# Reconstruction of secondary vertices with medical imaging methods and GPUs for the ATLAS experiment at LHC

Andreas Schulz

Masterarbeit in Physik
angefertigt im Physikalischen Institut

vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität
Bonn

November 2013

I hereby declare that this thesis was formulated by myself and that no sources or tools other than those cited were used.

Bonn, . . . . . .  . . . . . .
Date  Signature

1. Gutachter: Prof. Dr. Norbert Wermes
2. Gutachter: Prof. Dr. Klaus Desch

# Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Dr. Eckhard von Törne for the continuous support, for his patience, enthusiasm, and immense knowledge. I am very grateful to Prof. Wermes who gave me the opportunity to write this thesis.
I am thankful to my colleagues for the help and support. I would like to thank my physics teacher Günter Danne who gave me the motivation to study physics. Last but not least, I thank my parents for supporting me throughout all my studies.

# Contents


**1 Introduction** **1**

**2 Fundamentals of this thesis** **3**
2.1 Standard Model . . . 3
2.2 Large Hadron Collider . . . 4
2.2.1 Characterising quantities of an acclerator . . . 5
2.2.2 Major reaction types at the LHC . . . 5
2.2.3 Benefit of vertex reconstruction . . . 6
2.2.4 Beamspot of the LHC . . . 6
2.3 ATLAS detector at LHC . . . 7
2.3.1 Quantities that describe tracks in the ATLAS detector . . . 8
2.4 Properties of b-vertices . . . 9
2.5 Quantities for b-tagging evaluation . . . 10

**3 General-purpose computing on graphics processing units** **13**
3.1 Overview . . . 13
3.2 CUDA . . . 15
3.3 Hardware implementation of CUDA . . . 16
3.4 Compute capability . . . 18
3.5 Nvidia Tesla GPU . . . 19
3.6 PC specifications . . . 20

**4 Medical imaging vertex finder** **21**
4.1 Tomography . . . 21
4.2 Algorithm . . . 21
4.3 Backprojection . . . 22
4.4 Fourier transformation . . . 23
4.5 Filter application . . . 24
4.6 Window function . . . 25
4.7 Noise reduction . . . 28
4.8 Clustering . . . 28
4.9 Analysis of time-consumption in the standard MIVertexing algorithm . . . 28

**5 Implementation of parallel execution of MIVertexing algorithm** **31**
5.1 First steps . . . 31
5.2 Backprojection . . . 31

5.3 Fourier transformation . . . 32
5.4 Filter function, window function and noise reduction . . . 32
5.5 Clustering . . . 33

**6 Comparison of the time-consumption of the CPU and the GPU version of the MIVertexfinder 37**

**7 Preparations for b-tagging 39**
7.1 Size of the search volume . . . 39
7.2 Adjust the MIVertex finding algorithm . . . 40
7.3 Comparison b-tagger . . . 41

**8 Results 45**
8.1 b-tagging results . . . 45
8.2 Variation of $d_0$ cut . . . 46

**9 Summary and Outlook 49**
9.1 Outlook . . . 49

**A Useful information 51**
A.1 Measurement of the utilisation time . . . 51
A.2 Evaluation criteria . . . 51
A.3 Impact of using straight tracks instead of curved ones . . . 52
A.4 Z-Finder-Algorithm . . . 52
A.5 Precision of the GPU implementation . . . 53
A.6 Different properties of primary vertices and b-vertices . . . 54

**Bibliography 57**

**List of Figures 59**

**List of Tables 61**

CHAPTER 1

# Introduction

The medical imaging vertex finder was developed and implemented by Stephan Hageböck in his Diploma thesis [1]. The medical imaging vertex finder reconstructs primary vertices based on reconstruction and filtering techniques from medical imaging. This vertex finder is inspired by methods that are used in todays medicine to display organs of the human body. These medical imaging methods apply filter techniques to linear trajectories to gain a density distribution that discribes the investigated part of the human body.
These methods prove to be suitable to determine vertices in collisions in high energy physics. In the case of high energy physics the filters are applied to the tracks of charged particles to gain a vertex density distribution. On the basis of this vertex density distribution it is possible to determine the contained vertex positions.
In my thesis I pursue two aims. The first task is to adapt the medical imaging finder to reconstruct secondary vertices, especially b-vertices. b-quarks appear in many important reactions in high energy physics. A very famous example is the Higgs to $b\bar{b}$ decay. This decay is one of the search channels that are used to prove the existence of the Higgs particle. This is also one of the search channels in which the University of Bonn participates. The identification of this decay can only be achieved by the detection of the b-quarks. Therefore it is necessary to obtain efficient algorithms that are able to locate b-quarks and distinguish between other quarks.
However adapting the medical imaging vertex finder to reconstruct b-vertices requires additional computation power. This computation power can be provided by GPUs. A GPU is the computing unit on a graphics card. It is able to achieve a very high computation performance which exceeds the computation power of normal CPUs by far. This performance is made possible by the large number of cores a GPU contains. This leads to the second task in my thesis, the implementation of the medical imaging finder for GPUs. Due to the high number of cores, programing on GPUs differs a lot from the "normal" CPU programing. Therefore I have to become acquainted with the software environment CUDA that is able to access all the features of a GPU.

CHAPTER 2

# Fundamentals of this thesis

## 2.1 Standard Model

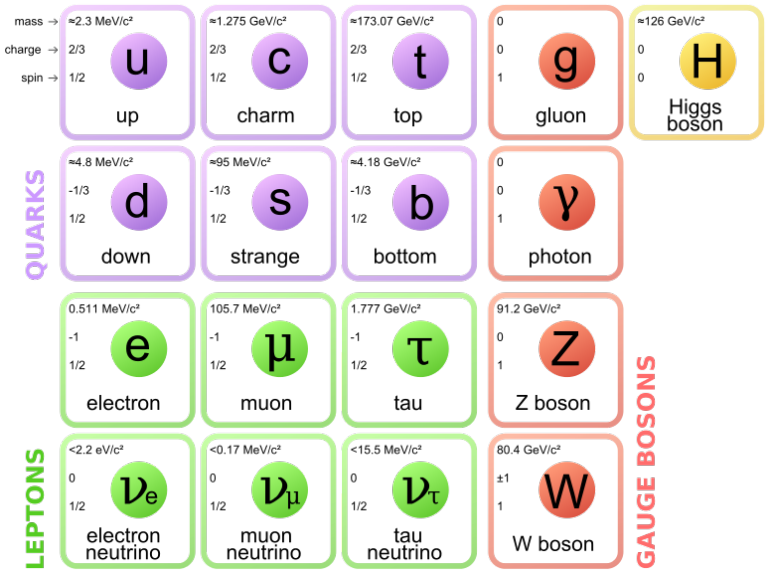



Figure 2.1: Particles of the Standard Model [2].

The Standard Model describes the structure of matter and the interactions of matter particles with each other. The particles of the Standard Model are illustrated in figure 2.1. Twelve of these particles have spin $1/2$ and are therefore fermions. Six fermions are called leptons and do not carry colour charge. Three of them are called electron, muon and tau. The remaining leptons are the very light neutrinos. They are called electron neutrino, muon neutrino and tau neutrino. Unlike the electron, muon and tau they are electrically neutral. But all leptons are influenced by the weak force.
There are six quarks (up, down, charm, strange, bottom, and top) which interact via the strong interaction. The strong interaction is described by a theory called Quantum chromodynamics (QCD). This theory defines three colour states (red, blue and green) which are carried by the quarks. In addition it defines three anti-colours that are carried by the corresponding antiquarks (anti-up, anti-down, anti-

charm, anti-strange, anti-bottom, and anti-top). According to QCD the strong interactions are mediated by gluons and occur only between coloured states. Gluons are spin one particles and therefore bosons. They carry their own colour charge that consists of a colour and an anti-colour.
Due to the colour confinement described by QCD, quarks appear only in uncharged bound states, called hadrons. There are different types of hadrons that satisfy this uncharged condition. Analogue to the fact that the superposition of the three primary colours leads to white, the superposition of the three colour states leads to an uncharged state. This bound state consisting of three quarks is called baryon. Since this superposition principle is still valid for the anti-colours, the combination of three antiquarks leads also to an uncharged state which is called anti-baryon. Apart from this possibility the superposition of colour and its anti-colour yields to an uncharged state is called meson. When trying to tear apart the quarks of a meson or a baryon the energy density between the quarks increases until a quark anti-quark pair is produced out of the vacuum. Due to this production the colour field is interrupted and two separate hadrons are formed. In most of the cases these hadrons are mesons because creating baryons requires the production of two quarks and two antiquarks. Because of the fact that tearing apart bound quarks creates at least one additional hadron, a scattered quark or gluon forms jets. These jets consist of several hadrons that fly approximately in the original direction of the scattered particle.
The gauge bosons are the force carriers of the Standard Model and have spin 1. The photon mediates the electromagnetic force between electrically charged particles. The weak interaction is mediated by the electrically charged $W^+$ and $W^-$ and the electrically neutral Z-Boson. Gluon and the photon are massless, whereas the $W^\pm$ and Z boson are massive.
The mass of the particles results from interactions with the Higgs field which are mediated by the Higgs boson. The Higgs boson was predicted a long time ago in 1964 and got tentatively confirmed this year. The Higgs boson is a spin zero particle and has probably a mass of 125 GeV. The tentative confirmation was triggered due to research results at the LHC.

## 2.2 Large Hadron Collider

The Large Hadron Collider is a proton-proton collider built by the European Organization for Nuclear Research (CERN). It is a ring collider with a circumference of 26.7 km. The LHC achieves the highest center-of-mass energy and the highest luminosity of all ever built colliders.
The protons are extracted from a hydrogen source and injected into an acceleration ring. After several acceleration steps (at small ring accelerators) the protons enter the LHC in bunches. In the LHC particles are further accelerated in two opposite directions in two seperate beam pipes. In every beam pipe a dipole magnet system which consists of superconducting magnets keeps the particles in their orbit. Because of the symmetric design of the LHC the center-of-mass of the reaction is at rest and the achievable center-of-mass energy is two times the energy of one beam $\sqrt{s} = 2E_{\text{beam}}$.
Before the shutdown this year the beam energy was 4 TeV resulting in a center-of-mass energy of 8 TeV. After the next commissioning scheduled for 2015 a center-of-mass energy of 14 TeV is planned.
The particles are brought to collision at four different points at the LHC. At each of these points there is located an experiment. They are the four main experiments at the LHC: ATLAS, CMS, LHCb, ALICE. ATLAS and CMS are the two experiments that are most involved in the Higgs search. In my thesis I will refer to the research ot ATLAS.

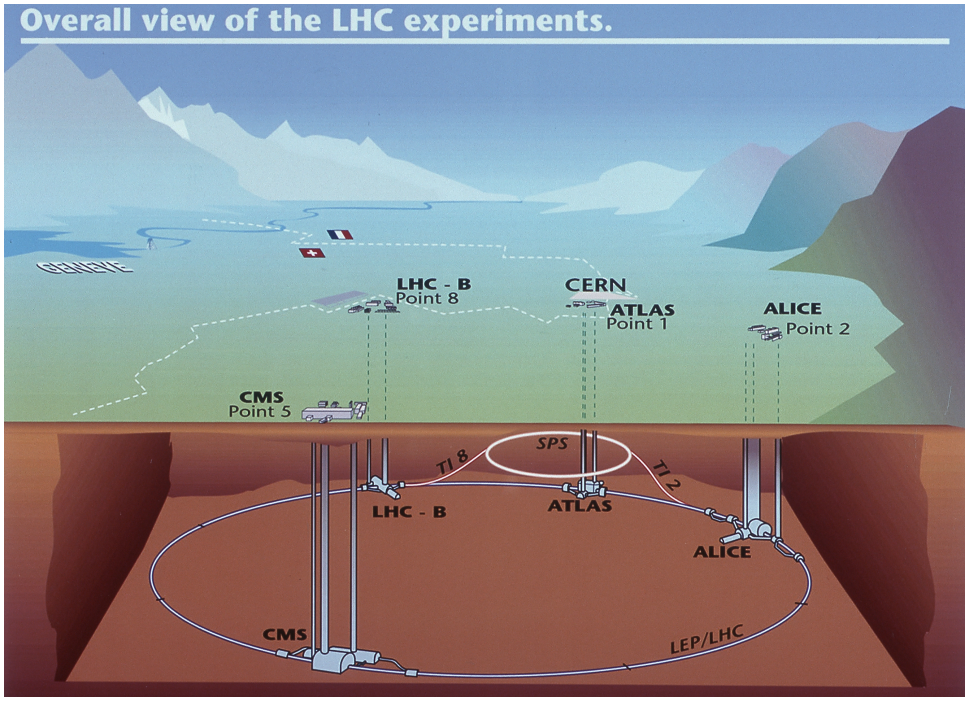



Figure 2.2: The Large Hadron Collider with its four main experiments [3].

### 2.2.1 Characterising quantities of an acclerator

A very important quantity that characterises an accelerator is the luminosity $L$. The luminosity describes the number of collisions that can be produced in a detector per $\mathrm{cm}^2$ and per second. It can be determined by:

$$L = \frac{n \cdot N_1 \cdot N_2 \cdot f}{A}, \quad [L] = \mathrm{cm}^{-2}\,\mathrm{s}^{-1}$$

In this equation $n$ is the number of bunches, $N_1$ and $N_2$ are the number of particles in the colliding bunches, $f$ is the orbital frequency of the bunches and $A$ is the cross sectional area of the beam.
The cross section $\sigma$ of a particular reaction is a measure for the probability that this reaction takes place. The unit of the cross section is $[\sigma] = \mathrm{cm}^2$. Since cross sections in high energy physics are very low, another unit *barn* is introduced.

$$1 \times 10^{-24}\,\mathrm{cm}^2 = 1\,\mathrm{b}$$

Multiplying the luminosity by the cross section of a specific reaction one gets the reaction rate $\dot{N}$ for this reaction. The reaction rate describes the number of reactions that happen per second, thus it has the unit $[\dot{N}] = \mathrm{s}^{-1}$.

$$\dot{N} = \sigma \cdot L$$

### 2.2.2 Major reaction types at the LHC

**Elastic scattering** In the process of elastic scattering the scattered protons stay intact, but they change their directions of propagation. This change is not very big so they progress further along the beam pipe. As a consequence, they cannot be detected by a detector like ATLAS.

**Inelastic diffractive scattering** In this case at least one of the participating protons does not stay

intact, but their quantum numbers do not change during the reaction. However the remnants progress further in forward direction and thus are not detected.

**Inelastic non-diffractive scattering** This reaction type is the most interesting, because the reaction products move along the beam pipe as well as transveral to the beam pipe. Hence these reactions can be detected very well. Within this reaction type one distinguishes between soft collisions and hard collisions. The soft collisions are also called minimum bias collisions. Particles that originate from such a minimum bias collision have only small transversal momenta. Because of this they are far less interesting compared to hard collisions. High transversal momenta indicate the production of heavy particles such as the Higgs boson.
In addition minimum bias collisions occur more often than hard collisions so it is very important to sort them out. Therefore events that consist only of minimum bias collisions are not recorded at the ATLAS detector.

### 2.2.3 Benefit of vertex reconstruction

A vertex is a point from which tracks of charged particles originate. Accordingly, at this point some kind of reaction happened. There are two sorts of vertices, primary vertices and secondary vertices. In case of the LHC primary vertices indicate the position of a proton-proton collision. Secondary vertices indicate the decay of a long-lived particle.
As already mentioned, important reactions are related to a hard collisions and therefore it is necessary to eliminate minimum bias vertices and their tracks. When reconstructing a vertex and its properties, it is possible to distinguish between a hard reaction and a minimum bias reaction. On the basis of the transverse momenta of the corresponding tracks one can easily determine minimum bias vertices and eliminate them.
In addition the information of a vertex can be used for further kinematical reconstructions. For example considering that a neutral particle does not leave any signal in the tracker it is only possible to measure the energy and the final position of it in one of the calorimeters. With the knowledge of the original vertex of this neutral particle it is possible to estimate the momentum of the particle.

