RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

David Rohr

ALICE TPC Online Tracking on GPU based on Kalman Filter

Diplomarbeit

HD-KIP-10-37

KIRCHHOFF-INSTITUT FÜR PHYSIK

Department of Physics and Astronomy

University of Heidelberg

Diploma thesis

in Physics

submitted by

David Rohr

born in Mannheim

2010

ALICE TPC Online Tracking on GPU based on Kalman Filter

This diploma thesis has been carried out by David Rohr

at the

Kirchhoff Institute of Physics

under the supervision of

Prof. Dr. Volker Lindenstruth

Abstract

ALICE TPC Online Tracking on GPU based on Kalman Filter

For online analysis in the ALICE HLT a new, fast TPC tracker was developed. This thesis presents an adaptation of it to run on graphics cards using the NVIDIA CUDA framework. As the former tracker was already well able to deal with proton-proton events, this work is focused primarily on heavy-ion events the previous tracker was not able to handle efficiently. The implementation of the algorithm had to be adjusted at many points to allow for an efficient usage of the GPU. In particular, achieving a good overall workload for many processor cores, efficient transfer to and from the GPU, as well as optimized utilization of the different memories the GPU offers turned out to be critical. To cope with these problems a dynamic scheduler was introduced, which redistributes the workload among the processor cores. Additionally a pipeline was implemented so that the tracking on the GPU, the initialization and the output processed by the CPU, as well as the DMA transfer can overlap. Besides the algorithmic implementation, the integration within AliRoot and the HLT framework is discussed, which are the standard analysis and trigger frameworks for the ALICE experiment.

ALICE TPC Online-Spurrekonstruktion auf Grafikkarten basierend auf dem Kalman Filter

Ein neuer, schneller TPC Tracker wurde für den ALICE HLT entwickelt. In dieser Arbeit wird eine Portierung jenes Trackers auf das NVIDIA CUDA Framework vorgestellt. Der bisherige Tracker war sehr wohl in der Lage, das Tracking für Proton-Proton Kollisionen effizient durchzuführen, weshalb im folgenden der Schwerpunkt auf Schwerionenkollisionen liegt, die zu Begin der Arbeit ein Problem darstellten. Die Implementierung des Tracking-Algorithmuses wurde in vielerlei Hinsicht angepasst, um eine effiziente Nutzung der Grafikkarte zu gewährleisten. Die größten Herausforderungen bestanden insbesondere in der vollständigen Auslastung der vielen unabhängigen Rechenkerne, einem effizienten Datentransfer von und zu der GPU sowie dem geeigneten Einsatz der verschiedenen heterogenen Speichersysteme, welche die Grafikkarte bereitstellt. Um diesen Problemen Herr zu werden wurde ein dynamischer Scheduler eingeführt, der die Last unter den vielen Prozessorkernen umverteilen kann. Zusätzlich wurde der Tracker um eine Pipeline erweitert, so dass das eigentliche Tracking auf der GPU, die Initialisierung sowie Weiterverarbeitung auf der CPU und der DMA Transfer gleichzeitig ablaufen können. Neben dem Tracking Algorithmus selbst wird die Integration in die AliRoot sowie HLT Frameworks erläutert, welche die Standarmittel für Ereignisrekostruktion, Analyse sowie Trigger bereitstellen.

Contents

Chapter 1

Introduction

1.1 The Experiment

1.1.1 Large Hadron Collider (LHC)

The Large Hadron Collider (Fig. 1.1) is a proton-proton (pp) and heavy-ion collider that started operating in November 2009. It is located at CERN (Conseil Européen pour la Recherche Nucléaire) in Geneva. The LHC is a ring collider placed in the tunnel of the former LEP (Large Electron Positron Collider) 50 to 175 meters below the surface, with a length of almost 27 km. Two beams of protons or heavy-ions (lead) are accelerated in opposing directions to a total energy of up to 14 TeV for protons and about 1150 TeV for lead. Thus the LHC supersedes Tevatron at Fermilab, which has been the most powerful collider yet with a maximum energy of 1.96 TeV. Contrary to Tevatron, the LHC is a symmetric accelerator colliding protons with protons instead of antiprotons. When fully operational, the proton beams are supposed to consist of $2,808$ bunches of about $1.15 \cdot 10^{11}$ protons circulating at a frequency of 11 kHz. Compared to Tevatron one major advantage of the LHC is its high luminosity of 10^{34} cm^{-2}s^{-1}. For more information see [Cer1] and [Cer2].

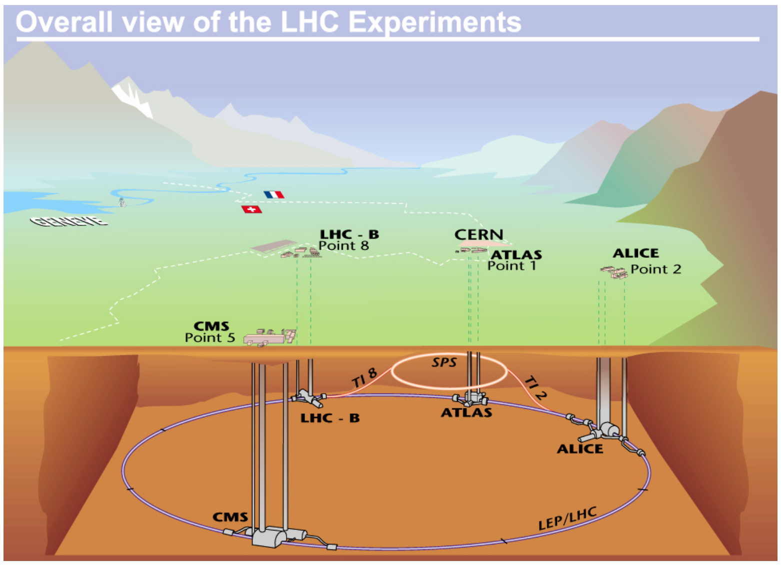


Figure 1.1: LHC Overview [Cer3]

There are four primary detectors (ALICE, ATLAS, CMS, LHCb) and some minor ones. ATLAS and CMS are general purpose detectors. LHCb is aimed at analyzing CP-Violation in the interaction of B-Mesons. ALICE's main objective is the observation of lead-lead collisions and the Quark-Gluon-Plasma, which is expected to be created in heavy-ion events.

1.1.2 A Large Ion Collider Experiment (ALICE)

ALICE (Fig. 1.2) is mainly aimed at the observation of lead-lead collisions. The ALICE sub-detectors can be split into two parts: firstly, the L3 magnet with the detectors ITS, TPC, TRD, TOF, PHOS, and HMPID, and secondly, the dipole magnet with the forward detectors ZDC, FMD, and PMD (see [Ali1]). Even though lead-lead collisions will only occur with reduced frequency, they will create many more particles. ALICE's main detector for tracking (see 1.1.5) is the TPC. Since the tracking algorithm described later is intended for tracking TPC data, the TPC will be described in more detail.

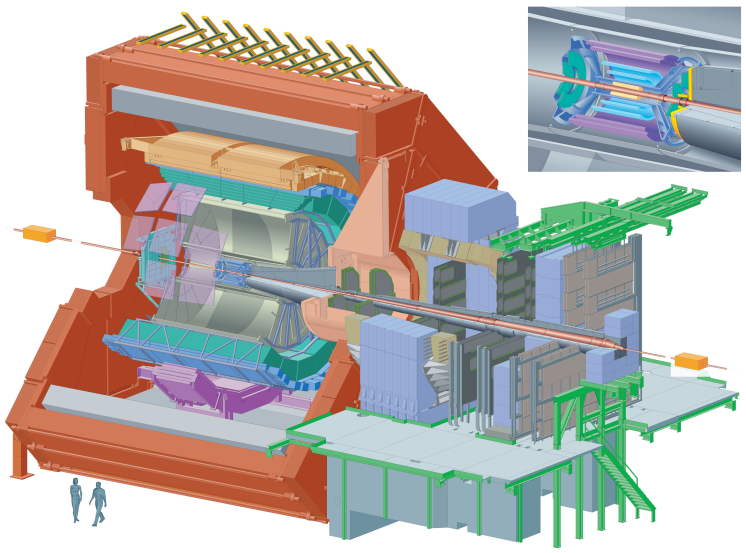


Figure 1.2: The ALICE Detector [Ali2]

1.1.3 ALICE Time Projection Chamber (TPC)

In general, a TPC is a cylindrical chamber filled with gas and divided in two halves as illustrated in Fig. 1.3. The z-axis (beam direction) is directed along the cylinder. A high voltage electrode disc is located in the center, establishing an electric field to both endplates. Charged particles passing through the TPC ionize gas molecules. The ions produced are accelerated by the electric field towards the end plates. The x- and y-coordinates of the ionized particles can be determined by measuring where the ions hit the plate, while the z-coordinate can be calculated by analyzing the drift time. Both halves of the TPC are further separated into 18 trapezoidal sectors, which will be referred to as **slices** hereafter. On the endplates there are 159 detectors (called **rows**) in radial directions which measure the angular position continuously. In each slice the local coordinate system is such that the x-axis will be radial. (The coordinate system differs for different slices.) Thus the TPC will deliver discrete x-coordinates (159 rows) and continuous y- and z-coordinates of the locations where the gas molecules were ionized by a particle. These space-points are called **clusters**. Since the TPC was designed to study heavy-ion collisions, it was not specifically built to operate at very high frequencies. Currently its readout frequency is several hundred Hz and is planed to rise to 1 kHz[1] [Lar]. Instead, the TPC must be able to track the huge number of particles produced in lead-lead collisions. See [Ali3] for more information on the TPC.

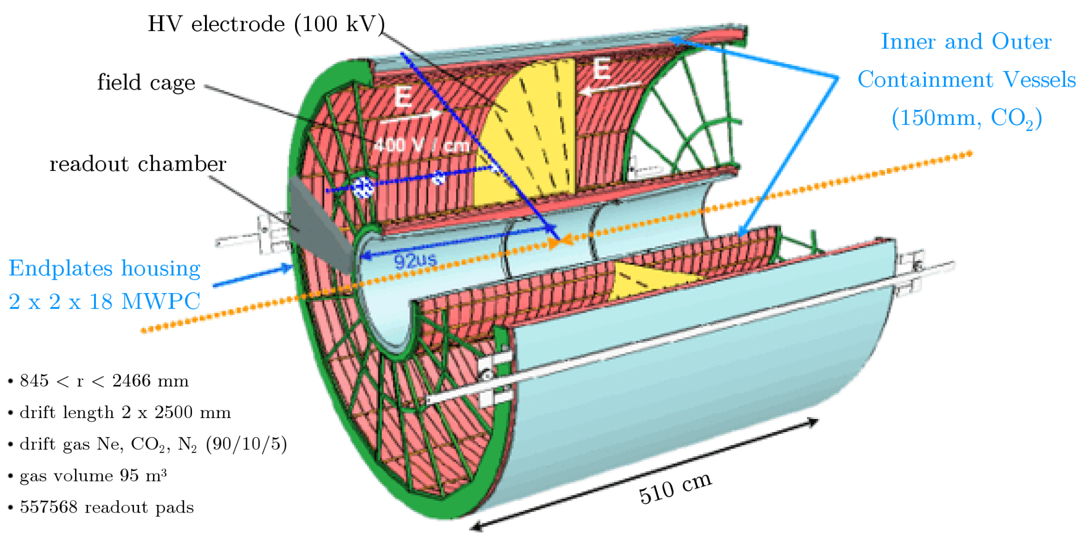


Figure 1.3: ALICE Time Projection Chamber [Ali3]

1.1.4 ALICE High Level Trigger (HLT)

The pp and even the heavy-ion interaction rates are much higher than the maximum TPC readout frequency and in addition the data volume read out by the TPC is way too high (more than a hundred megabytes per heavy-ion event) to be stored on any storage system. For these reasons the TPC cannot be read out after every single collision. Thus hardware triggers are present which analyze for every event whether a TPC readout is necessary or not. These triggers are categorized in L0 to L2 triggers, with L0 triggers reacting after 1.2 µs and L1 triggers after 6 µs. The longer the L1 trigger takes, the more active volume of the TPC is lost due to the drift of the ions. 6 µs is defined as limit, whereupon the L1 triggers such as the TRD [Ret] have to deliver the trigger decision whether a TPC readout should be done. This ensures that the TPC is only read out after interesting events. To cope with the excess amount of data, a software based High Level Trigger (in contrast to hardware L0 to L2 triggers) reduces the data volume further. This HLT currently consists of a compute farm of 120 frontend and 60 compute nodes[2] and is capable of performing an online event reconstruction processing a 30 GB/s input data stream. A set of triggers can then be applied to the result to decide whether to discard the event or to store the whole event or only parts of it. In this way the physically relevant data can be extracted. See [Hlt] for an HLT overview.

1.1.5 Track Reconstruction

Online event reconstruction is required for a precise trigger decision and includes the reconstruction of the particle trajectories, which are called **tracks**. Tracking therefore is the

[1]Proton-proton collision rate is much higher, than heavy-ion rate. The Atlas experiment, for example, is designed for a frequency of multiple kHz.
[2]The design allows for up to 1,000 nodes to deal with increased luminosity later.

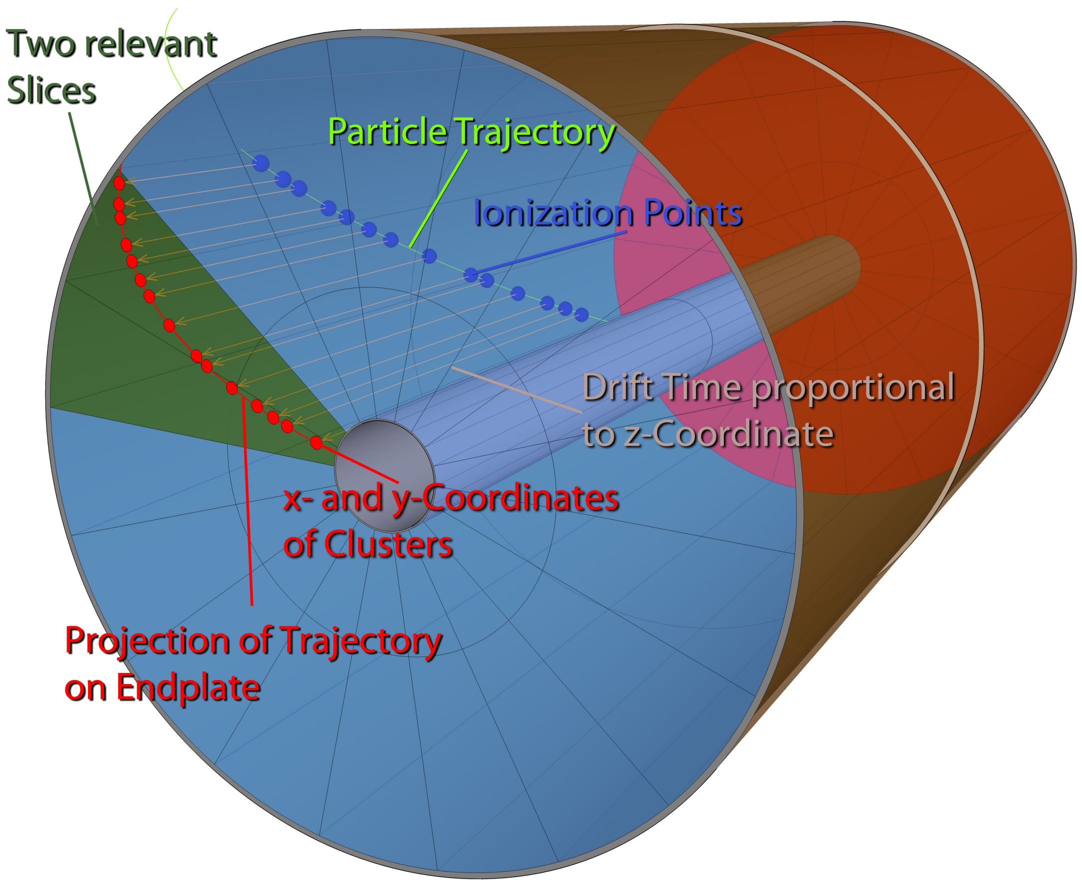

Two relevant Slices
Particle Trajectory
Ionization Points
Drift Time proportional to z-Coordinate
x- and y-Coordinates of Clusters
Projection of Trajectory on Endplate

Figure 1.4: Illustration of the Working Principle of a Time Projection Chamber

process of retrieving the trajectories of particles that passed through the detector. ALICE's primary detector for this purpose is the TPC.

The TPC output is digitized and then processed by an FPGA-based hardware cluster finder [Hlt] developed by Torsten Alt. The cluster finder creates a set of three-dimensional space-points where gas molecules where ionized. It then remains a combinatorial challenge to relate these space-points to particle trajectories. Fig. 1.5 shows the first pp event in ALICE which was tracked online by the HLT and rendered by the online event display.

1.1.5.1 Event display

For the analysis of different track reconstruction algorithms many internal non-essential parameters and intermediate results have to be observed that are not needed for the official ALICE online event display. Therefore, the official display does not offer support for visualizing them. An OpenGL based special event display, which can benefit from hardware accelerated graphics cards, was thus developed for this thesis. The visualization of a pp event can be seen in Fig. 1.6. Blue dots are ionization points measured by the TPC while green lines are the tracks interconnecting them reconstructed by the tracking algorithm. Clearly tracking gets extensively more complex for an increasing amount of clusters when switching from pp to heavy-ion collisions. Figures 1.7 and 1.9 show the clusters for a peripheral and a central heavy-ion collision respectively, Figures 1.8 and 1.10 show the tracks recovered from the input clusters by the tracker.

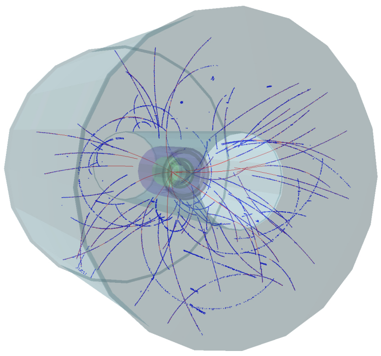


Figure 1.5: Real TPC Event in ALICE Online Event Display

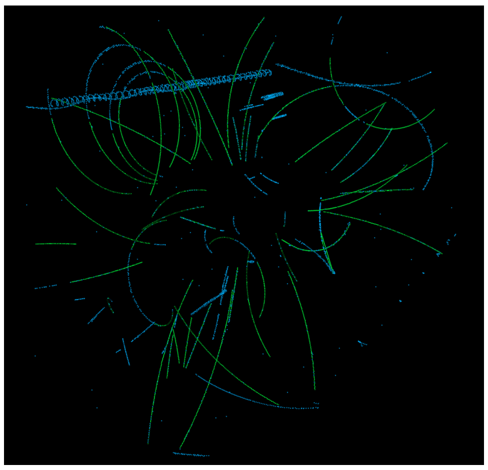


Figure 1.6: Clusters and Tracks of Simulated PP Event

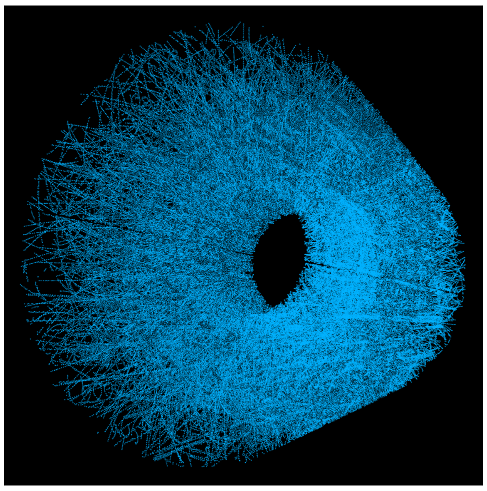


Figure 1.7: Clusters of Simulated Noncentral Pb-Pb Event

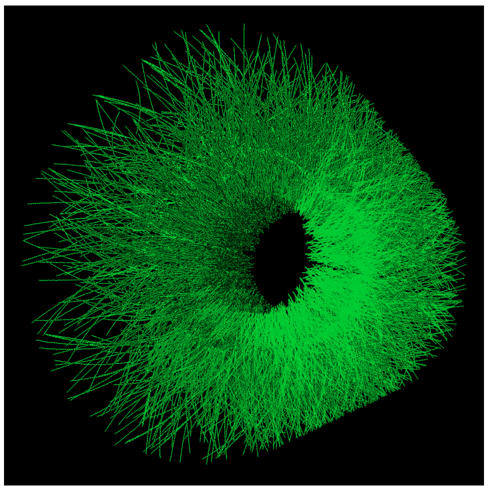


Figure 1.8: Tracks Found by GPU Tracker in Event

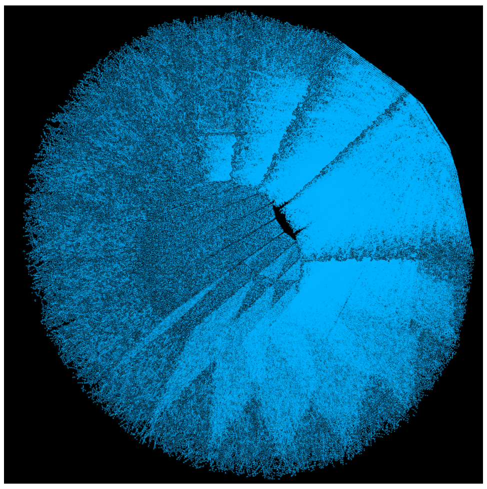


Figure 1.9: Clusters of Simulated Central Pb-Pb Event

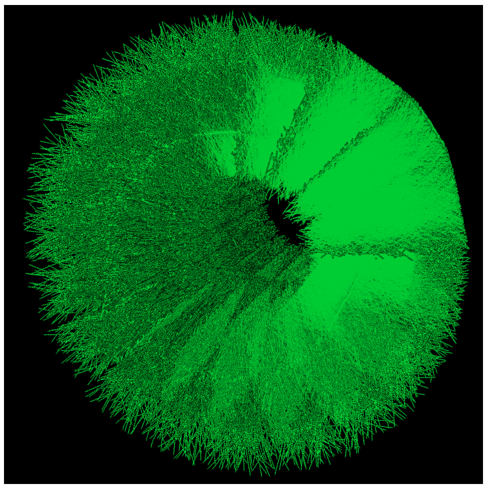


Figure 1.10: Tracks Found by GPU Tracker in Event

1.2 Physical Background

There are two aspects that greatly influenced the design of the LHC, namely the search for the Higgs-Boson and the investigation of the so called Quark-Gluon-Plasma. These phenomena will now be briefly described.

1.2.1 Standard Model of Particle Physics

The Standard Model of Particle Physics is a theory describing the elementary particles separated in quarks and leptons, three out of four fundamental interactions (the electro-magnetic, strong and weak interaction, but not gravitation) and their force meditating bosons. It explains the phenomena occurring in the world very well and also predicted new physics beyond that level such as the existence of the $W^{\pm}$ and Z bosons, whose existence was later proven in experiments.

The Standard Model is a very elastic theory, with 18 external parameters that need to be measured in experiments. Unfortunately, it falls short of being complete, since it does not include gravitation as well as dark matter and is unable to describe phenomena such as baryon asymmetry or nonzero neutrino masses.

The Standard Model is a gauge theory based on continuous groups, so called Lie Groups, where $SU(3)$ is the symmetry group of the strong interaction, and $SU(2) \times U(1)$ the group for the electroweak interaction. The mathematical framework is provided by quantum field theory.

1.2.2 The Higgs-Particle

The Higgs-Mechanism describes how the particles acquire mass. Consider the Lagrangian

$$\mathbf{L}_{scalar} = (\partial_\mu \Phi^\dagger)(\partial^\mu \Phi) + \mu(\Phi^\dagger \Phi) - \lambda(\Phi^\dagger \Phi)^2 \tag{1.1}$$

whose potential can be seen in Fig. 1.11 in the complex one-dimensional case. The ground state is degenerated. The vacuum expectation value (VEV) can be calculated to $v = |\langle \Phi_0 \rangle| = \sqrt{\frac{\mu}{2\lambda}}$, where Φ_0 can be chosen real because of the gauge freedom (unitary gauge). As a result only $U(1)$ is left as symmetry group for the vacuum. This process is called spontaneous symmetry breaking. After the insertion of the covariant derivative

$$\partial_\mu \to D_\mu = \partial_\mu - igT_a A^a_\mu - ig'YB_\mu \tag{1.2}$$

and the VEV into the kinetic term $(\partial_\mu \Phi^\dagger)(\partial^\mu \Phi)$ of the Lagrangian (1.1) the masses of the gauge fields can be obtained. By defining[3]

$$\begin{aligned} \vartheta &:= \arctan\left(\frac{g'}{g}\right) \\ W^{\pm}_\mu &:= \frac{1}{\sqrt{2}}(A^1_\mu \mp iA^2_\mu) \\ Z_\mu &:= \cos(\vartheta)A^3_\mu - \sin(\vartheta)B_\mu \\ A_\mu &:= \sin(\vartheta)A^3_\mu + \cos(\vartheta)B_\mu \end{aligned} \tag{1.3}$$

[3] ϑ is called the Weinberg angle.

the mass terms calculate to

$$-g^2\frac{v^2}{4}W^{+\mu}W^-_\mu - \frac{1}{2}\frac{g^2+g'^2}{g'^2}\frac{v^2}{4}Z^\mu Z_\mu = -M_W^2 W^{+\mu}W^-_\mu - \frac{1}{2}M_Z^2 Z^\mu Z_\mu. \tag{1.4}$$

The masses of the W and Z bosons were measured to be 80.4 and 91.2 GeV[4] respectively, while the photon remains massless as no $A^\mu A_\mu$ term appears. Obviously, there is an unbroken $U(1)$ subgroup[5] which is the gauge group of electromagnetism. Furthermore, a massive Spin 0 Higgs-Boson appears with the mass

$$m_H = \sqrt{2\mu} \tag{1.5}$$

More elaborate introductions can be found in [Pes+], [Sre], and [Pen]. The value of μ is not determined by the theory, and so the mass of the Higgs-Boson cannot be predicted. Experiments yet can only give a lower bound. By additional theoretical considerations the Higgs-Particles's mass can be restricted to an interval which is completely covered by the LHC center of mass energy by design.

1.2.3 Quark-Gluon-Plasma

The Quark-Gluon-Plasma is a state of matter without quark confinement where quarks and gluons are rather completely dissolved and free (in analogy to conventional plasma, in which the electrons are unbound). Both very high temperatures and very high baryon densities can result in this state. It is expected that the baryon density in neutron stars is sufficient as is the temperature of the universe about 1 µs after the Big Bang. Reaching such a state in an experiment is a great challenge and can only be realized by colliding ultra relativistic heavy-ions. It is assumed that a Quark-Gluon-Plasma will be created in lead-lead collisions at CERN and ALICE has been built specifically to examine such events.

Fig. 1.12 shows a state diagram, with the transition point where the confinement is canceled, and with the expected state for LHC lead-lead collisions.

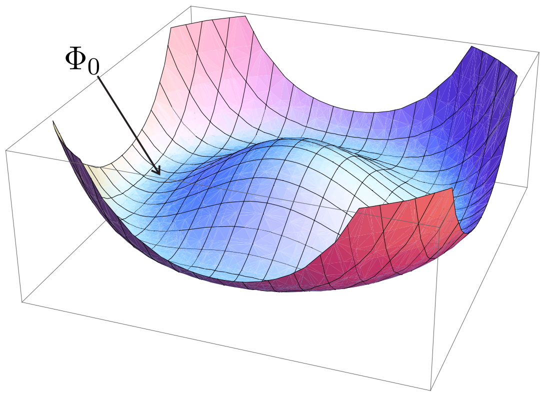


Figure 1.11: $V(\Phi) = -\mu(\Phi^\dagger\Phi) + \lambda(\Phi^\dagger\Phi)^2$

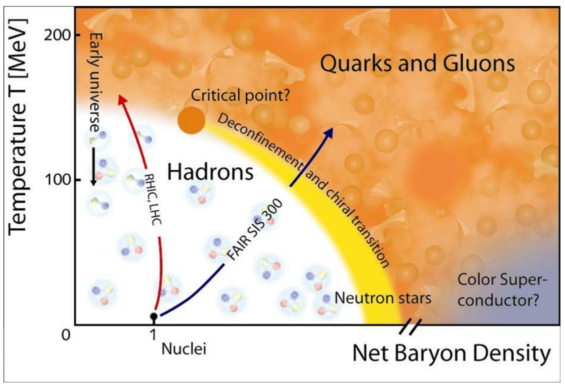


Figure 1.12: Quark-Gluon-Plasma [Cbm]

[4] $W^\pm$ and Z masses were precisely measured using the LHC predecessor LEP.

[5] This is not the projection to the $U(1)$ component of $SU(2)\times U(1)$ but a subgroup that is isomorphic to $U(1)$.

Chapter 2

Tracking

2.1 CA Tracking Algorithm (Theory)

In this section the theory of a Cellular Automaton and the Kalman filter will be presented. A combination of them will be used in the tracking algorithm.

2.1.1 Cellular Automaton

A Cellular Automaton is a model of a spatial and temporal discrete dynamic system, in which the state of a cell at the time $t + 1$ depends only on the states of cells in a predetermined neighborhood at the previous time t.[1] Because of this locality, Cellular Automata are highly suited for parallel approaches. Thinking of a computer program for such a model, every cell can be calculated by an independent thread, and on top of that the data exchange between the threads is restricted to those threads processing adjacent cells. The last aspect renders these automata well suited for cluster computers. In our application, however, the first aspect will be the more important one.[2]

2.1.2 Fitting with Least Squares Estimator

In the following $\mathbf{X}^T$ denotes the transpose of the vector $\mathbf{X}$, $< \mathbf{X} >$ the mathematical expectation value of $\mathbf{X}$ and $cov(\mathbf{X}) =< (\mathbf{X}- < \mathbf{X} >)(\mathbf{X}- < \mathbf{X} >)^T >$ the covariance matrix. Let $\mathbf{m}_k$ be measurements that linearly depend on state vectors $\mathbf{r}^t_k$. (Only k is an index here. The symbol t will later distinguish between the state vector $\mathbf{r}^t_k$ and the estimator $\mathbf{r}_k$.) The state vector is explicitly allowed to be different for distinct measurements, but it is required that the state $\mathbf{r}^t_{k+1}$ linearly depends on the previous state $\mathbf{r}^t_k$. The measurement error is denoted by η_k, the process noise by ν_k. All errors shall have a gaussian distribution. This yields the formulas:

$$\begin{aligned} \mathbf{r}^t_k &= A_k \mathbf{r}^t_{k-1} + \nu_k \\ \mathbf{m}_k &= H_k \mathbf{r}^t_k + \eta_k \end{aligned} \tag{2.1}$$

A_k and H_k are matrices. The measurement errors are assumed to be unbiased with their covariance matrices known. Errors of different measurements shall be uncorrelated. The

[1]The classical example is John Conway's Game of Life.

[2]Tracker processes on different nodes in the cluster process distinct data sets. Therefore, communication is not required.

same is assumed for the process noise:

$$\begin{aligned} &< \eta_k \eta_l^T > = 0 \qquad && < \nu_k \nu_l^T > = 0 \\ &< \eta_k > = 0 && < \nu_k > = 0 \\ &cov(\eta_k) \equiv V_k && cov(\nu_k) \equiv Q_k \end{aligned} \tag{2.2}$$

An estimator $\mathbf{r}_k$ is a set of estimates for the unknown real state vector $\mathbf{r}_k^t$ based on the measurements $\mathbf{m}_l$ (all measurements can be used for finding an estimator $\mathbf{r}_k$). The error ϵ_k of an estimator is defined as the difference to the real state vector and its covariance matrix is denoted by C_k.

$$\epsilon_k = \mathbf{r}_k^t - \mathbf{r}_k \qquad C_k = cov(\epsilon_k) \tag{2.3}$$

The estimator is called linear if it depends linearly on the measurement and unbiased if

$$< \epsilon_k >= 0. \tag{2.4}$$

The mean squared error σ_k^2 of an estimator $\mathbf{r}_k$ is defined by

$$\sigma_k^2 =< ||\epsilon_k||^2 > . \tag{2.5}$$

The fit problem can now be defined as determining a linear unbiased estimator $\mathbf{r}_k$ for the last state vector $\mathbf{r}_k^t$ with minimal mean squared error.

The solution of this fit problem will be used for the track fit later (track parameters will build the state vector). It is very well possible that additional clusters will be added to a track a set of fitted parameters already exists for. A method is required by which new measurements can easily be integrated into the existing estimator without having to start the calculation all over again. One general way to obtain an optimal estimator is the Least Squares Method described in [Fru+]. Equations for this method are comparably simple. However, a new measurement cannot be easily used to improve the current estimator, but the whole calculation must be repeated, involving all previous measurements. Additionally multiple scattering cannot easily be handled in this way. The Least Squares Method is therefore not suited. This leads to the Kalman filter, which can be shown to find a best estimator, too.

2.1.3 Kalman Filter

The Kalman filter works iteratively. It processes the measurements subsequently, thereby continuously improving the estimator that after the last step will be the best linear unbiased estimator searched for. The Kalman filter contains the following steps:

- **Initialization**: The state vector $\mathbf{r}_0$ is initialized with a guessed or even arbitrary value and the covariance matrix C_0 filled with huge numbers corresponding to the uncertainty of the initial state vector.

- **Extrapolation**: Before the measurement $\mathbf{m}_k$ can be used to calculate the estimator $\mathbf{r}_k$, the last estimator $\mathbf{r}_{k-1}$ and its covariance matrix are extrapolated to the current state.

$$\begin{aligned} \tilde{\mathbf{r}}_k &= A_k \mathbf{r}_{k-1} \\ \tilde{C}_k &= A_k C_{k-1} A_k^T + Q_k \end{aligned} \tag{2.6}$$

- **Filtering**: In the filter step the measurement information is collected, which in combination with the extrapolated estimator and the covariance matrix delivers the next estimator.

$$\begin{aligned}
K_k &= \tilde{C}_k H_k^T (V_k + H_k \tilde{C}_k H_k^T)^{-1} \\
\zeta_k &= (\mathbf{m}_k - H_k \tilde{\mathbf{r}}_k) \\
\mathbf{r}_k &= \tilde{\mathbf{r}}_k + K_k \zeta_k \\
C_k &= \tilde{C}_k - K_k H_k \tilde{C}_k \\
\chi_k^2 &= \chi_{k-1}^2 + \zeta_k^T (V_k + H_k \tilde{C}_k H_k^T)^{-1} \zeta_k
\end{aligned} \tag{2.7}$$

After the initialization, the extrapolation and filter steps are performed for every measurement, eventually delivering the desired best estimator. In [Gor1], some extensions to the Kalman filter are introduced which are also used in the tracking algorithm. (See also [Fru$^+$] and [Man] for more applications and examples.) Because the Kalman filter is the backbone of the tracking algorithm, a proof is sketched. (A more comprehensive elaboration can be found in [Kal].)

The notation is changed slightly for the proof and some simplifications are assumed. From now on $\mathbf{r}_k^t$ is seen as a gaussian distributed random variable. It is assumed that $\eta_k = 0$, so 2.1 becomes $\mathbf{m_k} = H_k \mathbf{r}_k^t$. The measurement error is assumed to be encoded in the gaussian distributed random variable $\mathbf{r}_k^t$. The objective is to find a linear estimator $\mathbf{r}_k^*$ for the $(k+1)^{\text{th}}$ step. ϵ_k is thus defined as $\epsilon_k = \mathbf{r}_{k+1}^t - \mathbf{r}_k^*$ and is to be minimized. This means one searches the linear estimator $\mathbf{r}_k^*$ such that $< \epsilon_k^2 >$ is minimal. ($\mathbf{r}_k^*$ is the best estimator for the current state vector when setting $A_{k+1} = Id$)

First some results from probability theory are needed.

Definition 1 *The conditional expectation value $< \mathbf{X} | \mathbf{Y} = \mathbf{Y}_0 >$ is the expectation value of the random variable $\mathbf{X}$, with the value of the random variable $\mathbf{Y}$ known to be $\mathbf{Y}_0$.*

$< \mathbf{X} | \mathbf{Y} = \mathbf{Y}_0 >$ is a function of $\mathbf{Y}_0$ and in that way it is a random variable itself (denoted by $< \mathbf{X} | \mathbf{Y} >$). Remember that the objective is to find the random variable $\mathbf{r}_k^*$ which is as a function of $\mathbf{m}_0$, $\mathbf{m}_1$, ..., $\mathbf{m}_{k-1}$ such that the average error $< \mathbf{r}_k - \mathbf{r}_k^t >^2$ is minimal.

Theorem 2 *Assume the above situation. Then the random variable $\mathbf{r}_k^*$ as function of $\mathbf{m}_i$ (with $1 \leq i \leq k$) minimizing the average error is given by the conditional expectation value*

$$\mathbf{r}_k^* = < \mathbf{r}_{k+1}^t | \mathbf{m}_1, \mathbf{m}_2, ..., \mathbf{m}_k > \tag{2.8}$$

Proof: See [Kal] and [She].

Definition 3 *The vector space $\mathcal{Y}_k$ is the closed subspace created by the components $\mathbf{m}_i^j$ of the random variables $\mathbf{m}_i$, $(1 \leq i \leq k)$, as a subspace of the vector space of all random variables $\mathcal{Y}$.*

$$\mathcal{Y}_k = \Big\{ \sum_{i,j} a_{i,j} \cdot \mathbf{m}_i^j \qquad a_{i,j} \in \mathbb{R} \Big\} \tag{2.9}$$

On $\mathcal{Y}$ a symmetric bilinear form is given by $b(\mathbf{x}, \mathbf{y}) = < \mathbf{xy} >$, which defines a scalar product on the vector space of all random variables with nonzero variance, and thus also on $\mathcal{Y}_k$ (as physical measurements naturally have nonzero variance). As $\mathcal{Y}_k$ is a closed subset of a Hilbert space, the projection onto it exists. Let Π_k denote the projection onto $\mathcal{Y}_k$.

Theorem 4 *Let $\mathbf{x}_k$, $\mathbf{y}_k$ be random processes with $<\mathbf{x}>=<\mathbf{y}>=0$. The optimal linear estimator in the above manner $\mathbf{x}_k^*$ for $\mathbf{x}_{k+1}$ given $\mathbf{y}_1$, $\mathbf{y}_2$, ..., $\mathbf{y}_k$ is given by the projection onto $\mathcal{Y}_k$:*

$$\mathbf{x}_k^* = \Pi_k(\mathbf{x}_{k+1}) \tag{2.10}$$

Proof: See [Kal].

For the remaining proof, it is assumed that theorem 2 holds also for $\mathbf{x}_k = \mathbf{r}_k^t$ and $\mathbf{y}_k = \mathbf{m}_k$. At this point all the prerequisites for the proof are available.

