH detector [50]. The center and the radius of the Čerenkov rings in the detector are related to the angle and the velocity of the charged particle through the RICH detector. These information could be used at the trigger level to increase the purity and the rejection power for events of interest and backgrounds.

The ring reconstruction could be useful both at L0 and L1 trigger levels. In both cases, because of the high rates (of 10 and 1 MHz respectively), the computing power required is significant. GPUs can offer a solution to the problem. The use of GPUs at the software L1 is straightforward trigger level: they can be used to offload the computation and run more complex algorithms not suited for hard real-time. On the other hand the L0 is a small latency synchronous trigger level, and the possibility to use GPUs for complex computations needs to be verified.

The requirements for the algorithm to be developed are:

- **latency**: it has to be strictly and stably below 1 ms;
- **throughput**: the GPU has to cope with an input bandwidth of $\approx$ 800 MB/s (Table 1.3). If this requirement cannot be satisfied without exceeding the latency, more GPUs could be added in parallel;
- **seedless**: the algorithm won't be given any other information about the ring positions, but the hit coordinates;

- **multi-ring**: it should allow identification of events with more than one ring in the detector (e.g. $K^+ \to \pi^-\mu^+\mu^+$)

- **robust**: it has to be insensitive to the precence of noise and partially-contained rings.

Since no algorithm satisfying all these requirements exists in literature, I started developing my own.

The developed algorithms assume that the time matching of the hits to form an event has already been done by the front end electronics.

Given that the time spent in computation has to be stable with respect to all the possible input datasets, I focused my study on algebraic methods, starting with single-ring event fitting.

4.1 Crawford Method

In order to test feasibility and performance, as a starting point I have implemented the algorithm described by Crawford [51], for single ring fitting in a sparse matrix (in our case 1000 points, centered on the PMs in each RICH spot) with N firing PMTs ("hits"). In our case N is 20 on average (Fig. 1.23).

Consider a circle of radius R, centered in (x_0, y_0) and a list of points (x_i, y_i) belonging to the circle. Its equation is given by:

$$(x - x_0)^2 + (y - y_0)^2 = R^2.$$

Applying the least squares method is not straightforward because involves the minimization with respect to x_0, y_0, R of a function like:

$$\sum \{\sqrt{(x_i - x_0)^2 + (y_i - y_0)^2} - R\}^2,$$

where the sum is over the N points to be fitted. If we consider instead the function:

$$S = \sum \{(x_i - x_0)^2 + (y_i - y_0)^2 - R^2\}^2,$$

differentiating it with respect to x_0, y_0, R the following equations are obtained:

$$\frac{\partial S}{\partial x_0} = 4\sum(x_0 - x_i)\{(x_i - x_0)^2 + (y_i - y_0)^2 - R^2\} = 0, \tag{4.1}$$

$$\frac{\partial S}{\partial y_0} = 4\sum(y_0 - y_i)\{(x_i - x_0)^2 + (y_i - y_0)^2 - R^2\} = 0, \tag{4.2}$$

$$\frac{\partial S}{\partial R} = -4\sum R\{(x_i - x_0)^2 + (y_i - y_0)^2 - R^2\} = 0. \tag{4.3}$$

Equation (4.3) shows that at the minimum:

$$\sum\{(x_i - x_0)^2 + (y_i - y_0)^2 - R^2\} = 0,$$

Therefore equations 4.1 and 4.2 are reduced to quadratic equations in x_0, y_0 and R. Furthermore, equation 4.3 can be re-written:

$$\begin{aligned} {x_0}^2 + {y_0}^2 - R^2 = \frac{1}{N}\{2x_0\sum x_i + 2y_0\sum y_i - \\ -\sum x_i^2 - \sum y_i^2\}. \end{aligned} \tag{4.4}$$

This relation enables us to eliminate R from equations 4.1 and 4.2.

Since x_0, y_0 and R occur in these equations only in the combination $x_0^2 + y_0^2 - R^2$, the elimination reduces the equations from quadratic to linear equations.

This substitutions yields:

$$\begin{aligned} x_0\{\sum x_i^2 - \frac{(\sum x_i)^2}{N}\} + y_0\{\sum x_i y_i - \\ -\frac{\sum x_i \sum y_i}{N}\} = \frac{1}{2}\{\sum x_i^3 + \sum x_i y_i^2 \\ -\sum x_i \frac{\sum x_i^2 + \sum y_i^2}{N}\}, \end{aligned} \tag{4.5}$$

and

$$x_0\{\sum x_i y_i^2 - \frac{\sum x_i \sum y_i}{N}\} + y_0\{\sum y_i^2 - \\ -\frac{\sum y_i^2}{N}\} = \frac{1}{2}\{\sum x_i^2 y_i + \sum y_i^3 - \\ -\sum y_i \frac{\sum x_i^2 + \sum y_i^2}{N}\}. \tag{4.6}$$

Equations (4.5) and (4.6) can be solved for x_0 and y_0, and eventually R can be determined by using equation (4.4).

If required, the circle found can be used as the start of an iterative search using a minimization function to cope with the actual errors.

4.1.1 Algorithm description

The evaluation of the coordinates of the center and the radius of the circle for a single event proceeds as follows:

- the average coordinates are computed:

$$x_m = \frac{\sum x_i}{N}, y_m = \frac{\sum y_i}{N};$$

- the differences between each hit photomultiplier coordinates and such averages are evaluated:

$$u_i = x_i - x_m, \quad and \quad v_i = y_i - y_m \tag{4.7}$$

- for each photomultiplier hit the algorithm computes: u_i^2, v_i^2, $u_i v_i$, u_i^3, v_i^3, $u_i^2 v_i$, $u_i v_i^2$;
- the sums $\sum u_i^2$, $\sum v_i^2$, $\sum u_i v_i$, $\sum u_i^3$, $\sum v_i^3$, $\sum u_i^2 v_i$, $\sum u_i v_i^2$ are evaluated;
- the coordinates of the center and the radius of the circle can be evaluated using the shifted coordinates (eq. 4.7) in equations (4.4), (4.5) and (4.6).

Parallelization Strategy

The parallelization occurs on three levels:

- instruction-level parallelism: GMEPs are processed on different streams, that form a pipeline allowing efficient latency hiding;
- event-level parallelism: many events are processed in parallel;
- inside-the-event parallelism: in each event the reductions are performed in parallel as discussed in Section 3.1.1.

4.1.2 Implementation

An event comes as an array of structures (x_i, y_i) $(i = 0, ..., N)$, where x_i and y_i are the coordinates of a photomultiplier that fired. By means of a simple transposition, these become a structure of arrays $(x_0, ..., x_{N-1}), (y_0, ..., y_{N-1})$ in order to achieve higher bandwidth and throughput (see section 3.1.1). The arrays `length[]` and `offset[]` keep track of the length and the position of an event in the structure of arrays.

To limit the overhead related to the data transfers from and to the GPU, events are not sent on the fly as they are produced (or arrive from the TEL62 in our trigger case, see later), even if this is not optimal for the latency: a GMEP of events is sent to the GPU memory only when a minimum size has been reached.

Tests

Tests have been carried out on two machines.

Machine 1:

- Host CPU: Intel Xeon E5630
- Host RAM: 12GB DDR2
- GPU device: NVIDIA Tesla Fermi C2050
- Communication bus: PCI Express Gen. 2 (Theoretical peak bandwidth: 8GB/s)

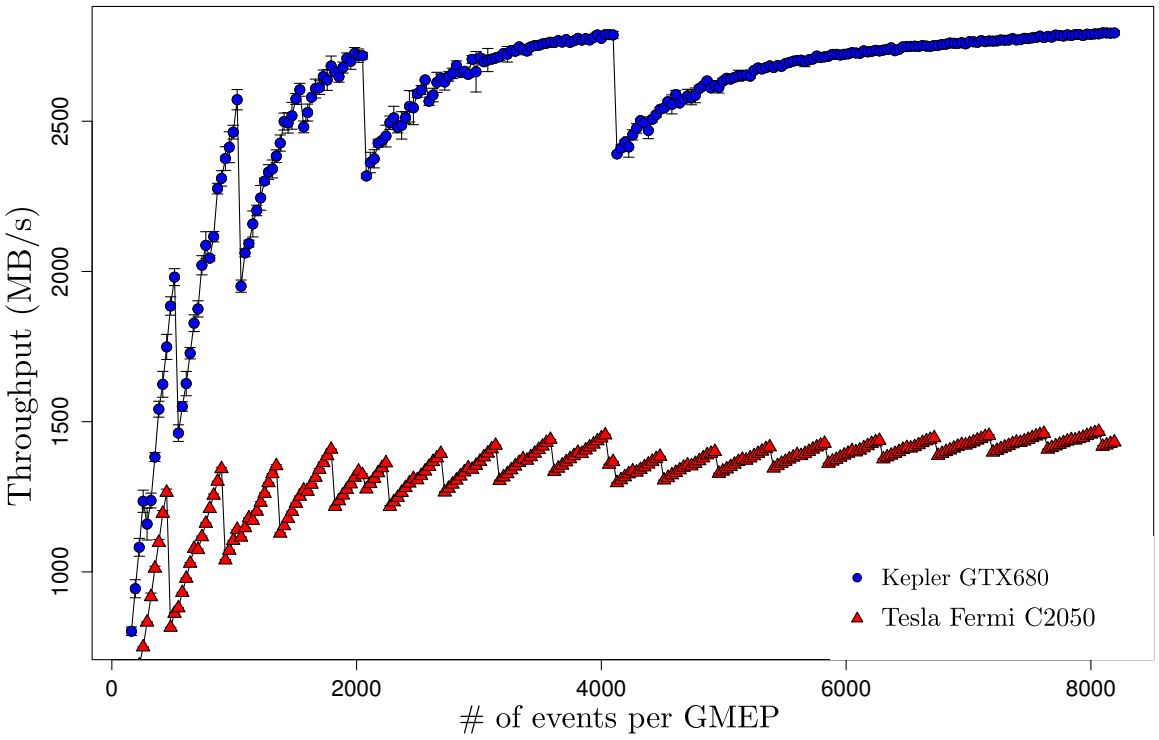


Figure 4.1: Throughput of the Crawford ring-fitting kernel for a variable number of events inside each GMEP for two GPU devices.

and Machine 2:

- Host CPU: Intel Ivy Bridge core i7 3770
- Host RAM: 4x4GB Corsair XMS DDR3 2000MHz dual channel
- GPU device: NVIDIA Kepler GTX 680[1]
- PCI Express Gen. 3

Machine 1 ran 64-bit Scientific Linux CERN 6.2 (based on Red Hat Enterprise Linux 6.2) while Machine 2 ran 64-bit Fedora 17. This choice was obligatory due to the fact that support for NVIDIA Kepler GTX 680 and PCI Express Gen. 3 bus comes with

[1]Each SM is now a "next-generation Streaming Multiprocessor", which Nvidia abbreviates as SMX; each SMX contains 192 CUDA cores, for a total of 1,536 cores in the entire Kepler GPU.

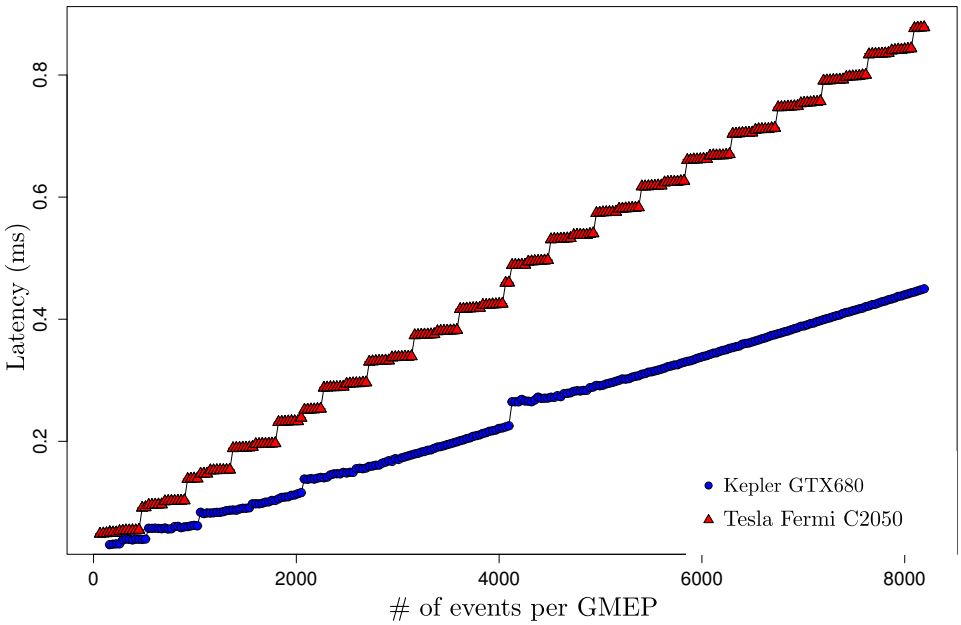


Figure 4.2: Latency for a variable number of events inside each GMEP for two GPU devices.

kernel versions 3.3 and later and Fedora 17 was the only compatible distribution at the time of the test.

On both machines NVIDIA CUDA Toolkit 4.2 and NVIDIA drivers v295.41 were installed.

In all the tests, the time spent copying the events from the host memory to the device memory is included in the latency figures, as well as the time spent copying the structure of the final results back to the host memory.

Figure 4.1 shows the GPU behavior when processing a variable number of events. For a low number of events, the latency is almost constant with respect to the event packing since an increasing number of SMs is activated. Increasing the event packing, two concurrent types of behavior can be distinguished:

- an oscillatory one, due to the discrete nature of a GPU;
- a plateau, since when all the SMs are busy the GPU is saturated.

The almost exact factor 2×, both in throughput (Figure 4.1) and in latency (Figure 4.2), between the GTX680 and the Tesla C2050 is due to the fact that this application is I/O bound, and the bandwidth of the PCI Express v3 bus is twice the PCI Express v2 one.

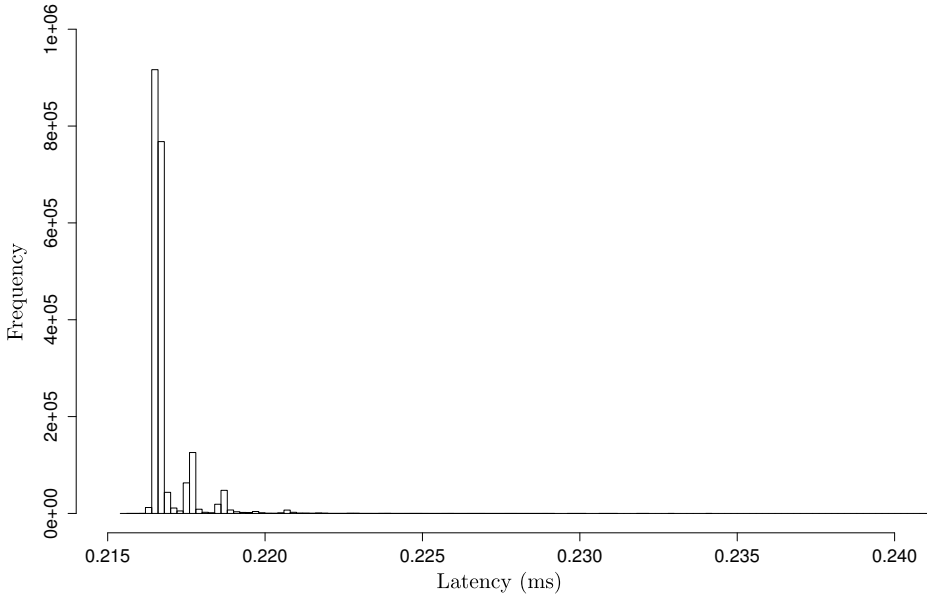


Figure 4.3: Latency stability for a fixed number of events inside each GMEP.

The tests show that, on the Kepler GTX680, it's easier for the application to reach the aimed throughput (greater than 800 MB/s) and latency (lower than 1 ms) and the choice of the right packet size is not an issue.

A stress latency test has been carried out. A realistic cycle of 4.8 s ON and 16.8 s OFF has been considered for 48 hours: these values have been chosen because they are compatible with the CERN SPS accelerator duty cycle (4.8 s ON /16.8 s OFF). During the OFF part of the cycle all the memory structures, both in the GPU global memory and in host memory, are allocated anew.

A histogram showing the latency times measured on the GTX 680 during this test is shown in Figure 4.3.

4.2 From single-ring to multi-ring

A multi-ring implementation is needed since the process $K^+ \to \pi^-\mu^+\mu^+$ is likely to produce events containing more than one ring.

A parallel multi-ring reconstruction algorithm could be implemented using the *divide and conquer* paradigm. It consists in recursively splitting a problem in simpler sub-problems that can be solved directly.

A criterion to divide a dataset containing hits that belong to multiple rings in smaller datasets, each accommodating only those hits that belong to the same ring, is necessary.

4.2.1 Ptolemy's theorem

In his treatise on astronomy, the Almagest, Ptolemy derived several geometrical results. In particular, in Euclidean geometry, one of Ptolemy's theorems connects the lengths of the four sides and the two diagonals of a cyclic quadrilateral, that is quadrilateral whose vertexes all lie on a single circle. *In a convex quadrilateral, if the sum of the*

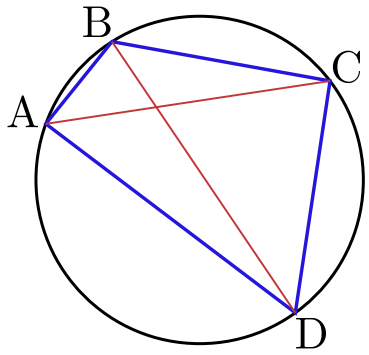


Figure 4.4: Ptolemy's theorem.

products of its two pairs of opposite sides is equal to the product of its diagonals, then the quadrilateral can be inscribed in a circle.

In the particular case shown in figure 4.4, the theorem states that:

$$\overline{AC} \cdot \overline{BD} = \overline{AB} \cdot \overline{CD} + \overline{BC} \cdot \overline{DA}. \tag{4.8}$$

This is a useful condition for our problem, since it does not require any other information but the distances between hits inside the RICH.

The maximum number of hits inside the NA62 RICH, that will be sent to the GPU for each event, is 64. During an offline analysis one could probably afford the time spent in the evaluation of all the possible combinations of four hits that, in the NA62 worst case scenario (an event made of 64 hits), are given by:

$$C_{n,k} = \binom{64}{4} = 635376. \tag{4.9}$$

However, in a synchronous, low-latency environment like the NA62 Level 0 trigger, one can only afford to examine few hundreds of these combinations per event.

4.2.2 Triplet finding

To reduce the number of tests to few hundreds per event, one possibility is to choose few *triplets*, i.e. a set of three hits assumed to belong to a single ring, and iterate through all the other hits while checking whether Ptolemy's relation is satisfied. In this way, in the worst case scenario of a 64 hits event, the number of iterations for each triplet turns out to be 64.

The choice of the test triplets is decisive for the efficiency of the algorithm in the selection of events with two or three charged particles. Considering events with two or three rings like the one shown in figure 4.5, so as to maximize the probability that all the points of a triplet belong to the same circle, I chose to form eight triplets with the hits at the edges of the set:

1. $t_0 = \{A(x_A, y_A), B(x_B, y_B), C(x_C, y_C)\}$, so that $x_A \leq x_B \leq x_C \leq x_i$
2. $t_1 = \{D(x_D, y_D), E(x_E, y_E), F(x_F, y_F)\}$, so that $x_i \leq x_D \leq x_E \leq x_F$
3. $t_2 = \{G(x_G, y_G), H(x_H, y_H), J(x_J, y_J)\}$, so that $y_G \leq y_H \leq y_J \leq y_i$
4. $t_3 = \{K(x_K, y_K), L(x_L, y_L), M(x_M, y_M)\}$, so that $y_i \leq y_K \leq y_L \leq y_M$
5. $t_4 = \{N(x_N, y_N), O(x_O, y_O), P(x_P, y_P)\}$, so that $u_N \leq u_O \leq u_P \leq u_i$
6. $t_5 = \{Q(x_Q, y_Q), R(x_R, y_R), S(x_S, y_S)\}$, so that $u_i \leq u_Q \leq u_R \leq u_S$
7. $t_6 = \{T(x_T, y_T), U(x_U, y_U), V(x_V, y_V)\}$, so that $v_T \leq v_U \leq v_V \leq v_i$

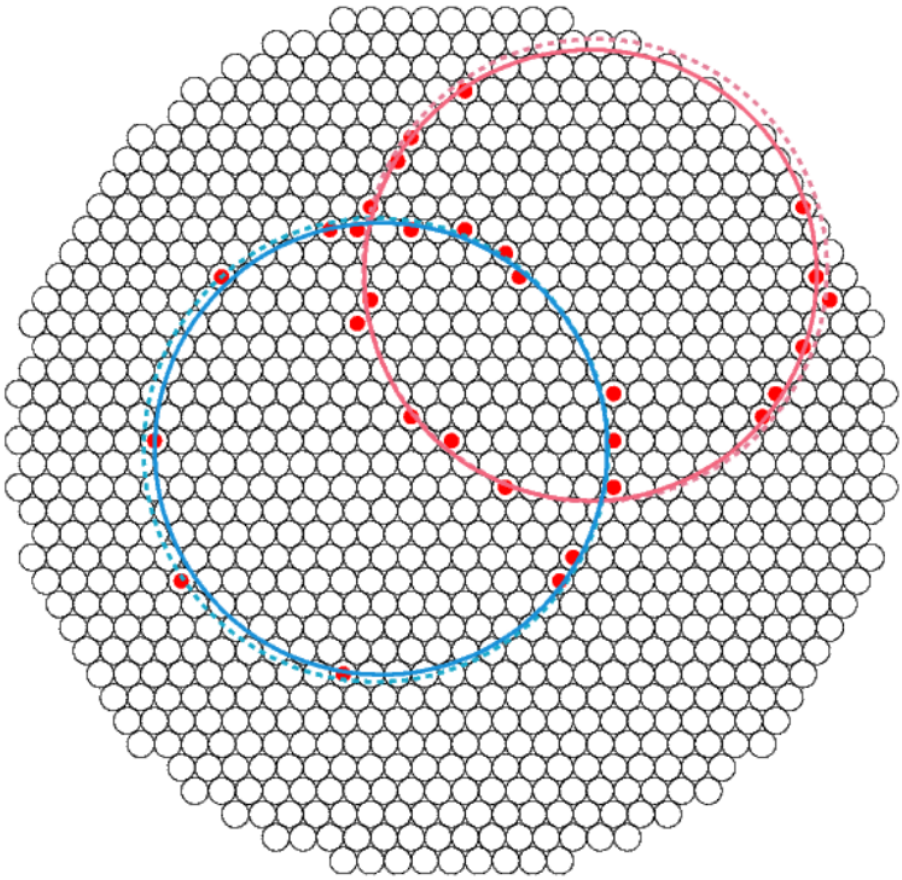


Figure 4.5: The reconstruction of a two-rings event.

