ENRICO CALORE

569811-IF

OPTIMIZATION OF THE AGATA PULSE SHAPE ANALYSIS ALGORITHM USING GRAPHICS PROCESSING UNITS

Ottimizzazione dell'algoritmo per l'analisi di forma di impulso per il rivelatore AGATA tramite l'uso di processori grafici

TESI DI LAUREA SPECIALISTICA

Relatore: Chiar.mo Prof. C. Ferrari[1]
Co-relatore: Dr. D. Bazzacco[2]
Co-relatore: Dr. F. Recchia[2]

Università degli Studi di Padova
Facoltà di Ingegneria
Dipartimento di Ingegneria dell'Informazione

Anno Accademico 2009/2010

1 Università Degli Studi di Padova, Dip. di Ingegneria dell'Informazione
2 Istituto Nazionale di Fisica Nucleare, Sezione di Padova

INTRODUZIONE

La spettroscopia nucleare γ ad alta risoluzione è uno degli strumenti più potenti e sensibili per lo studio dei nuclei atomici. A partire dagli anni ottanta sono stati utilizzati array di rivelatori al germanio con schermi anti-Compton capaci di risoluzioni sempre più elevate, permettendo la scoperta di nuovi fenomeni nucleari.

L'ulteriore sviluppo che è stato proposto negli ultimi anni si basa sulla possibilità di determinare la posizione e l'energia depositata dalla singola interazione di un fotone all'interno di un cristallo di germanio, e sulla capacità di ricostruire la sequenza delle interazioni attraverso sofisticati algoritmi di analisi dei segnali prodotti.

Attualmente due progetti mirano alla costruzione di un array di rivelatori di nuova generazione basato su questo paradigma: AGATA in Europa e GRETA negli Stati Uniti.

AGATA è attualmente in fase di sviluppo presso i Laboratori Nazionali di Legnaro dell'INFN e non essendo ancora costituito da tutti i rivelatori previsti è attualmente denominato *AGATA Demonstrator*.

A differenza degli array di rivelatori utilizzati in passato, la grande quantità di dati prodotti dal *Dimostratore* ed a maggior ragione in futuro da AGATA, non consente l'archiviazione degli stessi su supporti abbastanza economici e versatili da permetterne l'analisi successivamente all'acquisizione.

Per questo motivo il sistema di acquisizione dati utilizzato deve effettuare un'elaborazione a *run-time* dei dati acquisiti ed in seguito l'archiviazione dei soli risultati prodotti. Conseguentemente, la velocità del sistema di acquisizione/analisi dei dati risulta essere un possibile fattore limitante sulla massima frequenza di lavoro dell'array stesso.

Una delle parti piú critiche ed impegnative dal punto di vista computazionale del sistema di acquisizione è l'analisi di forma di impulso. Per evitare che questa parte dell'analisi sia il collo di bottiglia del sistema, questo lavoro di tesi affronta i problemi di ottimizzazione dell'algoritmo stesso, fornendone un'implementazione in grado di sfruttare la potenza di calcolo contenuta all'interno delle moderne schede grafiche.

INTRODUCTION

High-resolution γ-ray spectroscopy is one of the most powerful and sensitive tools to investigate Nuclear Structure. Since the eighties various nuclear phenomena has been discovered thanks to the use of high-purity germanium detectors arrays with Compton suppression, capables of higher resolution than before.

The further improvement which has been proposed since the mid-nineties relies on the possibility to determine the position and the energy deposition of the individual interaction points of a photon within a germanium crystal and on the capability to reconstruct the photon scattering sequence through data analysis algorithms.

Presently, two major projects aim at the construction of an array of germanium detectors based on the pulse shape analysis and γ-ray tracking techniques, namely GRETA in the USA and AGATA in Europe.

AGATA is currently under development at the Legnaro National Laboratories of the Italian National Institute of Nuclear Physics and since is not yet composed of all the planned detectors, is called the *AGATA Demonstrator*.

The Demonstrator (and moreover the final AGATA array) because of the great amount of data it produces, does not allow to store the acquired information on affordable and small devices to let the physicists analyze it after the acquisition, as has been done in the past.

For this reason the data acquisition system has to analyze the acquired data at run-time and eventually has to store only the results. Consequently the speed of the data acquisition and analysis system becomes an important specification of the whole project that may introduce a limit on the acquisition frequency of the array.

One of the operation requesting more computing power is the pulse shape analysis and to prevent it to be the bottleneck of the whole system, this dissertation work will investigate on the possibility to speed up the PSA algorithm execution, giving an implementation capable to run on modern GPUs.

CONTENTS

1 THE AGATA PROJECT

AGATA (*Advanced GAmma Tracking Array*) is an array of 180 highly-segmented HPGe (*High-Purity Germanium*) detectors. At present, this is the most ambitious joint project of the European Nuclear Structure community to design and construct a spectrometer with unprecedented efficiency and spectral resolution [6, 7, 43].

The first prototype of the AGATA array consisting of a reduced number of detectors (15) has been put into operation at the beginning of this year (2010) at Legnaro National Laboratories (LNL) of the Italian National Institute of Nuclear Physics (INFN) and is called the AGATA *Demonstrator*.

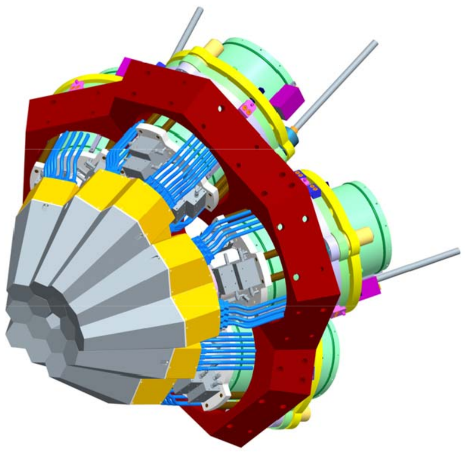


Figure 1: Sketch of the AGATA Demonstrator

The innovative techniques on which AGATA relies, namely the highly segmented Germanium detectors, the analogue and digital processing of their signals, the *pulse shape analysis* of the recorded waveforms to determine the positions of the interaction points inside the germanium crystals and the analysis of these points to track the gammas of the event through the detectors *γ-ray tracking* are being actively investigated since the mid-nineties.

In order to get some understanding of these techniques and with the main purpose of justifying the very high computational needs of the *pulse shape analysis* step, this chapter will give a brief introduction to γ spectroscopy and an overview of its basic working principles.

1.1 INTRODUCTION TO GAMMA-SPECTROSCOPY

γ-ray spectrometers are instruments used to measure as efficiently and as precisely as possible the energy of γ-rays emitted by excited atomic nuclei. The energy of these γ-rays corresponds to the energy difference between excited states of the emitting nucleus so that from their precise knowledge it is possible to reconstruct the level scheme of the nucleus of interest making it possible to obtain evidence an understanding of the particular behaviors and phenomena produced by the interplay of fundamental forces in such a complex system.

The investigation of the nuclear structure requires the study of nuclear species generated using various types of nuclear reactions that are produced by a beam of energetic particles impinging on a target with a velocity of a few percents of the speed of light.

As an example of detection system, figure 2 on the facing page shows the AGATA Demonstrator and its reaction chamber. In this picture the reaction chamber is open to show at its center the position of the target, which is where the nuclear reaction takes place and therefore the source of the gamma rays seen by the detectors.

The gamma rays produced in the nuclear reaction are emitted almost isotropically and those entering-into and interacting-with the germanium crystals generate small electrical signals which are transformed into useful information by the associated electronics, data acquisition system and processing algorithms. At the end of this complex chain, the derived information is normally displayed as energy spectra with characteristic peaks that are used by the physicists to define the excitation structure of the produced nuclei.

Two of the most important parameters that determine the performance of a γ spectrometer are the detector efficiency and the spectral resolution.

1.1.1 Detector efficiency

Not all gamma rays emitted by a radiation source enter into the detectors and, furthermore, those entering into a detector

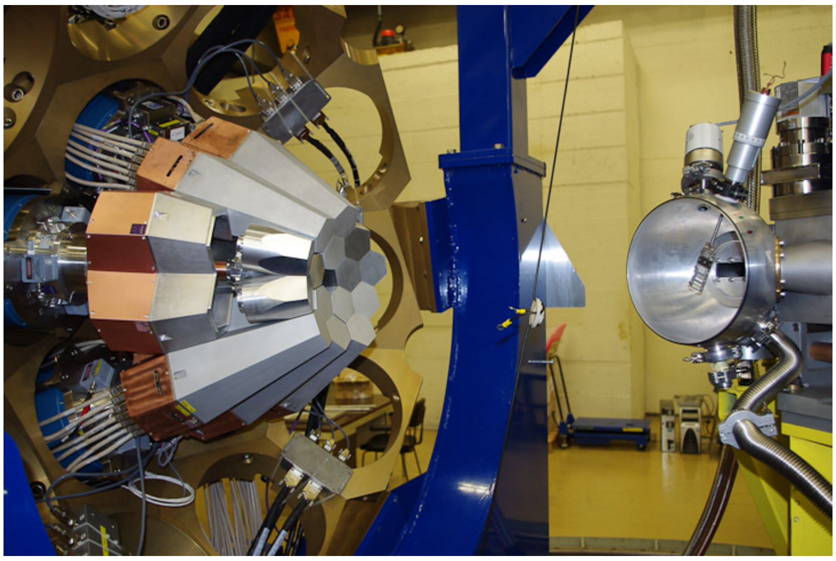


Figure 2: The AGATA demonstrator (on the left) positioned near the opened reaction chamber (on the right). The 15 germanium detectors are assembled three-by-three in 5 cryostats. In this picture, the front-most cluster is opened to show the position of the three germanium crystals

have a certain probability to pass through it without being seen. The efficiency of a detector is consequently given by the probability of detecting the radiation emitted by a source and depends obviously on the size, type and material of the detector.

High-efficiency detectors produce spectra in less time than low-efficiency ones and are essential in the research of rare nuclear species and the associated exotic phenomena. In fact, the reactions that populate the nuclei of interest for the most advanced research are limited in mumber and can produce (for technical and/or economical reasons) only limited number of events, the γ-rays of which should therefore be detected as efficiently as possible.

Detector efficiency is determined by comparing the measured count rate to the known activity of a standard calibration source. Relative efficiency values are often used for germanium detectors and compare the efficiency of the detector at 1332 keV to that of a 3inch x 3inch NaI detector. Relative efficiency values greater than one hundred percent are therefore commonly encountered when working with very large germanium detectors.

1.1.2 Spectral resolution

The energy resolution is the other essential feature of spectroscopic detectors. Resolution is analogous to resolving power in optical spectroscopy; it allows the separation of two gamma lines that are close to each other and the highlighting of transitions otherwise lying in the background.

The common figure of merit used to express detector resolution is the FWHM (*Full Width at Half Maximum*) which gives the width of the γ-ray peak at half of the highest point on the peak distribution. Resolution figures are given with reference to specified γ-ray energies. Resolution can be expressed in absolute (eV or MeV) or relative terms. The energy resolution of germanium detectors for 1 MeV gamma rays is 2 keV or 0.2 %, more than one order of magnitude better than the resolution of a good scintillator detector like sodium iodide (NaI).

1.2 AGATA WORKING PRINCIPLES

In the full configuration, AGATA will have 180 HPGe detectors, each of them consisting in a 36-fold electrically-segmented crystal, that will be positioned in a spherical arrangement covering the full 4π solid angle. In figure 3 part of the complete sphere is depicted [6]. As mentioned before, the target from where the γ-rays will be emitted during the physics experiments will be at the center of the sphere.

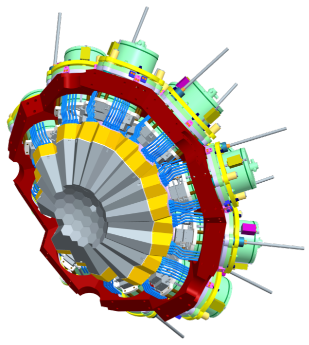


Figure 3: Part of the "full configuration" AGATA array

Ionizing radiation like γ-rays, interacts with matter producing energetic electrons (or electron-positron pairs) that release very rapidly the absorbed energy as atomic excitations of the detector material. In a semiconductor like germanium or silicon, this results in a number of electrons transferred to the conduction band and in a corresponding number of holes left back in the valence band. These free charges are collected to the detector electrodes under the influence of an the applied electric field and result eventually in a small electrical pulse that can be amplified and measured in an outer circuit.

As the amount of energy required to create an electron-hole pair is known, and is independent of the energy of the incident radiation, the measurement of the number of electron-hole pairs, i.e. the amplitude of the pulse, gives the energy of the absorbed radiation.

In comparison to silicon, germanium crystals can be produced with a much smaller concentration of impurities (hence the name of hyper-pure germanium). A smaller impurity concentration allows to produce thicker detectors and this is the main reason why HPGe are the detector of choice in advanced γ-ray spectroscopy. The bigger size and the small mean free path of the radiation increase the probability of interaction for the γ-rays passing through the crystal's matter, increasing consequently the detector efficiency.

Germanium detectors are important because of their large efficiency, but are relatively more difficult to use than scintillator detectors as they must be cooled to liquid nitrogen temperatures to obtain a good energy resolution. In fact, at room temperature, the thermal energy of the electrons is sufficient to cross the band gap in the germanium crystal and to reach the conduction band where they are free to respond to the applied electric field producing a high-enough steady current that would cover completely the small signals generated by the real radiation.

Cooling to liquid nitrogen temperatures (77K)[1], reduces thermal excitations so that only a γ-ray interaction can give to the electrons the energy necessary to cross the band gap and reach the conduction band.

As it can be seen in figure 2 on page 3, the AGATA detectors are grouped three by three and mounted in cryostats that are refilled with liquid nitrogen every 6-8 hours. Such a group of three crystals in a cryostat is normally called ATC (*AGATA Triple Cluster*) [50]. In figure 3 on the preceding page the cylindrical cryostats cooling every ATC can be seen in the outer part of the shell.

1 While working the crystals will maintain a temperature around 90K

1.2.1 Gamma-ray Tracking

As mentioned at the beginning of this chapter, one of the most innovative techniques used for AGATA is the so called γ-ray tracking [49].

The photoelectric effect causing the full absorption of the photon is not the only kind of interaction that may occur between a photon and the matter. In fact, there are three main radiation-matter interactions involved, namely: photoelectric, Compton scattering and pair production.

The probability for every interaction to occur depends on the energy of the photon and on the type of material constituting the detector. This can be seen in figure 4 where the cross section[2] of the main radiation-matter interactions in a germanium crystal is plotted as a function of the photon energy.

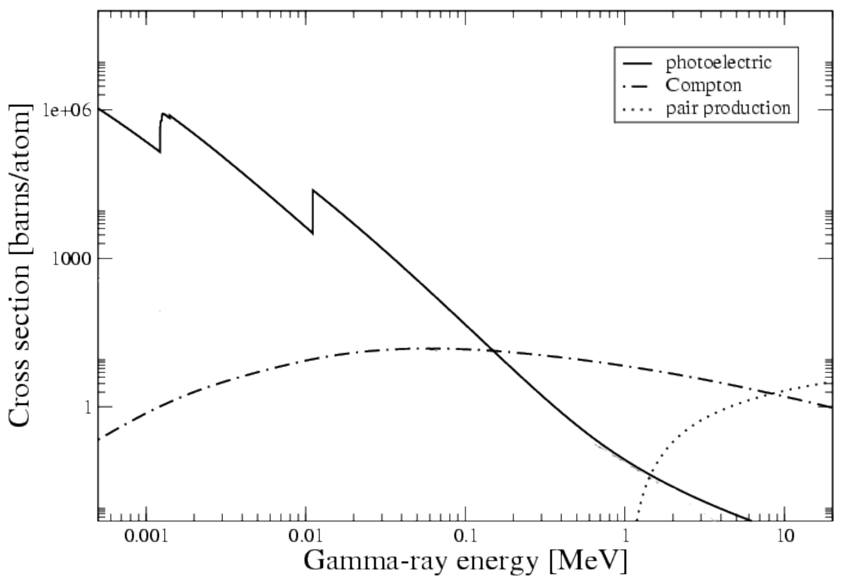


Figure 4: Cross section of the main radiation-matter interactions in germanium as a function of photon energy.

Compton scattering is an inelastic scattering of photons in matter that results in a decrease in energy (increase in wavelength) of a γ-ray photon. In other words, when a γ-ray undergo Compton scattering, part of the energy of the photon is transferred to a scattering electron, which recoils and is ejected from

2 Cross section is a hypothetical area measure around the target particles (usually its atoms) that represents a surface. It is proportional to the probability of interaction on a given amount of matter per unit area.

its atom, while the rest of the energy is taken by the scattered, "degraded" photon, as is presented schematically in figure 5.

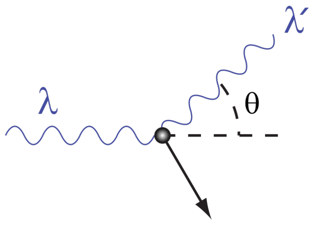


Figure 5: A photon of wavelength λ comes in from the left, collides with an electron at rest, and a new photon of wavelength λ' emerges at an angle θ.

Since, as can be seen in figure 4 on the facing page, Compton scattering is the dominant process between 150keV and 10MeV, algorithms should be developed to reconstruct the full energy of the γ-ray also in case of multiple interaction.

Gamma radioactive decay photons (the ones AGATA is designed for) commonly have energies of a few hundred keV, *and are almost always less than* 10MeV *in energy.*

As the purpose of the spectrometer is the measurements of the γ-rays coming from a nuclear reaction, one needs a way to recognize if a detected γ is coming from the nuclear reaction and has been detected thanks to a photoelectric effect, or if only part of its energy has been measured (point ① in figure 6) due to the detection of a "degraded" photon resulting from a previous Compton scattering thanks to another Compton scattering (point ② in figure 6) or to an eventual photoelectric effect (point ③ in figure 6).

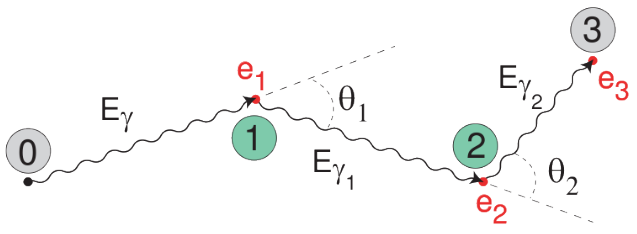


Figure 6: A photon going through two subsequent Compton scatterings in ① and ② and being eventually absorbed with a photoelectric effect in ③.

If a γ-ray undergoes (multiple) Compton scattering and photoelectric absorption within the same detector, the initial photon energy is correctly measured from the sum of all the energy deposits.

Instead, if a γ-ray undergoes (multiple) Compton scattering in the detector but the resulting scattered photon escape the detector, the spectrometer will detect only a partial energy resulting in a background event.

To solve this issue, detector arrays built in the past used a Compton suppression technique [24] in which the germanium crystal are surrounded by a high-efficiency and relatively low cost scintillator which can detect the escaping photons and veto the acquisition of the main event. The drawback of this technique is a reduced efficiency as a consequence of the space occupied between the detectors by the Compton suppression shields. A simple sketch depicting the working principle of a Compton suppression shield is reported in figure 7.

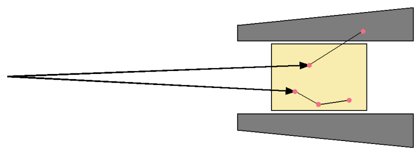


Figure 7: Compton Suppression shields around a germanium detector. A photon that undergoes Compton scattering inside one detector and ends up inside the Compton suppression shield, as the upper one, invalidates the acquisition of the germanium signal. On the other side, a photon that undergoes Compton scattering without escaping the detector, as the one in the lower part of the figure, not being vetoed by the shield is accepted by the electronics and taken into the data acquisition system.

Since Compton scattering is a known phenomenon that is well described by the Compton scattering formula 1.1, the knowledge of the position and deposited energy of every interaction allows, at least in principle to reconstruct, or in other words to track, the path of the γ-rays through the detector.

λ, λ' and θ have the same meanings as in figure 5 on the preceding page
h is the Planck constant,
m_e is the mass of the electron and
c is the speed of light.

$$\lambda' - \lambda = \frac{h}{m_e c}(1 - \cos\theta) \tag{1.1}$$

In fact formula 1.1 can be re-written as reported in formula 1.2 on the next page to show explicitly the relation between photon energies and the scattering angle and can therefore be used to test each measured interaction point for validity as a Compton vertex.

$$E'_\gamma = \frac{E_\gamma}{1 + \frac{E_\gamma}{m_e c^2}(1 - \cos\theta)} \tag{1.2}$$

In principle, a good positional information on each interaction point could be otained by using a very large number of small and independent detectors. However, due to technical and economical reasons this is not feasible in practice and the required positional precision has to be obtained by pulse shape analysis of the signals seen in electrically segmented detectors.

1.2.2 High-fold segmented Ge detectors

As mentioned before, in order to achieve a good tracking efficiency, the positions at which the γ-rays interact inside the detector volume should be determined and, it has to be known with an accuracy between 2 and 5mm. This corresponds to an effective granularity between 3000 and 50000 voxels[3] per germanium crystal.

Such granularity is technically impossible to achieve as a physical segmentation of the crystal for reasons of complexity, number of read-out channels and inclusion of large amounts of insensitive materials in the detection body. However it is possible to increase the number of electrodes connected to the germanium detector by electrical segmentation of the outer surface of the crystal and the positional information can be extracted from the shape of the pulses seen on the different contacts.

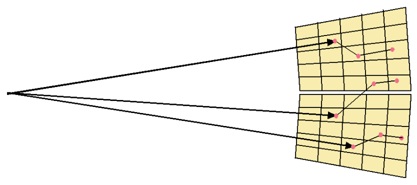


Figure 8: Three photons that undergo to Compton scattering inside two nearby segmented detectors.

For this reason pulse shape analysis methods have been developed [26, 42], which can provide this position accuracy together

3 A voxel (volumetric pixel) is a volume element, representing a value on a regular grid in three dimensional space. This is (in 3D) analogous to a pixel in 2D.

with a precise timing information ($\approx$ 10ns). These methods require a technically feasible level of segmentation of the outer detector contacts forming around 20 $\sim$ 40 segments.

After various simulations, the AGATA project has decided to use 36-fold segmented germanium detector, with six-fold azimuthal and six-fold longitudinal segmentation, schematically shown in figure 9 and in figure 10 on the next page.

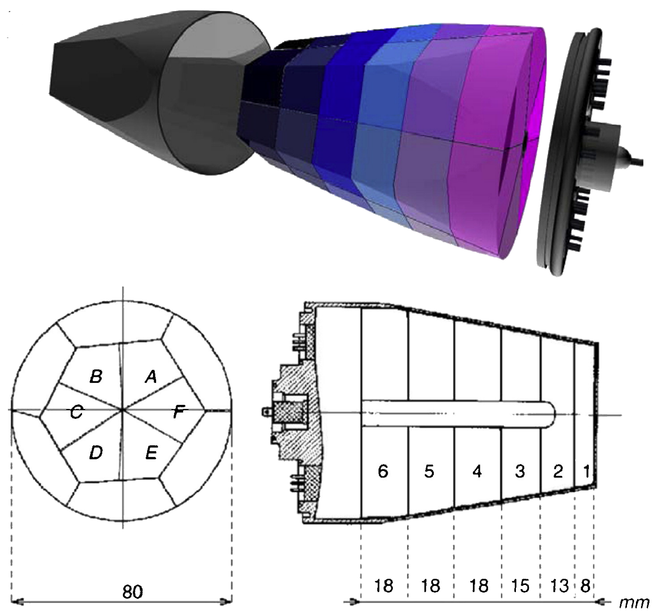


Figure 9: A 3D view of the germanium crystal of an AGATA Detector with its capsule on the left and its connections plate on the right. Colors are not realistic and are used to describe the electrical segmentation. Its dimensions and a view of the inside central electrode and segmentation are given below.

The AGATA germanium detector has a circular shape at the rear side with a diameter of $\approx$ 80mm and a hexagonal shape at the front face. The length of the detector is $\approx$ 90mm and the segmentation is achieved by a separation of the outer implanted contact into six slices and six orthogonal sectors. The 36 segments together with the inner common electrode are read out via individual preamplifiers and can be considered as separate detectors.

1.2.3 Pulse Shape Analysis

The interaction points of the γ-rays in the germanium detector can be localized with a much higher accuracy than defined

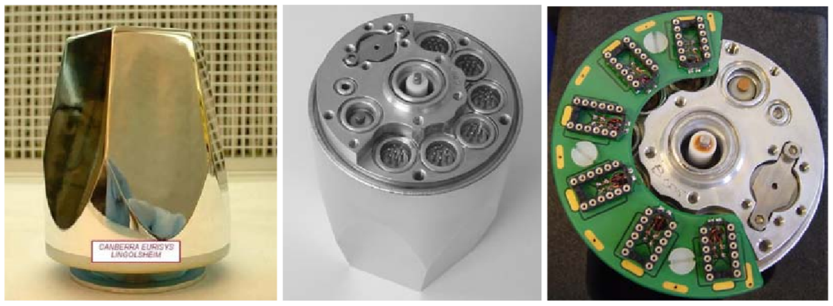


Figure 10: Pictures of the AGATA physically-unsegmented crystals. On the left the row germanium block, in the center the encapsulated crystal and on the right with the connectors PCB.

by the geometry of the segments if the spatial information contained in the shape of the detector signals is exploited.