Theorem 5 *In the given situation the Kalman filter algorithm delivers the best linear estimator.*

Proof: Assume by induction that the best estimator $\mathbf{r}_{k-1}^*$ is known. Let $\mathcal{Z}_k$ denote the orthocomplement of $\mathcal{Y}_{k-1}$ in $\mathcal{Y}_k$ such that:

$$\mathcal{Y}_k = \mathcal{Y}_{k-1} \oplus \mathcal{Z}_k. \tag{2.11}$$

Consider that for physically relevant measurements $\mathbf{m}_k$ the dimension of $\mathcal{Z}_k$ is positive, otherwise the k^{th} measurement would contain no information at all. Let now Φ_k denote the projection onto $\mathcal{Z}_k$, i.e. $\Pi_k = \Pi_{k-1} + \Phi_k$. $\mathbf{r}_k^*$ can be calculated to:

$$\begin{aligned}\mathbf{r}_k^* &= \Pi_k(\mathbf{r}_{k+1}^t)\\ &= \Pi_{k-1}(\mathbf{r}_{k+1}^t) + \Phi_k(\mathbf{r}_{k+1}^t)\\ &= A_{k+1}\mathbf{r}_{k-1}^* + \Pi_{k-1}(\nu_{k+1}) + \Phi_k(\mathbf{r}_{k+1}^t)\end{aligned} \tag{2.12}$$

Define $\tilde{\mathbf{m}}_k = \Phi_k(\mathbf{m}_k)$ and $\bar{\mathbf{m}}_k = \Pi_k(\mathbf{m}_k)$. It follows that the last term in 2.12 is a linear operation on $\tilde{\mathbf{m}}_k$

$$\Phi_k(\mathbf{r}_{k+1}^t) = \triangle_k^* \tilde{\mathbf{m}}_k \tag{2.13}$$

with some matrix $\triangle_k^*$. Because of

$$\tilde{\mathbf{m}}_k = \mathbf{m}_k - \bar{\mathbf{m}}_k = \mathbf{m}_k - H_k \mathbf{r}_{k-1}^* \tag{2.14}$$

it follows that

$$\mathbf{r}_k^* = \kappa_k^* \mathbf{r}_{k-1}^* + \triangle_k^* \mathbf{m}_k \tag{2.15}$$

with

$$\kappa_k^* = A_{k+1} - \triangle_k^* H_k. \tag{2.16}$$

Thus the best linear estimator linearly depends only on the best estimator for the previous step and the last measurement. Now consider the error of the estimator:

$$\begin{aligned}\epsilon_{\mathbf{k}} &= \mathbf{r}_{k+1}^t - \mathbf{r}_k^*\\ &= A_{k+1}\mathbf{r}_k^t + \nu_{k+1} - \kappa_k^* \mathbf{r}_{k-1}^* - \triangle_k^* H_k \mathbf{r}_k^t\\ &= \kappa_k^* \epsilon_{k-1} + \nu_{k+1}\end{aligned} \tag{2.17}$$

Using this and stochastic independence of ν_k with $\mathbf{r}_k^t$ and therefore with ϵ_k, the covariance matrix P_k of ϵ_k can be calculated.

$$\begin{aligned} P_k &=< (\epsilon_k - < \epsilon_k >)(\epsilon_k - < \epsilon_k >)^T >=< \epsilon_k \epsilon_k^T > \\ &= \kappa_k^* < \epsilon_{k-1}\epsilon_{k-1}^T > \kappa_k^{*T} + cov(\nu_{k+1}) \\ &= \kappa_k^* P_{k-1} \kappa_k^{*T} + Q_{k+1} \\ &= \kappa_k^* P_{k-1} A_{k+1}^T + Q_{k+1} \end{aligned} \tag{2.18}$$

Now an explicit formula for $\triangle_k^*$ will be obtained: Using

$$\begin{aligned} \Phi_k(\mathbf{m}_k) \perp \mathbf{r}_{k+1}^t - \Phi_k(\mathbf{r}_{k+1}^t) \\ = \mathbf{r}_{k+1}^t - \triangle_k^* \Phi_k(\mathbf{m}_k) \end{aligned} \tag{2.19}$$

and 2.13 the following equation holds:

$$\begin{aligned} 0 &=< (\mathbf{r}_{k+1}^t - \triangle_k^* \Phi_k(\mathbf{m}_k))\Phi_k(\mathbf{m}_k)^T > \\ &=< (\mathbf{r}_{k+1}^t \Phi_k(\mathbf{m}_k)^T > - \triangle_k^* < \Phi_k(\mathbf{m}_k)\Phi_k(\mathbf{m}_k)^T > \\ &=< \Phi_k(\mathbf{r}_{k+1}^t)\Phi_k(\mathbf{m}_k)^T > - \triangle_k^* H_k P_k H_k^T && \text{,as } \Pi_k(\mathbf{r}_{k+1}^t) \perp \Phi_k(\mathbf{m}_k) \\ &=< (A_{k+1}\Phi_k(\mathbf{r}_k^t + \nu_{k+1})\Phi_k(\mathbf{r}_k^t)^T H_k^T > - \triangle_k^* H_k P_{k-1} H_k^T \\ &= A_{k+1} P_{k-1} H_k^T - \triangle_k^* H_k P_{k-1} H_k^T && \text{,as } \nu_{k+1} \text{ and } \mathbf{r}_k^t \text{ are independent} \end{aligned} \tag{2.20}$$

This can be solved for $\triangle_k^*$ if H_k has full rank and P_{k-1} is positive definite. The latter can usually be assumed for a covariance matrix.

$$\triangle_k^* = A_{k+1} P_{k-1} H_k^T (H_k P_{k-1} H_k^T)^{-1} \tag{2.21}$$

This directly delivers an exact formula for κ_k^* as well. By plugging everything in, the formulas in 2.6 and 2.7 can be deduced. This is done for the covariance matrix exemplary. It is important to note that during the proof the best estimator for the next step is considered, so the index is shifted compared to the formulas 2.6 and 2.7. For the covariance matrix this means: $P_k = \tilde{C}_{k+1}$

Equation for $\tilde{C}_{k+1}$ in the Kalman filter:

$$\begin{aligned} \tilde{C}_{k+1} &= A_{k+1} C_k A_{k+1}^T + Q_{k+1} \\ &= A_{k+1}(\tilde{C}_k - K_k H_k C_k) A_{k+1}^T + Q_{k+1} \\ &= A_{k+1}(\tilde{C}_k - \tilde{C}_k H_k^T (V_k + H_k \tilde{C}_k H_k^T)^{-1} H_k \tilde{C}_k) A_{k+1}^T + Q_{k+1} \end{aligned} \tag{2.22}$$

$$\tag{2.23}$$

Equation in the proof for P_k:

$$\begin{aligned} P_k &= \kappa_k^* P_{k-1} A_{k+1}^T + Q_{k+1} \\ &= (A_{k+1} - \triangle_k^* H_k) P_{k-1} A_{k+1}^T + Q_{k+1} \\ &= A_{k+1} P_{k-1} A_{k+1}^T - A_{k+1} P_{k-1} H_k^T (H_k P_{k-1} H_k^T)^{-1} H_k P_{k-1} A_{k+1}^T + Q_{k+1} \\ &= A_{k+1}(P_{k-1} - P_{k-1} H_k^T (H_k P_{k-1} H_k^T)^{-1} H_k P_{k-1}) A_{k+1}^T + Q_{k+1} \end{aligned} \tag{2.24}$$

Under the assumption that $\eta_k = 0$ and thus $V_k = 0$ this shows the equivalence. The calculation for $\mathbf{r}_{k+1}$ and $\mathbf{r}_k^*$ is analogous. □

In fact, the basis for the induction was not handled during the proof. In order to actually carry out the iterations an initial estimator $\mathbf{r}_0^*$ and the covariance matrix P_0 must be known. The algorithm solves this by choosing an arbitrary start value, and setting the covariance entries to infinity. This resembles the fact that prior to the first measurement no information is available at all.

2.1.4 Track Model

As in most collider experiments, the solenoid magnetic field in ALICE is oriented along the z-axis. At first, in this situation, the trajectory of a particle of charge q, mass m and momentum $\mathbf{p}$ will be described. Only the magnetic field will be considered, but not energy loss or scattering. In the algorithm these effects are handled by treating noise in the Kalman filter. Clearly the z-component of the momentum will be constant. Let $P_t = \sqrt{p_x^2 + p_y^2}$ be the transversal momentum. Considering only the magnetic field and ignoring energy loss and scattering by equating the Lorentz and centripetal force

$$\mathbf{f} = m\mathbf{a} = q\mathbf{B} \times \frac{\mathbf{p}}{m} = -m\omega^2\mathbf{r} \tag{2.25}$$

it can bee seen that the trajectory projected to the x,y plane is a circle of radius $R = \frac{P_t}{q} \cdot \frac{1}{B_z}$. With $\mathbf{r}_0$ the center of the circle the trajectory can be described as

$$\mathbf{r}(t) = \mathbf{r}_0 + \begin{pmatrix} R \cdot \cos\omega(t - t_0) + \vartheta_0 \\ R \cdot \sin\omega(t - t_0) + \vartheta_0 \\ \lambda(t - t_0) \end{pmatrix} \tag{2.26}$$

Such a trajectory is called a helix. The above description is well suited for visualization. However, the parameters are redundant and not well applicable for the track fit. A set of parameters that directly includes the measured values is desirable. From now on a different set of parameters is used:

1. $Y = r_{0,y} + R \cdot sin(\omega(t - t_0) + \vartheta_0)$
2. $Z = r_{0,z} + \lambda(t - t_0)$
3. $\sin(\varphi) = \cos(\vartheta)$
4. $\lambda = \frac{dz}{ds} = \frac{p_z}{|\mathbf{p}|}$
5. $\kappa = \frac{q}{P_t} = \frac{B_z}{R}$

Obviously X (the x-coordinate) is not a track parameter in this model, but the X position is part of the model itself (instead of t). For every X position this set of parameters can describe the trajectory, with different sets of parameters describing the same trajectory at different X. As the measurements are taken at different X-positions, the possible to propagate the track parameters (state vector) to a new X-coordinate is desirable.[3] This is also called extrapolation. Clearly parameters 3 and 4 are not affected by this. The transformation of a state vector $\mathbf{r}^t = (Y, Z, \sin(\varphi), \lambda, \kappa)^T$ at x_0 to the state vector $\widetilde{\mathbf{r}^t} = (\widetilde{Y}, \widetilde{Z}, \sin(\widetilde{\varphi}), \widetilde{\lambda}, \widetilde{\kappa})^T$ at $x = x_0 + \triangle x$ is the following:

$$\begin{aligned} \sin(\widetilde{\varphi}) &= \sin(\varphi) + \triangle x \cdot B_z \cdot \kappa \\ \widetilde{Y} &= Y + \cos(\widetilde{\varphi}) - \cos(\varphi) = Y + \triangle x \cdot \tan\left(\frac{\varphi + \widetilde{\varphi}}{2}\right) = Y + \triangle x \cdot \frac{\sin(\varphi) + \sin(\widetilde{\varphi})}{\cos(\varphi) + \cos(\widetilde{\varphi})} \\ \widetilde{Z} &= Z + \lambda \cdot \underbrace{2\left(\kappa B_z\right)^{-1} \arcsin\left(\frac{1}{2}\kappa B_z \frac{\triangle x}{\cos(\varphi + \widetilde{\varphi})}\right)}_{\text{ds}} \end{aligned} \tag{2.27}$$

Different equivalent formulas can be obtained using trigonometric addition theorems. Unfortunately, the extrapolation for Y and Z is nonetheless not linear, as it would be required for the Kalman filter. Fig. 2.1 shows the helix and all relevant parameters.

[3]Transporting the track parameters is not necessarily required. Alternatively the measurement can be transported. However, the propagation of the track parameters is also used in other parts of the algorithm.

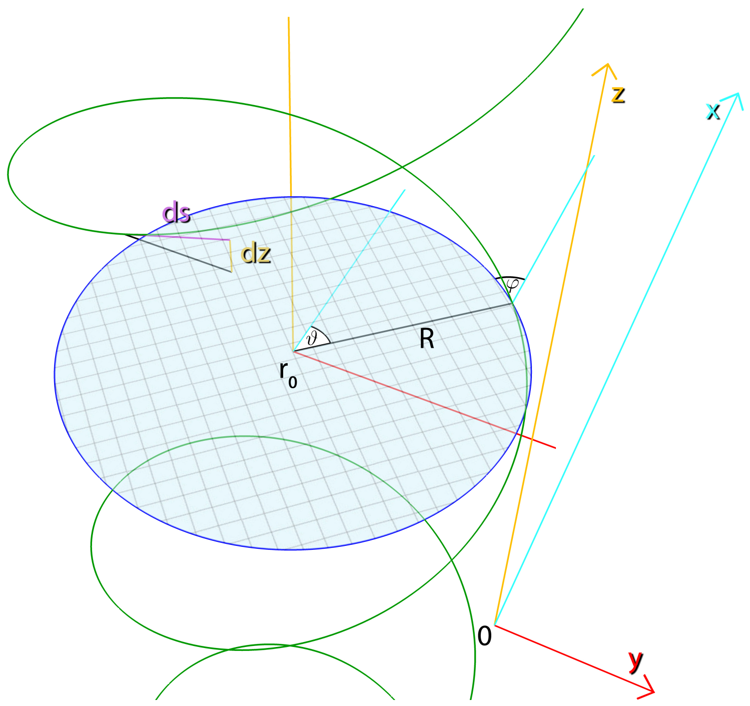


Figure 2.1: Illustration of Track Helix and Track Model

2.1.5 Linearization

Nonlinear propagation can be handled by linearizing the extrapolation function denoted by F_x hereafter, with $\widetilde{\mathbf{r}} = F_x(\mathbf{r})$. It has to be noted that F_x is considered as function of $\mathbf{r}$ where for every x a different function exists. The linearization is done in terms of the track parameters but not x, which is part of the model. Applying the linearization the linear extrapolation looks like

$$\widetilde{\mathbf{r}} \approx F_{lin}(\mathbf{r}) = F_x(\mathbf{r}_0) + \partial F_x\Big|_{\mathbf{r}_0}(\mathbf{r} - \mathbf{r}_0) \tag{2.28}$$

with $\partial F_x|_{\mathbf{r}_0}$ the Jacobian matrix of F_x at $\mathbf{r}_0$. An appropriate linearization point $\mathbf{r}_0$ must be chosen. In fact, for the exact point the linearized function matches the exact extrapolation. Usually the currently best estimator is used as linearization point. The algorithm then must be iterated where the best estimators from the previous iterations are subsequently used as linearization points. This is described in more detail in [Gor1].

2.1.6 The Algorithm

The tracking algorithm works as follows: Using the Cellular Automaton principle, **seeds** are created which are sets of connected clusters locally forming straight lines. The clusters in such a seed are then fitted with the Kalman filter. For the extrapolation step the functions in 2.27 are linearized. In the filter step it is assumed that there is no correlation between Y and Z. Then the following equations[4] hold, with y and z the measurement values and σ_y, σ_z the errors respectively.

[4] $C_k^{i,j}$ denotes the i, j^{th} entry of the covariance matrix, which is the covariance between the track parameters i and j.

$$
\begin{aligned}
Y_k &= \widetilde{Y}_k + \frac{C_k^{0,0}}{\sigma_y^2 + C_k^{0,0}}(y - \widetilde{Y}_k) \\
Z_k &= \widetilde{Z}_k + \frac{C_k^{1,1}}{\sigma_z^2 + C_k^{1,1}}(z - \widetilde{Z}_k) \\
\sin(\phi_k) &= \widetilde{\sin}(\phi_k) + \frac{C_k^{2,0}}{\sigma_y^2 + C_k^{0,0}}(y - \widetilde{Y}_k) \\
\lambda_k &= \widetilde{\lambda_k} + \frac{C_k^{3,1}}{\sigma_z^2 + C_k^{1,1}}(z - \widetilde{Z}_k) \\
\kappa_k &= \widetilde{\kappa_k} + \frac{C_k^{4,0}}{\sigma_y^2 + C_k^{0,0}}(y - \widetilde{Y}_k)
\end{aligned}
\tag{2.29}
$$

After the clusters in the initial seed are fitted, the formulas from the extrapolation step are used to propagate the trajectory to adjacent rows (defined by a different X) and finding new clusters close to the extrapolated position. These clusters will then be added to the track in additional Kalman filter steps. The procedure is described in the next section in detail.

2.2 Implementation and Data Structures

2.2.1 Slice Trackers and Merger

The TPC is separated into 36 slices which are further split into 6 readout chambers. Readout and cluster[5] finding is done separately for every chamber. To allow for a more parallel design and to avoid network bottlenecks, the clusters for each slice are tracked separately using a slice tracker. The track segments for all slices are later combined in the track merger. In this way it is not necessary to centralize all cluster data onto one compute node possibly exceeding its network bandwidth or computing capabilities. Fig. 2.2 shows a heavy-ion event split in slices.

2.2.2 Slice Tracker

The slice tracker algorithm is implemented in five sequential steps. Steps one to three represent the combinatorial part that searches for seeds. Step four does the track fit and extrapolation by means of the Kalman filter. Step five assigns clusters to the final tracks, resolving collisions where two tracks cross and a cluster could be interpreted to belong to both of them. This last task has to be an independent step succeeding the Kalman filter to allow for a parallel implementation.

In a conventional tracker algorithm the track fit would be a sequential algorithm not allowing for a parallel processing of multiple tracks. The algorithm would start with the longest seed e.g. as it contains most information about the track. This seed would then be fitted and extended by additional clusters using extrapolation. The clusters would be marked as used and would not be available for other tracks anymore. Only afterwards the tracker

[5]In the TPC tracker the words “cluster” and “hit” are used synonymously. This document is restricted to the word cluster, but the reader should be aware of this when reading other references.

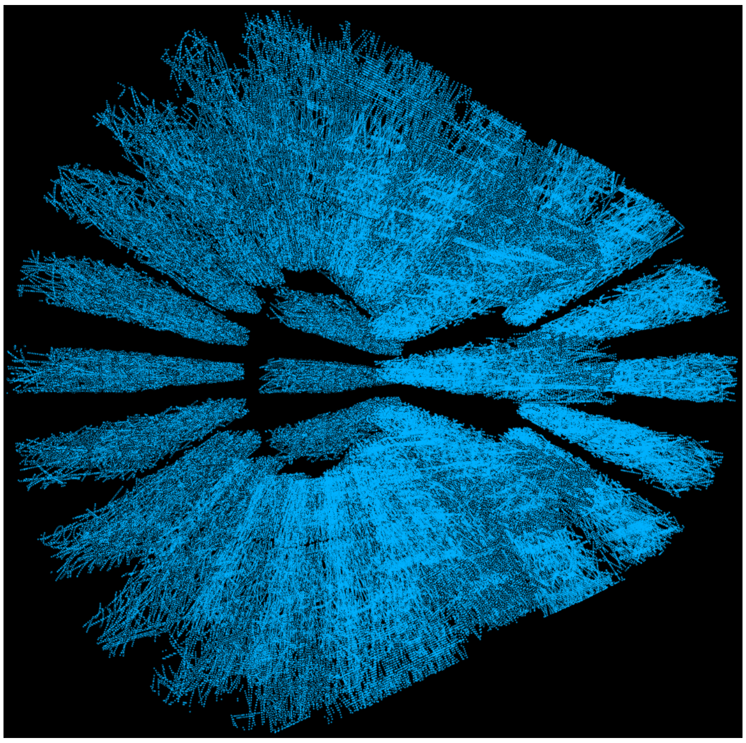


Figure 2.2: Heavy-Ion Event Split into 36 Slices

would process the next seed. The problem where a cluster could belong to different tracks is excluded. Obviously this algorithm does not allow for parallelization. The GPU tracker processes all seeds in parallel, and so the final cluster assignment can only be done after the track fit and extrapolation.

Most particles passing through the chamber can be expected to originate from the interaction point or secondary vertices in the center of the TPC. Recall that the coordinate system was chosen in a way that the z-axis points along the beam, the x-axis points along the radial direction and the interaction point is in the origin (see Fig. 2.3). This makes the x-axis distinct, since most particles can be assumed to have a trajectory alongside it. Hence it is convenient to start searching for trajectories along this axis.

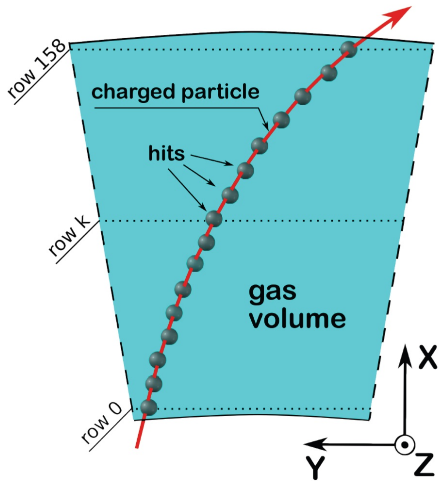


Figure 2.3: Geometry of a Single Slice [Gor1]

For this reason each tracking step is organized in a way that it processes data row by row in increasing order, which is possible as the x position is measured discretely (The x-coordinate is exactly the position of one of the rows). Therefore, it is reasonable to store all the cluster data for one row packed together. Data locality is increased and thus cache efficiency improved. This also justifies again why the x-position was chosen to be part of the track model but not as track parameter in section 2.1.4, as in this setup track parameters must be propagated to different X positions.

2.2.3 Grid

Since the algorithm works row by row it is obvious to sort the clusters according to rows and to store clusters packed for each row. Within one row two dimensions remain, therefore, the clusters cannot easily be sorted for fast access. To account for this a **grid** is introduced that splits the row (which is in fact a plane) into rectangular **bins**. Bins are counted according to their y- and z-position and clusters are sorted by their bin number. A list of **FirstCluster-InBin** - entries is maintained which, for every bin, points to the first cluster in the sorted array belonging to that particular bin. (See Fig. 2.6 for an illustration how clusters can be searched) Using this indirection a fast search for clusters in confined areas within one row is possible.

The bin size does not need to be constant for every row and is selected in such a way that the number of bins approximately matches the number of hits in one row. This ensures that neither too many clusters are contained within one bin nor many empty bins occur.[6] Furthermore, if the clusters are restricted to one area of the row, the grid will also be restricted to that area and not span the whole plane. (See Fig. 2.4 and 2.5)

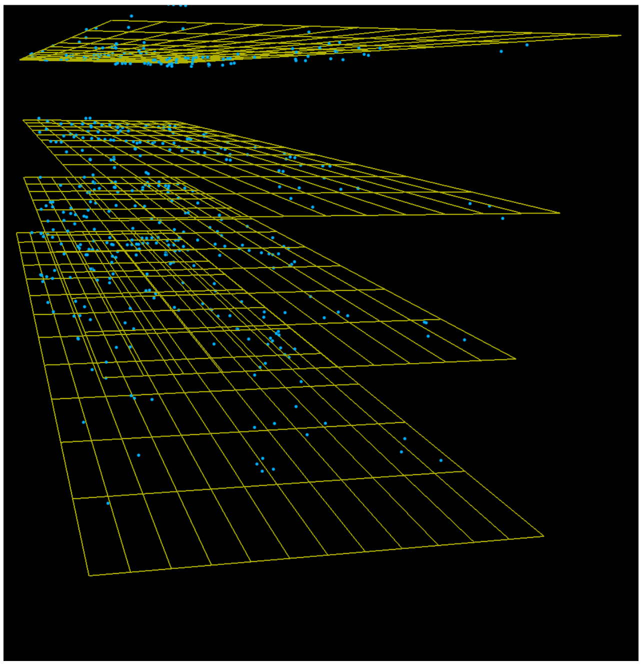


Figure 2.4: Grid of Four Rows in One Slice for a Peripheral Heavy-Ion Event

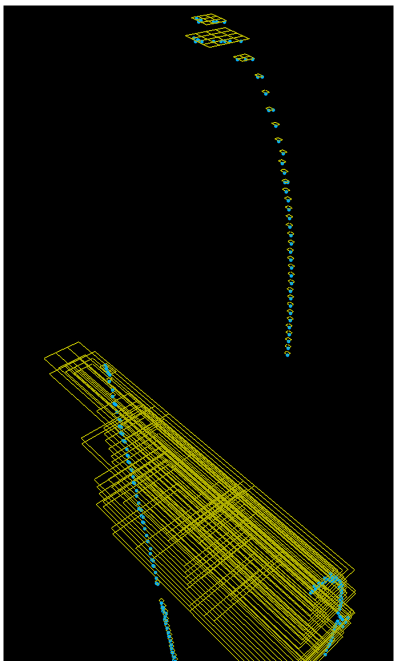
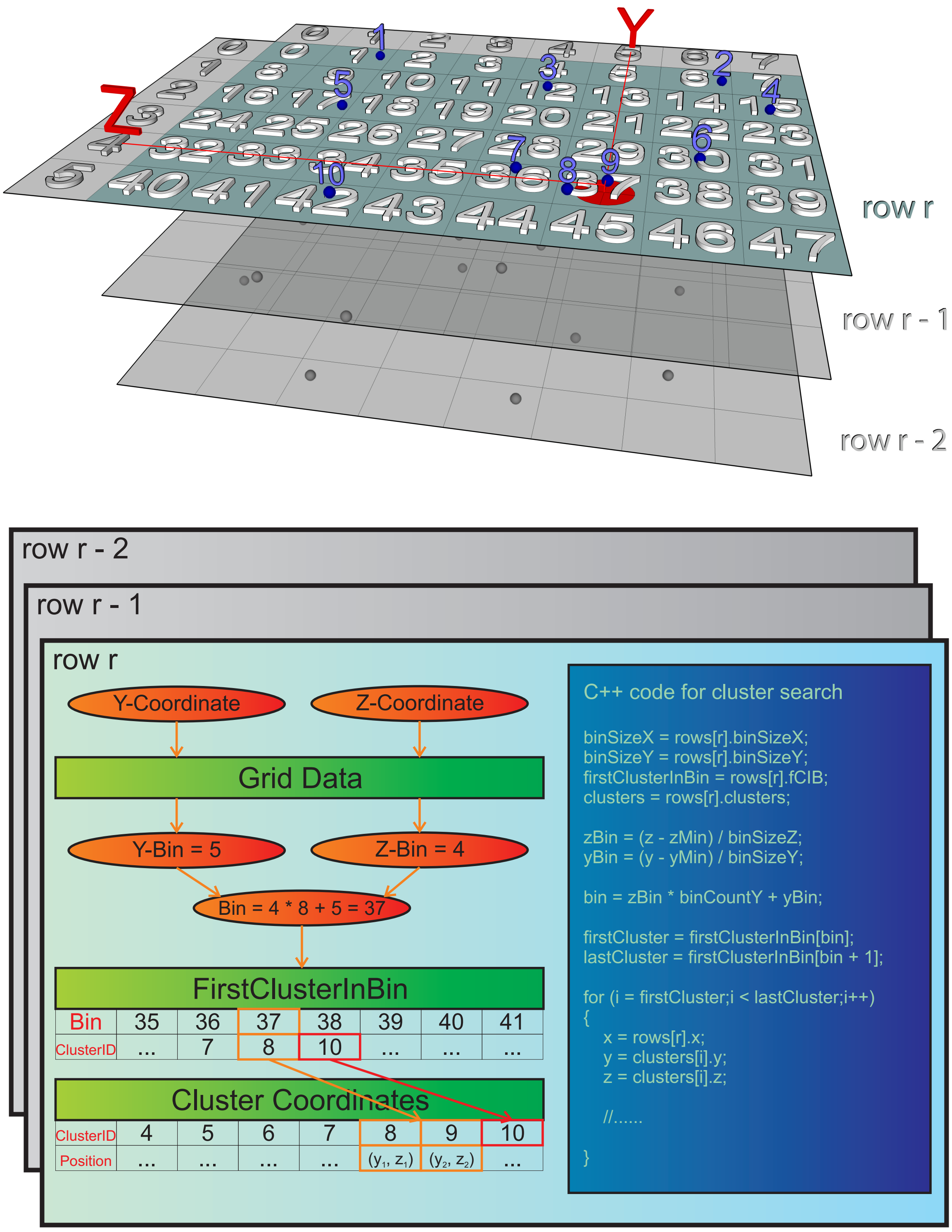


Figure 2.5: Grid of One Slice for a PP Event[7]

[6]For example, if the bin size was so small, that there were more bins than clusters in a row, some bins would necessarily be empty.

[7]The grid size is not constant.


Y
Z
row r
row r - 1
row r - 2
row r - 2
row r - 1
row r
Y-Coordinate
Z-Coordinate
Grid Data
Y-Bin = 5
Z-Bin = 4
Bin = 4 * 8 + 5 = 37
FirstClusterInBin
Bin | 35 | 36 | 37 | 38 | 39 | 40 | 41
ClusterID | ... | 7 | 8 | 10 | ... | ... | ...
Cluster Coordinates
ClusterID | 4 | 5 | 6 | 7 | 8 | 9 | 10
Position | ... | ... | ... | ... | (y1, z1) | (y2, z2) | ...
C++ code for cluster search

```
binSizeX = rows[r].binSizeX;
binSizeY = rows[r].binSizeY;
firstClusterInBin = rows[r].fCIB;
clusters = rows[r].clusters;

zBin = (z - zMin) / binSizeZ;
yBin = (y - yMin) / binSizeY;

bin = zBin * binCountY + yBin;

firstCluster = firstClusterInBin[bin];
lastCluster = firstClusterInBin[bin + 1];

for (i = firstCluster;i < lastCluster;i++)
{
    x = rows[r].x;
    y = clusters[i].y;
    z = clusters[i].z;

    //......

}
```

Figure 2.6: Illustration of Search for Clusters near a Given Location in the Grid

2.3 Slice Tracker Algorithm Steps

A detailed description of the five tracking steps follows. Several numerical parameters will appear which have been tuned by Sergey Gorbunov for optimal tracking efficiency.[8]

2.3.1 Neighbors Finder [I]

The Neighbors Finder algorithm is executed for every cluster. Given a cluster C_0 in row r it searches for the two clusters C_- and C_+ in row $r-2$ and $r+2$ (with discrete x-coordinates r_-^x / r_+^x), so that the three clusters compose the closest approximation to a straight line. The reason for which rows are skipped is explained later. To approximate straight lines the difference of the slopes between C_0/C_+ and C_0/C_- is minimized. Let (r_i^x, r_i^y, t_i^z) denote the coordinates of the cluster C_i ($i \in \{0,+,-\}$). Define:

$$\begin{aligned}
\triangle_i^j &:= r_i^j - r_0^j && \Big(i \in \{0,+,-\}, j \in \{x,y,z\}\Big)\\
S_i^j &:= \frac{\triangle_i^j}{\triangle_i^x} && \Big(i \in \{+,-\}, j \in \{y,z\}\Big)\\
S^j &:= S_+^j - S_-^j && \Big(j \in \{y,z\}\Big)\\
S &:= (S^y)^2 + (S^z)^2
\end{aligned}$$

The objective now is to minimize S. Fig. 2.7 visualizes the calculation. Clearly the calculation for different clusters C_0 is totally independent, and can therefore be done in parallel using multithreading or vectorization. However, the coordinates for the clusters C_- and C_+ cannot be streamed from memory but have to be fetched using gather operations (see Appendix C).

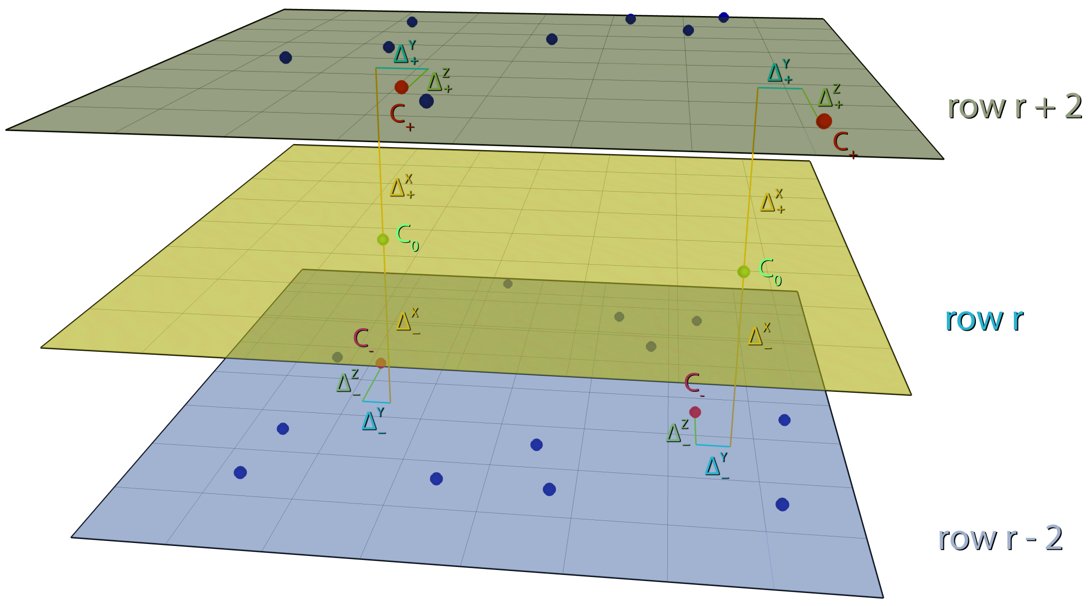


Figure 2.7: Neighbors Finder Slope Calculation

To reduce the complexity, **hit-areas** are defined in the upper and lower row, and only clusters within these areas are tested. Since the particle trajectory is supposed to start in the origin,

[8]For more information on the tracking steps see also [Gor$^+$2], [Gor1], and [Hlt].

the hit-areas are not centered around the y- and z-coordinate of C_0 but rather around the intersection point of the straight line connecting C_0 with the origin and the planes defined by $x = r^x_-$ / $x = r^x_+$ as can be seen in Fig. 2.8. The size of the hit-areas is defined by a restriction of the angle between the line through C_0 and the x-axis. All clusters belonging to the hit-areas can easily be identified and accessed using the grid.

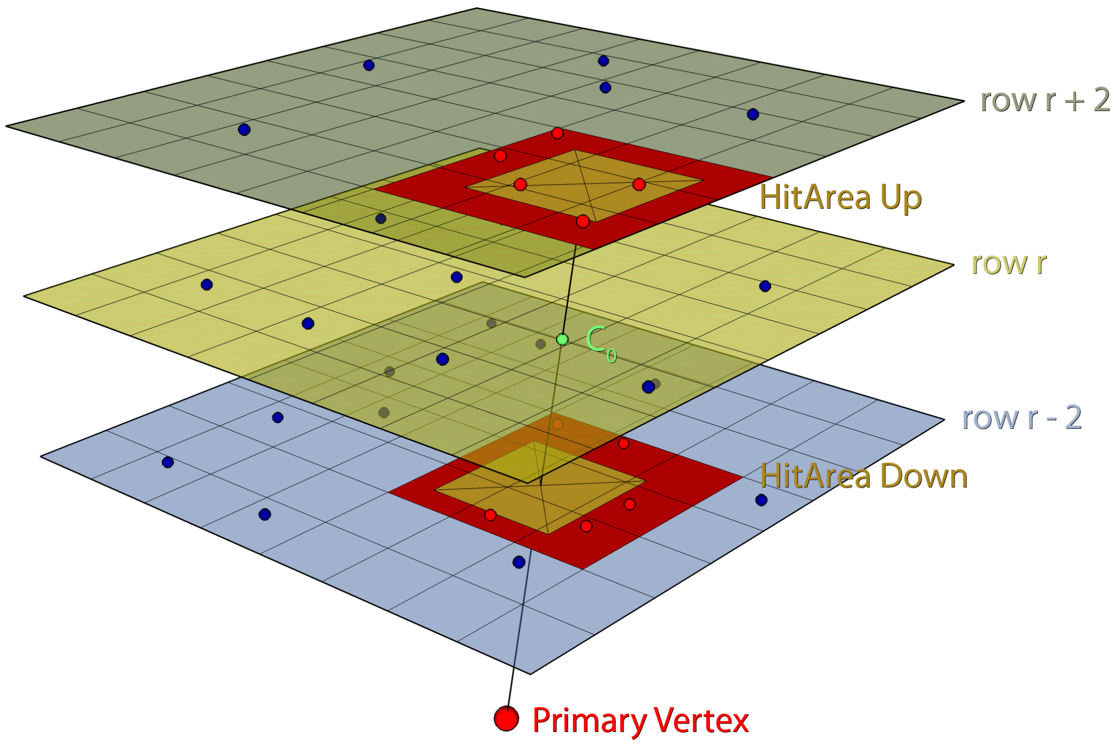


Figure 2.8: Illustration of Hit-Areas of Neighbors Finder[9]

The slope is then calculated for every pair of clusters in the upper and lower search area. Unfortunately, this still results in quadratic complexity in relation to the number of input clusters. However, even for central lead-lead events, this number regularly stays below 20, leaving little room for improvement. There is a more thorough discussion in section 6.2.1.2. Indeed the algorithm would work as well using rows $r - 1$ and $r + 1$ instead of $r - 2$ and $r + 2$, but skipping one row results in better seeds, especially for lead-lead events. There are two reasons for this: First, the amount of clusters lying within the restricted angle is higher opening up more possibilities for combination. Secondly the cluster positions have absolute errors that result in the smaller relative errors for the slope the wider the distance is. The connection to the clusters in the lower and upper row will hereafter be called a **link**. Fig. 2.9 shows a simplified two-dimensional illustration, while Fig. 2.11 shows an example of clusters and the links constructed by the Neighbors Finder using the event display.

2.3.2 Neighbors Cleaner [II]

The Neighbors Cleaner is again executed for every cluster independently. In the case that cluster C_0 links up to cluster C_+, it checks whether cluster C_+ has its downward link point to cluster C_0 and deletes the upward link of C_0 if they do not coincide. The same is done

[9]Red clusters are possible links for the green cluster C_0 in the Neighbors Finder algorithm.

for the downward links. This is illustrated in Fig. 2.10. In the event display (Fig. 2.11) links removed by the Neighbors Finder are colored grey.

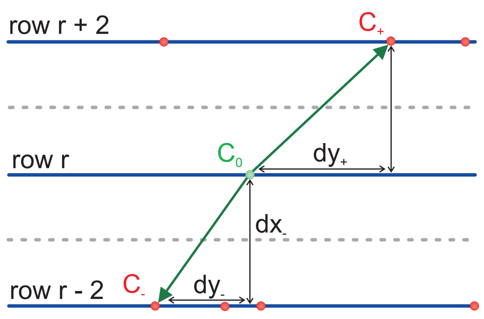


Figure 2.9: Best Links for Cluster C_0 Found by Neighbors Finder

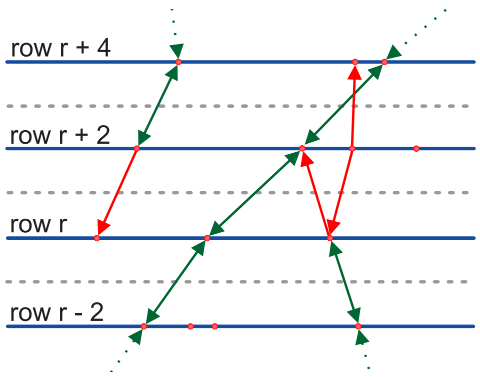


Figure 2.10: Links removed by Neighbors Cleaner[10]

2.3.3 Start Hits Finder [III]

The Start Hits Finder creates the seeds for the Kalman filter step. Every seed starts with a **start hit** which is a cluster with an upward link, but without a downward link (after the Neighbors Cleaner step). The seed is then defined by the sequence of upward links, for instance the start hit cluster C_0 in row r has its upward link set to the cluster C_1 in row $r + 2$ whilst C_1 has its downward link set to C_0 and its upward link set to another cluster C_2 in row $r + 4$. C_2 shall have no upward link and its downward link shall point to C_1. Then (C_0, C_1, C_2) would form a seed. The Start Hits Finder simply checks each cluster if it represents a start hit and arranges a list of start hits. Fig. 2.13 shows an illustration while the seeds created from the neighbors in Fig. 2.11 are shown in Fig.2.12.