### 2.2.4 Beamspot of the LHC

The beamspot is the region in the beam pipe where the particle bunches collide. Hence this is the region where primary vertices reside.
The particle bunches are Gaussian-shaped and have a standard deviation of 50 µm in XY direction and between 5 cm and 10 cm in Z direction. The standard deviations of the beamspot are [4]:

$$\sigma_{XY} = 32\,\mu\text{m} \tag{2.1}$$

$$\sigma_{Z} = 4.9\,\text{cm} \tag{2.2}$$

On the basis of these numbers and the assumption that 99.7 % of the primary vertices lie in the $6\sigma$ region, I estimate the dimensions of the beamspot as such:

$$\sigma_{\text{BeamspotXY}} = 200\,\mu\text{m} \tag{2.3}$$

$$\sigma_{\text{BeamspotZ}} = 30\,\text{cm} \tag{2.4}$$

## 2.3 ATLAS detector at LHC

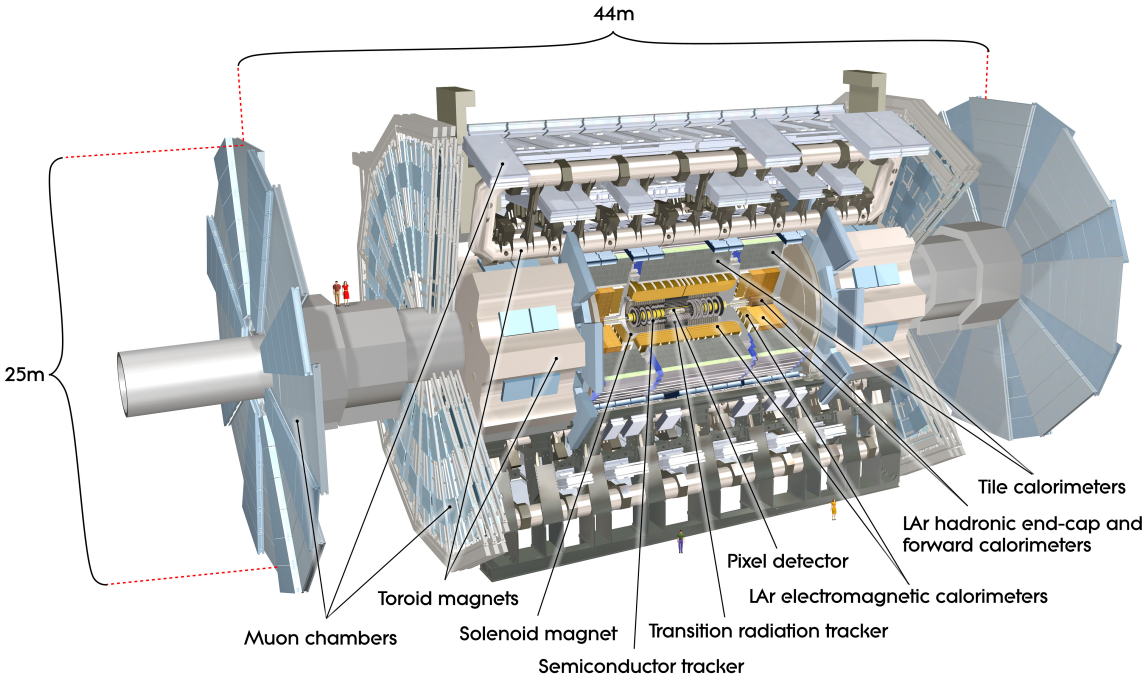



Figure 2.3: The ATLAS detector [5].

ATLAS was originally an abbreviation for A Toroidal LHC ApparatuS, but today ATLAS is the proper name for the detector. The ATLAS detector is 44 m long, measures 25 m in diameter and weighs about 7000 tons. It is built cylindrically symmetrical around the interaction point where the interaction takes place. It covers a solid angle of almost $4\pi$. The solid angle is restricted by necessary support pillars, data cables and the beam tube itself [4].
ATLAS is a general-pupose detector which has to detect many different particles. Therefore it has several differently specialised components. The inner components are specialised in tracking charged particles by detecting their interaction with material. Due to a strong magnetic field which is provided by a magnetic system that surrounds the inner components, the tracks of the charged particles are bent. This makes it possible to gain additional information from the tracks. The direction of the curve reveals the charge of a particle and the degree of a curvature reveals the momentum [6].
The innermost component is the pixel detector. It is made for extremely precise tracking close to the interaction point. Therefore it consists of over 80 million silicon pixels that can be read out simultaneously.
The next component is the semi-conductor tracker. It works similar to the pixel detector, but it does not achieve such a high resolution since it consists of long narrow silicon microstrip sensors instead of pixels.
Surrounding the semi-conductor tracker there is the transition radiation detector. This detector consists of drift chambers filled with gas. The impact of charged particles gets enhanced by transition radiation. Because of its structure this detector achieves a lower resolution compared to the two inner detectors. Around these three detectors the already mentioned magnet system is located.

The next components are the calorimeters. Calorimeters measure the energy of particles by absorbing them. There are two calorimeters, the electromagnetic calorimeter and the hadronic calorimeter. The electromagnetic calorimeter produces and detects electromagnetic showers. Therefore it is able to measure the energy of photons, electrons and positrons. The hadronic calorimeter measures the energy of the hadrons. Due to their high mass hadrons pass the inner components including the electromagnetic calorimeter only with small energy-losses. Compared to the electromagnetic calorimeter the hadronic calorimeter has a lower energy resolution.
The most outer detector is the muon system. It claims most of the volume of the ATLAS detector. The task of the muon system is to identify muons, since very few particles of other types are expected to pass through the calorimeters and leave signals in the muon system. To measure the momentum of the muons independently and more precise the muon system has its own magnet system.
The only particles which the ATLAS detector cannot detect at all are the neutrinos. Since they do not interact neither electromagnetically nor strongly they leave no signal in the detector. There is a way to detect them implicitly. Due to momentum conservation the sum of the transverse momenta of all created particles should be zero. When this sum deviates significantly from zero, there is a high probability that at least one neutrino was created.

### 2.3.1 Quantities that describe tracks in the ATLAS detector

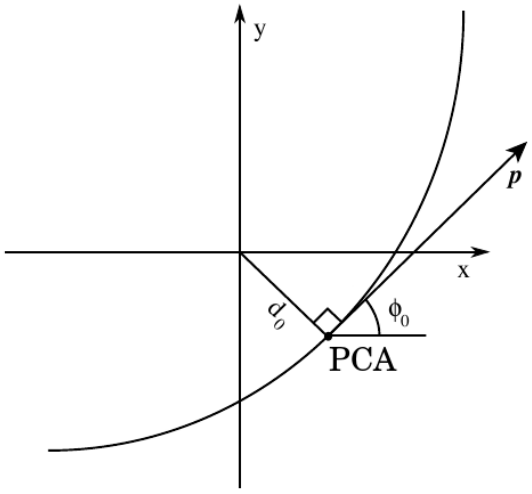



Figure 2.4: Track parameters [1].

To describe the route of a track it is necessary to define a coordinate system. The cartesian coordinate system at the ATLAS detector is defined in the following way:
Along the beam pipe lies the Z axis and the origin is put in the beamspot. The X and Y direction proceed in transversal direction to the Z axis. Due to the cylindrical symmetry of the ATLAS detector it is convenient to use cylindrical coordinates. Since the collision events show spherical symmetry, spherical coordinates are often used. In the spherical coordinate system the radial distance to the origin is depicted by the radius $r$. The azimuth angle $\phi$ lies in the XY plane and for $\phi = 0$ a point lies on the X axis. The polar angle $\Theta$ is measured from the Z axis and $\Theta = 0$ shows in Z direction. The polar angle is

often expressed by the Lorentz invariant pseudorapidity $\eta$.

$$\eta = -\ln\left(\tan\left(\frac{\Theta}{2}\right)\right) \tag{2.5}$$

In this equation $\Theta$ is measured in radiant. The pseudorapidity $\eta = 0$ corresponds to $\Theta = 90°$. When increasing $\Theta$ towards 180°, $\eta$ diverges towards negative infinity. When decreasing $\Theta$ towards 0°, $\eta$ diverges towards infinity. The acceptance range of the ATLAS detector in $\eta$ is $-2.5 < \eta < 2.5$ which corresponds to an opening angle of 80°.
As it is mentioned in section 2.3 charged particles move along bent tracks. There are several parameters that describe such tracks (see figure 2.4) [7]. These parameters refer to the point of the track where the distance to the Z axis is the smallest. This point is called simply point of closest approach (PCA).

$d_0$ This is the impact parameter of a track. It describes the distance of the PCA to the Z axis. This value also contains additional information about the angular momentum of the charged particle. When $d_0$ is positive the vector of the angular momentum shows in the direction of the positive Z axis. When it is negative the vector shows in the opposite direction. In the following chapters only the absolute value of $d_0$ is considered.

$\phi_0$ This is the angle between the momentum vector of the track and the X direction.

$\Theta_0$ This is the angle between the momentum vector of the track and the Z axis.

$z_0$ This is the Z coordinate of the PCA.

$\frac{q}{p}$ This is the fraction of the charge of the particle $q$ and the absolute value of the momentum of the particle $p$.

## 2.4 Properties of b-vertices

Before trying to identify b-vertices it is necessary to know the important properties of b-hadrons. As already mentioned in section 2.1 b-hadrons appear like the other quarks in jets. b-hadrons have an average mass of $m \approx 5.3\,\text{GeV}$ [9]. The lifetime of b-hadrons is $\tau \approx 1.6\,\text{ps}$. Both values are larger than the corresponding values for typical light or charm hadrons. Especially the high lifetime enables b-hadrons to travel macroscopic distances. To determine how long the b mesons fly, one has to consider that the lifetime is distributed exponentially.

$$N(t) = N_0 \cdot \mathrm{e}^{-\frac{\ln 2}{\tau}\cdot t} \tag{2.6}$$

Now one can calculate the time interval within 95 % of the b-hadrons decay.

$$0.05 \cdot N_0 = N_0 \cdot \mathrm{e}^{-\frac{\ln 2}{\tau}\cdot t} \;\Leftrightarrow\; t = 3\frac{\tau}{\ln 2} \tag{2.7}$$

The distance they cover in this time can be calculated with equation 2.8. For this calculation I assume an average $p_T$ of 50 GeV for b-hadrons. Dividing this $p_T$ by the mass of a b-meson $m = 5.3\,\text{GeV}$ results in a $\gamma$-factor of approximately 10. The result for the distance is $l = 15\,\text{mm}$.

$$l = \beta\,\gamma\,\mathrm{c}\,\tau \tag{2.8}$$

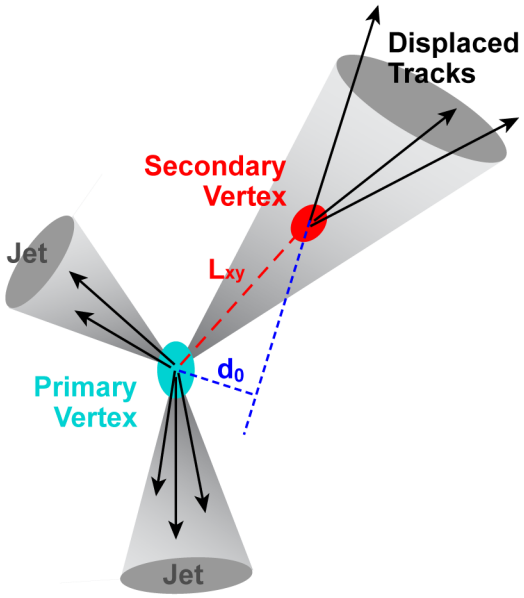



Figure 2.5: Illustration of a secondary vertex [8].

Because of the high range of b-hadrons the search volume has to be much larger than it is the case for primary vertices.
A further result of the high range of b-hadrons is an increased impact parameter of the tracks that belong to the b-vertex (see figure 2.5). Because of the high distance of the b-vertices to the Z axis the tracks are also displaced from the Z axis.
Another special property of b-hadrons is their ability to decay semileptonically. That means that the decay products may contain a lepton and its corresponding neutrino. The occurence of a neutrino may lead to problems in the reconstruction of the b-vertex (see figure A.4). Since a neutrino does not leave a track in the detector, it cannot be included in the vertex reconstruction. Especially the reconstruction of the vertex mass suffers from this fact, since it is determined by the four-momentum vectors of the tracks.
In the following chapters the identification of b-vertices is called b-tagging. Actually the term b-tagging refers to the investigation of a jet and determining if it contains a b-quark. However, since the consequences of finding a b-vertex in a jet and determining that a jet contains a b-quark are pretty much the same thus it is acceptable to use the term b-tagging like that.

## 2.5 Quantities for b-tagging evaluation

There are several quantities that emphasise different aspects of a b-tagger. On the basis of these values it is possible to quantify the results.

**b-tagging efficiency** The b-tagging efficiency describes the fraction of correctly tagged b-vertices. Hence if this fraction would be 1 if all of the existing b-vertices are found and if it is 0 none are found. This value is gained by investigating events that contain only b-vertices and one hard primary vertex.

$$\epsilon_b = \frac{n_{\text{btagged}}}{n_{\text{true}}} \tag{2.9}$$

**Charm rejection** This quantity is described by the fraction of the number of true charm vertices $n_{\text{charm}}$ and b-tagged vertices $n_{\text{btagged}}$. It is calculated by investigating a sample that contains only charm vertices and the unavoidable hard primary vertices. When investigating a sample with 1000 charm vertices and the b-tagger reconstructs 100 b-vertices the charm rejection would be 10. This means that every $10^{\text{th}}$ charm vertex is regarded as b-vertex. Since there are no b-vertices in the sample no b-vertices should be reconstructed and the charm rejection should be infinity.

$$R_{charm} = \frac{n_{\text{charm}}}{n_{\text{btagged}}} \tag{2.10}$$

**Light rejection** This quantity is calculated similar to the charm rejection, but refers to light vertices instead of charm vertices. Light particles include the gluons, the up, down and strange quark. Light rejection is described by the fraction of the number of true light vertices $n_{\text{light}}$ and the number of b-tagged vertices $n_{\text{btagged}}$. Once more this value has to be calculated on light quark samples. The higher the rejection is, the better is the result. The optimal value is infinity. In this case no vertices are b-tagged.

$$R_{light} = \frac{n_{\text{light}}}{n_{\text{btagged}}} \tag{2.11}$$

In general charm vertices are harder to distinguish from b-vertices than it is the case for light vertices. Hence the differences between charm and b-mesons are not as big as they are between light and b-mesons. Therefore the light rejection is higher than the charm rejection for most b-taggers. Nevertheless it is better this way since the number of light jets is gigantic compared to the number of other jets. Hence it is a very important task for b-tagging to achieve a high light rejection.

CHAPTER 3

# General-purpose computing on graphics processing units

## 3.1 Overview


Figure 3.1: This picture shows a typical graphics card with its GPU in the middle [10].

The main purpose of a graphics card is to calculate the pictures shown by one or several displays. These calculations are done on the graphics processing unit (GPU) on the graphics card. To produce a picture, a GPU has to calculate a colour for every pixel on the screen simultaneously. Driven by the high resolutions required by todays high-definition graphics, the computation power of GPUs has to be very high to accomplish this task (for example todays Full HD resolution consists of $2.07 \times 10^6$ pixel). The colour of a pixel may be determined by the position, the surface structure of an object or some properties of surrounding pixels. Instead of using these graphical informations to calculate a colour, one could change the set of input data to any kind data. This would allow us to perform any calculation on the GPU. Executing arbitrary calculations on a GPU is called general-purpose computing on graphics processing units (GPGPU). There are GPUs like the Nvidia Tesla series that are developed especially for this purpose. Their double precision is increased and since they do not have to provide any display, the graphics related components are cancelled.
The original purpose of GPUs affects their chip structure. Since they have to calculate a lot of numbers in parallel, a GPU consists of many small computing units. CPUs on the other hand have just a few computing units. This main difference in structure results in a big difference in computing performance

between the CPU and the GPU (see Fig. 3.2). But the high performance of the GPU can only be achieved if every computing unit is kept busy. Therefore algorithms have to be fully parallelized to achieve a fast execution speed. Hence the problem to be solved, has to be dividable in independent sub-problems. When just a few computing units are occupied the performance of a GPU drops far below the performance of a CPU. One reason is that the computing units of a GPU are slower than those of a CPU, but the more significant reason is that the computing units of a GPU have a smaller instruction set. More complex instructions have to be disassembled for the computing unit of a GPU while a computing unit of a CPU executes the instruction in one step.