As introduced in section 1.2 on page 4, signals are produced when the interaction between a photon and an electron, produces a large number of free electrons and holes which induce image charges of opposite signs on the detector electrodes.

As the charge carriers (electrons and holes) drift towards the electrodes, the amount of the image charges changes causing a flow of currents into or out of the electrodes.

Since signals are induced only if there are moving charge carriers within the detector, different signal shapes will result for interactions occurring at different distances from the electrodes, or at different distances from the segment borders.

Consequently, the observation of a net charge on the charge-collecting electrode can be used to identify the detector segment where the interaction took place. The transient image signals on the other non-collecting electrodes vanish when the charge carriers are collected.

Examples of calculated induced current signals in a coaxial detector are schematically shown in figure 11 on the next page. In the left part of the figure, a transversal cut through a coaxial detector is presented together with the drift directions of the charge carriers. Depending on the radius where the charge carriers are produced, the shapes of the induced current signals are different for different interaction radii. The right part of figure 11 on the following page depicts four such examples.

Experimentally, characteristic pulse shapes have been studied in detail with the various existing segmented germanium detectors before the construction of the first AGATA detector; fig-

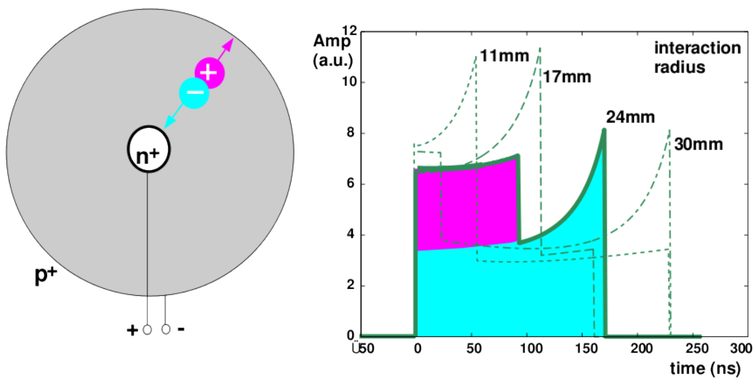


Figure 11: On the left there is a sketch of how the charges moves inside a coaxial germanium detector, while on the right is plotted a simulation of the different pulses collected for different interaction position radii.

MINIBALL detectors are 6-fold segmented HPGe detectors

ure 12 shows an example of pulse shapes measured at the core and at the six segments of a MINIBALL detector [48]. The signal of segment 4 has the same pulse height as the signal of the core, indicating that all the energy was deposited in this segment. The neighboring segments (3 and 5) show a positive mirror-charge signal, indicating that the main interaction occurred close to the core in segment 4. The pulse height of the mirror charge in segment 3 is larger than in segment 5, showing that the interaction occurred closer to the segment 3 than segment 5.

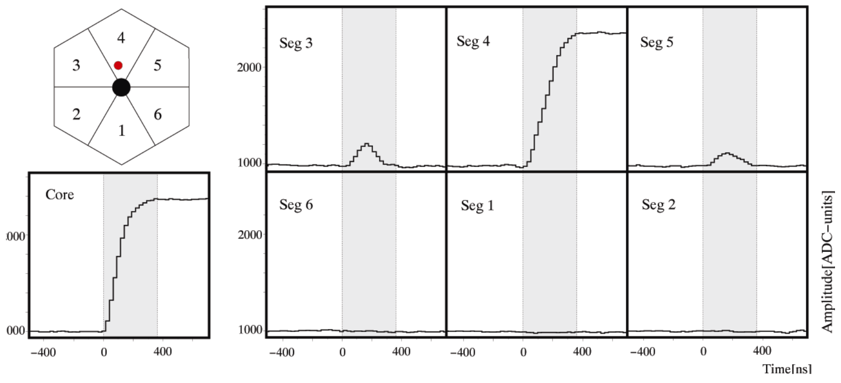


Figure 12: Signals produced in the core and in the segments for a 6-fold segmented detector when a photon is fully absorbed in segment 4. Since the interaction takes place closer to segment 3, the amplitude of the corresponding transient signal is larger than the amplitude for segment 5.

This is basically the technique used with the MINIBALL detectors to extract the azimuthal coordinate of the interaction [48]. Unfortunately, this simple method cannot provide the kind of precision needed by the tracking algorithms and more sophisticated techniques had to be developed, involving the digitization of the signals from each segment and the comparison of the transient and net charge signals with a database (called *basis*) of reference signals (that is, a set of signals corresponding to interactions taking place in specific locations within the detector).

Several algorithms have been developed and tested so far within the AGATA collaboration. The method used in the present work is a *grid search*, consisting of a comparison between the acquired signals and the basis signals, which are sampled over a uniformly-spaced grid. A more detailed description of the algorithm, as well of the results obtained, will be given in section 4.1 on page 47.

Since the construction of the reference basis is an essential ingredient for any pulse shape analysis algorithm, the techniques used to measure and construct such basis will be illustrated in section 1.2.4.

Following this approach AGATA will use pulse shape analysis methods to reduce the segmentation scheme to a technically feasible level while maintaining the position resolution needed for tracking. In fact, for an efficient tracking, especially in case of multiple interactions, not only the accurate positions of the interactions, but also their number, and the partial energies released at each interaction, have to be determined.

Pulse shape analysis can provide this information, however with a finite accuracy, which depends on various parameters, where detector geometry, segmentation level, impurity concentration, preamplifier bandwidth, signal-to-noise ratio and sampling frequency are some of them. In this respect pulse shape analysis plays a key role for the AGATA spectrometer, electronics design and for γ-ray tracking in general.

1.2.4 Basis Database Production

As introduced before, for any pulse shape analysis algorithm to work, a record of the pulse shapes read at every electrode of the detector, for every interaction position has to be recorded. Once obtained a database of pulse shapes (called reference basis) that associates some pulses read on the electrodes to an interaction position, the PSA algorithm can compare the acquired experimental data to the saved ones looking for a match and therefore finding the correct interaction position.

Hence every detector may have different responses due to its peculiarities (germanium impurities, pre-amplifiers characteristics, crosstalk phenomena, etc.), one database for each of them has to be provided.

Several devices, known as scanning tables, have been developed within the AGATA collaboration in order to construct the reference basis needed by the PSA algorithms or, in other words, to measure in a semi-automatic way the signals corresponding to each specific locations within the crystal (with the possibility to move such locations in any point of the detector).

In case a collimated[4] photon beam is available, the position of the interaction can be determined by requiring the coincidence of the germanium signal with a second collimated detector, as schematized in figure 13 on the facing page. The main problem with this kind of measurement is that the constraints on the recorded events are stringent and as a consequence, very long measurement times are needed to collect the required statistics. In addition, the measurement for points at the backward part of the detector is extremely difficult [41].

It should be observed that, in principle, the information gathered with this kind of scanning table is redundant. In fact, the energy deposited by the photon inside the germanium detector is fixed by the geometrical arrangement since the source emits constant energy photons and the scattering angle is fixed (with a minimum spread distribution around 90°). This information on the energy is used in combination with the equivalent information given by the coincidence detectors to clean the data from spurious events in which the photon underwent multiple Compton scattering.

Using the scanning method previously presented, it takes far too long time (two months) to scan a full detector with a fine grid ($1\text{mm} \times 1\text{mm} \times 1\text{mm}$). This means that years of scanning will be required to fully characterize the detectors composing AGATA, which will be 180. Additionally, the uncertainties of the scanning setup are too large to allow the production of a finely sized basis set that, if not precise enough, will compromise the whole array tracking performance [41].

In principle, knowing the electric fields inside the detectors, the signal shapes can be calculated and the data from the scanning tables can be used to tune the parameters of the calculations and eventually to validate the simulated results.

4 Collimated rays are nearly parallel, and therefore will spread slowly as they propagates.

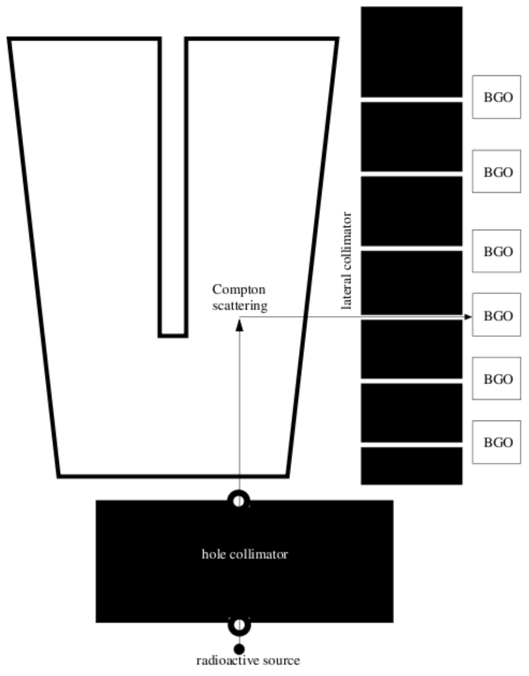


Figure 13: Schematic view of a scanning table. The collimated photon beam enters from the front face of the detector through a hole collimator. The trigger of the acquisition system selects only the events where the photon Compton scatters and the residual photon passes through a second collimator to reach one of the scintillator detectors placed around.

Consequently, to move around the scanning tables limitations, many software codes have been developed to calculate the signal shapes in segmented germanium detectors; an early example is reported in [8], but other examples are MGS [25] (*Multi Geometry Simulation*), JASS [41] (*Java AGATA Signal Simulation*) and the ADL (*AGATA Data Library*) detector simulation software.

Most of these programs can calculate the signal shapes for interactions distributed over a regular cubic lattice having 1mm step and their results can be validated using sophisticated scanning tables, e.g. using the PSCS (*Pulse Shape Comparison Scan*) characterization technique [15].

In particular, for this work, a 2mm step pulse shapes database produced by the ADL software, has been used.

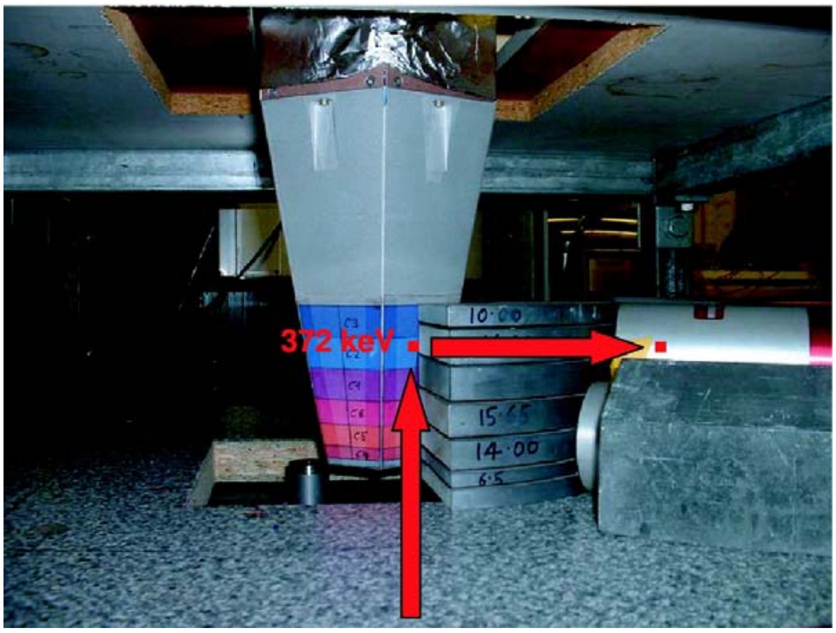


Figure 14: Photo of the scanning table in Liverpool. The lateral collimation and the BGO detectors are visible.

2 DATA ACQUISITION SYSTEM

The Data Acquisition (often referred-to as DAQ) is interacting with almost all the AGATA elements and it has two essential functions; the first one is to *read-out* and transport the data flow from the detectors up to the storage, while the second is to process data, applying Pulse Shape Analysis and Tracking algorithms.

As already discussed, the detectors of AGATA will be operated in a "position-sensitive" mode, by digitizing the signals from the preamplifiers and by using PSA algorithms to extract the information (energy and position) on the single interaction points.

Given that all the signals are digitized continuously at a sampling frequency of 100MHz using 14bit ADCs, the flow of digital data produced by each AGATA crystal is 38 channals $\times$ 200MB/s $\approx$ 7.6GB/s, which sums-up to the impressive value of $\approx$ 1.3TB/s for the 180 detectors of the whole array. This huge amount of data is reduced to more reasonable values ($\approx$ 10kB per event for a maximum event rate of 50kHz per detector) by the real time data processing algorithms of the AGATA front-end electronics. After this, the data can be read-out to the Data Acquisition computers but, keeping in mind that a typical experiment can run for a full week, its amount is still too big to be stored for off line analysis and has to be further reduced by the Pulse Shape Analysis algorithms.

Channals are 38 since the core signal is digitized two times with different amplification ratios

This chapter will describe the organization and the features of the AGATA Distributed Data Acquisition System, starting from an overview of its main components and later focusing on its software part.

2.1 OVERVIEW

As illustrated in figure 15 on the next page, the DAQ elements are made of analog electronics (dark gray boxes), digital electronics (light gray boxes) and pieces of software (white boxes).

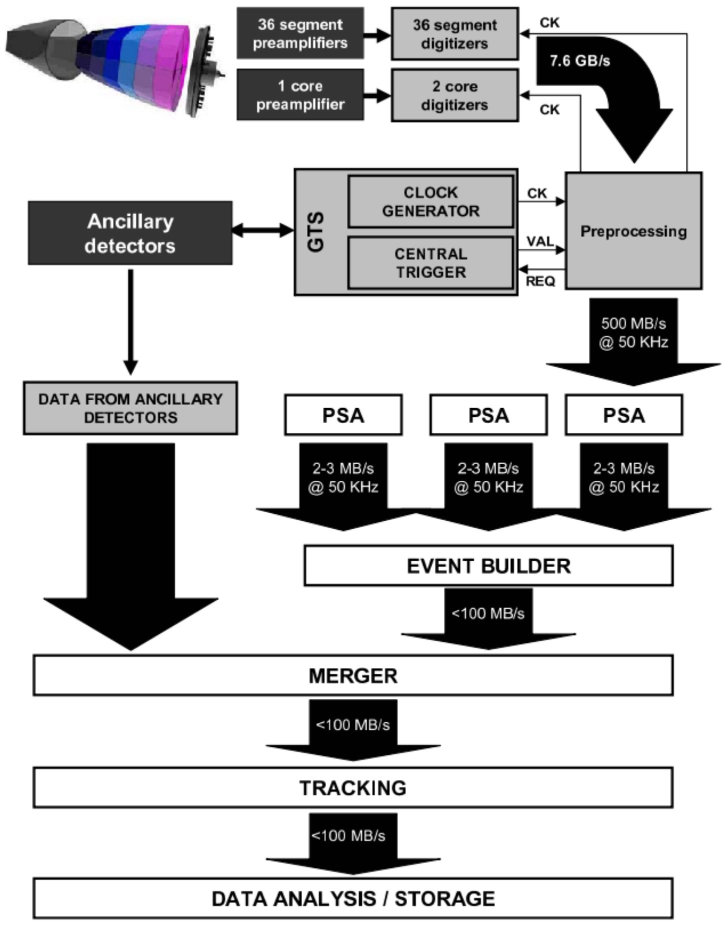


Figure 15: Schematic view of the AGATA Data Acquisition System.

2.1.1 Signals Digitization

For every detector data has to be acquired from the 36 segments of the cathode plus the two energy ranges of the central contact (anode) as mentioned in section 1.2.2 on page 9. For every one of them there is an analog amplifier composed of a cold part (kept at liquid nitrogen temperature) and a warm part (at room temperature).

Those amplifiers are mounted on the detectors structure and are needed to amplify the signals to let them reach throw a differential line the digitizers that are placed far from the detectors.

A graphical electrical scheme of the amplifiers can be seen in figure 16.

All of the digitizers work synchronously by receiving a 100MHz clock from a central clock generator, called the Global Trigger and Synchronization, often referred as GTS [46]. The continuous

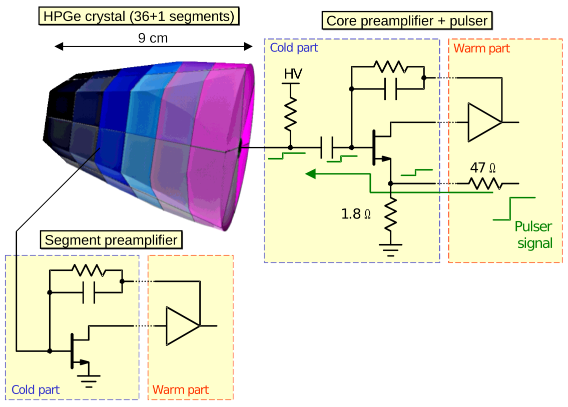


Figure 16: Electrical scheme of segment and core preamplifiers.

flux of samples from the digitizers is sent over optical fibers to the pre-processing digital electronics, which analyzes each crystal, extracting the information on the energy deposited in each segment, managing the synchronization, attaching to the data a timestamp and reading out the interesting part of the signals for further analysis.

2.1.2 Pre-Processing

To be able to perform the requested operation in *real time*[1], the signal-processing algorithms are implemented into powerful highly-parallel FPGAs (Field Programmable Gate Array).

The occurrence of useful signals in a crystal is detected by a digital trigger applied to the data stream of the anode (core), since a pulse there means that an interaction occurred in one of the segments of the crystal and the acquired samples after that moment may be useful. This local trigger forces the core and all 36 segments to generate an energy value (by means of the so-called Moving Window Deconvolution [20, 21]), and up to 160 samples (1.6μs) of the rise time of the pulse, discarding all the rest.

Although in this way a data reduction of two orders of magnitude is achieved, the throughput can still be as high as 500MB/s per detector and it is almost mandatory to have a second-level trigger selecting the most useful events according to to user-defined, experiment-specific conditions. These conditions can be simply that a minimum number of germanium crystals fire

1 Which in this case means to analyze a sample every 10ns

simultaneously, or request that other (ancillary) detectors participate in the event.

The time correlation among the detectors of AGATA is achieved by means of a time-stamping system where each clock pulse distributed by the GTS has an associated 48bit clock counter which gives, in steps of 10ns, the time elapsed since the beginning of the measurement run.

Whenever the local trigger in a detector fires, the clock number (a timestamp) is recorded in a local memory buffer together with the energies and rise time slices of the 36 segments plus the two core signals. At the same time, the timestamp is sent back to the GTS system which uses it to generate a global trigger according to the specified conditions. The locally-recorded data is validated and passed over to the next processing stage only if the global trigger fires, otherwise it is discarded, thereby reducing even more the data throughput.

To simplify the operation of the global event builder, the GTS generates an event number which is added as a tag to the validated data. The global level trigger can be defined in such a way that all local triggers are validated (e.g. in case of very low counting-rate experiments) achieving what in the AGATA specifications is called the "trigger-less" mode.

In the AGATA scheme, the ancillary detectors can use a similar digital electronics but can also use a classical VME-based [1] analogue DAQ. In this case, the time correlation to the AGATA detectors is performed by a dedicated VME module, called AGAVA, which interfaces to the GTS system by reading its clock and timestamp, by sending local trigger requests and getting the corresponding validations. After pre-processing, the validated local events are passed to the Pulse Shape Analysis stage.

2.1.3 Software Processing

The PSA algorithms are implemented in software and are applied inside a computer farm, in order to extract the coordinates of the interaction points from the pulse rise-time samples. Once this is done, the traces can be discarded, thereby reducing the data throughput by an order of magnitude. This analysis is local to each germanium crystal and the fastest PSA algorithms developed using CPUs need a few milliseconds of an E5420 2.5GHz Intel® Xeon® core to analyze one event.

This was the bottleneck of the whole data-processing sequence and to achieve the 50kHz singles rate of the AGATA specifications, a farm of about 100 CPUs cores should be dedicated to each crystal.

In the initial phase of AGATA the rate of accepted signals has been reduced by the global trigger to $\approx$ 1kHz per detector, meaning that the PSA farm could be limited to a few CPUs per crystal, but for the physical measurements campaign there is the need of an increased acquisition rate and consequently an higher PSA algorithms efficiency or computing power dedicated.

The PSA completes the data analysis at the detector level. After this, the global event has to be built collecting together, on the basis of the event number and/or timestamps, the information of all firing detectors.

In the first implementation of the AGATA DAQ, this is done by first assembling the germanium detectors and then merging the data from the ancillaries.

After the PSA results of all the detectors have been collected together in single events and they have been merged with ancillary detectors ones (in the case ancillary detectors are present), global events contains all information (energies and positions of the interaction points of the firing germanium detectors) needed to perform the γ-ray tracking and any other experiment-specific on-line analysis.

After the tracking operation, full tracked events will be saved on permanent storage [5] and will be made available to the experimental groups for the final off-line data analysis.

2.2 COMPUTING INFRASTRUCTURE

Due to the need of AGATA to have most of the analysis done during run-time, a computing farm has been installed near the experimental hall, where AGATA is placed.

Since AGATA is a mobile experiment and will be hosted in several laboratories in different countries, the computing farm has been organized to be an autonomous network inside the bigger hosting laboratory one. Thanks to this configuration and also to the use of the NARVAL Distributed Data Acquisition System described in section 2.3 on page 24, the computing farm of AGATA is often referred as the "DAQ Box" and appears to the physicists running experiments as a *black box* that can be operated from an intuitive GUI software running on workstations.

Actually, the farm, as reported in figure 17, is composed of IBM® $x3550$ $1U$ machines, equipped with two E5420 Intel® quad cores Xeon® CPUs (12MB of cache per CPU) and 16GB of RAM each[2].

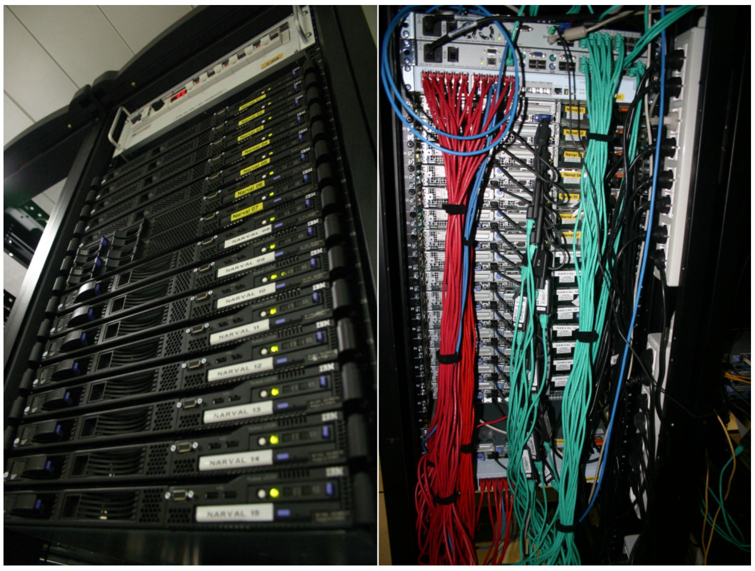


Figure 17: Photo of the machines installed in the AGATA computing room. Front view on the left and rear view on the right.

As depicted in the scheme in figure 18 on the next page, the machines are connected thanks to two different networks, one of them, called "data-flow network" is used only to transfer experimental data between computing nodes, while the other one, called "services network", is used to monitor the acquisition and to send commands to the machines.

The data flow coming from the FPGAs described in section 2.1.2 is transmitted to some machines inside the farm trough PCI Express connections over optic fibers, those machines are the ones where the data flow is injected inside the Distributed Data Acquisition Software described later in section 2.3.

Apart from the machines dedicated to computation, two other machines (on the upper left in figure 18 on the facing page) host some virtual servers, using Xen®, and are used as library-developers machines, to run monitoring tools, the run control software, an internal DHCP service and a documentation wiki.

Another machine (depicted in white in figure 18 on the next page), is used to implement a VPN service in order to let all the

2 The number of those machines is constantly increasing, since the number of crystals composing the array is increasing too.