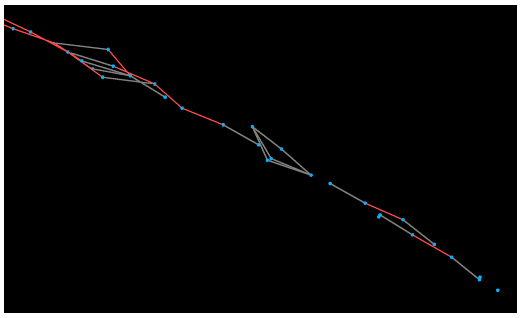


Figure 2.11: Links after Neighbors Finder / Cleaner in Event Display[11]

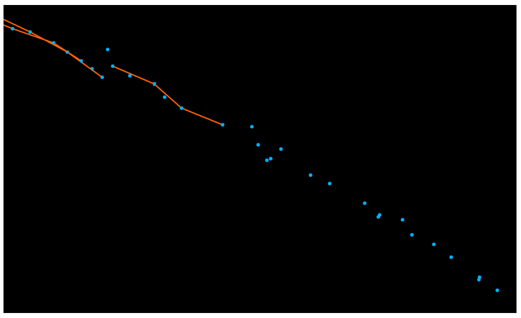


Figure 2.12: Seeds Created by Start Hits Finder in Event Display[12]

2.3.4 Tracklet Constructor [IV]

A **tracklet** is a candidate for a track and consists of clusters. Every seed found by the Start Hits Finder is now turned into a tracklet which initially consists of the clusters in the

[10]Green links are kept, red ones are removed by the Neighbors Cleaner.
[11]Links removed by Neighbors Cleaner are grey.
[12]The track clearly continues to the lower right which is not covered by the seeds. However, this is not even necessary because the remaining clusters will be found during the Tracklet Construction phase.

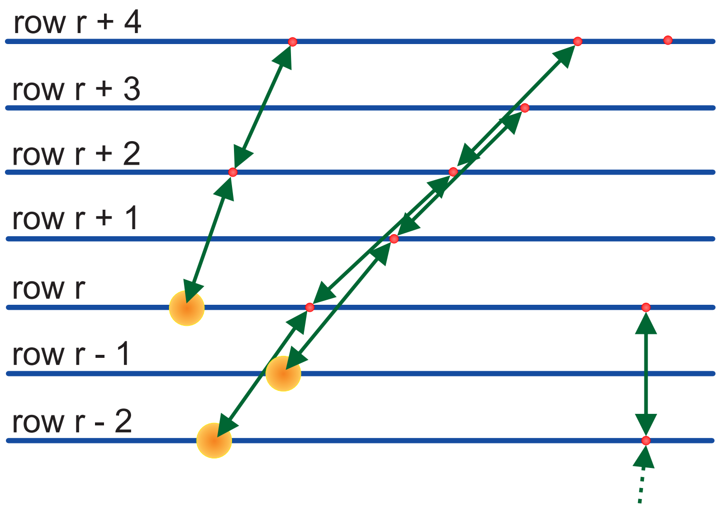


Figure 2.13: Illustration of Start Hits and Seeds[13]

seed defined by the sequence of links. The Tracklet Constructor fits track parameters to the tracklet and then tries to extend the tracklet using extrapolation. It then passes the tracklet to the next step or it may discard it, depending on several criteria. The Tracklet Constructor itself is divided into three sub-steps:

Track Fit [**IV (a)**] The Kalman filter is iteratively applied to the clusters in the initial seed creating a set of track parameters and a covariance matrix. The tracklet is dropped if the χ^2-value of the fit exceeds a defined bound or if the tracklet length is below three clusters.

Forward Extrapolation [**IV (b)**] For a seed ranging from row r to row $r+2n$, the process starts at row $r + 2n + 1$. First the tracklet parameters are extrapolated to this row. The cluster closest to the extrapolated position is determined by checking every cluster in the four grid bins next to it. If a match is found the following Kalman filter iteration is calculated for this cluster and the χ^2-value checked, which determines whether the cluster is added to the tracklet or not. The algorithm then continues with the next row, either until the last row is reached or until no cluster was found for more than four consecutive rows. Fig. 2.14 shows one extrapolation step.

Backward Extrapolation [**IV (c)**] In the same manner as above the tracklet is extrapolated to the lower rows. Since in the original seed every second row misses, the first row the tracklet is extrapolated to is $r+2n-1$. The rows are decremented in steps of 2 down to $r-1$ and then decremented in steps of one row while the same break criteria apply as above.

If, after both extrapolation steps, the tracklet consists of at least 10 clusters, it is stored and passed on to the Tracklet Selector, otherwise it is dropped. Clearly, after the Tracklet Constructor step, it is probable that two tracklets were extrapolated towards the same cluster, and thus they now share this cluster. Therefore, a criterion for selecting tracklets when

[13]The seeds are green, the start hits orange. One problem of the algorithm is apparent. The middle track is represented by two seeds, from which one contains the odd numbered rows and another the even numbered ones. They will have to be merged later. The right seed already ends in row r, but the cluster in row $r + 4$ might belong to the same track and thus has to be added in the Tracklet Constructor phase afterwards.

assigning clusters to final tracks is required. The longer tracklet is always preferred. Hence, when storing a tracklet, for each cluster a **cluster weight** is defined as the length of the longest tracklet the cluster belongs to. This is determined using an atomic max operation.[14] The cluster is then considered to belong to the tracklet whose length matches the cluster's weight. Because two tracklets might have the same length, the cluster weight is shifted to the left by 16-bits and the tracklet ID is added. This makes the algorithm deterministic, as long as tracklet IDs are persistent. The tracklets reconstructed for the previous example are shown in Fig. 2.15.

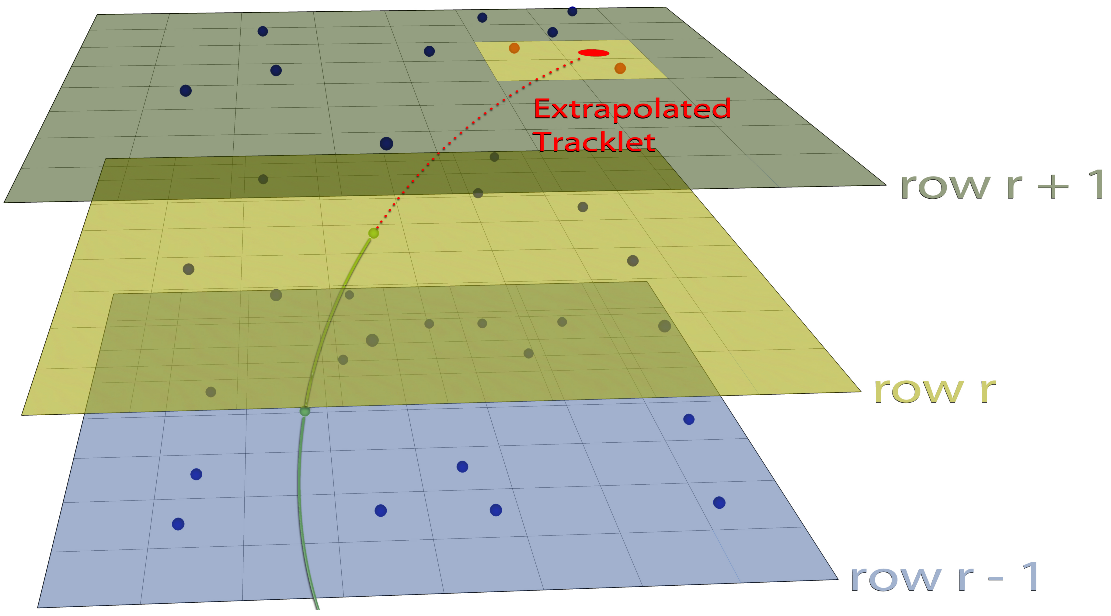


Figure 2.14: Illustration of Tracklet Constructor Extrapolation Step[15] from row r to row $r+1$

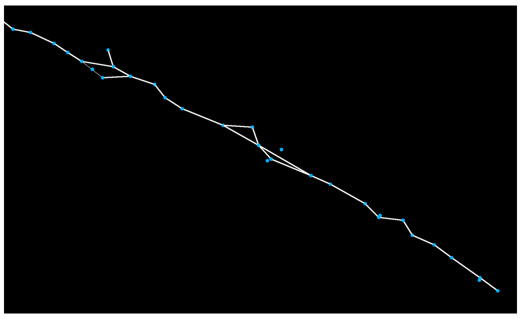


Figure 2.15: Tracklets Created by Tracklet Constructor[16]

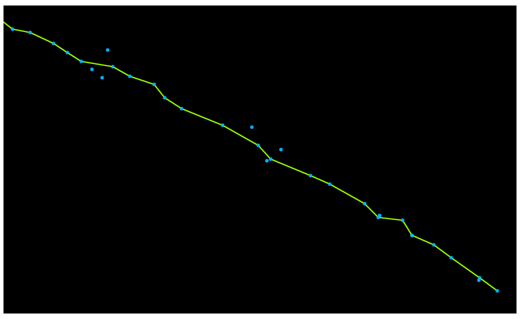


Figure 2.16: Final Tracks Produced by Tracklet Selector

2.3.5 Tracklet Selector [V]

This step elevates tracklets to final tracks and does the final cluster assignment. Since the clusters' weights are required to proceed, this task depends on the results of the Tracklet Construction phase. The Tracklet Selector processes every tracklet starting at the lowest

[14]Atomic operations read from and write to a memory location in one instruction that cannot be interrupted. See Appendix B for more information.
[15]The four bins next to the extrapolated position are highlighted.
[16]Apparently some tracklets share clusters.

row. Given that a cluster for the particular row was found during Tracklet Construction, it is verified whether the cluster weight is either equal to the tracklet's length or if the cluster possibly meets sharing conditions. A cluster may to be shared as long as the total number of shared clusters does not exceed the number of clusters that passed the check up to this point. This procedure continues until five sequential rows occurred without compatible cluster. If the set of clusters thereafter consists of 10 or more, it assembles a track. A final track ID is determined using an atomic add operation (see the example in Appendix B) and the track is stored. The procedure then starts again at the next row. The result of the Tracklet Selection on the previous example is shown in Fig. 2.16.

2.4 Initialization and Output

Though the algorithm itself consists of the five steps mentioned earlier, two additional ones are needed for the implementation. These steps are not considered a part of the algorithm itself, since their task is solely data reorganization.

2.4.1 Initialization

In an initialization step the grid is created and the clusters are sorted accordingly. Moreover, in order to save memory, and even more importantly, memory bandwidth, the cluster coordinates are converted from floats into short integers[17], linearly interpolating the range between the minimum and maximum value occurring in each row with the values 0 to 65535.

2.4.2 Track Output

In this final output step, the tracks are packed together in memory, and the local slice coordinates are transformed into global experiment coordinates.

2.5 Data Structures

It is now possible to define most of the data structures in the tracker. The seeds need no extra structure, as they are unambiguously defined by the start hit. The structures for start hits and tracklets are very simple because they are merely arrays of start hits and tracklets. This is slightly more complicated for the tracks. It is desirable to store the tracks as compact as possible. Unfortunately, since a track can consist of any number of clusters from 10 to 159, arrays are ill-suitable due to that dynamic nature. Therefore, the track data is divided into two parts, an array of the track parameters without clusters (the parameter size is constant) and a memory segment with clusters. The parameter structure contains a pointer to the first cluster belonging to the track and the number of clusters.

The situation gets even more complicated for the slice data which embraces the cluster coordinates, grid content, links, and cluster weights. Each of them is stored in one memory segment. Within a segment the data is sorted according to rows. The clusters themselves are further sorted by their grid bin.

For every row a set of pointers is maintained, pointing to the corresponding row data within each of these segments. Fig. 2.17 shows an overview of the data structures used.

[17]A short integer denotes a 16-bit integer.

2.5.1 Data Types

For performance reasons the whole slice tracker algorithm is solely processing single precision floating point numbers. As shown in [Gor$^+$3] and [Gor1], the Kalman filter had to be adapted to guarantee for numerical stability. To spare some memory, most integers are restricted to 16-bit, and floats are interpolated using 16-bit integers wherever possible. This leads to the following data stored as 16-bit integers: row IDs, cluster indices within one row, cluster y- and z-coordinates, upward and downward links, FirstClusterInBin entries.

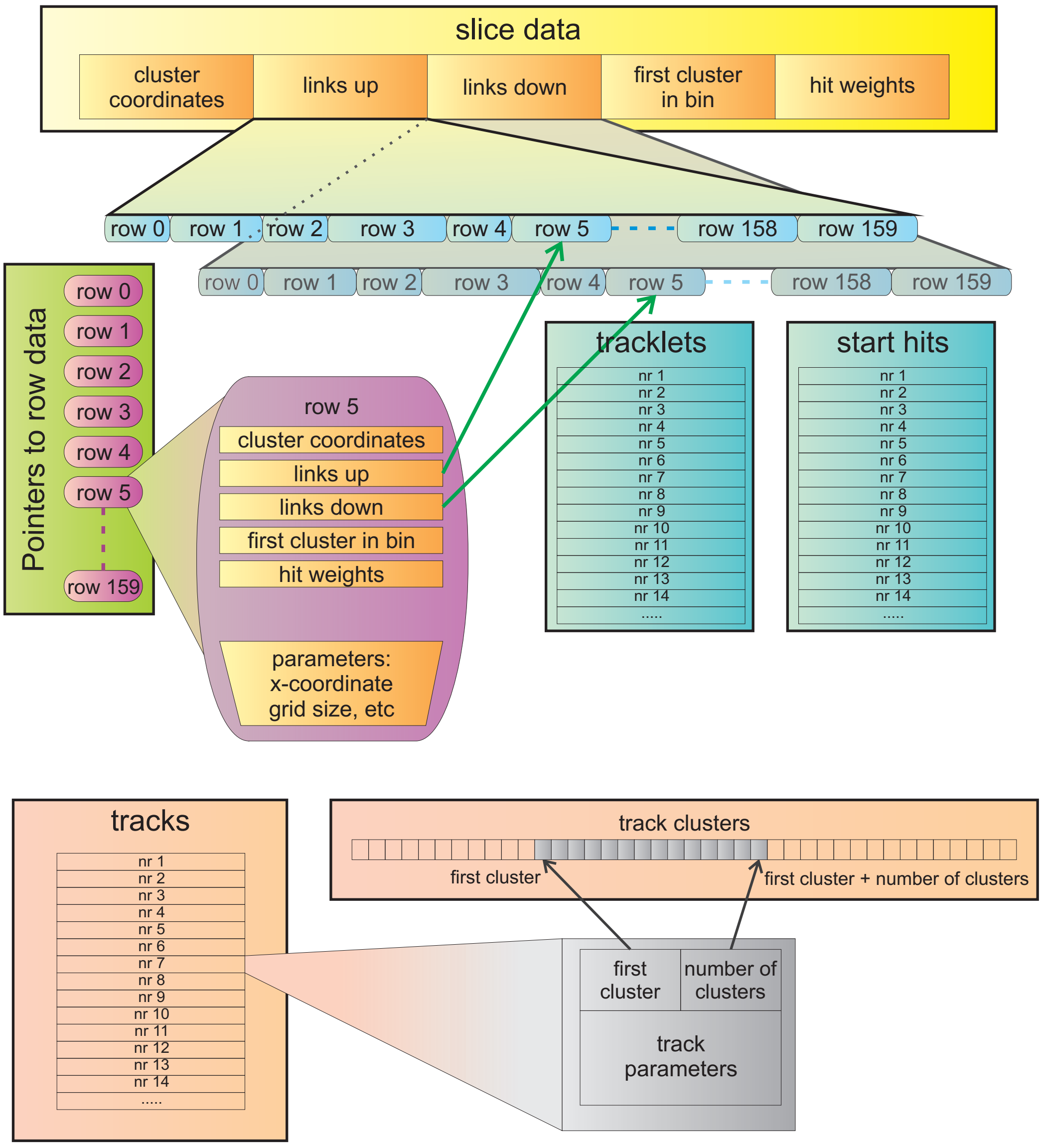


Figure 2.17: Data Structures in CA Tracker

Chapter 3

Hardware Accelerators

3.1 Overview

Since the tracking represents one of the most time consuming tasks in event reconstruction, it is evident that the tracker is a target for optimizations. While the clock speed of state of the art CPUs has stagnated in the last years, processor designers have improved the efficiency and integrated more parallel approaches instead. Graphics cards, however, have been designed with parallelism in mind for many years now. They have become more and more powerful, undergoing a higher and faster performance increase than CPUs. In addition, just recently the GPU support for high level languages greatly improved making it comparatively easy to run general purpose code on GPUs.[1]

For these reasons it is evident to parallelize the algorithm as much as possible and to attempt to fully exploit multithreading and SIMD[2] capabilities. Current CPUs all implement at least a basic set of vector instructions. Modern compilers can take profit from this by autovectorizing the code. Still, manually vectorized code generally performs much better. [Kre] introduces an abstraction for vectorization. Programs using this abstraction can benefit from extended hardware support for vector operations just by recompilation. Besides standard CPUs, graphics processors from both NVIDIA and AMD[3] provide an excellent platform to experiment with and to port the tracker code to. Moreover, Intel is currently working on the Larrabee (see [Int3]) which was long delayed, and it is still uncertain if or when a final product will be released. The Larrabee is a graphics processor with a different design in comparison to current chips from NVIDIA and AMD. It is capable of executing x86 general purpose code extended with enhanced vectorization instructions and might become an alternative in the future.

A vectorized version of the tracker code was written by Matthias Kretz for his diploma thesis [Kre]. It uses special Vector Classes (also introduced in [Kre]) allowing for an abstraction of the vector instructions and thus facilitating one single common source code optimized to benefit from SSE and LRBni.[4]

[1]This is often called GPGPU.

[2]SIMD stands for Single Instruction Multiple Data and is a low-level form of parallelism. As the name implies the operation is performed on a data vector instead of a scalar.

[3]Radeon graphics cards and the Stream framework were formerly released by ATI which now belongs to AMD.

[4]SSE is a vector extension introduced with the Pentium III. SSE was updated multiple times. The current version is SSE 4.2. LRBni is the set of new vector instructions of the Larrabee.

In this thesis, a port of the tracker code to the NVIDIA CUDA framework will be presented in detail. The CUDA framework has limited support for C++, in contrast to the AMD framework, which currently does not support C++ at all. Thus NVIDIA was favored over AMD. To avoid the creation of two unrelated tracker codes it was imperative to stick with C++, as the original tracker code is integrated in AliRoot (see 8.2 and [Ali4]) and relies on AliRoot classes, thus enforcing the usage of C++. Unfortunately, the new OpenCL framework, which supports various platforms, is also restricted to plain C making a future adaption difficult, to say the least.

3.2 NVIDIA GT200b GPU and CUDA Framework

In order to comprehend the optimizations applied later a basic understanding of the deployed GPU as well as the CUDA framework is necessary. With new graphics processors entering the market, NVIDIA integrated additional features and provided new framework versions. An increasing **compute capability** value is assigned to each GPU generation defining which features are available. To be more accurate, for the tracking algorithm described later to compile and execute, a framework version of at least 2.0 and a GT200 chip or newer is required, which has the compute capability 1.2. Most benchmarks presented were done using a GT200b and framework version 2.3, therefore these versions are described here. In the following the notations **device** and **host** identify the GPU board and the CPU / main memory respectively. A more comprehensive description of CUDA is available in [Nvi].

3.2.1 Multiprocessors

The GT200b chip possesses 30 independent **multiprocessors** that can be compared to cores in modern CPUs. Each multiprocessor has eight single precision floating point ALUs[5] and one double precision ALU. However, the single and double precision ALUs share components, so in each cycle either eight single precision float operations or one double precision float operation can be executed. Therefore the user is advised to use single precision wherever possible. (Fortunately, even before this work was started, the tracker code did not use double precision at all.) There are no special ALUs present for integer calculations. The ALUs can add and subtract 32-bit integers. Two 24-bit integers can be packed into two floats for multiplication. Multiplication of 32-bit integers, however, is not supported and has to be emulated using several floating point multiplications resulting in low performance. It is therefore convenient to stick with floating point values whenever possible. For a schematic of the chip see Fig. 3.1.

3.2.1.1 Warps

Every multiprocessor can run multiple threads in parallel. These threads are organized in **warps** of 32 threads each. The multiprocessor can uphold up to 512 concurrent threads. Each instruction issue cycle it selects one warp ready to execute without any scheduling overhead and issues the next instruction to the active threads of that warp. Since only one instruction decoder is present, threads of one warp are restricted to execute a common instruction. If in conditional code, different threads within a single warp take different branches, execution

[5]ALU stands for Arithmetic Logic Unit. A processor with n ALUs can calculate n arithmetical operations in parallel.

for every branch is serialized (**warp-serialization**). Different warps can nevertheless clearly execute distinct instruction independently.

All warps are further split into two half-warps, the higher and the lower numbered 16 threads of one warp. Simultaneous memory accesses by every thread in one half-warp can be **coalesced**, resulting in only one single memory transaction (see 3.7).

From this perspective the chip looks very similar to a vector processor.[6] Contrary to instruction sets like SSE the programmer does not have to write explicit vector code, but from the programmer's view each thread is independent. This makes an adoption of CUDA comparably easy. However, the drawback is, if a program is written without the warp concept in mind, only one sole ALU is used in the worst case resulting in greatly decreased performance. (The same is true for vector processors when scalar instructions are executed.)

3.2.2 Execution Configuration

Kernels C++ functions to be executed on the GPU which can be started from the host are called **kernels**. When a kernel is called, it is started many times in parallel on every multiprocessor according to the execution configuration. Such a configuration is an arrangement of **blocks** in a **grid**. Kernels are generally small functions. Their execution time has to stay below one second or they will timeout. Every function call within the kernel is inlined. Big, complex functions easily exhaust the register pool available (see below) leading to kernels of poor performance. Only one single shared kernel can be executed in parallel on all multiprocessors.[7]

Blocks A block is a set of threads which will be executed on the same multiprocessor having access to the same shared memory space. Therefore, data exchange within one block is easily possible, while there is limited functionality to do so between different blocks. A block can contain up to 512 threads organized in a one-, two- or three-dimensional way. Threads can access their thread ID within a block and the block dimensions by the variables *threadIdx* (resp. *threadIdx.x* etc.) and *blockDim* respectively.

Grid The grid is an arrangement of blocks. The blocks can be executed on different or even the same multiprocessor if there are enough resources available on it to start more than one block in parallel. Block execution order is not well defined, so one cannot assume two blocks to run in parallel or sequentially. Therefore, within one kernel, blocks should work on distinct data sets. Data exchange must be done afterwards, when execution of all blocks is known to be finished. The grid is organized in one or two dimensions, block IDs and grid dimension can be queried by the threads in the same way as thread IDs and block dimensions.

3.2.3 Registers

There is a pool of $16,384$ 32-bit registers available on every multiprocessor. These registers are assigned to the threads running on it. Each kernel has a fixed requirement of registers further limiting the maximum number of threads that can be executed concurrently on one multiprocessor. Registers are the only fast storage for temporary variables available. Thus it is wise not to store too many intermediate results. During compilation an upper limit for

[6]A vector processor operates with shared instructions on vectors instead of scalars.
[7]This is going to change with the next NVIDIA GPU generation called Fermi.

registers can be set for each kernel. Local memory (see below) will then be used in lieu of registers.

Another case where usage of local memory is obligatory occurs when arrays are dynamically accessed, since registers cannot be used in this way. This problem along with ways to circumvent it will be illustrated in section 6.2.1.3.

3.2.4 Memory

The GT200b boards have different types of memory. Some of them physically share the same memory chip, but are accessed in different ways.

Global Memory The GPU's global memory usually consists of 1 GB of GDDR3 for the GTX285 cards and 4 GB for the Tesla boards. The maximum achievable memory bandwidth is slightly above 100 GB/s. The global memory is neither cached in any way nor is it coherent.[8] Thus, within one kernel, reading from a memory position that was written to earlier yields unpredictable results as does writing to the same location twice.[9]

Constant Memory Constant memory is an isolated part of 64 kB of the global memory, which is readonly (in the sense that only the host can write to it) but cached.

Texture Memory Texture memory also is a part of global memory. Areas of global memory can dynamically be declared as textures and then be read through the texture cache. This inhibits direct access by pointers; instead texture fetches must be used (see 6.3.1.2). The texture memory is readonly. When reading data from texture memory through the texture unit, some conversions or even bilinear filtering can be done at no extra cost.

Shared Memory Shared memory is not part of global memory. Instead, every multiprocessor has its own 16 kB of fast shared memory. Threads within one block access the same shared memory space. If more than one block is concurrently executed on the same multiprocessor shared memory must be partitioned among the blocks. For a read after write access pattern to shared memory, the threads must be synchronized in between as explained later. When following the coalescing rules (see 3.7.2), shared memory is exactly as fast as registers are.

Local Memory Local memory is part of global memory. It is used if the register pool is insufficient. Local memory is not cached and thus usage should be used as a last resort. As the name indicates, different threads have disjoint areas of local memory in global memory.

[8]Fermi cards will have a global memory cache.

[9]There are possibilities for memory fences, eliminating this problem for threads within one block but not for the whole grid. However, there is no global flush instruction, that forces every thread to wait until all memory accesses are finished.

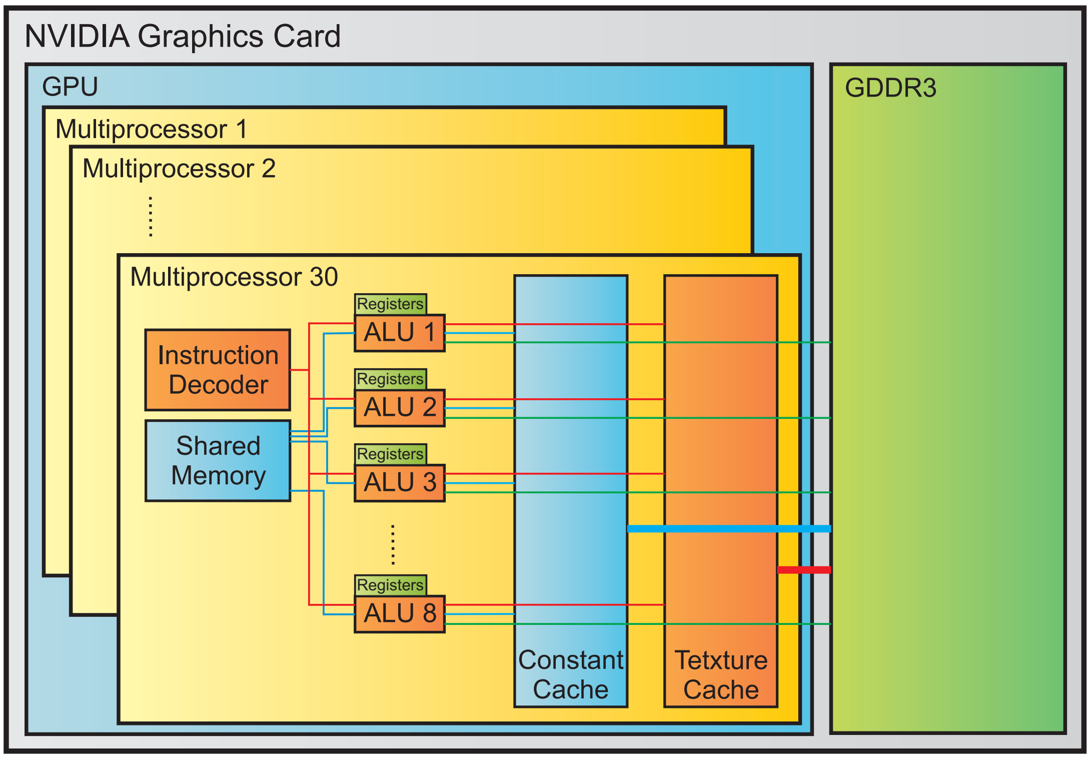


Figure 3.1: Schematic of Current NVIDIA GPU

3.3 C extensions

The following specifiers for functions (and class methods) exist:

- *__global__*: Functions declared as **global** represent CUDA kernels.
- *__device__*: Functions declared as **device** can be called from global functions or other device functions on the GPU.
- *__host__*: This declares regular host functions. Therefore, the keyword **host** can be omitted. However, a function can be specified as both host and device in what case the function is compiled twice, as host and as device function.

A kernel can now simply be executed by calling a global function from host code. The desired parameters are passed to the kernel and some further special parameters define the block and grid size. A kernel call has the following format:

```
cudaKernel<<<grid_dimension, block_dimension>>>(arg1, arg2, ..., argn);
cudaKernel<<<30, 256>>>(a, b);
```

Listing 3.2: CUDA Kernel call

The latter line will execute the kernel cudaKernel starting 30 blocks of 256 threads each, passing the parameters a and b to the kernel. See List. 4.4 for a real example.

3.4 Synchronization

As already indicated the execution is organized in terms of warps processing threads within one warp simultaneously. Different warps and especially different multiprocessors are independent (even though they have to execute an identical kernel). It is possible to synchronize all threads of one block with the *__syncthreads* intrinsic.[10] A call to *__syncthreads* requires only four clock cycles, afterwards all shared memory access is ensured to be complete and every thread in the block has arrived at the same position in the code.[11]

3.5 Compilation

When compiling a CUDA file (".cu" extension), the CUDA compiler will process all functions defined as *__global__* and create kernels out of them. It will inline all calls to *__device__* functions. Afterwards functions marked as *__global__* or as *__device__* are removed and the compiled CUDA kernels are integrated in the C++ source code as constant hexadecimal arrays. The calls to CUDA kernels are exchanged with calls to the CUDA runtime library, which loads the correct kernel to the GPU and executes it. The preprocessed C++ source code is then processed by the host compiler, which is GCC[12] for a Linux based system. The CUDA compiler can either directly call the host compiler or it can write the preprocessed CPU code to a file. In the latter case the host compiler must be called manually.

3.6 Compatibility

Throughout this work some incompatibilities between the CUDA and the host compiler appeared. Under certain conditions the CUDA compiler produces C++ code the host compiler is unable to process. Most of the errors were related to the *iostream* library and to templates. To cope with these problems the second compilation method producing a preprocessed intermediate file can be used. A patch is then applied to this file before it is processed by the host compiler.

3.7 Coalescing Rules

There are several restrictions on memory access patterns, which should be considered to achieve good memory performance. These rules are called coalescing rules. Coalescing rules affect threads within one half-warp only. The different rules for global and shared memory will now be explained.

[10]Intrinsic functions (or intrinsics) are often used to implement vectorization or parallelization. They are usually used to execute low level hardware instructions from high level languages. Typically Intrinsic functions are, similar to inline functions, substituted by a sequence of automatically-generated assembler instructions. Contrary to inline assembler code the compiler has an intimate knowledge of the intrinsic allowing for better optimization.

[11]Clearly the synchronization process itself can take more than four cycles if there are memory transactions pending. Also the first warp in the block executing *__syncthreads* has to wait for the remaining warps to arrive at the same code position. However, the instruction itself requires only four cycles.

[12]GCC is an abbreviation for the GNU Compiler Collection.

3.7.1 Global Memory Coalescing

Global memory coalescing policies greatly improved with the introduction of the GT200 chip. The rules as explained now are only valid for device with a compute capability of 1.2 and higher. Parallel memory access by all threads of a half-warp is coalesced into a single memory transaction, as soon as the overall access is restricted to one memory segment.[13] The segment size is:

- 32 bytes for a byte access (8-bit)
- 64 bytes for a word access (16-bit)
- 128 bytes for double word (dword) and quad word accesses (32/64-bit)
- 128 bytes for an access with data types bigger than 64-bit. This will always result in at least two memory transactions

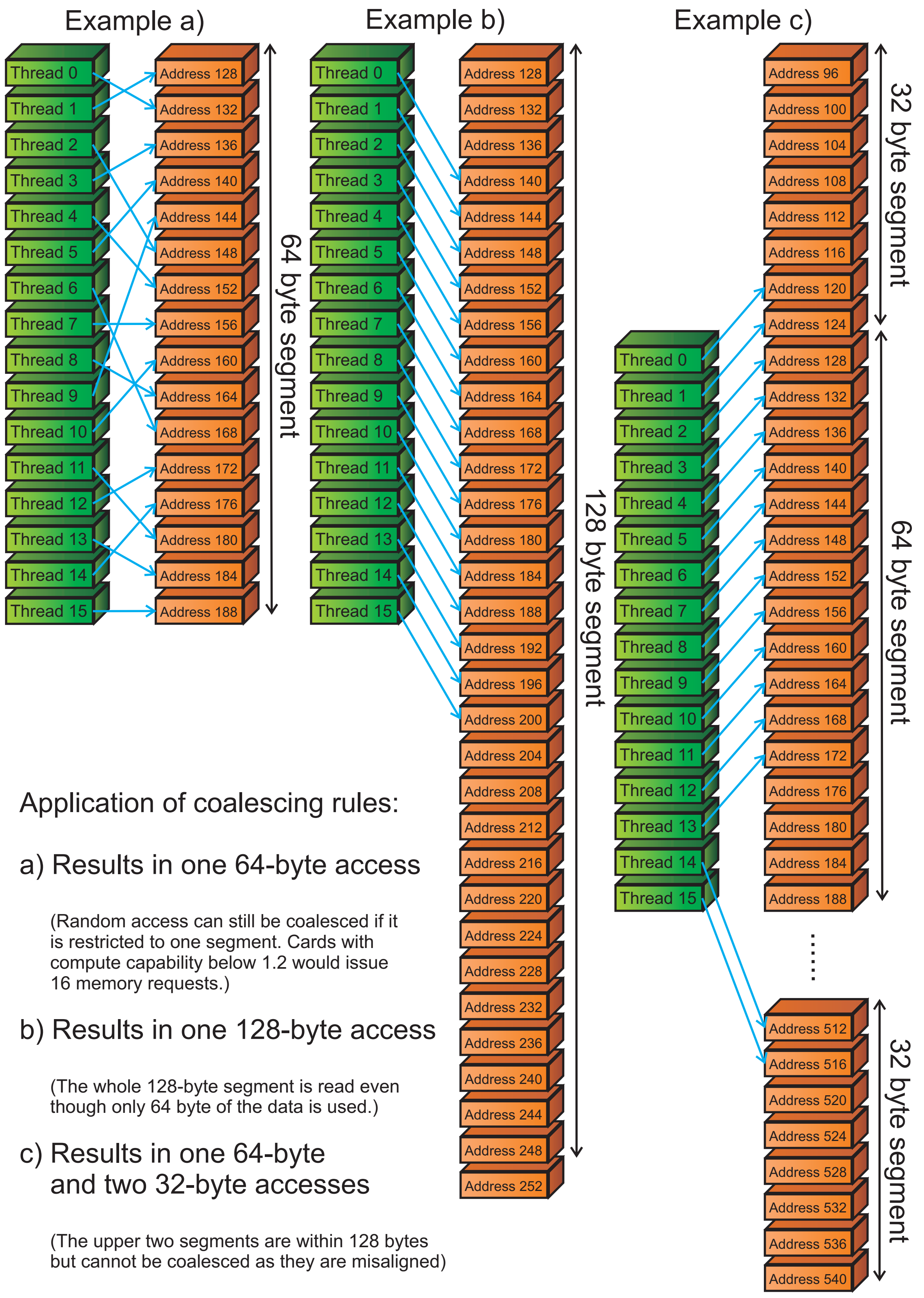
The coalescing is independent of the access order within one segment. If the memory access range spans across n segments, then n memory transactions will be issued. Every transaction will read an entire segment regardless of the access pattern. Since unused data is still read, wasting memory bandwidth, the smallest possible segment size is usually used. A segment of size s must be aligned to s bytes.[14] Examples of ordered and unordered as well as aligned and unaligned accesses are provided in Fig. 3.3.

3.7.2 Shared Memory Coalescing

The shared memory for each multiprocessor is divided into 16 equally-sized banks, which can be accessed in parallel. To achieve optimal performance each of the 16 threads of a half-warp should access a different memory bank. The banks are organized such that successive 32-bit double-words are assigned to successive banks. Access of 16 threads to an array using a stride of s (i.e. thread i accesses address [BaseAddress+$s \cdot i$]) is fully coalesced if s and 16 are coprime, which is the case if and only if 2 does not divide s. As an exception to this coalescing rule, data from one single memory bank can be distributed to multiple threads accessing the same address. If different threads access identical banks, collisions occur, and multiple memory requests are issued as necessary. Examples for both cases are provided in Fig. 3.4.

[13]Earlier devices could only coalesce memory access if consecutive threads accessed consecutive memory addresses.

[14]Aligning to s bytes means that the first address must be divisible by s.



Application of coalescing rules:

a) Results in one 64-byte access

(Random access can still be coalesced if it is restricted to one segment. Cards with compute capability below 1.2 would issue 16 memory requests.)