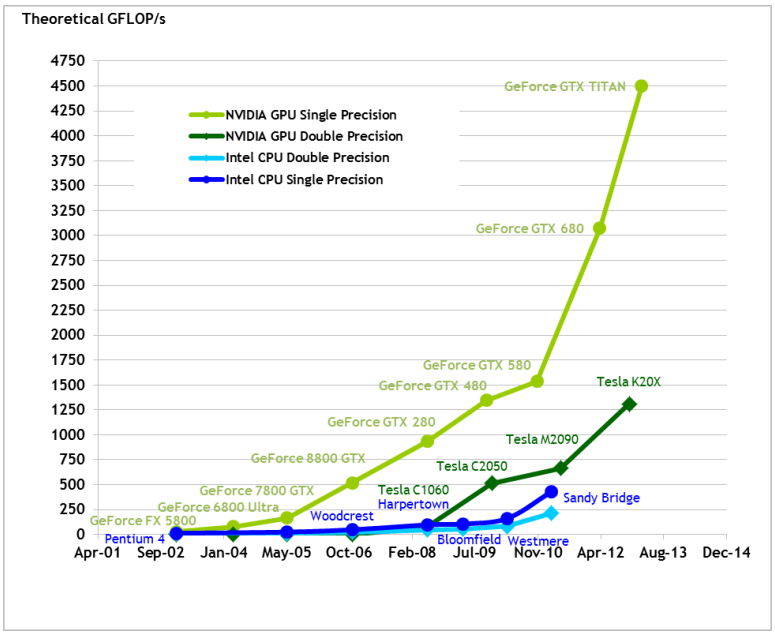



Figure 3.2: Performance comparison of different Intel CPUs and Nvidia GPUs. Performance is measured in billion floating point operations per second (GFLOP/s) [11].

When programing for GPUs it is very important to structure the program in a parallel manner and to have an environment which makes such an amount of computing units manageable. The two most common architectures for GPGPU are CUDA and OpenCl. CUDA (Compute unified device architecture) is introduced by Nvidia and can only be used on GPUs from Nvidia. OpenCL (Open computing language) is supported by much more vendors and it runs on a variety of different platforms. Except for platform dependent optimizations, OpenCL is portable between them. One of the supporters of OpenCL is also Nvidia thus in principle it is possible to run OpenCl programs on Nvidia GPUs. However Nvidia puts much more effort in its own general computing architecture CUDA. Nvidia already provides a better support of tools and libraries for CUDA than it is available for OpenCL. Some examples are the Fast Fourier Package CUFFT, the integrated development environment Nsight or the Nvidia Visual Profiler. In addition Nvidia makes new hardware features almost immediately available with new versions of CUDA. In contrast to this, new hardware features get adopted only very slowly in the OpenCL standard [12].
For my thesis I use the CUDA environment especially because of the well developed tool Nsight and the fourier transformation library CUFFT.

## 3.2 CUDA

CUDA is based on the programing language C, but it includes an additional set of instructions to control the special GPU features. The principle of CUDA is that the CPU controls the calculations of the GPU. A CUDA program begins on the CPU with the loading and preparing of data for processing. The next step is to allocate the necessary memory space on the GPU and to copy the data from RAM to the memory of the GPU. Afterwards the CPU hands over the data processing to the GPU. Within CUDA the processing on the GPU is represented by an abstract layer of "blocks" and "threads". A CUDA program consists of a grid of blocks and every block handles a part of the calculation (see figure 3.3). Every block in turn consists of threads that do the actual calculation. The function that the threads execute is called kernel. For example, when starting a grid with 10 blocks and 10 threads per block the kernel will be executed 100 times. To distinguish the work of the thread, every thread and every block has its own index number. So for example when adding two vectors, one thread can determine by its index which entries it should add. An additional feature is that the grid can be two dimensional and a block even three dimensional. This simplifies the distribution of threads for two or three-dimensional problems.

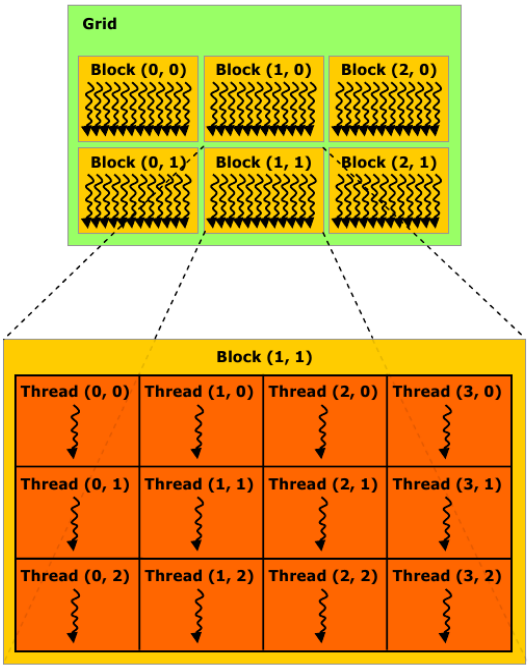



Figure 3.3: A CUDA program runs a grid of blocks. Each block consists of threads which actually do the calculations. The grid can be organized in one or two dimensions and a block can be organized in up to three dimensions. The dimensions are expressed in the index numbers of each block and thread [**programmingguide**].

In addition to the abstract layer based on blocks and threads, CUDA includes a memory model to handle the different types of memory on a graphics card (see figure 3.4). These types have different properties and they are accessed differently. When storing any data on a GPU one has to consider these differences and choose the appropriate memory type. These types of memory are introduced by CUDA:

- Private registers for every thread
- Local memory for every thread

- Shared memory for every block
- Global memory
- Texture memory
- Constant memory

Every thread has his own private registers where its local variables are stored. These variables can only be accessed by this particular thread and have the same lifetime as the thread. The required memory space has to be known before compilation. The registers are very fast though their capacity is very limited (see section 3.3). When variables do not fit into the registers they are allocated in much slower local memory by the compiler. The access properties of the variables in local memory stay the same. Local memory is only necessary due to memory space limitations of the registers, otherwise it should be avoided.
Variables in shared memory are allocated per block and can only be accessed by the threads of this particular block. The lifetime of the stored variables is the same as the lifetime of the block. The memory space for shared variables has to be known before invocation of the kernel. Hence this memory type is shared, the threads are able to communicate with each other. On the other hand it is possible that different threads read and write at the same position at the same time. To avoid these so called bank conflicts, CUDA provides several instructions. Here are two examples:

**`__syncthreads()`** Acts as a barrier at which all threads in the block must wait before any is allowed to proceed.

**`atomicAdd(`*`memory adress, value to be added`*`)`** Adds a value to the variable at the memory adress. During this operation no other thread can access this memory adress.

These instructions are also convenient for global memory, since it can be accessed by every thread of every block. The lifetime of variables stored in global memory is as long as the runtime of the CUDA program. These variables are allocated in the CPU part of a CUDA program. The required memory space has to be fixed before the allocation and can not be changed afterwards. The global memory is the biggest, but also the slowest memory on the GPU. Hence it should be used only for data that does not fit into the other memories.
Texture memory is an abstract layer that accelerates read accesses to global memory. So the allocation is done by allocating memory space on global memory. This memory space gets associated to a variable in texture memory. When actually accessing this variable, texture memory reads from global memory. The caching feature comes in handy because accessing global memory suffers from a high latency. Like global memory, variables can be accessed by every thread of every block and have a lifetime as long as the runtime of the CUDA program.
Constant memory is a read-only memory designed for constant values that get accessed frequently. Constant memory has the same access properties like global memory. So constant memory can be used to avoid accesses to global memory.

## 3.3 Hardware implementation of CUDA

To understand how the program gets executed on the hardware one has to distinguish between a computing unit which is called CUDA Core and a streaming multiprocessor. A streaming multiprocessor contains a group of CUDA Cores and manages them (see figure 3.5). The blocks are executed on

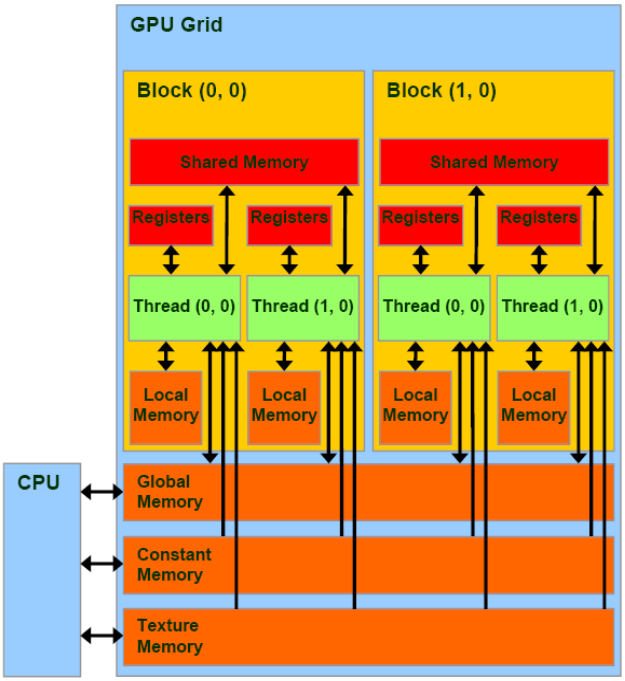



Figure 3.4: This figure depicts the memory model of the CUDA environment. Every thread owns private registers and private local memory. Each thread of a block has access to the shared memory of that block. Every thread in the grid has access to the global memory and can read from constant and texture memory [13]

these streaming multiprocessors. That means the threads of that block are distributed among the CUDA Cores. Depending on the number of threads and the memory space a block requires it is also possible that one streaming multiprocessor executes several blocks. This makes the CUDA environment more independent from the underlying hardware (see figure 3.6). In addition to the CUDA Cores a streaming multiprocessor contains other componentes that are related to the different memory types. One of these parts is the register file. This register file is occupied by the private registers of the threads that are executed by the CUDA Cores. This memory type is very fast, but the capacity is very low. Another part of a streaming multiprocessor is the shared memory. This memory cache is distributed among the blocks that get executed by this particular multiprocessor and has a similar bandwith like the register file.
The sizes of these two memory types are the main values that determine the parallelism of the execution. For example considering a block requiring 10 kB of shared memory and a streaming multiprocessor has only 16 kB shared memory this streaming multiprocessor can only execute one block. However when the memory usage for a block is low it can execute several blocks simultaneously. A corresponding problem occurs if the threads of a block have a high memory consumption. When the threads require a large amount of register memory space, it is not possible to execute all threads on the streaming processor. This problem forces some CUDA Cores to stay idle, wasting computation power. Even when considering local memory which can be occupied by the threads, the problem remains. That is because local memory is actually a part of global memory. Global memory has a high capacity, but is not localized on the chip making it by far the slowest memory on a graphics card. A quantity which describes the duration of a memory access is the latency. Measured in clock cycles the latency of global memory is approximately by a factor 100 longer than the latency of on-chip memory. So it is recommended to avoid global memory accesses as much as possible.

To reduce the latency of global memory one can use texture memory. Every streaming multiprocessor contains a texture cache that caches accesses to the global memory. For example when accessing an entry in a large array in global memory via texture memory the entries arround that entry get cached. This cache can only be used for reading from global memory. It is not possible to use it for accelerating write accesses. Another way to avoid global memory access is to use constant memory. Constant memory is also localized on-chip and caches frequently used variables. But once assigned, the stored values can not be changed during runtime.

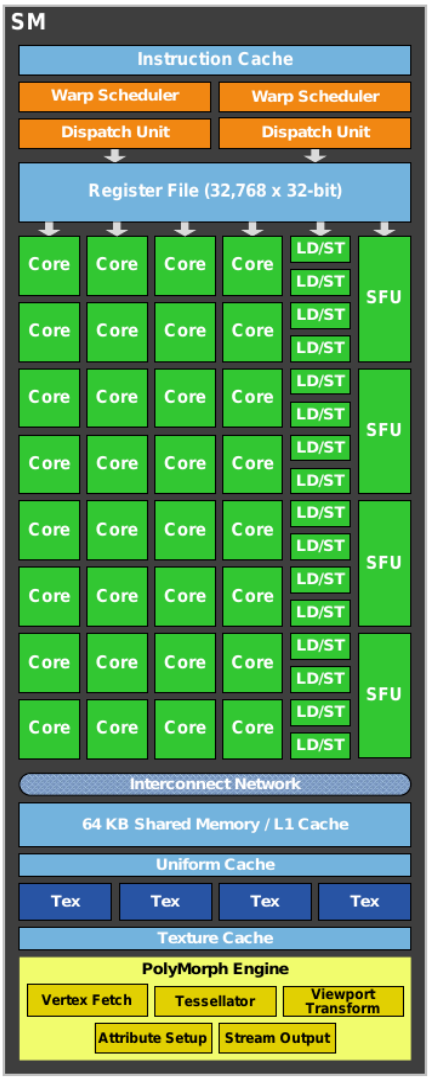



Figure 3.5: This is a scheme of a streaming multiprocessor based on the Fermi architecture of Nvidia (compute capability 2.0). It can be seen that the green coloured CUDA Cores dominate. There are also special function units (SFUs) which are responsible for executing transcendental instructions such as sine, cosine, reciprocal and square root. The load/store units (LD/ST) are supporting units that load and store data. The parts that schedule the CUDA Cores are located at the top of the picture. The instruction cache stores the instructions that will be scheduled and distributed to the CUDA Cores by the warp scheduler and the dispatch unit. The other blue parts represent the different memory types. The yellow parts at the bottom of this figure are just used for graphics applications.

## 3.4 Compute capability

The CUDA compute capability version number characterizes the properties of the core architecture of the GPU. The first digit determines of which generation of CUDA enabled architectures the GPU is. The second digit determines the feature set the GPU supports. For example all GPUs of compute capability 1.x have nearly the same multiprocessor structure. Every multiprocessor has 16 kB shared memory and 8 CUDA Cores. These CUDA Cores also are managed in the same way. The difference

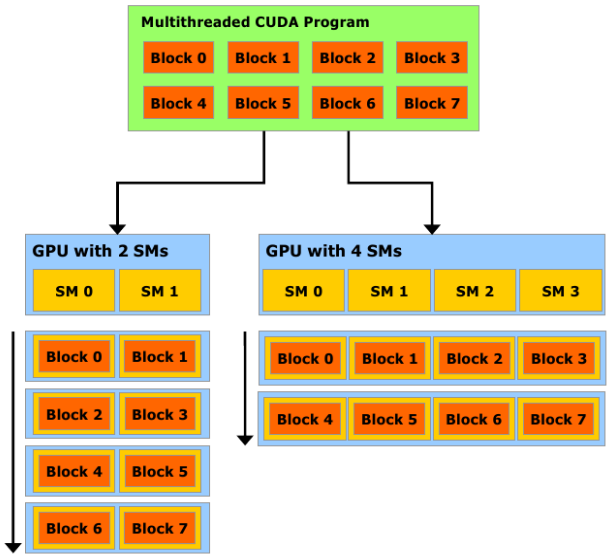



Figure 3.6: All blocks can be executed on any Nvidia GPU independent of the number of streaming multiporcessors.

between the 1.x GPUs lies in the support of certain scheduling commands and the access pattern of the different memory types. Comparing compute capability 1.x with the higher generations 2.x and 3.0, many components of the multiprocessors change. The amount of memory space and number of CUDA Cores per multiprocessor increases, the management of the CUDA Cores changes and new features and commands are implemented.

Another major difference between the compute capability versions is the precision. In principle all Nvidia GPUs follow the `IEEE 754-2008` standard for binary floating-point arithmetic. GPUs with compute capability 2.0 or higher deviate only slightly in the way floating-point exceptions are handled on a CPU. For GPUs with a lesser compute capability there arise additional more critical deviations between GPU and CPU calculations. GPUs below compute capability 1.3 do not support double-precision floating-point numbers. In calculations with single-precision floating-point numbers GPUs of compute capability 1.x show deviations that influence the precision. For example numbers that are close to zero get flushed to zero. Operations such as square root and division are implemented in a non-standard-compliant way also leading to reduced accurancy. This reduced accurancy leads to discrepancies between results calculated by a CPU and a GPU.