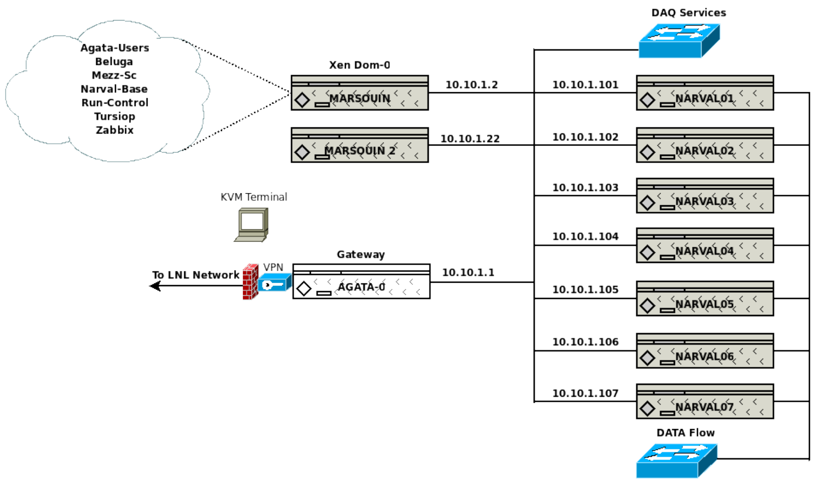


Figure 18: Network connections in the AGATA computing room.

computer administrators in the AGATA community to connect to the DAQ Box from their home institutions.

For the storage of experimental results, the computing farm is connected to a SUN® StorageTek™ 6540 disk array equipped with 64TB of SATA hard disks. In figure 19 can be seen the interconnections of the two disk server machines (in gray), used to write data on disks and the machine dedicated to read-only operations (in white).

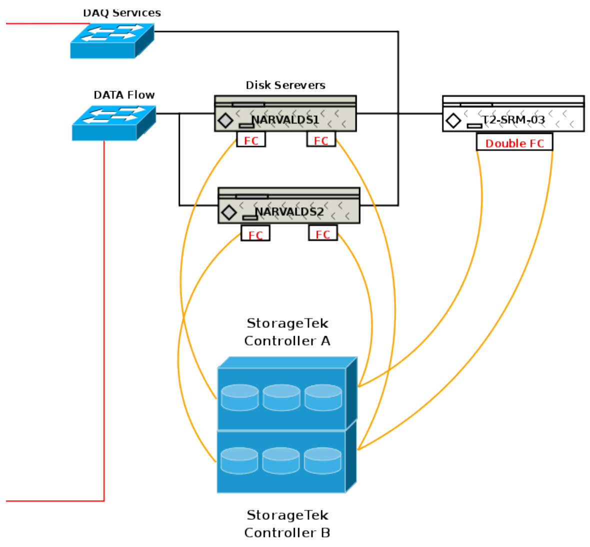


Figure 19: Network connections in the AGATA storage room. Copper 1Gbit/s *Ethernet* connections in black, optic fiber ones in red and 4Gbit/s *Fibre Channel* disk connections in orange.

This disk array has been planned as a fast temporary storage to be used while acquiring data, since after the experiments, the results will be moved to the GRID *Tier*-1 computing center in Bologna where will be archived on tapes [5].

All the disk server machines are part of a GPFS™ cluster; GPFS™ (*General Parallel File System*) [2] is a high performance shared-disk clustered file system developed by IBM® and it provides concurrent high-speed file access to applications executing on multiple nodes. In addition, thanks also to connections and data redundancy, it increase the fault tolerance of the storage system letting it work (with reduced performances) also in case of one machine/disk/connection failure.

Apart from providing filesystem storage capabilities, GPFS™ provides also some tools for the management and administration of the cluster and allows for shared access to the filesystem from remote NFS clients.

2.3 DISTRIBUTED DAQ: NARVAL

NARVAL (*Nouvelle Acquisition temps-Réel Version 1.6 Avec Linux*) is a distributed data acquisition system written in Ada95 programming language and partitioned with the use of the Ada DSA (*Distributed System Annex*), also known as Annex E [23].

NARVAL is a modular distributed system that allow to manage the DAQ data flow; its main task is to dispatch data buffers coming from the detectors to the different steps of the data analysis and eventually to the disk storage [5].

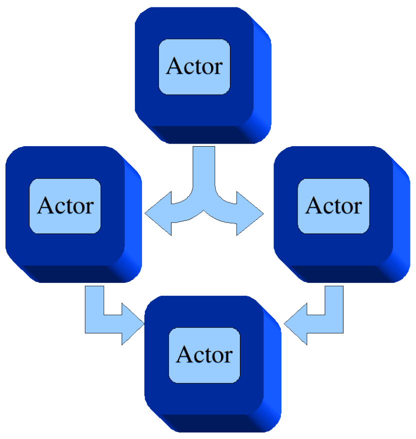
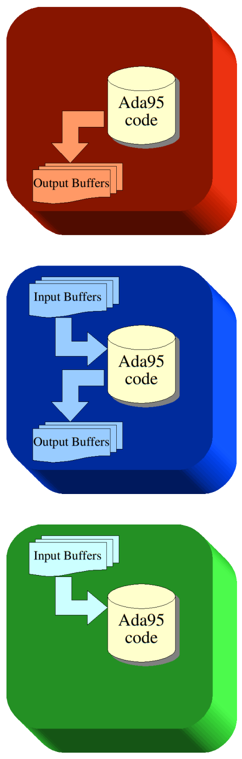


Figure 20: Generic sketch of some NARVAL Actors and the data flow passing through them.

Its modularity allows the separate management of different "blocks" of the system called actors, where every actor can be seen as a box containing some programming code that can be modified to determine what kind of operation will be performed on the data passing through it.

The NARVAL system takes care of the coordination of the various actors involved in the data acquisition. It can be configured in order to assign every actor to a computer (where it will be run) and to set the path of the data flow through the actors.

At execution time every actor represents a single process that can run on one core of the different machines of the distributed system. Actors can be of three different classes, every class is sketched in figure 21 by a different colored box and the main differences between them are summarized.



- producer, which collects data from the hardware and dispatch it to other actors. It's used to get data, which can be read from files or sockets, inside the NARVAL system.
- intermediary or filter, which receives/sends data from/to one or more actors. It's used to process data accordingly to its code and then dispatch it to other actors. It can be used to implement the PSA, the event building, the tracking, etc. accordingly to what is written into the library.
- consumer, which receives data from one or more actors. It's used to get data out of the system, for example writing it to disk.

Figure 21: Different kind of NARVAL Actors.

All the actors are written in Ada95, but, a *generic actor* (for each class) without data manipulation code is provided with an interface for C/C++ code inclusion. Consequently external libraries can be linked in order to manipulate data, while the

Ada95 part of the actor still manage input/output buffers, as can be seen in figure 22.

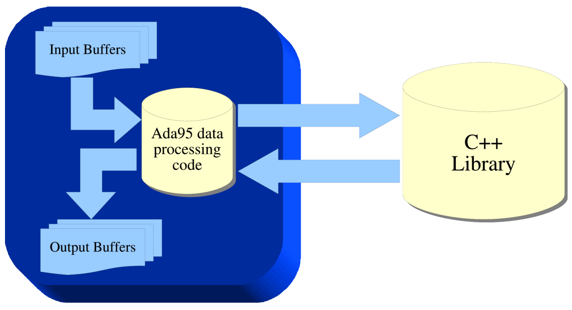


Figure 22: Sketch of a generic filter actor that can be provided with a C/C++ linked library implementing any algorithm for data processing.

Every NARVAL instance, is started loading a main *plain text* configuration file and a *XML* topology file.

The main configuration file is used, like a *script*, to automatically commit a list of commands to the NARVAL system; some of those commands are used to set the instance name, the XML topology file path, the libraries to be loaded by every kind of actor, etc.

The XML topology file is used instead to instruct the system on the number of actors, for every kind, that has to be run and to assign them the origins and/or destinations of the input/output data flow, to choose the input/output buffers size, and to designate the machine where every actor will run.

A sample XML file to load a chain of three actors running on three different machine is reported in listing 1 on the facing page.

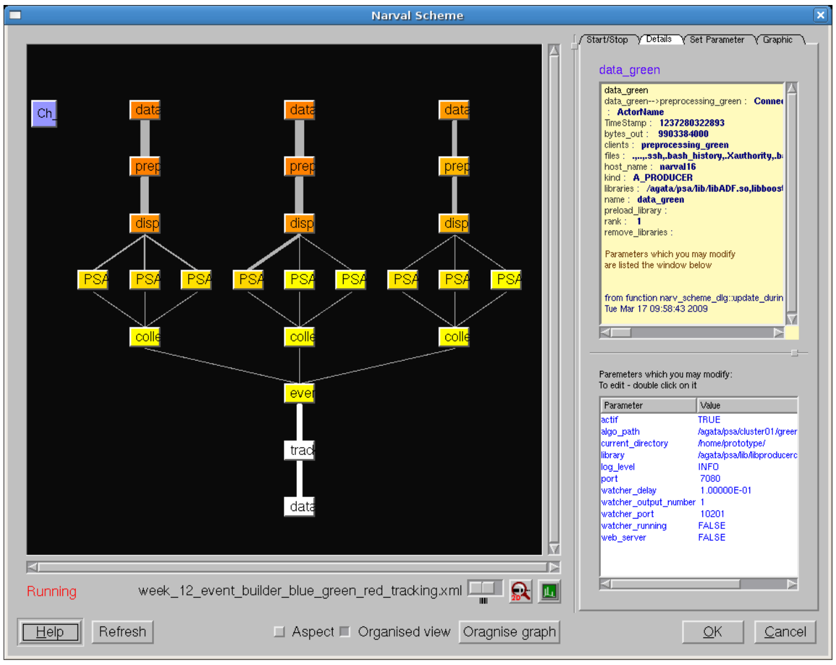
All the actors with the same processing code have in common the same name, but various instances of them can be run concurrently with different data origin and destination; in this way, once all the actors are programmed and tested, adding a new detector in the AGATA array from the DAQ point of view, is just a matter of changing the topology file and to have enough computing power.

Once running, all the actors provide various information about their status that can be read and visualized using a GUI software, called *Cracow*. The same interface is also used to start, stop and

Listing 1: Simple NARVAL XML topology file example:

```xml
<configuration>

  <producer>
    <name>file_reader</name>
    <hostname>narval01</hostname>
    <binary_code>generic_producer</binary_code>
    <output_buffer_name>data1</output_buffer_name>
    <size output_buffer="data1">1000000</size>
    <port output_buffer="data1">eth1</port>
    <debug>info</debug>
  </producer>

  <intermediary input_buffers="1" output_buffers="1">
    <name>filter</name>
    <hostname>narval02</hostname>
    <binary_code>generic_filter</binary_code>
    <data_source source_port="eth1" source_buffer="data1">
                      producer</data_source>
    <output_buffer_name>data2</output_buffer_name>
    <size output_buffer="data2">1000000</size>
    <port output_buffer="data2">eth1</port>
    <debug>info</debug>
  </intermediary>

  <consumer>
    <name>consumer</name>
    <hostname>narval03</hostname>
    <binary_code>generic_consumer</binary_code>
    <data_source source_port="eth1" source_buffer="data2">
                      filter</data_source>
  </consumer>

</configuration>
```

pause the acquisition system. A screenshot of this software can be seen in figure 23.



data_green

```
data_green
data_green-->preprocessing_green :   Conne
 :  ActorName
TimeStamp :   1237280322893
bytes_out :   9903384000
clients :   preprocessing_green
files :   .,..,.ssh,.bash_history,.Xauthority,.b
host_name :   narval16
kind :   A_PRODUCER
libraries :   /agata/psa/lib/libADF.so,libboos
name :   data_green
preload_library :
rank :   1
remove_libraries :

Parameters which you may modify
are listed the window below


from function narv_scheme_dlg::update_durin
Tue Mar 17 09:58:43 2009
```

Paremeters which you may modify:
To edit - double click on it

Parameter	Value
actif	TRUE
algo_path	/agata/psa/cluster01/greer
current_directory	/home/prototype/
library	/agata/psa/lib/libproducerc
log_level	INFO
port	7080
watcher_delay	1.00000E-01
watcher_output_number	1
watcher_port	10201
watcher_running	FALSE
web_server	FALSE

Figure 23: Screenshot of the *Cracow* GUI showing the DAQ system running on one triple cluster and splitting the PSA computations on three identical actors to occupy more processor cores.

2.3.1 Algorithms Integration

As it has been discussed, the design of NARVAL is based on the actor concept and these actors are unaware of the topology of the whole system; therefore they can be developed separately as long as they provide the interface layer used by NARVAL to load, control, and communicate with them [16].

For PSA and Pre-processing algorithms, C++ based classes have been developed providing a simple mean to connect a working algorithm with the data flow of the AGATA DAQ using the concept illustrated in figure 22 on page 26.

The interface that has to be implemented in a library to be linked as the processing-code inside a NARVAL actor, is very simple and it has to contain four required symbols plus six optional ones.

- The `process_config` function that will be called by the system only once, before any other calls. NARVAL will supply a directory path as an argument, specifying where configuration data can be found. This routine has the responsibility to read all necessary configuration data into local arrays.
- The `process_register` function that will be called by the system only once, at program load time for each algorithm process instance.
- The `process_initialise` function that will be called once per instance every time a (re)initialization of the loaded program is required. After the initial call, any further call will follow a `process_reset` function call.
- The `process_block` function that will be called every time an incoming data block is received and requires processing. This function receives as an argument a pointer to a full incoming buffer and a pointer to an outgoing empty buffer. This is where the algorithm to actually process data has to be implemented.

The six optional functions has self-explanatory names and are: `process_start`, `process_pause`, `process_resume`, `process_stop`, `process_reset` and `process_unload`.

To further facilitate the separation between data transport and algorithm development, has been provided the ADF framework (*AGATA Data Format*). The purpose of this software is to define a standard data format to let the actors pack and unpack frames contained inside data buffers, calling predefined functions. Thanks to the ADF library, NARVAL can completely ignore the content of data buffers, while the data-processing libraries can read/write data frames without re-implementing the formatting code.

2.3.2 NARVAL Emulator

Since to debug C++ libraries to be used with NARVAL actors, running them inside the whole system is very hard and in some cases actually impossible, a NARVAL emulator has been developed and provided.

The main problems that arise debugging C++ libraries inside NARVAL are caused by its distributed and concurrent nature, that make impossible the use of debuggers like *gdb*. The NARVAL emulator aims to facilitate the development and debugging of C++ libraries implementing a fake NARVAL environment that, simulating a running system, calls the developing library as the real system would.

This tool is written in C language and using `dlopen()` function calls can dynamically load C/C++ shared libraries implementing the processing-code, that should be loaded by an actor, according to the NARVAL configuration files.

The main difference in comparison to the real system, is that in the Emulator, all the processing-code libraries run on the same machine and they are executed sequentially from the first one, to the last one in the data-flow path.

Actually the NARVAL Emulator, will at first read the configuration files used by NARVAL, load all the needed libraries and then will execute the shared library that would be executed by the first "producer actor" in the NARVAL environment. When it will have filled its output buffer, the emulator will execute the next actor's library in the chain, using the filled buffer as an input. The emulator will run until the buffer will reach the library of the last consumer actor and then will start again from the principle "acquiring" another buffer.

Thanks to the Emulator all the data processing can be "followed" inside the libraries thanks to debuggers like *gdb*, and the code can be checked against memory leaks with tools like *valgrind*.

Unfortunately, some of the actor libraries could not be developed using the Emulator, as the *event builder* actor for example, since their features are strongly related to the concurrent and parallel execution of the preceding actors in the data flow path. In fact, these actors were not implemented in C/C++ language, but in Ada95.

In spite of this, the PSA actor, having only one incoming and one outcoming data stream, could be developed using the NARVAL Emulator in C++ code as will be discussed in chapter 1.2.3 on page 10.

3 GENERAL-PURPOSE COMPUTING ON GPUS

The GPGPU (*General-Purpose computing on Graphics Processing Units*) is the technique of using a GPU (*Graphics Processing Unit*), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the CPU.

In the last years, a technologic limit (caused by electronic components miniaturization) has been reached in the production of faster and faster CPUs, inducing the manufacturers to increase the number of cores contained in one processor instead of its clock frequency.

This rather new trend for CPUs has been introduced several years ago for GPUs that actually carried hundreds of "low clock" cores on the same GPU chip. The reason why GPUs manufacturers entered this trend sooner, was the need of producing low-cost chips helped by the fact that graphic calculations are easily parallelizable on a very large number of cores.

GPUs manufacturers exploiting the peculiarity of graphic computations, that are highly *data-parallel* by nature, reached an affordable processor model capable of a great computational power that moreover seems to follow Moore's Law [45] better than ordinary CPUs, as can be seen in figure 24 on the following page.

Data-parallelism is achieved when each processor performs the same task on different pieces of distributed data

However, the reason why GPUs did not supersede CPUs, is that they are not so well suited for every kind of computational problem. In figure 24 is plotted the theoretical number of single-precision floating point operations per second that different processors may handle, but it does not show how to reach it in practice.

For example, having a huge array of data where the same operations have to be applied on every cell of the array, independently from each others, a CPU and a GPU are likely to reach the GFLOP/s peaks indicated in figure 24. Instead a CPU will most likely outperform a GPU for the solution of a problem where the same number of operations has to be applied (on the same

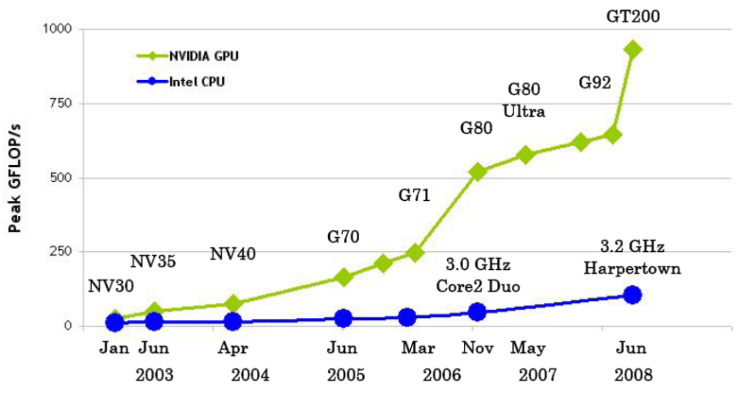


Figure 24: NVIDIA GPUs theoretical computational power in GFLOP/s compared to Intel CPUs. *Image courtesy of NVIDIA Corporation*.

amount of data), but with a lot of conditional statements and dependencies between different array cells[1].

In fact is quite infrequent in graphic computations to have a lot of conditional statements and if we could plot the percentages of transistors used to implement the control unit, the cache, and the arithmetic cores in a GPU, against the percentages relative to a CPU, we would obtain the graphic in figure 25, showing a fewer percentage for the control unit.

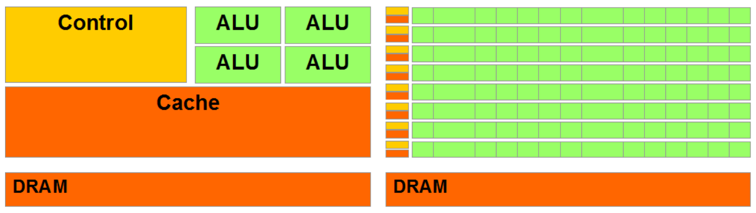


Figure 25: Average chip area (corresponding to the number of transistors) used to implement the various units of a processor for CPUs on the left and for GPUs on the right. ALU stand for *Arithmetic and Logic Unit*, while DRAM for *Dynamic Random Access Memory* (system or card memory). *Image courtesy of NVIDIA Corporation*.

Apart from the control unit, the GPUs have also a smaller cache and "smaller" ALUs (*Arithmetic Logic Unit*) (although in a higher number), than in the CPUs. The smaller cache can be explained since highly arithmetic independent operations, running in parallel on different data trunks (different threads), can easily hide memory latency, while simpler ALUs can be explained by

1 This example does not want to be exhaustive, but just an introduction to the more formal and detailed concepts discussed in section 3.1.

the fact that they have a restricted set of functions and basically have to be fast floating-point arithmetic units.

What General-Purpose computing on GPUs technique aim to, is the exploitation of the great computational power offered by modern graphics cards, providing to software developers the tools needed to execute some parts of their programs on GPUs processors. Selecting only the software parts that may be executed faster on a GPU and running them on a graphic card rather than on the CPU, huge execution speed improvements can be obtained.

3.1 GPUS ARCHITECTURE MODEL

GPUs are designed specifically for graphic computations and thus are very restrictive in terms of operations and programming. Because of this nature, GPUs are only effective at tackling problems that can be solved using a particular programming paradigm called *stream processing* that is perfectly suited for graphic computational needs.

This paradigm is strongly related to the SIMD (*Single Instruction Multiple Data*) class of computer architecture in Flynn's taxonomy [18] and allows software applications tu run on multiple computational units, without explicitly managing allocation, synchronization, or communication among those units.

To allow this simplification in the software parallelization, this paradigm restricts the parallel computations that can be performed. Therefore it can be defined as a SIMD paradigm since it allows the same set of instructions (often called *Kernel*) to be run on different processing units, on different sets of data. In figure 26 on the following page a graphical representation of the SIMD model is reported.

If the same code is executed on different processing units, only one control unit is needed for all of them and hence a lot of transistors in the GPUs can be used for arithmetic units instead of control units as shown in figure 25 on the preceding page.

This model is not new in the computing history and has been used in the vector supercomputers of the early 1970s. However for its lack of flexibility for general purpose computing, it has been used only for specific computing fields and for providing a few specific multimedia instructions inside general purpose processors (like the MMX "technology" of Intel processors, or 3DNow! for AMD ones).

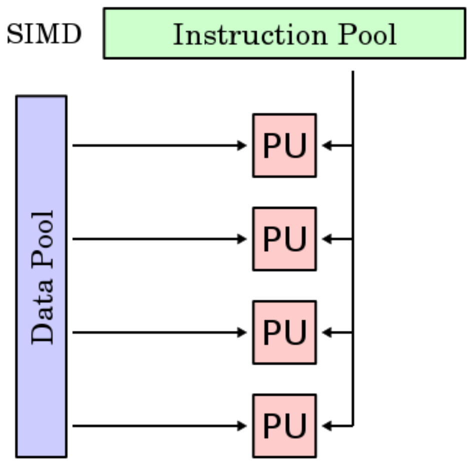


Figure 26: SIMD model representation. Each PU (*Processing Unit*) is some functional unit that can perform processing. The PU's are indicated as such to show relationship between instructions, data, and the processing of the data.

Not being forced to produce general purpose processors, graphics card manufacturers, pushed the development of wide SIMD hardware implementation until reaching the capabilities of modern GPUs and eventually providing the possibility to use such kind of processors for "more general" computations not strictly related to graphics. Consequently in the last years some efforts to slightly modify GPUs architecture have been done to make them more flexible and some standard frameworks for software developments have been released.

3.2 PROGRAMMING API & LANGUAGES

FANN (Fast Artificial Neural Network Library), for example, attempted to run part of its code on GPUs using the OpenGL library

In the past some software projects tried to use standard graphics libraries like OpenGL (*Open Graphics Library*) to use the graphics functions provided to execute non graphics computations on GPUs. Anyway, such approach did not spread, since not all computational problems that may benefits from running on GPUs can be translated in "graphical problems" solvable by the use of graphical functions.

One of the first attempts to provide a standard interface and language to program GPUs, to execute generic computations on them, dates back to the 2004, with the release of *BrookGPU*, the Stanford University Graphics group's compiler and runtime implementation of the Brook stream programming language [11].

Anyway, despite its BSD license that could help it to rapidly develop, it did not have a lot of success and has been in *beta version* for a long time without becoming of common use; afterward the development apparently stopped in the 2007.

Apart from the early experimentations, the first popular software to provide GPGPU functions to developers, has been created by NVIDIA, that in November 2006 introduced and later in February 2007 released, the first CUDA™ SDK (*Software Development Kit*).

3.2.1 Compute Unified Device Architecture

CUDA (*Compute Unified Device Architecture*) provides to developers the possibility to add to ordinary software codes some parts to be run on the GPU. This is done using *'C for CUDA'* (C with NVIDIA extensions), compiled through a PathScale Open64 C compiler that generates binary code suitable for all the modern NVIDIA GPUs [32].

The main idea on which the CUDA SDK is based is illustrated in figure 27 on the following page and can be summarized in 4 different steps that can be executed from an ordinary C/C++ or FORTRAN code, using some functions implemented in the CUDA shared library:

1. Copy data to be processed by the GPU, from the host memory to the GPU one.
2. Configure the GPU setting all the configurations and arguments needed to execute the computation on the right portions of data.
3. Launch the computation on the GPU instructing it to execute a *kernel* with the arguments provided at the step before. This kernel is a function written in *C for CUDA* and compiled with PathScale Open64 C compiler.
4. Once the same kernel has been executed on various cores on different portions of data, the results can be copied back to the host memory.