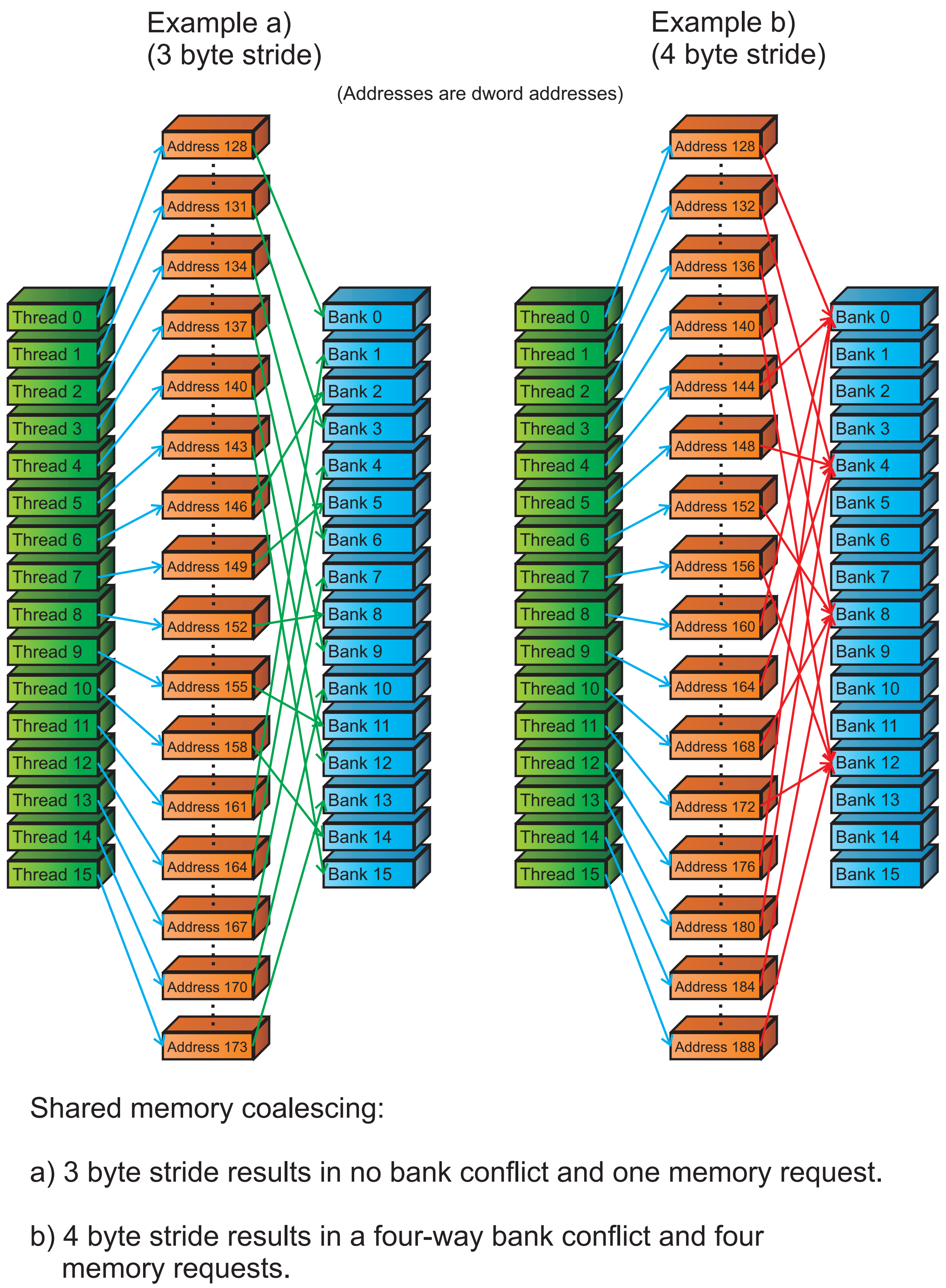
b) Results in one 128-byte access

(The whole 128-byte segment is read even though only 64 byte of the data is used.)

c) Results in one 64-byte and two 32-byte accesses

(The upper two segments are within 128 bytes but cannot be coalesced as they are misaligned)

Figure 3.3: Examples of Global Memory Coalescing



Shared memory coalescing:

a) 3 byte stride results in no bank conflict and one memory request.

b) 4 byte stride results in a four-way bank conflict and four memory requests.

Figure 3.4: Examples of Shared Memory Coalescing

Chapter 4

Implementations

4.1 Multithreading

To allow for a performance comparison later, a multithreaded CPU version is needed. The question arises which parallelization approach to choose. For not too many CPU cores the most trivial approach is also the most efficient one. As the slice trackers are totally independent, a trivial parallelization over the slices can easily be implemented and should scale almost perfectly. Clearly it could be attempted to use multithreading within one slice, too. However, this would be much more complicated and it is unclear how that approach would scale with more cores. Additionally this would only be needed if the number of processor cores exceeded the slice count.

A simple multithreaded version employing OpenMP was created which will be used throughout this work to allow for a fair comparison. However, this variant will not run on the HLT farm. There the cluster framework itself starts multiple tracker instances on one node to utilize all its processor cores. This is explained in section 8.3 in more detail. Fig. 4.1 shows how the multithreaded CPU performance scales with the number of CPU cores.

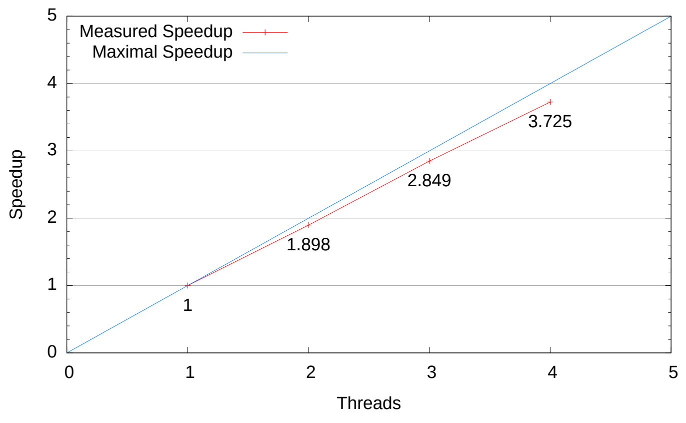


Figure 4.1: Multithreaded CPU Speedup[1]

[1]A quad core Nehalem was used for the benchmark. To avoid parasitic effects Hyperthreading was deactivated. As each number in the interval divides the slice count 36, every case could be parallelized optimally.

4.2 Tracking on GPU

This section describes what was done to make the tracker run on the GPU in the first place, especially how tracking is distributed among threads and blocks, and how slice data is stored on GPU memory.

4.2.1 Common Source

Obviously for maintenance reasons it is preferable to have only one single source code for GPU and CPU. As CUDA is in fact C code, this aim does not seem completely out of reach. Of course there has to be specialized wrapper code for both GPU and CPU. Sergey Gorbunov experimented with a GPU tracker code before research for this work was even started. Macros were introduced which made it possible to have one single tracker source code that can then be compiled by the C++ and the CUDA compiler. This realization was conserved, and thus it was easy to implement the GPU tracker on a common source principle. The modus operandi will now be briefly demonstrated.

The following macros are defined:

```
#ifdef GPUCODE
#define GPUd() __device__
#define GPUh() __host__ inline
#define GPUg() __global__
#else
#define GPUd()
#define GPUh()
#define GPUg()
#endif
```

Listing 4.2: definitions.h

For every tracking step there is one main processing function that either processes a cluster (steps I to III) or a tracklet (steps IV and V). For example, in the Tracklet Constructor case, there is the function *UpdateTracklet* which takes the tracklet ID as argument. These functions are defined as *GPUd()*. They can be compiled in a C++ file for the CPU tracker, but can be included in a CUDA file, to be compiled for the GPU tracker.

Functions that have to be executed on the host are defined as *GPUh()*. The clue is, that the host function will be an inline function when compiling the CUDA source code. The host compiler processing the CUDA file after the preprocessing step will not export the host function as a symbol in the object file. This is necessary, because the symbol for the host function is already exported when the code is processed by the C++ compiler for the CPU tracker code. If the CUDA compiler would also export the symbol, a collision would occur when linking the object files. It follows a simplified example, illustrating how this works for the Tracklet Constructor:

```
#include "definitions.h"

GPUd() void UpdateTracklet(int dwTrackletId)
{
    //Code to update the tracklet
}

GPUh() void TrackletConstructorCPU(int dwNumberOfTracklets)
{
    for (int i = 0;i < dwNumberOfTracklets;i++)
    {
        UpdateTracklet(i);
    }
}
```

Listing 4.3: TrackletConstructor.cpp

```
#define GPUCODE
#include "TrackletConstructor.cpp"

GPUg() void TrackletConstructorGPU(int dwNumberOfTracklets)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    if (i < dwNumberOfTracklets)
    {
        UpdateTracklet(i);
    }
}

void RunGPUTracker(...)
{
    int dwNumberOfTracklets;
    //Tracking steps I to III

    //Run the Tracklet Constructor with sufficient blocks
    //of 256 threads each
    TrackletConstructor<<<(dwNumberOfThreads / 256) + 1, 256>>>
        (dwNumberOfTracklets);

    //Step V
}
```

Listing 4.4: GPUTracker.cu

4.2.2 Tracking Steps

The actual tracking steps I to V were adapted for the GPU. The initialization and Tracklet Output tasks stay on the CPU for the time being. The main reason for this is that initialization and Tracklet Output require additional input data in comparison to steps I - V. This additional input data is accessed exactly once in the process rendering a transfer to the GPU inappropriate. The GPU tracker works as follows:

- Initialization is done on the CPU (creating the grid etc.).
- All relevant data is transferred to the GPU.
- Tracking steps I to V are executed on the GPU.
- The final tracks are transferred back to the host.
- Tracklet Output is performed by the CPU.

The execution configuration is the following: Every block will consist of 256 threads. In steps IV and V every tracklet is processed by a different thread. Enough blocks are started to ensure the creation of sufficient threads. There might be more threads than tracklets (because each block consists of 256 threads). The remaining threads are inactive. This is exactly the situation in List. 4.4. For steps I to III one block per row is started, where thread i within block b processes all clusters of index n in row b with $i \equiv n \pmod{256}$.

4.2.3 Row Synchronicity

Since it was integrated in the first GPU tracker implementation and will be used regularly, the principle of a **row synchronous** Tracklet Constructor will be explained here. Row synchronicity means, that the Kalman filter and the extrapolation for all tracklets within one warp or block respectively is always performed for a common row. A distinction is drawn between **warp based** and **block based** row synchronicity. Row Synchronicity is illustrated in Fig. 4.5.

As every tracklet has a **start row**, defined by the start hit of its seed, the corresponding thread must wait until the row iteration has reached that row. Although some threads will idle, row synchronicity can result in better locality because all threads access data in one single row. Furthermore, it allows for more optimizations discussed later. After a row is completely processed, the threads are synchronized by the *__syncthreads* intrinsic. List. 4.6 shows the above example extended with block based row synchronicity.[2]

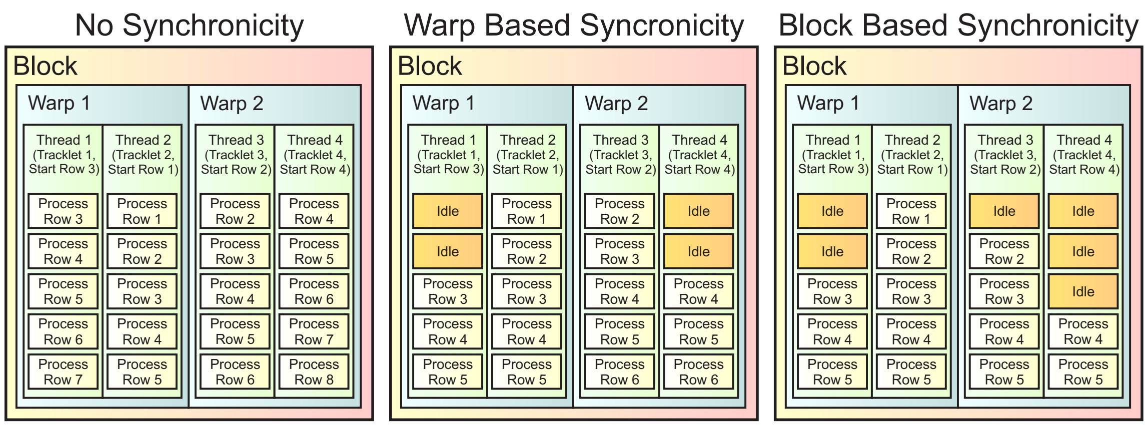


Figure 4.5: Illustration of Row Synchronicity

[2]A warp wide row synchronicity could be realized by omitting the *__syncthreads* intrinsic. The existence of only one single instruction decoder then implicitly enforces warp base row synchronicity.

```
GPUd() void UpdateTracklet(int dwTrackletId, int dwRow)
{
    if (dwRow < tracklets[dwTrackletId].firstRow) return;

    //Code to update the tracklet
}

GPUg() void TrackletConstructorGPU(int dwNumberOfTracklets)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    for (int j = 0;j < dwRowCount;j++)
    {
        if (i < dwNumberOfTracklets)
        {
            UpdateTracklet(i, j);
        }
        __syncthreads();
    }
```

Listing 4.6: Block based row synchronous *UpdateTracklet* function[3]

4.2.4 Memory

4.2.4.1 Memory Structure

Before getting to a memory layout, one has to recapitulate the memory requirements of the tracker. The memory allocated by the tracker can be categorized as follows:

- Geometry and Parameters: Constant parameters describing the geometry, etc., of the slice, independent of the event.
- Constant Pointers: As the data size for one row, etc., is not constant, and the data is sorted according to rows in memory, constant pointers point to the suitable data structures for every row. The same holds for other data. As the data sizes vary, these pointers depend on the event.
- Slice Data: This contains all actual data related to one slice of the events. This cluster coordinates and the grid. Slice Data is stored in just one memory segment with the constant pointers required for accessing it.
- Tracklet Memory: Temporary memory for the Tracklet Constructor and Tracklet Selector. It contains parameters and clusters for the tracklets.
- Track Memory: Area where the Tracklet Selector stores the final tracks. This memory area must later be transferred to the host.

Tab. 4.7 shows average values for central lead-lead events per slice. For a simplified comparison, the memory available on the GPU is listed again in Tab. 4.8.

[3]The check for the tracklet ID must be inside the loop. Otherwise the thread would never execute the *__syncthread* instructions resulting in a synchronization failure.

Data	Memory needed
Slice Data	6 MB
Tracklet Memory	12 MB
Track Memory	2 MB
Cluster Coordinates and Grid per Row (central event)	30 kB
Cluster Coordinates and Grid per Row (noncentral event)	6 kB
Geometry, Parameters and Constant Slice Pointers (slice)	2 kB
Constant Row Pointers (All Rows of One Slice)	13 kB

Table 4.7: Tracker Memory Requirements

Memory Type	Memory Available
Constant Memory (Cached)	64 kB
Shared Memory (Fast)	16 kB (per Multiprocessor)
Global Memory	1 GB

Table 4.8: GPU Memory Types

It is definitely preferable to store as much data as possible in shared and constant memory. Matching the memory requirements to the memory offered by the NVIDIA cards suggest the following assignment.

- Slice Data, tracklets, and tracks have to be stored in global memory as they do not fit anywhere else.
- Constant pointers for the slice itself and for all rows fit perfectly in constant memory.
- For noncentral event, cluster coordinates and the grid content could be stored in shared memory for faster access.
- The constant pointers could also be stored in shared memory.

4.2.4.2 Alignment and Field Map

Given that the start address of slice data memory (base address) is different for CPU and GPU, the pointers to the slice data in the host's main memory are invalid for the GPU. Therefore, the pointers have to be calculated twice, for the base addresses in host and in device memory. For a faster access the addresses of structures in the slice data memory are aligned. For different base addresses, this alignment could result in a different amount of bytes padded.[4] To ensure an equal padding, base addresses in host and device memory must have an alignment, greater than or equal to the largest alignment of the structures inside the slice data. Therefore, the base addresses are aligned to 64 kB.

As the magnetic field is not homogenous throughout the whole detector, tracking algorithms conventionally used a field map for the B-field. In general, this is a big lookup-table, and thus not suited for a GPU tracker at all. In [Gor$^+$3], it was shown that the magnetic field can be adequately approximated by a polynomial, making the field map obsolete. The work presented here could greatly benefit from the fact that the polynomial approximation was already implemented in the CPU tracker.

[4]Padded bytes are fake bytes appended to a data structure allowing the next structure to be aligned.

4.2.5 Classes

On CUDA versions 2.x, NVIDIA supports plain C with some extensions, e.g. templates. The compiler processes C++ source code with classes, but is not fully C++ compliant. One general problem arises with constructors. For the CPU tracker all the parameters in the tracker class are initialized by the constructor. But it is impossible to create a GPU tracker instance in global device memory (or constant memory) in such a way, that the constructor is automatically called and initializes the parameters. To solve this a trick is applied.

The tracker object is created in the host's main memory twice, one CPU tracker object and one GPU tracker object. The CPU instance is required for the CPU steps such as initialization. The constant pointers of the GPU tracker object (still in host memory) are set, to point to the correct location in device memory (making them currently invalid). Now the whole tracker object is copied bitwise[5] into constant device memory (where the pointers become valid). In this way, no changes to the CPU tracker code are required and classes can be used as usual.

4.3 First Run

Finally Fig. 4.9 shows the performance (for a peripheral lead-lead event) of the first working GPU implementation, compared to the CPU version. The results might look disappointing at a first glance, but admittedly a high end CPU was used and additionally, this was the very first attempt. Still, the plot shows that there is potential in the GPU tracker and suggests to optimize primarily the Tracklet Constructor. Before doing more benchmarks, the procedural methods for the measurements will be developed during the next chapter. This version is the basis for all the optimizations that will be applied later.

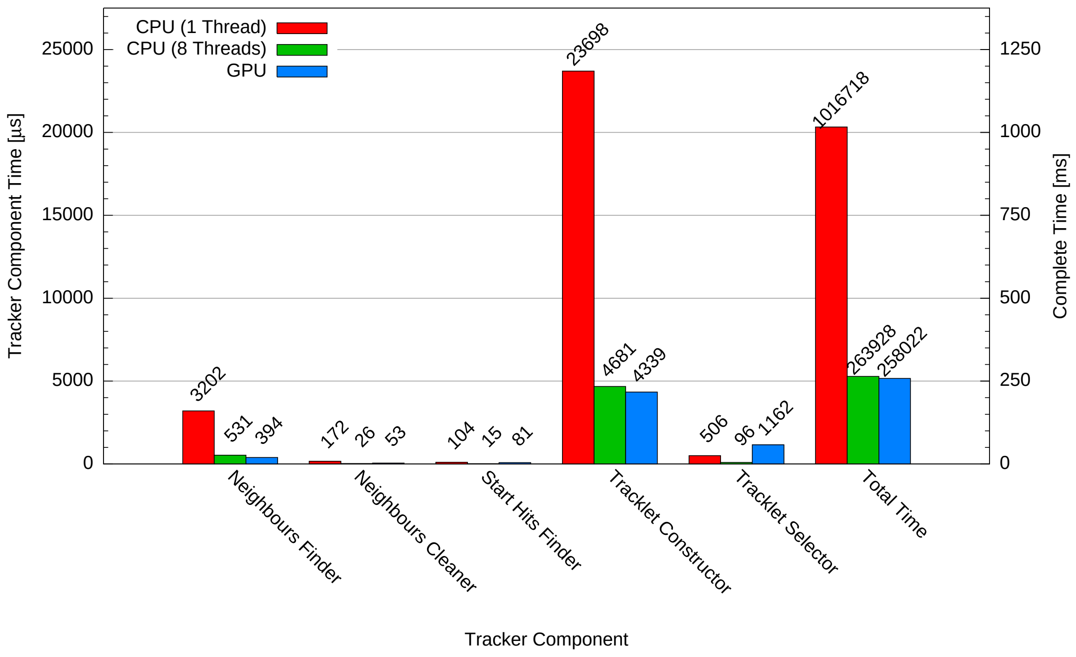


Figure 4.9: Performance of Initial GPU Implementation (Nehalem 3.8 GHz)

[5]The memory is simply copied, no copy constructor is called.

Chapter 5

Benchmarking

This chapter is intended to give an overview of the problems that appeared during the benchmarks and of ways to avoid them. Furthermore, some general information will be given, which might be interesting for an interpretation of the performance plots in the next chapters.

5.1 Raw Benchmarking Data

All benchmarks of tracking performance in this work were done using simulated data. This data was gathered using the AliRoot simulation framework. The framework can do Monte Carlo simulations of events in the ALICE experiment and outputs data in the same form as it is expected from the detectors such as the TPC. In addition it delivers the Monte Carlo source data of the simulation. Therefore, the track parameters of the Monte Carlo tracks are present, from which the simulated trajectories and TPC clusters are calculated. It is thus possible to compare the track parameters regained by the tracker to the initial ones to check the quality of the algorithm.

Since the GPU tracker is mainly aimed at tracking central heavy-ion events, most input data for the following benchmarks will use simulated central lead-lead collisions. To keep the results as comparable as possible, it was attempted to stick to one single central event of about $24,000$ tracks. This is the absolute worst case simulation with 100% centrality, resulting in the biggest event expected. In the experiment the most central events will be the most interesting ones and a tracker capable of tracking such events will be able to process less central events with sufficient performance anyway.

In some tests of different algorithms memory limitations on the GPU (usually the restriction to 16 kB of shared memory) prohibited tracking such big events. In such cases, a common heavy-ion event with reduced centrality and only $4,000$ tracks was used to benchmark an algorithm. If the algorithm turned out to be an improvement, it has to be considered how it could be applied on larger events. Otherwise it will be discontinued anyway. For an analysis of the dependency of tracking performance on event size a full range of events starting from pp events with and without pile up and going to peripheral and central heavy-ion collisions was used. The author would like to thank Sergey Gorbunov for the supply with a considerable amount of simulation data, which took several months to generate.

Because of changes in the tracking code and simulation framework, staying with the initial raw data was not always possible.[1] Updated events using the same simulation parameters resulted in different input clusters for the tracker. The event size was kept at $24,000$ tracks, but the events themselves used during this work were not identical.

5.2 Objectives

The GPU tracker algorithm is aimed at central heavy-ion collisions. The primary objective thus was the best performing GPU code on large events. In several cases GPU optimizations had a negative effect on CPU performance. Then two different codes were maintained to still allow for best CPU performance. To ensure a fair comparison also the CPU code was optimized as much as possible during this work. The thesis is intended to end with a CPU to GPU performance comparison on the newest and fastest hardware available at the time of its completion.

5.3 Benchmark Hardware

During development a wide range of machines was used, among others several Intel Core2 and Nehalem CPUs of different clock speeds. The GPUs employed were GTX280 / GTX285 / GTX295 cards as well as Tesla C1060 boards and Tesla S1070 computing systems. The final benchmarks for the most current tracker code were done on an Intel Nehalem i7-965 clocked at 3.8 GHz, 12 GB of DDR3-1600 memory and an EVGA GeForce GTX285 SSC.

5.4 Code Evolution

During this work the tracker code evolved. Updates improved the tracking efficiency, sometimes at the cost of some performance. (However, overall performance of the CPU code was improved dramatically during this work (see 9.2.1).) This makes it nearly impossible to compare benchmark results taken at different points in time. Many features were implemented for testing purpose only and were removed in code cleanups after the benchmarks were finished. Recapitulatory, regarding the following list, it would have been extremely difficult to do all the benchmarks on a common code base.

- Input data had to be restricted to small events for some measurements.
- Input data was resimulated several times.
- Updates to the tracker algorithm itself were included.
- No code base with all features implemented simultaneously exists.[2]
- Different features might affect each other.
- Different benchmarks were done on different hardware.

[1]In fact, the track parameters themselves should not have changed, but parameters like the magnetic field etc. changed its format. Also changes were made to the cluster finder, resulting in different clusters.

[2]E.g. the benchmarks testing feature A were done before the benchmarks for feature B, which in the meantime other optimizations were applied. In the code version containing feature B, feature A was already removed, as it hat turned out not to perform as expected.

- New CUDA versions were released.

For the above-mentioned reasons, every benchmark must be regarded standalone: Performance of the algorithms is compared within one plot, but different plots are not related to each other. For example, the absolute times between two plots can differ, since the benchmarks for one of them were done on a GPU with lower clock speed. Clearly all results within one figure were always collected on identical hardware (if not stated otherwise). Some plots compare CPU and GPU performance. These measurements were all performed on the same machine as the final benchmark, so the relation between CPU and GPU performance is kept constant. However, as different code versions were employed, the plots are still not comparable against each other.

5.5 CPU Cache Issues

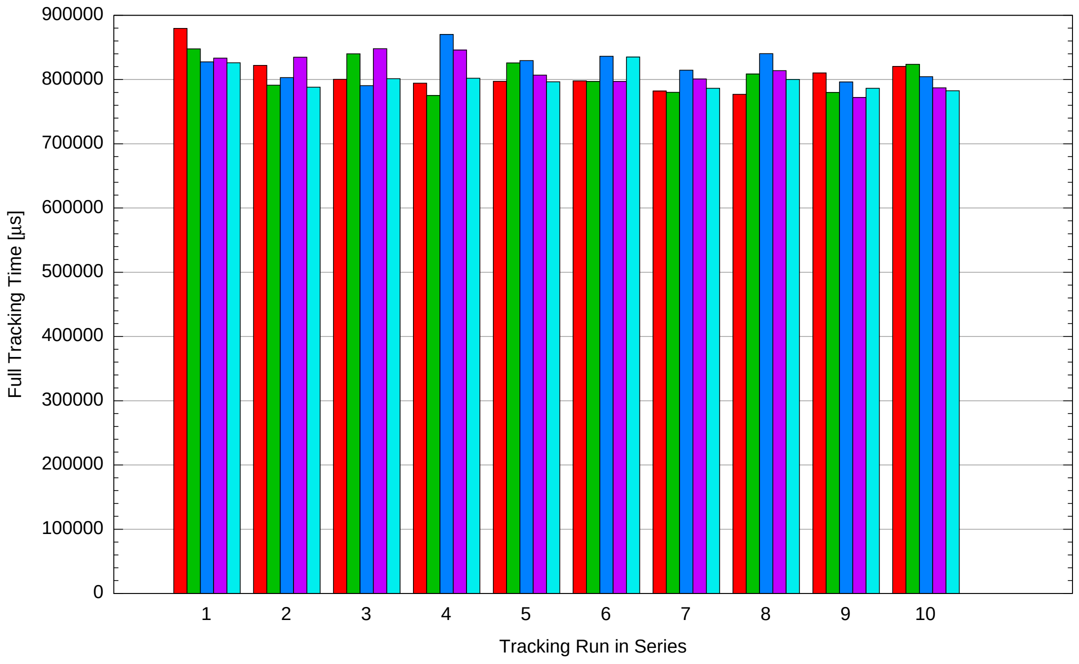


Figure 5.1: CPU Tracking Performance for Consecutive Reconstructions[3]

To allow for an unimpaired measurement, CPU cache effects must be considered. Fig. 5.1 shows the CPU tracking time for 5 measurement series of 10 consecutive tracking runs, each on identical input data sets.[3] Unrelated tasks were started in between the series to clear the CPU cache. It is obvious that the first run in each series is generally slower than the following ones, when the data is already in the cache. Tab. 5.2 shows the average times.

Every following benchmark was created by taking the average of a series of consecutive measurements. To account for the caching issue observed, the first tracking run in every series of measurements was discarded and not taken into account for building the average value.

[3]Each color corresponds to one series. For each series, the 10 runs were performed one immediately after another. The first (leftmost) runs of each series are in average slower than the following ones.

Run	Time [µs]	σ_{Time} [µs]
First	842860	8097
Cached	806476	3361

Table 5.2: Average CPU Reconstruction Time

5.6 Statistics

Before starting with the real benchmarks, in this section some statistical data about the distribution of tracking time is collected. It is known that by taking the average of n measurements, the statistical error is proportional to $\frac{1}{\sqrt{n}}$ if the measurements are uncorrelated. Definitely, this cannot be assumed here. Therefore, systematic errors will be analyzed and it will be attempted to eliminate their origins as far as possible. Anticipatory, this seems to work quite well for the GPU tracker, whose time distribution will resemble a gaussian, but not for the CPU tracker. In the end, to obtain the errors, the measurements will just be considered independent. Although this is not correct, the result can be used as an estimate, at least for the order of magnitude of the real errors. By averaging enough results both the estimated and the real error will become irrelevant.

Clearly the results will depend on the input data. Big events probably result in a lower relative error because the total tracking time increases. However, this work is targeted at central heavy-ion events and therefore, such an event will be principally analyzed here. As some smaller events will be regarded at a later stage, section 5.6.1 gives a summary on them.

Now the statistical measurements will be presented. The CPU and GPU tracking time for one identical event (central heavy-ion with 24.000 tracks) will be measured over and over again. According to this data it can then be decided how many measurements are needed for the statistical errors to become negligible. Figures 5.3 and 5.7 show a histogram of the distribution of tracking times on CPU and GPU respectively. All the data for the histograms in this section was exclusively collected on the Nehalem machine on which the final benchmarks were done (except for one figure showing a Core2 as comparison).

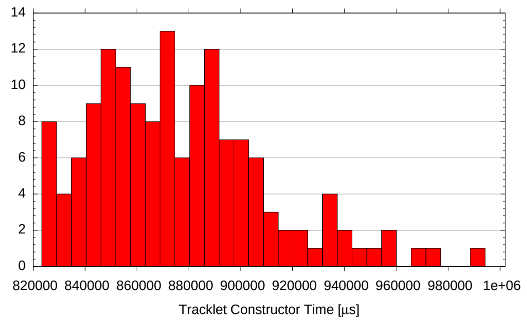


Figure 5.3: Distribution of CPU Tracking Time Using 8 Threads (Nehalem)

CPU Benchmarks One can see a strong variation in CPU reconstruction time distributed around 850 ms. The deviation results from scheduling inconsistencies between the runs. The plot was created from tracking runs with 8 threads in parallel. The distribution is much smaller for a single threaded run (see Figures 5.4 and 5.5), but this would not reflect the real application domain of the tracker. One can roughly see two peaks in the Core2 Quad plot, corresponding to the two different dies on the CPU, resulting in two pairs of cores with slightly different performance.[4] Employing the realtime FIFO scheduler and pinning the process to a single core yields a more precise measurement (Fig. 5.6).

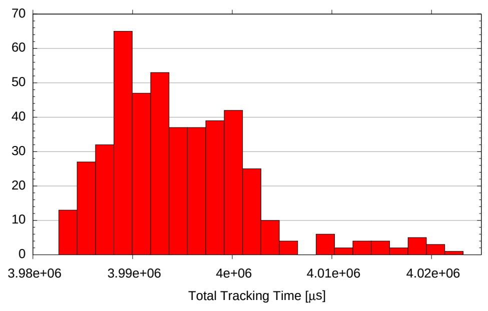


Figure 5.4: Distribution of CPU Tracking Time using One Thread (Nehalem)

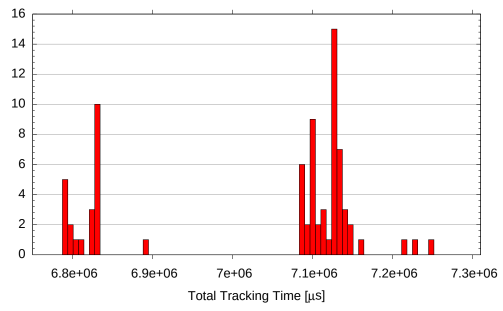


Figure 5.5: Distribution of CPU Tracking Time using One Thread (Core2 Quad)

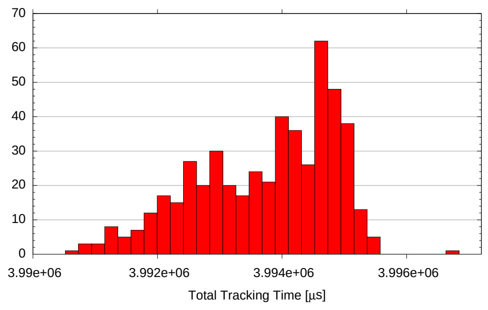


Figure 5.6: Time Distribution of CPU Tracker using FIFO-Scheduler with Pinned CPU-Core (Nehalem)

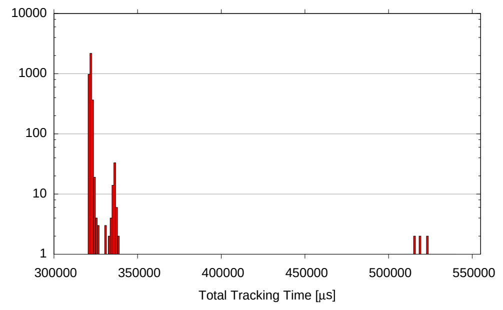


Figure 5.7: Distribution of GPU Tracking Time (Nehalem, GTX285)

Coming back to 8 threads, calculations show (Tab. 5.8) that for 50 runs the statistical error is below 1%. Therefore, for all intermediate benchmarks used to determine which of multiple variants of the algorithm was the fastest, doing statistics for 50 CPU runs is considered sufficient. It is evident that for the final benchmark a higher precision is desired. Benchmarks are run using the realtime FIFO scheduler. Obviously the tracker cannot be pinned to a single CPU core for more accurate results.

Measurements	Time [µs]	σ_{Time} [µs]	$\frac{\sigma_{\text{Time}}}{\text{Time}}$ [%]
150	$842,860$	$2,851$	0.324
50		$4,939$	0.561
1		$34,922$	3.967

Table 5.8: Statistical Errors for CPU Tracking Time

GPU Benchmarks The GPU plot (Fig. 5.7) is scaled logarithmically to show a wider range. A series of 150 measurements was used for the histogram. The peaks between $500,000$ µs and $550,000$ µs are produced by hard scheduling collisions (see 6.2.2.7). Since they only occur sporadically, produce unpredictable errors and render a detailed analysis of

[4]A Core2 Quad CPU consists of two dies. The two CPU cores within one die perform identically, but two cores from two different dies can have varying performance.

single steps of the tracking algorithm impossible, runs with hard collisions will be repeated for the benchmarks in the optimization chapters. Of course the final benchmarks include all runs, also those with hard collisions.

The main peak occurs at about 320, 000 µs and a smaller peak is visible at 350, 000 µs, though the second is suppressed by two orders of magnitude. To exclude the possibility that this is caused by the NVIDIA card being occupied by the operating system displaying graphics content, the test was rerun on a Tesla card, which has no video interface. The results are even more dramatic (Fig. 5.9). The smaller second peak does not occur when only measuring CUDA kernel execution times for a single tracking step (Fig. 5.10). Hence the delay must arise between the kernel launches or during the CPU parts of the algorithm. When running the tracker with the FIFO realtime scheduler and pinned to a fixed CPU core the small second peak does not appear (Fig. 5.11).

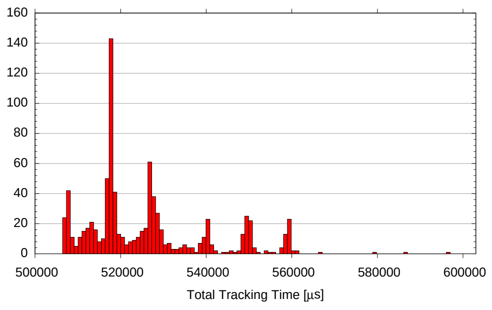


Figure 5.9: Distribution of GPU Tracking Time on a Tesla Card[5] (Core2 Quad, Tesla C1060)

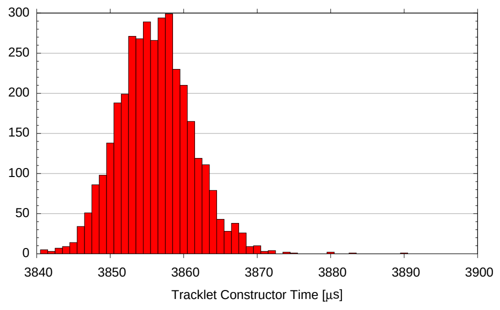


Figure 5.10: Distribution of Time Required by GPU Tracklet Constructor Step (Nehalem, GTX285)

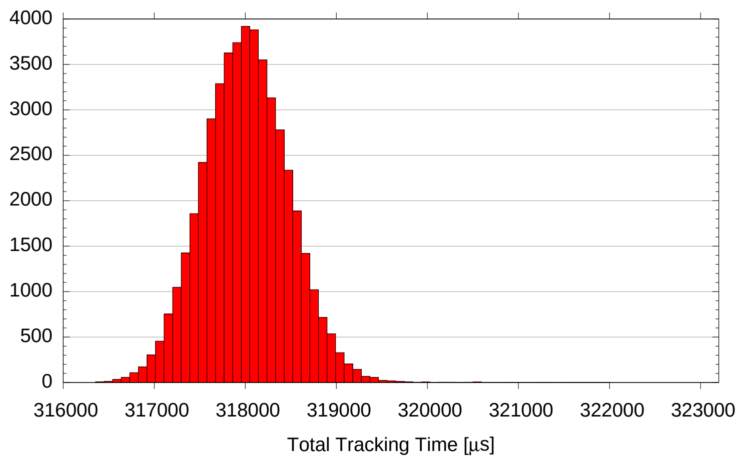


Figure 5.11: Time Distribution of GPU Tracker using FIFO-Scheduler with Pinned CPU-Core (Nehalem, GTX285)

For benchmarking the realtime FIFO scheduler is used, and contrary to the CPU case the GPU tracker is pinned to a CPU core (except for the multithreaded benchmark in section 6.3.2). This way both parasitic effects that were responsible for the additional peaks in Fig. 5.7 could be removed and the GPU tracking time distribution resembles a gaussian very

[5]The difference to the performance in Fig. 5.7 is slightly based on the different GPU clock speed but mostly on the Host performance, which is relevant for the CPU tracking steps.

well. Table 5.12 shows the statistical errors calculated for a full GPU tracking run, and for the Tracklet Constructor / Selector steps.

Type	Measurements	Time [µs]	σ_{Time} [µs]	$\frac{\sigma_{\text{Time}}}{\text{Time}}$ [%]
Full Run	48.287	322, 702	2	0.000
	50		66	0.021
	5		210	0.066
Tracklet Constructor	3, 601	3, 857	0.833	0.002
	50		0.707	0.018
	5		2.236	0.058
Tracklet Selector	3, 631	1, 480	0.1907	0.013
	50		1.625	0.110
	5		5.139	0.347

Table 5.12: Statistical Errors for GPU Tracking Time

One can see that the statistical error stays well below 1%, already for only 5 measurements. Every GPU result hereafter will be the average of at least 5 measurements, in most cases 10 or more. In fact, this means 5 / 10 runs over 36 slices, so at the least a total of 180 slices are processed. The error margin always remains below 1% and error bars are not included in diagrams if the are not necessary for facility of inspection.

5.6.1 Statistics for Small Events

For completeness it will be analyzed how the statistical error behaves for different track counts. Figures 5.13 to 5.15 show the distributions of tracking times for smaller events. As above the statistical error is calculated under the assumption that the measurements are independent. Tab. 5.16 shows the expected relative errors of the average values of 50 measurements in different cases. The measurements have a bigger error for a smaller track count. However, the order of magnitude of the errors does not change.

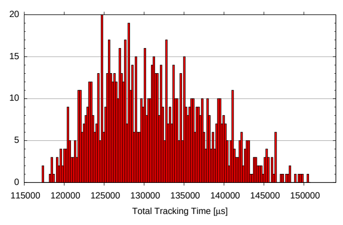


Figure 5.13: Time Distribution of CPU tracker (Peripheral lead-lead)

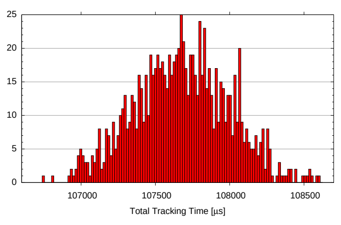


Figure 5.14: Time Distribution of GPU Tracker (Peripheral lead-lead)

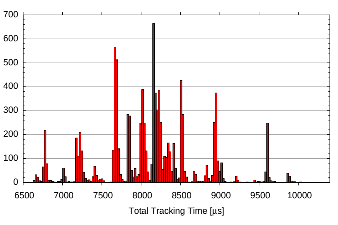


Figure 5.15: Time Distribution of CPU tracker (PP)

Track Count	$\frac{\sigma_{\text{Time}}}{\text{Time}}$ for CPU Tracker [%]	$\frac{\sigma_{\text{Time}}}{\text{Time}}$ for GPU Tracker [%]
24.000	0.561	0.021
4.000	0.731	0.042
223	1.100	

Table 5.16: Statistical Errors for GPU Tracking Time

Chapter 6

Optimizations for GPU

6.1 General Optimizations

This section examines two general aspects of GPU optimization which are, although applied in the tracker, of general concern. Another somewhat general tool is the texture cache, which is used in a very practical way and therefore discussed in section 6.3.1.2.