## 3.5 Nvidia Tesla GPU

I worked with a Nvidia Tesla M1060. The major specifications of this GPU are:

- CUDA Compute Capability version number: 1.3
- Total amount of global memory: 4096 MB
- 30 Multiprocessors with 8 CUDA Cores each: 240 CUDA Cores
- GPU Clock rate: 1.30 GHz
- Memory Clock rate: 800 MHz

- Memory Bus Width: 512 bit
- Total amount of constant memory: 64 kB
- Total amount of shared memory per multiprocessor/block: 16 kB
- Total number of 32 bit registers available per multiprocessor/block: 16384
- Maximum number of threads per multiprocessor: 1024
- Maximum number of threads per block: 512
- Maximum sizes of each dimension of a block: $512 \times 512 \times 64$
- Maximum sizes of each dimension of a grid: $65\,535 \times 65\,535 \times 1$

## 3.6 PC specifications

The installed RAM has 4 GB of memory space and the CPU is an Intel Core 2 Duo E6750 processor. The major specifications of this CPU are:

- Number of cores: 2
- Clock rate: 2.66 GHz
- L2 cache: 4 MB
- Front-side bus clock rate: 1333 MHz

The L2 cache is a memory cache that is shared among the cores. It is an additional memory to the two L1 caches each core owns. The front-side bus clock rate describes the speed of the communication between the CPU and the other components of the PC, like the RAM or the GPU.

CHAPTER 4

# Medical imaging vertex finder

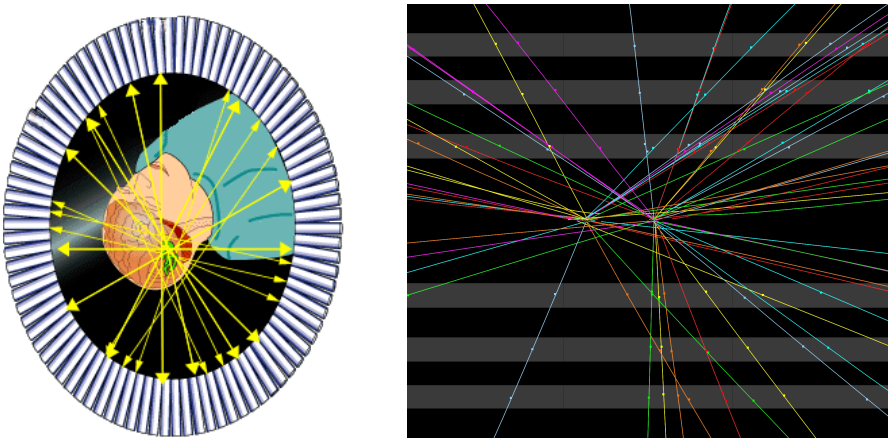
The medical imaging vertex finder has been developed and implemented by Stephan Hageböck in his diploma thesis [1]. This vertex finder is based on techniques that are used in medicine to display inner parts of the human body. In the following section I will use the abbreviation MIVertexfinder for medical imaging vertex finder. In the next section I show an example that explains why it is convenient to use medical methods. Then I will explain the medical imaging vertex finder itself.

## 4.1 Tomography

The term tomography refers to a group of imaging methods which reconstruct the inner structure of an object by sections. In medical branches several of these methods are used in applications like x-ray computed tomography, positron emission tomography, and magnetic resonance imaging. Using the example of the positron emission tomography one can explain very well the analogy between reconstructing parts of the body and reconstructing vertices. Positron emission tomography requires the injection of a positron emitter into the patient. The emitter isotope is part of a molecule suitably chosen such that it accumulates at points of interest, for example cancer metastases. The emitted positrons and local electrons annihilate and release two photons back to back. These photons are detected in a coincidence measurement by a ring detector which is arranged around the patient. After the data is recorded the reconstruction can start. The two essential steps that most tomography methods have in common are the backprojection of the data into a cartesian coordinate system and the application of a filter to reduce artefacts which occur due to the backprojection. For positron emission tomography the backprojection of the paths of the photons looks like figure 4.1a. As one can see the resulting image looks very similar to the image of primary vertices in the ATLAS detector 4.1b. The paths of the photons correspond to the tracks coming from a vertex and the vertices correspond to the points of interest at positron emission tomography. Due to these structural similarities it is convenient that the reconstruction method used for positron emission tomography is also able to find vertices. In the next section it is described how the medical imaging method works to find vertices.

## 4.2 Algorithm

The main object in the medical imaging vertexfinder is a three dimensional histogram. Every bin of this histogram covers a certain spatial range in each of the three dimensions and contains a value for



(a) PET [1]. (b) Collision event with two primary vertices [14].

Figure 4.1: Comparison between a PET-image and a collision event with two primary vertices.

the vertex density at its position. The vertex density describes the probability of the existence of a vertex. In an area with high bin entries in the histogram it is convenient that there is at least one vertex. The source of these density values in the histogram are particle tracks of an event. Every bin entry is increased according to the number of particle tracks that cross that particular bin. Increasing the bin entries according to the route of tracks is called backprojection.
After backprojecting the tracks into the histogram the histogram gets filtered to reduce artefacts and point out vertices in the vertex density distribution. Due to mathematical relations the filtering becomes easier when it is done in frequency space. Hence it is necessary to apply a fourier transformation to the vertex density distribution before applying a filter function. Due to the discrete structure of the fourier transformation it is necessary to apply a window function in frequency space. In addition to the filter function a simple noise reduction is applied. After transforming the vertex density distribution back into euclidean space the highest vertex densities in the distribution get clustered to probable vertex candidates. A brief recapitulation of the MIVertexfinder algorithm:

- Backprojection of the tracks
- Fourier transform
- Apply filter function
- Apply window function
- Reduce noise
- Inverse fourier transform
- Clustering

## 4.3 Backprojection

Backprojection creates a probability density distribution which is stored in a three dimensional discrete array. Therefore the backprojection procedure increases the entry of every bin along a track by a certain

value. There are several possibilities to choose this value. A common method is to increase the bin entry by a value proportional to the length of that part of the track that lies inside the bin. A simpler way to perform the backprojection is to increase the bins by the same value. This simpler method performs even better than the common one as Stephan Hageböck already showed in his thesis [1]. In this analysis the second method is chosen and the value by which the bins are increased is one. In figure 4.2 the result of a backprojection can be seen. It is already possible to recognize possible vertices in this picture.

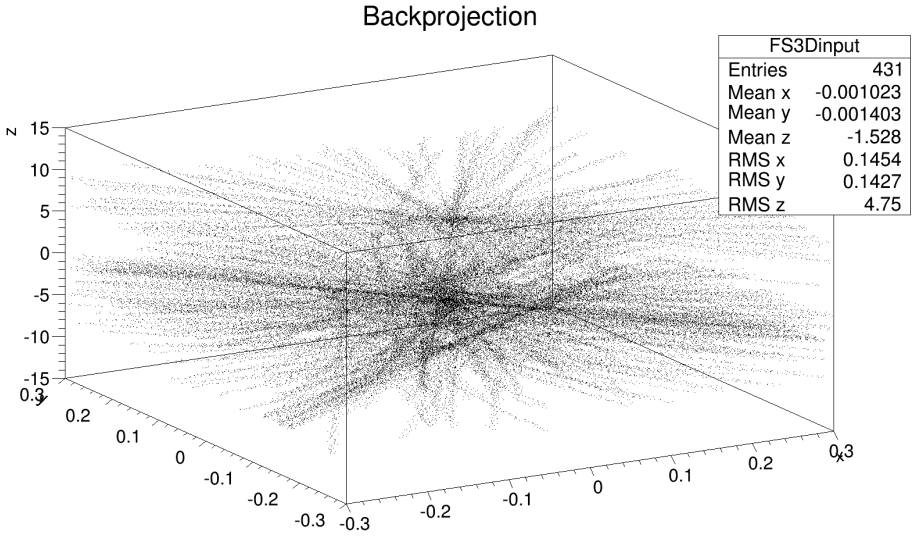



Figure 4.2: This figure shows 431 tracks of one event backprojected into a three dimensional histogram. It is already possible to spot some primary vertices on the Z axis.

## 4.4 Fourier transformation

The fourier transformation calculates the frequency spectrum of a given signal. A time interval $\Delta t$ is mapped to its corresponding frequency $\omega$.

$$\mathrm{FT}\{f(t)\}(\omega) = \int_{\mathbb{R}} f(t) \cdot e^{-i\omega \cdot t} \, \mathrm{d}t \tag{4.1}$$

Analoguely, a spacial interval is mapped to its spacial frequency $\vec{k}$. The inverse fourier transformation is denoted by a positive signed exponent.

$$\mathrm{FT}\{f(\vec{x})\}(\vec{k}) = \int_{\mathbb{R}^N} f(\vec{x}) \cdot e^{-i\vec{k} \cdot \vec{x}} \, \mathrm{d}\vec{x} \tag{4.2}$$

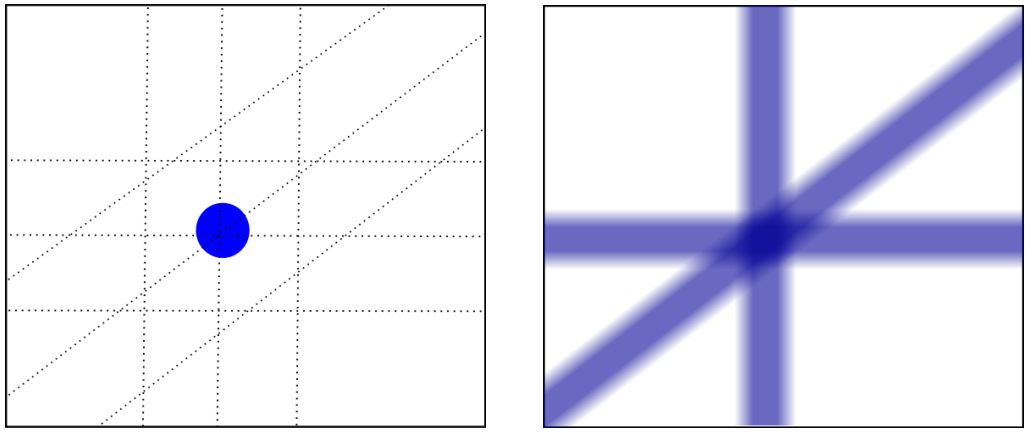
One has to consider that the calculations are done on a PC therefore the discrete fourier transformation is used. That means the area has to be divided into discrete points $x_j$. The spectrum of these points $X_k$ is calculated like this:

$$X_k = \sum_{j=0}^{N-1} x_j \cdot e^{\frac{-i2\pi kj}{N}} \tag{4.3}$$

The discrete fourier transformation causes several problems which require the application of a window function. These problems and the impact of the window function are described in chapter 4.6.

## 4.5 Filter application

The crucial part of the MIVertexfinder, the filtering, reduces noise and combinatorial artefacts created by the backprojection. As it can be seen in picture 4.3b when several tracks cross one point, the shape of the source of these tracks remains unclear. At the intersection point star-shaped artefacts occur. When more tracks are added the resulting shape would be an overlay of more such artefacts. This causes a blurred picture of the original shape. The blurr effect can be described by the a spread function $s(x, y, z)$. This function characterises the response of a backprojection of a point-like source. To calculate one



(a) Original (b) After backprojection

Figure 4.3: The result of a backprojection is a star-shaped artefact instead of the round original dot [1].

point of the blurred picture, every point of the original picture has to be multiplied by the point spread function of that point and all contributions have to be summed up. The summation over the contributions of all points can be explained with the example of two neighbouring points. Even if they do not touch each other, the blurr effect may cause an overlay of the edges, leading to another picture. The described blurring effect corresponds to a convolution of the original picture with the point spread function.

$$f_{\text{backproj}}(x, y, z) = \underbrace{\iiint_C f_{\text{orig}}(x - a,\ y - b,\ z - c) \cdot s(a, b, c)\, \mathrm{d}a\, \mathrm{d}b\, \mathrm{d}c}_{\text{Application of the filter function}} = \underbrace{(f_{\text{orig}} * s)(x, y, z)}_{\text{Convolution}}$$

$f_{\text{backproj}}$ stands for the backprojected picture that contains the blurred artefacts and $f_{orig}$ represents the original picture. $(x, y, z)$ are the coordinates of the point that is calculated in this line. $(a, b, c)$ are the integration variables which cover the whole space $C$. This calculation becomes simpler when looking at the fourier transformation of it. The convolution in euclidean space is equivalent to a product in fourier space.

$$\mathrm{FT}\{f_{\text{backproj}}(x, y, z)\}(u, v, w) = \mathrm{FT}\{f_{\text{orig}}(x, y, z)\}(u, v, w) \cdot \mathrm{FT}\{s(x, y, z)\}(|\vec{k}|)$$

FT represents for the fourier transformation of the following function. In the next equations the fourier transformation of a function is represented by its capital letter. $(u, v, w)$ are corresponding coordinates

in fourier space. $\vec{k} = (u, v, w)$ is a vector in frequency space. From this equation it is possible to deduce a method to determine the original picture.

$$\begin{aligned} F_{\text{backproj}}(u,v,w) &= F_{\text{orig}}(u,v,w) \cdot S(|\vec{k}|) \\ \Leftrightarrow F_{\text{orig}}(u,v,w) &= \frac{F_{\text{backproj}}(u,v,w)}{S(|\vec{k}|)} \\ \Leftrightarrow f_{\text{orig}}(x,y,z) &= \text{FT}^{-1}\left\{\frac{F_{\text{backproj}}(u,v,w)}{S(|\vec{k}|)}\right\}(x,y,z) \end{aligned}$$

With the help of the last equation one gains the original picture by multiplying the fourier transformation of the backprojection by the function 4.4 and transforming the product back into euclidean space. In this context it is convenient to define function 4.4 as filter function.

$$h(k) = \frac{1}{S(|\vec{k}|)} \tag{4.4}$$

The filter function that is applied in the MIVertexfinder was made by Colsher [15] and adapted to a cylindrical detector by Defrise [16].

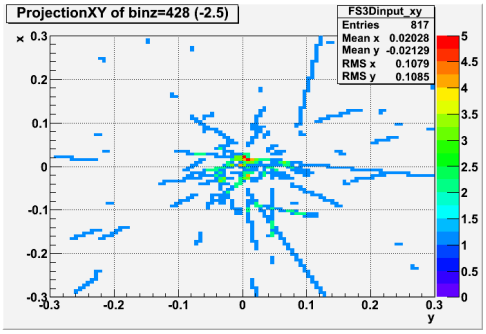
$$h_{\text{colsher}}(x) = \begin{cases} \frac{|\vec{u}|}{2\pi} & \text{if } \cos\phi \geq \cos^3\Theta \\ \frac{|\vec{u}|}{4\arcsin\left(\frac{\sin\Theta}{\sin\phi}\right)} & \text{if } |\cos\phi| < \cos\Theta \end{cases} \tag{4.5}$$

$\vec{u}$ is a vector in frequency space, $\phi$ the polar angle and $\Theta$ the acceptance angle of the detector measured in relation to the XY plane.

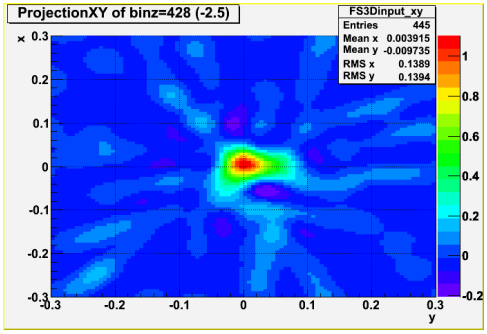
The impact of the filter function can be seen in these figures 4.4. Both figures display a XY plane of the histogram at the position of a primary vertex. Figure 4.4a shows this plane after backprojection and figure 4.4b shows it after the application of the filter function, the window function and the noise reduction.
In figure 4.4a one can already guess the position of the vertex, but a precise determination is not possible because of the spurios parts of the tracks. The colours in the histogram represent the entries of the bins. Since the bins along the track are increased by one, only integer numbers occur in the histogram.
After the filtering procedure the vertex can be spotted very clearly. One may also recognize that the area around the vertex is on a very low level showing that the unwanted parts are filtered out well. The colours still represent the bin entries, but without any specific unit. As already mentioned, before applying the filter function a fourier transformation is necessary.