Also CUDA did not spread as much as expected, even if it is much more used for GPGPU than OpenGL or BrookGPU. A possible reason is that being an NVIDIA Corporation creation, it is intended to work only on NVIDIA graphic cards. Indeed

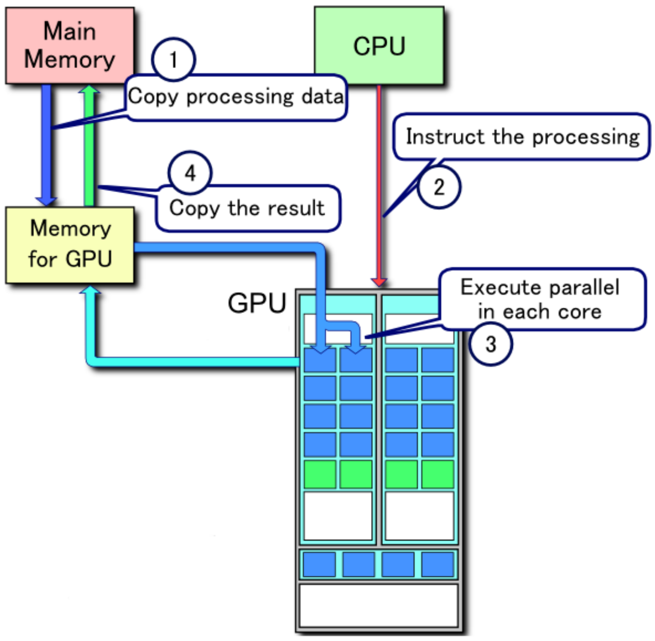


Figure 27: Processing flow with CUDA

in the field of GPUs, in contrast with the CPUs one, there is a lack of architectures standardization, making it impossible to let NVIDIA's CUDA implementation works on other brands cards. On the other side, other manufacturers have no interest in the implementation of a CUDA compatible driver for their boards being CUDA a property of one of their competitors.

The limitation of CUDA running only on NVIDIA boards, postponed the research on the PSA algorithm implementation for GPUs. Being AGATA a detector expected to run for the next decades, a limitation to only one manufacturer implementation of a so young technology, is a serious risk because NVIDIA could decide to stop supporting CUDA in favour of a more established standard.

3.2.2 Open Computing Language

Fortunately several companies operating in the computing industry being interested on GPGPU understood the problem and decided to define an open standard for a *middle-ware* software to interpose between GPUs drivers and user software, as has been done in the past with OpenGL. These companies leaded by Apple inc. through the Kronos Group[2] created the OpenCL (*Open Computing Language*) royalty-free standard [4].

This has been a fundamental step also for AGATA because the idea of a PSA algorithm implementation for GPUs was born in 2007, but it started to be actually developed only in 2009 after the release of the first OpenCL specification (December 8, 2008) [4] and the first beta version of the NVIDIA OpenCL implementation (April 20, 2009) for its boards [34, 35].

The creators of OpenCL evidently treasured the experience of NVIDIA with CUDA and in fact OpenCL is based on the same model exposed for CUDA, and the steps illustrated in figure 27 on the preceding page can be considered perfectly suited also for OpenCL.

Conceptually the greatest difference between CUDA and OpenCL is that the first is provided as an SDK (*Software Development Kit*), while the second is "only" a specification that in particular is composed of three parts:

- A *language specification*, defining an extension of a subset of ANSI C99 language to program kernels to be run on GPUs. Manufacturers had to provide JIT (*Just In Time*) compilers from this language to their processors bytecode.

- A *platform layer API*[3] that gives to the developers a set of functions to access to routines to query and give commands to the devices in the system.

- A *runtime API*, that allows the developers to queue up compute kernels for execution and manage the computing and memory resources in the OpenCL system.

Those parts define what the GPUs manufacturers have to provide to let third party software run on their hardware using OpenCL.

NVIDA, for example, added inside the CUDA SDK a compatibility layer enabling OpenCL software support, that actually uses the CUDA architecture previously implemented, as illustrated in figure 28 on the following page.

2 The Khronos Group is a member-funded consortium focused on the creation of royalty-free open standards for parallel computing, graphics and dynamic media on a wide variety of platforms and devices. Some of the consortium members are: Apple, NVIDIA, AMD, IBM, etc.

3 API stands for *Application programming interface*

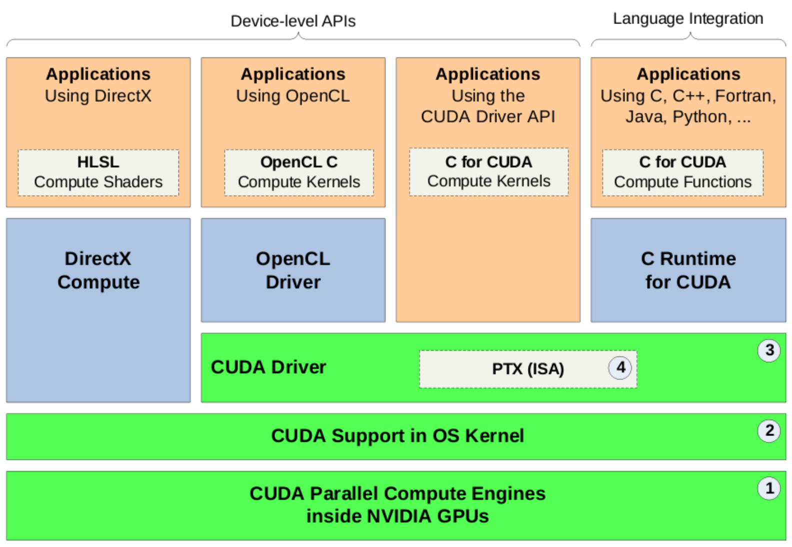

Device-level APIs
Language Integration
Applications Using DirectX
HLSL Compute Shaders
Applications Using OpenCL
OpenCL C Compute Kernels
Applications Using the CUDA Driver API
C for CUDA Compute Kernels
Applications Using C, C++, Fortran, Java, Python, ...
C for CUDA Compute Functions
DirectX Compute
OpenCL Driver
C Runtime for CUDA
CUDA Driver
PTX (ISA)
4
3
CUDA Support in OS Kernel
2
CUDA Parallel Compute Engines inside NVIDIA GPUs
1

Figure 28: Scheme of the CUDA software architecture showing how the OpenCL implementation has been inserted. ① Parallel compute engines inside NVIDIA GPUs. ② OS kernel-level support for hardware initialization, configuration, etc. ③ User-mode driver, which provides a device-level API for developers. ④ PTX (*Parallel Thread Execution*) ISA (*Instruction Set Architecture*) for parallel computing kernels and functions. *Image courtesy of NVIDIA Corporation.*

Together with the specifications, the Khronos Group, provides also a lot of documentation explaining the conceptual abstractions introduced by OpenCL, which in contrast to CUDA, aims to provide an interface suitable to program various kind of processors, multi-core CPUs, some FPGAs and not only GPUs.

The Platform Model

The platform on which OpenCL is supposed to run, is composed by a host (a computer) connected to one or more "OpenCL devices", allowing applications to use the host and the OpenCL devices as a single heterogeneous parallel computer system.

Indeed an OpenCL device can be a CPU, a GPU or some other kind of processor. These processors are divided into one or more CUs (*Compute Units*) which are further divided into one or more PEs (*Processing Elements*, referred before as PUs *Processing Units*).

An OpenCL application runs on the host machine and it can submit commands from the host to execute computations on the processing elements within a device. The processing elements within a compute unit execute a single stream of instructions as

SIMD units or as SPMD[4] (*Single Process, Multiple Data* or *Single Program, Multiple Data*) units.

This level of abstraction grants the possibility of writing OpenCL applications capable of exploiting all the processors (CPU, GPU, vector processors, etc.) that they may find on a host machine, no matter the manufacturer or the internal hardware architecture. What is needed is only an OpenCL implementation provided by the manufacturer (or a third party) for a specific processor. Different processors may be more or less efficient for a particular application, but a well programmed OpenCL application may "decide" which processor, on the host machine is better suited for running each part of its code.

The Memory Model

As different devices may implement different memory models, OpenCL provides a standardization defining some "kind" of memory space that may be available on supported devices. Some of them may not be available on some devices, or may have different size and throughput.

OpenCL defines four memory spaces that are summarized later and showed as a memory hierarchy diagram in figure 29 on the next page:

- *Private Memory* can be seen as registers in ordinary CPUs; a fast small sized memory, accessible only by the thread running on the PU the memory belongs to.

- *Local Memory* is accessible by a group of threads and can be used to share data among different threads.

- *Constant Memory* is a read-only memory that is writable only by the host system before the execution of a Kernel. It can be used as a fast cache to store data that has to be readable by all the threads running on all the CUs of one device.

- *Global Memory* is the mass memory, usually is off-chip and is readable and writable by all the threads running on the same device. Usually is the biggest, but slowest memory space available.

4 In SPMD, multiple autonomous processors simultaneously execute the same program at independent points, rather than in the lockstep that SIMD imposes on different data. SPMD is a subcategory of MIMD (*Multiple Instruction stream, Multiple Data*) in the Flynn's taxonomy. Different cores of a multi-core CPU execute instruction as SPMD.

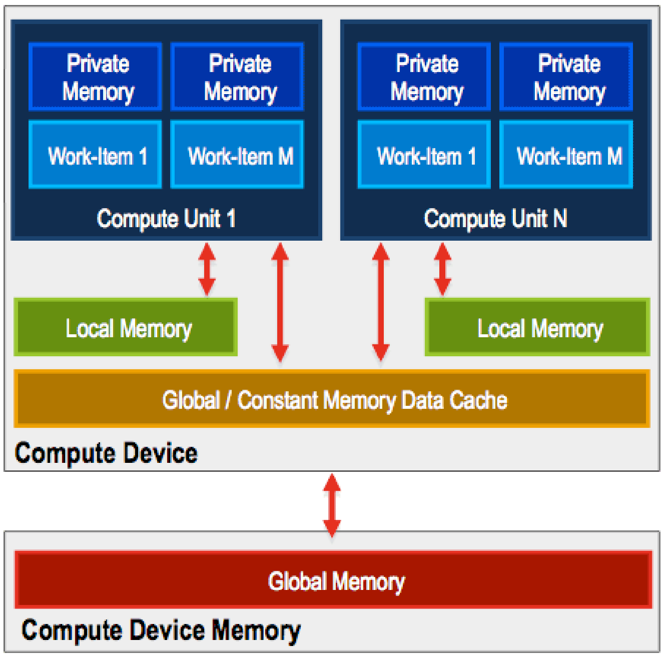


Figure 29: OpenCL Memory Model

The Execution Model

Compute kernels can be thought of either as data-parallel (SIMD paradigm), which is well-matched to the architecture of GPUs, or task-parallel (SPMD paradigm), which is well-matched to the architecture of CPUs.

A compute Kernel is the basic unit of executable code and can be thought of as a function.

In terms of organization, the execution domain of a kernel is defined by an N-dimensional computation domain. This lets the system know how large of a problem the user would like the kernel to be applied to. Each element in the execution domain is called a work-item and it can be seen as a single thread.

OpenCL gives the possibility to group together work-items into work-groups for synchronization and communication purposes; every work-group runs on a CU and every work-item running in a PE, belonging to the same work-group, can access the same Local Memory space. On devices without physical Local Memory space, it will be mapped on Global Memory, obtaining lower performances, but transparently from the programmer point of view.

3.3 HARDWARE ARCHITECTURE

Apart from the standard software organization that OpenCL defines in order to grant the possibility to run on different devices, the optimization of an algorithm should take into account also the hardware architecture where it has to be run. In the next page an overview will be given on the graphics cards used and about the NVIDIA's GPUs architecture in general.

Thanks to the diffusion of NVIDA graphic cards at the Legnaro National Laboratories where AGATA is hosted and also to the fact that NVIDIA appeared the strongest GPGPU supporter company when this project started, this hardware brand has been chosen for the PSA algorithm optimization.

Despite the choice of using this manufacturer cards, a particular care has been taken to ensure the portability of the algorithm's code also on other devices.

3.3.1 NVIDIA Quadro FX 1700

The first implementation of a PSA algorithm exploiting GPUs computational power has been developed on the NVIDIA Quadro FX1700 that was installed on an available workstation computer.

The NVIDIA Quadro FX1700 is a graphic card architected for engineering and design professionals. It mounts the G84GL GPU chip and is not intended for high intensity 3D computations (like the GeForce series) or for scientific computation (like the Tesla or Fermi series). Anyhow it is supported by the NVIDIA CUDA/OpenCL implementation and it was well suited to perform some feasibility and preliminary tests.

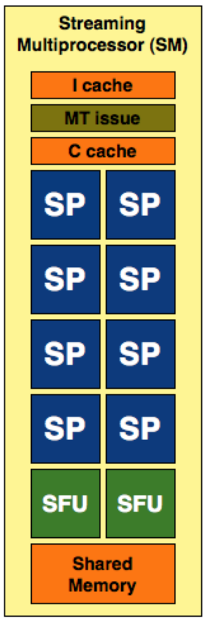
The specifications of the Quadro FX1700 board are listed in table 1 and its theoretical computing power is about 58.88GFLOP/s. This is lower than the theoretical computing power of modern multi-core CPUs (as can be seen from figure 24 on page 32).

Table 1: NVIDIA G84GL (NVIDIA Quadro FX1700) Specifications

SMs		**Memory**			
SPs	Clock	Bus Width	Bandwidth	Type	Size
32	400MHz	128bit	12.8GB/s	DDR2	512MB

In NVIDIA GPUs, Processing Units are called SP (*Straming Processor*) and are grouped 8 by 8, sharing 16kB of Local Memory[5], in units called SM (*Streaming Multiprocessor*) as shown in figure 30. In the OpenCL taxonomy every hardware SM maps to an abstract CU while every one of the 8 SPs (or cores) that compose an SM, maps to a PE.

CU (Compute Units) and PE (Processing Element) definitions were given in section 3.2.2



The image on the left shows a schematic view of the content of a Symmetric Multiprocessor unit of a NVIDIA GPU. Every SM contains:

- 1 Instruction Cache (Orange)
- 1 Multi-thread Instruction Issuer Unit (Brown)
- 1 Constants Cache (Orange)
- 8 Symmetric Processors (Blue)
- 2 Special Functions Unit (Green)
- 1 Shared Memory space (Orange)

Figure 30: NVIDIA Streaming Multiprocessor architecture. *Image courtesy of http://www.anandtech.com/.*

Every SM, referring to figure 30, does not contain only SPs, but also some caching, control and math-coprocessor units.

The multi-thread instruction issuer unit has the task to dispatch instructions to the different threads running on the 8 SPs of the SM, using the instructions and constants caches. The 8 SPs are the cores where the threads are executed, while the two SFUs (*Special Functions Unit*) are used to execute a subset of floating-points instructions not implemented in the SPs cores (every SFU contains for example 4 floating-point multipliers).

In the latest NVIDIA GPU chips models, SMs contain also a double-precision units to remedy to the lack of double-precision support of the SPs; in the future double-precision instruction will probably be incorporated inside SPs.

In addition, NVIDIA GPUs chips group SM in bigger groups called TPC (*Thread Processing Cluster*, often called in the graphic field *Texture Processing Cluster*), that may comprise two or more

5 The value of 16kB for the Local Memory size is for the cards used for this project, it may change on different models.

SMs. The G84GL chip has every TPC formed by two SM, and consequently embed two TPC (32 SPs in total). In spite of this, for example the GT200b chip has every TPC formed by three SM, as is shown in figure 31.

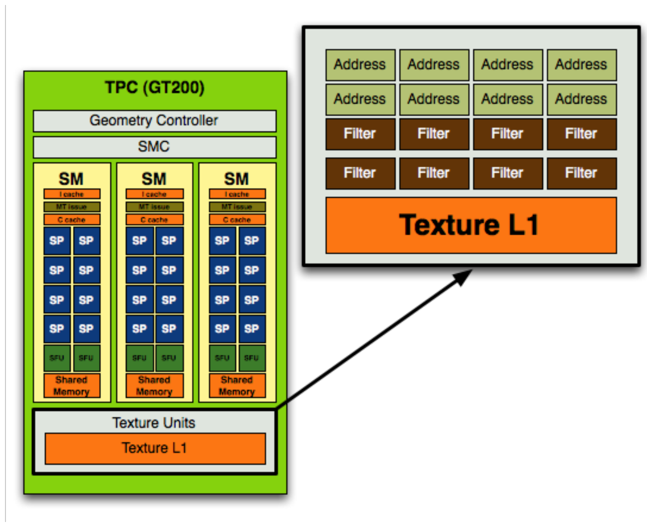


Figure 31: NVIDIA GT200/GT200b GPU chip Thread Processing Cluster scheme. *Image courtesy of http://www.anandtech.com/.*

Every TPC provides to the contained SMs a Geometry Control Unit[6], an SMC (*Symmetric Multiprocessor Controller*) handling external memory access (load, store, atomic functions) and an on-chip L1 cache to store Constant Memory read-only data.

3.3.2 ASUS ENGTX 285

After the initial tests, which confirmed the feasibility of the project, the development of the PSA algorithm for GPUs proceeded on a more performing graphic card: the ASUS ENGTX285, mounting the NVIDIA GTX285 GPU chip codenamed GT200b.

This card, has been purchased for the scope, since it is an affordable card architected for intense 3D computation, and it uses a slightly newer GPU chip with respect to that used in the more expensive Tesla C1060 card.

ASUS ENGTX285 is architected mainly for modern computer games.

Tesla series card, produced by NVIDIA and not by third parties companies, are specifically produced for GPGPU scientific computing [30] but, despite the guarantee of better memory quality and quantity, they rely on the same chips used on the

6 The Geometry Control Unit, is called so, because it was used in graphics computations to manage input and output vertex data to the SMs. In GPGPU it is instead used as a generic input/output dispatcher unit.

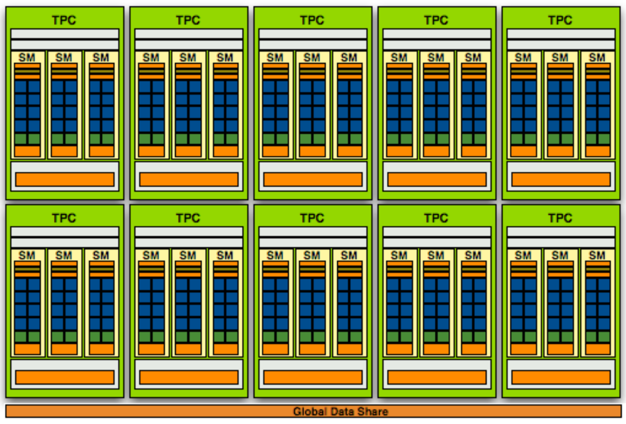


Figure 32: NVIDIA GT200/GT200b GPU chip global architecture. *Image courtesy of http://www.anandtech.com/.*

GeForce series. NVIDIA claims that in future the GPU chips used for the Tesla series will follow a different development path, according to the GPGPU computational needs.

The PSA algorithm has been developed on the ASUS ENGTX285 card that is equipped with 240 SPs (or PEs or cores) divided in 30 SMs (or CUs), further divided in 10 TPCs, where every TPC contain three SM as can be seen in figure 31 on the previous page and in figure 32. Clock and memory frequencies for this card are plotted in table 2 giving an impressive theoretical computing power of 1062.72GFLOP/s.

Table 2: NVIDIA GT200b (ASUS ENGTX285) Specifications

SMs		**Memory**			
SPs	Clock	Bus Width	Bandwidth	Type	Size
240	648MHz	512bit	159.0GB/s	GDDR3	1GB

3.3.3 SIMT Model

NVIDIA claims [30, 32] of having implemented in its GPUs (programmable with CUDA or OpenCL), a processing model that is an improved variant of the SIMD model (Flynn's taxonomy) used in previous GPUs, named SIMT (*Single Instruction, Multiple Thread*).

Even if it can be ignored for the programming correctness point of view (and indeed is not exposed in OpenCL), this NVIDIA variant must be taken into account to achieve best performance. Consequently, even though processing models have already been discussed in section 3.1 on page 33 and 3.2.2 on page 38, the strict relation of the SIMT model with the NVIDIA architecture, forces this subject to this section.

The main idea behind this model is the *warp* concept: every SM's multi-thread instruction issuer creates, manages, schedules and executes threads in group of 32 parallel threads called warps.

The term warp originates from weaving, the first parallel thread technology

Every SM (in the GT200 chip) manages a pool of 32 warps (32 threads each), for a total of 1024 threads, that multiplied for 30 SM give a maximum number of 30,720 concurrent threads for one GPU chip.

Every thread in the same warp executes the same instruction at the same time. If one (or more) of the threads in the same warp stops (for example waiting for a memory fetch), the whole warp stops its execution and another warp ready to execute is loaded on the SM with no context switching overhead. Every thread in the same warp executes with its own independent registers state, but if different threads in the same warp diverge, due to a data-dependent conditional branch, all the different paths executes serially, lowering performances. When all the different paths in the same warp complete the threads converge to the original execution path.

However, branch divergence occurs only within the same warp, since different warps can execute different data paths with no performance degradation.

As mentioned before, the programmer may ignore the warp concept while dealing with programs correctness, but should take it into account for performance tuning, in particular should take care to avoid thread branch divergences within the same warp of threads.

The main difference between SIMT and SIMD is that in the SIMD model every instruction is applied to a different trunk of data, exposing the size of the trunk to the software, while SIMT instructions are executed by different independent threads in parallel, that may take different branches.

4 PSA ALGORITHM OPTIMIZATION

As introduced in section 1.2.3 on page 10, the Pulse Shape Analysis algorithm analyzes the shapes of the pulses from all the electrical contacts of a segmented detector and gives as an output the points where the interactions causing the pulses occurred.

The PSA algorithm used for AGATA is called *GridSearch* and is provided in two versions: the Full GridSearch Algorithm [40] and the Adaptive GridSearch [47] one.

In this chapter a detailed description of the algorithm will be given and then the core of this dissertation work will be treated, illustrating the development choices to implement a GridSearch Algorithm for GPU devices.

4.1 GRIDSEARCH ALGORITHM

Several PSA algorithms to calculate interaction positions inside segmented γ-ray detectors have been developed in the past [8, 40, 47]. On the basis of the position resolution obtained during the first test experiments [40] and processing speed tests, the GridSearch algorithm [10] by Roberto Venturelli has been chosen for the AGATA array. In particular this thesis is focused on the optimization of the Full GridSearch algorithm that, despite its simplicity, is granted to provide a global optimum, in contrast to the faster, but potentially less precise Adaptive GridSearch.

Anyway research of new algorithms is still going on [41]

The kind of data that are processed by the PSA algorithm are here exemplified by the experimental event reported in figure 33 on the next page. In this figure all the signals from the 36 segments are reported together with the signal from the central contact.

The segments with a net charge collected are C4 and F3

Actually in this implementation the event is assumed to have only one interaction in the same segment, but in future developments also multiple interactions have to be considered.

The GridSearch algorithm was originally developed and tested on the experimental data from the MARS in-beam experiment.

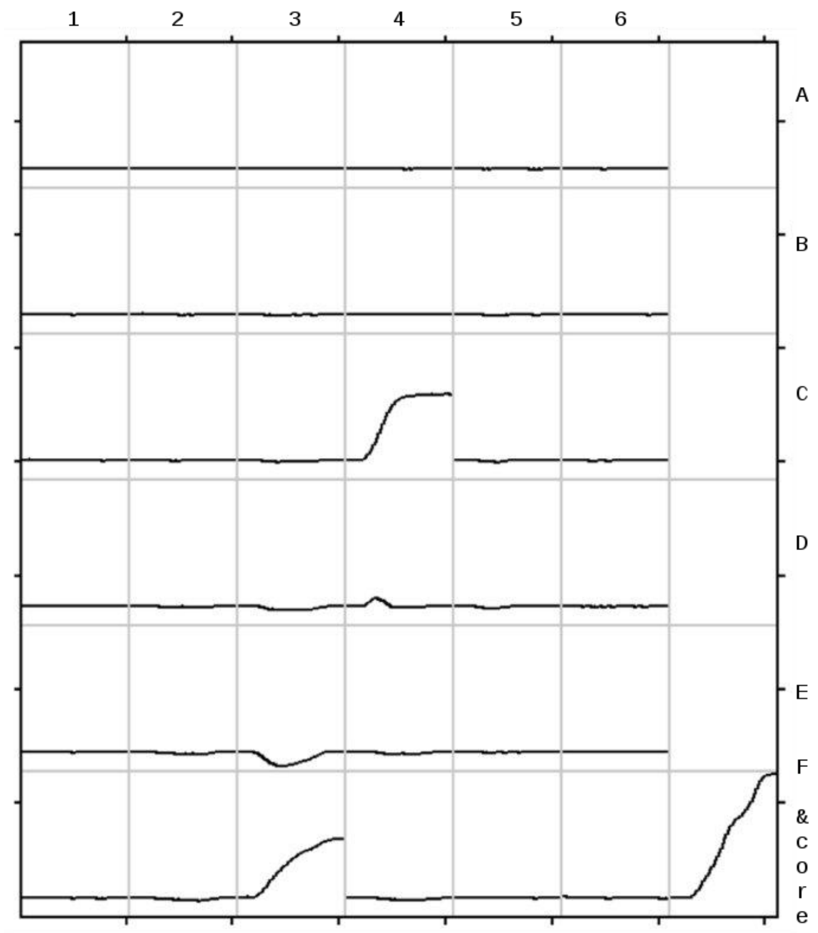


Figure 33: Signal waveforms of the 36 segments given as input to the PSA algorithm showing an event (interaction) inside two segments. Transient signals can be noticed in the surrounding segments. In the lower-right corner also the core signal waveform is plotted.