8. $t_7 = \{X(x_X, y_X), Y(x_Y, y_Y), Z(x_Z, y_Z)\}$, so that $v_i \leq v_X \leq v_Y \leq v_Z$

where u and v are the coordinates on the diagonal axes. It is also useful to define a lower threshold d_{th} for the distance between points belonging to the same triplet: before adding a new point to the triplet, the distance is checked to be above d_{th}. The chosen value for d_{th} is 27 mm, equivalent to three times the diameter of a PMT.

4.2.3 RICH lattice parametrization

The performance of a trigger based on GPUs sets some requirements on the efficiency of the communication with the electronics connected to it. In particular, the format of the data that carry the hit information can be optimized to be directly used by the GPU without having to resort to a lookup table or complicated mapping functions e.g.

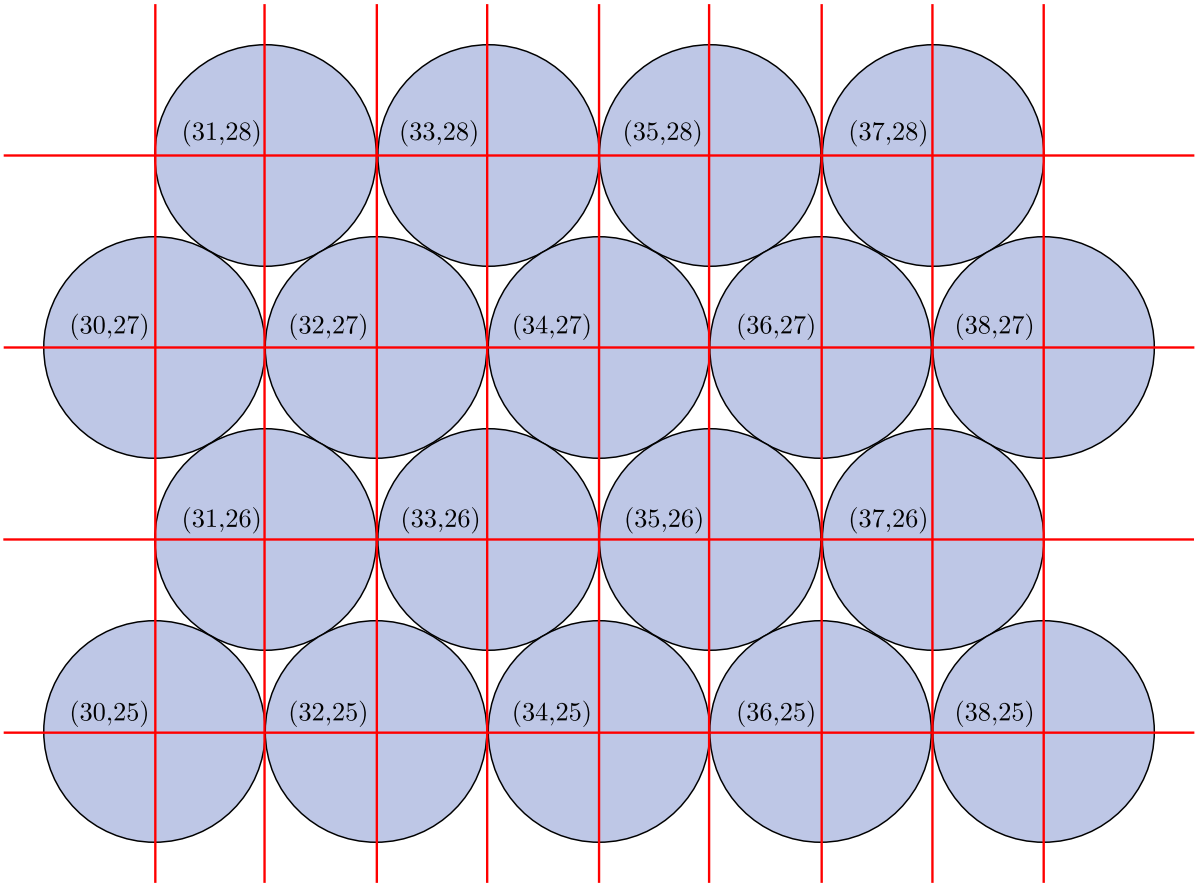


Figure 4.6: The ids of the PMTs were modified to indicate the coordinates of their center.

containing modulo, square root, approximation operations, which are time consuming, or that have to take into consideration coordinate shifts or holes in the matrix.

In NA62, the two circular flanges of the RICH detector host 976 PMTs each, distributed on a compact hexagonal lattice. Hence, the minimum number of bits that can host the information on the PMT identity is 11 ($2^{11} = 2048$).

A grid like the one shown in figure 4.6 can be used to ease the computation of coordinates and to set a unit for the measurement of the distance between PMTs. It uses 12 bits since half of the grid points do not host a PMT center. The unused 4 bits might be used to work with a 4 times finer grid, so as to account for little displacements of the PMTs and to allow more precise results.

I used 8 bits to store the information about the x coordinate and 8 bits for the y coordinate.

The squared distance between two PMTs, $A(x_A, y_A)$ and $B(x_B, y_B)$, in units of a PMT radius will be hence given by:

$$d^2(A, B) = (x_A - x_B)^2 + 3(y_A - y_B)^2 \tag{4.10}$$

where the factor 3 accounts for the fact that the lattice is compact hexagonal.

4.3 Multi-ring algorithm

The multi-ring algorithm proceeds then as follows:

1. Eight triplets are formed as described in section 4.2.2.

2. For each triplet, iteratively all the other points are considered for the Ptolemy's condition (equation 4.2.1) with a tolerance of 5mm, accounting for possible displacements from the grid crossings.

3. For each triplet, if a point satisfies the Ptolemy's condition, it is added to a *positive-match* list.

4. At the end of the loop on the points, each positive-match list will contain all the points that likely belong to the ring characterized by the triplet.

5. For each triplet, a single-ring fitting algorithm is executed.

6. The radius and the coordinates of the center are returned for each triplet.

7. The rings returned by each triplet are compared (it is possible that two triplets belong to the same ring) and then checked for some trigger condition (e.g. length of the radius, position of the center, fraction of the total number of hits that belongs to the ring, etc.)

4.3.1 Parallelization Strategy

First, the formation of the triplets is not computational intensive. Since a certain number of events has to be collected before the GPU computation can start, the triplets can be found on the fly as the event are enqueued in the buffer, together with the AoS-to-SoA conversion (see section 3.1.1).

Second, each event is associated to eight blocks of threads, one for each triplet. Hence, the algorithm will run four times in parallel, allowing to exploit more GPU resources already with a low number of events per packet.

Subsequently, the Ptolemy's condition can be checked in parallel inside each block of threads. Since, for each triplet, every thread has to be able to read all the elements of the positive-match list, the latter can be located both in the GPU global memory or inside the shared memory, which is much faster.

Finally, the single-ring algorithm is executed in parallel on each positive-match list, using the parallelization strategy described in section 4.1.1.

4.3.2 Implementation

The input dataset transferred to the GPU global memory for the evaluation is shown in Listing 4.1. Each event is received in the format described by the header at line 6. This header is then followed by a number `eventLength_` of `HIT` structures (line 8).

Events are converted and accumulated into a structure of arrays called `Element` (line 17).

As an event is received, the four triplets are formed and pushed inside the array at line 22. Point `i` of triplet `j` belonging to event `k` can be accessed by linear indexing:

```
hit = i + 3 * j + 3 * numberOfTripletsPerEvent * k        (4.11)
```

where `numberOfTripletsPerEvent` is 8.

I choose to buffer 256 events ($\sim 32kBytes$) before tagging the packet as ready to be sent to the GPU, because at lower packet sizes the PCI Express Gen.2 effective transfer

bandwidth could be lower than 1 GB/s and the throughput would be still in the initial raising part of the curve shown in figure 4.1.

Listing 4.1: Input dataset for the multi-ring kernel.

```
1  #ifndef MAXEVENTS
2  #define MAXEVENTS 256
3  #endif
4  #define MAXHITS 64
5
6  struct MEPEVENT_RAW_HDR {
7      uint32_t timestamp_; // timestamp of the event
8      uint16_t eventLength_; // number of hits in the event
9      uint16_t reserved_; // reserved space
10 }__attribute__ ((__packed__));
11
12 struct HIT {
13     uint8_t y; // y coordinate of the hit
14     uint8_t x; // x coordinate of the hit
15 } __attribute__ ((__packed__));
16
17 typedef struct  {
18   uint32_t timestamp[MAXEVENTS]; // timestamp of each event
19   uint16_t actualsize; //actual number of events
20   uint8_t id_x[MAXHITS*MAXEVENTS]; // x pos for each hit
21   uint8_t id_y[MAXHITS*MAXEVENTS]; // y pos for each hit
22   int triplet[3*numberOfTripletsPerEvent*MAXEVENTS]; // three points per triplet with ↩
         four triplet per event
23   uint16_t length[MAXEVENTS]; // number of hits inside each event
24 } Element;
```

Once `Element` is full, it is transferred to the GPU global memory (line 10, Listing 4.2) and the kernel is launched (line 11). The kernel launch parameters are:

- **`blocksPerGrid`**: It represents the number of CUDA thread blocks that will be used for computing on the whole `Element`. Since one block is used for each triplet for each event, the value of this parameter will be given by the total number of triplets in the dataset.

- **`threadsPerBlock`**: It determines the number of CUDA threads that each block will execute and coordinate through synchronization and shared memory. With a maximum number of 64 hits per event, so as to evaluate Ptolemy's condition for each hit in only one step, the value of this parameter is 64.

- **`smemSize`**: It is the amount of shared memory needed by each block, expressed in bytes. It is determined by the information that needs to be shared among all

the threads in the block and cannot be larger than 48 kB. Every thread in a block needs to be able to read and write:

  - the length of the positive-match list (`sizeof(int)`),
  - the id of every hit on the same ring, in order to be able to run the reductions (`64*sizeof(float)`),
  - the coordinates of the points that form the triplet (`2*numberofHitsInATriplet*sizeof(int)`).

- **`stream_id`**: The CUDA stream on which the kernel will run.

Listing 4.2: The kernel launch and the transfers to and from the GPU.

```
1  const unsigned int threads = MAXHITS;
2  const unsigned int eventsPerBlock = 1;
3  const unsigned int numberOfTripletsPerEvent = 8;
4  dim3 threadsPerBlock(threads,1,1);
5
6  const int blocksPerGrid = ceil( MAXEVENTS * numberOfTripletsPerEvent / (float)↩
       eventsPerBlock);
7  const int smemSize = sizeof(int) + 6*sizeof(int) + 64*sizeof(float);
8
9
10 cudaMemcpyAsync((void*)Element_GPU, (void*)Element_Host, sizeof(Element),↩
       cudaMemcpyHostToDevice, stream_id);
11 multiring<<<blocksPerGrid, threadsPerBlock, smemSize, stream_id>>>(Element_GPU, ↩
       Result_GPU, utils );
12 cudaMemcpyAsync((void*)Result_Host,(void*)Result_GPU,sizeof(Result),↩
       cudaMemcpyDeviceToHost, stream_id);
```

Listing 4.3: The output of the multi-ring kernel.

```
1  typedef struct {
2    uint32_t timestamp[MAXEVENTS];
3    uint16_t actualsize;
4
5    float xCenter[MAXEVENTS*numberOfTripletsPerEvent];
6    float yCenter[MAXEVENTS*numberOfTripletsPerEvent];
7    float radius[MAXEVENTS*numberOfTripletsPerEvent];
8    int nHitsPerCandidate[MAXEVENTS*numberOfTripletsPerEvent];
9  } Result;
```

Finally, once the kernel has finished execution, a `Result` is produced (Listing 4.3), and transferred back to the host memory. It contains the arrays of the coordinates of the

centers (line 5-6), the radii of the reconstructed rings (line 7), together with the number of hits belonging to each ring (line 8).

Listing 4.4: The multi-ring kernel.

```
1  __global__  void multiring(Element* Event, Result* Result, UTILITY* utils)
2  {
3    __shared__ int triplet_s[6];
4    __shared__ unsigned int length;
5    __shared__ float reduction[64];
6    unsigned int eventIdx = blockIdx.x/numberOfTripletsPerEvent;
7    unsigned int tripletIdx = blockIdx.x%numberOfTripletsPerEvent;
8    __syncthreads();
9
10   if(threadIdx.x < 3){
11           triplet_s[2*threadIdx.x] = Event->id_x[Event->triplet[3*↩
                 numberOfTripletsPerEvent*eventIdx+threadIdx.x+3*tripletIdx]];
12           triplet_s[2*threadIdx.x+1] = Event->id_y[Event->triplet[3*↩
                 numberOfTripletsPerEvent*eventIdx+threadIdx.x+3*tripletIdx]];
13     }
14
15   __syncthreads();
16
17   unsigned int hitIdx = threadIdx.x + eventIdx * MAXHITS;
18   unsigned int hitIdx_device = MAXHITS*blockIdx.x + threadIdx.x;
19
20   if(Ptolemy(Event->id_x[hitIdx], Event->id_y[hitIdx], triplet_s) ) {
21       utils->x_hits_on_ring[hitIdx_device] = Event->id_x[hitIdx];
22       utils->y_hits_on_ring[hitIdx_device] = Event->id_y[hitIdx];
23     }
24   else {
25       utils->x_hits_on_ring[hitIdx_device] = 0;
26       utils->y_hits_on_ring[hitIdx_device] = 0;
27     }
28
29   __syncthreads();
30     length = 0;
31   if(utils->x_hits_on_ring[hitIdx_device] != 0)
32           atomicAdd(&length, 1);
33   __syncthreads();
34   if (threadIdx.x == 0){
35           Result->actualsize = Event -> actualsize;
36           Result->nHitsPerCandidate[blockIdx.x] = length;
37   }
38   __syncthreads();
39   singlering(utils, reduction, Result);
40 }
```

The multi-ring kernel source code is shown in Listing 4.4 and proceeds as follows:

- the information that needs to be shared among the threads of a block is declared at lines 3-5;
- to ease the indexing, every eight blocks `eventIdx` is increased by one, while

`tripletIdx` runs from 0 to 2 for each event (lines 6-7);

- the first three threads in a block load the triplet to the shared memory, concurrently in one step (lines 10-13);
- for each hit, Ptolemy's condition is checked and, in case of positive match, the hit coordinates are copied in a utility temporary array (lines 20-27);
- the number of hits on the ring is counted (lines 30-33);
- the single-ring algorithm is executed only on the hits that belong to the ring (line 39) and modifies the `Result` structure in listing 4.3, with the coordinates of the center of the ring and its radius length.

The `__syncthreads()` function is a barrier: any thread in the block that reaches the barrier must wait for all the other threads to reach it. `__syncthreads()` is used to avoid race conditions and data hazards (see section 2.4).

4.4 Conclusion

This chapter shows that GPU architectures are well suited to run pattern recognition algorithms on detector data in a trigger environment: a seedless, fast and stable algorithm was developed for the reconstruction Čerenkov light rings in the NA62 RICH detector.

From the physics perspective, such an enhancement of the trigger capabilities, would allow inclusion of new triggers and the selection of events that are currently not recorded efficiently.

A framework to feed this ring-fitting GPU kernel with data coming from the TEL62 data acquisition boards is required and will be discussed in the next chapter.

Chapter 5

Parallel trigger framework

In the previous chapters I have shown that GPUs are able to satisfy the requirements in throughput and in latency of a real-time trigger system such as that of NA62. However this fact would not have much value if the whole system would not meet the requirements and miss deadlines, thus degrading the trigger system efficiency.

In this chapter I describe a complete parallel trigger framework that was developed to make off-the-shelf hardware able to work in a low-level trigger environment. It is a *multi-threaded heterogeneous software platform* designed with optimized latency of executing instructions and communication. The latency is guaranteed by the *deadline-aware mode* described in section 5.2.2.

Finally, the test system and the results of a complete set of measurements are discussed.

5.1 Parasitic Level 0 trigger based on GPUs

At the time of this writing, triggering in HEP experiments is done in multiple stages so as to maintain its discovery potential while keeping the costs low (see section 1.5).

The dominant technology in triggering at the lowest level are *Field Programmable Gate Arrays* (FPGAs). FPGAs are digital integrated circuits that are (re)programmable for any logic function. The decision time as well as its predictability, are much better than any general-purpose computer (in the range between few microseconds and milliseconds).

However FPGAs filter the event stream based upon a quick look at the data, because they are usually not flexible enough to run complex algorithms.

At the high-level stages of the trigger, the maximum time to reach a decision is much more relaxed (in the range of seconds). Systems at this stage do not require any real-time hardware processing on reduced event information.

The intent of this chapter is to show that a system based on GPUs can fill the gap between the hardware and the software stages. This enhancement of the trigger capabilities would allow inclusion of new triggers and the selection of events that are currently not recorded efficiently. At NA62, this system was designed to run in *parasitic mode* (Fig. 5.1): DAQ boards collect hits and form *Multi GPU Packet* (MGP) that are sent through 1 Gbit Ethernet links to the host where they are buffered inside a GMEP. After this buffering stage, the host transfers the GMEP to the GPU memory and the computation can start. Once the trigger algorithm execution has finished, the trigger decision are sent back to the host memory.

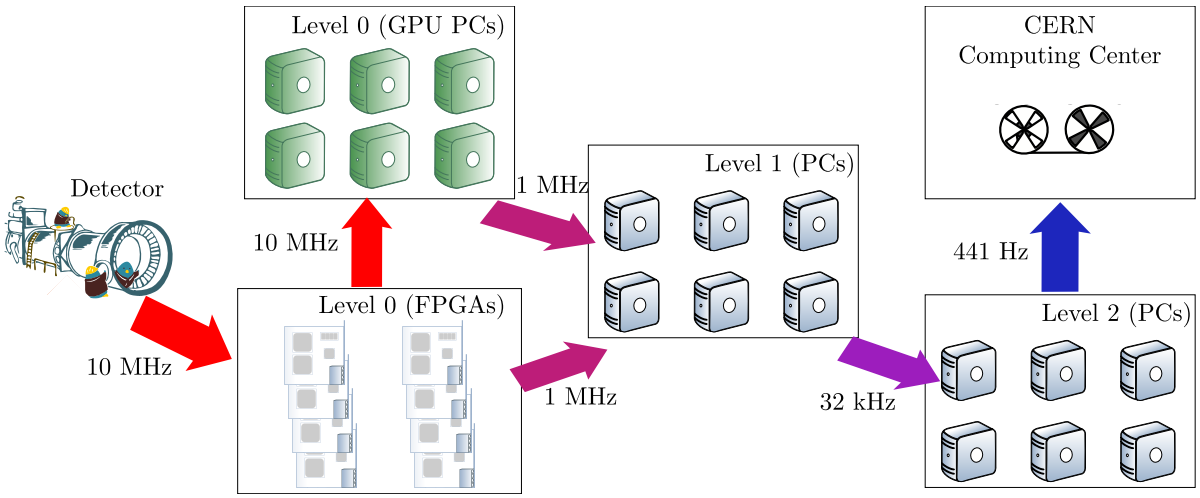


Figure 5.1: NA62 computing infrastructure: systems based on GPUs will act as an enhanced Level 0 trigger receiving trigger primitives from the DAQ boards in parasitic mode.

5.2 Trigger framework

In order to verify that the possibility to use such system in the NA62 Level 0 trigger is viable, a careful assessment of the real-time performance is required. The previous

chapters show that GPU alone satisfy the requirements in latency, throughput and stability. However the performance measurement is based on the assumption that algorithms run on data that are already located in the host memory.

The developed framework is in charge of data transport from the Ethernet wire to the GPU memory, the launch of the trigger algorithm kernel(s) and the transfer of the results back to the host memory.

It consists of three main components:

- *Network communication* in the back end;
- *Multithreaded scheduler* in the back end;
- *GPU kernel* in the front end.