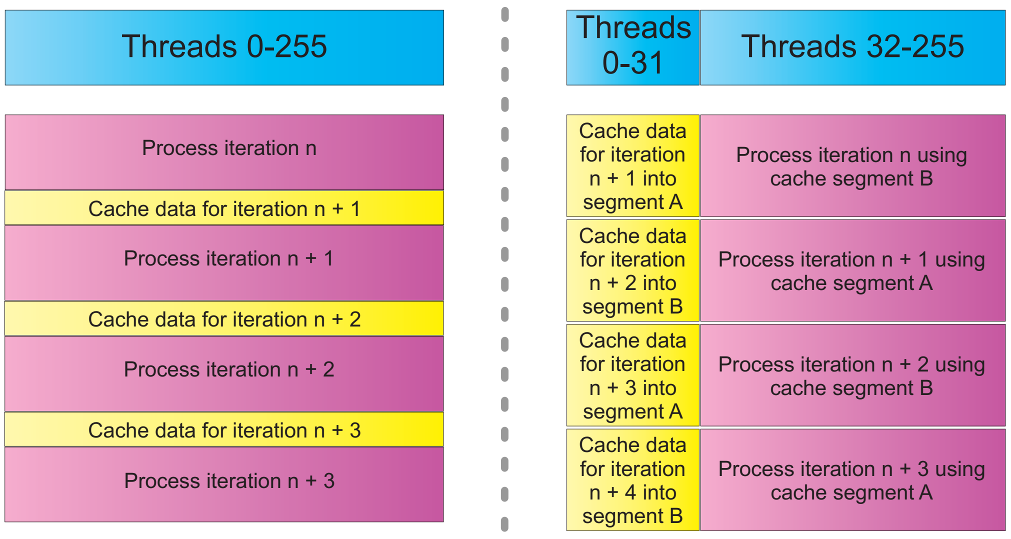
6.1.1 Shared Memory Transfer Performance

Shared Memory Caching Algorithms Since the global memory on the GPU is not cached, access to it is extremely expensive, even if the access is of streaming type or restricted to a small area. One general approach to overcoming this is to introduce a cache in shared memory. The shared memory cache will not be transparent, instead the data must be cached explicitly. This restricts the usage of a shared memory cache to cases where it can be predicted which data will be required in the near future, e.g. when processing a stream, or where access is restricted to a small range that fits into shared memory. Furthermore, the cache will be read only, since also the shared memory is not coherent when it is not synchronized. In general, there are two common approaches to implement a shared memory cache:

- a) In frequent intervals the algorithm is paused, and the shared memory cache is updated with new data from global memory.
- b) The shared memory cache is split into halves that are alternately and concurrently updated. The update is done by a small set of threads (usually one warp) while all the remaining threads (worker threads) process the actual algorithm, always using the cache halve which is not currently getting updated.

Both cases are illustrated in Fig. 6.1. As the shared memory is very small anyway, different cache sizes were not analyzed but only the biggest cache size possible.

Shared Memory Transfer Performance In the first case, it is easy to achieve the full global memory bandwidth during the caching step, when the algorithm is paused, since the full GPU is available. In the second case, on the one hand, using only a reasonably small amount of threads leaves more processing power available for the algorithm itself. On the other hand sufficient threads have to be used for the caching to be fast, so the worker threads



Threads 0-255
Process iteration n
Cache data for iteration n + 1
Process iteration n + 1
Cache data for iteration n + 2
Process iteration n + 2
Cache data for iteration n + 3
Process iteration n + 3
Threads 0-31
Threads 32-255
Cache data for iteration n + 1 into segment A
Process iteration n using cache segment B
Cache data for iteration n + 2 into segment B
Process iteration n + 1 using cache segment A
Cache data for iteration n + 3 into segment A
Process iteration n + 2 using cache segment B
Cache data for iteration n + 4 into segment B
Process iteration n + 3 using cache segment A

Figure 6.1: Illustration of Different Caching Algorithms

do not have to wait for the cache to be filled. Figures 6.2 and 6.3 show the time required for caching and the number of memory requests issued respectively, for different data types and thread counts.[1] The "short" data type is only present in this diagram, as short integers are widely used in the tracker, and at some points it might make sense not to cache a sequential area of global memory, but to gather[2] short integer values distributed in global memory. To copy a contiguous memory segment short integers should be unqualified, since the NVIDIA GPU cannot access data smaller than 32-bit.

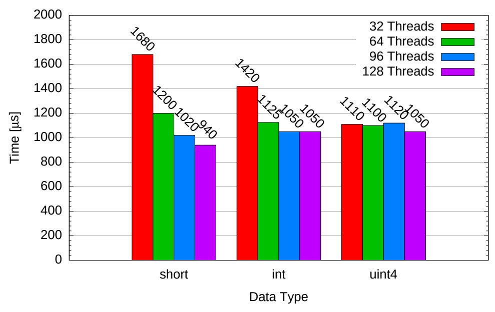


Figure 6.2: Time Required for Data Prefetching into Shared Memory Cache

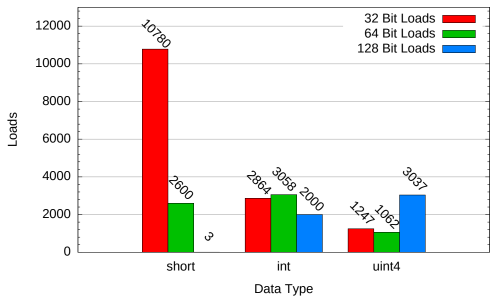


Figure 6.3: Load Operations Performed while Prefetching Data

Fig. 6.2 shows that for a high thread count the data type is irrelevant. This is evident since the usage of the entire GPU must provide the full memory bandwidth. For a big data type however the thread count gets irrelevant. Since employing less then 32 threads ($\equiv$ 1 warp) would unavoidably result in branching within one warp, it does not make any sense to drop below this number. Therefore, using 32 threads and the *uint4* data type is optimal. The

[1] The memory accesses can be counted using the NVIDIA CUDA profiler.

[2] The process of gathering is the transfer of data, distributed with a common stride or even randomly into a consecutive memory segment. It is the inverse process of scattering. Both of them are avoided wherever possible, since they to not reflect the architecture of DRAM well. Gather and scatter operations produce lots of short memory accesses whereas DRAM is optimized for the transfer of large continuous memory segments. See Appendix C for more information.

performance for this constellation is close to the maximum performance measured, while still most of the GPU is available for data processing.

Fig. 6.3 shows the number of 32-bit, 64-bit, and 128-bit load operations performed by the GPU for different data types. These numbers do not depend on the thread count. It goes without saying that the usage of bigger data types results in fewer but larger memory transactions. (Even though the number of 128-bit accesses increases, the sum decreases.) Considering this, it appears appropriate to use large data types.

Shared Memory Cache Alignment In general, CUDA assumes a memory access to a data structure of n bytes as being aligned to 2^k bytes, with k minimal such that $2^k \geq n$. When filling the shared memory cache using the *uint4* data type, all accesses must be aligned to 16 bytes, especially the source address in global memory. In fact, tests revealed that the cache contains corrupted data if one does not comply with this. For these and some other general performance reasons (the memory controller is optimized for 16 byte alignment), all data structures within the whole tracker are aligned to at least 16 bytes (Actually, benchmarks show a performance benefit of only about 0.5%, which is negligible. However, it comes at no cost and simplifies the remaining code, as alignment must no longer be considered).

6.1.2 Parallel Threads / Register Usage

Since each multiprocessor has a limited pool of registers, the number of threads it can execute in parallel is determined by the number of registers required by each thread. With $16,384$ registers in the GT200 chip this allows for the configurations in Tab. 6.4, limited by the product of the register requirement and the thread count which has to stay below $16,384$.

Registers	32	50	64	80	128
Threads	512	320	256	192	128

Table 6.4: Possible Register and Thread Configurations

If performance is not capped by other effects such as saturated memory bandwidth it is self-evident to run as many threads as possible. In contrast to that, restricting the register count limits the possibilities for compiler optimizations. It will possibly even lead to local variables being stored in local memory instead of registers. Therefore, it is always a trade-off: a register count that is too high will result in only few threads, while a low register count can result in excessive local memory usage.

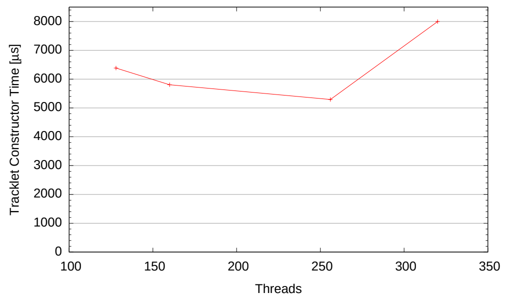


Figure 6.5: Tracklet Constructor Performance for Different Block Configurations

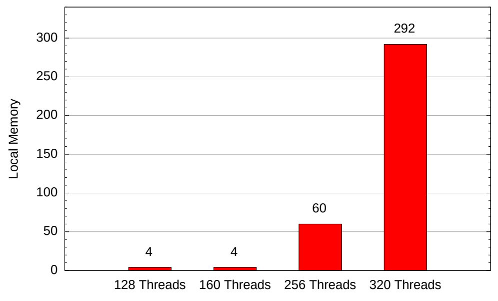


Figure 6.6: Local Memory Required by Tracklet Constructor

It can be seen from Fig. 6.6 that for 320 threads the amount of local memory increases tremendously. Simultaneously tracking time rises dramatically. However, when comparing thread counts of 256 and 160, local memory size is higher for the bigger thread count, but performance is superior. Therefore, 256 threads will be used throughout the Tracklet Constructor.

6.2 Optimizations of Tracking Steps

Contrary to the above examples, which are related to the tracker code but are very general, in this section, optimizations directly related to the tracker algorithm are analyzed. Out of the five tracking steps, three have non negligible runtime. These three steps are now subject to optimizations. Most effort was put into the Tracklet Constructor, which is the most complex and time consuming part, especially when starting the work (see Fig. 4.9). Some optimizations could be applied to the Neighbors Finder and the Tracklet Selector, while it would not make much sense to invest time in accelerating the remaining steps.

6.2.1 Neighbors Finder

As is clear from the design of the algorithm (see 2.3.1), the critical part of the Neighbors Finder is the calculation and the comparison of the slopes for the clusters in the rows above and below (Row above and below respectively refers to the rows $r + 2$ and $r - 2$, for the reasons stated in section 2.3.1). It is preferable not to calculate the slope for the same cluster twice, and especially not to read the data from global memory multiple times. Therefore, the algorithm works according to the following scheme:

- Calculate the slope for every cluster in row $r - 2$ and store the values in temporary memory.
- Iterate over all clusters in row $r + 2$, start with calculating the slope for one cluster C_+.
- Compare the slope for cluster C_+ with the slopes for all clusters in row $r - 2$.
- Proceed with the next cluster C_+.

In this way, the slopes are calculated only once, and even more importantly, cluster coordinates in global memory are accessed only once. Naturally the temporary data has to be stored in temporary memory. According to simulations the number of clusters in row $r - 2$ remains below 5 for pp as well as for peripheral lead-lead events, but can reach about 20 for central heavy-ion events. For each cluster, 10 bytes are needed. (Two single precision floats for the slope in y- and z-direction and one short integer for the cluster index in the row). It follows that a temporary storage for 20 clusters would require 60 registers (since registers can only save 32-bit values). For two reasons registers cannot be used as temporary storage:

- The storage requirement exceeds the register space.
- The data is organized as an array with dynamic access (which is incompatible with the register file).

The second point results from the design of the GPU register file. It is not possible to use one register as an index for an array of registers. This will be dealt with later (see 6.2.1.3).

6.2.1.1 Shared Memory

For a temporary storage this leaves only local memory and shared memory as alternatives. For performance reasons it is desirable to use as much shared memory as possible. A disadvantage is that every thread needs its own temporary storage leaving $\frac{16384}{256} = 64$ bytes per thread. This allows for storing slopes of 10 clusters in shared memory, leaving 1 kB left, which is required by the Neighbors Finder for other purposes.

As seen above, 6 clusters are insufficient for heavy-ion events, which are the main field of application of the GPU tracker. The Neighbors Finder will thus store the first 6 clusters in shared memory, and the remaining ones, if any, in local memory. Unfortunately, this complicates the implementation but is well worth the effort (see Fig. 6.8).

To measure the actual impact of the local memory on performance a trick is used. A shared memory only storage is implemented, allowing for a larger address space by taking the index modulo six before doing the actual memory access. In this way, cluster 6 will overwrite cluster 0, cluster 7 will overwrite cluster 1, etc. Of course the result will be wrong, but this overflow will not change the algorithm's (the Neighbors Finder part of it) runtime. Fig. 6.8 shows the performance for the three versions discussed here.

It can be deduced that the mixed storage performs well. As assumed the shared memory only version is even faster. Since the next NVIDIA GPU generation will offer three times the amount of shared memory, it will then be possible to implement a working Neighbors Finder using only shared memory.

One last interesting fact to note is that the number of tracks found by the tracker is only very slightly affected when using the overflow variant. The reason is that it is merely necessary to find one part of a track in the Neighbors Finder. This is obviously already fulfilled in the overflow variant when roughly every third cluster is taken into account. Still, tracking quality is more important than performance, but it would very well be possible to restrict to a part of the clusters when performance was an imminent issue. This would even speed up the Neighbors Finder further, since the overflow implementation used still processes every cluster but then neglects 2 out of 3 clusters afterwards. Also the tracklet constructor could perform faster when fewer seeds exist.

6.2.1.2 Alternative Approach

When comparing the runtimes to the tasks for the Tracklet Constructor and Neighbors Finder (final result on Fig. 9.20), it is noticed that the Neighbors Finder takes more than half of the time the Tracklet Constructor requires, even though it performs only a simple combinatorial task. Therefore, the Neighbors Finding algorithm seems suboptimal. On top of that all steps in the whole tracking algorithm have linear runtime, except for the Neighbors Finder, which does an all to all comparison resulting in quadratic runtime. It would be desirable to come up with an entirely linear algorithm.

In a first simplification, a two-dimensional Neighbors Finding algorithm is discussed. The y-axis is ignored, but only the row (x-coordinate) and the z-coordinate within the row is taken into account. In this case an algorithm with a perfectly linear runtime is possible, delivering the optimal result. Let the clusters be sorted according to z-coordinates within the rows, with n clusters C_1 to C_n in row $r - 2$ and m clusters D_1 to D_m in row $r + 2$, and C_0 the reference cluster in row r. The algorithm starts calculating the slopes for C_1 and D_m. Now, given the current position is (C_i, D_j) slopes for C_{i+1} and D_{j-1} are calculated. Now the position is changed to $(C_{i+1},\ D_j)$ or $(C_i,\ D_{j-1})$, namely the pair that forms the better straight line with C_0. The algorithm is continued until the position $(C_n,\ D_1)$ is reached. Clearly all pairs of clusters $(C_i,\ D_j)$ not regarded during the algorithm have a greater slope difference than a pair of clusters that was considered, and so this algorithm will find the optimal pair. For completeness a proof is given. The problem is clearly equivalent to the one solved in the following

Lemma 6 *Let M, N be sets of sorted real numbers, $M = \{a_1, a_2, ..., a_m\}$, $N = \{b_1, b_2, ..., b_n\}$, $a_i \leq a_{i+1}$, $b_i \leq b_{i+1}$. Consider the sequences of numbers defined by:*

$$\begin{aligned} i_1 &= 1 \\ j_1 &= 1 \\ i_{k+1} &= \begin{cases} i_k + 1 & \text{if } |a_{i_k+1} - b_{j_k}| \leq |a_{i_k} - b_{j_k+1}| \wedge i_k < m \\ i_k & \text{otherwise} \end{cases} \\ j_{k+1} &= \begin{cases} j_k & \text{if } |a_{i_k+1} - b_{j_k}| \leq |a_{i_k} - b_{j_k+1}| \wedge i_k < m \\ j_k + 1 & \text{otherwise} \end{cases} \end{aligned}$$

It then follows that

$$\exists k \text{ s.t. } |a_{i_k} - b_{j_k}| = \min_{0 \leq i \leq m, 0 \leq j \leq n} (|a_i - b_j|)$$

Proof: Let $|a_i - b_j|$ be minimal for $i = i_0$, $j = j_0$. Clearly $\exists k$ s.t. $i_k = i_0$. W.l.o.g. assume $j_k < j_0$, otherwise swap M and N. It follows $b_{j_k} \leq b_{j_0}$ but also $b_{j_k} \leq a_0$, otherwise $a_0 < b_{j_k} \leq b_{j_0}$ but then $|a_{i_0} - b_{j_0}|$ would not be minimal. $\lightning$
Now assume first $b_{j_k} < b_{j_k+1} \leq a_{i_k}$: $\Rightarrow |a_{i_k} - b_{j_k+1}| < |a_{i_k+1} - b_{j_k}| \Rightarrow i_{k+1} = i_k \wedge j_{k+1} = j_k + 1$.
Alternatively $b_{j_k} \leq a_{i_k} \leq b_{j_k+1}$, but then clearly $j_k = j_0 \vee j_k + 1 = j_0$.
Per induction it follows after finitely many steps that: $i_{k+l} = i_k = i_0 \wedge j_{k+l} = j_0$, which completes the proof. $\square$

The algorithm is illustrated in Fig. 6.7.

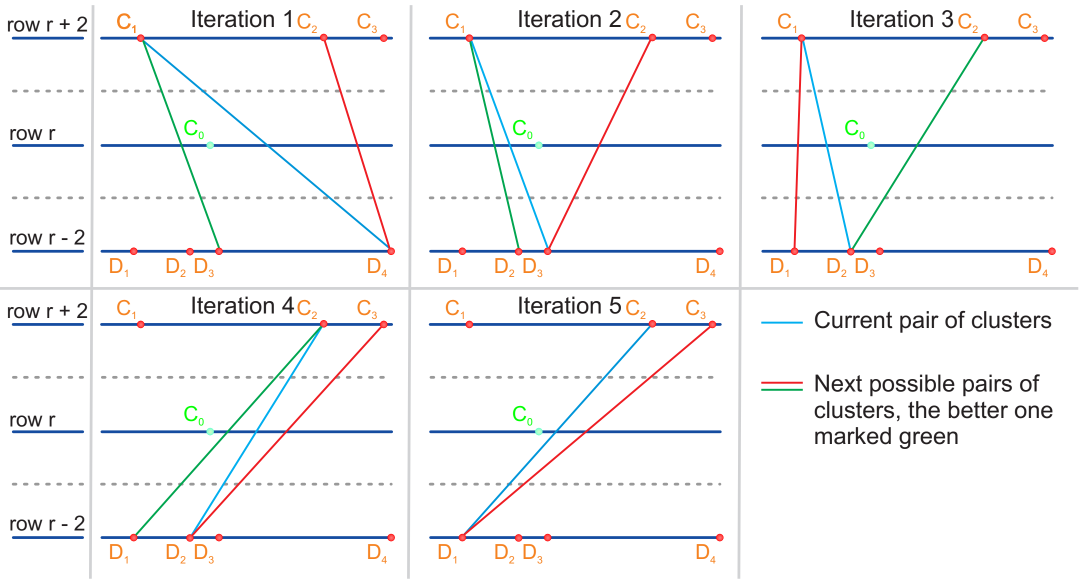


Figure 6.7: Illustration of Fast 2D Neighbors Finder Algorithm

Clearly this is not so simple for three dimensions. Due to the magnetic field, however, the trajectory in y-direction is not even a straight line, while it is very well in z-direction. This makes a straight line fit in z-direction more urgent than in y-direction. As a first fast variant, the clusters are therefore sorted only according to their z direction, and the algorithm above is applied as in the two-dimensional case. Naturally for the actual slope calculation the y- and z-coordinates are used. Clearly it is not ensured that this algorithm will find the best pair of clusters, in fact it is not even probable that it will. Fig. 6.9 shows the performance of the algorithm. As it will be needed for comparison, the algorithm with shared memory overflow is included, too.

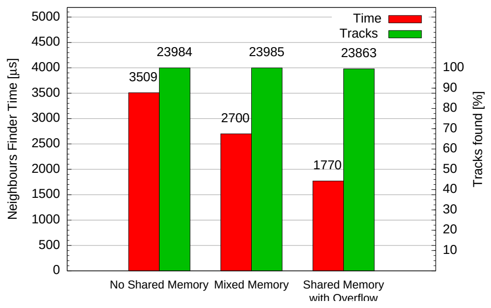


Figure 6.8: Performance of Neighbors Finder for Different Shared Memory Configurations

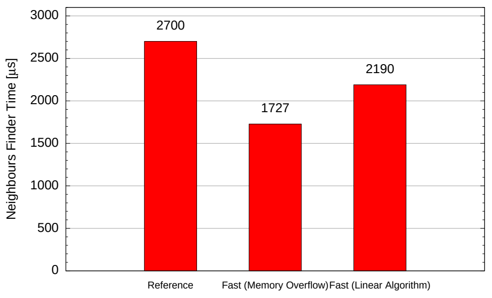


Figure 6.9: Performance of Neighbors Finder Algorithms

Unfortunately, but understandably this algorithm misses a lot of tracks. Tests result in a tracking efficiency[3] of 50% or less, which is infeasible. Since the test algorithm producing memory overflow is even faster than this naive linear implementation, but still delivers a comparably good efficiency, it renders the newly developed linear algorithm useless. It is highly improbable that the algorithm can be improved to beat the efficiency of the overflow implementation. In addition, any improvement to the combinatorics would increase the complexity and decrease the performance. The discussion shows, that the current implementation is not as bad as initially assumed. Therefore, no further attempt to create a different algorithm was started, but instead the existent algorithm was optimized.

There is one additional fact to consider. As stated above, the straight line fit in z direction should theoretically be more important than for the y coordinate. Thus it is imaginable that tracking efficiency could be improved by weighting the y- and z-slopes in the minimization. This was attempted by Sergey Gorbunov already and it turned out that tracking efficiency did not depend on this weights. Hence, for the remaining section, more optimizations to the original Neighbors Finder algorithm will be discussed.

6.2.1.3 Dynamic Register Access

Using registers as temporary storage would require three registers per cluster, one to store the cluster ID, one to store the slope in y-direction and one for the slope in z-direction. Storage for 20 clusters hence would require 60 registers, using registers 0 to 19 for cluster indices, 20 to 39 for y-slopes and 40 to 59 for z-slopes. When initially filling the temporary storage a loop is iterated writing to the i^{th} (and $(20+i)^{\text{th}}$, $(40+i)^{\text{th}}$) register until no more clusters are found in the hit-area. Now the index i is a variable itself, stored in a register. It is not possible to use a register as index for an access to the register file. In other words: The i^{th}

[3]Percentage of reference tracks in the Monte Carlo data found by the tracker.

register cannot be accessed dynamically. Instead, the index of the register to be accessed must be known at compile time. There are two possibilities to realize a dynamic access:

Branching Dynamic register access can be emulated using a binary branching tree. For every possible index a different instruction is needed in the code, and by branching according to the desired index the correct instruction is chosen and executed. Of course this produces much overhead and results in warp serializations.

Unrolling A loop with a fixed number of iterations can be unrolled[4]. In this way, no branching is required. Naturally the overhead introduced hereby is even bigger.

Listings 6.10, 6.11 and 6.12 show examples in C++.

```
for (i = 0;i < 4;i++)
{
    register[i] = FetchCluster(i);
}
```

Listing 6.10: Real dynamic access

```
#pragma unroll loop
for (i = 0;i < 4;i++)
{
    register[i] = FetchCluster(i);
}
```

Listing 6.11: Access by unrolling

```
for (i = 0;i < 4;i++) {
    tmp = FetchCluster(i);
    if (i < 2) {
        if (i == 0) {
            register[0] = tmp;
        } else {
            register[1] = tmp;
        }
    } else {
        if (i == 2) {
            register[2] = tmp;
        } else {
            register[3] = tmp;
        }
    }
}
```

Listing 6.12: Access by branching

In theory, List. 6.11 differs from List. 6.12 in the following way: The unrolling variant will unroll the loop and inline the *FetchCluster* function, which will be present four times. This greatly increases the complexity. If branching is used, the function is only present ones, but the binary tree increases the complexity.

Finally, out of these examples only List. 6.12 resulted in register usage for the temporary memory by the compiler. The problem with the loop in List. 6.11 is that it cannot easily be unrolled. This is due to the fact that, in contrast to the example presented here, the number of clusters is not known ahead of time. Running the loop up to a maximum number and branching out using a *break* statement does not work because the compiler will not unroll such loops. Moreover, the compiler will always unroll the innermost loop first. But since

[4]Unrolling means the loop is replaced by n times the loop body, with the loop variable i replaced by the fixed value for the i^{th} iteration

all function calls are inlined, the compiler will attempt to unroll loops in the *FetchCluster* function, which is difficult as those loops are more complex. The only possible solution here is manual loop unrolling.

Both manual unrolling and branching were realized for the Neighbors Finder. However, both turned out to perform worse than the shared memory implementation. In a final attempt a mixed implementation was tested, storing clusters in shared memory, registers and local memory (using branching for the register access). But again the shared memory only version was faster. One reason for this might be the small number of registers left. The Neighbors Finder in reference implementation uses 51 registers. When restricting the kernel to 64 registers (resulting in 256 threads), this leaves 13 registers for temporary storage, which is sufficient for only 4 clusters.

Given that the amount of shared memory available will increase undoubtedly with the next GPU generation, this approach was not pursued any further.

6.2.1.4 Shared Memory Data Types

The current implementation reads the cluster coordinates represented by short integers, calculates the slopes represented by single precision floats and stores the result in temporary memory. For a better memory utilization it is imaginable to store 16-bit short integer cluster coordinates in shared memory instead of 32-bit floating point slopes. That would double the capacity with the drawback that the slope would have to be calculated multiple times. This would reflect the modern programming paradigm to rather recalculate expressions instead of using lookup tables. An adequate version of the Neighbors Finder was implemented, but was again slower than the original version. To understand the problems, it is necessary to inspect the assembler code of both versions in Listings 6.13 and 6.14. (See Appendix A for an introduction to CUDA assembler code)

```
//Load cluster coordinates
ld.global.v2.u16 {%r1,%r2}, [%rd1];
cvt.rn.f32.u32    %f1, %r1;
cvt.rn.f32.u32    %f2, %r2;

..... //Calculate the slope

//Store slope in shared memory
st.shared.f32     [%rd2+48], %f3;
st.shared.f32     [%rd2+52], %f4;

.....

//Load slope from shared memory
ld.shared.f32     [%rd3+48], %f5
ld.shared.f32     [%rd3+52], %f6
```

Listing 6.13: Disassembly for float storage

```
//Copy cluster coordinates
//    to shared memory
ld.global.u32     %r1, [%rd1];
st.shared.u32     [%rd2], %r1;

.....

//Load coordinates
//    from shared memory
ld.shared.u32     %r2, [%rd3];
st.local.u32      [%rd4], %r2;
ld.local.v2.u16   {%r3,%r4}, [%rd4]

..... //Calculate the slope
```

Listing 6.14: Disassembly for short storage

The problem stems from the way in which the two short integers are read from shared memory. The GPU has special instructions to read vectors of up to four entries from global memory. Instead of using two short integer fetches from shared memory, the compiler copies the data to local memory and uses a vector fetch. Obviously this causes a read-after-write delay to the local memory, which is very slow even without that.

This is clearly a compiler problem. Using explicit bit shift operations to access the desired parts of the 32-bit dword would be a solution. However, this issue might be solved with new compiler versions and the changes would make the code more complex and reduce its readability. For the moment the tracker sticks with the storage of floating point values.

6.2.2 Tracklet Constructor

A first canonical GPU version of the Tracklet Constructor processed the tracklets in parallel in a way that a separate thread was started for every tracklet. Within one block only one single row was processed at a time (block based row synchronicity). Threads processing tracklets without clusters in the current row therefore had to wait for the remaining threads. The first analysis will regard this naive implementation.

6.2.2.1 GPU Utilization

In the end the most critical design issue for the Tracklet Constructor turned out to be a good overall GPU Utilization. Canonical implementations resulted in lots of warp serialization and dead threads. For easy analysis of this issue, a tool to visualize the GPU utilization was developed. The output for the first canonical implementation is shown in Fig. 6.15. For a more comprehensive view, an augmented output is shown in Fig 6.16.

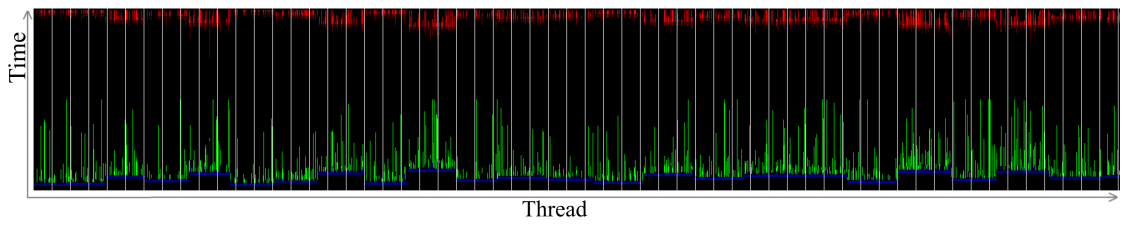


Figure 6.15: GPU Utilization during Tracklet Construction for Unsorted Start Hits

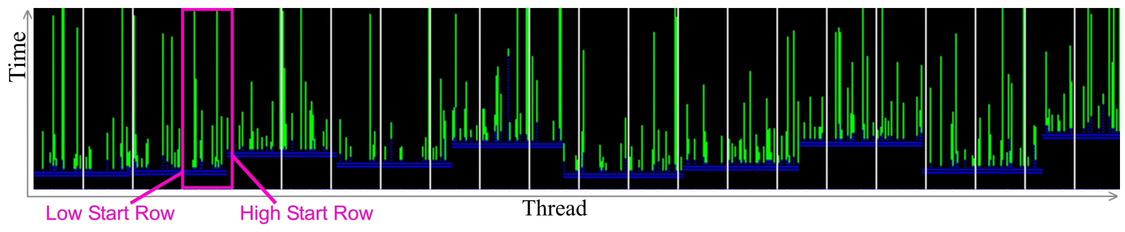


Figure 6.16: GPU Utilization during Tracklet Construction for Unsorted Start Hits

The GPU utilizations plots (Figures 6.15 and 6.16) show the different threads on the x-axis, where each thread is represented by one pixel. The warps of 32 threads are separated by white vertical lines. The y axis corresponds to iteration steps, where each step corresponds to the treatment of one row. This can roughly be seen as the time axis. However, different blocks do not even have to be executed in the same time. Since threads within one warp have a common instruction decoder in between of two white lines the y-axis perfectly corresponds to the time.

The diagram is created in the following way: A pixmap is allocated in GPU memory. Each thread has a register counting the iteration steps. For each step the thread writes its current task to the pixmap. The y-coordinate equals the counter while the x-coordinate can be calculated from *threadIdx*, *blockDim*, and *blockIdx*. This procedure is minimal invasive, as the register was not used before and the number of memory accesses is insignificant. Tracklet Construction time with and without the visualization feature differ by less than 1%.

A particular tracklet is unlikely to pass through each row within one slice. It is more likely it possesses clusters in a fairly consecutive sequence of rows, say from row n to row m, but none outside that interval. Tracklets are called **inactive** outside the rows between n and m.

In the current version each tracklet is processed both up- and downwards through all rows, regardless whether it is active or not. Therefore, in Fig. Figures 6.15, the y axis corresponds directly to the row. (This will change when a scheduler is introduced later.)

The color codes represent different steps during track reconstruction. The color codes stand for:

- Black: Thread Idling
- Blue: Track Fit [IV (a)]
- Green: Forward Extrapolation [IV (b)]
- Red: Backward Extrapolation[IV (c)]

In the first implementation, every thread processed a tracklet for the entire row interval. Therefore, the starting point of the actual processing, indicated by the blue color, is different for the threads.

Obviously black pixels in the diagram are not desirable. However, the interpretation is not that easy. If all threads of one warp are idling, this means that none of the tracklets processed by these threads is active. This does not lead to a problem since all threads will just skip the row in common so no warp serialization will occur. The absolute worst case scenario occurs when one single tracklet within one warp is active while the remaining 31 tracklets are inactive. In that case 31 threads will branch and skip the row, hence waiting for the one single thread processing the row. The inherent problem here is that tracklets have different lengths. The best solution would be to sort tracklets by length, which is obviously not possible in advance, since tracking must have happened prior to this.

The Start Hits Finder does not guarantee that the seeds are ordered by their start row. Warp 4 in Fig. 6.16 contains threads with start hits in different rows. Threads with a high numbered start row (start row of start hit) have to wait for the threads with low numbered ones, until the latter processed their tracklets up to the high start row. This can be avoided by sorting the start hits according to rows.

6.2.2.2 Start Hit Sorting

Simple Sorting As an advancement to the first naive implementation a new tracking step was introduced between the Start Hits Finder and the Tracklet Constructor: The Start Hits Sorter (This step cannot easily be integrated in the Start Hits Finder, as for efficient sorting all start hits must be known in advance). The GPU Utilization for the improved implementation is shown in Fig. 6.17.

It is obvious that the problem described above no longer occurs in Fig. 6.17. The performance for both implementations is shown in Fig. 6.18.

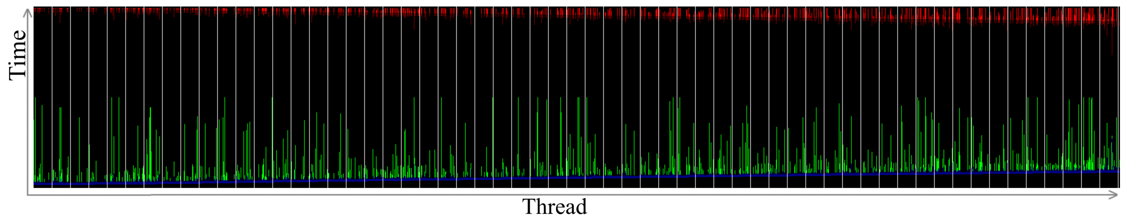


Figure 6.17: GPU Utilization during Tracklet Construction for Sorted Start Hits

Advanced Sorting An inspection of the utilization plot 6.16 reveals that each tracklet is only processed within every other row during the track fit stage (IV (a)). In fact, this is clear from the algorithm, as the Neighbors Finder skips one row. As a result, as long as tracklets are in this first stage, either tracklets with even start row or those with odd start rows are processed, with the other threads idling. An advanced sorting algorithm was implemented which accounted for this by grouping tracklets with odd and even start rows respectively. In the end the Tracklet Constructor performance increased by only 1%, which was nullified by the more complex Start Hits Sorter. Having no effect on total tracking time, this type of sorting is not used anymore.

6.2.2.3 Shared Cache

In the extrapolation step the Tracklet Constructor performs a more or less random search in global memory. An obvious solution is the introduction of a shared memory cache. Variant b) from section 6.1.1 was used for this purpose. While threads 32 to 256 process the tracklets in row n, threads 0 to 31 prefetch the cluster coordinates and grid content for row $n+1$. Due to the limited shared memory, this is only possible for heavy-ion events with reduced centrality.

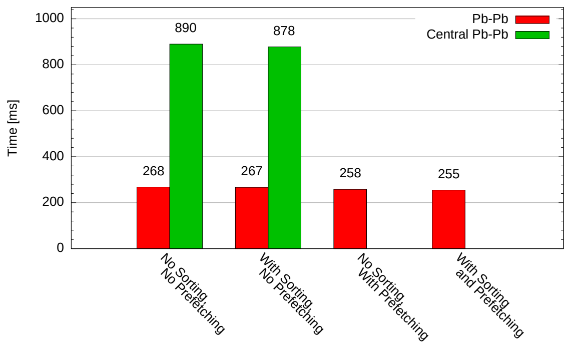


Figure 6.18: Performance Comparison for Shared Memory Cache and Sorted Start Hits

Fig. 6.18 shows that the optimizations result in only a small speedup. Variants with and without shared cache and sorting respectively are shown. For those without a shared cache a 100% central event is included, too. The highest possible speed gain by a shared memory cache can be determined in the following way: The algorithm was changed so that the worker threads 32 to 255 idle while the cache threads 0 to 31 cache data for the next iteration and vice versa. Using GPU internal counters the accumulated time required by the worker threads is measured. In this way, the pure calculation time can be determined. Measurements by Sergey Gorbunov showed that the accumulated calculation time does not depend on whether the caching is done simultaneously or not. It can therefore be concluded that the caching itself works optimal.

Regarding the shared memory cache, there is one factor yet unconsidered. Each block processes a fixed amount of tracklets, so only a restricted amount of clusters belong to tracklets handled by one block. With an increasing overall cluster count for more central events, the size of the data to be transferred into the shared memory cache increases, while the number of cluster coordinates actually accessed remains constant. This results from the fact that the bin size of the grid scales with the number of clusters. Therefore, the shared memory cache should perform better for small events, whereas the GPU tracker is developed for the biggest events possible. For the time being, because of limited memory the shared memory cache feature will not be used. In contrast to that the Start Hit Sorting feature is maintained. It improves the performance only slightly but comes at zero penalty and will even be required for upcoming improvements.

6.2.2.4 Scheduling

As a matter of fact Tracklet Constructor performance stands or falls with the GPU utilization. Therefore, an implementation where most threads are dead waiting for the one processing the longest tracklet is not acceptable. Moreover, in the first version every block acts on a precisely defined set of tracklets. Thus it can happen that multiprocessors are idling until other multiprocessors finished their remaining blocks. The only possible solution for this is to drop the fixed assignment of tracklets to threads. Only the active tracklets should be processed and redistributed among multiprocessors and threads.

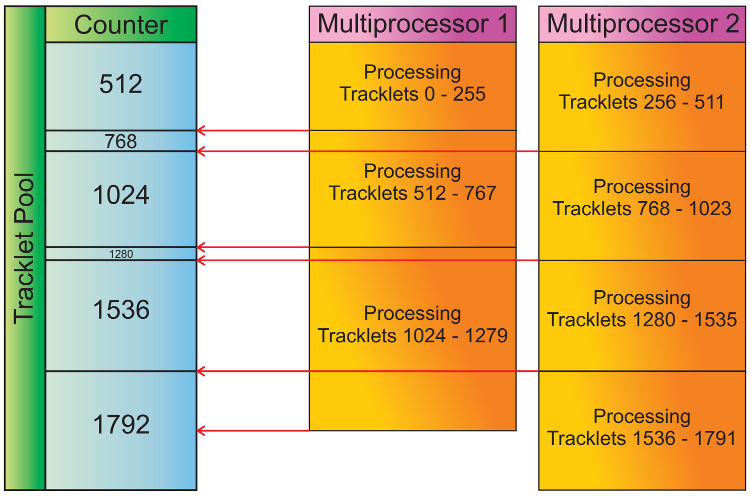
With this final goal in mind, dynamic scheduling was integrated in the Tracklet Constructor step by step. Since for the CPU Tracklet Constructor no rescheduling is needed it seems better to maintain two different versions: one for CPU and one for GPU. The main Tracklet Constructor algorithm (contained in the *UpdateTracklet* function) with the extrapolation and track fit should still remain in shared source code and then be called by the two wrappers for CPU and GPU. Since two versions have to be maintained anyway, the CPU code can easily be optimized, too. Compared to the other steps I to III and V, the GPU adaptation of the Tracklet Constructor produced more overhead because of the shared memory caching and the row synchronous processing of tracklets within one block. Removing all this in a specialized CPU version resulted in a speedup of about 5%.