The original implementation was modified to cope with the new AGATA data format and it was further optimized as well. The algorithm is based on the comparison between measured net and transient signals of the segments and calculated signals from a fine grid of points in the crystal. The calculated signals (often called reference basis), discussed in section 1.2.4 on page 13, are grouped in a sort of database of signal waveforms, where for every point of the ideal 3D grid inside the detector, is stored a record. These records contain the signal waveforms that would be read on all the segments in correspondence of an interaction at the grid point the record is about.

Although it can assume one or two interaction points per segment, in its simpler implementation the GridSearch method searches for just one interaction point per firing segment. It has been verified in Monte Carlo simulations that the single interaction hypothesis is correct in more than 95% of the cases.

Using simulated signals, it was proven that, in case of multiple interactions within one segment, the result of this algorithm is indeed a fictitious single interaction point positioned at the center of gravity of the real interactions. An energy equal to the sum of the individual energy depositions is assigned to such fictitious interaction point. This justifies the approximation considered with the simulated data, namely packing the interaction points within the same segment. The signal comparison is done by evaluating the following FoM (*Figure of Merit*):

$$FoM = \sum_{j \in \text{Segm}} \sum_{i=T_0}^{T_{\text{end}}} |S_{ij}^m - S_{ij}^b|^p \tag{4.1}$$

The algorithm has to find the basis record that minimize the evaluation of the fiugre of merit (FoM) shown in formula 4.1, where the j index iterates on the segments, i index iterates on the samples of the waveform (starting from the T_0 first sample to the T_{end} last one), S_{ij}^m is the i^{th} sample of the j^{th} segment measured waveform, while S_{ij}^b is the corresponding basis sample and p is a positive number.

It can be proven that this figure of merit is a metric and for instance an exponent $p = 2$ will make it correspond to the Euclidean metric. The p exponent in particular has been adjusted experimentally, showing that best results can be obtained for $p = 0.3$ that is the value actually used.

This parameter, as well as other parameters entering the algorithm, has been adjusted in order to minimize the peak FWHM following Doppler correction, using the point position to infer the photon direction, or in other words to produce the best effective energy resolution. For the events where more than one segment was firing, a hit pattern ("neighbouring pattern" in the following) was chosen to avoid the use for a given interaction of the transient signals due mainly to another one. A possible neighbouring segment pattern for multiple interactions is depicted in figure 34 on the following page.

This segment pattern is deduced from simple geometrical considerations, however in some cases it is possible to improve the performance by modifying it. Consequently, it has to be a modifiable parameter for the GridSearch algorithm.

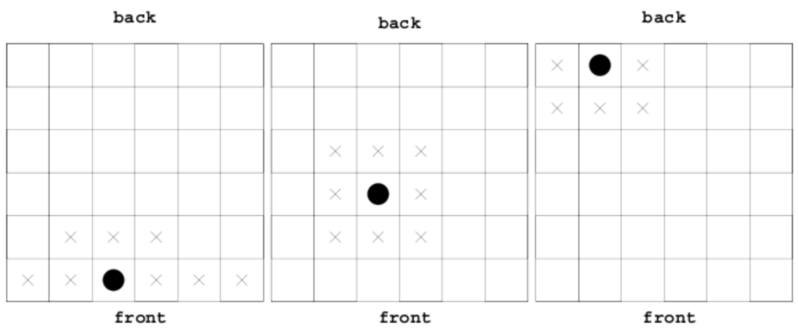


Figure 34: Pattern used to define the set of neighbouring segments for events with more than one segment firing. The closed-end shape of the detector is reflected in an extended neighbouring pattern when a frontal segment is hit.

In order to deduce the position of an interaction, its full set of neighbouring segments is used, except the segments where another net-charge signal is present. The best performance is obtained using for the FoM calculation also the segments where the transients of two interactions are overlapping, i.e. a further reduction of the set of neighbouring segments worsens the peak width. Recent versions of the algorithm allow to search for the interaction points in decreasing energy order, subtracting at each step the resulting basis signal from the experimental data, in an iterative way.

The signal registered from the central contact and from the net-charge segment are not used since it turns out that their inclusion in the FoM calculation results in a worsening of the energy resolution obtained using the hit pattern in figure 34.

The comparison of a signal with the basis is done independently of the position of the tested point inside the detector and the result for the matching of a certain point does not depend on the matching of the neighbouring points. Hence the algorithm has no particular requirements about the geometry of the grid of calculated signals, allowing the use of irregularly spaced signal basis which can be constructed with a density distribution matching the position sensitivity of the detector [22].

As introduced before, different versions of the algorithm were developed in order to meet various experimental situations.

A possibility, if enough computing power is available, is to search for two interactions within the same firing segment. When two points are searched, S^b_{ij} is a linear combination of signals for two possible points in the real segment while their amplitudes represent the energy partition between the two deposits. While searching for the position of a single interaction inside a segment is a 3-dimensional problem, the search for two interactions is a 7-dimensional problem: $x_1, y_1, z_1, x_2, y_2, z_2$ and the energy partition $k = E_1/E_2$ between the two interactions should be estimated.

Thus, the algorithm turned out to be significantly slower than the version searching for single interaction points. On the other hand, it turned out that its performance in terms of the resulting Doppler correction is not better than the case where only a single interaction point is searched. Therefore, our results were produced assuming only one interaction per segment.

Finally, it should be pointed out that variants of the algorithm have been implemented in order to speed up the search in case the computing time is an issue, such as during the on-line analysis. Such variants rely on adaptive methods based on a first rough search, which can be a parametric one or a grid search on a coarsely-spaced grid, followed by a fine search in the region identified by the rough search (like the mentioned Adaptive GridSearch).

Actually the Full GridSearch algorithm for CPUs can process about 300 events per second as illustrated in table 3 on page 73

This solution however gives worst results than the extensive grid search, since it is not granting to find optimal FoM minimums, but may find local ones.

Another solution to move around the execution speed problem has been to reduce the acquired and basis data precision, available as floating point values, processing it inside the PSA algorithm as 16-bit integers.

A more performing PSA algorithm, or a more performing hardware to execute it, would permit to exploit all the available information and the realization of more sophisticated techniques to analyze multiple interactions.

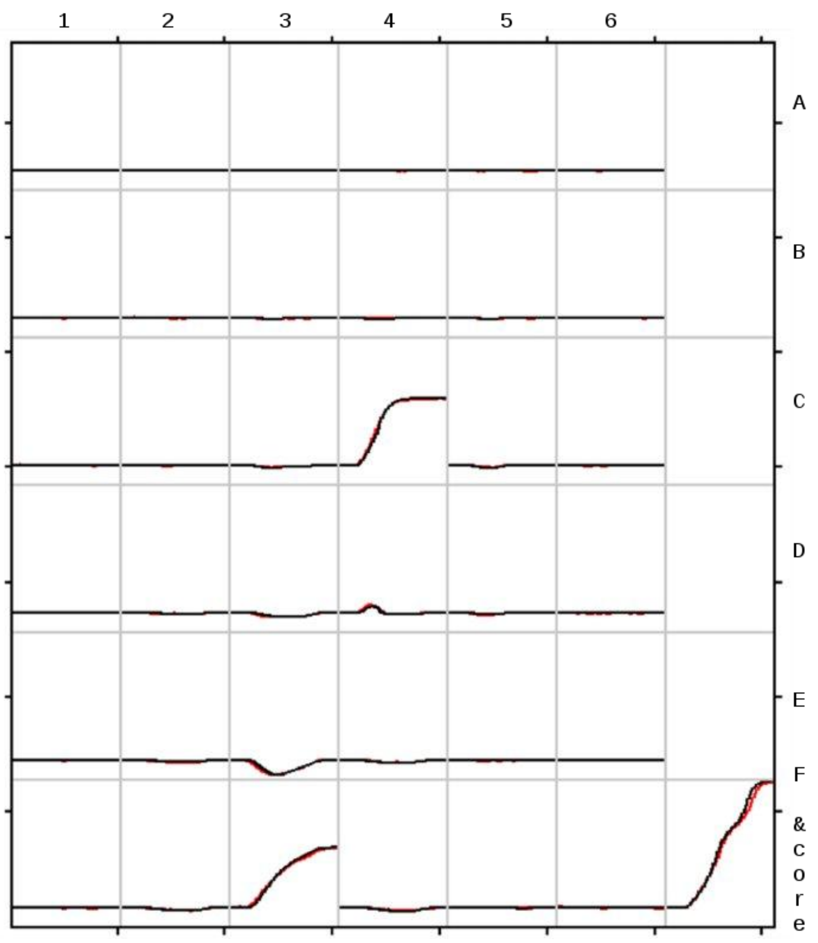


Figure 35: Signal waveforms of the 36 segments given as input to the PSA algorithm plotted in black and the closest waveform set contained in the basis database found by the PSA algorithm plotted in red. In the lower-right corner the core signal waveform.

4.1.1 GridSearch for CPU

In listing 2 is reported the core part of the C++ code used to run on the CPU the GridSearch algorithm

The C++ code that implements the figure of merit calculation for one event, before the optimization attempt using GPUs, is given in listing 2. On line 10, `pS->sAmplitude[iSegm]` is a pointer to the acquired signal coming from the `iSegm` segment, so the experimental signal.

On line 11, `fBasis.Pts[netChSeg][iPts].amplitude[iSegm]` is a pointer to the reference basis signal that would be read from segment `iSegm` if an interaction would occur in the point `iPts` of the net-charge segment `netChSeg`.

The actual GridSearch library is composed by more than a thousand lines of code, here is reported only the core of the algorithm that needs to be optimized

Listing 2: Main Full Grid Search loop

```
 2  for( int iPts=0; iPts < nPts; iPts++) { // loop over the base points

 4      chi2 = 0;

 6      for( int iSegm=0; iSegm < NCHAN; iSegm++) { // loop over the segments

 8        if(lMask[iSegm] != '0') {

10          short * realTrace = pS->sAmplitude[iSegm];
11          short * baseTrace = fBasis.Pts[netChSeg][iPts].amplitude[iSegm];
12          realTrace += samp_first;
13          baseTrace += time_first;

15          for(int nn = 0; nn < uSamples; nn++) {
16            chi2 += metrics[(*realTrace++) - (*baseTrace++)];
17      }

19          if(chi2 > chi2min)
20            break;
21        }

23      }  // end loop over the segments

25      if(chi2 < chi2min) {
26        bestPt  = iPts;
27        chi2min = chi2;
28      }

30  }  // end loop over the base points iPts

32  pS->bestPt  = bestPt;
33  pS->chi2min = chi2min;
```

The outer cycle at line 2 in listing 2, as mentioned in the commented text, loops over the points of the grid inside the net-charge segment where the interaction occurred. At every cycle it compute the FoM of formula 4.1 on page 49 for the current point. The second nested cycle at line 6, loops over the segments and correspond to the outer summation in formula 4.1 on page 49, while the inner cycle at line 15, loops over the signal samples and correspond to the inner summation in formula 4.1 on page 49.

The partial FoM for every sample is calculated at line 16 and at the end the ID of the point with the smallest FoM with its FoM value get saved at line 32 and 33.

In summary this code is performing a "brute force" search comparing the signal recorded from the detector with all the possible corresponding signals in the database. Other details about the code will be given later showing the code optimization path followed.

4.2 OPENCL GRIDSEARCH IMPLEMENTATION

The GridSearch algorithm has been studied in detail in order to determine if the problem it solves can be parallelized and how to do it in the most efficient way for its execution on GPU devices.

It can be noticed that the core of the algorithm, reported in listing 2 on the preceding page, comprise an outer `for` loop (line 2) over the base points of the net-charge segment where at every cycle one point get "processed" and the value of its figure of merit get calculated. The FoM of every point can be calculated independently and in any order, thus allowing the subdivision of the problem in different threads where every thread process one point.

Since segments are of different volumes, the number of points in each of them is variable.

Consequently it can be concluded that the problem is parallelizable at least in a number of thread corresponding to the number points comprised in the net-charge segment. This means about one thousand for a $2 \times 2 \times 2$mm grid.

In order to predict the speed improvements obtainable from a GPU implementation the various possible bottlenecks should be considered.

There could be two memory access bottlenecks:

- The *PCI-Express* 16*x* *Gen2* connection from the host computer to the graphics card.
- The memory bus between the GPU chip and the on-board memory on the graphics card.

Other slowing factors could be:

- The parallelization of the problem in an insufficient number of threads to occupy all the cores available on the GPUs
- The use of mathematical functions that are not common in the graphics calculations and consequently that are not optimized in GPUs
- The presence of a lot of divergent branching threads (see section 3.3.3 on page 44)

The PCI-Express bandwidth that is 8GB/s is much more than the maximum 1Gbit/s of experimental data that may arrive inside one processing node due to the network speed and consequently it should not be a bottleneck. Moreover, to move the basis data inside the graphics card on-board memory, the PCI-Express link will be used only once, at the algorithm initialization and in this phase execution speed is not relevant.

The bandwidth from the GPU chip to the on-board memory, is more critical, since it will be used to read the basis database while calculating the FoM related to every net-charge segments' grid points.

The GPU to on-board memory bandwidth is specified as 12.8GB/s and 159.0GB/s for the Quadro FX1700 card and the ENGTX285 respectively. The memory accessed to calculate the FoM of every point is variable according to the number of segments considered in the neigbourhood of the net-charged one, but considering to compare an average of eight segments' waveforms per event with the acquired ones, the GridSearch algorithm would need to fetch $2 \times 60 \times 9 \times 1000 = 1.08$MB of data, to calculate the best matching point in a segment containing 1000 points. Giving an impressive result of more than 10 thousand and more than 100 thousand events per second for the Quadro FX1700 card and the ENGTX285 respectively, assuring that memory bandwidth will not be bottleneck.

The bytes to fetch are calculated considering 60 samples signals, using short data type values (2 bytes wide). The result has to be doubled for float data type values (4 bytes wide))

What can not be easily taken into account are the negative effects of the GPU on-board Global Memory latency (in the order of 400-600 GPU clock cycles) that heavily depends on the data access pattern, on the number of threads and on their mathematical complexity. These effects may prevent to reach the maximum available bandwidth.

The latency effects depend on the data access pattern since if threads with consecutive IDs access consecutive memory locations, most of the GPU chips (the used ones for sure) have the capability to execute a *coalesced memory load*, that result in a single memory access moving the biggest possible trunk of data, lowering the average latency perceived by the threads.

Indirectly also the number of threads and their mathematical complexity can influence the memory latency effects, hiding them. This because if a warp of threads is waiting for a memory load, the cores time that is wasted can be used to execute other warps of threads if they exists. Moreover, if threads have a high mathematical complexity, their execution can occupy more easily the cores while others are waiting for memory loads.

4.2.1 First Implementation

Assured by the fact that the GridSearch algorithm seamed easily parallelizable and that graphics cards bandwidth to the host and to its on-board memory was enough, the easiest way to test memory latency impact on performances and to adjust threads number and complexity, was to profile a running implementation written in OpenCL of the GridSearch core.

Some restraints to the possible code manipulations had to be considered during all the developing process. In particular, only the core part of the GridSearch algorithm could be re-organized, while all the rest should have been modified as less as possible since other developers not willing to deal with OpenCL complexity were still working on it.

Consequently the main idea to organize the development has been to add to the PSA library function `process_initialise()` (that is run by NARVAL only once at the system initialization, as illustrated in section 2.3.1 on page 28) the code to find, initialize and return handlers to the GPU device, or devices, found on the host machine and the code to load the basis database on the GPU's Global Memory as a read-only buffer.

The code to run the OpenCL kernel inside the GPU has been implemented inside the `SearchFullGrid()` function, loading the needed data like the acquired samples, the pointer to the array where to store the results, etc. to its on-board memory.

In listing 3 on the next page the first OpenCL Kernel code implementation of the GridSearch algorithm is reported. All the omitted host side code running on CPU can be found in the appendix A on page 79. As can be clearly seen the code is very conservative compared to the C++ version for CPUs reported in listing 2 on page 53 and only minor changes can be noticed; this has been done in order to state a reference implementation to be used as a base point to start the profiling and optimization[1].

This Kernel code is executed by every thread lunched on the graphics card and in fact the outer loop on points visible at line 2 in listing 2 on page 53 disappeared, since there will be one thread for every point in the net-charge segment running this code. In fact every thread at line 9 in listing 3 on the next page (using the `get_global_id()` OpenCL defined function), gets its

1 Following a famous Knuth's advise: «*premature optimization is the root of all evil (or at least most of it) in programming*»

Listing 3: First OpenCL GridSearch implementation using pre-calculated powers array.

```
__kernel void GridSearch(__constant const short* pSsAmplitude,
             __constant const char* lMask,
             __global const short* fBasis,
             __constant const float* metrics,
             __global float* chi2,
             const int netChSeg,
             const int nPts) {

  int iPts = get_global_id(0);
  if (iPts >= nPts) return;

  float this_iPts_chi2 = 0;

  for(int iSegm = 0; iSegm < NSEGS; iSegm++) {
    if(lMask[iSegm] != '0') {

      __constant const short* realTrace = pSsAmplitude + iSegm*USAMP;
      __global const short* baseTrace = fBasis +
                     ((netChSeg*MAXPTS + iPts)*NSEGS + iSegm)*USAMP;

      for(int nn = 0; nn < USAMP; nn++) {
        this_iSegm_chi2 += metrics[realTrace[nn] - baseTrace[nn]];
      }
    }
  }
  chi2[iPts] = this_iPts_chi2;
}
```

ID that in this implementation correspond to the grid point number for which the thread has to compute the FoM.

All the threads running, receive as arguments some pointers to memory areas in the GPU's on-board memory that may have been already loaded with data from the host side code[2].

The `pSsAmplitude` pointer to `short`, point to the memory space initialized with the experimental data of the event being processed, while `fBasis` to the basis. Both of the memory spaces are defined as `const` since they will not be changed by the algorithm, but they are defined as different OpenCL spaces; one as `__constant`, while the other as `__global`, since we want the experimental data to be cached, but not the basis data. This is due to the fact that the same experimental data ($2 \times 60 \times 36 =$ 4.320kB) is needed by all the threads and fits inside the cache memory that is 64kB wide, but basis data, besides of its dimension (60 samples $\times$ 2Byte $\times$ 36 segments $\times$ No. of whole detector 2mm-grid points $\approx$ 300MB), its samples will not be read more than once for every event, so it make no sense to cache its values processing one event at a time.

2 As explained in figure 27 on page 36 phase ①.

The `lMask` array is needed to avoid the inclusion in the computation of segments that are far from the net-charge one, that otherwise will simply add noise to the FoM calculation. Since `lMask` is a 36Byte array and is read by all the threads, it make sense to keep it in cache memory.

Which segments are to be considered "too far" is a parameter that may change and has not to be statically implemented.

All the results are stored in the `chi2` array and every cell of the array contain the FoM of the corresponding point; the smallest FoM will be found on the host side after coping it to the host memory.

The `netChSeg` and `nPts` variables contains respectively the number of the net-charge segment end the number of points contained in that segment. If the ID of a thread for some reason is bigger than the number of point contained in the current segment the thread returns (line 10 in listing 3 on the previous page).

It can be noticed that in listing 3 on the preceding page at line 22, the variable `this_iPts_chi2`, containing the partial FoM of the current segment, is not updated by an explicit calculation, but by reading a value inside an array called `metrics`. This array contains all the pre-calculated powers of the possible differences between acquired and basis samples and has been implemented to speed up the calculation for the CPU implementation, since the array can be easily cached and a cache read is much faster (15 clock cycles of latency and 2 clock cycle throughput) with respect to an absolute value plus an exponentiation (few hundreds of clock cycles) [3].

On different GPUs with bigger caches or different memory organizations this may change.

This revealed to be slow on the GPUs used, since the array does not fit inside the cache memory. Moreover, a read from the constant memory costs as a read from a local register only if all the threads in a half-warp read the same address. Unfortunately, this is not the case and hence the reads would be serialized anyway.

For this reason the first simple optimization has been the removal of the `metrics` array and its substitution it with the explicit calculation as can be seen in listing 4 on the next page (line 21 and 22).

The first version using the metrics array, running on a NVIDIA Quadro FX1700 graphics card was running at half the speed of a single Intel Xeon 2.5GHz core (about 150 events per second), while the second version with the explicit computation reached almost the speed of the CPU core (260 events per second). This is indeed a good result considering that the card used is not

Listing 4: First OpenCL GridSearch Implementation computing powers on the fly.

```
__kernel void GridSearch(__constant const short* pSsAmplitude,
             __constant const char* lMask,
             __global const short* fBasis,
             __global float* chi2,
             const int netChSeg,
             const int nPts) {

  int iPts = get_global_id(0);
  if (iPts >= nPts) return;

  float this_iPts_chi2 = 0;

  for(int iSegm = 0; iSegm < NSEGS; iSegm++) {
    if(lMask[iSegm] != '0') {

      __constant const short* realTrace = pSsAmplitude + iSegm*USAMP;
      __global const short* baseTrace = fBasis +
                  ((netChSeg*MAXPTS + iPts)*NSEGS + iSegm)*USAMP;

      for(int nn = 0; nn < USAMP; nn++) {
         this_iPts_chi2 += native_powr((float)abs_diff(realTrace[nn],
                              baseTrace[nn]), METRIC);
      }
    }
  }
  chi2[iPts] = this_iPts_chi2;
}
```

architected for intense calculation and it is rated as having a theoretical computing power in GFLOP/s of the same order of magnitude[3] of the Intel processors used for comparison.

4.2.2 Code Profiling

The *Visual Profiler* software provided by NVIDIA has been used to obtain more detailed information to understand if there were still inefficiencies. This software, runs the target code to be profiled for a predefined number of times collecting statistics about GPU's memory and cores usage and provides the users also with some suggestions about the kind of operations that could be slowing down the calculation.

A screenshot of a plot produced by the Visual Profiler that permitted to analyze the code reported in listing 4 is reported in figure 36 on page 61. This plot reports the percentage of counters registered by the profiler software executing the kernel,

3 Since the GFLOP/s value associated to a processor can be measured in various ways (e.g. according to the floating-point operation used), it really make no sense to compare it precise value measured by different companies; its order of magnitude instead can give an idea of the computing capabilities.

highlighting the fact that the code is not suffering about divergent branching (although the number of branches is high, since all the threads belonging to the same event are using the same `lMask` array) or warp serialization.

Another information produced by the Visual Profiler is the cores' *Occupancy*, expressed as a percentage, representing the kernel capability to occupy the available cores. In the case of the code reported in listing 4 on the previous page the produced value (on the Quadro FX1700) was of 33.3% executing work-groups of one thread (consequently running only 8 thread per SM) and 100% running work-groups of 128 threads. In this second case the number of work-groups, for a thousand points per event, become only about eight; remembering that a work-group can not be split on different SMs, the weakness of this code can be easily seen, since it will not scale on bigger graphics cards with a bigger number of SMs.

The problem of Occupancy will be extensively discussed in section 4.2.4 on page 64 looking at profiling data produced by the run of the OpenCL kernel on the ENGTX285 graphics card.