## 4.6 Window function

A window function is a periodic function limited to a certain range and outside this range it is zero. It is used to avoid spectral leakage in the fourier transformation.
Spectral leakage is an effect that occurs during a fourier transformation when the form of a signal causes the appearance of additional frequencies. There are two kinds of spectral leakage in a discrete fourier transformation. The first one is that a recorded signal cannot be of unlimited length. The consequence can be seen in figure 4.5. An infinite signal with a single frequency gets transformed into a single value in frequency space. Nevertheless if the same signal is restricted in its range several additional frequency contributions in the transformation occur. The second kind of spectral leakage occurs especially in the




(a) This figure shows a XY plane of the histogram after backprojection. The colour stands for the bin entry. A bin entry of two means that two tracks cross. It can be guessed that in the middle of picture should be a vertex.




(b) This figure shows a XY plane of the histogram after the application of the filter function, window functionand noise reduction. In this case the colour still represents the bin entry, but in arbitrary units. The vertex can be spotted very clearly.

Figure 4.4: Both figures show the same XY plane of the three dimensional histogram, but at different processing steps. The spacial measure is cm.

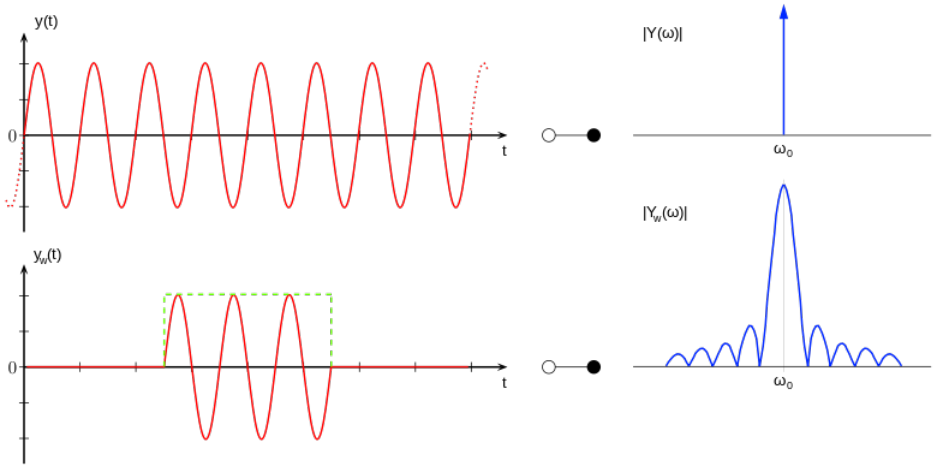



Figure 4.5: In the upper row can be seen that a signal with a single frequency gets transformed into a single peak in frequency space. The fourier transformation of a limited signal produces additional unwanted frequencies [17].

discrete case of the fourier transformation. But if the observed interval is a multiple of the signal period, the periodicity gets captured correctly and the transformation works fine. But when this is not the case the frequency of the signal gets shifted. This can be seen in figure 4.6. In the upper row of this figure the ideal case is shown. The scanning points capture exactly the three periods of the signal and the fourier transformation on the right side works correctly. In the lower row the signal period seems to be longer caused by the choice of the scanning points. This apparent longer signal period leads to a lower frequency in the fourier transform. In most cases there are a lot of different frequencies in the signal making it impossible to adjust the observed intervall to every occuring period. So spectral leakage cannot be avoided in discrete fourier transformations. Window functions help to reduce the impact of

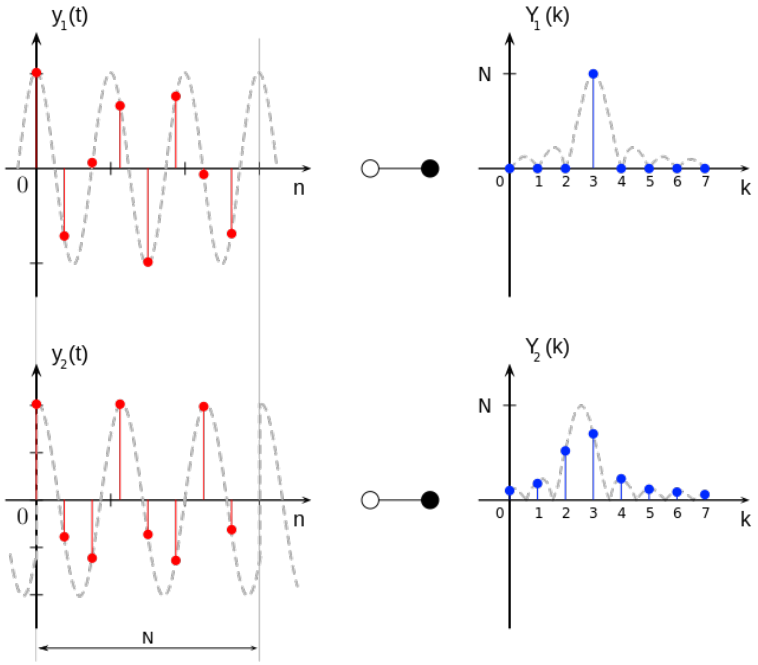



Figure 4.6: In the upper row the observed intervall is a multiple of the signal period the periodicity gets captured correctly and the transformation works fine. In the lower row the signal period seems to be longer caused by the choice of the scanning points. This apparent longer signal period leads to a lower frequency in the fourier transform [17].

the spectral leakage. In the first case of spectral leakage caused by the limited signal, the application of a window function leads to a smoother behaviour at the edges. This reduces the additional contributions in the frequency space. Adjusting the period of the window function to the length of the observed intervall reduces frequency shifts which occur due to the second kind of spectral leakage.

$$w(n) = 0.64 + 0.46 \cdot \cos\left(\frac{2\pi n}{M}\right), \quad n = -\frac{M}{2}, ..., \frac{M}{2} - 1 \tag{4.6}$$

This equation 4.6 is an example for a one dimensional window function. It is called Hamming window function. $M$ is the width of the window and $n$ is the current index of the signal. The window function can be extended to three dimensions by using the window function for all three dimensions. In this case one has three sizes $M_1$, $M_2$, $M_3$ and indices $n_1$, $n_2$, $n_3$. In the application of the window function the sizes of the window function correspond to the dimensions of the histogram.

## 4.7 Noise reduction

In addition to the already mentioned filtering, a noise reduction is applied in the fourier space. It is based on the Nyquist–Shannon sampling theorem. According to this theorem a bandlimited signal can be properly reconstructed if the sampling rate is twice as high as the Nyquist frequency. This frequency is the highest expected frequency occuring in the signal.

$$2 \cdot f_{Nyquist} = f_{sampling} \tag{4.7}$$

$$f_{signal} < f_{Nyquist} \tag{4.8}$$

In the case of the MIVertexfinder the Nyquist frequency is determined by the bin width of the 3D histogram. For example a bin width of 1 mm corresponds to a sampling rate of 1 $\mathrm{mm}^{-1}$ which leads to a Nyquist frequency of 0.5 $\mathrm{mm}^{-1}$. The Nyquist–Shannon sampling theorem also indicates that frequencies higher than the Nyquist frequency correspond to noise. Therefore these frequencies can be cut off by simply multiplying the corresponding bins in the histogram by zero. The downside of the noise reduction is that the vertex density distribution is not as sharp as before which also influences the effects of the above-mentioned filter function. So it might be convenient to choose a cut off frequency that is higher than the Nyquist frequency.

## 4.8 Clustering

In the clustering procedure vertex positions are extracted from the histogram. First the bin with the largest vertex density is found and neighbouring bins are clustered together to form a vertex candidate. The explicit algorithm is explained in section 5.5.

## 4.9 Analysis of time-consumption in the standard MIVertexing algorithm

To prepare the optimisation of this algorithm it is necessary to determine which parts take most of the time on a CPU. Especially these parts have to be considered to save computation time. Tools that can provide information about the time-consumption are included in the Valgrind toolsuite [18]. These tools are able to measure different parameters of the program and the containing procedures. The result can be seen in figure 4.7. It can be seen that these are the most time-consuming procedures:

23 % **`SetDataForTransform`** Converts the histogram into a complex array

30 % **`FastFourierTransformation`** Executes the fourier transformation of the complex array.

27 % **`filterUsing`** Applies the filter function, window function and the noise reduction.

17 % **`FillHistogramEuclideanSpace`** Converts the complex array back into a three dimensional histogram.

All together they take 97 % of the runtime. So it is very important to accelerate especially these four procedures to save the most computation time.

The two conversions are necessary because of the reasonable fact that the histogram format is not able to store complex numbers which are necessary to perform fourier transformations. To store the

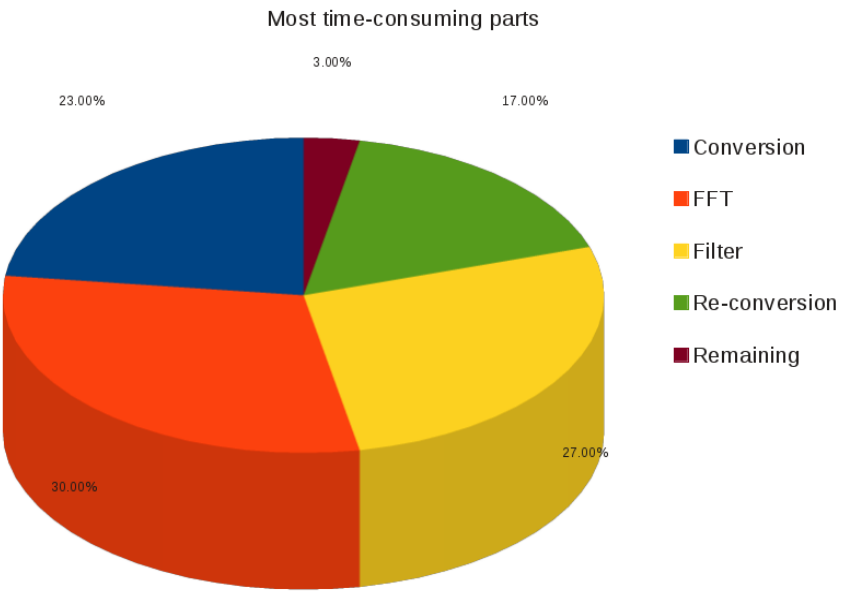



Figure 4.7: This diagram shows the four most time-consuming parts of the MIVertexfinder.

histogram the `TH3D` class from the ROOT package is used [19]. To perform the fourier transformation the FFTW library is used [20]. This library requires an array of `fftw_complex` elements. `fftw_complex` is a data type contained in the FFTW library that stores a complex number.

CHAPTER 5

# Implementation of parallel execution of MIVertexing algorithm

## 5.1 First steps

As it is mentioned in the previous chapter the foundation of the algorithm is the three-dimensional histogram. Since the histogram occupies a lot of memory it is necessary to store it in global memory. For primary vertex reconstruction the dimensions of the array are $96 \times 96 \times 1024$. Considering a 32 bit floating point number for every bin this results in 36 MB of required memory space. To avoid the time-consuming conversions mentioned in 4.9. I built the histogram according to the requirements of the fourier transformation library (see section 5.3). Because of this the necessary memory space for the histogram increases to 72 MB. To store the dimensions of the histogram the constant memory of a GPU comes in very handy. Since the dimensions are accessed very often, but stay constant all the time.
In the following sections I will explain the algorithms I used to parallelise the different components of the MIVertexfinder.

## 5.2 Backprojection

This part of the vertex finder consumes not a lot of time compared to the already mentioned procedures. This is mainly because the number of bins which have to be accessed during backprojection is considerably less ($\sim 1 \times 10^4$ compared to $9 \times 10^6$). But the backprojection can be implemented very well for parallel execution. The tracks are obviously independent from each other so they can be backprojected all at the same time. Therefore every track is assigned to one block and the threads of these blocks are assigned to one bin in X direction.
Since every bin has a spatial extent, it has a lower and an upper edge in X direction. When a track intersects these edges the two crossing points define a range in Y direction within several bin entries may have to be increased. In figure 5.1 a two-dimensional backprojection is illustrated. In this case every bin enclosed by the intersection points in Y direction is increased, provided that these bins are still located within the histogram limits. In the three-dimensional case additional intersection points of every bin in Y direction have to be calculated. These new intersection points determine a range in Z direction within the bins have to be increased, provided that they lie inside the histogram range. These steps can be executed independently for every single bin in X direction. Because of this massive par-

allelism one has to consider that two or more threads may try to access the same bin at the same time. To avoid such problems every operation which accesses the histogram has to be made with `atomicadd`. This command forces following threads to wait until the current add operation is finished. Before the

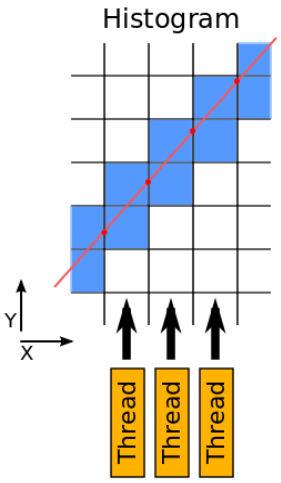



Figure 5.1: The red dots mark intersection points of the track and the edges of the bins in X direction.

actual backprojection can be started one has to consider that the information about the tracks have to be converted into a CUDA compatible format and copied to GPU memory. This task has to be done in addition to the backprojection. This is a typical disadvantage of GPU programs since the CPU can start processing without this step. However when utilising the GPU optimally this can be compensated by the computation power of the GPU.

## 5.3 Fourier transformation

The fourier transformation on the GPU is done by the CUFFT library from Nvidia. Similar to the FFTW library, the CUFFT library also requires a certain data format. The histogram has to be stored in an array of `cufftComplex` entries. The `cufftComplex` type is a simple data type consisting of two 32 bit floating point numbers. These two floating point numbers are necessary for storing complex numbers. The size of that array has to be written in the form $2^a \cdot 3^b \cdot 5^c \cdot 7^d$. Other sizes are also possible, but slow down the algorithm. Especially sizes of the form $2^a$ are recommended (see figure 5.2). When adjusting the histogram size it has to be considered that the algorithm requires additional memory while running. This additional memory may rise up to a multiple of the histogram size. To avoid time-consuming conversions as mentioned in chapter 4.9, I build a CUDA histogram class based on the conditions required by CUFFT. Except for these conditions one has to consider that the fourier transformation is not normalized. But this can be easily achieved by dividing every bin entry by the overall number of bins.

## 5.4 Filter function, window function and noise reduction

Since the application of the filter function, the window function and the noise reduction work the same way, they can be performed in the same procedure. These three steps are executed by multiplying every bin by a value which is determined only by the particular position in the histogram. The filter value is determined by the function 4.5 and the value of window function by a function like 4.6. For

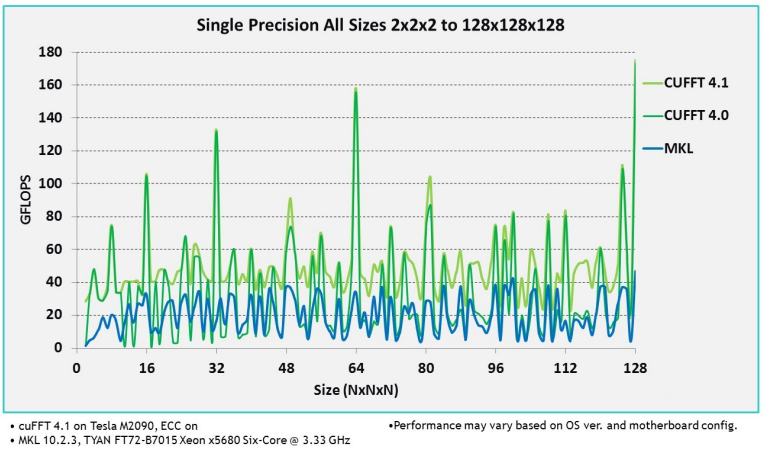



Figure 5.2: Nvidia compared the performance of the CUFFT library depending on the size of the used data array. The performance peaks at array sizes that can be expressed by the power of two. Additionaly the performance between CUFFT and MKL is compared. MKL stands for Math Kernel Library and is a math library made by Intel.

noise reduction the value is one and becomes zero when the limits given by the cut off frequency is exceeded (see section 4.7). That means that the calculations for a single bin are independent from the calculations for the other bins. This can be exploited perfectly for parallel execution. In addition the memory consumption of these calculations is very low since there are only few input variables and it is not necessary to store any intermediate results. The low memory consumption makes it possible for the streaming multiprocessors to run several blocks simultaneously since the number of blocks is often restricted by the memory consumption. At the beginning of the execution, the histogram is distributed among the threads. Every thread is assigned to a row in Z direction where all bins have the same X and Y coordinates. The threads iterate through this row and multiply every bin entry by the product of the mentioned functions.