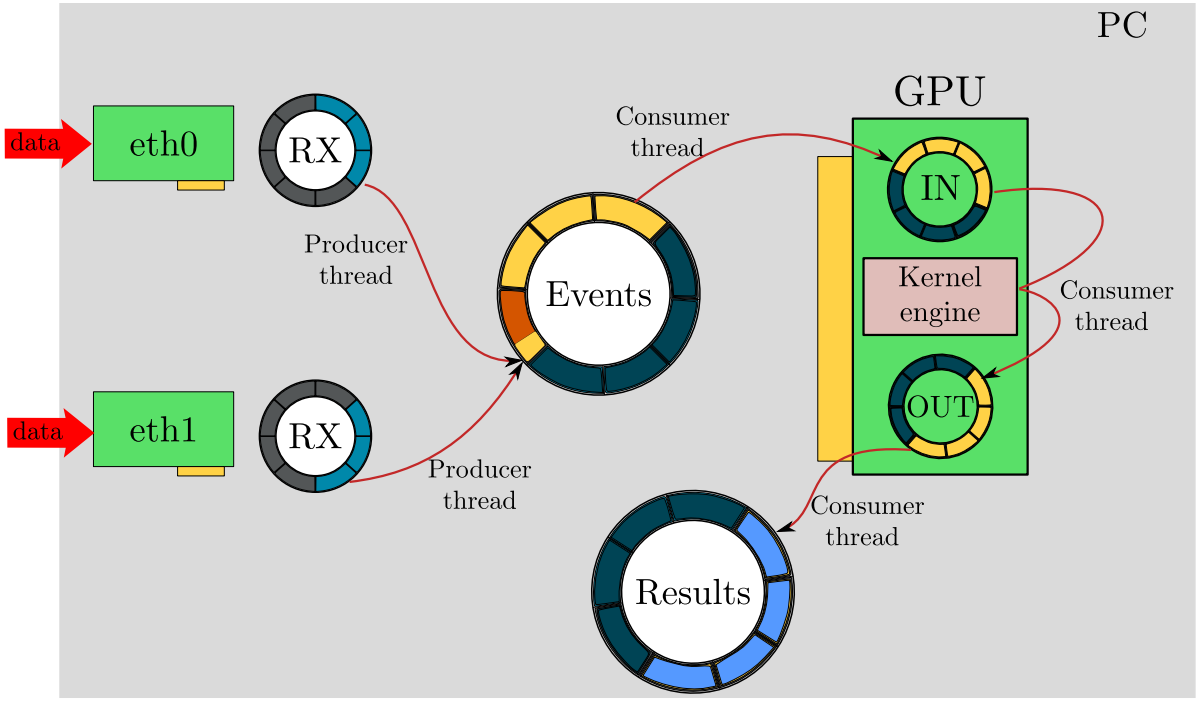


Figure 5.2: Scheme of the parallel trigger framework design.

5.2.1 Network communication

The computer on which the framework is installed communicates with the data acquisition electronics by means of *User Datagram Protocol* (UDP) messages (Fig. 5.3), referred to

as *datagrams*, on an IP-based Ethernet physical network interface.

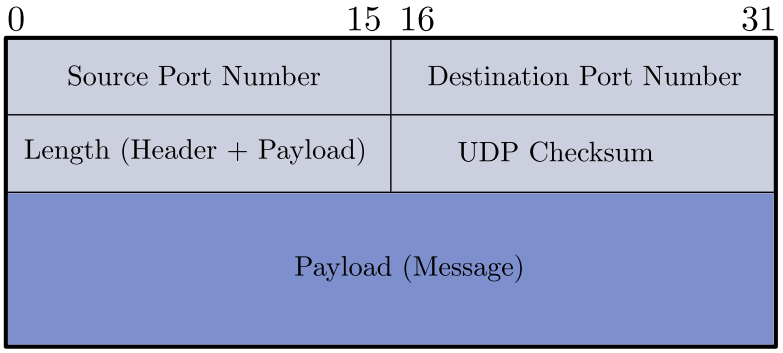


Figure 5.3: Structure of a *UDP datagram*. *Source port* is an optional field and indicates the port of the sending process. *Destination port* represents the port on which the payload is being sent. *Length* is the size in bytes of the UDP header and the payload. The *checksum* is used for error detection purposes. *Payload* contains the message to be sent to the destination host.

The main reason why the use UDP datagrams was chosen over the more reliable Transmission Control Protocol (TCP) is that UDP is *connectionless*. This means that information can be sent without having to set up a hard connection and without the need of sending back a positive response for every packet. Dropping packets is preferable to waiting for delayed packets, which is not an option in a real-time system. Furthermore, packet dropping due to data corruption won't be related to cable length. Packet loss can be detected and monitored, and could only be expected to lead to a small reduction in trigger efficiency.

Event data are encapsulated in a MGP (Fig. 5.4), located inside the UDP datagram payload and stored in little-endian format.

Finally, the payload of the MGP can host one or more event data in the format described in figure 5.5. The number of events in one MGP is called MGP factor.

PF RING

Software based solutions become popular whenever the performance of CPUs and I/O buses matches or exceeds that of network hardware. As of this writing, we are in this

UDP source port: 54818
UDP destination port: 54914

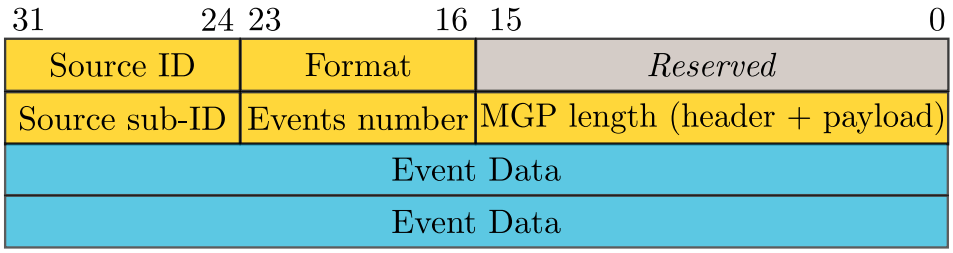


Figure 5.4: Structure of a NA62 Multi GPU Packet. The MGP contains information about the packet source (detector and sub-detector), the number of events encapsulated inside it, and the total length of the MGP, header included.

situation, with 10 Gbit/s data rate links becoming more and more manageable in software with modern multi-core architectures and multi-queue devices.

The network communication is based on *PF RING* [52], a framework for accelerating network packet capture that implements a memory-mapped buffer allocated at socket creation, where the incoming packets are copied.

PF RING can work both on top of standard NIC drivers, or on top of specialized drivers. The PF RING kernel module is the same, but depending on the drivers being used some functionalities and performance are different:

- **PF RING-aware Drivers**: These drivers are designed to improve packet capture by pushing packets directly to PF RING without going through the standard Linux packet dispatching mechanisms.

- **Direct NIC Access (DNA)**: These drivers allows packets to be read directly from the network interface by simultaneously bypassing both the Linux kernel and the PF RING module in a zero-copy fashion. While working in DNA mode, the circular buffer that PF RING allocates at socket creation is replaced by a ring buffer allocated by the device driver which contains pointers to the incoming packets in the host memory. The trigger framework can read packets by simply dereferencing those pointers and updating the index to the next slot that will host the next incoming packets. Since the memory containing the packet ring and the registers are both mapped to user-space, communication with the network adapter

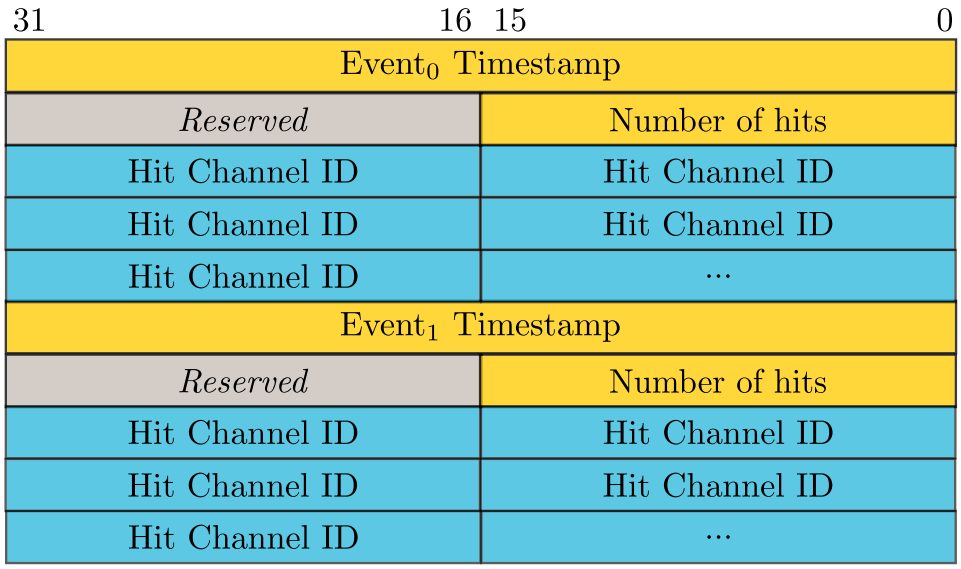


Figure 5.5: Each event contains the coordinates information of every hit in a time window. If one is interested in the fine timestamps of each hit, half of the payload can be used to store such time information. Timestamp is a suitable quantity for identifying the time at which the event occurred within an accelerator spill and it can be used to determine the trigger time. The present scheme assumes that the time matching of the hits is done outside the proposed system.

happen in *Direct Memory Access* (DMA). This means that the network adapter can access system memory independently of the CPU, generating only one interrupt per transferred block (instead of one interrupt per transferred byte).

The DNA and PF RING Linux modules must be loaded before being able to open and use a PF RING buffer.

The network communication is implemented in the back end of the trigger framework since the user could potentially set the adapter registers to invalid values.

I implemented a class `PFRingHandler` (Listing 5.1) that takes care of the socket management and the capture of the packets:

- the `PFRingHandler` constructor initializes the socket by calling the function `pfring_open` at line 2. This function returns a handle of type `struct pfring` that can be used in subsequent calls. The function call needs the symbolic name of the network device on which the ring will be open (e.g. `eth0`), together with the maximum packet capture length.

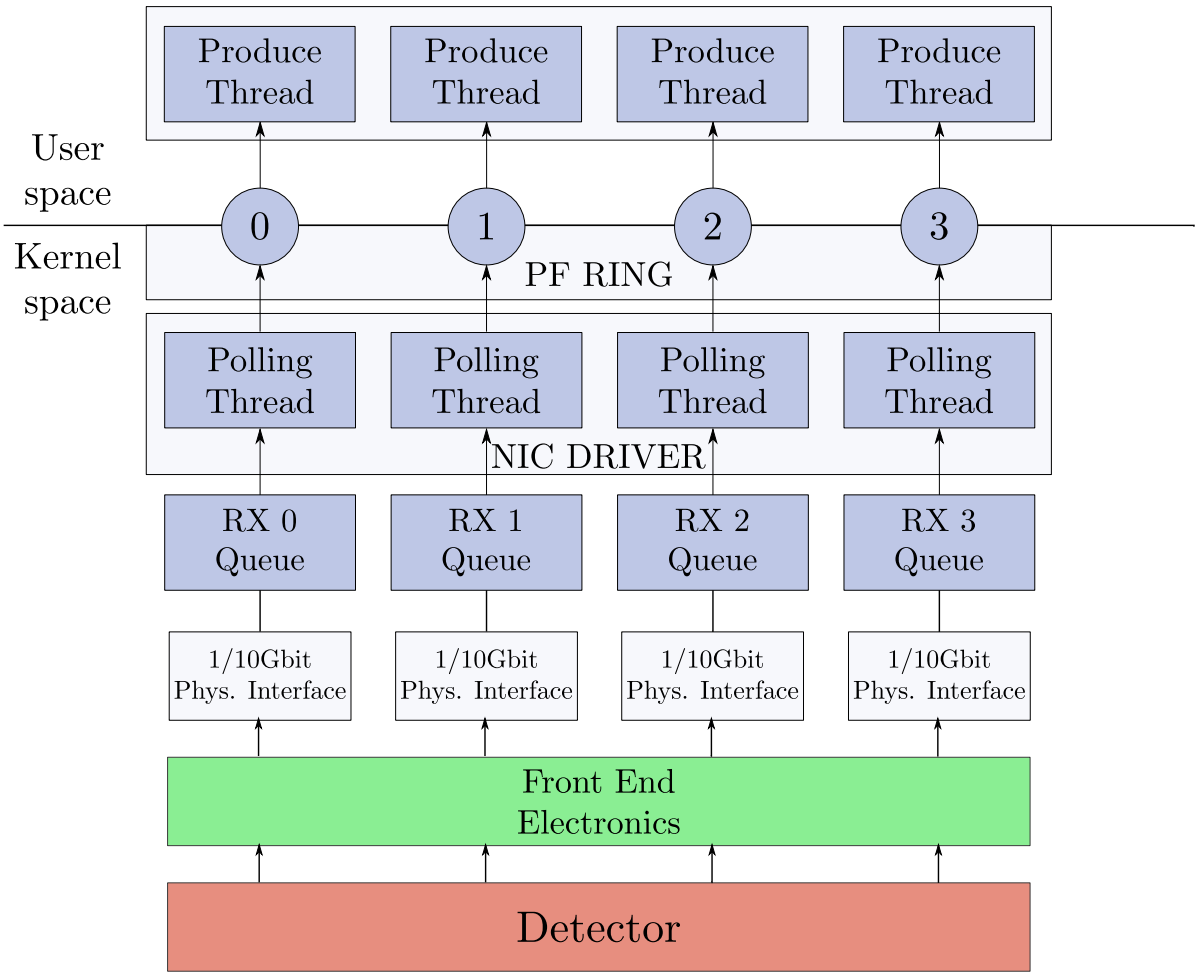


Figure 5.6: The system is connected to the data acquisition electronics through a certain number of Gigabit Ethernet links. Each thread creates a PF RING structure and can have zero-copy access to the packets payload.

- A ring has to be enabled before being ready for use (line 5), and disabled after use (line 8).
- Line 14 shows that the function `pfring_recv()` modifies where `*buffer` is pointing, to the memory address in the NIC memory, in order to enable zero-copy implementation. Similarly a packet could be sent using the function `pfring_send()` at line 18.

CPU threads as there are network interfaces are spawned, with each one having a different and independent instance of `PFRingHandler`. Because these threads will take the data from the network interfaces and feed them to the whole system, they will be referred to

Listing 5.1: PF RING interface used by `PFRingHandler` class.

```
// Socket initialization
pfring* pfring_open(char *device_name, u_int32_t snaplength, u_int flags);

// Ring is enabled
int pfring_enable_ring(pfring *ring);

// Ring is disabled
int pfring_disable_ring(pfring *ring);

// Device Termination
void pfring_close(pfring *ring);

// Capture of the incoming packets
int pfring_recv(pfring *ring, u_char** buffer, u_int buffer_len,
    struct pfring_pkthdr *hdr, u_int8_t wait_for_incoming_packet);

// Transmission of a packet
int pfring_send(pfring *ring, u_char *pkt, u_int pkt_len,
    u_int8_t flush_packet);

// Ring statistics (packets received and dropped)
int pfring_stats(pfring *ring, pfring_stat *stats);
```

as *Producer threads*.

5.2.2 Multi-threaded scheduler

Once a Producer thread has read a UDP datagram from the NIC memory space, it strips the header off it, hence leaving only the NA62 MGP that is the building block of a GMEP. In order to manage the GMEPs formation and their movements inside the system, the class `StreamQueue` was developed.

Furthermore, the class `StreamQueue` ensures the thread-safety of the data structures.

`StreamQueue` makes use of four ring buffers that correspond to a FIFO queue that is read and written cyclically (Fig. 5.7). According to figure 5.2, they will be referred to as:

- **`Events` ring buffer**,
- **`IN` ring buffer**,
- **`OUT` ring buffer**,

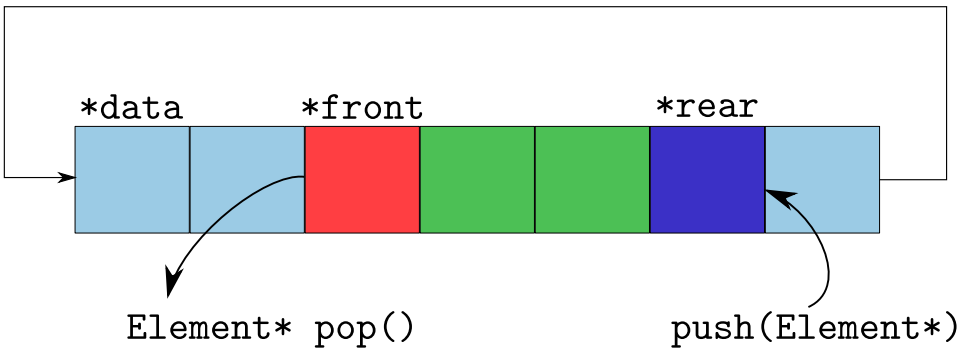


Figure 5.7: Array representation of a ring buffer.

- **`Results` ring buffer**.

Events ring buffer

The `Events` ring buffer is allocated in the host pinned memory (described in section 2.5.2) during the initialization, as shown at line 9 of listing 5.2. Its elements are GMEPs, that are gradually filled by the Producer threads with the MGPs coming through the network interfaces.

A MGP can be *pushed* in a GMEP by calling the `Streamqueue::push()` method (listing 5.3). Depending on the algorithm, these data could be converted into structures of arrays before being copied in the `Events` ring buffer (lines 32-33 of listing 5.3).

Once a GMEP has been completely filled (or the latency deadline has been reached), the Producer threads notifies with a *condition variable broadcast* (see section 2.4) to all the *Consumer threads* that a GMEP is ready to be *popped* out from the `Events` queue.

Deadline-aware mode

The framework can also work in a *deadline-aware* mode, by defining at compile time the variables `TRIGGER__PUSHTIMEOUT`, which activates the deadline-aware mode, and `__TIMEOUTINSECONDS`. The variable `__TIMEOUTINSECONDS` indicates the maximum duration of the time interval between the arrivals of the first and the last events in the same GMEP. In case the event rate decreases and a GMEP cannot be executed in a time

Listing 5.2: `StreamQueue` initialization.

```
1  void StreamQueue::create(size_t n)
2  {
3    // memory allocations
4    data = 0; //ptr to the 1st element of the array hosting the host ring buffer
5    data_gpu = 0; // ptr to the 1st element of the array hosting the GPU ring buffer
6    r_data_gpu = 0; // ptr to the 1st element of the array hosting the results ring ↩
         buffer on GPU
7    r_data = 0; // ptr to the 1st element of the array hosting the results ring buffer
8    pitch = NearestSuperiorPow2(sizeof(GMEP));
9    CudaSafeCall(cudaMallocHost((void**)&data, n*sizeof(GMEP)));
10   CudaSafeCall(cudaMalloc((void**)&data_gpu, n*pitch));
11   CudaSafeCall(cudaMalloc((void**)&r_data_gpu, n*sizeof(Result)));
12   CudaSafeCall(cudaMallocHost((void**)&r_data, n*sizeof(Result)));
13
14   // initialization of the ring buffers
15   front = data;
16   rear=data;
17   rear->actualsize = 0;
18   front->actualsize = 0;
19   r_front_gpu = r_data_gpu;
20   r_rear_gpu = r_data_gpu;
21   r_front = r_data;
22   r_rear = r_data;
23   front_gpu = data_gpu;
24   rear_gpu = data_gpu;
25   count = 0;
26   gpu_count = 0;
27   r_count = 0;
28   max_size = n;
29
30   // initialization of mutexes and condition variables
31   pthread_mutex_init(&mtx, NULL);
32   pthread_mutex_init(&r_mtx, NULL);
33   pthread_mutex_init(&gpu_mtx, NULL);
34   pthread_cond_init(&element_available_on_cpu, NULL);
35   pthread_cond_init(&result_available_on_cpu, NULL);
36 }
```

Listing 5.3: `StreamQueue push()` method.

```
1  void StreamQueue::push(const char *dataPtr, const uint16_t& dataLength,
2              const char *originalData)
3  {
4    //acquire lock
5      pthread_mutex_lock(&mtx);
6      struct MEP_RAW_HDR* rawData = (struct MEP_RAW_HDR*) (dataPtr);
7      uint8_t offset = sizeof(MEP_RAW_HDR);
8      struct MEPEVENT_RAW_HDR* rawHitData;
9      uint8_t* hitInfo;
10
11 #ifdef TRIGGER__PUSHTIMEOUT //if the time deadline is active
12     double time = omp_get_wtime(); //take the current time and compare it with the ↩
           timestamp of the last event
13     if ((rear->actualsize !=0) && ((time - rear->timeoflastevent) > 0.5e-3))
14     {
15         rear ++ ;
16         if(rear == data + max_size )
17             rear = data;
18         rear->actualsize = 0;
19     }
20     if(rear->actualsize==0) // if this is the first event of the GMEP
21         rear->timeoffirstevent = time;
22     rear->timeoflastevent = time;
23 #endif
24     for (uint16_t i = 0; i < rawData->eventCount; ++i) //cycle through the number of ↩
           events in the UDP datagram
25     {
26         rawHitData = (struct MEPEVENT_RAW_HDR*) (dataPtr + offset);
27         offset+=sizeof(MEPEVENT_RAW_HDR);
28         hitInfo = (uint8_t*)(dataPtr+offset);
29         rear->length[rear->actualsize] = 0;
30         for(uint16_t j = 0; j < rawHitData->eventLength_; ++j) // cycle through the ↩
               hits in the event
31         {
32             rear->id_x[rear->actualsize*MAXHITS+j] = hitInfo[2*j]; // data is copied ↩
                   in the rear GMEP
33             rear->id_y[rear->actualsize*MAXHITS+j] = hitInfo[2*j+1];//AoS to SoA ↩
                   conversion
34             rear->length[rear->actualsize] = rawHitData->eventLength_;
35             rear->timestamp[rear->actualsize] = rawHitData->timestamp_;
36             offset+=rawHitData->eventLength_*sizeof(HIT);
37             rear->actualsize++;   // increment the size of the GMEP
38 #ifdef TRIGGER__PUSHTIMEOUT // check the lateness of the GMEP
39           if((rear->actualsize == MAXEVENTS) ||
40             (( rear->timeoflastevent - rear->timeoffirstevent > __TIMEOUTINSECONDS) &&
41                 rear->actualsize))
42 #else
43         if(rear->actualsize == MAXEVENTS) // if the GMEP is full ask for a Consumer
44           {
45 #endif
46             count++;
47             pthread_cond_broadcast(&element_available_on_cpu); // condition var. ↩
                   broadcast
48             rear ++ ;
49             if(rear == data + max_size ) //if the end of the linear buffer is reached,↩
                    restart from the first element
50               rear = data;
51             rear->actualsize = 0; // set the length of the new GMEP to 0
52             if(count == max_size) // if the ring buffer is full, the consumers are not↩
                    fast enough and data is lost.
53               std::cerr<< "\npop method is back pressuring" << std::endl;
54             }
55         }
56
57   pthread_mutex_unlock(&mtx); // unlock the mutex when finished
58
59 }
```

interval smaller than __TIMEOUTINSECONDS, it is flagged as complete and nevertheless the Producer thread asks the Consumer threads to pop it out from the ring buffer.