As a first step towards integrating a scheduling algorithm into the Tracklet Constructor, the execution configuration must be changed. Until now the number of blocks has been proportional to the number of tracklets to be processed and the GPU internal scheduler has distributed them among the multiprocessors. For the new configuration a fixed block count equal to the amount of multiprocessors is desired. Then all blocks in the configuration should run in parallel and can schedule the work by themselves. For the remaining scheduling section a block corresponds exactly to one multiprocessor.

Naive Scheduling In a first naive implementation a configuration with 30 blocks is executed, with $30 \cdot 256 = 7680$ threads. Scheduling is fixed in the way that thread i processes all tracklets n with $i \equiv n \pmod{7680}$. This is clearly the simplest scheduling imaginable, but it delivers a basis to implement scheduling algorithms. The tracking time increased by a factor of 1.234 in comparison to the initial version (see Fig. 6.41).

Improved Scheduling As a first improved algorithm a **tracklet pool** is integrated. A counter is initialized to zero. Every multiprocessor (or every block, precisely the first thread in it) issues an atomic add instruction, increasing the counter by 256 (the number of threads running on this multiprocessor). The instruction returns the counter value n before the addition, after which thread i processes tracklet $n + i$ (of course only as long as $n + i$ is smaller than the number of tracklets. Synchronization within the block is done using shared memory). When all threads have finished their tracklet, the procedure is repeated until the value returned by the counter exceeds the number of tracklets. Fig. 6.41 shows that this first improved scheduling algorithm already performs faster than the initial implementation.

Three more optimizations can be applied to this algorithm. It turns out that a warp based row synchronous processing of the tracklets performs better than no row synchronicity. (Block based row synchronicity would be required if a shared row data cache should be reintegrated at a later stage.) Furthermore, the initial tracklet for each thread does not have to be determined by the scheduling algorithm, but a thread i in block b can start with tracklet $256 \cdot b + i$, with the counter initialized to $7,680$. In this way, each thread has a fixed start tracklet, and only the remaining tracklets are scheduled. Fig. 6.19 shows a simplified example with reduced multiprocessor and tracklet count. Finally, when including Start Hit Sorting as described in section 6.2.2.2 the tracking time is reduced to only 85.7% of the initial tracking time with GPU integrated scheduling.



Tracklet Pool
Counter
512
768
1024
1280
1536
1792
Multiprocessor 1
Processing Tracklets 0 - 255
Processing Tracklets 512 - 767
Processing Tracklets 1024 - 1279
Multiprocessor 2
Processing Tracklets 256 - 511
Processing Tracklets 768 - 1023
Processing Tracklets 1280 - 1535
Processing Tracklets 1536 - 1791

Figure 6.19: Illustration of Tracklet Pool Principle

There are clearly still threads idle after they have finished their tracklet before other threads on the same multiprocessor. This would not happen if every thread would issue its own atomic add instruction, but then the warps would serialize frequently. Supplementary atomic add instructions are in principle slow, and this variant would require 256 times as many of them. As a second variant every warp could issue an atomic add. This would preclude warp serializations, but then another problem arises: The one thread within the warp that accessed the tracklet pool must share the information with other threads. This could be done using shared memory. However, to ensure consistency a *__syncthread* instruction must be issued, but that would sync the whole block. An algorithm following this principle will be presented in section 6.2.2.13.

6.2.2.5 Dynamic Scheduling

Until now, the problem with dead threads waiting for other threads to finish a tracklet has not even been approached. The only possibility to accomplish this is to interrupt the row synchronous processing in between. Tracklets whose extrapolation is finished are replaced by new ones. In this way, dead threads do not have to wait for the last tracklet, but only for the next interrupt. The procedure will now be described in detail.

The rows are split in rowblocks of constant size. The former tracklet pool is divided into multiple pools, precisely two pools per rowblock where one pool stores tracklets in the upward extrapolation stage and the second one those in the downward stage. A temporary storage space in global memory is reserved, containing an array that will store track parameters for each tracklet. An algorithm step for one block works as follows:

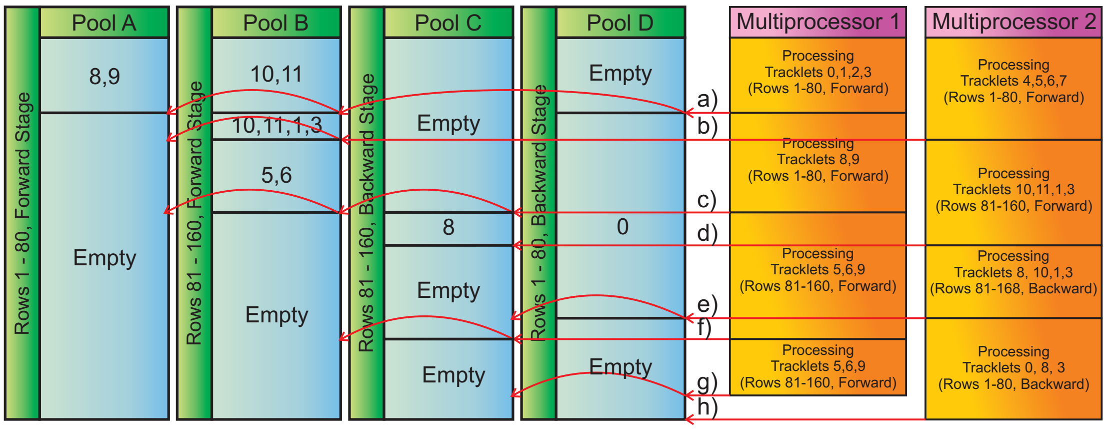
First a rowblock is chosen and it is decided whether to extrapolate up or downwards. Thread 0 fetches 256 tracklets n to $n + 255$ from the corresponding tracklet pool. Threads are synchronized via shared memory so thread i processes tracklet $n + i$. Now each thread initializes its local track parameters. If the tracklet was processed within another rowblock before, track parameters are copied from global memory, otherwise they are initialized with default values. Then the Tracklet Constructor algorithm iterates over all rows within that rowblock. Tracklets that are still active afterwards must be inserted into the following rowblock and the track parameters stored to global memory. If the tracklet went inactive, the next step depends on whether it was extrapolated up or downwards. In the first case, it must still be extrapolated downwards and is inserted in the appropriate pool, in the second case the tracklet has been completely processed and can thus be stored if it fulfills the criteria in section 2.3.4. Effects of incoherent memory will be discussed later in section 6.2.2.7.

The criteria for which rowblock to choose are the following. During the Start Hit Sorting up to 256 start tracklets, starting within the same rowblock are assigned to each block. In this way, as for the improved scheduler above, for the first iteration no scheduling is needed. When a block has finished its rowblock, it tries to fetch more tracklets from the pool of the rowblock it just finished. If the pool is empty it takes the next rowblock. After the last pool for upward extrapolation is empty, it starts with downwards extrapolation from the same block, then with the previous block and so on (since downward extrapolation processes the rows in the other way around).

Several points that led to the algorithm being implemented in the above described way have to be considered:

- Implementing multiple thread pools reduces the load on them.
- The pools are further relieved since only one thread per multiprocessor accesses the pool while others are synchronized via shared memory.
- The global memory is not coherent leading to schedule collisions (see 6.2.2.7).
- It is preferable to stay in the same rowblock first, in this way the next block will be filled better, reducing the possibility that less then 256 tracklets are fetched from one block.
- The optimal rowblock size is a parameter which has to be experimentally determined (see Fig. 6.22). Clearly rowblock sizes dividing the number of rows are optimal, otherwise the last rowblock would be of different size.

Fig. 6.20 shows an illustration of the dynamic scheduler. There is one suboptimal point in the current implementation. It regularly happens that a block stores tracklets to a pool, and immediately rereads them. In the future this should be avoided. However, it should be noted that in reality more tracklets are processed than in Fig. 6.20, hence this does not happen so often.


Pool A
Pool B
Pool C
Pool D
Rows 1 - 80, Forward Stage
Rows 81 - 160, Forward Stage
Rows 81 - 160, Backward Stage
Rows 1 - 80, Backward Stage
8,9
Empty
10,11
10,11,1,3
5,6
Empty
Empty
8
Empty
Empty
Empty
0
Empty
Multiprocessor 1
Multiprocessor 2
Processing Tracklets 0,1,2,3 (Rows 1-80, Forward)
Processing Tracklets 8,9 (Rows 1-80, Forward)
Processing Tracklets 5,6,9 (Rows 81-160, Forward)
Processing Tracklets 5,6,9 (Rows 81-160, Forward)
Processing Tracklets 4,5,6,7 (Rows 1-80, Forward)
Processing Tracklets 10,11,1,3 (Rows 81-160, Forward)
Processing Tracklets 8, 10,1,3 (Rows 81-168, Backward)
Processing Tracklets 0, 8, 3 (Rows 1-80, Backward)
a)
b)
c)
d)
e)
f)
g)
h)

a) Tracklets 0 stored to pool D, tracklets 1,3 stored to pool B, tracklet 2 dropped, tracklets 8,9 read from pool A
b) Tracklets 5,6 stored to pool B, tracklets 4,7 dropped, no tracklets found in pool A, tracklets 10,11,1,3 read from pool B
c) Tracklet 8 stored to pool C, tracklet 9 stored to pool B, no tracklets found in pool A, tracklets 5,6,9 read from pool B
d) Tracklet 10,1,3 stored to pool C, tracklet 11 dropped, no tracklets found in pool B, tracklet 8,10,1,3 read from pool C
e) Tracklets 8,3 stored to pool D, tracklets 10,1 dropped, no tracklets found in pool C, tracklets 0,8,3 read from pool D
f) Tracklets 5,6,9 stored to pool C, no tracklets in pool B, tracklets 5,6,9 read from pool C
g) Tracklets 5,6,9 dropped, no tracklets found in pool C, no tracklets found in pool D
h) Tracklets 0,8,3 dropped, no tracklets found in pool D

Figure 6.20: Illustration of Dynamic Scheduling Behavior

6.2.2.6 Dynamic Scheduling Analysis

Fig. 6.21 shows the GPU utilization using the Dynamic Scheduling algorithm with the optimal rowblock size which turned out to be 40. Clearly, on the one hand, the GPU utilization has increased as compared to Fig. 6.17. On the other hand it is obvious that during most of the time only a reduced number of warps within one multiprocessor are active. This results from the fact that simply not enough tracklets are available for scheduling.

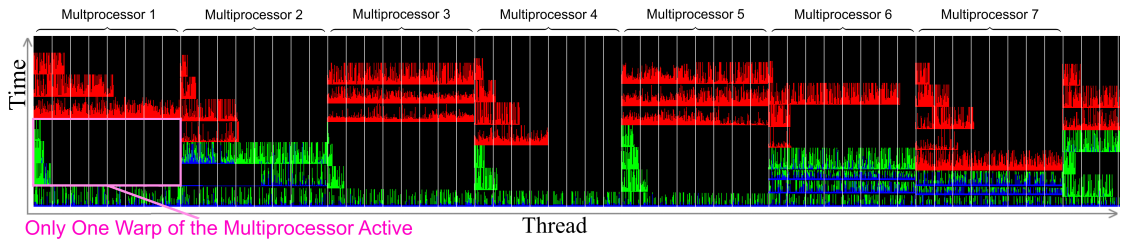


Figure 6.21: GPU Utilization during Tracklet Construction with Dynamic Scheduling

The problem that too few tracklets are available will now be addressed in detail. A simple way to extrapolate the GPU performance at an even better GPU utilization, is to reduce the number of blocks in the configuration. This reduces the number of multiprocessors used on the GPU. Hence the number of tracklets per multiprocessor increases. The optimal rowblock size of 40 is not necessarily optimal for lower GPU utilizations. Fig. 6.22 shows tracking times for reduced block counts and various rowblock sizes. In this way, the optimal rowblock parameter is found for each block count. Fig. 6.23 shows the results for full GPU utilization in more detail.

Fig. 6.24 shows a comparison of the dynamic and non-dynamic schedulers employing different block counts. As expected, the speedup of the dynamic scheduler is higher for lower block counts.

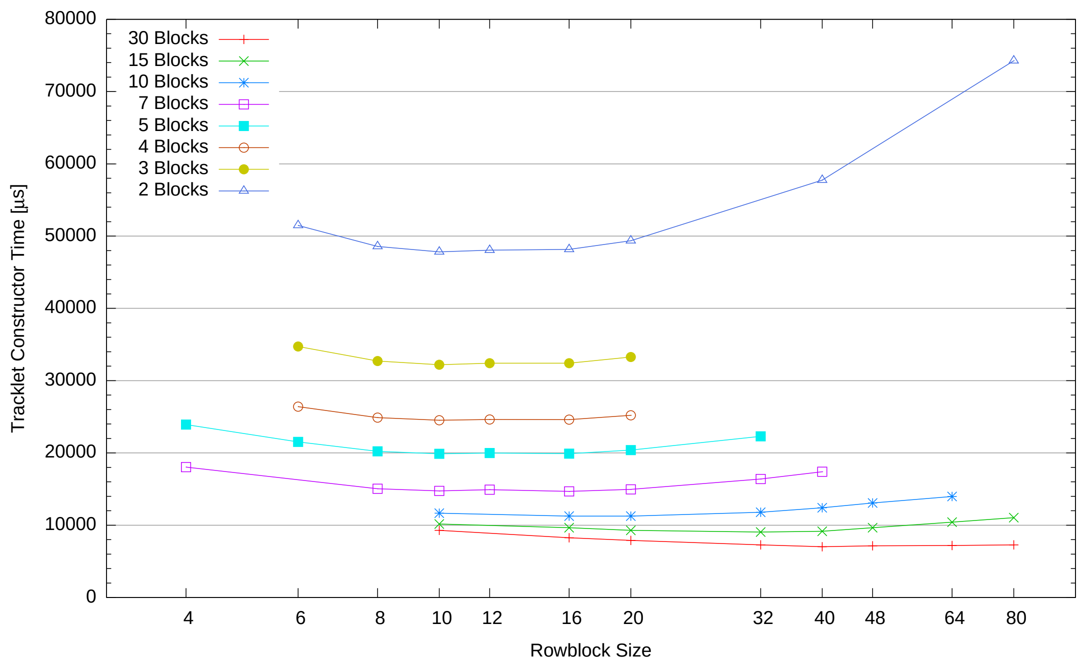


Figure 6.22: Tracklet Constructor Performance for Different Rowblock and Block Configurations

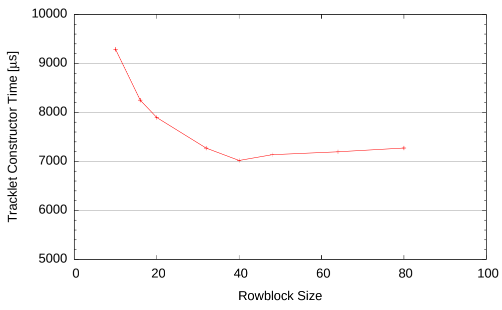


Figure 6.23: Tracklet Constructor Performance for Different Rowblock Sizes and Full GPU Utilization

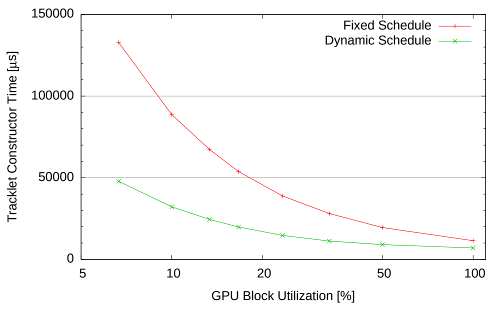


Figure 6.24: Performance of Dynamic Scheduler in Contrast to Fixed Scheduler

In the next step, the optimal block count or the optimal GPU block utilization respectively is determined. Theoretically the tracker performance should linearly scale with the number of blocks used. As one can see in Fig. 6.25, the performance rises almost linearly up to a GPU utilization of about 25%. As a consequence, the summarized execution time for all active blocks remain almost constant up to this point (see Fig. 6.26). By using only 25% of the GPU already 50% of the maximum performance can be achieved. The saturation effect can be explained by examining Fig. 6.21: After the first interrupt, many multiprocessors contain only up to two warps with active tracklets. Fig. 6.27 shows the same plot with only 25% of the GPU utilized. Obviously, the utilization of the reduced number of multiprocessors is greatly improved. Naturally there are other aspects, such as limited memory bandwidth, which contribute to the saturation that is observed.

As was noticed, up to 25% of the GPU can be well utilized by the given number of tracklets. Neglecting other effects as bounded memory bandwidth, an increase of the thread count by a factor of four should result in a good overall GPU utilization.

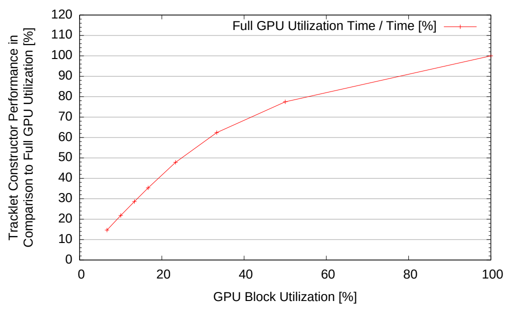


Figure 6.25: Relative Tracklet Constructor Performance for Different Multiprocessor Counts

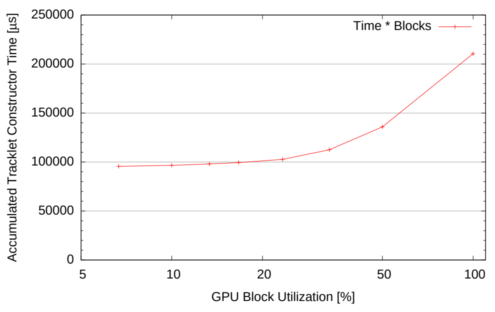


Figure 6.26: Accumulated Tracklet Constructor Time of All Multiprocessors

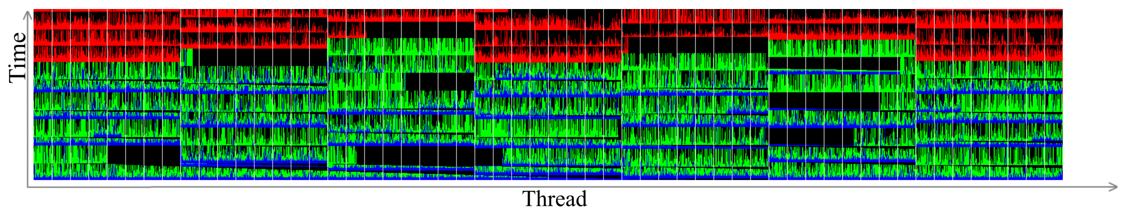


Figure 6.27: GPU Utilization during Tracklet Construction with Dynamic Scheduling using only 25% of the GPU multiprocessors

This enables several opportunities to proceed. Either multiple threads must process the same tracklet. For example, the search for clusters near the extrapolation point could be parallelized. Unfortunately, this would result in far too much parallelization overhead, when realized within the CUDA framework. This procedure would be better suited for parallelism of finer granularity, such as explicit vectorization, and, in fact, such a realization can be found in [Kre]. Another possibility to increase the number of threads virtually would be to run the up- and downward extrapolations in parallel doubling the active thread count. Here it would be difficult to merge the two sets of track parameters afterwards. Additionally this would result in a total increase of extrapolation steps, since during upward extrapolation tracklets with a too high χ^2-value of the covariance matrix can be dropped and not extrapolated downwards in the first place. In the end this could only result in a factor of two, not four anyway.

The last possibility, which is easy to implement, is the processing of multiple slices in parallel. According to the measurements four slices should be enough. Having analyzed the performance and performance bottlenecks for the dynamic scheduler, before proceeding to a multi-slice implementation, the problems resulting from the incoherent global memory will be examined.

6.2.2.7 Scheduling Collisions

A scheduling error provoked by memory inconsistency due to the incoherent memory design will be called a **scheduling collision**. For creating a dynamic tracklet pool the question arising is: How to maintain a list, with only incoherent memory available. In a usual thread-safe list, the operation to add items, would at first increase a counter using an atomic operation and then write the items to a buffer. The position inside the buffer is defined by the initial counter value returned by the atomic operation. Up to this point, this is not thread-safe, since another thread could attempt to access the items in the buffer before they have been completely stored.[5] For a thread-safe implementation, the new items are marked invalid until they are completely stored in the buffer.[6] For fetching the items from the list a second counter is maintained, pointing at the next element in the buffer. To read from the list, a thread has to increase this read counter using an atomic add instruction.

Soft Collisions However, for incoherent memory the above algorithm for a thread save list still does not work correctly. It requires the memory to be written in the order it issued the requests. This is not ensured on the GPU. An error occurs when the memory operation marking the data as valid finishes prior to the operation actually storing the data. A thinkable solution is to read the data from memory first and check whether it is valid before unlocking it. However this does not work for the GPU. Even if one thread already reads the new data it is possible that another thread would still read the old data. Memory consistency between different blocks is never ensured until the kernel has finished. The problem just described is called a soft collision. Fig. 6.28 shows the principle of a thread-safe list, including the example just mentioned.

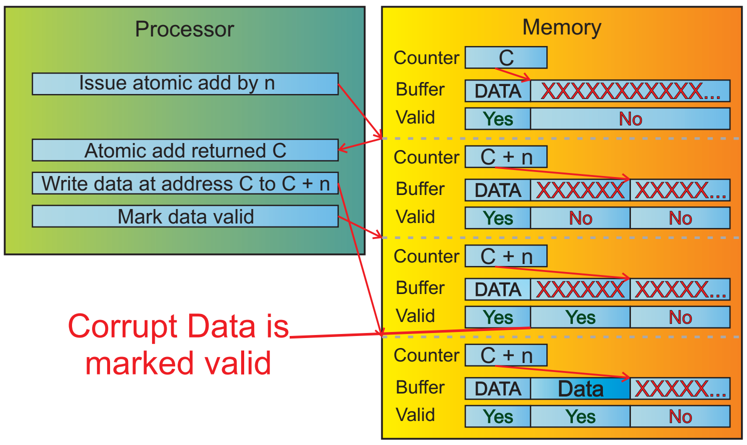


Figure 6.28: Thread-safe List not Working with Incoherent Memory

To account for the incoherent memory the algorithm is altered in the following way: The buffer is initialized with a value that will never be stored within it, in the tracker case -1. Every position in the buffer can only be used once.[7] No extra flag for valid and invalid data is needed. Instead, data in the buffer is valid as soon as it does not contain a -1 dword. A thread fetching tracklets from the pool will check the buffer data for -1 values, and keeps polling the buffer until no such value is detected anymore. In this way, soft collisions can be prevented.

Hard Collisions The solution presented above made the scheduler stable in most cases. In theory collisions should be precluded completely, but because of the GPU design they are still possible. The problem is that there is no guarantee, when the -1 dword marking data invalid will be overwritten by the correct content. A case can occur, when the polling thread keeps reading -1 values until the kernel is terminated by the driver after a timeout of about one

[5]Flipping the counter increase and the store operation would not work either. In that case a problem would arise if two threads would try to simultaneously add items resulting in a collision as both would write to the same location.

[6]Marking items valid can either be realized by another counter, or by mutual exclusion.

[7]Lists are usually implemented using ring buffers, which is impossible here.

second.[8] Such an event is called a hard collision. The occurrence of this phenomenon could be greatly relieved by introducing wait cycles for the thread polling between the attempts. In doing so the load of the corresponding memory address is greatly reduced, yet, hard collisions cannot be completely excluded.

To ensure that the kernel is not terminated because of the timeout, the number of attempts is limited, and 30 was determined to be a good limit. Afterwards the thread will start fetching other tracklets, whereby the tracklets that could not be retrieved from the buffer are lost. Experiments demonstrated that hard collisions regularly occur for a sequential memory area of 10 to 30 values simultaneously. It is questionable whether it can be accepted to loose up to 30 tracklets. Since this only affects parts located in one slice of the total track, the track itself should still be found in the majority of cases. Nonetheless, the handling chosen for hard scheduling collisions is to simply reinitialize and rerun the Tracklet Constructor. An implementation in which only the tracklets lost during the collision are processed is imaginable and would clearly be faster. But since hard collisions are very rare this is not done for the time being. To provide some numbers: In a series of 48550 runs (36 slices each) 226 collisions occurred. Therefore, the probability for a hard collision during a full tracking run is (0.47 ± 0.03) %. See 6.29 for an illustration of the handling of scheduling collisions.

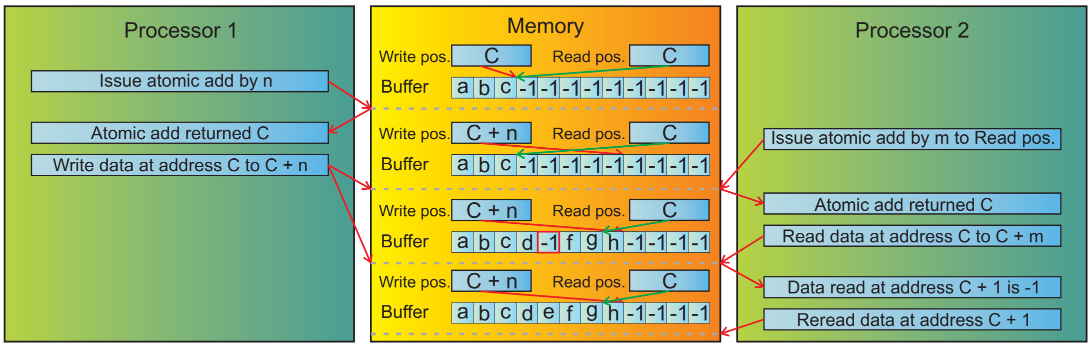

Processor 1
Issue atomic add by n
Atomic add returned C
Write data at address C to C + n
Memory
Write pos.
C
Read pos.
C
Buffer
a b c -1 -1 -1 -1 -1 -1 -1 -1 -1
C + n
a b c d -1 f g h -1 -1 -1 -1
a b c d e f g h -1 -1 -1 -1
Processor 2
Issue atomic add by m to Read pos.
Atomic add returned C
Read data at address C to C + m
Data read at address C + 1 is -1
Reread data at address C + 1

Figure 6.29: Schedule Collision Handling

Data Inconsistencies Obviously not only the scheduler suffers from the incoherent memory. One other particular problem that came up during the tests is the following: In the initial fitting phase of the Tracklet Constructor (IV (a)) every other row is skipped. In the array storing the indices of the clusters assigned to the tracklet for each row, every second index in the seed's interval must be initialized to -1 (no cluster). Afterwards during backward extrapolation (IV (c)) the tracker might find clusters in these rows, and overwrite the -1 with the correct cluster index. Now it might happen that the memory is written in the order opposed to the sequence, the writes were issued. In that case the correct cluster index would be overwritten by -1.

The Tracklet Constructor writes a -1 in the extrapolation phase when no cluster is found, which could lead to the assumption that the preinitialization is not necessary at all. Unfortunately, this is wrong, as the extrapolation might stop before it wrote all the -1 values, when the χ^2-value of the covariance matrix rose too high. The algorithm was therefore altered to respect this GPU behavior.

[8] This does not implicate a bug in the NVIDIA memory controller but is in accordance with the specifications stating that memory consistency is ensured only after the kernel is finished. And in fact, in situations where a kernel terminated because of the timeout, reading the corresponding memory address in the next kernel results in the correct data.

There is one other part in the algorithm which could be influenced by inconsistent memory. The track parameters are written to and reread from global memory by the scheduler. Unfortunately, there is no way to circumvent the problem, except for the variant explained in section 6.2.2.10, which was found not to deliver the estimated performance. In the end the tracking quality analysis in section 9.1 shows that the final tracking results are obviously not influenced that much, if at all. One possibility for the future would be a check similar to the one used for the tracklet pool. However, this is not implemented for the time being as experiments show that the tracker is in fact working as is.

6.2.2.8 Multi-Slice Scheduling

The dynamic scheduler can easily be extended to support multiple slices in parallel. More tracklet pools for all the slices simply have to be added. The number of slices is restricted only by the available memory.

As stated in 4.2.4 the constant row data requires about 13 kB per slice, giving a limit of four slices for 64 kB of constant memory available. When storing constant row data in global memory more concurrent slices are possible. This course of action is called "**global row data**". An analysis of the performance impact, when the constant cache is not available for row data, is done in section 6.2.2.12 later.

Obviously concurrent slice counts dividing the total slice count should be given preference, as long as slices from different events are not mixed up. Otherwise when processing 11 slices in parallel for example, a full tracking run would process three times 11 slices and one time 3 slices only. Figures 6.30 and 6.31 show GPU utilization for 4 and 9 slices. One can see that the overall utilization has greatly improved. Still, in some places of the diagram inactive warps appear. This happens, when a tracklet pool contains less than 256 tracklets.

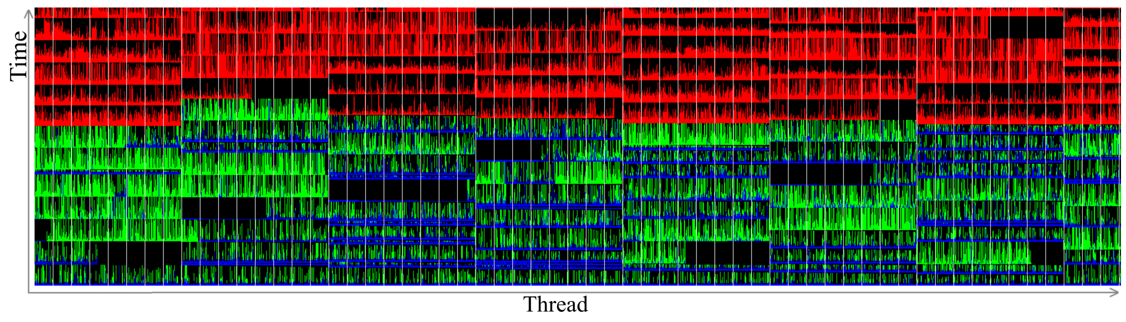


Figure 6.30: GPU Utilization during Stage IV with Dynamic Scheduling of 4 Slices

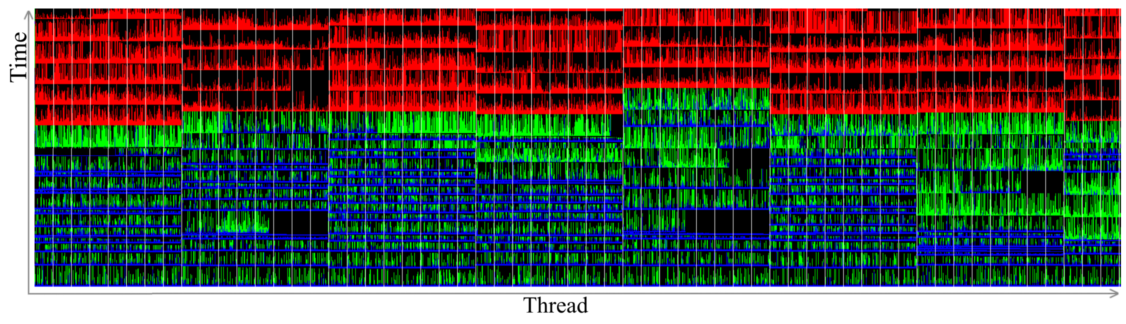


Figure 6.31: GPU Utilization during Stage IV with Dynamic Scheduling of 9 Slices (Using Global Row Data)

6.2.2.9 Start Hit Filtering

When looking closely at the utilization plots, it becomes apparent that many tracklets are dropped directly after a track fit stage over 3 clusters. This follows from the Tracklet Constructor criterion to demand a seed of at least 4 clusters.

A filter was integrated in the Start Hits Finder, dropping seeds with less than 4 clusters immediately. This decreased the number of tracklets significantly, speeding up the Tracklet Constructor and Tracklet Selector. Even though the Start Hits Finder itself gets more complex, it performs better with the filter active, since it has to store fewer start hits. See Fig. 6.32 for a comparison and Fig. 6.33 for the resulting GPU utilization.

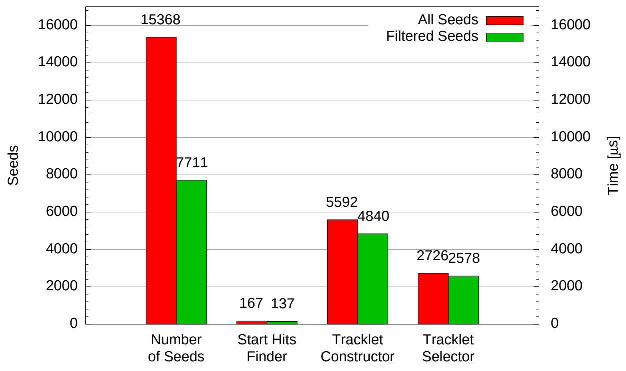


Figure 6.32: Results of Start Hit Filtering

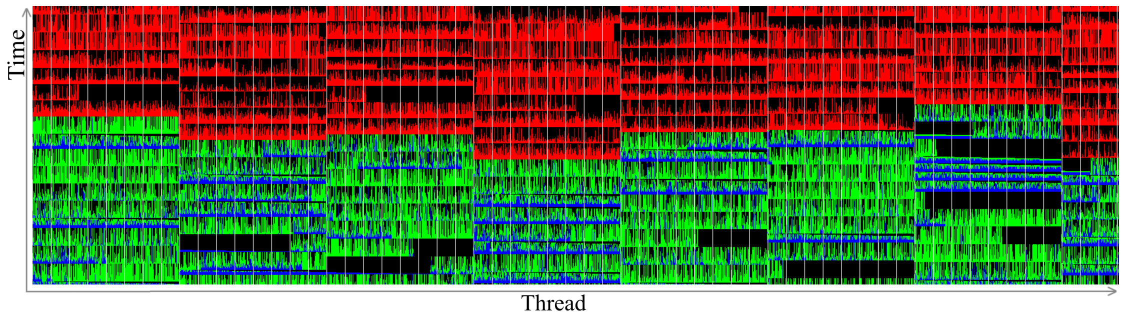


Figure 6.33: GPU Utilization during Tracklet Construction with Dynamic Multi-Slice Scheduling and Filtered Start Hits

6.2.2.10 Other Dynamic Scheduling Variants

Several variants and parameters of the scheduling algorithm are conceivable. Two attempts are presented here.

Fixed or Dynamic Slice Using the multi-slice scheduler it is possible to fix every multiprocessor to a particular slice or to give the multiprocessors freedom to fetch tracklets from any slice. The first variant reduces the load of the tracklet pools, while the latter delivers a better load balance. Figure 6.34 shows a clear advantage for the second one.

Host Synched Scheduling To exclude completely the possibility of scheduling collisions, host synchronization points can be used. After a kernel execution is finished, the global memory is in a coherent state. An implementation was realized, in which the Tracklet Constructor kernel stopped after storing the tracklets to the pools. Afterwards the kernel must be called again, reading the tracklets from the pools which are now in a coherent state. This is repeated until all pools are empty. As illustrated in Fig. 6.35 the host synchronization generates an unacceptable delay, and for that reason only device synchronized scheduling is used.

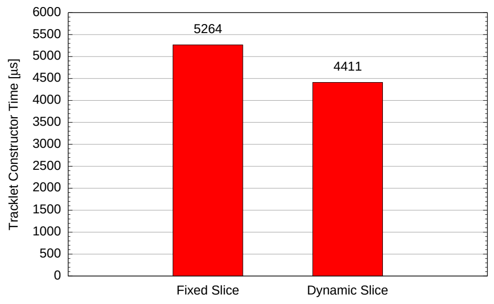


Figure 6.34: Performance for Fixed and Dynamic Slice Scheduling

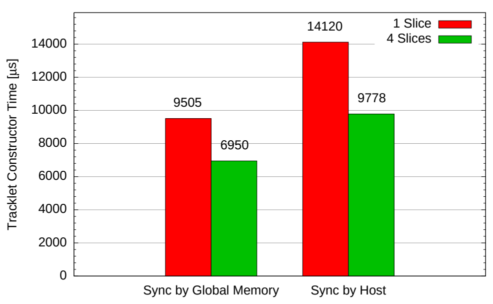


Figure 6.35: Comparison of Host- and Device-Synched Scheduling

6.2.2.11 Dynamic Scheduling Performance Analysis

Figures 6.36 and 6.37 show the initial and final GPU utilization for direct comparison. In these figures the full Tracklet Constructor run is included, while the previous figures were restricted to only a part of the Tracklet Constructor run for easier inspection.

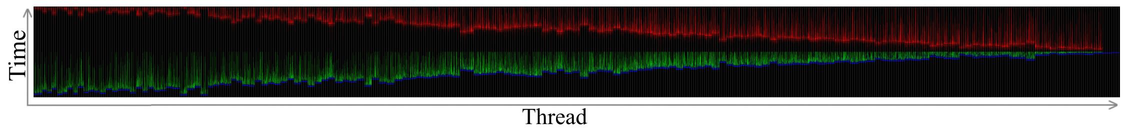


Figure 6.36: Initial GPU Utilization

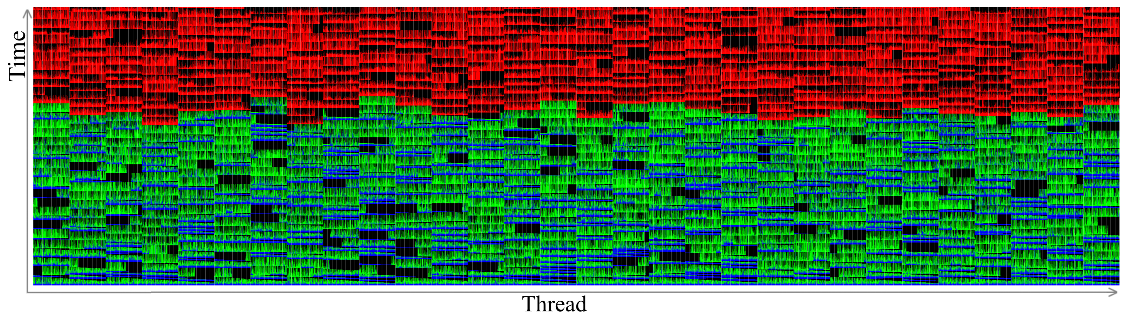


Figure 6.37: Best GPU Utilization

To evaluate how well the GPU is utilized, the measurement from Fig. 6.25 is repeated. Fig. 6.38 shows the Performance of both scheduling algorithms when using only a restricted number of multiprocessors normalized to the maximum performance. Contrary to Fig. 6.25 the performance of the multislice scheduler still scales well for a GPU utilization above 25%. Fig. 6.38 also illustrates, that when using only 25% of the GPU, or even less, the multi-slice scheduler is even slightly slower because of the increased complexity.