## 5.5 Clustering

Before the actual clustering takes place, a lower threshold for the vertex densities has to be set. Therefore a distribution of vertex density values is created. Based on an adjustable quantil, this threshold is calculated. Since it determines the number of considered bins, this threshold has a big impact on the number and the dimensions of the clusters. If it is chosen very high only a few clusters with a small extent are created. This may contain the risk of ignoring many vertices. When choosing a very low threshold this may lead to many incorrect clusters.
After this calculation the bins with an entry above the threshold are picked and sorted in decreasing order. The bin with the highest entry is used as seed for the first vertex cluster. Then the bin with the second highest entry is picked. If it lies next to the first cluster it will be added to the cluster. If it does not, a new cluster is started. These steps are applied to every bin in the sorted list. When a bin lies next to two different clusters it is also possible that it is added to both of them. Afterwards the different clusters are checked whether they can be distinguished from each other. Therefore the Jackson criterion is used [21]. If the condition 5.1 is true the clusters get merged. The probability to merge the clusters can be regulated by the constant $R$. The smaller it is chosen the more often the clusters are merged.

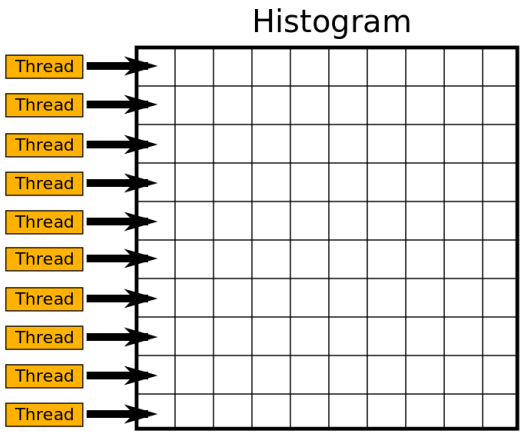



Figure 5.3: Every thread is assigned to a row of the histogram.

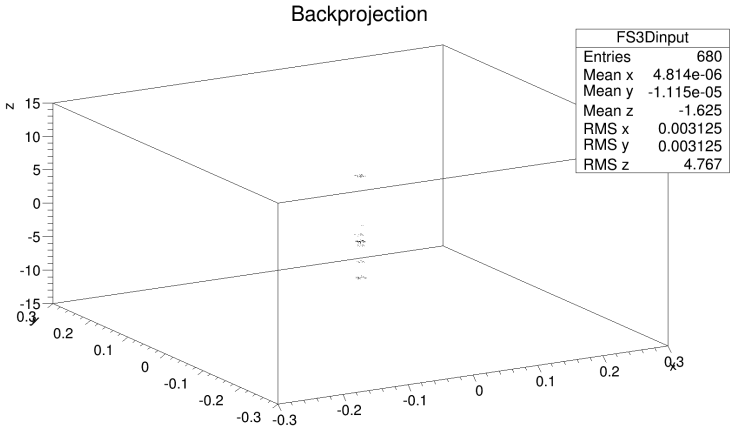



Figure 5.4: This figure shows which pixels are left after clustering. Each bunch of pixels is assigned to a vertex candidate.

When $R$ is chosen 0 all clusters are merged. For $R = 1$ no clusters are merged.

$$\frac{\text{Bin entry of touching bin}}{Min(MaxBin(\text{Cluster1}), MaxBin(\text{Cluster2}))} > R \tag{5.1}$$

This part of the MIVertxing algorithm cannot be implemented easily on the GPU because it is constructed in a consecutive way. For example the clustering process has to start with the bins with the highest entries. A high entry stands for a high vertex density meaning that the probability for the existence of a vertex at this position is very high. Starting with lower bins may cause misguided clusters. So it is not possible to start clustering at many different bins in parallel. An additional problem is that during the clustering every bin gets checked whether it lies next to a cluster. Therefore it is necessary to know at the time of the check how many clusters there are and how large they have gotten. Hence every check depends on previous steps. When executing these checks in parallel, the amount and the structure of the clusters does not stay constant. This behaviour prohibites parallel execution. This problem passes on to the merging procedure. When two clusters get merged, the conditions change for following criterion checks. This may cause completely different vertex candidates at the end.
An additional issue in this context is that CUDA does not support dynamic memory allocation. This means that the amount of required memory has to be known before the execution of a kernel. But since one does not know how many clusters will be there, this condition cannot be fulfilled.
To sum up the clustering can not be executed in parallel. However the histogram still resides in GPU memory thus it is not possible to simply reuse the CPU code. So I rebuilt the clustering algorithm by myself. Therefore it was necessary to omit most of the ROOT functions and to replace them by self-made functions. In these functions I was able to optimise parts of the CPU code and skip unnecessary steps. Nevertheless especially the functions that refer to the histogram cannot be implemented efficiently. Since the CPU executes many single accesses to the histogram, many small data transfers between the GPU and the CPU have to be processed. Such behaviour is very inefficient for the GPU because its advantage is the high computation power and not the tansfer rate to the CPU.

CHAPTER 6

# Comparison of the time-consumption of the CPU and the GPU version of the MIVertexfinder

In this chapter the time-consumption of the CPU version and the GPU version of the MIVertexfinder is shown. The time-consumption of each single procedure is compared as well as the overall time-consumption of both versions. Both versions processed 100 events with the recommended settings of Stephan Hageböck [1] and achieved the same results. The presented numbers are averaged over these 100 events and each of those events contains 100 minbias vertices. The method of measurement and the used settings for gaining these values is shown in section A.1. To grant full comparability the achieved results

| | CPU [ms] | GPU [ms] | Parallelised |
|---|---|---|---|
| Backprojection | 207.0 ± 24.5 | 0.46 ± 0.09 | Yes |
| Conversion | 3042.0 ± 19.6 | 0.00 ± 0.00 | No longer required |
| FFT | 455.9 ± 10.9 | 0.05 ± 0.01 | Yes |
| Filter | 717.1 ± 39.8 | 0.01 ± 0.00 | Yes |
| Inverse FFT | 464.5 ± 11.8 | 0.03 ± 0.00 | Yes |
| Re-conversion | 652.4 ± 40.0 | 0.00 ± 0.00 | No longer required |
| Clustering | 1202.4 ± 12.7 | 816.92 ± 5.88 | No (redesigned) |
| Total | 6738.0 ± 112.6 | 818.72 ± 6.00 | |

Table 6.1: Comparison of the time-consumption of the CPU and the GPU version of the MIVertexfinder.

The procedures that are accelerated by the GPU show huge improvements in terms of speed. When looking at absolute values the most time is saved by skipping the conversions of the three dimensional histogram. This is done by storing the histogram from the beginning in an array of complex numbers. Apart from these great improvements the clustering remains a bottleneck which reduces the overall performance of the GPU version. In total the GPU version of the MIVertexfinder is about a factor 8 faster than the CPU version. The time-consumption of the initialisation of the MIVertexfinder is 77 ms

for the CPU version and 135 ms for the GPU version. These values are not presented in the table because the initialisation is only done once at the beginning of the processing and its contribution is negligible if a large number of events is processed. So the initialisation does not play an important role in the overall performance. However comparing the time-consumption of the initialisation of both versions, it becomes obvious that the GPU version is more expensive in this step. This can be explained by the fact that the initialisation steps are started on the CPU and take much time to get executed by the GPU.

Some of these differences might appear very extreme. This may have several reasons. One reason is related to the data analysis framework ROOT that is used in the CPU implementation. This framework is focused rather on stability and precision than performance in terms of speed. In addition it is to mention these results are achieved by an Intel Core 2 Duo E6750-CPU (see section 3.6). By choosing another CPU the results may change in favor of the CPU. But even if it is possible to reduce the time-consumption of the CPU by a factor of 10 by considering all possible hardware and software improvements, the CPU version still would not be competitive.

CHAPTER 7

# Preparations for b-tagging

In this chapter I will deal with the second part of my master thesis, the adaption of the MIVertexfinder to find b-vertices. I will start with the first step that is already implied in section 2.4, the enhancement of the search volume.

## 7.1 Size of the search volume

One major aspect when extending the MIVertexfinder to a b-tagger is the size and the range of the histogram. For finding primary vertices it suffices that the histogram is focused on the central beamspot. However b-mesons have a mean lifetime of $\tau = 1.6\,\text{ps}$. That enables them to fly out of the beamspot. So the range of the histogram has to be increased. According to section 2.4 b-mesons with a transverse momentum of 50 GeV are able to fly 15 mm. This is the distance the histogram has to cover around the beam axis. So the histogram range in X and Y direction goes from −15 mm to 15 mm.
The range in Z direction is still depending on the size of the beamspot. To cover 99.7 % of the beamspot in this direction, I use the $3\sigma$ range. With $\sigma_{\text{BeamspotZ}} = 4.9\,\text{cm}$ this results in a range from −15 cm to 15 cm. Now it is important to determine the binning. The optimal binning should be in the order of the track resolution.

$$\sigma_{\text{trackXY}} = 43\,\mu\text{m} \tag{7.1}$$

$$\sigma_{\text{trackZ}} = 115\,\mu\text{m} \tag{7.2}$$

Dividing the range of every direction by the particular track resolution gives the binning for the direction. For X and Y direction that results in 576 bins and for Z direction in 2556 bins. This makes 850 million bins in total. The required memory size of such an histogram is 3.2 GB when using 32 bit floating point numbers. Considering the fact that the C2050 GPU has 4 GB of memory space the histogram would fit. But due to the fact that the fourier transformation algorithm requires a multiple of the histogram size, this size is far too large. One solution for this problem could be using a sparse array library. A sparse array is an array in which most of the elements have the value zero. In this case it is only necessary to store the non-zero elements. There are libraries which do exactly that to save memory space. This behaviour comes in very handy for this application since most of the entries in the histogram are empty after the backprojection of the tracks. But the CUFFT library is not compatible to such sparse array formats. This is reasonable since any non-harmonic signal gets transformed into a lot of frequencies.

So there will not be any empty bins left after a fourier transformation. Another solution would be to cut the area that should be examined into slices. These slices can be investigated separately with smaller histograms.
Verifying this resolution can be achieved by decreasing the histogram range and just looking at events that contain vertices within this range.

## 7.2 Adjust the MIVertex finding algorithm

In figure 7.1 there is a backprojection of an event with two b-vertices and one hard primary vertex. The positions of the vertices are:

**Primary vertex** $x = 9.73 \times 10^{-5}$ cm, $y = 5.70 \times 10^{-5}$ cm, $z = 2.28$ cm

**b-vertex 1** $x = -0.54$ cm, $y = -0.30$ cm, $z = -1.34$ cm

**b-vertex 2** $x = 1.17$ cm, $y = 0.75$ cm, $z = -1.43$ cm

It can be seen a b-vertex looks very different compared to a primary vertex. So there arise two main problems. The first problem is the stretched shape of the b-vertices. Due to the boost of the b-hadrons the tracks that belong to a b-vertex are not distributed isotropically around the vertex. This leads to problems caused by the filter function. As mentioned in section 4.5 the filter function points out round structures which occur due to the isotropic distribution of tracks around a vertex. Even if only tracks that are assigned to a b-vertex on truth level are considered, the algorithm does not find b-vertices. Without the application of the filter function and a higher bin size the MIVertexfinder finds b-vertices when still considering only tracks that are assigned to them on truth level. The increased bin size got necessary because the tracks do not cross each other exactly at the vertex position, but rather at points around the vertex. Due to the increased bin size the vertex densities near the vertex are increased and more focused. In addition the required memory space decreases with increasing bin size thus the lack of memory space mentioned in the previous section is no problem anymore.
Considering all tracks instead of just the tracks that are assigned to a b-vertex on truth level, the primary vertex remains a problem. The vertex density at the position of the primary vertex is much higher compared to the vertex density at the positions of the b-vertices. Even in the vicinity of the primary vertex the density is higher or at least at the same level. This difference can be explained by the fact that the average number of tracks belonging to a b-vertex is 5 and for a primary vertex 29 (see figure A.3). So it is very important to remove the primary vertex and its tracks. Hence we know that the b-vertices lie in a jet it is convenient to just consider tracks belonging to a single jet. Since a jet contains tracks that come from primary vertices, I introduced additional cuts on the properties of the tracks. Especially cuts on the track $p_t$ and the impact parameter $d_0$ have a large impact on the result. To reconstruct the jets I used the software package FastJet [22].
A further improvement is obtained by rotating the jet axis into the Z axis. To accomplish this step the position of the primary vertex from which the particular jet originates is necessary. Without the knowledge of this position it is only possible to rotate the jet parallel to the Z axis. In addition it is very convenient to consider only tracks for jet finding that come from one vertex. Hence one can ensure that no additional unwanted tracks interfere. Therefore the tracks have to be associated to the primary vertices before feeding them into the jet finder. This is achieved by using a simple distance criterion. Tracks whose distance to a primary vertex is smaller than 1 mm are associated to the particular vertex. The criterion is not restricted regarding multiple assignments. For determining the position of the primary vertices I use the Z-Finder-Algorithm which is shown in section A.4.

The last step is to cut on the properties of the reconstructed vertices. Due to this step the number of misreconstructed vertices is reduced. The algorithm for the medical imaging b-tagger looks like this:

- Reconstruct primary vertices with Z-Finder-Algorithm
- Associate tracks to primary vertices
  - Maximum distance: 1 mm
- Find jets
- Cut on jet properties
  - Minimum jet $p_T$: 20 GeV
  - Minimum number of tracks per jet : 3
- Cut on track properties
  - Minimum track $p_T$: 1.8 GeV
  - Minimum impact parameter $d_0$ : 105 µm
- Rotate the coordinate system
- Start the MIVertexfinder
- Rotate reconstructed vertices back into the original coordinate system
- Cut on vertex properties
  - Minimum vertex $p_T$: 7 GeV
  - Minimum distance to Z axis: 1.4 mm

## 7.3 Comparison b-tagger

To compare the results of the medical imaging b-tagger I build a b-tagger which is inspired by the ATLAS CSIP b-tagger [23]. CSIP is an abbreviation of counting significant impact parameter tracks. This b-tagger works as follows. The first step is to determine the jets of the particular event. As the name suggests it counts the tracks of a single jet which satisfy certain conditions. These conditions are cuts on the impact parameter $d_0$ and the track $p_T$. If a jet contains enough significant tracks it is tagged as b-jet.
Based on this method I built a vertexfinder. Before reconstructing the jets the primary vertices have to be identified. Then the jets are reconstructed from the tracks that are assigned to a primary vertex. For indentifying the primary vertices the Z-Finder-Algorithm is used (see A.4). The tracks are assigned to the primary vertex by a simple distance criterion. Tracks whose distance to primary vertex is smaller than 1 mm are associated to the particular vertex. There is no restriction for multiple assignments. As already mentioned, the tracks associated to one primary vertex are fed into the jet finder [22]. Jets that have a transverse momentum lower than 10 GeV and consist of less than 3 tracks are rejected. After the proper jets are determined, the tracks of each jet are investigated. Tracks that have a transverse momentum below 0.5 GeV and an impact parameter below 200 µm are rejected. If more than 3 tracks remain and the sum of the transverse momenta of the tracks is above 5 GeV, a vertex $\vec{v}$ is calculated for the investigated jet. Therefore a two-track-vertex $v_{2\vec{\text{track}}}$ between every possible pair of remaining good