For the above reason the scheduler is said to be *partially preemptive*: it is aware of the lateness of the packet and can start the computation before an input buffer element has collected the required amount of events. The attribute "partially" is used because memory transfers from/to the GPU and kernel executions cannot be manually stopped once started.

GPU and Results ring buffers

The `IN` and `OUT` ring buffers are allocated inside the GPU global memory during the `StreamQueue` initialization (lines 10-11 listing 5.2).

The `IN` buffer elements are the GMEPs transferred from the host `Events` ring buffer. The `OUT` contains instead the results produced when a GPU kernel is launched on a GMEP.

These two buffers are managed by three *Consumer threads*, that independently and concurrently try to perform the following sequential tasks:

1. If a GMEP flagged as complete exists in the `Events` ring buffer, the Consumer thread will try to acquire the lock associated to the `Events` ring buffer and pop the GMEP out of it. Then the GMEP will be moved to the rear element of the `IN` ring buffer, which is located in the GPU global memory. This operation is done by calling the `StreamQueue::pop()` method (listing 5.4).

   Before the CUDA API function `cudaMemCpyAsync()` is called, the lock on the `Events` ring buffer is released so that other threads can run operations on that data structure. Thanks to this expedient, CUDA streams will be automatically interlaced in a pipeline, and the framework will be able to sustain higher input rates.

2. Once the GMEP has been completely transferred to `IN` ring buffer it is ready to be computed. The Consumer thread launches the kernel (see section 5.2.2) taking the GMEP in the front of the `IN` ring buffer as input data and writing the results inside the rear element of the `OUT` ring buffer (line 14 in listing 5.5)

3. After the computation has completed, the Consumer thread transfers the front element of the `OUT` ring buffer pushing it into the host `Results` ring buffer (line 18 in listing 5.5).

Listing 5.4: `StreamQueue pop()` method.

```
1  Element* StreamQueue::pop(cudaStream_t& stream_id)
2  {
3    // pop and push are contending the same lock
4    pthread_mutex_lock(&mtx);
5    int rc = 0;
6    // if there is no GMEP available wait for a signal on the condition variable
7    while(count == 0 && rc == 0)
8      rc = pthread_cond_wait(&element_available_on_cpu, &mtx);
9    //the pointer to the GMEP is saved so that the lock on the event ring buffer can be released
10   void* front_lock = (void*)front;
11   ++front;
12   if(front == data + max_size)
13     front = data;
14   --count;
15   // release the lock on the Event ring buffer
16   pthread_mutex_unlock(&mtx);
17   // try to acquire the lock on the IN ring buffer on the GPU
18   pthread_mutex_lock(&gpu_mtx);
19
20   // the IN ring buffer is reorganized to host the new GMEP
21   ++gpu_count;
22   if(gpu_count + NSTREAMS == max_size)
23     std::cerr << "The GPU is backpressuring, try increasing the buffer size" << std::endl;
24   // the pointer to the rear element of the GMEP is saved, the lock on the IN ring buffer can be released
25   Element* gpu_ptr = (Element*)((void*)rear_gpu);
26   rear_gpu = (unsigned char*)rear_gpu + pitch;
27   if(rear_gpu == (unsigned char*)data_gpu + max_size*pitch)
28     rear_gpu = data_gpu;
29   // lock released, other threads can now issue commands on the IN ring buffer
30   pthread_mutex_unlock(&gpu_mtx);
31   // data is copied asynchronously to the rear element of the IN ring buffer using the stream stream_id
32   CudaSafeCall(cudaMemcpyAsync((void*)gpu_ptr, (void*)front_lock,
33                 sizeof(Element),cudaMemcpyHostToDevice, stream_id));
34   // wait for the jobs on the current stream to finish
35   CudaSafeCall(cudaStreamSynchronize(stream_id));
36
37   return gpu_ptr;
38 }
```

GPU kernel

The file `kernel.h`, shown in listing 5.6, contains the kernel and its launch parameters. By modifying it and changing the definition of the `Element` and `Result` data structures

Listing 5.5: `StreamQueue push_result()` method.

```
Result* StreamQueue::push_result(Element* gpu_ptr, cudaStream_t& stream_id)
{
    // try to acquire the lock on IN ring buffer
    pthread_mutex_lock(&gpu_mtx);
    // reorganize the ring buffers to host a new Result packet
    Result* r_gpu_ptr = (Result*)((void*)r_rear_gpu);
    Result* r_rear_lock = (Result*)r_rear;
    r_rear_gpu = (unsigned char*)r_rear_gpu + sizeof(Result);
    if(r_rear_gpu == (unsigned char*)r_data_gpu + max_size* sizeof(Result))
        r_rear_gpu = r_data_gpu;
    // release the lock on the IN ring buffer
    pthread_mutex_unlock(&gpu_mtx);
    // run the kernel saving the results in r_gpu_ptr
    kernel<<<blocksPerGrid, threadsPerBlock, smemSize,stream_id>>>(gpu_ptr, r_gpu_ptr)↩
        ;
    // check for kernel errors (active only in debug compilation)
    CudaCheckError();
    // copy the result to the Result ring buffer on the stream stream_id
    CudaSafeCall(cudaMemcpyAsync((void*)r_rear_lock,(void*)r_gpu_ptr,
                    sizeof(Result),cudaMemcpyDeviceToHost, stream_id));
    // lock the Result ring buffer
    pthread_mutex_lock(&r_mtx);
    // reorganize the ring buffers
    --gpu_count;
    Result* result_ptr = r_rear;
    r_count++;
    r_rear++;
    // the condition variable broadcast signal is sent
    pthread_cond_broadcast(&result_available_on_cpu);

    if(r_rear == r_data + max_size)
        r_rear = r_data;
    // if the results are not used in time they are lost
    if (r_count == max_size)
        std::cerr<< "\npop_result method is back pressuring"<< std::endl;
    // release the lock on the Result ring buffer
    pthread_mutex_unlock(&r_mtx);
    // return a pointer to the host result
    return result_ptr;

}
```

to fit the needs of the problem, the user is able to run many of the algorithms that could be suited for a real-time trigger system, without having to change the `StreamQueue` class implementation.

Listing 5.6: The file `kernel.h` contains the kernel and its launch configuration

```
1  #ifndef KERNEL_H_
2  #define KERNEL_H_
3  // choose the kernel configuration that better
4  //suits your problem
5  const unsigned int threads = 32;
6  const unsigned int eventsPerBlock = 4;
7
8  dim3 threadsPerBlock(threads,eventsPerBlock,1);
9  const unsigned int blocksPerGrid = ceil( MAXEVENTS / (float)eventsPerBlock);
10 // type here the amount of shared memory needed by each block
11 const unsigned int smemSize = 32 *sizeof(float);
12
13 __global__  void kernel(Element* Event, Result* Result)
14 {
15     //Your kernel goes here!
16     ...
17 }
18
19 #endif /* KERNEL_H_ */
```

5.3 Tests

The measurements of the total latency for the complete system cannot be performed using timers managed internally by the system itself. This is due to the fact that the time at which the framework is able to see a MGP is delayed with respect to the time the MGP was actually produced.

Therefore, the assessment of the performance must be carried out in an external time reference frame. While the DAQ boards and PCs both have accurate time references

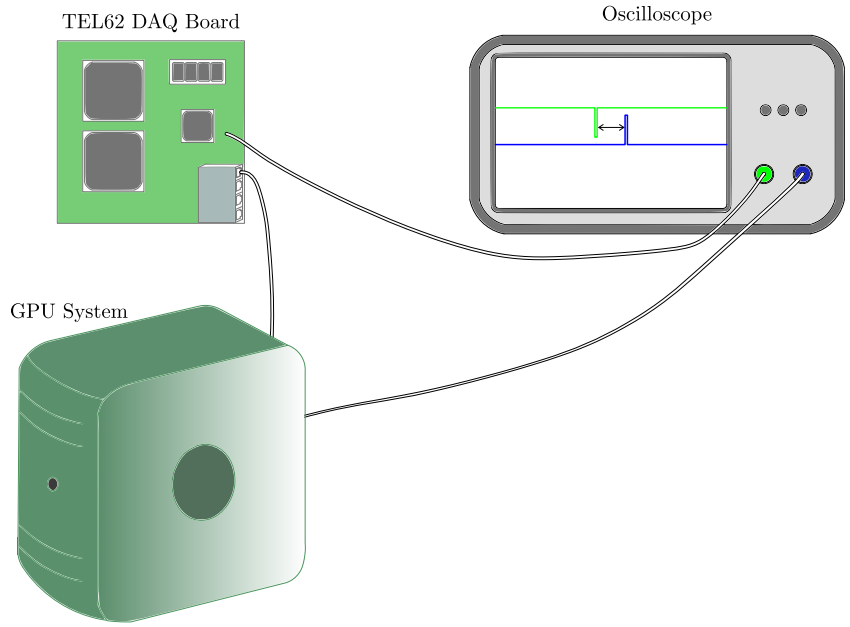


Figure 5.8: The latency is measured with an oscilloscope, by using the *transmission of packet* in the TEL62 as *start* and the *end of computing* in the system (through the PC LTP port) as *stop*.

internally, they cannot be easily synchronized. The chosen configuration for carrying out the tests is shown in figure 5.8:

- The MGPs are sent by the TEL62 DAQ board. The TEL62 was programmed to send a signal on its LEMO connector. This signal can be visualized on an oscilloscope every time the first word of a MGP is transmitted on the GbE link.
- The PC is connected to the oscilloscope through its *parallel port* (LPT). A signal can be produced by calling the function in Listing 5.7.

The latency of the execution of the code between these two signals can hence be measured on the oscilloscope.

Listing 5.7: The inline function used to raise a signal on the pin indicated by `value` of the parallel port mapped at the memory address`0xabcd`.

```
1 #include <unistd.h> /* for libc5 */
2 #include <sys/io.h> /* for glibc */
3
4 inline
5 void signalLPT(const unsigned char& value)
6 {
7         outb(value,0xabcd);
8         outb(0x00,0xabcd);
9 }
```

Hardware configuration

For all the tests carried out in this section, the following hardware has been used:

- Computer:
  - Host CPU: Intel Xeon E5-2620 2.00 GHz (12 physical cores divided on two sockets)
  - Host Network Interface: Intel I350-T2
  - Device GPU: NVDIA Tesla Kepler K20
  - Communication bus: PCI Express Gen. 2
- Readout board: NA62 TEL62 DAQ Board
- Oscilloscope: LeCroy WaveRunner 104MXi

The presence of CPUs on two different sockets poses the problem typical of *Non-Uniform Memory Access* (NUMA) architectures: the access latency depends on the position of the memory location with respect to the CPU.

Since the PCI Express slot that the GPU uses is directly connected to the second CPU, in order not to have threads accessing data structures at different latencies, each thread

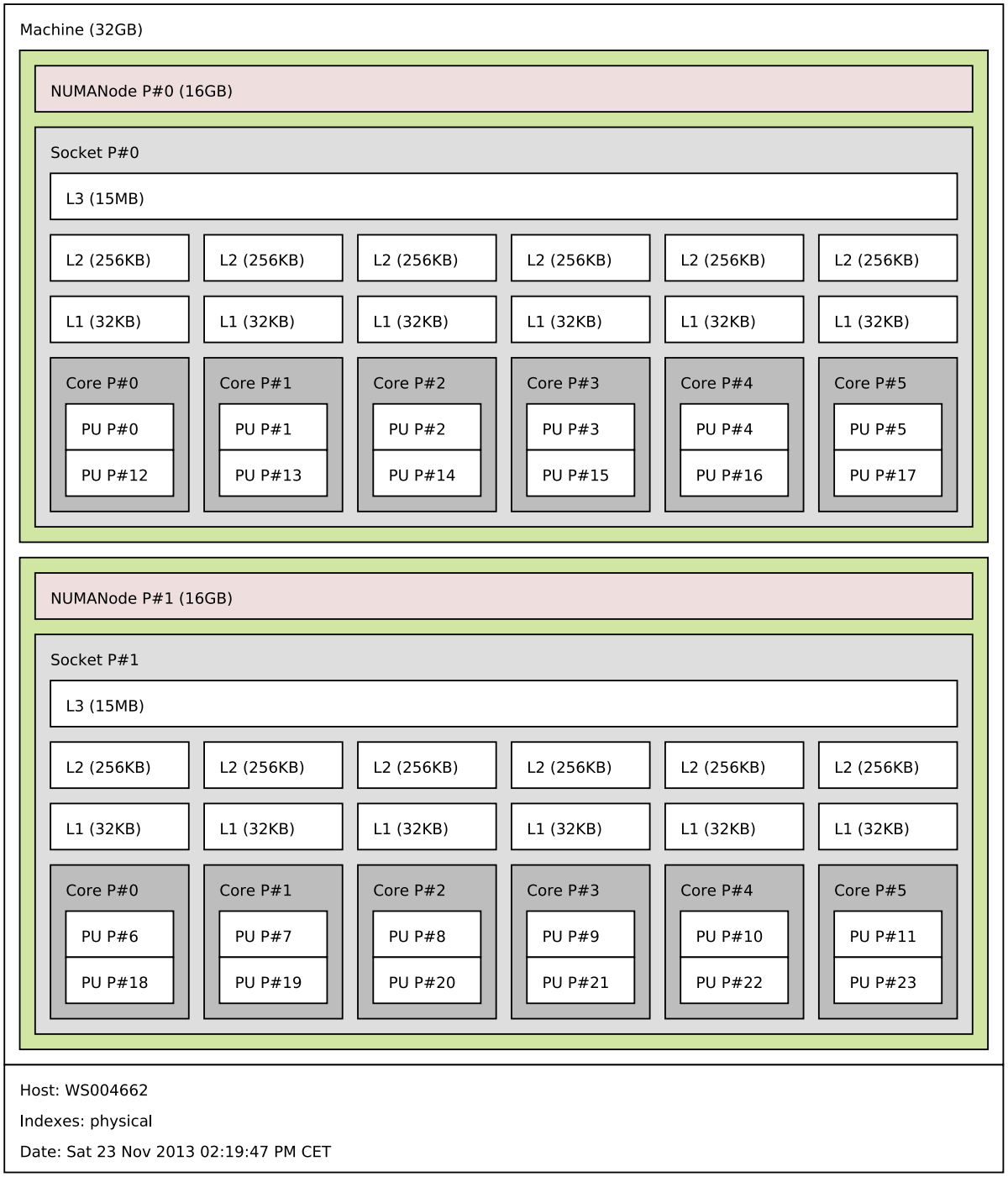


Figure 5.9: In a processor with multiple cores, each *processing unit* (PU) has their own *Level 1 and 2 caches* (L1/2). These allow the core to read and write from and to the cache without worrying about interfering with other cores. The cores need shared storage to exchange information easily. The *Level 3 cache* (L3) is shared among all cores, so it is used as a sort of communal storage space where information is available for all threads.

was associated to a different core on the second CPU, in a procedure called *CPU pinning*.

5.3.1 Parallel port latency

The delay and the latency instability introduced by the call of the function `signalLPT()` in Listing 5.7 has to be quantified so as to validate the measurement technique.

The measurement of the LPT signal delay was made in two steps (Fig. 5.10):

1. The time interval Δt_1 between a signal sent by the TEL62 DAQ board when a single MGP is being transmitted and the response signal sent by the computer when the MGP is seen by `PFRingHandler` is measured.

2. The time interval Δt_2 between the same two signals when the function `signalLPT()` is called right before the `PFRingHandler` response is measured.

The delay Δt_{LPT} introduced by the function `signalLPT()` is hence given by $\Delta t_{LPT} = \Delta t_2 - \Delta t_1$.

The result of this measurement shows that the latency introduced by raising a signal on the LPT port has an average $\Delta t_{LPT} = 0.97\mu s$ with a standard deviation $\sigma_{LPT} = 0.06\mu s$ calculated on a sample of 300 measurements.

Since both the latency and its delay are well under control, the measurement approach can be considered valid.

GPU kernel and data format

The GPU kernel used for the following tests is the ring-fitting algorithm described in section 4.1. Furthermore, at the time the tests were performed the Event data format inside a MGP contained information not needed by the ring-fitting algorithm in the payload 5.11.

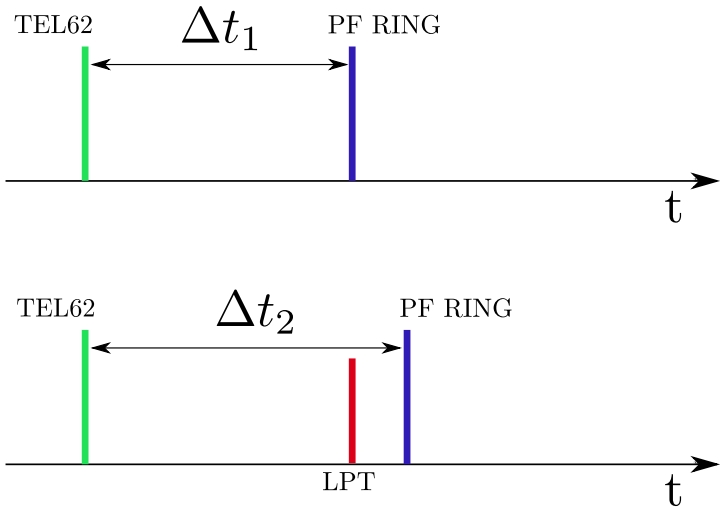


Figure 5.10: Measurement of the delay introduced by raising a signal on the LPT port.

5.3.2 Latency measurement

The experimental method described above was used to measure the *trigger latency* and all its components. The trigger latency is defined as the time interval that starts in the moment of the production of one event inside the detector, and ends when the result of the kernel computation on this event has finished.

In the NA62 Level 0 trigger, after the trigger decision has been taken on the PC, it should be sent to the L0TP (see section 1.5). After a resynchronization, if the trigger decision is positive, all the detectors transmit the event data to the Level 1 stage of the trigger. The total Level 0 latency will hence include also L0TP, resynchronization and retransmission.

The total trigger (L0) latency Δt_{tot}, for the L0 trigger based on GPUs, is the sum of the following contributions:

- *Event collection time*;
- *Communication latency*;
- *Conversion of AoS to SoA* and *management* of the `StreamQueue` ;
- *GPU data transfers and computation.*

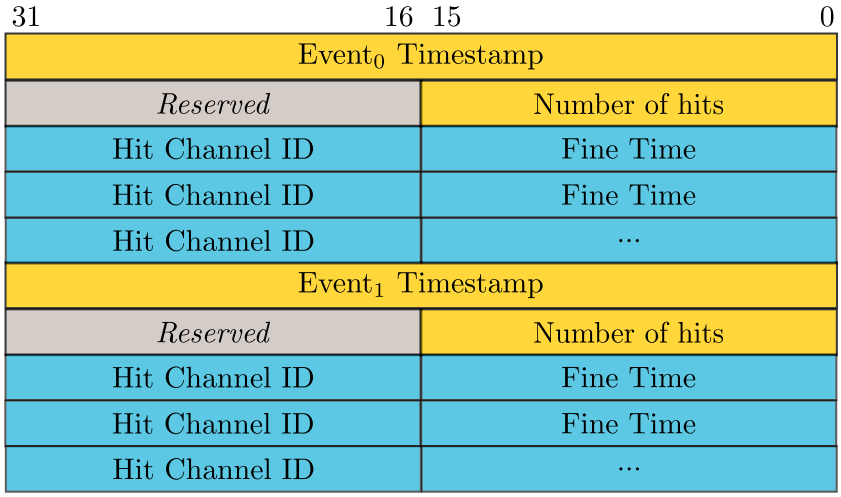


Figure 5.11: Event data format payload used for the tests. Information about the hit fine time is not used since the algorithm assumes that the hit time matching has already been done by the TEL62 DAQ boards.

5.3.3 Event collection

The event collection time $\Delta t_{gathering}$ is the time needed by the experiment to physically produce the amount of events N_{events} given by the chosen size of the GMEP. It is connected to the event production rate R by the relation:

$$\Delta t_{gathering} \simeq \frac{N_{events}}{R} \tag{5.1}$$

where R is expected to be ~ 10 MHz.

5.3.4 Communication latency tests

A packet processing system usually exercises many system components, and it is important to identify individual contributions to the performance of the overall system. CPU cycles are one of the main resources that we account for, but depending on the system under test there might be other resources that are in short supply, such as memory or I/O bus bandwidth. Two cost factors can be distinguished:

- *system costs*: account for the resources consumed in bringing packets from the network to the application, and vice-versa (e.g. interrupt processing, system call overhead, data copies);
- *application costs*: account for the specific processing done by applications (e.g. timestamping, classification, checksumming)

All my tests were focused on determining the system costs.

Cost factors can be split into per-packet and per-byte components. The latter are usually simple to deal with, because they are proportional to the memory bus bandwidth of the system. Modern CPUs easily reach $4 \div 10$GB/s on the memory bus, meaning that the per-byte costs are in the microsecond range even for the largest packets with size in the range of thousands of kBytes.

On the contrary, per-packet costs are highly variable depending on the system and the application. From these considerations it follows that the correct way to evaluate the performance of packet processing systems is to drive the system with variable size packets.

The communication time Δt_{comm} contribution to the total latency is made up of:

- a *transmission time* Δt_{transm} needed to send information on the GbE link;
- a *PF RING time* interval Δt_{PFRING} between the time the MGP is physically in the NIC memory and the time in which the MGP is received by `PFRingHandler`.

Since the class `PFRingHandler` can work both with the PF RING DNA driver and with the vanilla one, I measured the communication time Δt_{comm}, with increasing MGP size, for both solutions to quantify the advantage of the DNA driver approach Fig. 5.12.