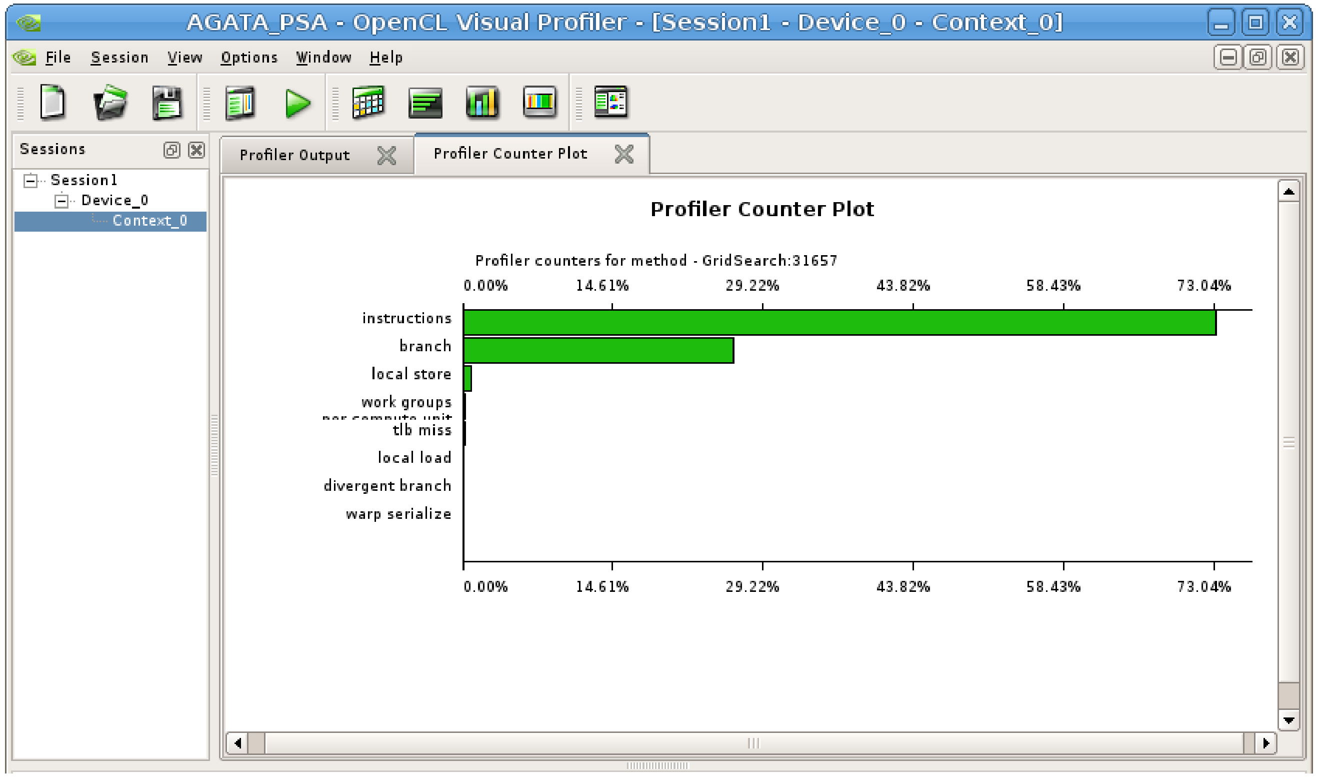


Figure 36: Profiler counters plot produced by the Visual Profiler software running kernel in listing 4 on page 59. This software while executing the OpenCL kernel collects statistics about various events; in this graph, the name of the event on the left is plotted against the value of its counter as a percentage over all the counted events. This graph gives various information, an important one is that in this kernel there was not warp serialization or divergent branching although there were lot of branches in comparison to the number of instruction executed.

4.2.3 Code Optimization

Thread indexing in OpenCL is not "mono-dimensional", but indexes are vectors of a maximum three coordinates. If only one dimension is used, the OpenCL function `get_global_id(0)` outputs the thread ID, but actually, being the thread index a vector of the form (x, y, z) where every dimension has a size (D_x, D_y, D_z), the thread ID (determining for example the threads grouping in warps) is calculated as $ID = (x + yD_x + zD_xD_y)$.

The 3-dimension indexing system provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume and can be exploited also to further divide the problem to be solved in work-groups.

The OpenCL GridSearch implementation has been modified in view of its use on a more performing graphics card, removing the nested `for` loops and augmenting the threads number. The problem has been further divided assigning to the first dimension the grid points (as before) and adding a second dimension representing the 36 segments corresponding to the iterations of the cycle at line 13 in listing 4 on page 59.

After this modification, to save in the `chi2` array the final results for every point and not all the partial FoM related to every segment and every point, some kind of synchronization and data sharing between threads cooperating to calculate the FoM for the same point is needed. The work-group concept exposed in section 3.2.2 on page 40, has been used in order to execute all the threads calculating the FoM of a segment, related to the same point, in the same work-group. In fact, running in the same work-group, threads can store the partial FoM related to "their" segment in the shared memory and before exiting apply a reduction algorithm to sum all the partial results and store the sum in the right cell of the `chi2` array.

The modified version of the code is reported in listing 5 on the next page in order to use a two-dimensional thread index; every thread has an index that is given by two numbers (x, y), where x span from 0 to `nPts`, while y span from 0 to `NSEGS` (that equals to 36: the number of segments).

Moreover, since the GPU used are 32-bit architectures, the use of `short` data type was a source of inefficiency, afterward in the version reported in listing 5 on the facing page, acquired and basis data is read as an array of vectors, where every vector contains two short. The GPU used are not vector processors, but scalar ones, consequently the operation at line 23 will be executed as two operations, one to the first vector's component and the second to the other, but as mentioned before, moving trunks of 32-bit instead of 16-bit ones, increased memory transfers efficiency.

Listing 5: OpenCL GridSearch Implementation using 2-dimensional thread indexing and reading data as *short2* vectors.

```
 1  __kernel void GridSearch2(__constant const short2* pSsAmplitude,
 2                            __constant const char* lMask,
 3                            __global const short2* fBasis,
 4                            __global float* chi2,
 5                            const int netChSeg,
 6                            const int nPts)
 7  {

 9    unsigned int iPts = get_global_id(0);
10    unsigned int iSegm = get_global_id(1);

12    if (iPts >= nPts) return;
13    if (iSegm >= NSEGS) return;

15    float2 this_iSegm_chi2 = 0.0f;

17    __local float local_chi2[36];

19    if(lMask[iSegm] != '0') {

21      __constant const short2* realTrace = pSsAmplitude + iSegm*USAMP2;
22      __global const short2* baseTrace = fBasis + ((netChSeg*MAXIPTS +
23                            iPts)*NSEGS + iSegm)*USAMP2;

25      for(int nn = 0; nn < USAMP2; nn++) {
26        this_iSegm_chi2 += native_powr(convert_float2(abs_diff(realTrace[nn],
27                            baseTrace[nn])), (float2)METRIC);
28       }
29    }

31    local_chi2[iSegm] = this_iSegm_chi2.s0 + this_iSegm_chi2.s1;

33    barrier(CLK_LOCAL_MEM_FENCE);

35    if (iSegm == 0) {
36      float this_iPts_chi2 = local_chi2[0];
37      for (int i = 1; i < NSEGS; i++) {
38        this_iPts_chi2 += local_chi2[i];
39      }
40      chi2[iPts] = this_iPts_chi2;
41    }

43  }
```

At line 33 in listing 5 can be noticed one of the synchronization function provided by OpenCL. This particular one, impose that every thread in the same working-group has to reach this instruction before anyone can continue; this grant the `local_chi2[iSegm]` array to be completed with all the partial FoM related to every segment. Between line 35 and 41 for every grid point x, only the threads with index $(x, 0)$ execute the reduction, summing all the `local_chi2[iSegm]` values produced by threads $(x, 0), (x, 1), \ldots, (x, 36)$ (that are in the same work-group) and writing the results in the `chi2` array's cell with index x, corresponding to the x^{th} grid point.

Other more sophisticated reduction algorithm demonstrated to be slower, probably because the sum concern only 36 elements.

4.2.4 Profiling and Occupancy Calculation

After the promising experience on the Quadro FX1700 graphics card, as mentioned before, the code has been tested on the ASUS ENGTX285 board. The program could run on the new device without any modification or code recompilation.

However, since the first board had 4 SMs and 32 SPs, while the second one had 30 SMs and 240 SPs, one question that arose is whether the proposed modification in listing 5 on the preceding page could correctly scale on the new device using all the computational power available. This question rises from the consideration that, although the threads number may be enough, they may be divided in work-groups with sizes that can not divide well among all the available cores.

The Visual Profiler provided some information that have been used not only to measure the reached occupancy, but also to calculate, with a tool called *Occupancy Calculator*, the theoretical occupancy of the SPs, according to the kernel needs of registers, shared memory and work-group size.

The Visual Profiler, was initially created for CUDA, hence still uses CUDA taxonomy, in its OpenCL version; thus for clearness, it must be remembered that a *Grid* is a collection of work-groups, *Block* is a synonym of work-group and thread (often used before) is a synonym of work-item.

Running the implementation reported in listing 5 on the previous page inside the Visual Profiler and then running its *Analyze Occupancy* function results in the following information for every GridSearch kernel call:

```
Kernel details : Grid size: 1080 x 1, Block size: 1 x 36 x 1
Register Ratio = 0.5  ( 8192 / 16384 ) [13 registers per thread]
Shared Memory Ratio = 0.25 ( 4096 / 16384 ) [180 bytes per Block]
Active Blocks per SM = 8 : 8
Active threads per SM = 288 : 768
Occupancy = 0.5  ( 16 / 32 )
Achieved occupancy  = 0.5  (on 30 SMs)
Occupancy limiting factor = Block-Size
```

Reading it from the first row, it can be noticed that threads were executed in work-groups of 36 work-items each and that in this particular GridSearch kernel call, the net-charge segment had 1,080 points and consequently 1,080 work-groups were launched.

The lines about registers and shared memory usage suggest that were used only 50% of the available registers and only 25% of the available shared memory, so these factors are not limiting cores occupancy.

Moreover it can be noticed that 8 work-groups were assigned to every SM (that is the maximum possible) and consequently $36 \times 8 = 288$ work-items were run on every SM.

The last three rows give the result of a maximal theoretical occupancy of 50% that is equal to the actual achieved occupancy, eventually suggesting to modify the work-group size in order to achieve a better one. This is due to the fact that no more than 8 work-groups can share a single SMs at the same time, therefore if eight times the work-items of a single work-groups are not enough threads to fully occupy the SMs cores, for some times they will be idle causing an inefficiency.

To know how to modify the work-group size to better optimize the code for this kind of device, the already mentioned Occupancy Calculator (that is a simple spreadsheet file) can be used. In figure 37 on page 67 is reported its main page where the user has to select the *Compute Capability*[4] of the device, the threads per work-group, the number of used registers by one thread and the shared memory used by one work-group. Once inserted all the needed information the Occupancy Calculator provide the user with some plots, reported in figure 38 on page 68.

The calculations accomplished by the Occupancy Calculator are based on some simple rules that for a GPU of Compute Capability 1.3 are:

- The total number of threads active on a SM is limited to a maximum of 32 per warps and a maximum of 32 warps per SM (1024 threads), but also by the available registers (16,384 32-bit registers) since every thread has its own private registers. Registers usage per SM is calculated as the register used by every thread multiplied by the number of threads (the actual allocated value has to be calculated

4 The *Compute Capability* is a number associated by NVIDIA to its GPU chip series identifying common physical limits like the ones in the gray box in figure 37 on page 67. The Quadro FX1700 has Compute Capability 1.1, while the ENGTX285 1.3.

according to the allocation unit sizes and allocation granularity provided in the gray box in figure 37 on the next page).

- The total number of work-groups sharing the same SM is limited by the number of threads they contain, since the total number of threads can not to exceed the previously mentioned limitations.

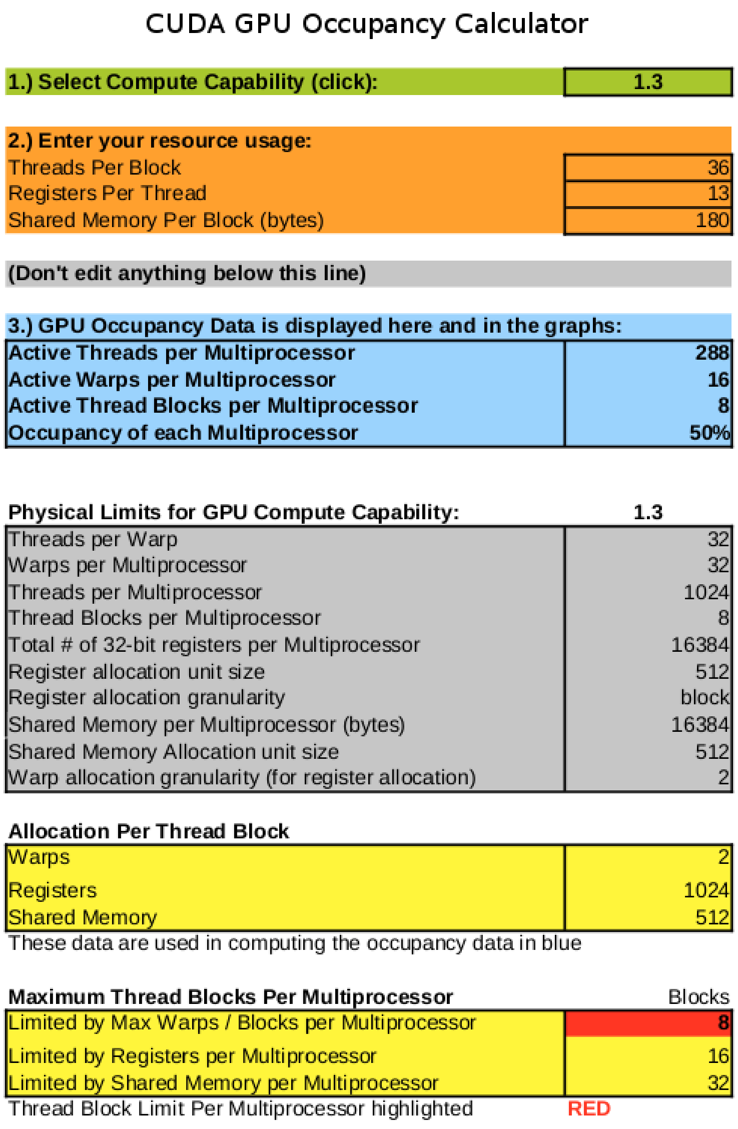
- The total number of work-groups per SM is also limited to 8 and by the shared memory they use, since $16,384$Bytes of shared memory is available for every SM and it has to be divided between the work-groups sharing the same SM.

In the graphs produced by the Occupancy Calculator and in particular in the first one, it can be noticed that increasing the number of threads per work-group a better occupancy can be achieved. Moreover the second graph indicates that increasing the usage of shared memory per work-group will not negatively effect occupancy till it will be less than $16,384/8 = 2,048$Bytes.

Following the Occupancy Calculator results, the code has been modified again to calculate three grid points for every work-group in order to augment the number of thread per work-group. The obtained code is reported in listing 6 on page 69 and as can be noticed, it contains also some other modifications like a more sophisticated reduction algorithm, a change in the vector data type used and some initial hints for the Just In Time OpenCL compiler that, providing information about the used vector data type and work-group dimensions, let it slightly optimize the produced code.

The used data type has been changed in this version, because basis and acquired data is available as floating-point values. In the CPU version of the GridSearch algorithm data was cast as `short` loosing precision in order to speed up the computation. Since with this GPU implementation the speed performance improved significantly, it has been introduced the possibility to decide at the library compile time if to process data as `short` (losing precision) or as `float`.

The user defined `gs_type` type can be defined as a `short` or as a `float` at compile time and thanks to the optimization of the GPU chips for floating-point data processing, a great improvement can be seen in table 3 on page 73 comparing speeds, operating with `float` between CPU and GPU.



CUDA GPU Occupancy Calculator

1.) Select Compute Capability (click):	**1.3**

2.) Enter your resource usage:	
Threads Per Block	36
Registers Per Thread	13
Shared Memory Per Block (bytes)	180

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	**288**
Active Warps per Multiprocessor	**16**
Active Thread Blocks per Multiprocessor	**8**
Occupancy of each Multiprocessor	**50%**

Physical Limits for GPU Compute Capability:	**1.3**
Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384
Register allocation unit size	512
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	2
Registers	1024
Shared Memory	512

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	8
Limited by Registers per Multiprocessor	16
Limited by Shared Memory per Multiprocessor	32

Thread Block Limit Per Multiprocessor highlighted RED

Figure 37: NVIDIA Occupancy Calculator main page

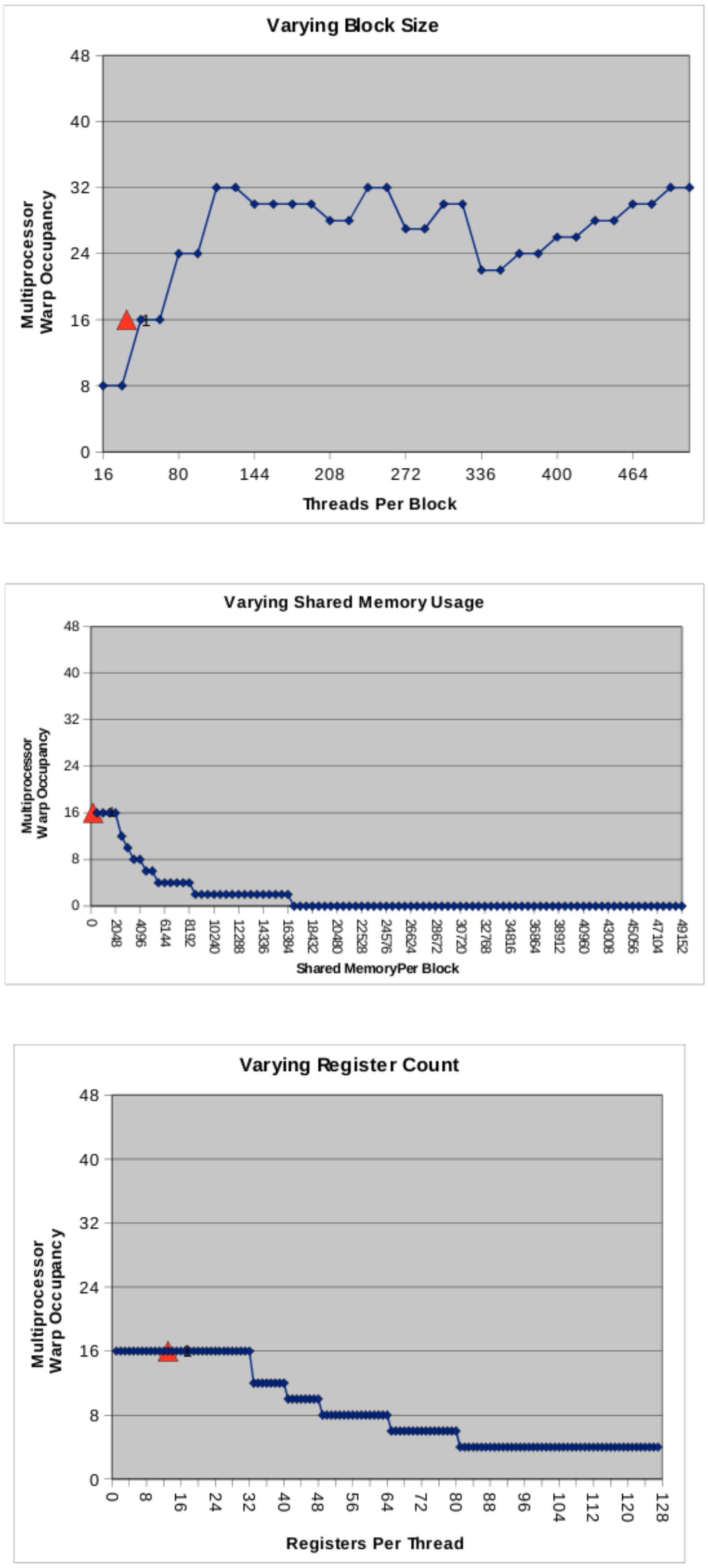


Figure 38: Plots produced by the NVIDIA Occupancy Calculator. Red triangles show the maximum number of allocable warps per SM limited by (from top to bottom): threads per work-group, shared memory used per work-group and registers used per thread respectively. The other data points represent the range of possible work-group sizes, register counts, and shared memory usage.

Listing 6: OpenCL GridSearch Implementation using 2-dimensional thread indexing, reading data as *gs_type4* vectors and calculating 3 points per work-group.

```
__kernel __attribute__((vec_type_hint(gs_type4)))
         __attribute__((work_group_size_hint(3, 36)))
         void GridSearch2(const __constant gs_type4* pSsAmplitude,
                          const __constant char* lMask,
                          const __global gs_type4* fBasis,
                          __global float* chi2,
                          const int netChSeg,
                          const int nPts)
{

  unsigned int iPts = get_global_id(0);
  unsigned int lPts = get_local_id(0);
  unsigned int iSegm = get_global_id(1);

  if (iPts >= nPts) return;
  if (iSegm >= NSEGS) return;

  float4 this_iSegm_chi2 = 0.0f;

  __local float local_chi2[3][36];

  local_chi2[lPts][iSegm] = 0;

  if(lMask[iSegm] != '0') {

  __constant const gs_type4* realTrace = pSsAmplitude + iSegm*USAMP4;
  __global const gs_type4* baseTrace = fBasis + ((netChSeg*MAXIPTS + iPts)*
                                                 NSEGS + iSegm)*USAMP4;

     for(int nn = 0; nn < USAMP4; nn++) {
#ifdef GS_FLOAT
          this_iSegm_chi2 += native_powr(fabs((float4)(realTrace[nn] -
                                        baseTrace[nn])), (float4)METRIC);
#else
          this_iSegm_chi2 +=
               native_powr(convert_float4(abs_diff(realTrace[nn],
                                        baseTrace[nn])), (float4)METRIC);
#endif
     }
  }

  local_chi2[lPts][iSegm] = this_iSegm_chi2.s0 + this_iSegm_chi2.s1 +
                            this_iSegm_chi2.s2 + this_iSegm_chi2.s3;

  barrier(CLK_LOCAL_MEM_FENCE);

  if (iSegm == 0) {
    local_chi2[lPts][0] += (local_chi2[lPts][32] + local_chi2[lPts][33] +
                            local_chi2[lPts][34] + local_chi2[lPts][35]);
  }
  if (iSegm < 16) { local_chi2[lPts][iSegm] += local_chi2[lPts][iSegm+16]; }
    barrier(CLK_LOCAL_MEM_FENCE);
  if (iSegm < 8) { local_chi2[lPts][iSegm] += local_chi2[lPts][iSegm + 8]; }
    barrier(CLK_LOCAL_MEM_FENCE);
  if (iSegm < 4) { local_chi2[lPts][iSegm] += local_chi2[lPts][iSegm + 4]; }
    barrier(CLK_LOCAL_MEM_FENCE);
  if (iSegm < 2) { local_chi2[lPts][iSegm] += local_chi2[lPts][iSegm + 2]; }
    barrier(CLK_LOCAL_MEM_FENCE);
  if (iSegm == 0) {
    local_chi2[lPts][iSegm] += local_chi2[lPts][iSegm + 1];
    chi2[iPts] = local_chi2[lPts][0];
  }

}
```

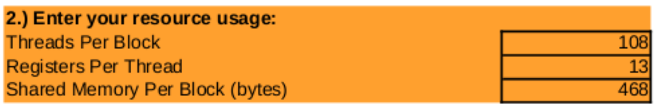
Also eight components vectors were tried, but the JIT compiler crashed giving an undocumented error message. This problem was reported as a bug.

Another optimization introduced is the read of basis and acquired data as four components vectors instead of two; the reason why it should increase performance is not reported in any manual, but, since the NVIDIA chip is a scalar processor, it has to be an improvement related to data fetching and not to the processing parallelization.

The Visual Profiler output produced running the code in listing 6 on the preceding page is reported here:

```
Kernel details : Grid size: 360 x 1, Block size: 3 x 36 x 1
Register Ratio = 1  ( 16384 / 16384 ) [13 registers per thread]
Shared Memory Ratio = 0.25 ( 4096 / 16384 ) [468 bytes per Block]
Active Blocks per SM = 8 : 8
Active threads per SM = 864 : 1024
Occupancy = 1  ( 32 / 32 )
Achieved occupancy  = 1  (on 30 SMs)
Occupancy limiting factor = None
```

The new input to the Occupancy Calculator is reported in figure 39 and the new graphs in figure 40 on the facing page.