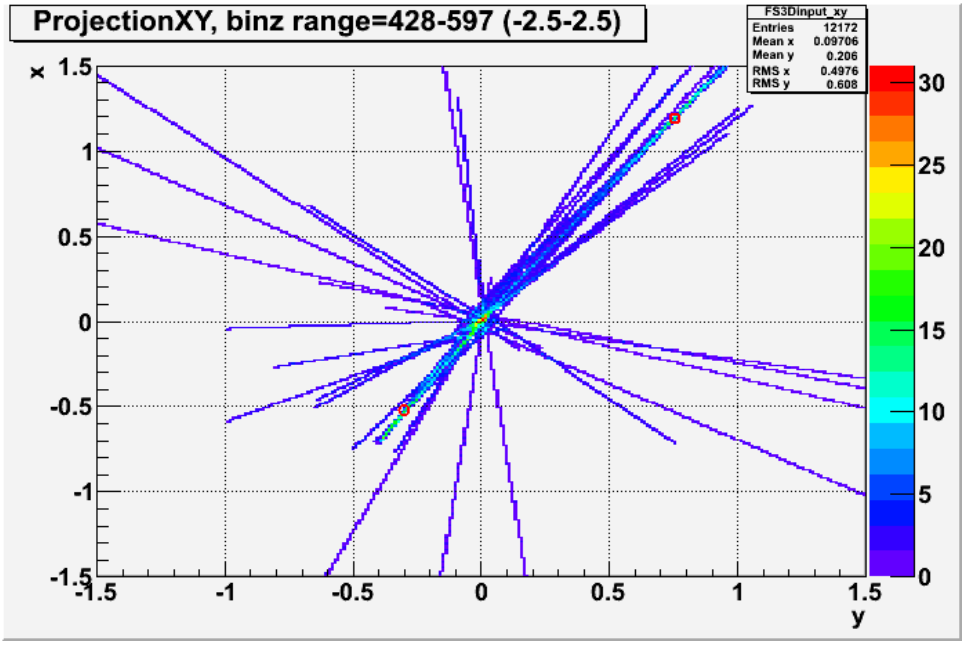



Figure 7.1: This figure shows a XY plane of the three dimensional histogram after backprojection. The spacial measure is cm. The colour stands for the bin entry. A bin entry of two means that two tracks cross. The two red circles mark the position of the two b-vertices and the primary vertex can be spotted in the middle.

tracks is determined. To figure out the position of a two-track-vertex one has to calculate the two points on both tracks where the distance is the smallest. The two-track-vertex lies right in the middle of these points (see figure 7.2). After calculating a two-track-vertex for every possible combination of tracks in one jet, the final vertex for this jet is reconstructed with this equation:

$$\vec{v} = \frac{1}{N_{\text{trackpairs}}} \sum_{i=0}^{N} v_{2\vec{\text{track}},i} \tag{7.3}$$

This vertexfinder is used as reference for the medical imaging b-tagger. The results that are achieved with this CSIP b-tagger are presented in section 8.1 together with the results the medical imaging b-tagger. Listed below is a short summary of this algorithm:

- Indentify primary vertices
- Associate tracks to every primary vertex
  - Maximum distance: 1 mm
- Reconstruct jets from the tracks of each single primary vertex
- Reject bad jets
  - Minimum jet $p_T$: 10 GeV
  - Minimum number of tracks per jet : 3
- Reject bad tracks
  - Minimum track $p_T$: 0.5 GeV

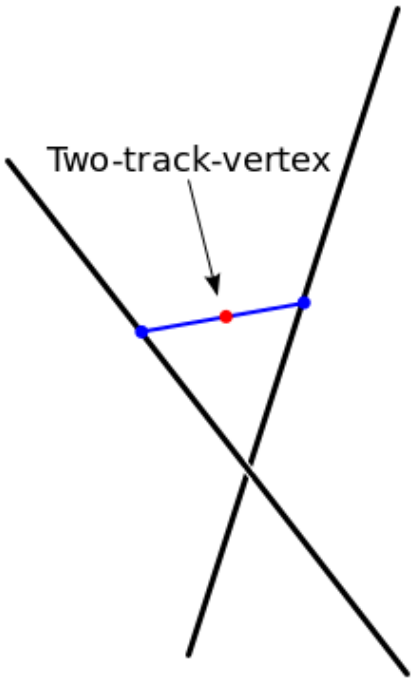



Figure 7.2: The two-track-vertex of two tracks lies in the middle between the closest points of both tracks. The two-track-vertex is marked with a red dot and the closest points of both tracks are marked with blue dots.

  - Minimum impact parameter $d_0$ : 200 µm

- Determine two-track-vertices of every possible track pair
  - Minimum number of remaining tracks: 3
  - Sum of track all $p_T$s has to be bigger than: 5 GeV
- Reconstruct final vertex for every jet

CHAPTER 8

# Results

## 8.1 b-tagging results

This chapter presents the results which are achieved with the medical imaging b-tagger by using the settings presented in section 7.2. The quantities which describe the results are the b-tagging efficiency, charm rejection and the light rejection. As explained in section 2.5 the higher these quantities are the better is the result. To gain the values in table 8.1 the medical imaging b-tagger processed several different samples that are generated with PYTHIA [24]. The samples with zero and five minbias vertices per event contain 10 000 events. The samples with 20 minbias vertices per event contain 5000 events.
Events with 20 minbias vertices correspond to the luminosity that is achieved at the ATLAS detector at a centre-of-mass energy $\sqrt{s} = 8\,\text{TeV}$ [25].
The errors for the light rejection are that high because the medical imaging b-tagger only reconstructs few vertices in the light samples which leads to statistical difficulties. But this fact also highlights the light rejection of the medical imaging b-tagger since it is very important to reject as many light partons as possible, to distinguish real b-vertices in typical ATLAS events.
From the light rejection values one can also recognize that the medical imaging b-tagger is very sensitive to the number of minbias vertices. This behaviour can also be detected in the b-tagging efficiency. Due to this sensitivity the b-tagging efficiency suffers from additional minbias vertices. Whereas the charm rejection remains almost constant. Comparing the charm rejection with the light rejection it becomes obvious that it is more difficult to reject charm vertices than to reject light vertices. This is mainly because the properties of charm hadrons are more similar to those of b-hadrons. For the comparison with

| NminBias | b-Efficiency [%] | Charm Rejection | Light Rejection |
|---|---|---|---|
| 0 | 47.3 ± 0.4 | 12.5 ± 0.3 | 1000 ± 220 |
| 5 | 46.3 ± 0.4 | 13.3 ± 0.3 | 1660 ± 480 |
| 20 | 42.7 ± 0.5 | 14.0 ± 0.5 | 910 ± 270 |

Table 8.1: Medical imaging b-tagger results.

the CSIP inspired b-tagger I tuned the medical imaging b-tagger to similar efficiency values. Table 8.2

shows the results achieved by the CSIP inspired b-tagger compared to those of the medical imaging b-tagger. These results are gained by processing the samples that are mentioned above. The comparison values from the medical imaging b-tagger are also gained from these samples, but to adjust the b-tagging efficiency, the impact parameter $d_0$ cut is increased to 180 µm. This makes the already mentioned statistics problem worse, since the medical imaging b-tagger does not reconstruct any vertices in the light sample. But in principle the light rejections should be higher than before.
By comparing the b-tagging efficiencies it becomes obvious that the CSIP inspired b-tagger is not as sensitive to the number of minbias vertices as the medical imaging b-tagger. But the charm rejection is higher for the medical imaging b-tagger. As already mentioned the light rejection can not be compared properly, but nevertheless since the medical imaging b-tagger rejects all light vertices in the light sample and the CSIP inspired b-tagger does not, the light rejection should be a lot higher for the medical imaging b-tagger.

| Results for medical imaging b-tagger | | | |
|---|---|---|---|
| NminBias | b-Efficiency [%] | Charm Rejection | Light Rejection |
| 0 | 32.2 ± 0.4 | 31.9 ± 1.3 | 1000< |
| 5 | 31.4 ± 0.3 | 33.9 ± 1.4 | 1000< |
| 20 | 27.8 ± 0.5 | 38.3 ± 2.3 | 1000< |
| **Results for CSIP b-tagger** | | | |
| NminBias | b-Efficiency [%] | Charm Rejection | Light Rejection |
| 0 | 33.6 ± 0.4 | 20.9 ± 0.7 | 132 ± 11 |
| 5 | 32.9 ± 0.4 | 21.5 ± 0.7 | 106 ± 8 |
| 20 | 31.2 ± 0.5 | 16.4 ± 0.6 | 47 ± 3 |

Table 8.2: Comparison

## 8.2 Variation of $d_0$ cut

The cut on the impact parameter $d_0$ is the most important cut to distinguish good tracks for b-tagging. In figure 8.1 I varied the cut from 90 µm to 120 µm in steps of 5 µm. To gain this plot I processed samples with 0, 5, 20 and 50 minbias vertices per event. For a higher $d_0$ cut the b-tagging efficiency drops, but the charm rejection rises.
As already noticed in the previous chapter the efficiency of the medical imaging b-tagger is very sensitive to the number of minbias vertices.
From this figure one can also deduce that the working point of the medical imaging b-tagger can be very well adjusted with the impact parameter cut.

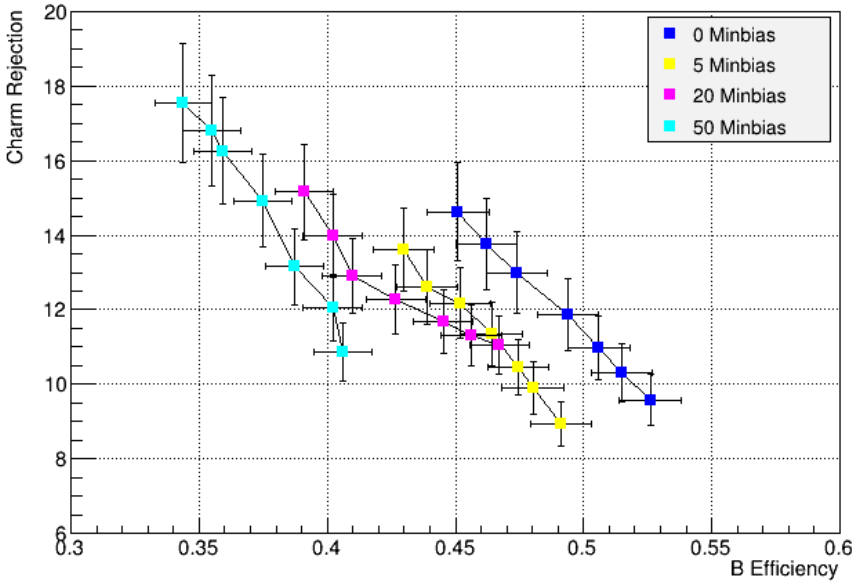



Figure 8.1: b-tagging efficiency plotted against charm rejection

CHAPTER 9

# Summary and Outlook

In my thesis I adapted the medical imaging vertex finder to reconstruct b-vertices. At first the algorithm seemed to be not suitable to find b-vertices since the properties of b-vertices and primary vertices differ a lot. In particular the anisotropic distribution of the tracks around the b-vertices lead to problems for the applied filter techniques. However, due to modifications of the filters and several additional changes, the medical imaging b-tagger achieves good results. Especially the ability to reject light quark flavors is remarkable and leads to advantages in further analysis. The results are even better than the results of comparable b-taggers. To obtain such comparison results I implemented another b-tagger that is inspired by the CSIP b-tagger of the ATLAS group.
The parallel implementation of the medical imaging vertex finder for GPUs was very successful. I got acquainted with the features of GPUs and parallelised the most time-consuming components of the medical imaging finder. The effort lead to improvements in terms of speed of a factor of about 8.

## 9.1 Outlook

During the implementation of the medical imaging vertex finder the problem arose that the clustering algorithm cannot be executed in parallel. Since the other components are accelerated very much the clustering procedure impedes the complete vertex finder. Hence further improvements are necessary for the clustering. Especially a parallelisable clustering algorithm may strongly increase the overall performance of the medical imaging vertex finder.
There is room for additional improvements for the medical imaging b-tagger. As already mentioned, the anisotropic distribution of the tracks around the b-vertices causes problems for the filter techniques that are applied originally in the medical imaging vertex finder. Since this track distribution is caused by the boost of the b-hadrons, avoiding Lorentzboosted kinematics may result in a more isotropic distribution. This can be done by boosting the corresponding jet of a b-vertex into restframe. If an isotropic track distribution can be achieved, it might be beneficial to apply the original filter. These steps may lead to additional improvements in the b-tag efficiency.

APPENDIX A

# Useful information

## A.1 Measurement of the utilisation time

The measured time values in chapter 6 are gained by using the ROOT class TStopwatch. This way of time measurement is necessary because there is no tool available that is able to measure the utilisation of a GPU and a CPU. For example the Valgrind toolsuite is not able to measure the time that is used to execute GPU functions and the profiler tool provided by Nvidia is not able to measure the time that is used to execute CPU functions. Since both tools use different measurement techniques it cannot be granted that the results are comparable.
To ensure that the presented values are comparable I chose the TStopwatch class. This class contains functions that work exactly like a stopwatch. There is a command to start the stopwatch and to stop it. I simply put these commands around the components I wanted to analyse. An additional feature of the TStopwatch class is to read out the CPU time and the realtime that is elapsed. The CPU time describes how long the CPU is utilised for the instructions between the start and the stop command. This time deviates substantially from the realtime if the particular instructions require many memory accesses. However this value is not appropriate for this application since the CPU time is nearly zero during the GPU utilisation. Therefore the times measured in chapter 6 are the realtimes given by TStopwatch.

## A.2 Evaluation criteria

In the evaluation process the true b-vertices are assigned to the reconstructed b-vertices. For this assignment I use two criteria. The first criterion is a loose distance criterion. The distance between the reconstructed b-vertex and the true b-vertex should be smaller than 50 mm. The second criterion refers to $\Delta R$ between the reconstructed b-vertex and the true vertex. This value should be smaller than 0.15. This is chosen on the basis that the reconstructed b-vertex should be located in the center of the jet. If $\Delta R$ between the b-vertex and the true vertex is too large the reconstructed b-vertex might reside outside of the jet, this would be very unlikely.
In addition multiple assignments to one true b-vertex are excluded. That means that every true b-vertex can only be associated to one reconstructed b-vertex, if this one matches above-mentioned criteria.

## A.3 Impact of using straight tracks instead of curved ones

As it is mentioned in section 2.3 charged particles move along bent tracks because of the applied magnetic field. Nevertheless in this thesis tracks are described by straight lines instead of curves. That is because straight lines are easier to handle in different calculations. Since the radius of the bent tracks is very large, these tracks do not deviate much from straight lines. The deviation $d$ that arises after a path of flight $l$ can be estimated by the following equation:

$$d = r - \sqrt{r^2 - l^2} \tag{A.1}$$

The radius $r$ of a bent track is estimated by this equation:

$$r = \frac{p_T}{0.3 \cdot B} \tag{A.2}$$

To calculate the deviation I assume a transverse momentum of 50 GeV. For the path of flight I use $l = 15$ mm. This is the value I calculated in section 2.4. With a magnetic field of $B = 2$ T the resulting deviation is $d = 1.4\,\mu$m.

## A.4 Z-Finder-Algorithm

The Z-Finder algorithm reconstructs primary vertices. This vertexfinder exploits the fact that primary vertices are located in the beamspot whose transversal extent is very small (see section 2.2.4). Since dimensions in X and Y direction are negligible to the dimension in Z direction, two primary vertices with the same Z coordinate are difficult to distinguish. On the other hand they can be distinguished very well in Z direction. Therefore it is convenient to just determine the Z coordinate of a primary vertex and assume the X and Y coordinates as zero. For this purpose a one dimensional histogram which covers

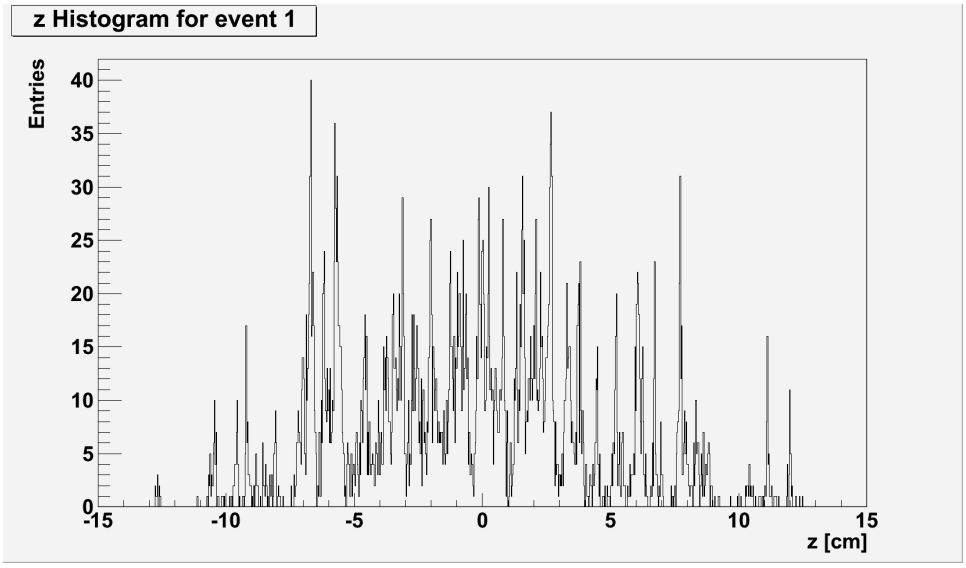



Figure A.1: This histogram shows the vertex density distribution along the Z axis. High peaks represent vertex candidates.

the range of the beamspot in Z direction is required. This histogram stores a vertex density distribution.