These measurements were taken with an event rate of 300 kHz (almost 25 MB/s payload size) using one GbE link. By allowing the mapping of the NIC buffers and registers to the userspace, the DNA driver shows a smaller latency and, above all, a better latency stability. Regarding the vanilla driver, the main reason why the latency and its stability improve with increasing MGP size, is that packets are copied to the userspace in bursts: small packets (few hundreds bytes) will have to wait for the arrival of other packets before being copied, whereas larger packets will be copied as they are received in the NIC.

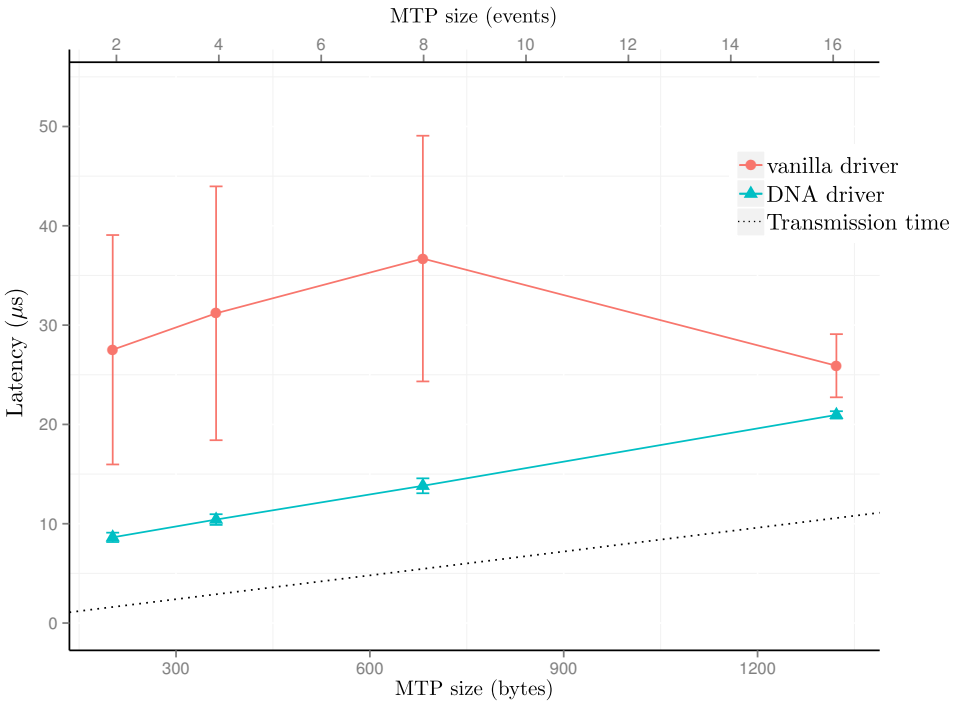


Figure 5.12: Communication time for different MGP sizes, with and without using the PF RING DNA driver. The test was done with an incoming event rate of 300kHz with each event containing 15 hits. The average and the standard deviation are calculated on a sample of 300 measurements. The dotted line represents the transmission time estimated for a GbE link.

The performance shown by the implementation with the PF RING DNA driver enabled justifies the usage of this driver in my final implementation and in the following tests.

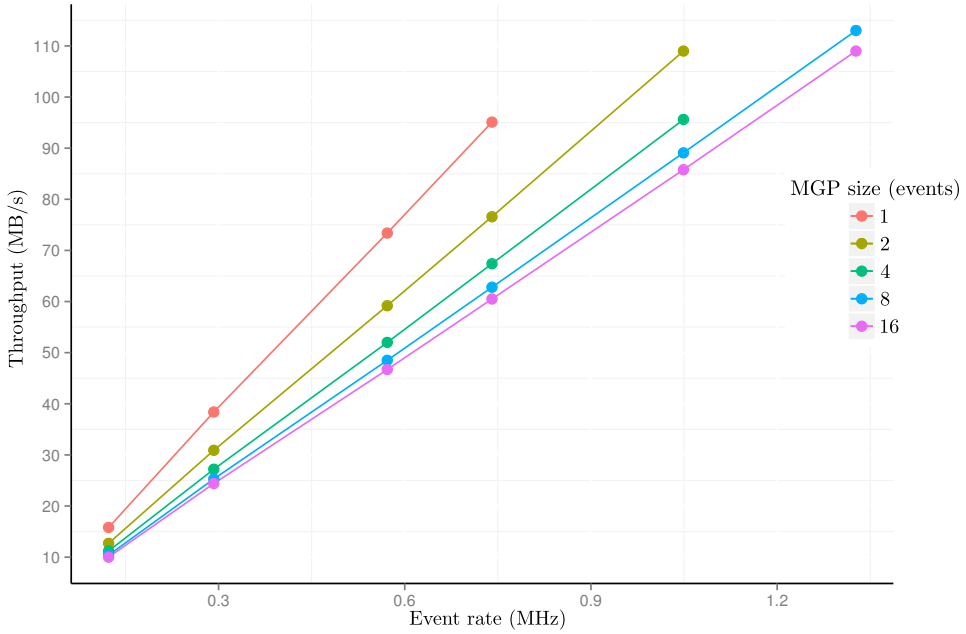


Figure 5.13: Throughput as function of the event rate for different values of the incoming event rate.

To better characterize the communication time, it was measured with a varying input event rate, for different MGP sizes (Figure 5.14). This measurement shows that, for a constant number of events per MGP, communication time does not depend on the incoming event rate.

The measurements performed do not span the entire range of parameter values since, for values of the MGP size between 1 and 4, the TEL62 DAQ board does not saturate the GbE link bandwidth. The same issue did not allow the measurements of the throughput shown in Fig. 5.13. This last measurement shows the additional contribution of the packing overhead to the effective throughput, and leads the choice of the MGP size to larger values.

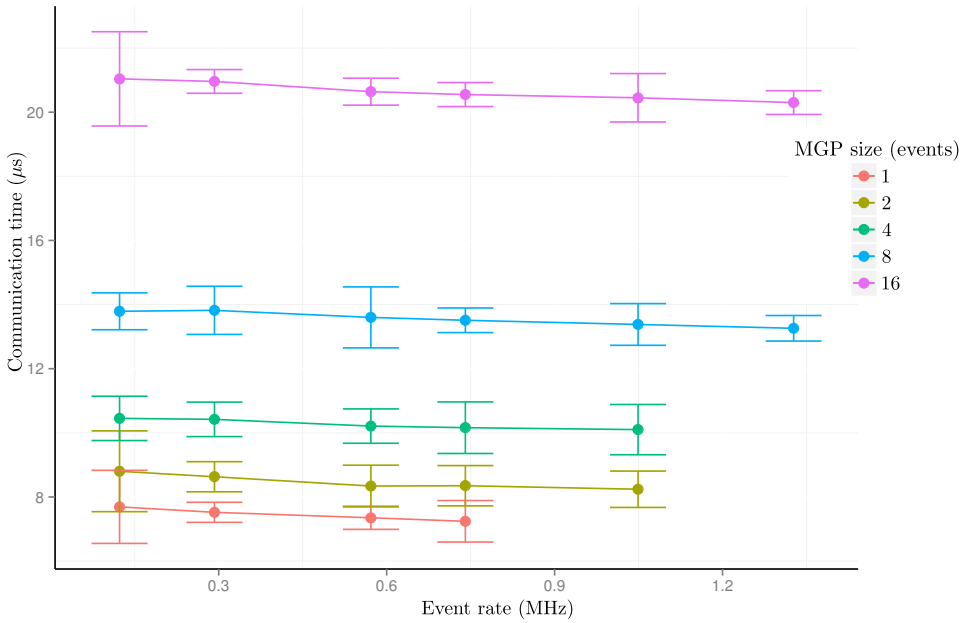


Figure 5.14: Communication time as a function of the incoming event rate, for different MGP sizes.

5.3.5 AoS to SoA conversion

Any parallel architecture, and especially vector processors, benefit from the conversion of the input data from Array of Structures (AoS) to Structure of Arrays (SoA), as previously explained in 3.1.1. The gain in the kernel performance could not be justified if the time spent in this operation of "transposition" of the input data (done on the fly) were comparable with the time spent in the kernel.

To verify that this is not the case, the time spent by `StreamQueue` in the AoS-to-SoA conversion $\Delta t_{conversion}$ was measured by adding and removing the part of the code that applies it. The results (Fig. 5.15) show that time spent in data conversion is negligible if compared to potential speed up.

5.3.6 GPU weak scaling

Figures 5.16 and 5.17 show the behavior of each operation run on the GPU when processing a GMEPs of different sizes.

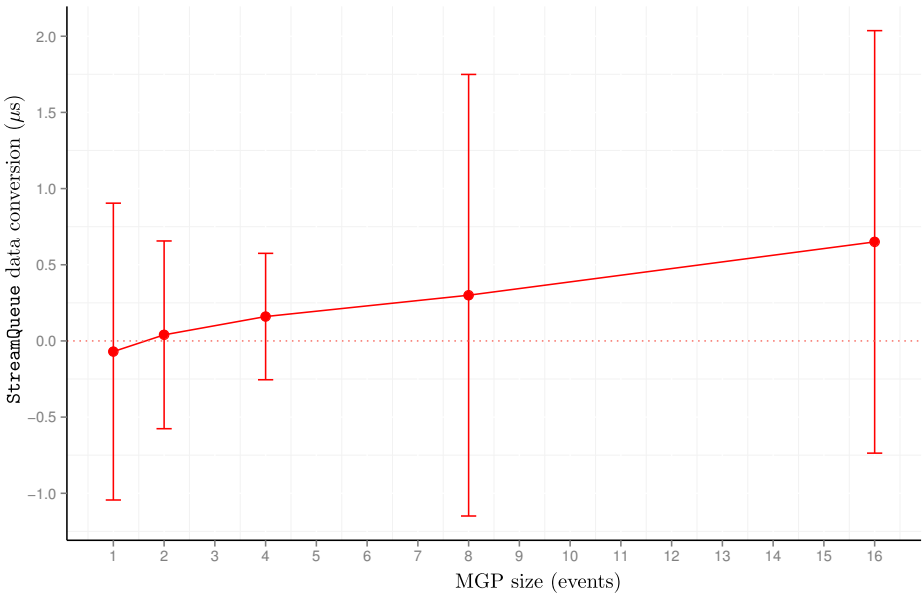


Figure 5.15: The time spent in the transposition of the data structures as a function of MGP size: such time is negligible with respect to other sources of latency, and sometimes the error is larger than the time interval to be measured. The test was done with an incoming event rate of 300 kHz (25 MB/s payload). The average and the standard deviation are calculated on a sample of 300 measurements.

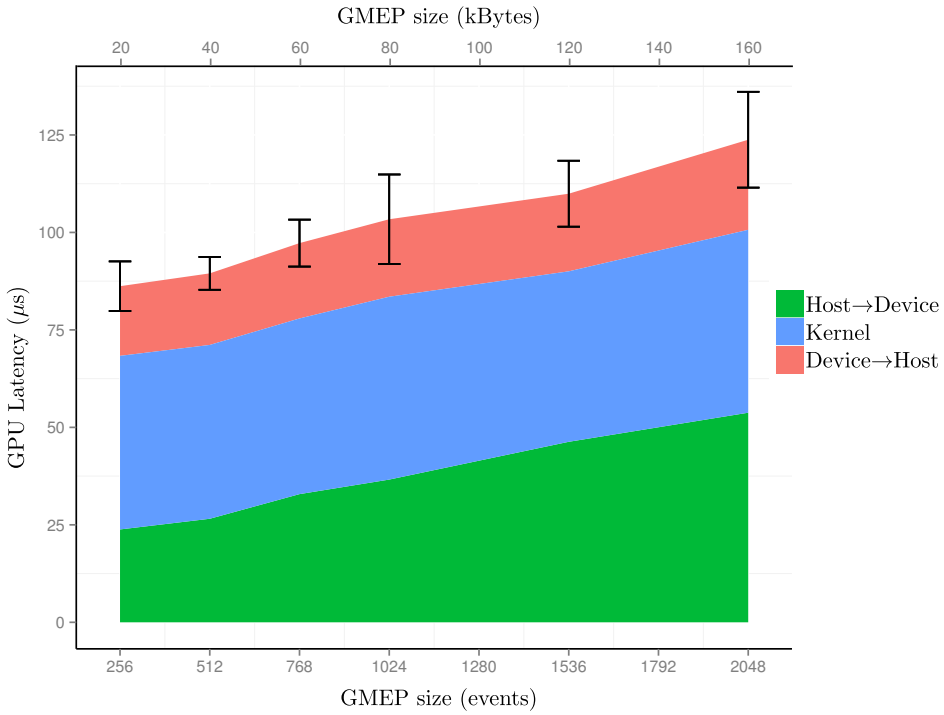


Figure 5.16: The time spent in each operation of the GPU processing by the Consumer threads. Since the copies and the kernel cannot be ungrouped, the error bars refer to the total time spent in the three operations, based on a sample of 300 measurements, with an incoming event rate of 1327kHz and a MGP size of 16 events.

The growth in latency for an increasing size of the GMEP is mainly due to the transfer of the GMEP from the host to the device memory. However, the transfer latency measured for a GMEP of 512 events is not double the latency for a GMEP of 1024 events. This is due to the fact that the effective bandwidth of the PCI Express bus increases with a growing GMEP size. The fact that the kernel time remains almost constant is due to the underutilization of the GPU resources for such number of events in the GMEPs.

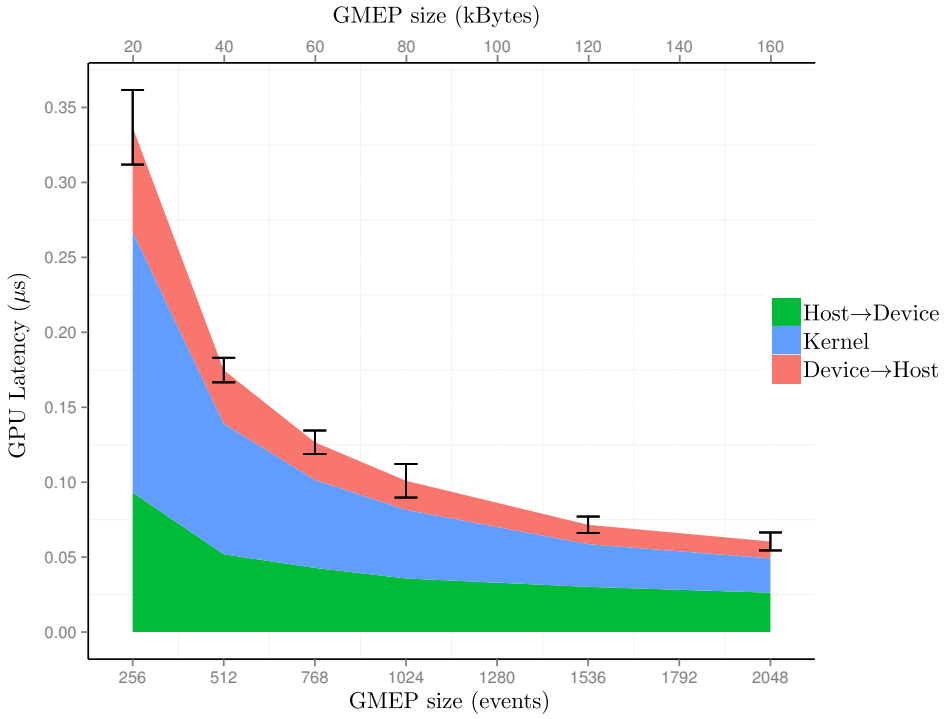


Figure 5.17: The time spent per event in each operation of the GPU processing by the Consumer threads.

5.3.7 Total latency

The last tests are dedicated to show that the entire system developed can produce trigger decisions about the MGPs sent by the TEL62 DAQ board staying within the maximum latency of 1 ms.

For these tests one GbE network link was used, and the size of the GMEP was set to be 256 events.

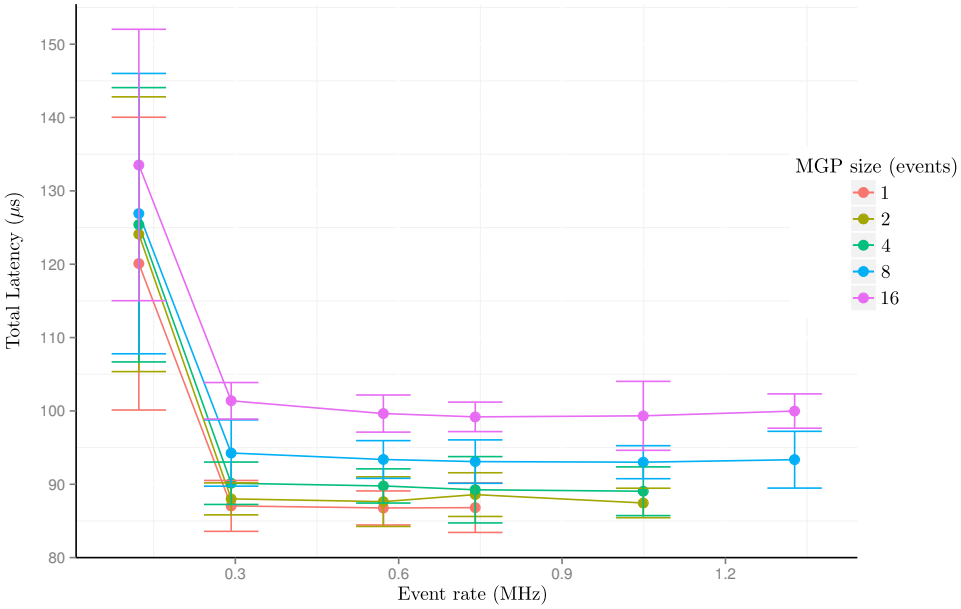


Figure 5.18: Total latency as a function of the incoming event rate.

At first, the time interval between the 256th MGP (containing 16 events) of a GMEP starting to be sent and the time in which the results of the computation are ready in the host memory was measured as a function of the event rate for different MGP sizes (Fig. 5.18). For event rates higher than 300 kHz, the latency is almost constant and stable with fluctuations of the order of few microseconds. The reason why for event rate of 120 kHz both the latency and its stability are way larger than for higher values of the event rate is connected to the frequency at which the condition variables are notified, as explained in section 5.2.2.

However, the total latency has to be referred to the first MGP that is pushed inside the GMEP, and must take into account the collection time $\Delta t_{gathering}$. For this reason the last test was done with the TEL62 sending a signal when the first of 256 events is sent, and the PC sending a signal when the results of the computation of the GMEP are located in the host memory.

This test was executed before and after the deadline-aware mode was activated with a threshold of 700μs (figures 5.19 and 5.20).

The effect of the deadline-aware mode is visible at input rates lower than $\sim$ 400 kHz. In this range the collection time, together with the system latency are higher than the

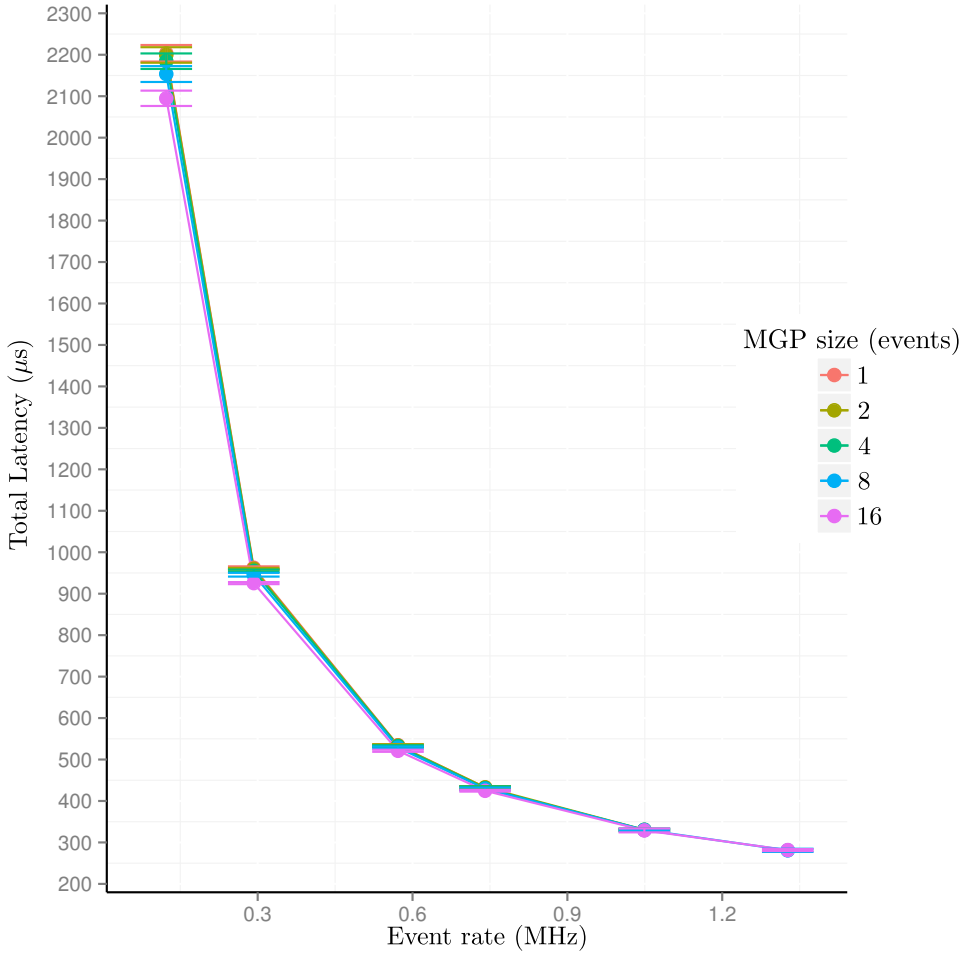


Figure 5.19: Total latency as a function of the incoming event rate referred to the time of first event received.