2.) Enter your resource usage:

Threads Per Block	108
Registers Per Thread	13
Shared Memory Per Block (bytes)	468

Figure 39: NVIDIA Occupancy Calculator input cells for the code in listing 6 on the preceding page

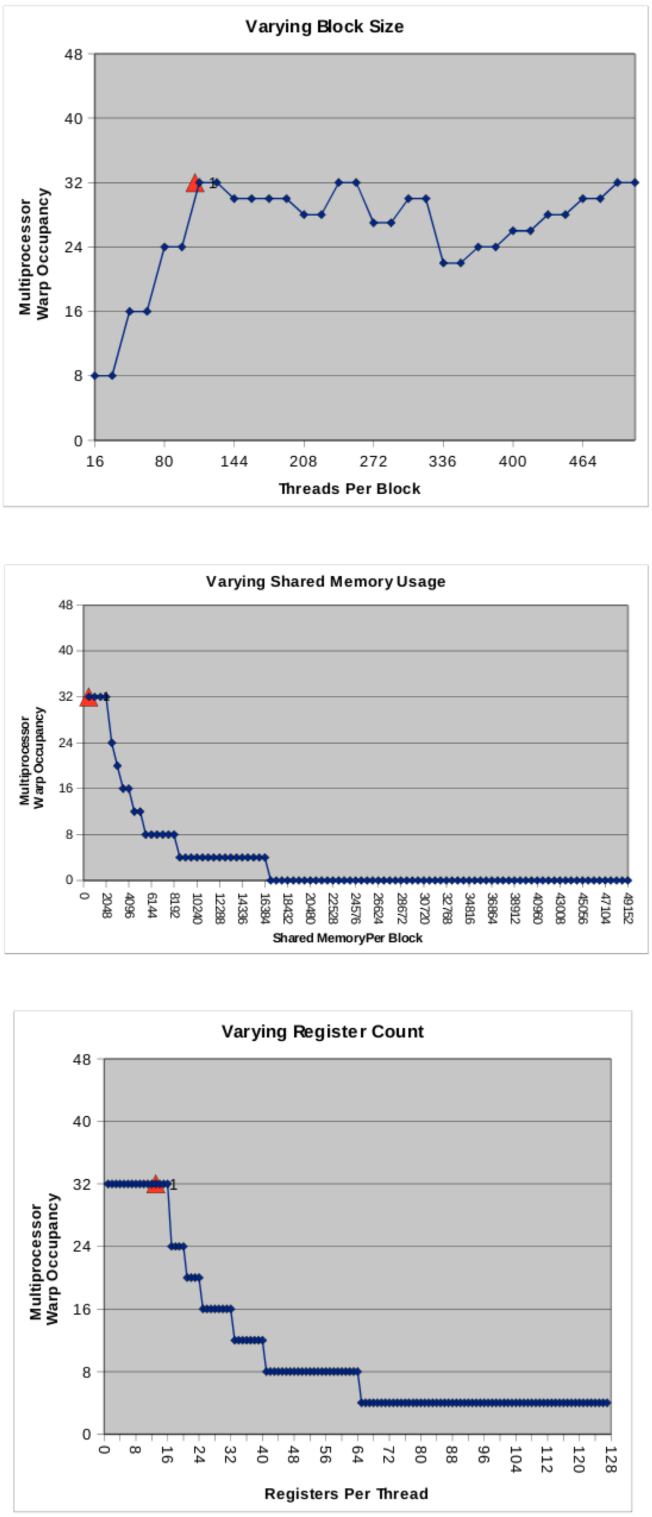


Figure 40: Plots produced by the NVIDIA Occupancy Calculator with the input displayed in figure 39 on the preceding page relative to the code in listing 6 on page 69.

5 CONCLUSION

Table 3: Full GridSearch implementations comparison. All the runs for the CPU were performed on an Intel® Core™ 2 Quad CPU at 2.40GHz, while for the GPU on the ASUS® ENGTX285 graphics card. Values are averages over various runs, expressed as events per second. All the runs use a $2 \times 2 \times 2$mm 3D grid basis database.

	CPU	**GPU**			
	Original version in listing 2	Mono-dimensional in listing 4	Bi-dimensional in listing 5	Occupancy optimized in listing 6	Speed increase between CPU and best GPU versions
`short`	275 ev/s	650 ev/s	1250 ev/s	1650 ev/s	$\times$ 6.00
`float`	65 ev/s	500 ev/s	1100 ev/s	1350 ev/s	$\times$ 20.77

Table 3 on the previous page presents the performance measurements comparison for the main developing steps presented in chapter 4 on page 47.

In this chapter the obtained results will be analyzed and plans for further optimization will be introduced, together with hints about the possible future perspective on upcoming hardware solutions.

5.1 BENCHMARKS

In the results presented in table 3 on the previous page all the values (except the final ratio) are expressed as processed events per second and all the code versions were modified in order to use `gs_type` data type to measure performances using both integer and floating-point input data precision.

The tests were executed on a workstation (since the chosen graphics card, due to its form factor, does not fit inside the NARVAL's computing nodes) equipped with an Intel® Core™ 2 Quad CPU at 2.40GHz and one ASUS® ENGTX285 graphics card. All the runs were executed using the NARVAL Emulator, reading real experimental data from the AGATA network storage and using a $2 \times 2 \times 2$mm 3D grid basis database.

Other more expensive GPU's form factors suited to fit 1U severs are available with external enclosures, connected through PCI Express links, embedding various graphics cards each.

Due to the lack of multiple PCI Express slots on the used workstation, the ENGTX285 card was the one and only graphics card in the system; consequently it was used shared to drive the monitor display, using it instead, only for GPGPU computations, would give slightly better results.

5.2 CONSIDERATIONS ABOUT RESULTS

From the results of the previously introduced benchmark, reported in table 3 on the preceding page, one can note the good speed improvement using `short` integer input data, but there is a much more interesting performance increase using `float` input data.

The obtained results will allow to run the PSA GridSearch algorithm operating on floating point data and therefore exploiting all the available information, obtaining a better position resolution.

The processing speed achieved will permit to run the Full GridSearch algorithm for on-line analysis reaching over 1kHz of events processed per computer and probably much more using multiple GPU cards or faster devices as explained in section 5.4 on page 77.

Moreover the use of the NARVAL Emulator surely worsen the results, since executing every actor in a serial sequence, there is idle GPU time while actors different from the PSA one are executing. Consequently using the real NARVAL environment, the performance will certainly increase, although it is hard to predict reliably by how much.

This phenomenon is easily observable in figure 41, where idle GPU time can be noticed between blocks of events that are the NARVAL buffers.

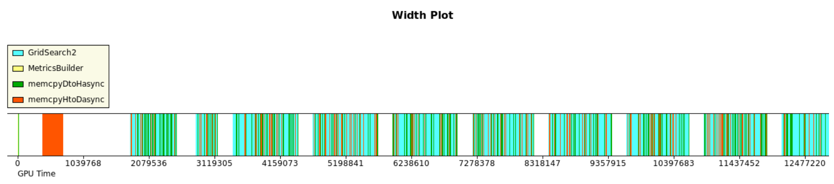


Figure 41: This is another graph produced by the Visual Profiler running the code in listing 6 on page 69. The elapsing GPU time is plotted from left to right reporting the operations executed. Idle time can be seen between blocks of events. At the beginning the large memory copy operation is the load of the basis database to the GPU Gloabl Memory.

Anyhow, apart from the proposed code implementation, this work opens also a wide range of possibilities to exploit the computing power of GPU devices for the AGATA Pulse Shape Analysis. Other algorithms, considered too time consuming using CPUs, may be developed using OpenCL and may provide good results also for multiple interactions.

5.3 FUTURE CODE DEVELOPMENT

The proposed GridSearch for GPUs implementation can be further improved in the future, since a strong limitation on its development has been the need of modifying as less as possible the GridSearch library structure to avoid modifications that may clash with the mainstream GridSearch development.

This limitation constrained the processing to one event at a time like for the CPU implementation. This is a source of inefficiency, in the GPU version, since the instruction, loading of arguments and lunching of a kernel, cost in terms of time and would be much more efficient the lunch of only one kernel processing more events. A whole NARVAL buffer of acquired data could be loaded in the GPU memory, in a single memory transfer, optimizing the host to device achieved bandwidth. Moreover, processing a lot of events in the same kernel call, could be also useful to dramatically increase the number of threads granting the code to scale more easily on newer hardware.

Another great limitation on the GridSearch for GPU performances derived from the fact that, as mentioned in section 4.2 on page 54, Memory Coalescing can be achieved only for memory locations with consecutive addresses.

To compute the FoM in formula 4.1 on page 49 the comparison between acquired and basis signals is not performed on all the segments as explained in section 4.1 on page 47, but is performed on segments that are near to the net-charge one. These segments are close to each other in a 3D fashion, but there is no way to collect segments data[1] in order to have all the needed samples loaded in consecutive memory addresses as would be accessed looking for the neighbouring of every possible net-charge segment.

Actually in the provided implementation, coalesced global memory loads happens for all the data belonging to the same segments, or to segments that are near in the basis array and in the detector as well, but it would be more efficient (and all the bandwidth could be exploited) if all the accessed data could be consecutive.

This is not elegant, since redundant data has to be stored in the GPU's Global Memory

A possible solution, although not very elegant, would be to have enough memory on the GPU to load different basis arrays, one for every possible net-charge segment (consequently one for every segment), containing all the samples that would be used for the comparison in the right order they are accessed.

Furthermore, all the events coming from a NARVAL buffer could be sorted in order to process together all the events with a net charge in the same segment and therefore accessing for every one of them the same basis segments' samples. This would decrease the global memory accesses and would let the caching system be exploited also for basis data.

1 Basis data in particular, since the acquired one is cached and does not suffer about uncoalesced global memory loads.

5.4 HARDWARE IMPROVEMENT

The cutting edge architecture of NVIDIA is not anymore represented by the GT200b chip series and moreover a new Compute Capability major version has been released, the 2.0.

Actually the latest available technology produced by NVIDIA is the Fermi architecture, implemented in the codenamed GT300 GPU chip series providing 512 SPs (32 SPs for each SM), the capability to execute concurrently multiple kernels, ECC memory, L2 caching, optimized double precision floating-point execution, unified 64-bit memory addressing, etc. [38]. NVIDIA claimed this card to reach over a TFLOP/s in single precision and over 500 GFLOP/s in double precision, confirming that hardware research and improvements are quickly developing in this field.

Also AMD (owner of the ATI brand) is releasing GPU chips whith GPGPU in mind like the AMD FireStream series reaching 1.2 TFLOP/s in single precision, and 240 GFLOP/s in double precision. In addition AMD is providing an OpenCL implementation capable of exploiting also CPU's cores in order to execute kernels on AMD processors to easily handle threads.

The idea to produce hybrid processors including inside CPU some characteristics of GPU's architectures is gaining ground and with good probability OpenCL will be the standard to program these new kind of processors.

The most famous examples of these hybrid processors are probably the Intel Larrabee (originally expected for the first month of 2010, but Intel postponed the release to the end of this year), and the Cell (*Cell Broadband Engine Architecture*) processor produced by IBM, Sony and Toshiba corporations.

The Cell processor (architected initially for the PlayStation3 gaming console), takes a radical departure from conventional multiprocessor or multi-core architectures. Instead of using identical cooperating cores, it uses a conventional high performance PowerPC main core that controls eight simple SIMD cores, called SPEs (*Synergistic Processing Elements*), where each SPE contains an SPU (*Synergistic Processing Unit*), a local memory, and a memory I/O controller. Moreover it implements vector processing [51], that could be already exploited by the proposed implementation. IBM already provided in late 2009 an OpenCL implementation for POWER6 and Cell processors.

A future development path for the GridSearch algorithm could be the comparison of the performance obtainable with different processors slightly modifying the provided OpenCL implementation optimizing it for each of them, and identifying the best matching architecture to the GridSearch problem.

Once identified, the development could be “stabilized” and the NARVAL nodes dedicated to PSA could be equipped with the chosen processors.

A CODE FRAGMENTS

Some code fragments have been collected in this appendix to better understand how the Grid Search Algorithm has been implemented using OpenCL.

The actor libraries directory tree is reported to have an overview of the code organization, focusing on the `filters/PSA` directory contents:

```
|-- common
|   |--
|
|-- consumers
|   |--
|
|-- filters
|   |-- PSA
|   |   |-- PSAFilter.cpp
|   |   |-- PSAFilter.h
|   |   |-- PSAFilterGridSearch.cpp
|   |   |-- PSAFilterGridSearch.h
|   |   |-- includePSA
|   |   |   |-- GridSearchClasses.h
|   |   |   |-- GridSearchParams.h
|   |   |   |-- SignalBasis.cpp
|   |   |   ‘-- SignalBasis.h
|   |   |-- libPSAFilter.cpp
|   |   ‘-- openCL
|   |       ‘-- gridsearch.cl
|   |
|   |--
|
|-- myADF0.2
|   |--
|
|-- producers
|   |--
```

A.1 ORIGINAL IMPLEMENTATION FOR CPUS

A.1.1 libPSAFilter

This is the library called by the NARVAL distributed data acquisition system; the actual implementation of the functions is loaded from the selected class, that in this case is the `PSAFilterGridSearch` one.

This is common to both the implementations, CPU and GPU one.

Listing 7: *libPSAFilter.cpp*

```
#include "PSAFilter.h"
#include "PSAFilterGridSearch.h"

extern "C" {
  PSAFilter *process_register(UInt_t *error_code)
  {
    if( (PSAFilter::gActualClass.size() == 0) ||
                                 (PSAFilter::gActualClass == "basic") )
                                      {
      std::cout << "\nPSAFilter::gActualClass == \"PSAFilter base
           class\"\n";
      return new PSAFilter();
    }
    else if(PSAFilter::gActualClass == "PSAFilterGridSearch") {
      std::cout << "\nPSAFilter::gActualClass == \"PsaFilterGridSearch\"\n";
      return new PSAFilterGridSearch();
    }
    else {
      std::cout << "\nERROR : PSAFilter::gActualClass " <<
                              PSAFilter::gActualClass << " not
                                   recognised\n";
      return NULL;
    }
  }
  void process_config(const char *directory_path, UInt_t *error_code)
  {
    PSAFilter::process_config (directory_path, error_code);
  }
  void process_block( PSAFilter *algo_data,
                      void *input_buffer,
                      UInt_t size_of_input_buffer,
                      void *output_buffer,
                      UInt_t size_of_output_buffer,
                      UInt_t *used_size_of_output_buffer,
                      UInt_t *error_code)
  {
    algo_data->process_block(
                      input_buffer,  size_of_input_buffer,
                      output_buffer, size_of_output_buffer,
                      used_size_of_output_buffer, error_code);
  }
  void process_initialise(PSAFilter *algo_data, UInt_t *error_code)
  {
    algo_data->process_initialise (error_code);
  }
  void process_reset(PSAFilter *algo_data,UInt_t *error_code)
  {
    algo_data->process_reset(error_code);
  }
  void process_start(PSAFilter *algo_data, UInt_t *error_code)
  {
    algo_data->process_start(error_code);
  }
  void process_stop(PSAFilter *algo_data,UInt_t *error_code)
  {
    algo_data->process_stop(error_code);
  }
  void process_pause(PSAFilter *algo_data,UInt_t *error_code)
  {
    algo_data->process_pause (error_code);
  }
  void process_resume(PSAFilter *algo_data, UInt_t *error_code)
  {
    algo_data->process_resume (error_code);
  }
};
```

A.1.2 PSAFilterGridSearch

This is the interface of the `PSAFilterGridSearch` class that is not entirely reported due to its dimension. This is the original version of the CPU implementation.

Listing 8: *PSAFilterGridSearch.h*: interface of the PSAFilterGridSearch library (include file)

```
//! Implementation of a simple grid search method.
/**
  Coded by Roberto Venturelli
  Ported first to Narval by Joa Ljungvall
  Modified and maintained by Dino Bazzacco
*/

#ifndef ADF_PSAFILTERGRID_SEARCH_H
#define ADF_PSAFILTERGRID_SEARCH_H

#include "NarvalInterface.h"
#include "AgataFrameFactory.h"
#include "AgataKeyFactory.h"
#include "Trigger.h"

#include "commonDefs.h"

#include "PSAFilter.h"
#include "SignalBasis.h"

#ifdef _FromGRU_
//#include <TAttFill.h>
  #include <TROOT.h>
  #include <TApplication.h>
  #include <TH1.h>
  #include <GAcq.h>
  #include <TCanvas.h>
  #include <TGraph.h>
  #include <TMultiGraph.h>
  #include <TH3.h>
  #include <TFile.h>
  #include <GSpectra.h>
  #include <TSystem.h>
#endif  // _FromGRU_

const int WCHAN =  42;  // number of channels to write in the output waves
const int WSAMP =  60;  // number of samples per channel in the output waves

class PSAFilterGridSearch : public PSAFilter
{
private:
  Float_t   fHitSegThreshold;
  bool      bDoSpec;
  bool      bDoMats;

  SignalBasis fBasis;
  float       fMetrics[NMETRIC];
  char        hmask[NCHAN][NCHAN+1];

#ifdef LOCALSPECTRA
  nDhist<unsigned int> *specEner;
  nDhist<unsigned int> *specTzero;
  nDhist<unsigned int> *specSigma;
  nDhist<unsigned int> *matrXYZR;
  nDhist<unsigned int> *matrSeg;
#endif  //LOCALSPECTRA

  std::string fnPsaTraces;
  FILE       *fpPsaTraces;

#ifdef _FromGRU_
  GNetServerRoot *PSANetworkRoot;
  GSpectra *PSASpectraDB;
  TH1I *PSACCE_raw,*PSACCE1_raw, *PSACCE, *PSACCE1;
  TH1I *PSAseg;
  TH1F *PSAxy_xproje, *PSAxz_xproje, *PSAyz_xproje;
  TH2F *PSAxy_Proje, *PSAxz_Proje, *PSAyz_Proje;
  TH3F *PSAxyz;
#endif  // _FromGRU_

  void MakeSegmentMap(int neighbours);
  void PrepareEvent  (PsaData *pD, pointExp *pS);
  void SetToSegCenter(PsaData *pD, pointExp *pS);
  int  SearchFullGrid(pointExp *pS, int netChSeg, char *lMask,
                      int addr_first, int addr_last);
  int  SearchAdaptive(pointExp *pS, int netChSeg, char *lMask,
                      int addr_first, int addr_last);
protected:
  // this is written in a thread-safe mode and can be called in parallel,
  // using different data slots
  Int_t Process(int slot = 0);
  // this is not thread-safe and must be called sequentially
  Int_t PostProcess(int slot = 0);

  Int_t AlgoSpecificInitialise();
  Int_t AlgoSpecificReset() { return 0; }

public:
  PSAFilterGridSearch();
  virtual ~PSAFilterGridSearch();

  void    SetHitSegThreshold(Float_t thres) {fHitSegThreshold = thres;}
  Float_t GetHitSegThreshold() {return fHitSegThreshold;}
  int  WriteTraces(int slot);

private:
  // this is needed to write traces in a thread-safe mode

  // locally saved experimental trace
  gs_type slot_fAmplitude[TCOUNT*TMODULO][NCHAN*WSAMP];

  // locally saved "fitted" trace
  gs_type slot_rAmplitude[TCOUNT*TMODULO][NCHAN*WSAMP];
};

#endif // ADF_PSAFILTERGRID_SEARCH_H
```

A.2 IMPLEMENTATION FOR GPUS

A.2.1 PSAFilterGridSearch

This is the interface of the `PSAFilterGridSearch` class that is not entirely reported due to its dimension. This is the modified version of the GPU implementation.

It can be noticed that the modifications are quite conservative and that the use of the GPU to accomplish the calculation can be selected at compile time using the `#define USEGPU` switch.

Listing 9: *PSAFilterGridSearch.h*: interface of the PSAFilterGridSearch library (include file)

```
//! Implementation of a simple grid search method.
/**
  Coded by Roberto Venturelli
  Ported first to Narval by Joa Ljungvall
  Modified and maintained by Dino Bazzacco
  GPU Implementation added by Enrico Calore
*/

#ifndef ADF_PSAFILTERGRID_SEARCH_H
#define ADF_PSAFILTERGRID_SEARCH_H

#include "NarvalInterface.h"
#include "AgataFrameFactory.h"
#include "AgataKeyFactory.h"
#include "Trigger.h"

#include "commonDefs.h"

#include "PSAFilter.h"
#include "SignalBasis.h"

#ifdef _FromGRU_
//#include <TAttFill.h>
  #include <TROOT.h>
  #include <TApplication.h>
  #include <TH1.h>
  #include <GAcq.h>
  #include <TCanvas.h>
  #include <TGraph.h>
  #include <TMultiGraph.h>
  #include <TH3.h>
  #include <TFile.h>
  #include <GSpectra.h>
  #include <TSystem.h>
#endif  // _FromGRU_

#ifdef USEGPU
  #include <CL/cl.h>
#endif // USEGPU

const int WCHAN =  42;  // number of channels to write in the output waves
const int WSAMP =  60;  // number of samples per channel in the output waves

class PSAFilterGridSearch : public PSAFilter
```

```
{
private:
  Float_t   fHitSegThreshold;
  bool      bDoSpec;
  bool      bDoMats;

  SignalBasis fBasis;
  float       fMetrics[NMETRIC];
  char        hmask[NCHAN][NCHAN+1];

#ifdef LOCALSPECTRA
  nDhist<unsigned int> *specEner;
  nDhist<unsigned int> *specTzero;
  nDhist<unsigned int> *specSigma;
  nDhist<unsigned int> *matrXYZR;
  nDhist<unsigned int> *matrSeg;
#endif  //LOCALSPECTRA

  std::string fnPsaTraces;
  FILE       *fpPsaTraces;

#ifdef _FromGRU_
  GNetServerRoot *PSANetworkRoot;
  GSpectra *PSASpectraDB;
  TH1I *PSACCE_raw,*PSACCE1_raw, *PSACCE, *PSACCE1;
  TH1I *PSAseg;
  TH1F *PSAxy_xproje, *PSAxz_xproje, *PSAyz_xproje;
  TH2F *PSAxy_Proje, *PSAxz_Proje, *PSAyz_Proje;
  TH3F *PSAxyz;
#endif  // _FromGRU_

#ifdef USEGPU

  cl_context GPUContext;
  cl_command_queue GPUCommandQueue;
  cl_device_id* GPUDevices;
  cl_kernel OpenCLMetrics;
  cl_kernel OpenCLGridSearch;
  cl_kernel OpenCLGridSearch2;
  cl_kernel OpenCLGridSearch3;
  int cl_err;

  cl_mem fMetricsGPU;
  cl_mem localMaskGPU;
  cl_mem pSsAmplitudeGPU;
  cl_mem numPtsGPU;
  cl_mem fBasisGPU;
  cl_mem chi2GPU;

  char* load_program_source(const char *filename);
  int device_stats(cl_device_id device_id);

#endif // USEGPU

  void MakeSegmentMap(int neighbours);
  void PrepareEvent  (PsaData *pD, pointExp *pS);
  void SetToSegCenter(PsaData *pD, pointExp *pS);
  int  SearchFullGrid(pointExp *pS, int netChSeg, char *lMask,
                      int addr_first, int addr_last);
  int  SearchAdaptive(pointExp *pS, int netChSeg, char *lMask,
                      int addr_first, int addr_last);
protected:
  // this is written in a thread-safe mode and can be called in parallel,
  // using different data slots
  Int_t Process(int slot = 0);
  // this is not thread-safe and must be called sequentially
  Int_t PostProcess(int slot = 0);

  Int_t AlgoSpecificInitialise();
  Int_t AlgoSpecificReset() { return 0; }

public:
  PSAFilterGridSearch();
  virtual ~PSAFilterGridSearch();

  void    SetHitSegThreshold(Float_t thres) {fHitSegThreshold = thres;}
  Float_t GetHitSegThreshold() {return fHitSegThreshold;}
  int  WriteTraces(int slot);

private:
  // this is needed to write the traces in a thread-safe mode
  // locally saved experimental trace
  gs_type slot_fAmplitude[TCOUNT*TMODULO][NCHAN*WSAMP];
  // locally saved "fitted" trace
  gs_type slot_rAmplitude[TCOUNT*TMODULO][NCHAN*WSAMP];
};

#endif // ADF_PSAFILTERGRID_SEARCH_H
```

Here are reported the code fragments common to all the kernels discussed previously. They are needed to initialize the GPU device (listing 10), to instantiate memory buffer inside the Global memory and to load in it the basis database 11 on the next page.

As mentioned before, this code is executed only once by the NARVAL data acquisition system.