The source of this vertex density distribution are the tracks of the event. For every track the point of closest approach (PCA, see 2.3.1) to the Z axis is calculated and the bin entry corresponding to the Z coordinate is increased by one. In that way a region in the Z histogram that is passed by many tracks has a high probability to contain vertices. After the histogram is filled the bins get clustered. Therefore the bin entries are sorted and the highest bin entry seeds the first cluster. The other bins are processed in decreasing order. If a bin lies near to a cluster on the Z axis it is appended, if this is not the case it forms a new seed. Due to statistical fluctuations it is possible that too many clusters are created. These additional clusters result from peaks in the vertex density distribution that are misinterpreted as single vertices although they belong to another cluster. To avoid these mistakes the Jackson resolution criterion is applied to check whether two clusters may belong to a single cluster.

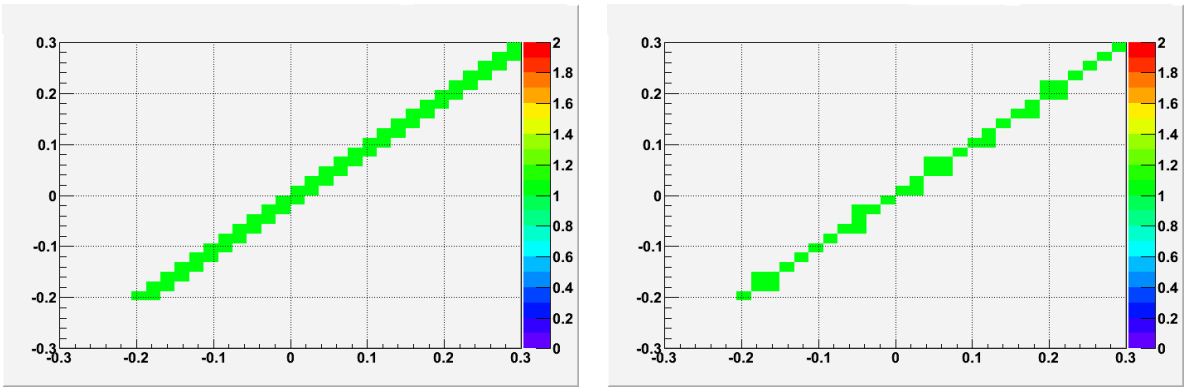
$$\frac{\text{Bin entry of touching bin}}{Min(MaxBin(\text{Cluster1}), MaxBin(\text{Cluster2}))} > R \tag{A.3}$$

The vertex candidates are gathered from the remaining clusters. The positions in the Z direction are determined by the average of the bin centers $\bar{z}$. A step by step summary of this algorithm:

- PCAs of all tracks are calculated.
- A one dimensional histogram is filled according to the Z coordinates of the PCAs.
- The histogram is sorted in decreasing order.
- The bin with the highest vertex density seeds the first cluster.
- Remaining bins seed new clusters or are appended to existing ones.
- Clusters that can not be distinguished from each other are merged.
- From every remaining cluster a vertex candidate at position $(0, 0, \bar{z})$ is created.

## A.5 Precision of the GPU implementation

In this section I want to show examples about some precision issues of the GPU implementation. The figures A.2 show an extract of a backprojection of the same track done by the GPU version and the CPU version. This extract is a single XY plane of a three-dimensional histogram. The vector that describes the direction of this track has the coordinates $(1, 1, 1)$ and the track starts at a corner of four bins in the XY plane. This setting is built such that the track crosses many corners in the XY plane. This is an extreme scenario where the precision plays an important role. As one can see the result of the GPU version differs strongly from the result of the CPU version. Fortunately this scenario shows the worst case for the backprojector since tracks that cross many edges and corners are very rare in typical events. When counting different bins in the histograms of natural events the number of deviating bins is below 0.01 %.
As explained in section 5.4 the function that applies the filter function and the window function multiplies every bin of the histogram by certain values. Therfore the precision of this function depends on the determination of these values. Comparing the precision of the GPU and CPU version the GPU version achieves a precision of $10^{-4}$. That means when these functions are applied to the histogram the differences between the bins are less than $10^{-4}$. Compared to the defined precision of $10^{-7}$ of 32 bit floating point numbers this is less. This can be explained by the impact of the reduced accurancy of compute capability 1.3 when handling square root and division operations (see section 3.4).



(a) CPU version

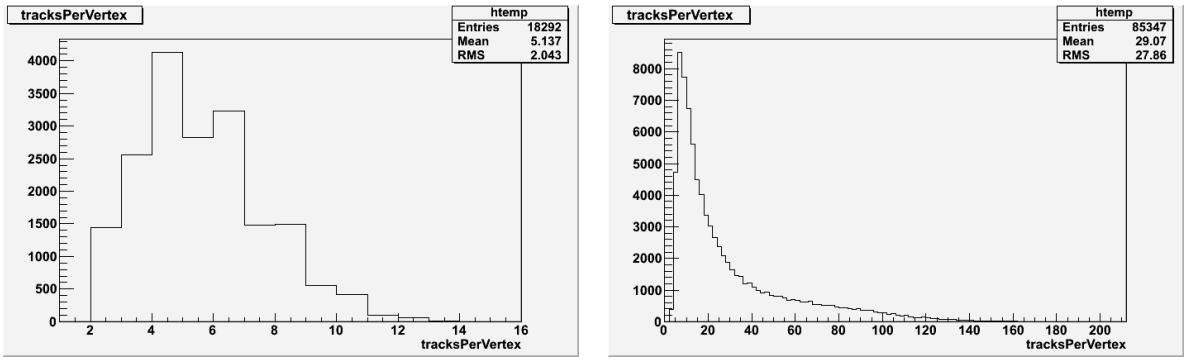
(b) GPU version

Figure A.2: Both pictures show a single XY plane of the backprojection of the same track into a three dimensional histogram. This backprojection is done with the CPU and the GPU.

All in all these differences may add up to shifted vertices or even different numbers of reconstructed vertices. The deviations strongly depend on the number of vertices that are contained by the investigated events. If the events contain only few vertices the CPU version and the GPU version reconstruct the same vertices. When investigating events with 100 minimum bias, 15 percent of the reconstructed vertices deviate in their position about 30 µm. Fortunately these deviations result in very small changes in the reconstruction efficiency which are compatible within the errors.

## A.6 Different properties of primary vertices and b-vertices

The figrues A.3 show distributions of the number of tracks per vertex for b-vertices and primary vertices. The mean value of number of tracks per vertex for b-vertices is 5 and for primary vertices 29. This explains the large differences in vertex density that are mentioned in section 7.2. Figure A.4 shows a




(a) Tracks per vertex for b-vertices

(b) Tracks per vertex for primary vertices

Figure A.3: Comparison of the number of tracks per vertex for primary vertices and b-vertices.

distribution of the vertex mass of true b-vertices. The vertex mass is calculated on the basis of tracks that are assigned to the true b-vertex on truth level. Although the mass of a b-hadron is $m \approx 5.3\,\text{GeV}$ the mean mass value of the distribution is 2.8 GeV. By this example one recognise the impact of neutrinos.

Since they do not leave tracks in the detector, there is missing information for the determination of the mass of a vertex.

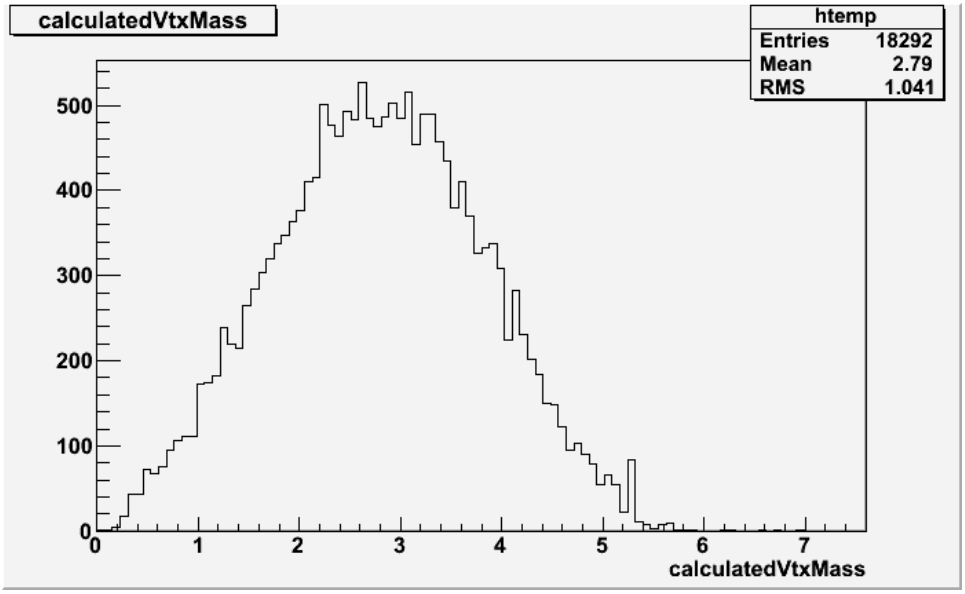



Figure A.4: Mass of a b-hadron reconstructed on the basis of tracks on truth level.

# Bibliography


[1] S. Hageböck,
"Anwendung von medizinischen Bildgebungsverfahren auf die Vertexrekonstruction am LHC",
BONN-IB-2012-07, Diploma Thesis: University of Bonn, 2012.

[2] Wikipedia, *Standard model of elementary particles*, URL: http://en.wikipedia.org/wiki/File:Standard_Model_of_Elementary_Particles.svg.

[3] ATLAS Experiment ©2013 CERN, *The four main LHC experiments.*
URL: http://www.atlas.ch/photos/atlas_photos/selected-photos/lhc/9906026_01_layout_sch.jpg.

[4] G. Aad et. al., "Expected performance of the ATLAS experiment: detector, trigger and physics" (2009).

[5] ATLAS Experiment ©2013 CERN, *Computer generated image of the whole ATLAS detector*,
URL: http://cds.cern.ch/record/1095924.

[6] P. F. Åkesson et. al., "ATLAS Tracking Event Data Model", ATL-SOFT-PUB-2006-004, ATL-COM-SOFT-2006-005, CERN-ATL-COM-SOFT-2006-005 (2006).

[7] T. G. Cornelissen, *Track fitting in the ATLAS experiment*, 2006, ISBN: 9789064640513.

[8] V. M. Abazov et al. D0 Collaboration, *B-Jet Identification*,
URL: http://www-d0.fnal.gov/Run2Physics/top/singletop_observation/.

[9] Particle Data Group, K. Nakamura et al., "Review of Particle Physics",
*J. Phys. G* 37 (2010) 075021, URL: http://pdg.lbl.gov.

[10] W. Andermahr, *GeForce GTX 280 ohne Kühler*,
URL: http://pics.computerbase.de/2/1/8/1/8/163.jpg.

[11] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, vol. 4.2, Apr. 2012.

[12] G. Marcus, "Comparison of CUDA and OpenCL", Graphics Processing Units (GPUs) in High Energy Physics Workshop, Deutsches Elektronen-Synchrotron DESY, Hamburg, Apr. 2013.

[13] J. v. Oosten, *CUDA Memory Model*,
URL: http://3dgep.com/wp-content/uploads/2011/11/CUDA-memory-model.gif.

[14] ATLAS Experiment ©2013 CERN, *Collision Event at 7 TeV with 2 Pile up Vertices.*
URL: http://atlas.web.cern.ch/Atlas/public/EVTDISPLAY/events.html.

[15] J. G. Colsher, "Fully-three-dimensional positron emission tomography",
*Physics in Medicine and Biology*, 25.1, 1980 p.103.

[16] D. M. Defrise and R. Clack,
"Three-dimensional image reconstruction from complete projections",
*Physics in Medicine and Biology*, 34.5, 1989 p.573.

[17] Wikipedia, *Leck-Effekt*, URL: `http://de.wikipedia.org/wiki/Leck-Effekt`.

[18] R. N. Nethercote and J. Fitzhardinge,
"Building Workload Characterization Tools with Valgrind", IEEE International Symposium on Workload Characterization (IISWC 2006), San Jose, California, USA, Oct. 2006.

[19] R. Brun and F. Rademakers, "ROOT - An Object Oriented Data Analysis Framework",
*Proceedings AIHENP'96 Workshop* (Sept. 1996), Nucl. Inst. and Meth. in Phys. Res. A 389 (1997) 81–86.

[20] M. Frigo and S. Johnson, "The Design and Implementation of FFTW3",
*Proceedings of the IEEE* 93.2 (2005), Special issue on "Program Generation, Optimization, and Platform Adaptation" 216–231.

[21] D. J. Jackson, "A Topological Vertex Reconstruction Algorithm for Hadronic Jets",
*Nucl. Inst. and Meth.* A 388 (1997) 247–253.

[22] G. M. Cacciari and G. Soyez, "FastJet user manual", Eur. Phys. J. C **72** (2012) 1896,
arXiv:`1111.6097[hep-ph]`.

[23] V. et. al. D0 Collaboration, "b-Jet identification in the D0 experiment",
*Nuclear Instruments and Methods in Physics Research* 620 (2010) 490–517.

[24] S. T. Sjöstrand and P. Z. Skands, "PYTHIA 6.4 Physics and Manual", *JHEP 05* S.026 (2006),
hep-ph/: `0603175`.

[25] E. v. Törne, "Overview of the ATLAS Higgs Measurements",
Talk at BW 2013 – Beyond the Standard Models, Vrnjačka Banja, Serbia, Apr. 2013.

# List of Figures


2.1 Particles of the Standard Model. . . . 3
2.2 Large Hadron Collider. . . . 5
2.3 ATLAS detector. . . . 7
2.4 Track parameters. . . . 8
2.5 Illustration of a secondary vertex. . . . 10

3.1 Graphics card. . . . 13
3.2 Performance comparison of different platforms. . . . 14
3.3 Grid consisting of blocks . . . . 15
3.4 CUDA memory model . . . . 17
3.5 Streaming multiprocessor . . . . 18
3.6 Scalability of a CUDA program . . . . 19

4.1 Comparison between a PET-image and a collision event with two primary vertices. . . 22
4.2 One event backprojected into a 3D histogram. . . . 23
4.3 The result of a backprojection is a star-shaped artefact instead of the round original dot [1]. 24
4.4 Both figures show the same XY plane of the three dimensional histogram, but at different processing steps. The spacial measure is cm. . . . 26
4.5 Spectral leakage of a limited signal. . . . 26
4.6 Spectral leakage of a discrete signal. . . . 27
4.7 Profiling results. . . . 29

5.1 Sketch of the backprojecting procedure . . . . 32
5.2 CUFFT performance. . . . 33
5.3 Applying filters. . . . 34
5.4 Histogram after clustering . . . . 34

7.1 B-vertex after backprojection. . . . 42
7.2 Two-track-vertex. . . . 43

8.1 b-tagging efficiency plotted against charm rejection. . . . 47

A.1 Typical vertex density distribution along the Z axis. . . . 52
A.2 Both pictures show a single XY plane of the backprojection of the same track into a three dimensional histogram. This backprojection is done with the CPU and the GPU. . 54
A.3 Comparison of the number of tracks per vertex for primary vertices and b-vertices. . . 54
A.4 Reconstructed mass of a b-hadron. . . . 55

# List of Tables

6.1 Comparison of the time-consumption of the CPU and the GPU version of the MIVertexfinder. . . . . 37

8.1 Medical imaging b-tagger results. . . . . 45
8.2 Comparison . . . . . 46