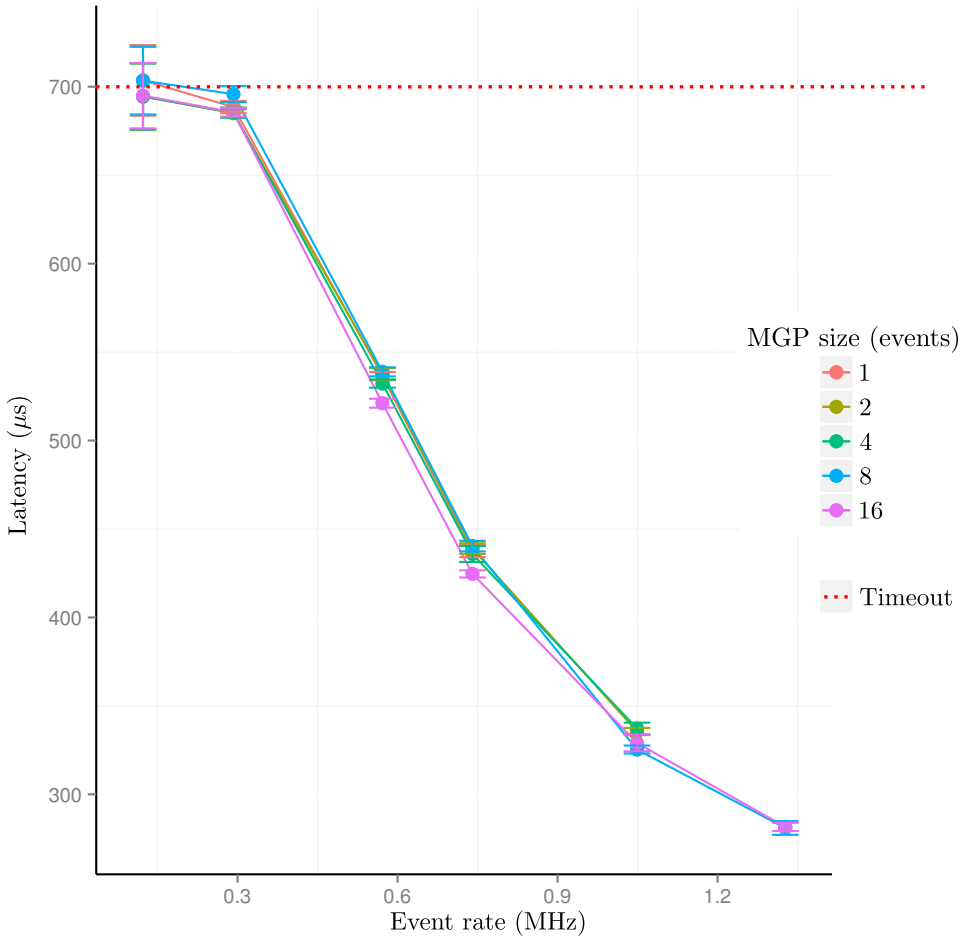


Figure 5.20: Total latency as a function of the incoming event rate referred to the time of first event received when deadline-aware mode is active with a deadline set at 700μs.

set threshold of 700μs. Therefore, deadline-aware mode increases the flexibility of the framework avoiding input data to be lost for lower rates.
Unfortunately I was unable to explore the rate region near 10 MHz input rate, since there was no availability of a four-links ethernet card compatible with PF RING.

5.4 Conclusion and prospects

In this chapter I have shown that an enhancement of a real-time trigger with the requirements compatible with the NA62 experiment ones is possible using off-the-shelf hardware, fast software and drivers. Such improvement in the trigger capabilities will allow to run more complicated algorithms and to exploit all the physics potential of a detector to achieve higher efficiency. Reaching this result will increase the sensitivity to signals of New Physics and to reject background events. Furthermore, the use of an affordable system, developed entirely by the video games industry, that is programmable using high-level software, should not be underestimated.

Chapter 6

$K^+ \to \pi^-\mu^+\mu^+$ events selection

In this chapter I describe a preliminary application of the multi-ring fitting algorithm (section 4.2) and of the parallel trigger framework (Chapter 5) to the selection of $K^+ \to \pi^-\mu^+\mu^+$ event candidates at NA62. In particular, the advantages and the limits of employing the GPU trigger system either directly at Level 0, to enhance the power of this trigger level, or following the standard NA62 Level 0 trigger to add a selection before L1 are studied on a significant use case.

At the moment of this writing the NA62 detector is under construction. For this reason I will refer only to events generated using Monte Carlo simulations. The software suite used for both simulation and reconstruction is the NA62 Framework [54] that is split in two applications: NA62MC and NA62Reconstruction. NA62MC is a software based on Geant4 [55], developed for the simulation of physics processes and the passage of particles through the NA62 detector. NA62Reconstruction is based on ROOT [56], that is an Object Oriented framework for large scale data analysis. It is modularized in libraries for individual subdetectors.

Table 6.1: Branching ratios of the considered processes.

Process	Branching Ratio
$K^+ \to \pi^-\mu^+\mu^+$	$< 1.1 \times 10^{-9}$
$K^+ \to \pi^+\pi^+\pi^-$	$(5.59 \pm 0.04) \times 10^{-2}$
$K^+ \to \pi^+\mu^+\mu^-$	$(9.4 \pm 0.6) \times 10^{-8}$
$K^+ \to \pi^+\pi^0$	$(2.066 \pm 0.008) \times 10^{-1}$
$K^+ \to \pi^0\mu^+\nu_\mu$	$(3.353 \pm 0.034) \times 10^{-2}$
$K^+ \to \pi^0 e^+\nu_e$	$(5.07 \pm 0.04) \times 10^{-2}$

6.1 Geometrical acceptance

6.1.1 Background identification

Using NA62MC, I was able to generate specific kaon decay processes with decay vertex between 105 m and 165 m along the beam line starting from the production target.

Since the purpose of a trigger algorithm is to select the desired event candidates and to discard contingent background events, I identified the possible background processes based on the number of rings that they could produce in the RICH detector, along with their branching ratios [57] (Table 6.1).

The number of rings produced in the RICH detector depends on the number of charged particles that traverse it and on their momentum (Fig. 6.1). Processes with a π^0 in the final state were considered as well because its presence may lead to multiple rings in case of:

- Dalitz decay: $\pi^0 \to e^+e^-\gamma$;
- Inelastic scattering on the experimental apparatus material;
- Photon conversion $\gamma \to e^+e^-$.

In order to determine the geometrical acceptance of the experimental apparatus, 10,000 events for each of the processes above were generated requiring that:

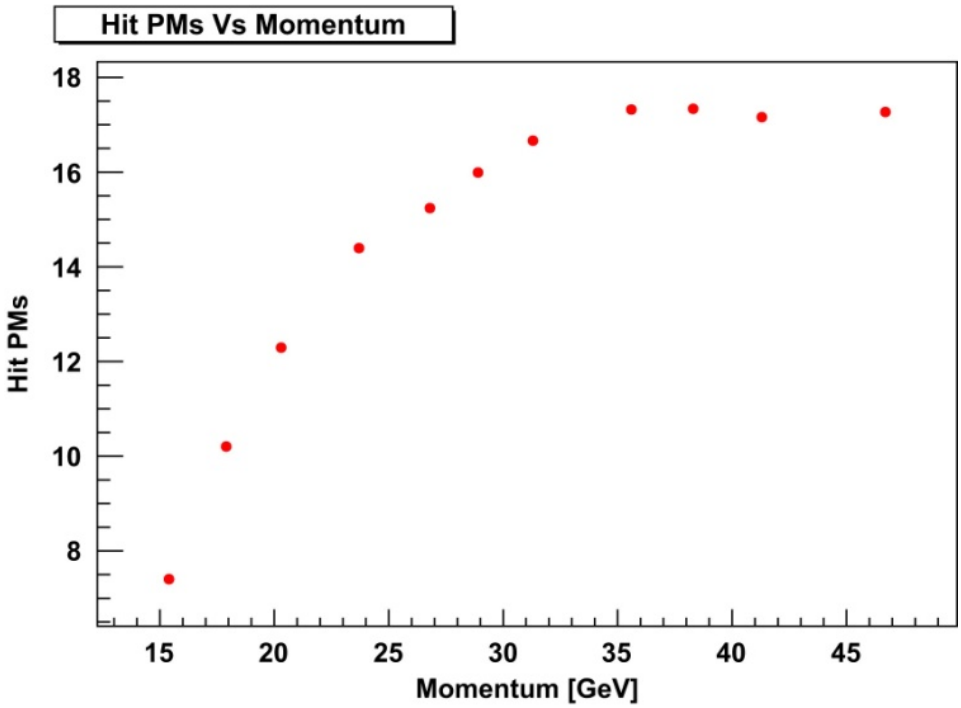


Figure 6.1: Number of hit PMTs as a function of the charged particle momentum.

- the tracks of all the charged particles from the primary vertex traverse completely the RICH detector active volume, excluding the beam pipe;

- the momentum p of the charged particles traversing the RICH is at least 20% higher than the Čerenkov threshold p_T in Neon at room temperature and pressure ($n - 1 = 6.13422636 \times 10^{-5}$).

  The minimum value for the momentum is then given by:

$$p_{min} = 1.2 p_T = 1.2 \frac{m}{\sqrt{n^2 - 1}}. \tag{6.1}$$

  For the charged particles involved in the processes taken into consideration:

$$p_{min}(\pi^{\pm}) = 15.1 \text{ GeV/c} \quad \text{and} \quad \mathrm{p_{min}}(\mu^{\pm}) = 11.4 \text{ GeV/c} \tag{6.2}$$

- the photons produced in π^0 decays enter the Liquid Krypton calorimeter active volume.

Table 6.2: NA62 geometrical acceptance for different decay processes.

Process	**NA62 geometrical acceptance (%)**
$K^+ \to \pi^-\mu^+\mu^+$	34.14 ± 2.36
$K^+ \to \pi^+\pi^+\pi^-$	17.53 ± 3.06
$K^+ \to \pi^+\mu^+\mu^-$	32.23 ± 1.99
$K^+ \to \pi^+\pi^0$	22.37 ± 1.61
$K^+ \to \pi^0\mu^+\nu_\mu$	24.29 ± 1.27
$K^+ \to \pi^0 e^+\nu_e$	19.79 ± 1.39

The results after applying the geometrical acceptance conditions above are shown in Table 6.2.

6.2 Standard Level 0 trigger

Before a GPU implementation of the Level 0 trigger selection conditions are described, a description of the implementation of possible conditions using the NA62 standard Level 0 trigger is discussed.

Since the response time is critical, the Level 0 exploits only the information from the fastest detectors. For this reason the detectors involved in the standard Level 0 trigger decisions here described are the CHOD, the RICH and the MUV3.

CHOD detector

A reconstruction and digitization routine of the CHOD for each scintillator slab does not exist yet. The digitization used divides each of the two planes of the hodoscope (Fig. 1.25) in four quadrants, joining the digitized signal corresponding to all the hits inside the same quadrant.

A conservative cut for the selection of three charged tracks in the CHOD detector consists in requiring the coincidences of (Figures 6.2 and 6.3):

- two corresponding quadrants on the two different planes;
- $n_{\mathrm{CHOD}} \geq 2$ quadrants on the same plane.

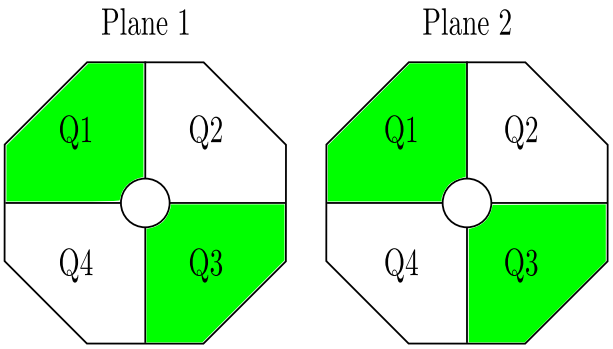


Figure 6.2: Accepted event in CHOD: at least two hits in each plane and the same quadrant in both planes must be hit.

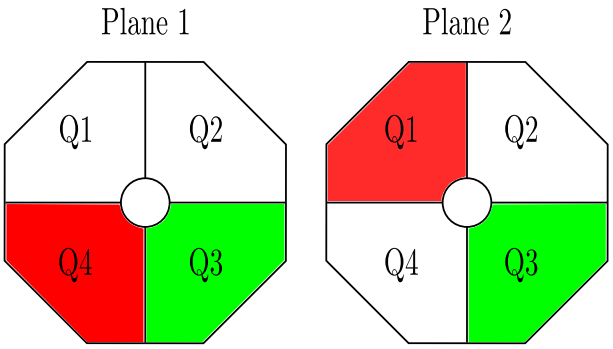


Figure 6.3: Rejected event in CHOD: although there are two hit quadrants in each plane, these do not correspond in both planes.

RICH detector

Without using a multi-ring fitting algorithm for the reconstruction of the rings inside the RICH detector, at the Level 0 trigger stage there is no information about the coordinates of the center and the radius of a ring.

The only information available for the selection is the number of hits in each event. A cut on the minimum number of hits per event allows to select events with more charged particles crossing the RICH detector.

The cut chosen is:

$$25 \leq n_{\mathrm{RICH}} \leq 64 \tag{6.3}$$

that, referring to figure 6.4, would allow to reject a big fraction of the 2-body decay

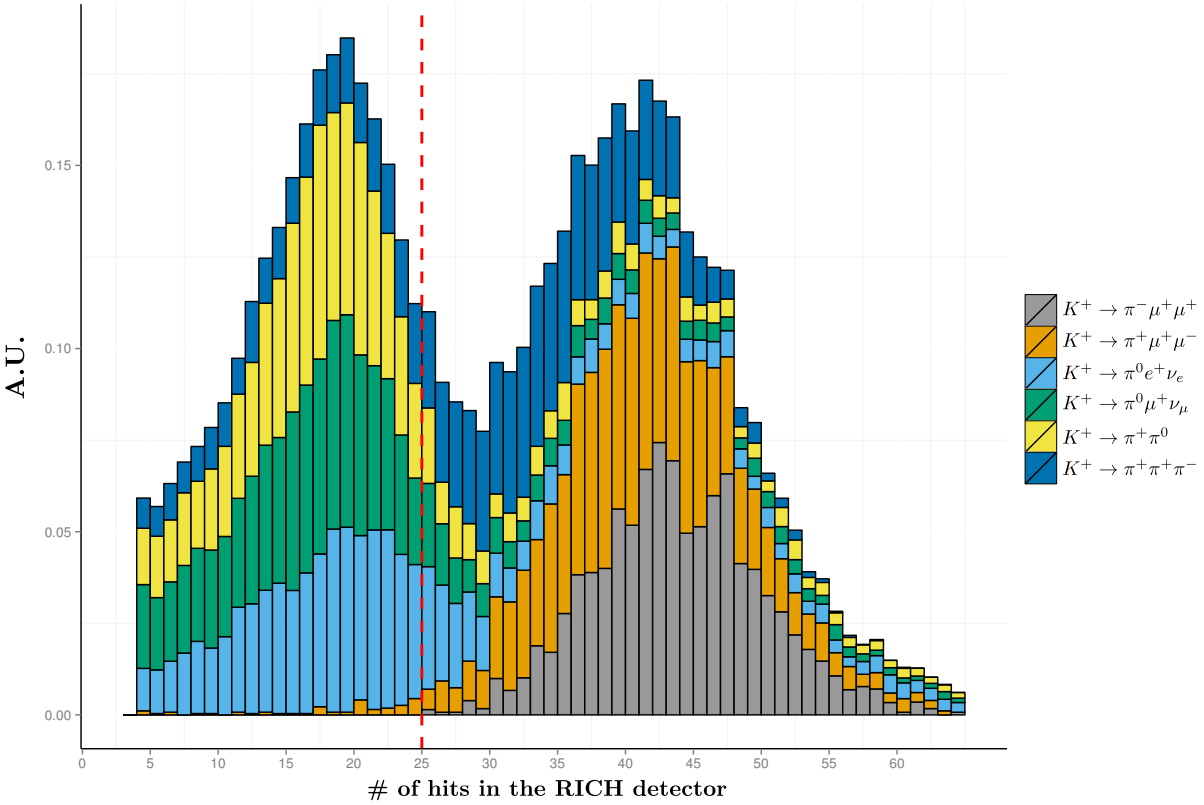


Figure 6.4: Stacked distributions of hits in the RICH detector for simulated events with a possible cut on the RICH multiplicity.

events.

MUV3 detector

The MUV3 is normally used as a veto. For the decay under study it is employed as a positive detector. Since the process $K^+ \to \pi^-\mu^+\mu^+$ has two muons in the final state, a conservative approach consists in requesting a number of hits n_{MUV3} in the MUV3:

$$n_{\mathrm{MUV3}} \geq 2. \tag{6.4}$$

This cut allows to reject much of the background without two muons in the final state. However, a secondary muon could be produced by the decay of a pion, and still traverse the MUV3 detector (Fig. 6.5).

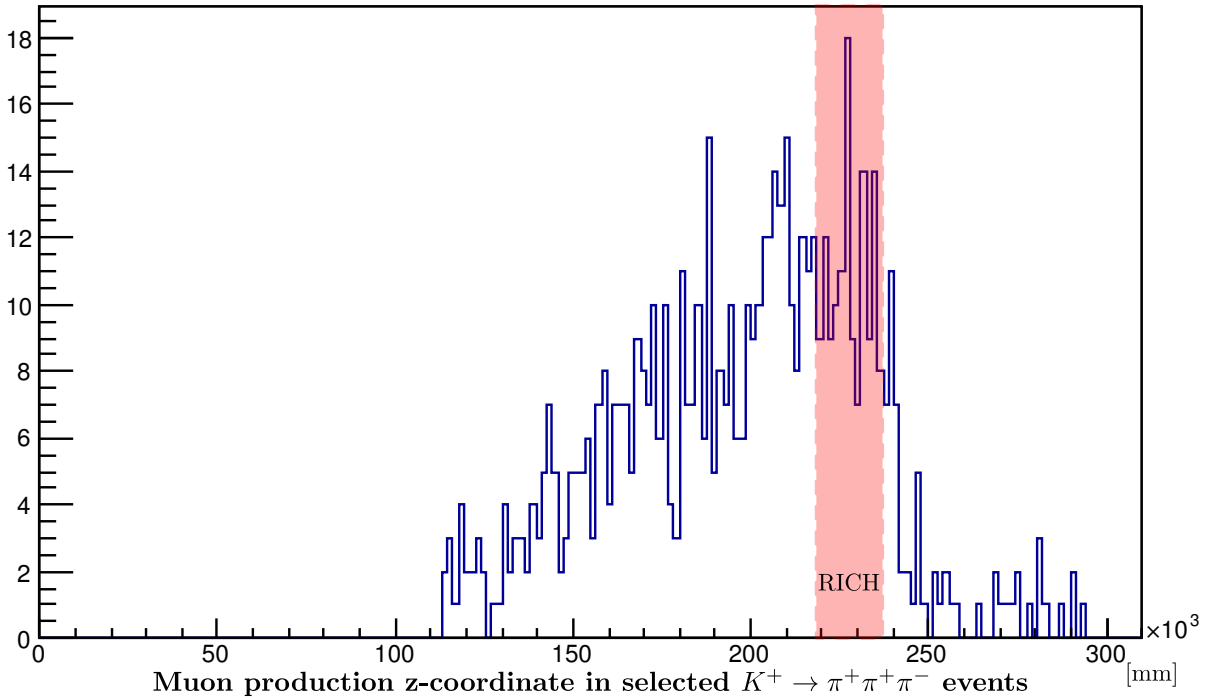


Figure 6.5: z-coordinate of production of muons for simulated $K^+ \to \pi^+\pi^+\pi^-$ events that passed the MUV3 L0 selection $n_{\mathrm{MUV3}} \geq 2$.

6.2.1 Standard Level 0 trigger efficiency

A sample of 10,000 events for each of the processes in Table 6.1 was generated. Only those events that satisfy the Level 0 trigger conditions described above are selected and can be sent to the the next stages of the NA62 trigger chain.

Table 6.3 shows that most part of the background events are rejected by the L0 trigger proposed. The efficiency of the standard Level 0 trigger proposed for the selection of $K^+ \to \pi^-\mu^+\mu^+$ events is $\epsilon = (28.2 \pm 0.4)\%$.

With a finer-grained digitization in the CHOD it could be possible to distinguish between $K^+ \to \pi^+\mu^+\mu^-$ and $K^+ \to \pi^-\mu^+\mu^+$ events already at Level 0, by analyzing the spatial distributions of charged particles.

Table 6.3: Selection after each step of the L0 trigger, efficiency and background rejection.

Process	**Sample**	**CHOD**	**RICH**	**MUV3**	**Trigger efficiency (%)**
$K^+ \to \pi^-\mu^+\mu^+$	10000	3410	2921	2824	28.2±0.4
Process	**Sample**	**CHOD**	**RICH**	**MUV3**	**Background rejection (%)**
$K^+ \to \pi^+\pi^+\pi^-$	10000	1748	1709	21	99.8±0.2
$K^+ \to \pi^+\mu^+\mu^-$	10000	3219	2765	2714	72.8±0.2
$K^+ \to \pi^+\pi^0$	10000	1210	364	1	99.9±0.1
$K^+ \to \pi^0\mu^+\nu_\mu$	10000	1177	389	15	99.8±0.2
$K^+ \to \pi^0 e^+\nu_e$	10000	1114	403	1	99.9 ± 0.1

6.3 GPU based trigger

6.3.1 Four-momentum reconstruction

The knowledge of the coordinates of the centers and the radii of the rings in the RICH, reconstructed by the multi-ring fitting algorithm proposed, allows to set several cuts on kinematics parameters.