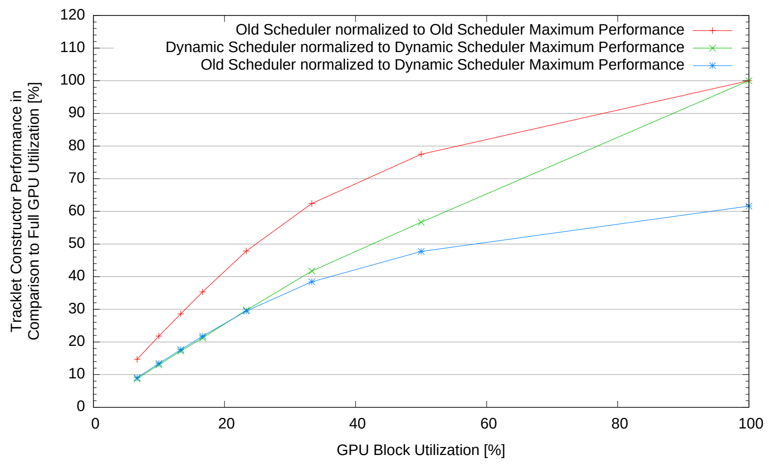


Figure 6.38: Relative Tracklet Constructor Performance for Different Multiprocessor Counts

Fig. 6.39 shows the efficiency of the scheduler, which is the average percentage of active threads. Efficiency rose from a poor 19% to 62%.

A timebin is defined as one pixel in the GPU utilization plot. Clearly the number of timebins does not directly reflect the performance, since warps can skip rows in common, resulting in many timebins but hardly long delays. Still, more timebins lead to more overhead. Fig. 6.40 shows that the number of timebins could be tremendously reduced.[9]

Finally Fig. 6.41 shows a comparison of all the scheduling algorithms presented.

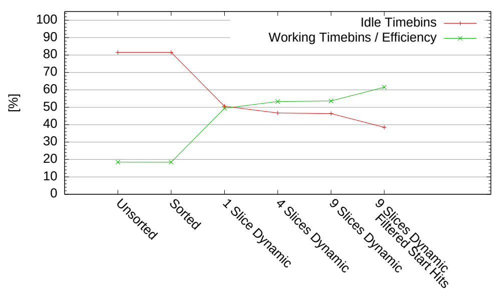


Figure 6.39: Efficiency of Scheduling Variants

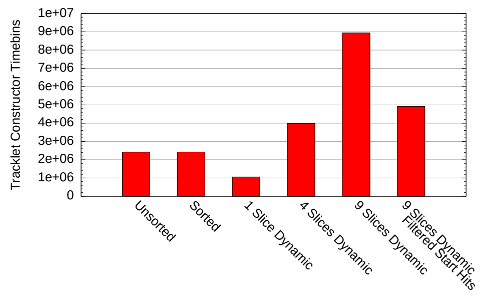


Figure 6.40: Timebins used by Scheduling Variants

[9]The number of timebins for the multi-slice variants is higher, since the plot shows the count for one Tracklet Constructor kernel. To get a comparable value it must be divided by the slice count.

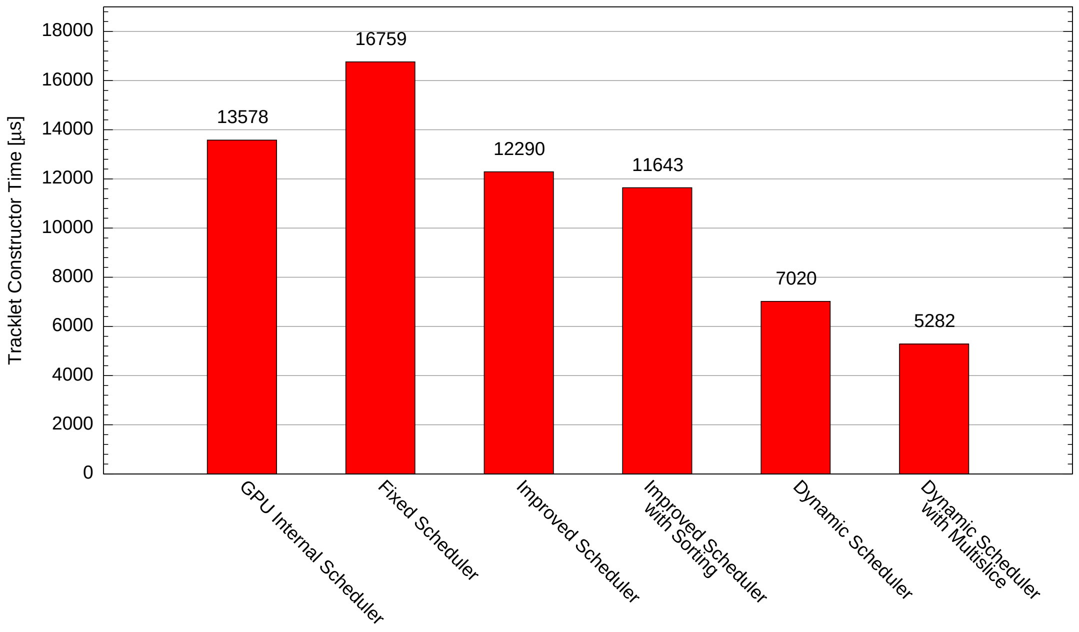


Figure 6.41: Comparison of Scheduling Algorithms

6.2.2.12 Dynamic Scheduling with Caching

Having created a new Tracklet Constructor wrapper with a dynamic scheduling algorithm, a second attempt was started to implement a shared memory cache. A shared cache is expected to save a constant period of time, but not (to save) a relative factor. The number of memory accesses should not have changed with the integration of the scheduling. Hence the relative effect should now be larger, since the total Tracklet Constructor time has shortened. The problem with the shared memory limitation remains, but shall be ignored for now.

One additional step needed to implement the shared cache is the integration of synchronization points between the rows for all threads in one block. Up until now only the threads within one warp were forced to process identical rows which is now extended to the entire block (block based row synchronicity instead of warp based). To test the shared cache on the most advanced implementation yet, the global row data variant will be used allowing for more than 4 slices.

Adding the shared memory cache increased the complexity of the algorithm forcing the compiler to extend the local memory usage by a factor of 2. For comparison also benchmarks with 128 registers but hence only 128 threads are included, as no local memory is required then. Clearly this has a negative effect, since a fixed set of 32 threads is reserved for filling the cache, leaving only three fourth of the threads available for tracking.

Fig. 6.42 shows the Tracklet Constructor performance for the additional steps just introduced, and for the version with shared memory cache. Finally, it becomes clear that a shared memory cache will not bring any benefit on the current hardware. For 64 registers the new version is significantly slower and still for 128 registers, the cached and non-cached (with all features required for caching enabled) versions are almost on a par.

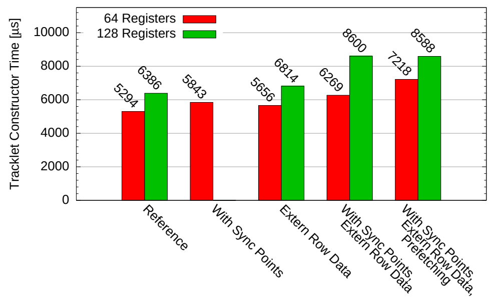


Figure 6.42: Performance of Tracklet Constructor using Scheduling and Caching

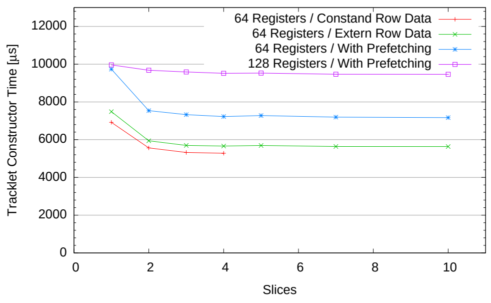


Figure 6.43: Performance for Scheduling and Caching for Different Multi-Slice Configurations

Fig. 6.43 shows the performance for different slice counts. As predicted, a slice count of 4 seems optimal. The constant row data version is steadily faster than the global row data version. Additionally the latter one is not accelerated any further when exceeding 4 slices. So at the current point, using global row data does not make sense.

6.2.2.13 Simple Scheduling

It is clear now that no shared memory cache will be employed. This gives freedom to change the algorithm and especially to think about row synchronicity again. In section 6.2.2.4 it was already noted that it might be suboptimal to access the tracklet pool block-wise since threads within one block still have to wait for the last one or the next interrupt. A simplified scheduling algorithm without row synchronicity and rowblocks is conceivable as follows.

- All threads are assigned a start tracklet and process the first row for that tracklet (the start row might and will be different among the threads)
- Tracklet Construction is iterated row by row until a tracklet becomes inactive.
- The warp is serialized into two branches where threads with an active tracklet will just wait for the moment.
- All threads with inactive tracklets will access the thread pool and fetch a new tracklet (Only one thread per multiprocessor will do the actual access, while the other threads are synchronized via shared memory).
- Warp serialization ends and the threads process the next row.

This scheduling scheme has one characteristic that requires an adaptation of the Tracklet Constructor algorithm itself. The threads will access different rows and can extrapolate up and downwards within one warp. It has to be ensured that internally all these parameters will be handled just by pointers to the correct data, but never by branching.

The first simple scheduler resulted in a disillusioning performance (see Constant Row Data performance in Fig. 6.44). The constant memory cache turned out to be unable to handle the simultaneous access to different memory locations when threads were accessing different rows. With the decision not to use a shared memory cache in the former sense, almost 16 kB of shared memory are available, enough to store the 13 kB of row data. The simple scheduler was accelerated enormously using the new cache, whereas the former dynamic scheduler's performance remained unchanged. The dynamic scheduler is faster than the simple scheduler for every row data configuration, so the new approach was dropped. Nevertheless, the result shows the possibility of moving the row data from constant to shared memory without any impact on performance. This makes it possible to use the multi-slice scheduler with more than four slices. This might sound unreasonable, as four slices were found out to be the optimal parameter. However, this number is related to the amount of tracklets. All benchmarks were done using absolute worst case simulation setting. Therefore, it is probable that the tracklet count will be less in the real experiment, and thus increasing this number is desirable. Another reason for an increased slice count will appear in section 6.3.3.

Finally, it should be noted that the global row data variant is tremendously faster than the constant row data version when using the simple scheduler. Obviously in this constellation the cache slows the algorithm down. The only possible reason seems to be that the constant memory cache in the current hardware is incapable of handling simultaneous random access over the entire cached area.

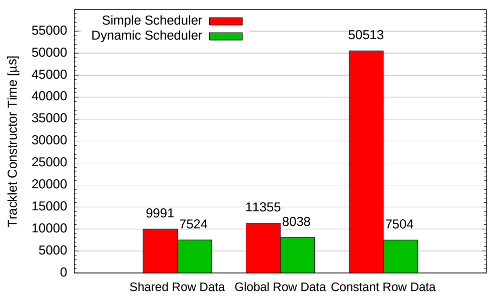


Figure 6.44: Performance of Simple Scheduler using Different Storage Types

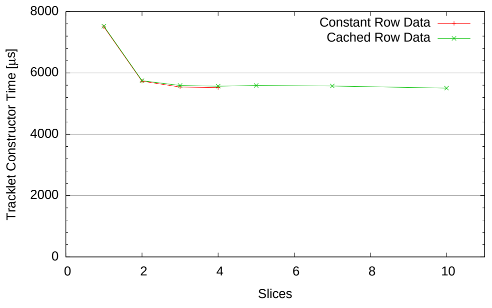


Figure 6.45: Performance of Dynamic Scheduler Using Constant Row Data Cache

Fig. 6.45 shows that performance of the constant row data and shared row data version is equal for up to four slices, but again the algorithm does not profit from more slices. Nonetheless, the dynamic multi-slice scheduler with shared row data is both the fastest and most flexible version. Therefore, this the preferred configuration for the Tracklet Constructor.

The changes to the shared memory cache shall be summarized once more: Within the first shared cache, stored cluster coordinates for one row were stored. The cache had to be updated for every row that was processed. The new shared row cache only stores row parameters and pointers to row data, but not the data itself. It must only be updated if a different slice is processed.

One last aspect of the cache still has to be addressed:

Shared Memory Cache Size The dynamic scheduler always processes one rowblock at the same time, and might even jump to another slice afterwards. This opens up the opportunity to cache parameters and pointers for rows in the current rowblock only instead of the entire slice. This reduces the time required to perform the caching itself. However, for the final storage of the tracklets row parameters for all rows are required. Hence the reduced cache could not be used during the storage but global memory would have to be accessed instead. Fig. 6.46 shows that storing the entire row parameters for one slice is preferable.

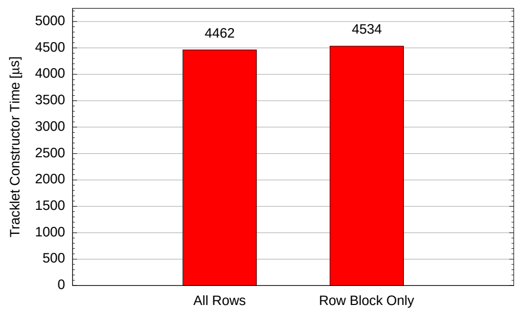


Figure 6.46: Analysis of Shared Cache Size

6.2.3 Tracklet Selector

In a comparison of the three most time consuming tracking steps the Tracklet Selector seems to be the most insignificant one to optimize. It becomes even less relevant when considering some aspects presented in section 6.3.3.5. Nonetheless, some optimization approaches will be described briefly.

Shared Memory For the Neighbors Finder and the Tracklet Constructor, the shared memory proved to be a more or less valuable tool. The Tracklet Selector runs through the clusters assigned to the tracklets, and, if they pass some checks, forms a track out of them. These clusters, or at least some of them, should be stored temporarily in a fast memory, so when actually creating the track they do not have to be reread from global memory. For the same reason as for the Neighbors Finder, i.e., dynamic array access, registers cannot be used as storage. Thus the shared memory is reserved for this, which comes at no cost, as it is not used otherwise. As Fig. 6.47 shows, this results in a small improvement.

Multiple Slices The Tracklet Selector suffers from the same problems as the Tracklet Constructor. The number of tracklets is small anyway, and those present strongly vary in length. At least the first issue can be accounted for by processing multiple slices in parallel. Fig. 6.48 shows how the performance scales with the slice count. It must be noted that slice counts dividing 36 are favored, as otherwise the last run processes a reduced number of slices only. This explains why in particular 7 and 8 parallel slices perform suboptimally. Again this could be solved by intermixing slices of different events.

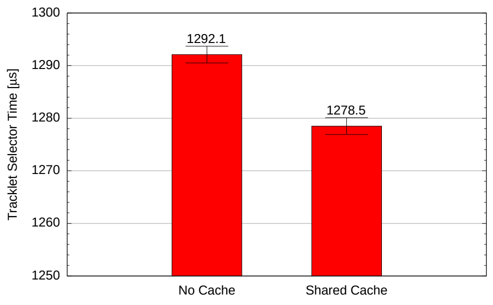


Figure 6.47: Speedup for Shared Memory Cluster Cache

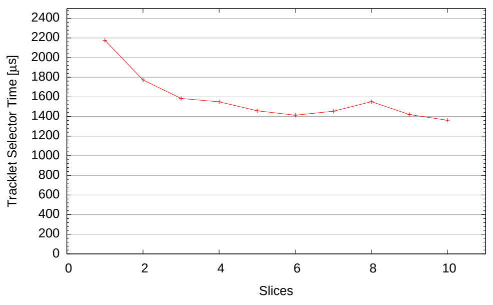


Figure 6.48: Performance of Multi-Slice Tracklet Selector

6.3 Overall Optimizations

The optimizations presented in this section do not apply to a single tracking step but rather to the entire GPU tracker. Still, benchmarks may show the performance of single steps, when the differences were most significant there.

6.3.1 Memory Optimizations

6.3.1.1 Memory Layout

Various optimizations to the memory layout were done but presenting them to the full extent would be beyond the scope of this document. Most changes attacked data initialization steps which were either unnecessary, done multiple times, or could be done concurrently during the algorithm. In addition, by the removal of some obsolete content in the output data structure, memory as well as both memory and network bandwidth could be economized on. The boost of the CPU tracker performance resulted mainly from these changes. However, they are in no way GPU specific and therefore do not belong here. Instead, two particular changes will be illustrated as an example to show that the best implementation for GPU and CPU can differ.

Row Hits In the original tracker code the data structure for the tracklets contained an array of cluster indices where the i^{th} entry was the index of the cluster in the i^{th} row assigned to the tracklet. For the CPU implementation this is clearly the best way to store the clusters. When processing a tracklet row by row, the indices are needed one after another, which is perfectly handled by the CPU cache. This is different for the GPU. There, 16 threads of one half warp will access the cluster index for a common row simultaneously (because of row synchronicity). Using the traditional layout (internal cluster indices), the addresses will be distributed with literally no two belonging to one 128 byte segment. In fact, the distance in memory is the size of the tracklet data structure. Therefore, the GPU cannot coalesce the access at all. It would thus be better to store all indices for one row and adjacent tracklets externally together, but not within the data structure for the tracklet (external cluster indices).

Fig. 6.49 shows both variants for the GPU and CPU version. As predicted, the new version performs better on the GPU, whereas the traditional one runs faster on a CPU. To achieve maximum performance on both platforms, two versions are maintained using conditional compilation directives.

Cluster Coordinates For the cluster coordinates, the situation is different. The x-coordinates are already defined by the rows, hence only y- and z-coordinates must be stored. It is possible to maintain two arrays of floats: one array for the x and the other for the y coordinates. An array of float2 vectors can be used alternatively (packed coordinates). As usually y- and z-coordinates are needed simultaneously for one cluster, the second variant seems preferable. In fact, traditional CPU code would have been written like this, and as Fig. 6.50 shows this is the right choice for the GPU as well. But here modern CPU code might follow a new paradigm because the first variant is preferable when vectorization comes into play.[10] Then two vectors of y- and a vector of z-coordinates can be directly loaded into two vector-registers. The vectorized tracker in [Kre] implements the storage in this way.

[10]The CPU code used in Fig. 6.50 was not using vector instructions.

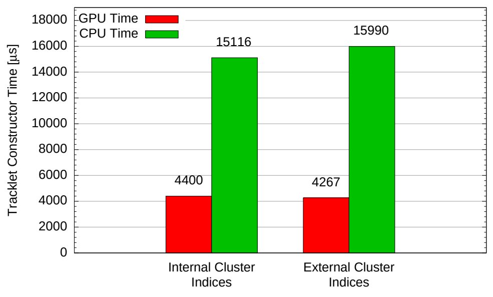


Figure 6.49: Performance of Optimized Storage of Row Hits

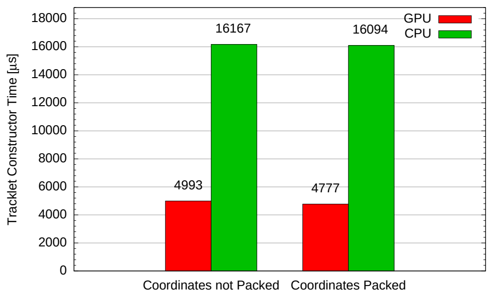


Figure 6.50: Performance for Different Storage Variants of Cluster Coordinates

6.3.1.2 Texture Fetches

The current NVIDIA GPU family does not possess a general global memory cache but has a special cache for textures. It is a read only cache and dedicated to texture fetches. Therefore, the obvious way to obtain a memory cache is to read the data via texture fetches. At this point, some theory about textures on the NVIDIA GPU is needed. Textures are one- to three-dimensional pixmaps stored in global memory. However, they are not necessarily present bit by bit. For example, a three-dimensional pixmap is defined by the following data.

- Width and height of the pixmap (dwWidth, dwHeight)
- Start Address in memory (pAddress)
- Pitch between rows in bytes (dwPitch)
- Data type specifier (type)

The element at position (x, y) can then be accessed at the address $x \cdot \text{sizeof(type)} + y \cdot \text{dwPitch}$. Textures do not necessarily have to be addressed using integer coordinates but floating point coordinates can be used as well. The adjacent pixels are then interpolated accordingly by the texture unit with zero overhead (except that multiple pixels must be read). Furthermore, integer data types can be seamlessly converted to floats by the texture unit. As textures are usually used to store colors, data vectors are also possible (e.g. a vector of four floats for red, green blue and an alpha value).

To access textures, a texture reference containing the texture descriptor must be created on the host. These references cannot be altered, but accessed only, from within CUDA kernels. For HPC applications the interpolation feature is usually not used and texture coordinates are integral instead. The data conversion feature, however, can be helpful.

Two proceedings seem reasonable for the GPU tracker. One big one-dimensional texture can span the entire memory. This is the easiest way, since then all addresses can simply be transformed to positions inside the big texture. As another possibility small textures can be used. A small texture approach has been implemented for the cluster coordinates for testing in the following way:

Up to this point, cluster coordinates were stored sorted by row. The number of clusters within one row is not constant. Thus there is no constant pitch between the first clusters of consecutive rows. This can be changed by determining the maximum cluster count within one row and then storing the clusters of each row with that maximum stride (see Fig. 6.51).

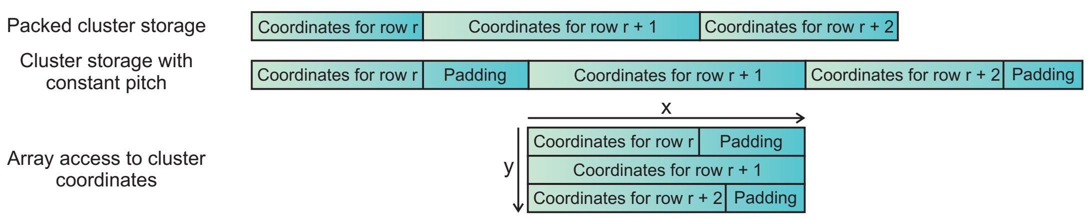

Packed cluster storage
Coordinates for row r
Coordinates for row r + 1
Coordinates for row r + 2
Cluster storage with constant pitch
Coordinates for row r
Padding
Coordinates for row r + 1
Coordinates for row r + 2
Padding
x
y
Array access to cluster coordinates
Coordinates for row r
Padding
Coordinates for row r + 1
Coordinates for row r + 2
Padding

Figure 6.51: Cluster Storage with Constant Pith

Then the cluster coordinates can be accessed as a two-dimensional array, with y the row index and x the cluster index within that row. Clearly the value is undefined if x is greater than the cluster count of row y. Using a vector of two 16-bit integers as data type one texture fetch can read x- and y-coordinate of a cluster and simultaneously convert them into floats. As the texture cache is optimized for two-dimensional access, the second variant should bring some benefit.

Fig. 6.52 shows an advantage of small two-dimensional textures fetches over fetches using one big texture. However, small textures result in one problem: Since arrays of texture references are not allowed, it is not possible to easily choose between different two-dimensional textures when processing multiple slices. This can be solved by introducing three-dimensional textures with the slice ID as the third coordinate. However, measurements show that the advantage of the small texture is nullified by this. Therefore, in the end, one single big texture is used in the tracker right now.[11]

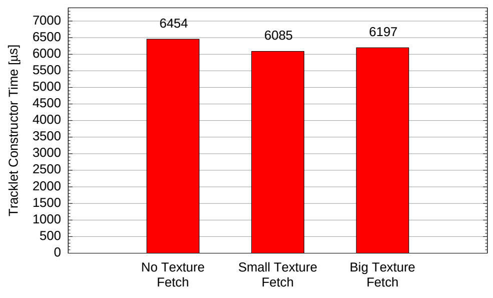


Figure 6.52: Performance Comparison of Texture Fetch Algorithms

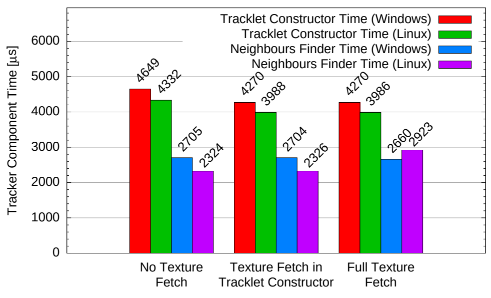


Figure 6.53: Texture Fetch Speedup for Various Configurations

Performance for the texture fetch version is shown in Fig. 6.53. The figure shows a general trend towards a faster tracking on Linux based systems. It is strange that on Linux the use of texture fetches has a negative effect, while on Windows based systems the measured performance coincides with the speculation that texture fetches should accelerate the tracker. This behavior occurs for all CUDA versions, and cannot be explained yet.

Using texture fetches in the Neighbors Finder will fill the cache before the Tracklet Constructor is even started. One theory was that the Tracklet Constructor time could be decreased when using texture fetches in both tracking steps instead of the Tracklet Constructor only. Fig. 6.53 shows, that this is not the case. Hence either the effect is negligible or the texture cache is not conserved between kernel executions.

[11]This keeps the implementation simpler and additionally the texture cache will be replaced with a general purpose cache in the next GPU generation.

6.3.2 CPU Multithreading

As explained in section 4.2.2, the initialization and Tracklet Output have not yet been adapted to run on the GPU, but stayed on CPU. It is therefore evident to process these tasks for multiple slices in parallel using multithreading. As the GPU tracker already processes multiple slices, implementing multithreading is easy. Before the GPU Tracker itself processes the slices, all the slice data structures are initialized, which is already realized in a loop over the slice ID. As everything is thread-safe by now, for multithreading the GPU Tracker, only a simple "*#pragma omp parallel for*" is needed. The same holds true for the Tracklet Output.

Fig. 6.54 shows the time required for initialization and Tracklet Output for one and for eight threads.[12] All measurements are done for the CPU and the GPU code. In the CPU code, the OpenMP loop iterates over the entire slice tracking instead of the slice initialization only. Thus, the overhead for starting and synchronizing threads is less important.

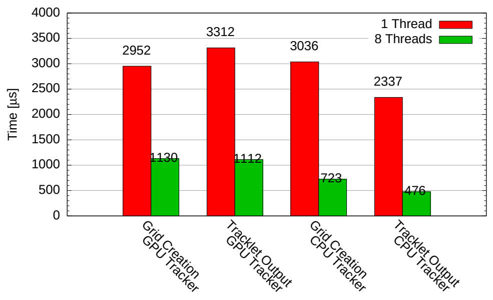


Figure 6.54: Multithreaded Performance for Tracking Steps of CPU- and GPU-Tracker[12]

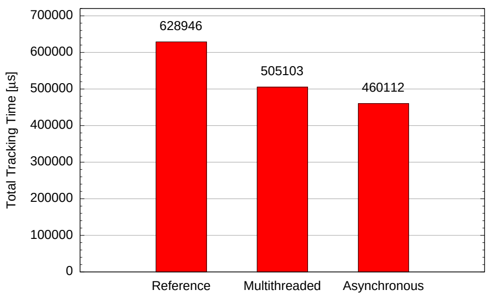


Figure 6.55: GPU Tracker Performance for GPU Tracker Variants

Fig. 6.55 shows the total performance gain for the multithreaded version. Clearly the numbers look good suggesting further analysis of multithreading. However, using an asynchronous, single-threaded processing approach presented in the next section the performance even improves. Though the approach is not explained yet, the performance numbers are integrated in the diagram to explain why multithreading was discontinued.

6.3.3 Pipelining

When reflecting on the current GPU Tracker implementation it becomes clear that the CPU idles during the GPU tracking wasting a lot of computational power. The GPU is also idling when the CPU does the initialization and Tracklet Output. The workflow can be pipelined in a way that CPU and GPU tasks overlap.

If different slices are to be processed asynchronously and concurrently, it seems evident to go one step further and make the memory transfer to and from GPU asynchronous, too. This means that three tasks can be performed in parallel:

- The CPU initializes slice $n-2$.
- Data for slice $n-1$ is asynchronously transferred to GPU.
- The GPU performs the tracking for slice n.

[12]The Nehalem CPU used possess 4 real but 8 virtual cores because of Hyperthreading. 4 threads therefore can increase in a fourfold performance increase, whereas performance does not scale linearly to 8 threads.

It is obvious that multiple slices must be processed in parallel to maintain a pipeline in this way. Especially the restriction to four slices, which was perfect for the Tracklet Constructor in the special case of the worst case simulation, may be insufficient to achieve optimal performance here. At this point the shared row cache comes into play, which permits going beyond four parallel slices.

It is difficult to separate pipelining and asynchronous memory transfer. Particularly there is no sense in doing all benchmarks twice, with and without asynchronous transfer. Therefore, for the time being the asynchronous transfer will be assumed as given and well performing when analyzing the pipeline. Section 6.3.3.3 will handle the actual asynchronous transfer. Nevertheless, the requirements will be stated here, as they affect the design of the entire asynchronous pipeline.

Requirements For the hardware to be able to perform a DMA memory transfer, source and destination memory areas respectively must be page locked.[13] To allow for an efficient memory management, the amount of page locked memory should be limited.[14] Most operating systems therefore only allow for a restricted amount of page locked memory. A first attempt to allocate all host memory page locked failed due to this limit.

To handle this the host memory is split into several parts: DMA destination memory, DMA source memory, and host only memory. The separation in destination and source memory is based on the fact that source memory is only read during the DMA transfer but the CPU will never read from that area but only write to it. Hence deactivating the CPU cache for source memory can bring a performance boost about. With these changes the host side is ready to perform asynchronous DMA transfers to the device.

The usage of streams is obligatory on the device side for DMA transfer and kernel execution to overlap. Every memory transfer and kernel execution can be assigned to a stream. A stream is a sequence of kernels calls and memory transfer instructions which are ordered according to the FIFO[15] principle. Multiple streams cannot execute kernels in parallel, nor can two data transfers take place concurrently. However, kernel execution for one stream and DMA transfer for another can overlap. The tracker creates one independent stream for every slice.

Implementation The tracker now works in the following way: Slice by slice the initialization is performed on the CPU and then, before the initialization of the next slice starts, all memory transfers and kernel calls up to the Tracklet Constructor are issued to the stream pipeline. The stream ID always equals the number of the slice. Then the Tracklet Constructor kernel is called without being attached to a stream. Kernels that are not assigned to a stream will first wait for all streams to finish before they start execution. This ensures that tracking steps I to III are finished for all slices, before the Tracklet Constructor is started. As the Tracklet Constructor processes multiple slices in parallel, this is the only possible way. The only other option is to set a manual synchronization point to wait for all streams to finish, and not until then start the Tracklet Constructor. But then device to host synchronization is required, whereas the first approach does all synchronization on the device.

[13]Page locked means that the mapping of physical to virtual pages must stay fixed for all pages involved because the DMA transfer will work on physical, not virtual addresses.

[14]Page locked memory areas cannot be rearranged, but rearrangement is essential to avoid memory fragmentation. Furthermore, page locked memory must immediately be reserved when allocating.

[15]FIFO stands for first in first out. The tasks are processed exactly in the order, they were issued.

Next, the Tracklet Selector kernel calls are issued. It shall stay open for the moment whether the multi-slice version is used or not. Pro and cons for this will be discussed in section 6.3.3.5. Unfortunately, the memory transfer copying the tracks back to the CPU cannot be issued instantly. The problem is that at this point it is still unknown how much data to transfer as this depends on the number of tracks found in step V. Therefore, two transfers are needed: The first will only fetch the number of tracks, while the second one will copy the actual tracks. See Fig. 6.57 for an illustration.

The simplest realization is to issue n Tracklet Selector kernels for the n slices in n streams again, then issue the small memory transfer copying the track count for every slice in the corresponding stream. After the i^{th} stream has finished (the last step in the pipeline up to now was the transfer of the track count) the tracks for stream i are transferred to the host and the Tracklet Output is performed on the CPU while the GPU is still executing the Tracklet Selector for the remaining slices. Unfortunately, it does not work in that way. The GPU does not reorder memory transfers even if they belong to different streams. If the above algorithm were used, the memory transfer for the tracks of the first slice would be issued after the memory transfer for the track count of the last slice. Therefore, both would only take place when the Tracklet Selector for the last slice will have finished (see Fig. 6.56 as compared to Fig. 6.57). Therefore, the transfers for the track counts and for the actual tracks have to be issued alternately, with manual synchronization points in between.

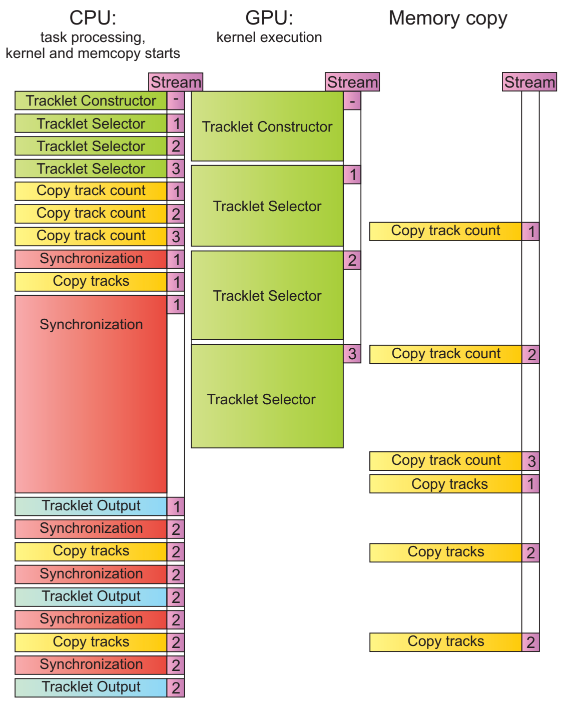


Figure 6.56: Kernel and DMA Issue Diagram for the Simple Algorithm

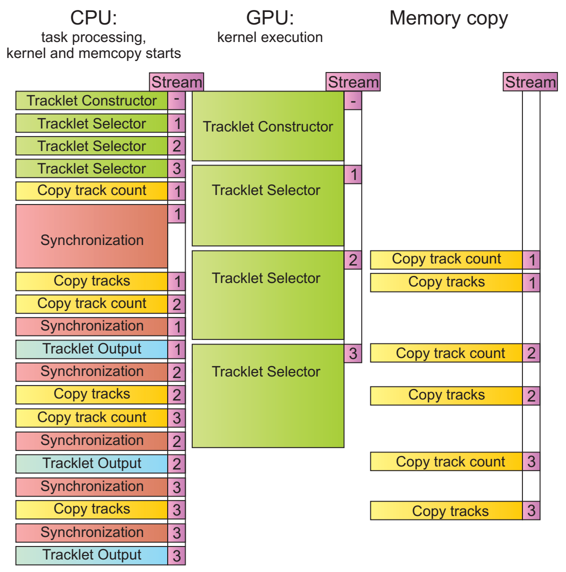


Figure 6.57: Kernel and DMA Issue Diagram for the Improved Algorithm

6.3.3.1 Workflow

In this and the following sections a new kind of diagram will be used regularly. To save some space it will be exemplified and explained in Fig. 6.58. The legend will be omitted afterwards. The x-axis represents the time within one run, and multiple tracking runs are shown one below the other. For each run, three rows represent the DMA, the GPU, and the

CPU steps.[16] Each step is assigned a different color. Fig. 6.58 illustrates a pipelined run with asynchronous memory transfer of 15 simultaneous slices. As 15 does not divide the slice count 36, the third and last run contains a reduced number of 6 slices.

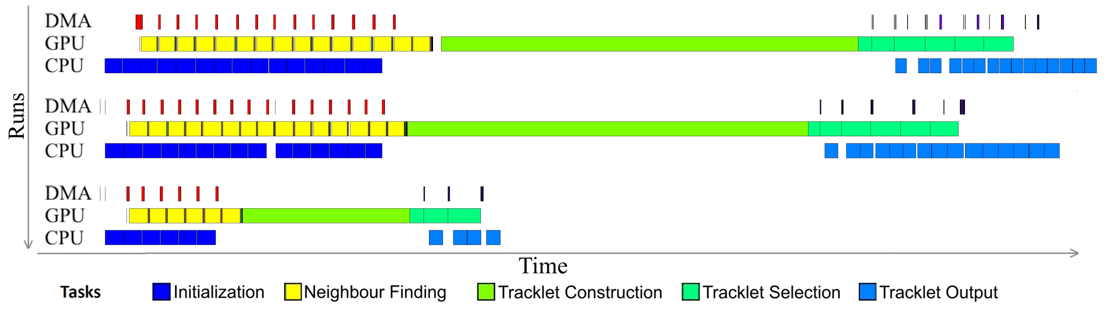


Figure 6.58: Workflow for a Pipeline with 15 Slices and Asynchronous Data Transfer

6.3.3.2 Pipeline Performance and Pipeline Length

For the pipeline there is only one relevant parameter: the number of slices in the pipeline. Clearly this should be pushed as far as possible. Figures 6.59, 6.60, 6.61, 6.62, and 6.63 show the workflow for 3, 6, 9, 12, and 18 slices respectively. As time for initialization and for Neighbors Finding as well as the duration of Tracklet Selection and Tracklet Output are similar, the CPU time of the algorithm is almost completely hidden by the GPU time, with only the runtimes of initialization and Tracklet Output of few slices as overhead. As seen in the figures this overhead is almost negligible for high slice counts.