Listing 10: *PSAFilterGridSearch.cpp*: initialization of the GPU device.

```
#ifdef USEGPU

  cl_platform_id platforms[3];
  unsigned int numPlatforms;
  cl_err = clGetPlatformIDs(3, platforms, &numPlatforms);
  if (cl_err != CL_SUCCESS) cout << "CL_ERROR (clGetPlatformIDs): " <<
                                                      cl_err << endl;

  cl_context_properties properties[] = { CL_CONTEXT_PLATFORM,
                                       (intptr_t)platforms[0], 0 };

  // Create a context to run OpenCL
  GPUContext = clCreateContextFromType(properties, CL_DEVICE_TYPE_GPU, NULL,
                                     NULL, &cl_err);

  if (cl_err != CL_SUCCESS) cout << "CL_ERROR (clCreateContextFromType): "
                                             << cl_err << endl;

  // Get the list of GPU devices associated with this context
  size_t ParmDataBytes;
  clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, 0, NULL,
                   &ParmDataBytes);
  GPUDevices = (cl_device_id*)malloc(ParmDataBytes);
  clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, ParmDataBytes,
                   GPUDevices, NULL);
  // Create a command-queue on the first GPU device
  GPUCommandQueue = clCreateCommandQueue(GPUContext, GPUDevices[0],
                                         0, NULL);

#endif // USEGPU
```

Listing 11: *PSAFilterGridSearch.cpp* building of the kernel source with the JIT compiler, GPU memory buffer allocation and load of the basis data in the GPU memory.

```
#ifdef USEGPU
  const char* filename ="PSA/openCL/gridsearch.cl";
  char *OpenCLSource = load_program_source(filename);

  // Create OpenCL program with source code
  cl_program OpenCLProgram = clCreateProgramWithSource(GPUContext, 1,
                                          (const char**)&OpenCLSource,
                                          NULL, NULL);

  // Build the program (OpenCL JIT compilation)
  const char* cl_build_options = "-cl-no-signed-zeros -cl-finite-math-only
                                    -cl-mad-enable";
  cl_err = clBuildProgram(OpenCLProgram, 0, NULL, cl_build_options, NULL,
                                                                     NULL);
  if (cl_err != CL_SUCCESS) {
     // Print some logs about the building process...
     char logbuffer[2048];
     clGetProgramBuildInfo(OpenCLProgram, *GPUDevices, CL_PROGRAM_BUILD_LOG,
                                          sizeof(logbuffer), logbuffer,
                                               NULL);
     cout << cout << "CL_ERROR (building CL source): " << cl_err << endl <<
                                          logbuffer << endl;
  }

  // Create handles to the compiled OpenCL functions (Kernels)
  OpenCLMetrics = clCreateKernel(OpenCLProgram, "MetricsBuilder", NULL);
  OpenCLGridSearch = clCreateKernel(OpenCLProgram, "GridSearch", NULL);
  OpenCLGridSearch2 = clCreateKernel(OpenCLProgram, "GridSearch2", NULL);
  OpenCLGridSearch3 = clCreateKernel(OpenCLProgram, "GridSearch3", NULL);

  cout << "Allocating GPU memory to store the basis... ";

  uint uSamples = USAMP;
  uint time_first = TSHIFT;

  localMaskGPU = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY, sizeof(char) *
                              NSEGS, NULL, NULL);
  sgridIndFineGPU = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY,
                                    sizeof(int) * MAXPTS, NULL, NULL);
  pSsAmplitudeGPU = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY,
                                    sizeof(gs_type) * NCHAN * NTIME,
                                    NULL, NULL);
  fBasisGPU = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY, sizeof(gs_type) *
                             NSEGS * uSamples * MAXPTS * NSEGS, NULL, NULL);
  numPtsGPU = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY,
                             sizeof(int) * NSEGS, NULL, NULL);
  chi2GPU = clCreateBuffer(GPUContext, CL_MEM_READ_WRITE, sizeof(float) *
                           MAXPTS, NULL, NULL);

  clFinish(GPUCommandQueue);

  cout << "Done." << endl << "Creating in the local RAM the basis buffer
```

```
                            to be copyed to GPU... ";

  size_t byte_offset = 0;

  fBasis_tmp = (char*)malloc(sizeof(gs_type) * NSEGS * uSamples * MAXPTS *
                                                                   NSEGS);

  for (int netChSeg = 0; netChSeg < NSEGS; netChSeg++) {
    int nPts = fBasis.numPts[netChSeg];
    for(int iPts  = 0; iPts < nPts; iPts++) {
      for(int iSegm=0; iSegm < NSEGS; iSegm++) {
         byte_offset = ((netChSeg*MAXPTS + iPts)*NSEGS + iSegm)*uSamples*
                                                          sizeof(gs_type);
         gs_type * baseTrace =
                   &fBasis.Pts[netChSeg][iPts].amplitude[iSegm][time_first];
         memcpy((void*)(fBasis_tmp + byte_offset), (void*)baseTrace,
                                         (size_t)(uSamples*sizeof(gs_type)));

      }  // end loop  over the segments

    }  // end loop over the base points iPts

  }

  cout << "Done." << endl << "Copying the Basis file to GPU... ";

  cl_err |= clEnqueueWriteBuffer(GPUCommandQueue, fBasisGPU, CL_TRUE, 0,
                     sizeof(gs_type) * NSEGS * uSamples * MAXPTS * NSEGS,
                     (void*)fBasis_tmp, 0, NULL,NULL);

  free((void*)fBasis_tmp);

  if (cl_err != CL_SUCCESS) cout << "CL_ERROR (EnqueueWriteBuffer writing
       Basis
                                          file to GPU): " << cl_err << endl;

  clFinish(GPUCommandQueue);

  cout << "Done." << endl;

#else //#ifdef USEGPU
```

Here is reported the `SearchFullGrid()` function used to launch the code in listing 6 on page 69. This function is called every time a new event has to processed.

`pS` is an object containing the acquired experimental data, `netChSeg` contains the number of the net-charge segment, `lMask` is an array giving information about which are the segments to be used for the FoM computation, while `samp_first` and `samp_last` give information about the samples interval for every segment to be used in the FoM calculation.

Listing 12: *PSAFilterGridSearch.cpp*: SearchFullGrid() function

```
int PSAFilterGridSearch::SearchFullGrid(pointExp *pS, int netChSeg,
                                        char *lMask, int samp_first,
                                        int samp_last)
{

float *metrics = fMetrics + NMETRIC2;

// number of samples to use (is the minimum of experimental and basis)
#ifdef USEGPU
int uSamples = USAMP;
#else
int uSamples = min(samp_last-samp_first+1, NTIME-TSHIFT);
#endif

int time_first = TSHIFT;  // start of basis
int bestPt     = 0;

float chi2min = pS->chi2min;
float chi2    =  0;

int nPts = fBasis.numPts[netChSeg];

#ifdef USEGPU
//Load the local Mask on the GPU buffer
cl_err = clEnqueueWriteBuffer(GPUCommandQueue, localMaskGPU, CL_TRUE, 0,
                              sizeof(char) * NSEGS, (void*)lMask,
                              0, NULL, NULL);

gs_type * realTrace_tmp = (gs_type*)malloc(NSEGS*uSamples*sizeof(gs_type));

for (int i = 0; i < NSEGS; i++) {
   memcpy((void*)&realTrace_tmp[i*uSamples],
          (void*)(&pS->sAmplitude[i][samp_first]),
          (size_t)(uSamples*sizeof(gs_type)));
}

cl_err |= clEnqueueWriteBuffer(GPUCommandQueue, pSsAmplitudeGPU,
                               CL_TRUE, 0, sizeof(gs_type) * uSamples *
                               NSEGS, (void*)realTrace_tmp, 0, NULL,
                               NULL);

free((void*)realTrace_tmp);

if (cl_err != CL_SUCCESS) cout << "CL_ERROR (EnqueueWriteBuffer): " <<
                                                          cl_err << endl;

cl_err |= clSetKernelArg(OpenCLGridSearch2, 0, sizeof(cl_mem),
                         (void*)&pSsAmplitudeGPU);
cl_err |= clSetKernelArg(OpenCLGridSearch2, 1, sizeof(cl_mem),
                         (void*)&localMaskGPU);
cl_err |= clSetKernelArg(OpenCLGridSearch2, 2, sizeof(cl_mem),
                         (void*)&fBasisGPU);
cl_err |= clSetKernelArg(OpenCLGridSearch2, 3, sizeof(cl_mem),
                         (void*)&chi2GPU);
cl_err |= clSetKernelArg(OpenCLGridSearch2, 3, sizeof(cl_mem),
                         (void*)&chi2GPUbis);
cl_err |= clSetKernelArg(OpenCLGridSearch2, 4, sizeof(float)*512, 0 );
cl_err |= clSetKernelArg(OpenCLGridSearch2, 5, sizeof(int),
                         (void*)&netChSeg);
cl_err |= clSetKernelArg(OpenCLGridSearch2, 6, sizeof(int), (void*)&nPts);

if (cl_err != CL_SUCCESS) cout << "CL_ERROR (SetKernel2Args): " <<
```

```
                                                          cl_err << endl;

clFinish(GPUCommandQueue);

unsigned int SMs = 30; // Number of GPU's SMs

// Launch the Kernel on the GPU (2 DIMENSIONs)

size_t GWsizeX = ((nPts%(SMs*3))==0) ? nPts : ((nPts - (nPts%(SMs*3)) +
                                                (SMs*3)));
size_t GWsizeY = (size_t)NSEGS;

size_t LWsizeX = 3;
size_t LWsizeY = (size_t)NSEGS;

const size_t GlobalWorkSize[] = { GWsizeX, GWsizeY };
const size_t LocalWorkSize[] = { LWsizeX, LWsizeY };

cl_err = clEnqueueNDRangeKernel(GPUCommandQueue, OpenCLGridSearch2, 2, NULL,
                                GlobalWorkSize2, LocalWorkSize2,
                                0, NULL, NULL);

if (cl_err != CL_SUCCESS) cout <<
    "CL_ERROR (EnqueueNDRangeKernel 'GridSearch2'): " << cl_err << endl;

// Wait the execution of the kernel to finish
cl_err = clFinish(GPUCommandQueue);

if (cl_err != CL_SUCCESS) cout <<
  "CL_ERROR (clFinish after EnqueueNDRangeKernel 'OpenCLGridSearch2'): " <<
                                                           cl_err << endl;

float chi2_pts_array[nPts];

clEnqueueReadBuffer(GPUCommandQueue, chi2GPU, CL_TRUE, 0,
                    sizeof(float) * nPts, chi2_pts_array, 0, NULL, NULL);

float chi2min_gpu = chi2min;

for (int iPts = 0; iPts < nPts; iPts++) {
   if (chi2_pts_array[iPts] < chi2min_gpu) {
      bestPt = iPts;
      chi2min_gpu = chi2_pts_array[iPts];
   }
}

    pS->bestdt  = 0;
    pS->bestPt  = bestPt;
    pS->chi2min = chi2min_gpu;

#else //USEGPU

[...]

#endif // USEGPU

return uSamples;

}
```

A.2.2 OpenCL Kernels

Listing 13: *gridsearch.cl: first rows common to all the kernels*

```
/*
* Implementation of a simple grid search method in OpenCL
* Based on the C++ version coded by Roberto Venturelli
*
* Created and maintained by Enrico Calore
*
*/

const float   METRIC    = 0.3f;   // norm for the figure of merit
const int     NMETRIC   = 131072; // 2^17
const int     NMETRIC2  = 65536;  // 2^16
const int     NSEGS     = 36;     // the 36 segments
const int     NCHAN     = 37;     // including the Core (after the segm.)
const int     USAMP     = 48;     // number of gs_type samples
const int     USAMP4    = 12;     // number of gs_type4 samples
```

BIBLIOGRAPHY

[1] IEEE standard for a versatile backplane bus: VMEbus, 1987. (Cited on page 20.)

[2] *GPFS V3.2.1 Concepts, Planning, and Installation Guide*, 2008. (Cited on page 24.)

[3] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2009. (Cited on page 58.)

[4] *The OpenCL Specification*, v.1.0 rev.48 edition, 2009. (Cited on page 37.)

[5] S. Badoer, L. Berti, M. Biasotto, E. Calore, S. Fantinel, M. Gulmini, G. Maron, P. Molini, and N. Toniolo. AGATA data storage system. *LNL Annual Report 2008*, INFN-LNL-226:46, 2009. (Cited on pages 21 and 24.)

[6] D. Bazzacco. The advanced gamma ray tracking array AGATA. *Nuclear Physics A*, 746:248c–254c, 2004. (Cited on pages 1 and 4.)

[7] D. Bazzacco, B. Cederwall, J. Cresswell, G. Duchene, J. Eberth, W. Gast, J. Gerl, W. Korten, I. Lazarus, R. M. Lieder, J. Simpson, and D. Weißhaar. Technical proposal for an advanced gamma tracking array. Technical report, European Gamma Spectroscopy Community, Sept. 2001. (Cited on page 1.)

[8] D. Bazzacco and T. Kröll. Simulation and analysis of pulse shapes from highly segmented hpge detectors for the γ-ray tracking array MARS. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, (463):227–249, 2001. (Cited on pages 15 and 47.)

[9] D. Bazzacco and T. Kröll. A genetic algorithm for the decomposition of multiple hit events in the gamma-ray tracking detector MARS. *Nucl.Instr.Meth. A*, 565:691, 2006.

[10] B. Bruyneel, P. Reiter, and G. Pascovici. Characterization of large volume hpge detectors. part ii: Experimental results. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 569:774–789, 2006. (Cited on page 47.)

[11] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus. *ACM Transactions on Graphics*, 23(3):777–786, 2004. (Cited on page 34.)

[12] E. Calore, E. Farnea, D. Mengoni, and N. Toniolo. Implementation of on-line analysis library in NARVAL: the PRISMA case. *LNL Annual Report 2008*, INFN-LNL-226:52, 2009.

[13] E. Calore and D. Mengoni. AGATA DAQ: a NARVAL prototype installation and test. *LNL Annual Report 2007*, INFN-LNL-222:197, 2008.

[14] F. Crespi, F. Camera, A. Bracco, B. Million, O. Wieland, V. Vandone, F. Recchia, A. Gadea, T. Kröll, D. Mengoni, E. Farnea, C. Ur, and D. Bazzacco. Application of the recursive subtraction pulse shape analysis algorithm to in-beam HPGe signals. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 604(3):604–611, 2009.

[15] F. Crespi, F. Camera, B. Million, M. Sassi, O. Wieland, and A. Bracco. A novel technique for the characterization of a hpge detector response based on pulse shape comparison. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 593(3):440–447, 2008. (Cited on page 15.)

[16] J. Cresswell and X. Grave. AGATA PSA and tracking algorithm integration. Technical report, University of Liverpool & IPN Orsay, 2006. (Cited on page 28.)

[17] E. Farnea, A. Gadea, G. de Angelis, et al. Coupling of the AGATA Demonstrator Array with the PRISMA Magnetic Spectrometer. *LNL Annual Report 2008*, INFN-LNL-226:40, 2009.

[18] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 100:21, 1972. (Cited on page 33.)

[19] A. Gadea, J. Nyberg, F. Recchia, et al. First in-beam Commissioning Experiment of AGATA. *LNL Annual Report 2008*, INFN-LNL-226:39, 2009.

[20] W. Gast, R. Lieder, L. Mihailescu, M. Rossewij, H. Brands, A. Georgiev, J. Stein, T. Kroll, and F. GmbH. Digital signal

processing and algorithms for γ-ray tracking. *IEEE Transactions on Nuclear Science*, 48(6):2380–2384, 2001. (Cited on page 19.)

[21] A. Georgiev and W. Gast. Digital pulse processing in high resolution, high throughput, gamma-ray spectroscopy. *IEEE Transactions on Nuclear Science*, 40(4):770–779, aug 1993. (Cited on page 19.)

[22] A. Görgen. The position sensitivity of the agata prototype crystal analyzed using a database of calculated pulse shape. Technical report, DAPNIA/SPhN, CEA Saclay, France, 2003. (Cited on page 50.)

[23] X. Grave, R. Canedo, J.-F. Clavelin, S. Du, and E. Legay. NARVAL a modular distributed data acquisition system with ada 95 and rtai. *IEEE-NPSS Real Time Conference*, 0:65, 2005. (Cited on page 24.)

[24] L. Hildingsson, C. Beausang, D. Fossan, W. P. Jr., A. Byrne, and G. Dracoulis. Transverse BGO compton suppression shield. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 252(1):91–94, 1986. (Cited on page 8.)

[25] Instrumentation and Measurement Technology Conference. *A Simple Method for the Characterization of HPGe Detectors*, 2005. (Cited on page 15.)

[26] H. Jung, H. Cho, J. Lee, and C. Lee. Improvement of the compton suppression ratio of a standard bgo suppressor system by a digital pulse shape analysis. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 580(2):1016–1019, 2007. Imaging 2006 - Proceedings of the 3rd International Conference on Imaging Techniques in Subatomic Physics, Astrophysics, Medicine, Biology and Industry. (Cited on page 9.)

[27] G. F. Knoll. *Radiation detection and measurement*. Wiley, New York, 3rd edition, 2000.

[28] T. Kröll and D. Bazzacco. A genetic algorithm for the decomposition of multiple hit events in the γ-ray tracking detector MARS. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 565(2):691–703, 2006.

[29] T. Kröll, D. Bazzacco, R. Venturelli, P. Pavan, and C. Ur. Pulse shape analysis by a genetic algorithm with the γ-ray tracking detector MARS. *LNL Annual Report 2004*, LNL-INFN-204:220–221, 2005.

[30] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro-Institute of Electrical and Electronics Engineers*, 28(2):39–55, 2008. (Cited on pages 43 and 44.)

[31] A. Lopez-Martens, K. Hauschild, A. Korichi, J. Roccaz, and J. Thibaud. γ-ray tracking algorithms: a comparison. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 533(3):454–466, 2004.

[32] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2), Mar. 2008. (Cited on pages 35 and 44.)

[33] NVIDIA. *NVIDIA OpenCL Best Practices Guide*. NVIDIA Corporation, Santa Clara CA, 1.0 edition, Aug. 2009.

[34] NVIDIA. *NVIDIA OpenCL JumpStart Guide*. NVIDIA Corporation, Santa Clara CA, 0.9 edition, Apr. 2009. (Cited on page 37.)

[35] NVIDIA. *OpenCL Programming Guide for the CUDA Architecture*. NVIDIA Corporation, 2.3 edition, Aug. 2009. (Cited on page 37.)

[36] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[37] L. Pandola, C. Cattadori, and N. Ferrari. Neural network pulse shape analysis for proportional counters events. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 522(3):521–528, 2004.

[38] D. Patterson. The top 10 innovations in the new NVIDIA Fermi architecture, and the top 3 next challenges, 2009. (Cited on page 77.)

[39] F. Recchia. *In-beam test and imaging capabilities of the AGATA prototype detector*. PhD thesis, Scuola di Dottorato di Ricerca

in Fisica Ciclo XX, Dipartimento di Fisica, Università degli studi di Padova, Jan. 2008.

[40] F. Recchia, D. Bazzacco, E. Farnea, A. Gadea, R. Venturelli, T. Beck, P. Bednarczyk, A. Buerger, A. Dewald, M. Dimmock, G. Duchêne, J. Eberth, T. Faul, J. Gerl, R. Gernhaeuser, K. Hauschild, A. Holler, P. Jones, W. Korten, T. Kröll, R. Krücken, N. Kurz, J. Ljungvall, S. Lunardi, P. Maierbeck, D. Mengoni, J. Nyberg, L. Nelson, G. Pascovici, P. Reiter, H. Schaffner, M. Schlarb, T. Steinhardt, O. Thelen, C. Ur, J. Valiente Dobon, and D. Weißhaar. Position resolution of the prototype AGATA triple-cluster detector from an in-beam experiment. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 604(3):555–562, 2009. (Cited on page 47.)

[41] M. C. Schlarb. *Simulation and Real-Time Analysis of Pulse Shapes from segmented HPGe-Detectors*. PhD thesis, Fakultät für Physik der Technischen Universität München Physik-Department E12, Nov. 2009. (Cited on pages 14, 15, and 47.)

[42] G. J. Schmid, D. Beckedahl, J. J. Blair, A. Friensehner, and J. E. Kammeraad. HPGe compton suppression using pulse shape analysis. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 422(1-3):368–372, 1999. (Cited on page 9.)

[43] J. Simpson, J. Nyberg, W. Korten, et al. AGATA technical design report. Technical report, AGATA Collaboration, Dec. 2008. (Cited on page 1.)

[44] G. Suliman and D. Bucurescu. Fuzzy clustering algorithm for gamma ray tracking in segmented detectors. *Romanian Reports in Physics*, 62(1):27–36, 2010.

[45] S. E. Thompson and S. Parthasarathy. Moore's law: the future of si microelectronics. *Materials Today*, 9(6):20–25, 2006. (Cited on page 31.)

[46] A. Triossi, B. Travers, C. Santos, G. Rampazzo, C. Oziol, P. Medina, D. Linget, I. Lazarus, R. Isocrate, X. Grave, R. Edelbruck, P. Coleman-Smith, D. Bortolato, L. Berti, and M. Bellato. Global trigger and readout system for the AGATA experiment. *IEEE Trans.Nucl.Sci.*, 55(1):91, feb 2008. (Cited on page 18.)

[47] R. Venturelli and D. Bazzacco. Adaptive grid search as pulse shape analysis algorithm for γ-tracking and results. *LNL Annual Report 2004*, LNL-INFN-204:220–221, 2005. (Cited on page 47.)

[48] N. Warr, J. Eberth, G. Pascovici, H. G. Thomas, and D. Weißhaar. MINIBALL: The first gamma-ray spectrometer using segmented, encapsulated germanium detectors for studies with radioactive beams. *The European Physical Journal A - Hadrons and Nuclei*, 20(1):65–66, Apr. 2003. (Cited on page 12.)

[49] O. Wieland, T. Kroll, D. Bazzacco, R. Venturelli, F. Camera, B. Million, E. Musso, B. Quintana, C. Ur, M. Bellato, R. Isocrate, C. Manea, R. Menegazzo, P. Pavan, C. Alvarez, E. Farnea, A. Gadea, D. Rosso, R. Spolaore, A. Pullia, G. Casati, A. Geraci, G. Ripamonti, and M. Descovich. Gamma-ray tracking with segmented hpge detectors. *Brazilian Journal of Physics*, 33(2):206–210, June 2003. (Cited on page 6.)

[50] A. Wiens, H. Hess, B. Birkenbach, B. Bruyneel, J. Eberth, D. Lersch, G. Pascovici, P. Reiter, and H.-G. Thomas. The AGATA triple cluster detector. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 618(1-3):223–233, 2010. (Cited on page 5.)

[51] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 9–20. ACM, 2006. (Cited on page 77.)

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity,
and especially because it produces objects of beauty.

— *Donald E. Knuth*

RINGRAZIAMENTI

I would like to thank the whole AGATA Community and in particular the DAQ working group, in which I had the possibility to work with wonderful people in the last years.

Ringrazio innanzi tutto la mia famiglia per avermi supportato e sopportato in questi anni universitari e quindi per avermi permesso di studiare fino ad arrivare a questo giorno.

Ringrazio il Prof. Carlo Ferrari in quanto Relatore per questo lavoro di tesi.

Ringrazio Dino Bazzacco e Francesco Recchia per avermi sopportato pazientemente durante la redazione dello stesso, per il tempo che hanno perso a correggere le mie contorte frasi inglesi, per avermi sempre dato preziosi consigli e per avere creduto che potessi veramente fare andare più veloce la PSA sulle schede video. Ringrazio anche Enrico Farnea per le sue correzioni e per la grande disponibilità.

Ringrazio Daniel Napoli per avermi sempre fatto da mentore all'interno dei Laboratori di Legnaro, fin dal mio primo stage.

Ringrazio inoltre tutte le altre persone con cui ho avuto il piacere di lavorare ai LNL ed in particolare Daniele Mengoni con cui ho fatto le prime installazioni di NARVAL e Roberto Venturelli per il suo GridSearch.

Ringrazio Francesco anche per avere contribuito, prima di me, a mantenere in vita le aule studio Acquario e Pollaio del Dip. di Fisica che per anni con i loro frequentatori hanno costituito per me una seconda casa ed una seconda famiglia; permettendomi inoltre di avvicinarmi all'affascinante mondo della Fisica e senza delle quali forse questo lavoro di tesi non esisterebbe.

Mi scuso invece con i futuri studenti che, forse anche per la mia colpa di non averlo difeso abbastanza strenuamente, potrebbero non trovare più in futuro un posto così incredibilmente perfetto per studiare, socializzare e condividere le proprie conoscenze, nella più reale synusia platonica, all'interno della nostra Università.

Colpevole dell'eventuale scomparsa di questo luogo, è molto più però quella parte dei vertici dell'Ateneo che per scarse vedute, pregiudizi o per la cattiva consuetudine di prendere decisioni su argomenti che non si conoscono, hanno fatto di tutto per ostacolarne la sopravvivenza.

Prima di infervorarmi e scadere nel linguaggio, torno ai ringraziamenti, rivolti a tutti quelli che invece, come me, nel Pollaio e nell'Acquario ci hanno creduto e li hanno vissuti; in ordine sparso: Paride, Diego, Anne, Beucco, il Marsigliese, Dima, LucaNardo, LucaGuidetti, Marina & Francé, Marketto, Coma, Annina, il Moro, Chosa, l'Anita, Jaleh, Fernando, il Doc, Massimiliano, Batta Mariachiara & figlio/a, Anna veterinaria, Alice veterinaria, Gian, Nick fisico, Nick chimico, Sergio, Pippaccio, Mario, Tonuzzo, la squadra di Pallavolo del Pollaio, la Casella, Spalla, Nevenka, i Manzini, le sorelle DeMarchi e l'enormità di altre persone che in questo momento mi sto dimenticando, ma che mi torneranno in mente dopo avere stampato. Ringrazio anche Giovanni per i pochi momenti passati insieme prima che ci lasciasse.

Ringrazio tutti quegli amici che mi sono sempre stati vicini sin dalle superiori: Yargo, Manza, i Cani, l'Anzela, i Vercellesi, la Dana e tutti gli altri.

Infine un ringraziamento speciale per la mia ragazza, Erica, per essermi sempre stata vicina, per avermi aspettato tutte le volte che sono arrivato in ritardo ed avermi quasi sempre perdonato, per sorbirsi un sacco di discorsi da nerd su tutti i miei invasi, ma anche per essersi fatta un po' contaminare, per essere venuta ad un HackMeeting, per usare Linux ed avere letto il Neuromante.

La ringrazio per avere sempre cercato di darmi una mano nei momenti difficili, per avere imparato a capirmi e per un milione di altre cose, piccoli e grandi. Grazie Erica.

Padova, anno accademico 2009/2010