Let (x_c, y_c) and R be respectively the coordinates of the center and the radius of the reconstructed Čerenkov ring generated by a charged particle of four-momentum $P = (E, p_x, p_y, p_z)$ traveling through a RICH detector of focal length f filled with a gas of refraction index n.

The velocity β of the particle is directly connected to the radius R, through the Čerenkov angle θ_C, by the following relation:

$$\theta_C = \tan^{-1}\left(\frac{R}{f}\right), \quad \beta = \frac{1}{n\cos(\theta_C)}. \tag{6.5}$$

The computation of the three-momentum is hence straightforward (with $c = 1$):

$$|p| = m\beta\gamma = \frac{m\beta}{\sqrt{1-\beta^2}} \tag{6.6}$$

where m is the mass of the particle.

Furthermore, the direction (θ_x, θ_y) of the particle with respect to the original direction is connected to the coordinates (x_c, y_c) of the center of the ring by the following relations:

$$\theta_x = \tan^{-1}\left(\frac{x_c}{f}\right), \quad \theta_y = \tan^{-1}\left(\frac{y_c}{f}\right). \tag{6.7}$$

The value of θ_x has to be corrected by $\Delta\theta = -17$ mrad that accounts for the angular displacement of the RICH detector with respect to the beam line.

The obtained angles allow to compute the momentum components:

$$\begin{cases} p_x = |p| \sin\theta_x \\ p_y = |p| \sin\theta_y. \end{cases} \tag{6.8}$$

Finally, the energy E is:

$$E = \sqrt{p^2 + m^2}. \tag{6.9}$$

6.3.2 Three rings case

The evaluation above assumes that the mass m of the particle is known.

Let P_K be the four-momentum of the kaon and assume that the multi-ring fitting algorithm has successfully reconstructed three rings of centers (x_1, y_1), (x_2, y_2), (x_3, y_3) and radii R_1, R_2, R_3.

In the hypothesis that the event is a candidate $K^+ \to \pi^-\mu^+\mu^+$ there are three possible mass assignments:

$$(m_1, m_2, m_3) = \begin{cases} (m_\pi, m_\mu, m_\mu) & 1) \\ (m_\mu, m_\pi, m_\mu) & 2) \\ (m_\mu, m_\mu, m_\pi) & 3) \end{cases} \tag{6.10}$$

Each of these mass assignments allows to determine (P_1, P_2, P_3).

Therefore, it is possible to determine one invariant mass for each mass assignment:

$$M_{inv} = \sqrt{\left(\sum_j E^j\right)^2 - \left(\sum_j p_x^j\right)^2 - \left(\sum_j p_y^j\right)^2 - \left(\sum_j p_z^j\right)^2}. \tag{6.11}$$

Finally, the comparison of these three invariant masses with the known value of $m_K = 493.677 \pm 0.013 \mathrm{MeV/c^2}$ could be used as selection condition.

6.3.3 Two rings case

The case in which only two rings have been successfully reconstructed can be treated similarly to the three rings case discussed in section 6.3.2.

As usual, let P_K, be the four-momentum of the kaon and (x_1, y_1), (x_2, y_2) and R_1, R_2 the centers and the radii of the two reconstructed rings.

Assuming that the event is a candidate $K^+ \to \pi^-\mu^+\mu^+$, the three possible mass assignments are:

$$(m_1, m_2) = \begin{cases} (m_\pi, m_\mu) & 1) \\ (m_\mu, m_\pi) & 2) \\ (m_\mu, m_\mu) & 3) \end{cases} \tag{6.12}$$

The mass of the missing particle can be hence evaluated for each of the three cases above:

$$P_3 = P_K - P_1 - P_2 \Rightarrow m_3^2 = P_3^2. \tag{6.13}$$

By comparing these three missing masses with the known values of $m_\pi = 139.57018 \pm 0.00035$ MeV/c^2 and $m_\mu = 105.6583715 \pm 0.0000035$ MeV/c^2, it is possible to decide whether to select or reject the event.

6.3.4 GPU Level 0 cuts

The GPU-based trigger framework developed was run both on unbiased data samples and on events that passed the standard L0 selection.

In order to determine the efficiency of the algorithm, one has to evaluate the target data sample of the algorithm (acceptance of the algorithm). Accordingly, the input data sample for $K^+ \to \pi^-\mu^+\mu^+$ events is selected requiring that each of the particles from the primary vertex generates at least 7 hits in the RICH PMTs. The evaluation of the

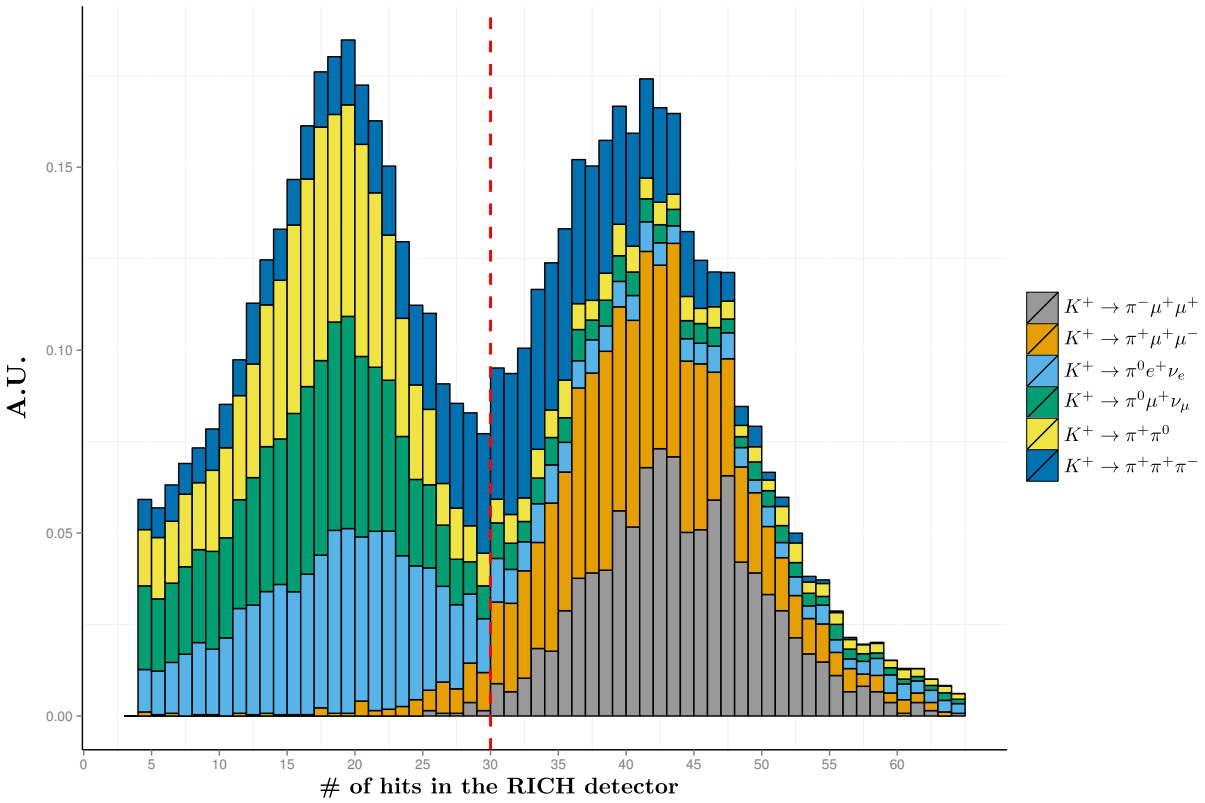


Figure 6.6: Stacked distribution of the number of hits in the NA62 RICH detector for $K^+ \to \pi^-\mu^+\mu^+$ events inside the acceptance of the algorithm compared with other processes.

acceptance of the algorithm for a data sample of 50,000 generated $K^+ \to \pi^-\mu^+\mu^+$ events yields:

$$\epsilon_{AA} = (13.6 \pm 2.8)\%. \tag{6.14}$$

The parameters that can be tuned to improve the efficiency of the algorithm are:

- the minimum number of hits inside a ring, n_{Hmin};
- the minimum distance between the centers and the minimum difference between the radii of two rings, respectively δ_c and δ_R, so that the rings can be assumed to be different.

After several tests, the chosen values are:

$$n_{Hmin} = 6, \quad \delta_c = 10 \text{ mm} \quad \text{and} \quad \delta_{\text{R}} = 10 \text{ mm}. \tag{6.15}$$

The distribution of the number of hit PMTs in the NA62 RICH for events inside the acceptance of the algorithm (Fig. 6.6) is different if compared with the distribution shown in figure 6.4. The number of events with a number of hits in the RICH in the range between 25 and 30 is lower. For the above reason it is possible to set a new cut on the number of hit PMT in the RICH:

$$30 \leq n^{algo}_{\text{RICH}} \leq 64 \tag{6.16}$$

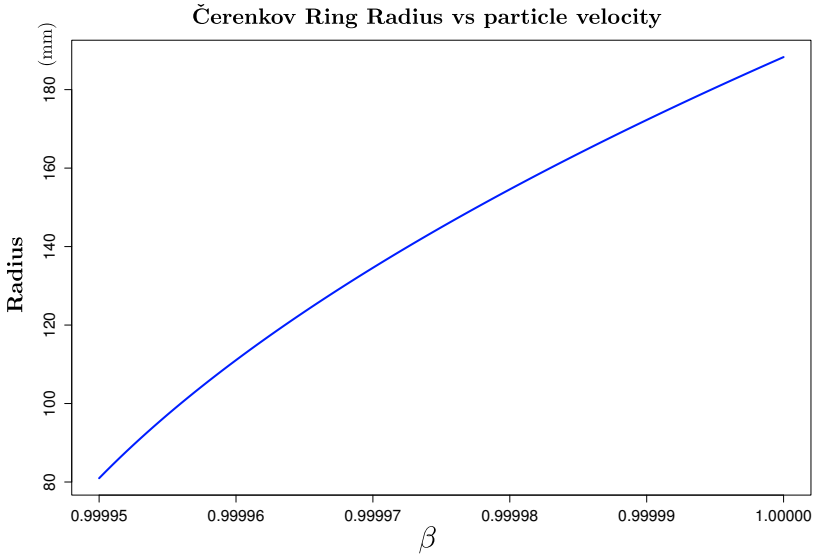


Figure 6.7: Radius length with respect to the particle velocity β, ranging in $\left[\frac{1}{n}, 1\right]$, while traversing the NA62 RICH.

Table 6.4: Initial selection conditions of the GPU L0 trigger.

Process	**Sample**	$30 \leq n_{\mathrm{RICH}}^{algo} \leq 64$	$n_{rings} \geq 2$
$K^+ \to \pi^-\mu^+\mu^+$	10000	9897	3409
$K^+ \to \pi^+\pi^+\pi^-$	10000	4012	1096
$K^+ \to \pi^+\mu^+\mu^-$	10000	5628	2249
$K^+ \to \pi^+\pi^0$	10000	1087	515
$K^+ \to \pi^0\mu^+\nu_\mu$	10000	1058	324
$K^+ \to \pi^0 e^+\nu_e$	10000	1223	395

Next, the limit values of the radius of a Čerenkov ring can be set by considering the behavior of the radius as a function of the particle velocity shown in figure. 6.7 and the lower limit values of the momentum in equation 6.2, yielding:

$$103.52 \text{ mm} \leq \mathrm{R} \leq 186.7 \text{ mm} \tag{6.17}$$

6.3.5 Invariant and missing mass cuts

The algorithm was fed with a data sample containing 10,000 $K^+ \to \pi^-\mu^+\mu^+$ events that passed the algorithm acceptance, i.e. each of the particles from the primary vertex generates at least 7 hits in the RICH PMTs.

The initial data sample is reduced to 9897 events after the cut on n_{RICH}^{algo} and to 3409 events after the ring reconstruction (see Table 6.4). This inefficiency is due to the fact that the algorithm uses only 8 triplets, and most of the time it is not able to fit a ring with a radius that satisfies the condition 6.17. In fact, although the triplets are formed with the distance between the points above a threshold (see section 4.2.2), it is possible that the points are almost aligned along one axis leading a to reconstructed radius that is much bigger than the upper threshold of 186.7 mm (equation 6.17).

The evaluation of the invariant mass for events reconstructed with three rings gives the distribution shown in Figure 6.8.

I chose to apply a generic cut for the invariant mass, since the distribution for background events is not clearly localized.

$$M_K - 100 \text{ MeV/c}^2 \leq \mathrm{M_{inv}} \leq \mathrm{M_K} + 100 \text{ MeV/c}^2 \tag{6.18}$$

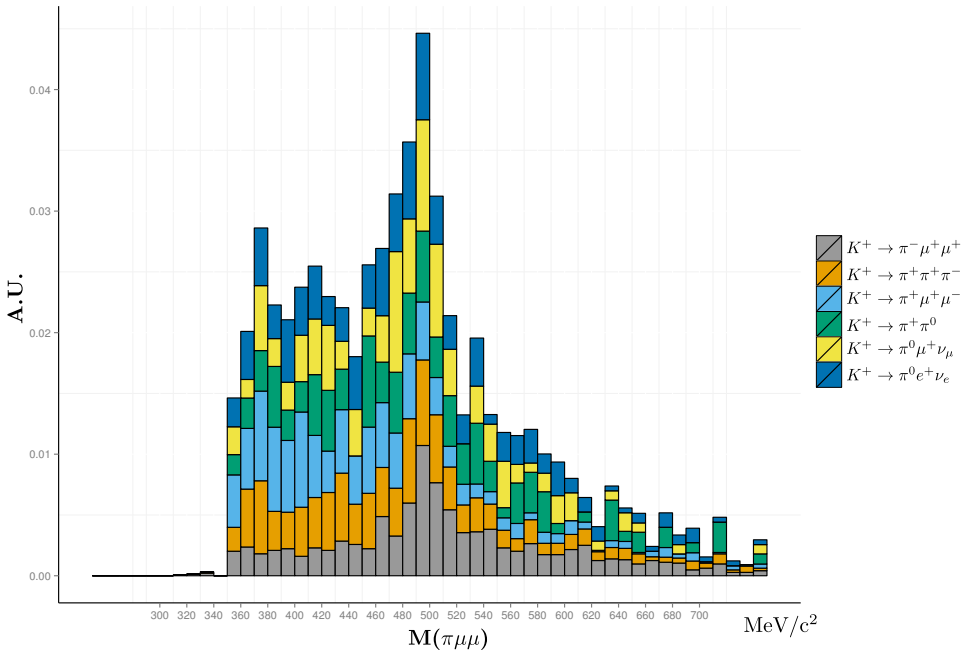


Figure 6.8: Invariant-mass distribution of $\pi^+\mu^-\mu^-$ event candidates when the multi-ring fitting algorithm reconstructs three rings.

When the multi-ring fitting algorithm finds two rings and the squared missing mass is positive, the reconstructed difference between the missing mass M_{miss} and the mass of the expected missing particle M_{π^-/μ^+} distributes like shown in figure 6.9. The cut chosen to reject a fraction of the $K^+ \to \pi^+\pi^+\pi^-$ background events is:

$$M_{miss} - M_{\pi^+/\mu^-} \leq 20 \text{ MeV/c}^2 \tag{6.19}$$

The resulting GPU trigger efficiency after applying these two cuts is (Table 6.5)

$$\epsilon_{GPU_{L0}} = (12.4 \pm 1.5)\%. \tag{6.20}$$

The trigger efficiency is stable with respect to changes in the tuning parameters and its low value is mainly given by the low number of triplets used and the method used to choose their points.

An example of $K^+ \to \pi^-\mu^+\mu^+$ event that is reconstructed correctly using the missing mass information by the multi-ring fitting algorithm is shown in figure 6.10. The two fitted rings are clearly visible, however, although one of the requirements is that all

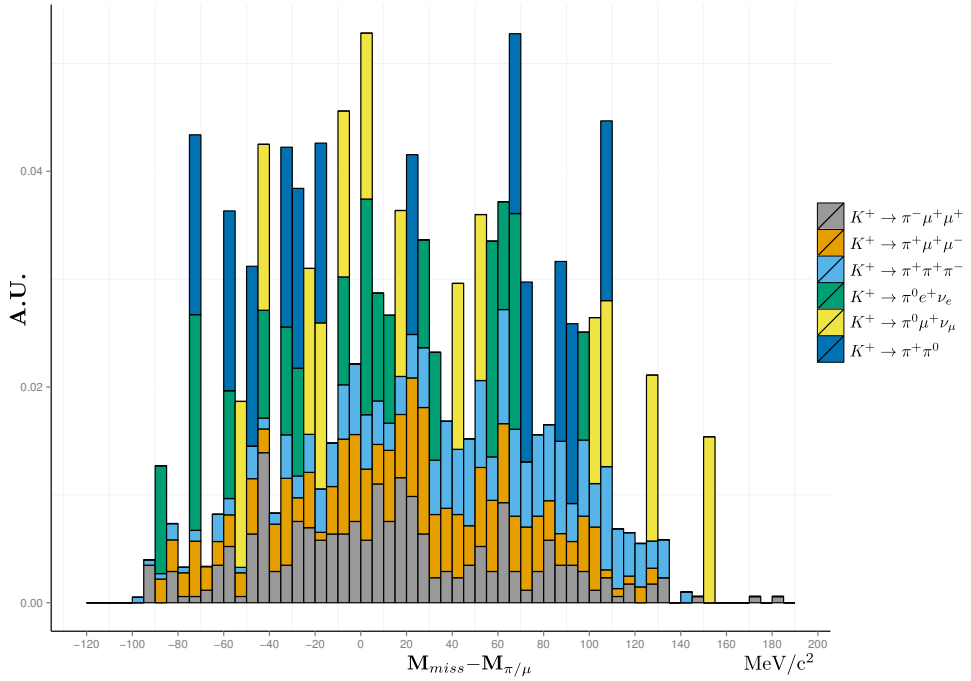


Figure 6.9: Distribution of the difference between the Missing-mass and the mass of the missing particle if two rings are reconstructed.

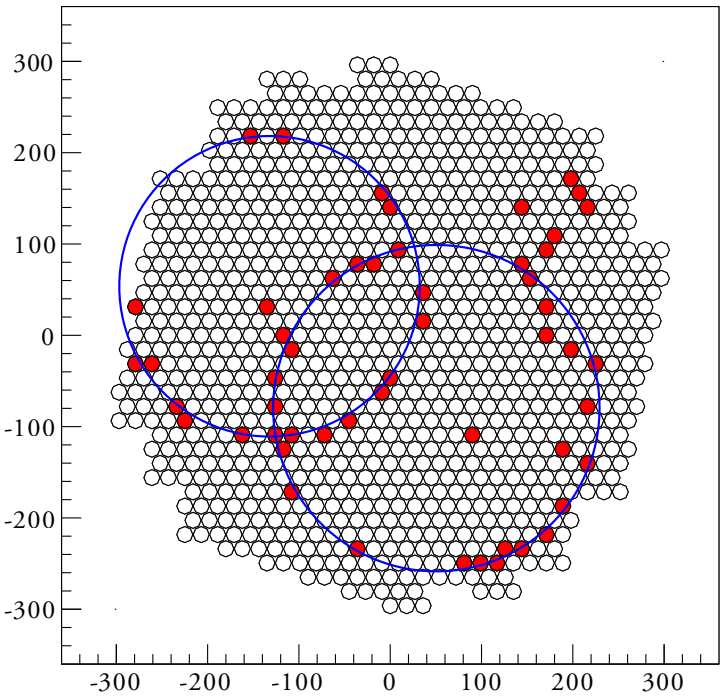


Figure 6.10: $K^+ \to \pi^-\mu^+\mu^+$ event that passed the GPU trigger with two rings reconstructed and missing mass compatible with a pion or a muon.

the three particles have hit at least 7 PMTs, these are not disposed in a clearly visible circular shape, and the third ring could not be reconstructed correctly.

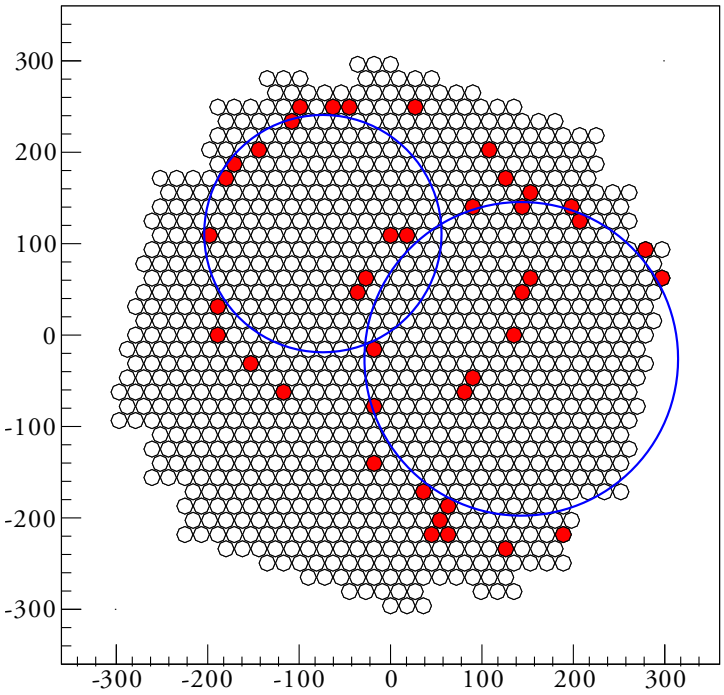


Figure 6.11: $K^+ \to \pi^0\mu^+\nu_\mu$ event that passed the GPU trigger with two rings reconstructed for the multi-ring algorithm inefficiency. The points of the triplets corresponding to y_{max}, x_{min} and v_{max} are aligned and lead to a reconstructed ring radius larger than the upper limit. These rings are thus rejected.

The same reasons that make the signal efficiency low, in some cases can also decrease the background rejection. Figure 6.11 shows a $K_{\mu 3}$ event with the ring on the left-hand side that is mis-reconstructed. In this case, the mis-reconstruction is due to the alignment of the elements of the y_{max}, x_{min} and v_{max} triplets. The algorithm reconstructs for these triplets rings with a radius that is larger than the upper limit, thus leading to the rejection of the rings.

The possibility to employ the GPU-based trigger system on top of the standard Level 0 trigger (section 6.2) has been explored as well. The multi-ring algorithm was therefore fed with the events after they have passed the standard L0 trigger (Table 6.6) yielding a value of the efficiency:

$$\epsilon_{GPU_{L0.5}} = (8.6 \pm 0.7)\%. \tag{6.21}$$

Table 6.5: Invariant and missing mass selection conditions of the GPU-based Level 0 trigger.

Process	**Sample**	$n_{rings} \geq 2$	**Mass cuts**	**GPU Trigger efficiency (%)**
$K^+ \to \pi^-\mu^+\mu^+$	10000	3409	1236	12.4±1.5
Process	**Sample**	$n_{rings} \geq 2$	**Mass cuts**	**Background rejection (%)**
$K^+ \to \pi^+\pi^+\pi^-$	10000	724	207	97.0±0.8
$K^+ \to \pi^+\mu^+\mu^-$	10000	477	153	98.5±0.9
$K^+ \to \pi^+\pi^0$	10000	171	40	99.6±0.4
$K^+ \to \pi^0\mu^+\nu_\mu$	10000	381	128	98.7±0.5
$K^+ \to \pi^0 e^+\nu_e$	10000	216	40	99.6±0.5

Table 6.6: Invariant and missing mass selection conditions of the GPU L0 trigger after the selection applied by the standard L0.

Process	**Sample**	$n_{rings} \geq 2$	**Mass cuts**	**GPU Trigger efficiency (%)**
$K^+ \to \pi^-\mu^+\mu^+$	10000	2604	864	8.6±0.7
Process	**Sample**	$n_{rings} \geq 2$	**Mass cuts**	**Background rejection (%)**
$K^+ \to \pi^+\pi^+\pi^-$	10000	66	13	99.9±0.2
$K^+ \to \pi^+\mu^+\mu^-$	10000	336	109	98.9±0.2
$K^+ \to \pi^+\pi^0$	10000	4	1	99.9±0.2
$K^+ \to \pi^0\mu^+\nu_\mu$	10000	13	2	99.9±0.2
$K^+ \to \pi^0 e^+\nu_e$	10000	0	0	100±0.1

The efficiency could be improved by increasing the number of triplets used or by forming doublets and evaluating a number of rings 64 times higher at most.

However the number of possible rings evaluated is limited by the latency, that has to be evaluated. This performance assessment was made with the following hardware:

- Computer:
  - Host CPU: Intel Xeon E5-2620 2.00 GHz (12 physical cores divided on two sockets)
  - Host Network Interface: Intel I350-T2
  - Device GPU: NVDIA Tesla Kepler K20
  - Communication bus: PCI Express Gen. 2

- Readout board: NA62 TEL62 DAQ Board
- Oscilloscope: LeCroy WaveRunner 104MXi

The experimental approach is the same used in section 5.3: the TEL62 DAQ board sends a signal when the 256th MGP of a GMEP starts to be sent, while the software trigger framework sends a signal on the LPT port when the results of the computation are ready in the host memory. This time interval has been measured 5,000 times in a continous stream with the oscilloscope like shown in Figure 6.12 giving a value of the latency for $K^+ \to \pi^-\mu^+\mu^+$ of:

$$T(K^+ \to \pi^-\mu^+\mu^+) = (435 \pm 4)\ \mu s \tag{6.22}$$

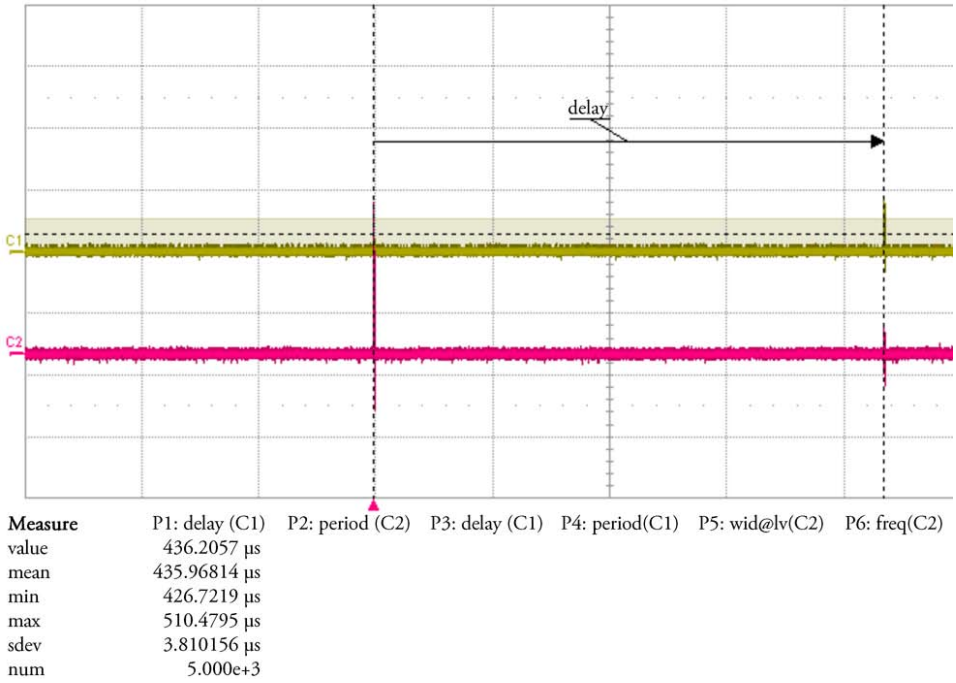


Figure 6.12: Picture of the measurement of the time interval between the TEL62 and the computer signals on the oscilloscope.

The independence of the latency on the process sent in input was also verified by measuring the latency for $K^+ \to \pi^+\pi^0$ events (Figure 6.13). This measurement yields a a value of the latency of

$$T(K^+ \to \pi^+\pi^0) = (433 \pm 4)\ \mu s \tag{6.23}$$

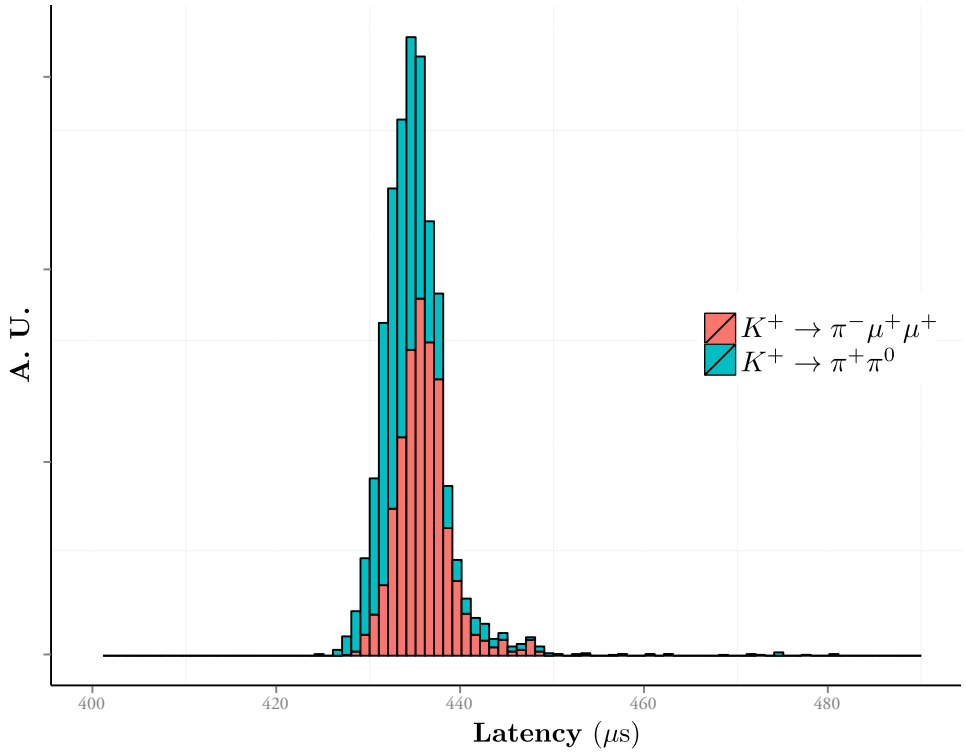


Figure 6.13: Latency distribution for the trigger.

This value, 2 μs lower than the latency of events, accounts for the fact that GMPs filled with $K^+ \to \pi^+\pi^0$ events contain on average a smaller number of hits per event, thus making the communication latency smaller.

6.3.6 Summary and improvements

Trigger efficiency is mainly limited by the quantity and the quality of the triplets. The values of the signal selection efficiency before and after the standard L0 trigger are promising considering that only a small sample of rings per each event is evaluated over the thousands possible ones. Furthermore, without an ad-hoc trigger implementation no $K^+ \to \pi^-\mu^+\mu^+$ events are expected to be selected.

However, a value of 435 μs for the latency does not allow to increase the number of triplets used without exceeding the maximum latency. Therefore more than one GPU must be used to distribute the workload and make possible a more varied choice of the points inside a triplet. Up to four GPUs can be installed on the same machine allowing a fourfold bigger variety of hits that can be fed to the ring-fitting algorithm and also to smooth the constraints on the triplet forming (e.g. the minimum distance between the points of a triplet could be decreased).

Conclusion

The work described in this thesis aims at integrating a software solution based on GPUs into the central Level 0 trigger processor for computing trigger primitives at the NA62 experiment.

The development of such a system required a feasibility study both on the algorithmic and on the data transportation sides.

Studies performed on the algorithmic side, as described in Chapters 3 and 4, have laid the foundations for the implementation of a fast, seedless multi-ring algorithm for the reconstruction Čerenkov light rings in the NA62 RICH detector.

A parallel trigger framework has been developed to host the ring-fitting algorithm (Chapter 5). It represents the first implementation of a software-based solution for real-time event selection in High Energy Physics. A careful assessment of the real-time performance has been carried out in order to verify that it is viable to use such a system in the NA62 Level 0 trigger. The results of the tests are promising and have allowed setting limits on parameters (e.g. data-frames sizes) so as to satisfy the latency and throughput requirements of the NA62 RICH detector.

Finally, a preliminary study showing the application of the above framework and algorithm to the search of the lepton flavor-violating process $K^+ \to \pi^-\mu^+\mu^+$ is described in Chapter 6. A kinematic analysis has been performed both on signal and background simulated events in order to provide early trigger decisions. The results obtained are promising and show the way to future improvements and implementations.

Chapter 7

Bibliography

[1] NA62 Collaboration, *NA62: Technical Design Document*, `https://cds.cern.ch/record/1404985`, 2010.

[2] J. Brod, et al., *Two-Loop Electroweak Corrections for the $K^{\pm} \to \pi^{\pm}\nu\bar{\nu}$ Decays*, arXiv:1009.0947, 2011.

[3] N. Cabibbo, *Unitary Symmetry and Leptonic Decays*, PRL 10, pp. 531–533, 1963.

[4] M. Kobayashi, T. Maskawa, *CP Violation in the Renormalizable Theory of Weak Interaction*. Prog. Theor. Phys. 49, pp. 652–657, 1973.

[5] K. Nakamura, et al., *Review of Particle Physics*, Journal of Physics G: Nuclear and Particle Physics 37.7A, 2010.

[6] F. Ambrosino et al., *Measurement of the absolute branching ratio for the $K^+ \to \mu^+\nu(\gamma)$ decay with the KLOE detector* Phys. Lett. B 632, 76, 2006.

[7] L. Wolfenstein, *Parametrization of the Kobayashi-Maskawa Matrix*, Physical Review Letters 51 (21), 1945.

[8] A. Buras, *Weak Hamiltonian, CP Violation and Rare Decays*, eprint arXiv:hep-ph/9806471, 1998.

[9] G. Isidori, F. Mescia, C. Smith, *Light-quark loops in $K \to \pi\nu\bar{\nu}$*, Nucl. Phys. B 718, pp. 319–338, 2005.

[10] G. Buchalla, A. Buras, M. Lautenbacher. *Weak Decays Beyond Leading Logarithms*, Rev.Mod.Phys.68:1125-1144, 1996.

[11] CDF Collaboration (Abulencia, A. et al.), *Observation of $B_s^0 - \bar{B}_s^0$ Oscillations*, Phys. Rev. Lett. 97, 242003 hep-ex/0609040, FERMILAB-PUB-06-344-E, 2006.

[12] LHCb Collaboration (R. Aaij, et al.), *Measurement of the $B_s^0 - \bar{B}_s^0$ oscillation frequency Δm_s in $B_s^0 \to D_s^-(3)\pi$*, Phys. Lett. B 709, pp. 177–184, 2012.

[13] J. Laiho, et al., *Lattice QCD inputs to the CKM unitarity triangle analysis*, Phys. Rev. D 81, 2010.

[14] G. Buchalla and A. J. Buras, *The Rare Decays $K^+ \to \pi^+\nu\bar{\nu}$ and $K_L \to \mu^+\mu^-$ Beyond Leading Logarithms*,Nucl. Phys. B412 106, 1994.

[15] A. Sher, et al., *Improved upper limit on the decay $K^+ \to \pi^+\mu^+e^-$*, Phys. Rev. D 72, 2005.

[16] R. Appel, et al., *Search for Lepton Flavor Violation in K^+ Decays*, PRL 85, 2877, 2000.

[17] J. Batley, et al., *NA48 Collaboration*, Phys. Lett. B 697, 107, 2011.

[18] A. Diamant-Berger, et al., *Study of some rare decays of the K^+ meson*, Phys. Lett. B 62, 1976.

[19] D. Ambrose et al., *New Limit on Muon and Electron Lepton Number Violation from $K_L^0 \to \mu^\pm e^\mp$ Decay* Phys. Rev. Lett. 81, 5734, 1998.

[20] KTeV Collaboration, E. Abouzaid, et al., Phys. Rev. Lett. 100, 2008.

[21] M.C. Gonzalez-Garcia, M. Maltoni, *Phenomenology with Massive Neutrinos*, Physics Reports 460.1, 1-129, 2008.

[22] R. Barbieri, L. Hall, A. Strumia, *Violations of lepton flavour and CP in supersymmetric unified theories*, Nucl.Phys.B445:219-251, 1995.

[23] T. Appelquist, et al., *Flavor-changing processes in extended technicolor*, Phys.Rev.D 53, 242, 1996.

[24] S. Choudhury, et al., *Lepton flavour violation in The Little Higgs model*, Phys.Rev.D 75 055011, 2007.

[25] A. Boyarsky, O. Ruchayskiy, M. Shaposhnikov, *The role of sterile neutrinos in cosmology and astrophysics*, arXiv:0901.0011, 2008.

[26] J. Schechter, J. Valle , *Neutrinoless double beta decay in $SU(2) \times U(1)$ theories*, Phys. Rev. D, 25:11, 1982.

[27] R. K. Kutschke, *The Mu2e Experiment at Fermilab*, arXiv:1112.0242, 2011.

[28] R. Cahn, H. Harari, *Bounds on the masses of neutral generation-changing gauge bosons*, Nucl. Phys. B 176 135, 1980.

[29] G. Haefeli *et al.*, *The LHCb DAQ interface board TELL1*, Nucl. Instrum. Meth. A 560, 494, 2006.

[30] M. Moulson, *Forbidden Kaon and Pion Decays in NA62*, Proceedings of the 2013 Kaon Physics International Conference (KAON '13), Ann Arbor, MI, USA 29 April-1 May 2013.

[31] NA48/2 Collaboration, *New measurement of the $K^+ \to \pi^-\mu^+\mu^+$decay*, Physics Letters B, Volume 697, Issue 2, Pages 95-172, 2011.

[32] A. Romano [for the NA62 Collaboration], Astroparticle, Particle, Space Physics and Detectors for Physics Applications, World Scientific, 895, 2012.

[33] *The NIST Definition of Cloud Computing.* National Institute of Standards and Technology, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[34] J. Dongarra, *Performance of Various Computers Using Standard Linear Equations Software*, http://www.netlib.org/, 2013.

[35] http://www.top500.org/

[36] G. Moore, *Cramming more components onto integrated circuits*, Proceedings of the IEEE, vol.86, no.1, pp.82,85, 1998.

[37] A.P. Chandrakasan, et al., *Optimizing power using transformations*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.14, no.1, pp.12-31, 1995.

[38] L. F. Menabrea, *Sketch of The Analytical Engine Invented by Charles Babbage*, Bibliothèque Universelle de Genève, No. 82, October, 1842.

[39] M. J. Flynn, *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput. C–21, 1972.

[40] G. Amdahl, *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, AFIPS Conference Proceedings (30): 483–485, 1967.

[41] J. L. Gustafson, *Reevaluating Amdahl's Law*, Communications of the ACM, **vol. 31**, 532-533, 1988.

[42] M. Herlihy, J.E.B. Moss, *Transactional memory: Architectural support for lock-free data structures*, Proceedings of the 20th International Symposium on Computer Architecture, 289–300, 1993.

[43] Khronos Group, *The open standard for parallel programming of heterogeneous systems*, `http://www.khronos.org/opencl/`

[44] *The OpenMP® API specification for parallel programming*, `http://openmp.org/wp/resources/`

[45] *Threading Building Blocks*, `https://www.threadingbuildingblocks.org/`

[46] G. Haefeli *et al.*, *The LHCb DAQ interface board TELL1*, Nucl. Instrum. Meth. A **560** (2006) 494.

[47] *Timing, Trigger and Control (TTC) Systems for the LHC*, `http://ttc.web.cern.ch/TTC/intro.html`

[48] J. Christiansen, *High Performance Time to Digital Converter* `http://cds.cern.ch/record/1067476/files/cer-002723234.pdf`, Version 2.2, 2004.

[49] M. Harris, *Optimizing Parallel Reduction in CUDA*, `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf`

[50] F. Pantaleo, et al., *Real-time use of GPUs in NA62 experiment*, 13th International Workshop on Cellular Nanoscale Networks and Their Applications (CNNA), 2012.

[51] J. F. Crawford, *A Noniterative Method For Fitting Circular Arcs To Measured Points*, Nucl. Instrum. Meth. **211** 223-225, 1993.

[52] L. Deri, *Improving Passive Packet Capture: Beyond Device Polling*, Proceedings of SANE 2004

[53] *IEEE 1588 Standard for A Precision Clock Synchronization Protocol for Networked Measurement and Control Systems.* National Institute of Standards and Technology, `http://www.nist.gov/el/isd/ieee/ieee1588.cfm`

[54] NA62 Framework, `http://sergiant.web.cern.ch/sergiant/NA62FW/html/`.

[55] S. Agostinelli, et al., *Geant4 - a simulation toolkit*, Nuclear Instruments and Methods in Physics Research A 506, 250-303, 2003.

[56] R. Brun, F. Rademakers, *ROOT — An object oriented data analysis framework*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment Volume 389, Issues 1–2, April 1997.

[57] K. Nakamura et al. (Particle Data Group), *Review of Particle Physics*, J. Phys. G, 37:075021, 2010.

Acknowledgments

I would like to express my sincere gratitude to my supervisor Prof. Marco Sozzi, who, with passion, indefatigable diligence and deep knowledge of the subjects, has managed to organize my "thread-unsafe" streams of thought.

I gratefully thank Dr. Vincenzo Innocente for always supporting my ideas and for the experience he has helped me gain by working with him at CERN.

I would like to express my special appreciation and thanks to Dr. Augusto Ceccucci, Dr. John Harvey and Dr. Pere Mato for making CERN feel like home.

I am very grateful to Dr. Gianluca Lamanna for his scientific advice, knowledge and many discussions about the Two Chief World Systems.

I will forever be thankful to Dr. Alfio Lazzaro for always being a good friend and for teaching me how to follow The Code in the first place.

I would like to show my greatest appreciation to Dr. Benedikt Hegner. I feel motivated and encouraged every time I explain my doubts and problems to him.

Completing this work would have been much more difficult were it not for the support and friendship provided by Dr. Danilo Piparo.

I would also like to thank Dr. Massimo Lamanna for his assistance in interpreting 15-years-old Perl 3 code.

A very special thanks goes out to Dr. Domenico Giordano, without whose encouragement and motivation my academic career would have probably followed a different path.

I would like to warmly thank those with whom for different reasons I have worked: Dr. Roberto Piandani, who was my interface with the TEL62 DAQ Board, Dr. Antonio

Cassese and Dr. Bruno Angelucci without whose knowledge and assistance this study would not have been successful.

I wish to express my love and gratitude to my beloved family for their understanding and endless love and support throughout the duration of my studies.

A very special acknowledgement goes to my girlfriend Isabel, who loved and supported me during these years, and made me feel like anything was possible. I love you, Isabel.

I would like to express my deepest gratitude to Nicola for being like a brother to me for almost a year in Geneva.

Gabriel deserves a special mention for being a generous and wonderful friend and for his precious advice during the preparation of this work. Together with Petya, we had the most productive coffee breaks in human history.

The nights during the last six months in Pisa would have been so boring without Romualdo, Stefano and the hedgehogs. Thank you, this work would have been much harder to complete without you.

I have special friends to thank for making the time spent in Geneva so wonderful: Attilio, Chiara, Claire, Dario, Federica, Jonas, Noemi, Robert, Salvatore, Simone, Sara & Silvio.

I would like to thank my friends: Adele, Alberto, Anna, Elena, Emilio, Filippo, Giacomo, Katia, Ilenia, Laura, Lele, Letizia, Marco, Niccolo', Nino, Roberto and Stefania, for making Pisa feel like home.

I wish to express my gratitude to my best friends Anna, Felice, Giuseppe, Marco, Mary and Vito. Although we meet only once a year our friendship strengthens day by day.

Finally, I would like to thank Cesare, Emilio, Giacomo, Luca, and Tommaso for almost ten years of conspiracy and many secrets encoded in this thesis text.