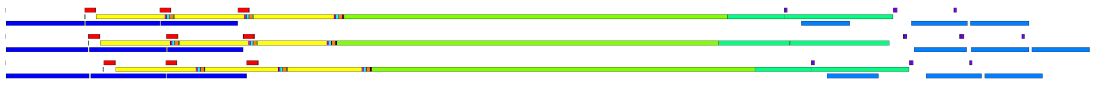


Figure 6.59: Pipelining 3 Slices

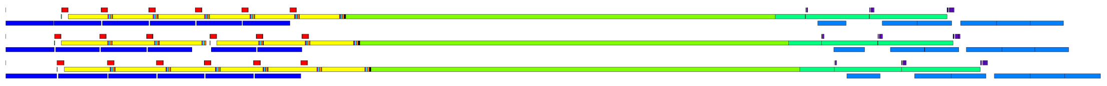


Figure 6.60: Pipelining 6 Slices

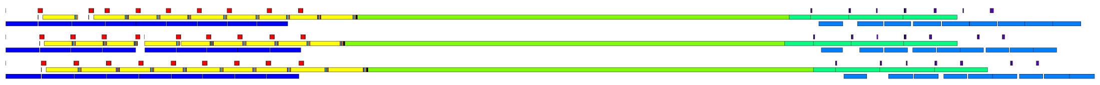


Figure 6.61: Pipelining 9 Slices

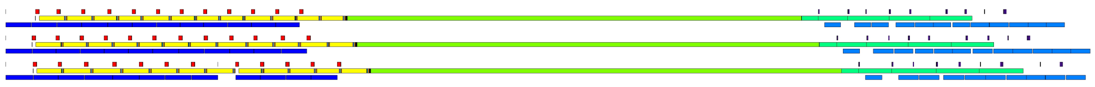


Figure 6.62: Pipelining 12 Slices

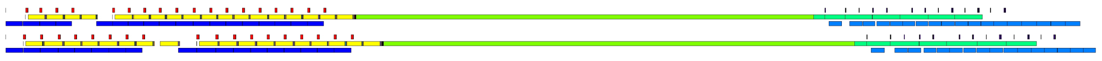


Figure 6.63: Pipelining 18 Slices

Fig. 6.64 shows a performance plot for different slice counts. Of course one should take into consideration that the Tracklet Constructor and Tracklet Selector also benefit from more slices and thus the performance gain is not solely based on pipelining. Moreover, the total slice

[16]The NVIDIA CUDA profiler was used to obtain the kernel start times and durations.

count 36 should be a multiple of the concurrent slice count to achieve optimal performance. Other slice counts will lead to the situation shown in Fig. 6.58. This explains why the optimal slice counts are 12 and 18. Running 36 slices in parallel is not possible because of the limited memory of the GPU used.[17]

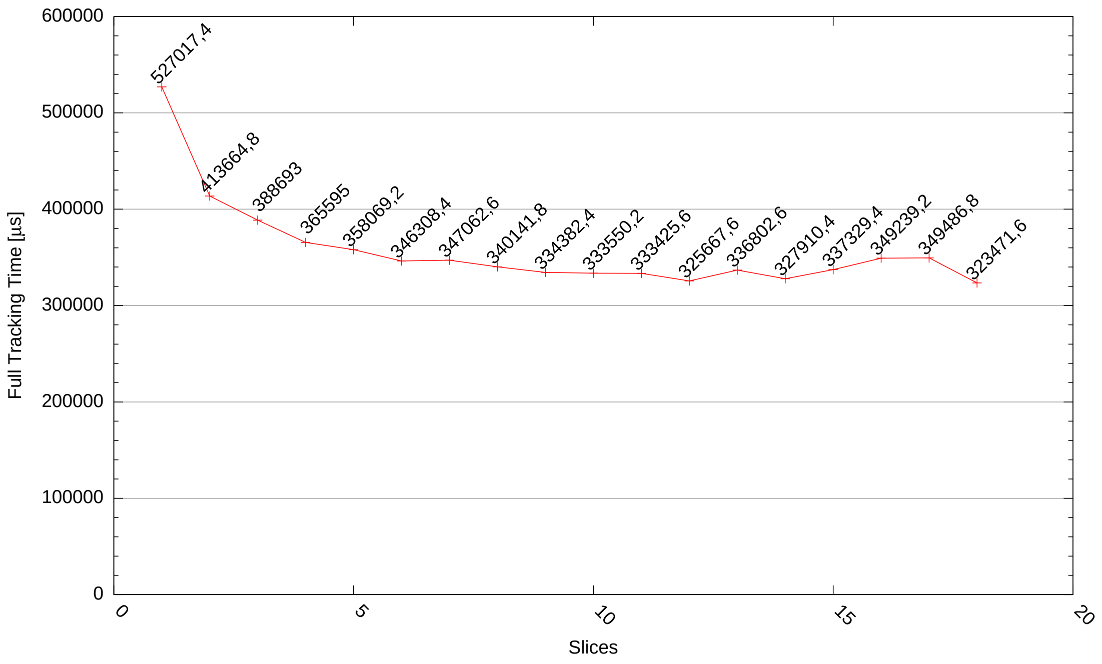


Figure 6.64: Pipeline Performance for Different Slice Numbers

6.3.3.3 Performance of Asynchronous Transfer

Up until now the performance benefits of pipelining and asynchronous memory transfer have been clearly illustrated. This section will deal with the effect of the asynchronous transfer itself. Besides the fact that DMA transfer and kernel execution may overlap, there is another major advantage of asynchronous memory transfers that has not even been mentioned yet. The synchronization of the pipeline is in fact much easier and more efficient when the memory transfer is asynchronous. For a synchronous transfer the GPU must not execute a kernel and the transfer must be supervised by the CPU, which means that at least the current CPU thread is also blocked by the transfer. CPU and GPU must be synchronized before the transfers can take place. This synchronization takes time in which neither CPU nor GPU can process data and especially where one has to wait for the other.

Figures 6.65 and 6.66 show workflow diagrams for synchronous memory transfer. Obviously the synchronization is much more expensive on a Windows platform. In contrast, Fig. 6.67 shows the initialization part of the diagram for the asynchronous case in more detail. It is clear, that besides the delay caused by the actual transfer, even more time is lost if not using the asynchronous transfer. The third figure also clearly shows that all three tasks overlap.

Fig. 6.68 finally shows a performance comparison of synchronous and asynchronous transfer for Windows and Linux platform. Clearly the asynchronous transfer is worth the effort for both platforms. Especially the Windows versions with synchronous transfer falls far behind.

[17] 36 slices would be possible on a Tesla card with more memory. However, there is no difference between 12 and 18 slices and thus 36 slices is also not expected to perform better.

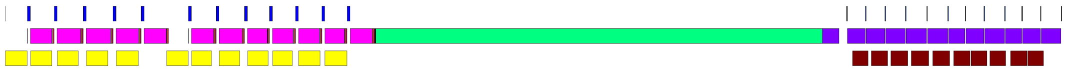


Figure 6.65: Synchronous Data Transfer (Linux Platform)

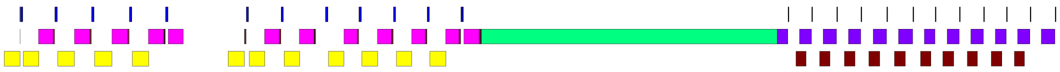


Figure 6.66: Synchronous Data Transfer (Windows Platform)

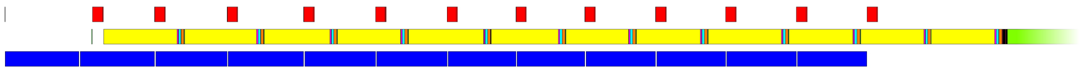


Figure 6.67: Asynchronous Data Transfer

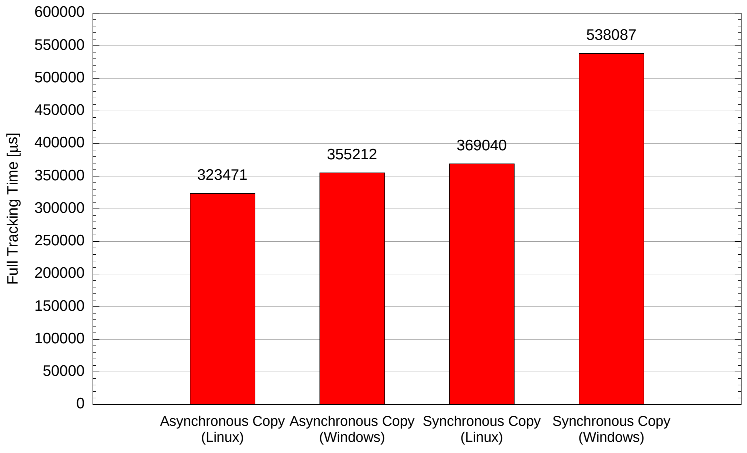


Figure 6.68: Asynchronous Transfer Performance

6.3.3.4 Zero Copy Memory

As of CUDA version 2.2, a new feature called Zero Copy was introduced. A virtual memory area on the GPU can be directly mapped to an area of physical page locked memory inside the host's main memory. The NVIDIA memory controller is built in a way that accesses to host and device memory are independent. Thus in theory the PCIe bandwidth of about 6 GB/s can be added to the 100 GB/s of GPU global memory bandwidth to obtain the maximum memory bandwidth. Clearly the delay for accessing host memory is much higher. The same coalescing rules still apply. However, fulfilling them is much more critical as the relative PCIe overhead is greatly reduced when transferring larger segments of memory instead of only 4 or 8 bytes.

To utilize this feature, the Tracklet Selector was changed to store the tracks directly in host memory. Unfortunately, these stores are not coalesced at all. In fact, they cannot easily be coalesced because all the threads process different tracks. The host access resulted in a ten-fold increase of the Tracklet Selector time. It is thus clear, that the transfer must be done blockwise, but then the current asynchronous transfer seems optimal. Therefore, the Zero Copy feature is not used.

6.3.3.5 Tracklet Selector Slices

This section will focus on which Tracklet Selector configuration to use to achieve optimal performance. When looking at the results for multi-slice performance (Fig. 6.48) it seems obvious to use as many concurrent slices as possible. However, when taking the asynchronous memory transfer into account, this is no longer clear, since Tracklet Output and Tracklet Selection cannot overlap if only one Tracklet Selector kernel is executed processing all slices. It has to be determined experimentally which slice count is optimal.

One additional optimization can be included. The slice count can be increased for every kernel execution. If starting with a slice count of one, the CPU can almost immediately start the Tracklet Output. Since the CPU is already occupied then the slice count can be increased speeding up the GPU execution. Figures 6.69, 6.70 and 6.71 show the rightmost parts of the workflow diagrams for all three versions.

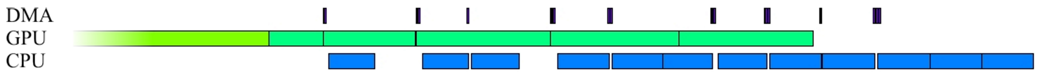


Figure 6.69: Pipelining with Increasing Tracklet Selector Slice Count

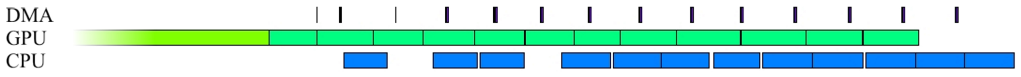


Figure 6.70: Pipelining using Single-Slice Tracklet Selector

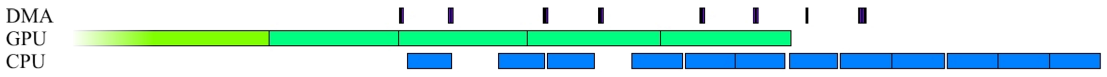


Figure 6.71: Pipelining using Fixed Three-Slice Tracklet Selector

Fig. 6.72 shows performance numbers for all configurations. It turns out that in the end the performance is only marginally affected by the choice with the single slice variant being the fastest. The explanation is that the time for one-slice Tracklet Selection and Tracklet Output match closely. Speeding up the Tracklet Selection does not decrease the total time, as the total time is simply given by the accumulated Tracklet Output time plus the delay produced by one Tracklet Selector kernel call. But the absolute time for one kernel execution is clearly minimal for a single slice. The experiment should be repeated if the Tracklet Output part could get accelerated.

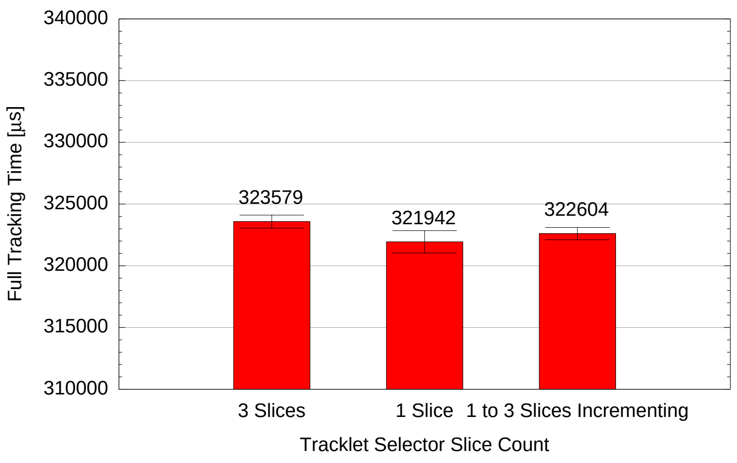


Figure 6.72: Performance for Tracklet Selector Operation Modes

Chapter 7

Optimizations for small events (pp)

Up until now all optimizations applied were targeted at huge events. Of course many assumptions taken in that case do not apply for pp events. As a GPU is designed for massive parallelism, big events seem appropriate for a GPU tracker. Additionally, small events can already easily be handled by the CPU tracker in the HLT compute farm. Nonetheless, some ideas how the GPU tracker could be changed for a fast processing of small scale events will be presented here.

First some preliminary considerations will be made. In [Kre] it was shown that the vectorized tracker could accelerate heavy-ion events, but was even slower for pp events. The overhead for the trivial parallelization over the slices increases with shorter execution times. As the CPU tracker performance is already affected by small event sizes, it is expected that there will be a large influence on GPU tracking time.

7.1 PP Performance of Heavy-Ion Tracker

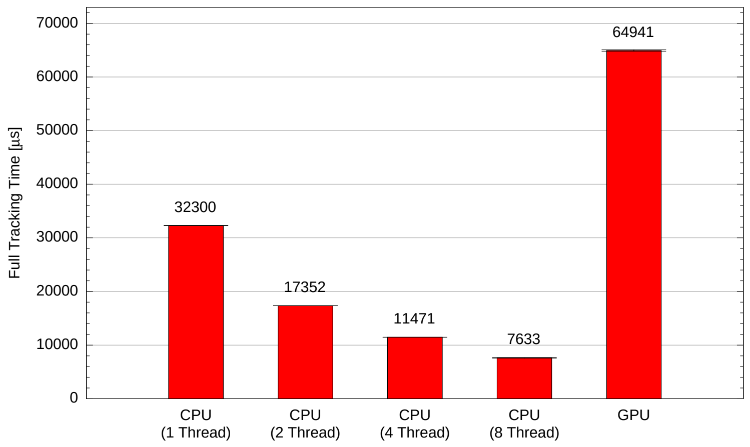


Figure 7.1: Performance for PP Event of Heavy-Ion Tracker

This chapter starts with a comparison of pp performance. Fig. 7.1 shows the result for the GPU tracker including all optimizations from the last chapter and results for the CPU tracker using different thread counts. The event contains 48, 189 clusters in which the CPU tracker found 223 tracks. Going to even smaller events really does not make any sense, as then the GPU's parallelism clearly cannot be exploited. The CPU multithreading performs surprisingly well (speedup by a factor of 4.2 for pp compared to 4.8 for heavy-ion when going from 1 to 8 threads running on 8 virtual Nehalem cores), while the GPU performance drops far below single threaded CPU speed. The scalability of the CPU tracker is clearly limited by the overhead for thread management. This does not apply to the HLT farm however, as there the tracker threads process data for one fixed slice. No threads have to be created.

7.2 Synchronization and Memory Transfer

For small scale events, kernel times and especially memory transfer times are extremely short. The delays for synchronization and kernel calls are constant. Furthermore, it seems reasonable not to launch too many kernels and especially not to have many synchronization points. The latter aspect refers primarily to the Tracklet Selector, which has one synchronization point per slice for obtaining the track count.

As a second fact for good utilization, it might also be interesting to process multiple slices in parallel for the Neighbors Finder. Since there is not much to schedule, the dynamic scheduling within the Tracklet Constructor should be deactivated. A look at the Tracklet Constructor in particular shows that for pp events there are less than 256 tracklets per slice present. To account for this the small scale tracker follows one particular paradigm: one multiprocessor per slice. This is realized for the whole tracker, so all tracking steps always process 30 slices in parallel (because of 30 multiprocessors). The pipeline is not included. All slices are initialized on the CPU. (One might think about parallelization here, but probably the overhead would be too big.) Then the slice data is transferred. The kernel for each tracking step is executed exactly once. Only one memory transfer is done for the track counts. However, transfers for the slice data and the tracks themselves are still separated and combined in a single transfer because they do not transfer consecutive memory segments.

The problem arising at this point is that there are 30 multiprocessors available on the GPU which does not match well to 36 slices. For efficient processing slices, of different events have to be intermixed. So the first run will process slice 0 to 29 from event 0, the second run slices 30 to 35 of event 0 and slices 0 to 23 of event 1, and so on. The process is shown in Fig. 7.5. However, this will be difficult to realize for the HLT farm in the real experiment because firstly, much data would have to be transferred to a single node, and secondly, the assignment of slices to nodes would be dynamic.

In the end the pp-tracker is a rather theoretical analysis. If it turned out that the GPU tracker also performed very well for pp events, further analysis could be interesting. But for the moment this shall just be a proof of concept approach to examine the possibilities. Fig. 7.2 demonstrates the simplifications for the pp-tracker using a workflow diagram.

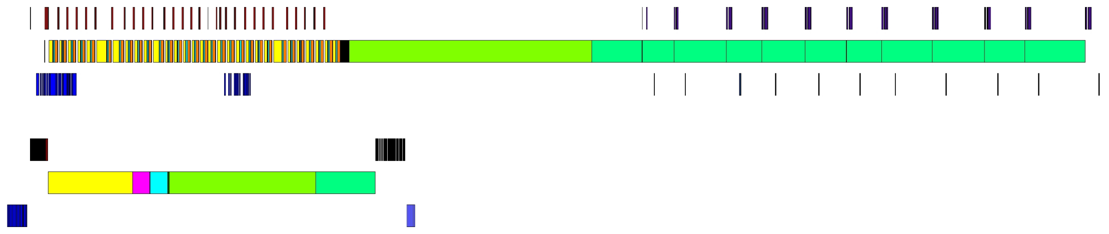


Figure 7.2: Workflow Comparison for Heavy-Ion- (Top) and PP-Tracker (Bottom) processing 30 Slices of a PP Event

7.3 Performance of PP-Tracker

Performance for PP Events Fig. 7.3 shows the performance gain that was achieved by the changes described above. CPU Tracking time is included for comparison. The adapted tracker can beat a single Nehalem core, but cannot compete with all four cores with Hyper-threading enabled. However, the pp performance of the pp-tracker is much better than of the heavy-ion-tracker.

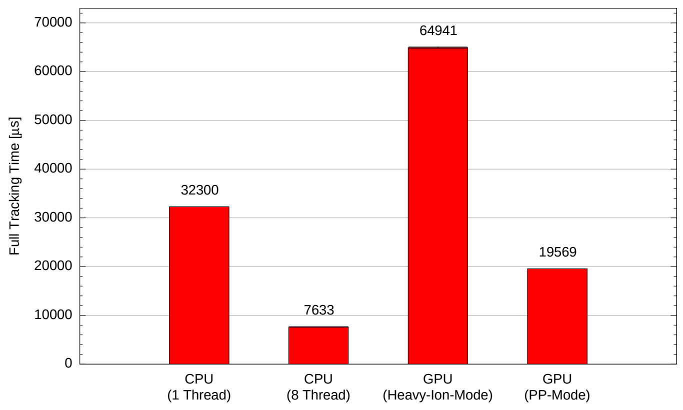


Figure 7.3: Performance for PP Event

Performance for Heavy-Ion Events Although not designed for lead-lead evens, the tracking time will still be analyzed to measure the performance loss. The difference for pp events is bigger than for heavy-ion events, where for lead-lead most time is lost because of a disabled pipelining. Clearly the tracking steps themselves perform well when running on 30 slices in parallel (although dynamic scheduling is disabled).

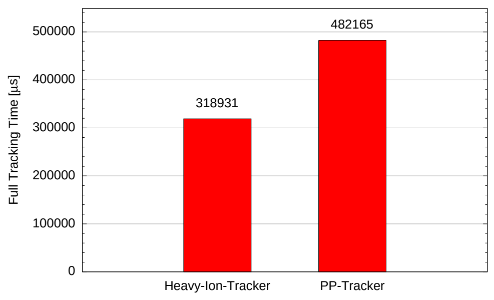


Figure 7.4: Performance for Lead-Lead Event

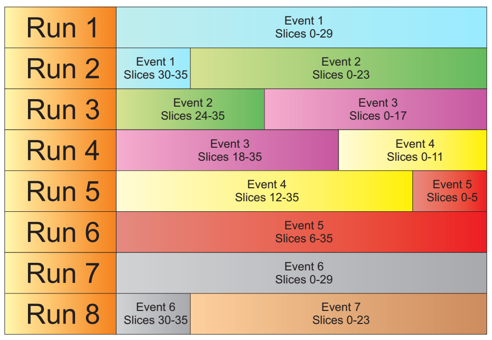


Figure 7.5: Slices Processed by PP-Tracker

Finally it can be concluded that the modus operandi of the heavy-ion tracker is anything but optimal for pp events. By some simple changes the pp performance could be improved dramatically, whereas the heavy-ion performance did not suffer in the same order of magnitude. However, the multithreaded CPU tracker still outperforms even the optimized GPU tracker for pp events. Additionally the changes required to improve the pp performance resulted in a negative effect on the applicability in the HLT farm. Especially running 30 slices in parallel is infeasible. For these reasons no plans exist to employ a GPU based pp-tracker in the High Level Trigger.

Chapter 8

Integration and Deployment in the HLT

8.1 ROOT

ROOT is an open source general purpose framework for data analysis with its focus originally set on particle physics. It is organized as a layered class hierarchy grouped in libraries with most classes inheriting from a common base class *TObject*. ROOT is developed at CERN and was initially developed to build a framework capable of handling the enormous amount of data produced at the LHC. See [Roo] for details.

ROOT offers a C++ interpreter (CINT) and third party programs employing ROOT can either run natively if compiled and linked against the libraries or use the interpreter.

8.2 AliRoot

AliRoot is the ALICE Offline framework for data analysis, event reconstruction, and simulation. As the name suggests, it is based on ROOT and some other third party libraries such as Geant and Fluka. AliRoot can simulate data as expected to be delivered by the ALICE detectors, and can also include detector inefficiencies. Furthermore, AliRoot has the ability to perform a full event reconstruction. Reconstruction algorithms are divided in online and offline algorithms. In general, Online algorithms are fast and will be executed while the experiment is running for triggering and online analysis. Offline algorithms are generally slower but in many cases more accurate and are used for further data analysis.

8.3 PubSub

The PubSub framework based on the publisher / subscriber principle was designed for fast data transport and controls the components of the High Level Trigger. Components can be data processing components, data sources, or data sinks. Sources normally are the detector outputs but they can also be raw files for simulation and debugging. Sinks are components that receive data but do not process the data nor pass the data to further components (within the framework). An example for a sink is a file in which the output of a data processing component is stored. Every data processing component can subscribe to other data processing components or to sources. So for the case of the tracker there are sources, either 216 FilePublishers or 216 RORCPublisher (for the TPC readout) delivering the input data for the 216 TPC pads. 216 cluster finder components subscribe to these sources. Furthermore, 36 slice trackers are instantiated, each subscribing to the 6 Cluster Finders responsible for its slice. Finally, one merger subscribes to the 36 slice trackers and possibly writes its output to a file sink afterwards. Obviously, the HLT structure resembles a tree where the data of many sources is combined on the way towards a sink. For more detailed information see [Ste].

8.4 HLT Libraries

All components responsible for event reconstruction are stored in libraries. These libraries have a common ANSI C interface and can be used in the PubSub framework as well as in AliRoot. In fact, AliRoot has an HLT simulation mode, in which it uses the HLT libraries and simulates an online reconstruction. All these HLT libraries rely on ROOT.

8.5 Interface

The question arises where to place the GPU tracker in all these frameworks. Clearly the GPU tracker has to run as a component in the PubSub framework and should therefore be placed in an HLT library. Furthermore, it should be possible to run the GPU tracker also from within AliRoot, which is already possible when present in the HLT library.

The user should be able to transparently run both the CPU and GPU tracker (if a GPU is available) without any changes to the calling procedure. Therefore, a generic tracker framework was developed, providing a common interface and being able to process multiple slices in parallel (as needed for the GPU tracker). The framework is capable of running the GPU and the CPU tracker. It could be extended in the near future to support both a multithreaded CPU tracker and a vectorized tracker version.

This new framework can be used as a component in the HLT. Furthermore, AliRoot implements its own procedure for tracking, which was adapted to use the new framework instead of the CPU tracker itself. As a third variant, mainly for debugging purposes, a standalone version of the tracker was created containing the newly developed event display. The standalone tracker does not rely on AliRoot at all but can still use the tracker framework.

The framework has the ability to detect the availability of GPUs. Then it runs the GPU tracker automatically or falls back to the CPU version if no GPU present. Of course, one can also define explicitly which tracker version to run.

8.6 C++

The HLT libraries rely on ROOT and most of them inherit from *TObject*, as did the CPU version of the tracker. CUDA does not offer full C++ support yet. For this and other reasons, it is not possible to use ROOT on the GPU. All functionality that relied on ROOT was replaced by inline wrapper functions. Using conditional compilation, they can be directed to the ROOT functions or to a replacement for the GPU and the standalone case. The inheritance of ROOT objects could be removed from the tracker classes. However, this does not mean that the tracker is plain C code but only direct references to ROOT were removed.

8.7 Compilation

The HLT libraries are usually compiled within AliRoot. However, there is also the possibility to build them standalone. Both make scripts had to be changed to allow for compilation of CUDA files. Another problem is that the debug message system relies on ROOT. Special ROOT macros have to be present in the header files for the debug message macro to identify the class and method name in which a debug message is produced. For this the source files must be interpreted by CINT, but this is not possible for CUDA sources. This is solved by letting CINT interpret a fake file with all CUDA content removed and build its internal lists accordingly.

8.8 Running in an HLT Chain

A configuration with sources, data processing components, and sinks is called an HLT chain. The first attempt to run the GPU tracker inside an HLT chain failed because the CUDA context is thread local. But within the HLT chain there is no guarantee that the component's methods are called from one single thread consistently. Therefore, a check for the thread ID had to be included, which forces a GPU reinitialization if the thread has changed.

Naturally the GPU tracker requires the CUDA runtime library. This is a C++ library that has to be bound when starting the tracker and when loading the HLT tracking library respectively, and does not support late binding. On the HLT farm all libraries are stored on a shared file system (AFS). A problem arises now, as the CUDA library is not present on each node. If the HLT tracking library is linked against the CUDA runtime library, it cannot be loaded on nodes without CUDA library installed. However, it is not desired to install the CUDA library on every node.

Therefore, the GPU tracker class is realized as an abstract class. There are two derived classes: the actual GPU tracker class and a dummy class. The GPU tracker class is stored in an additional library which exports plain C functions to create and destroy a GPU tracker instance. The tracker framework in the HLT library will check the availability of the CUDA runtime library. If it finds the runtime, it loads the GPU library with "*dlopen*" and use the plain C interface to create an instance of the GPU tracker class. Otherwise, it creates an instance of the GPU tracker dummy class. This instance with the abstract GPU tracker class interface can now be used inside the HLT library. The hierarchy of classes and interfaces is shown in Fig. 8.1.

8.9 Hardware

The GPU tracker is supposed to run on GeForce GTX295 dual GPU cards in Super Micro servers with Intel Nehalem CPUs. The GTX295 boards have slightly reduced clock speeds compared to the GTX285 cards used for the benchmarks, but instead offer two independent GPU chips with independent memory on a single card. The power connectors of the servers had to be adapted in order to plug in the NVIDIA cards. Because the CUDA context is thread local, a direct multi GPU implementation would be at least difficult to realize. Instead, it was decided to run multiple instances of the GPU tracker component on one node. During the initialization, the GPU tracker class queries all available GPUs for available memory, selects the GPU with the most memory available, and reserves the memory it requires. In

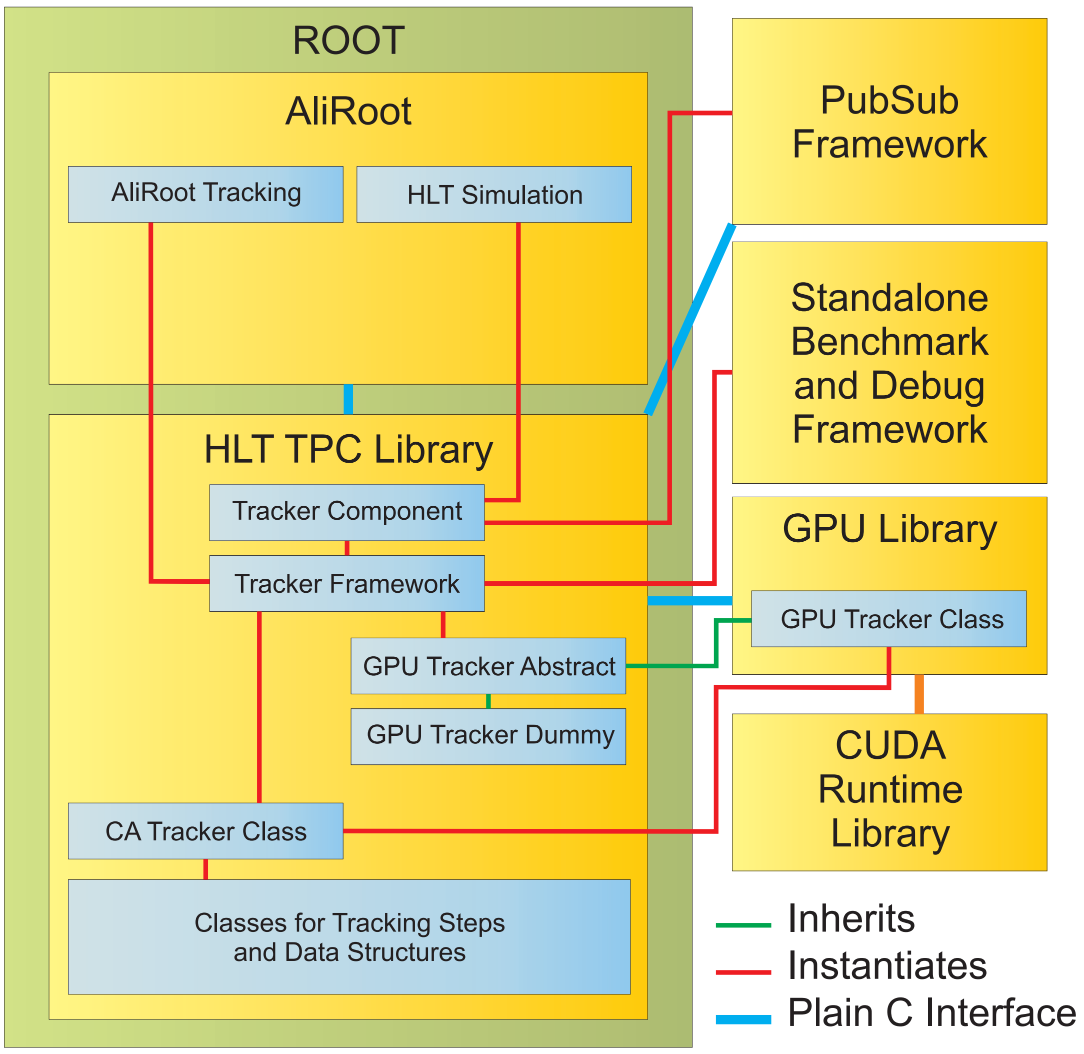


Figure 8.1: Diagram of Libraries and Class Hierarchy

this way, it is ensured that two components will use different GPU chips.[1] It is even possible to instantiate more than two GPU trackers (per one GTX295 card). As long as the GPU memory is sufficient, multiple components can share one physical chip. This can be used to hide latencies produced by data transfer in the PubSub framework.

One node with a dual GPU card has been installed in the HLT farm. It was tested with six GPU tracker components, of which the first four were automatically placed on the GPU board (two per chip), while the remaining two automatically fell back to the CPU since GPU memory was insufficient. The GPU tracker was running successfully for several hours in ERP test runs. Unfortunately, due to network problems with the node, the GPU has not yet taken part in physics runs.

[1]Mutual exclusion based on system wide named semaphores is used to ensure synchronization of the threads when querying the GPUs. There is also a method to lock the GPU on the driver side for controlling access to it. But this would restrict GPU access to a single component.

Chapter 9

Results

The tracker was benchmarked according to two criteria: the tracking quality and the tracking performance. The tracking quality is measured in terms of efficiency, clone and fake rates, and resolution which will be described in detail in the next section. Tracking performance is simply proportional to the reciprocal of the tracking time. A tracking efficiency of 100% is unrealistic. Therefore, a lower bound should be set for the quality and then the performance maximized according to this quality assurance demand.

It is evident that the GPU tracker is not able to beat the CPU tracker in tracking quality, since the algorithm is essentially the same except for some drawbacks for the GPU such as incoherent memory and a non 100% compliance with the IEEE floating point standard. The goal is to maintain the CPU tracking quality, at best to achieve a bitwise match of the result.

9.1 Tracking Quality

9.1.1 Efficiency, Resolution, and Pull

The tracking quality is measured for several categories:

- Efficiency (the percentage of Monte Carlo reference tracks found by the tracker)
- Clone Rate (the ratio of two tracks found by the tracker representing identical Monte Carlo tracks)
- Fake Rate (the ratio of tracks not even corresponding to a Monte Carlo track)
- Resolution (resolution of the fitted track parameters)
- Pull (for simulated data the pull is the difference between the parameters from the Monte Carlo tracks and the parameters found by the trackers normalized by the error expected according to the covariance matrix)

Clearly a low clone rate is wanted, but this is not as important as an efficiency of almost 100% and a fake rate close to 0%. A framework for measuring tracking quality was developed by Sergey Gorbunov. The results for the CPU and GPU tracker are shown in Tab. 9.1. Only a very tiny advantage for the CPU variant can be seen.

In a more detailed analysis, the efficiency and fake rate can be compared for different P_t (transversal momentum, see 2.1.4) values (Figures 9.4 and 9.7). Furthermore, the resolution (Figures 9.5 and 9.8) and the pull (Figures 9.6 and 9.9) for the track parameter Z are analyzed for the GPU and CPU version exemplary. Figures 9.10 to 9.15 show the same plots in the pp case. As can be seen, all diagrams for GPU and CPU resemble each other. The resolutions and pull for the other track parameters are equal for both the CPU and the GPU version, too. Resolutions for heavy-ion events are shown exemplary in Figures 9.2 and 9.3.[1] Finally it can be concluded that the tracking quality for the CPU and GPU version is comparable for both heavy-ion and pp events.

Processor	Event	Efficiency [%]	Fake rate [%]	Clone Rate [%]
CPU	pp	100.000	0.138	6.061
GPU	pp	99.853	0.138	6.069
CPU	heavy-ion	99.028	1.063	10.897
GPU	heavy-ion	98.954	1.130	10.897
CPU	central heavy-ion	89.962	6.814	14.825
GPU	central heavy-ion	89.962	6.804	14.816

Table 9.1: Tracking Quality

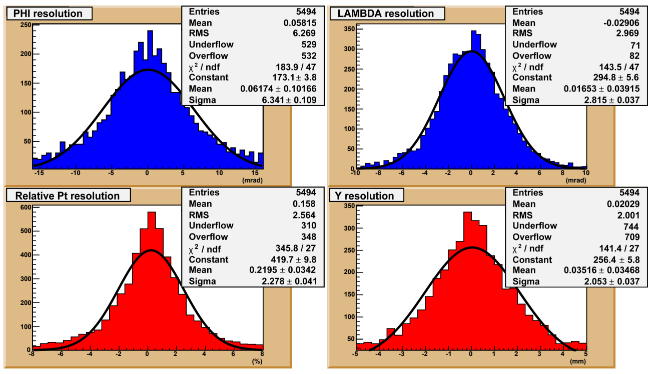


Figure 9.2: ϕ, λ, P_t and Y Resolutions of CPU tracker (Central Heavy-Ion)

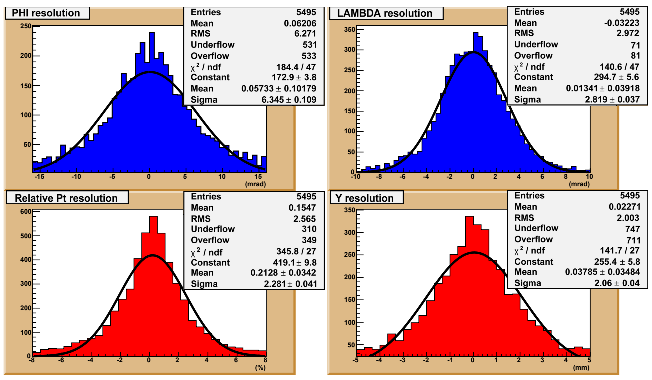
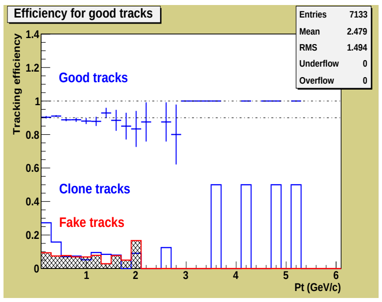
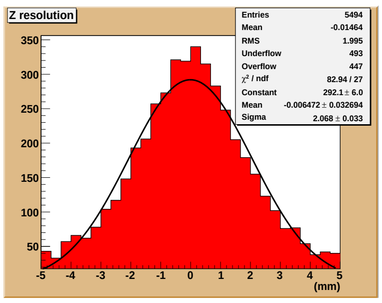


Figure 9.3: ϕ, λ, P_t and Y Resolutions of GPU tracker (Central Heavy-Ion)

[1] All the Figures are arranged in a way, such that CPU and GPU results for the same data are one upon the other.





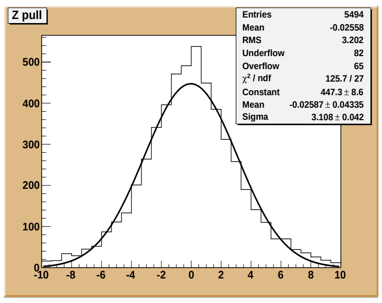
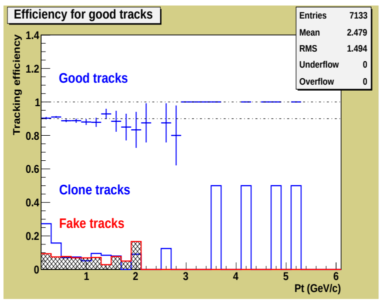
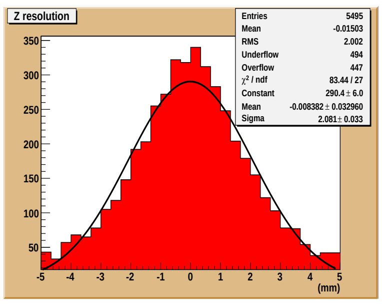


Figure 9.4: CPU Tracker Efficiency and Fake Rate (Central Heavy-Ion)

Figure 9.5: CPU Tracker Z Resolution (Central Heavy-Ion)

Figure 9.6: CPU Tracker Z Pull (Central Heavy-Ion)





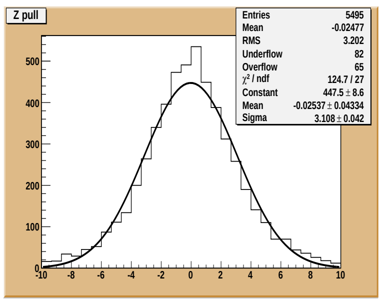
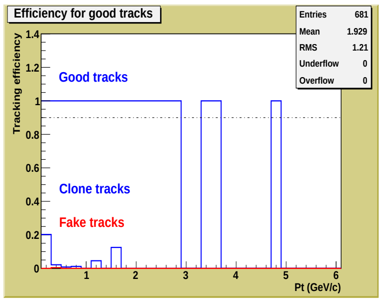
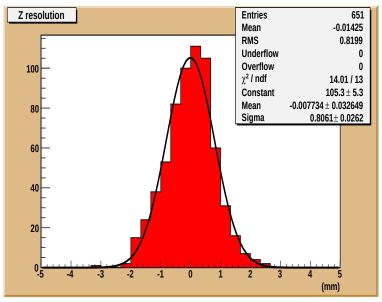


Figure 9.7: GPU Tracker Efficiency and Fake Rate (Central Heavy-Ion)

Figure 9.8: GPU Tracker Z Resolution (Central Heavy-Ion)

Figure 9.9: GPU Tracker Z Pull (Central Heavy-Ion)





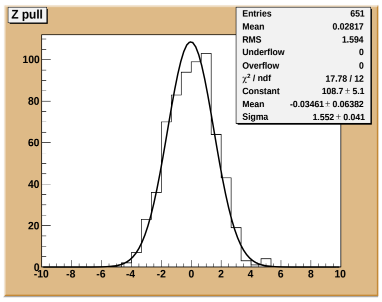
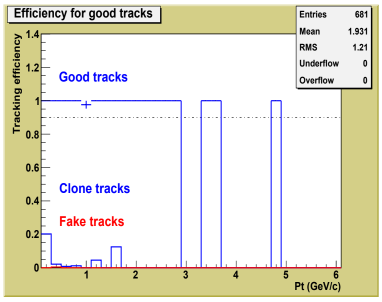
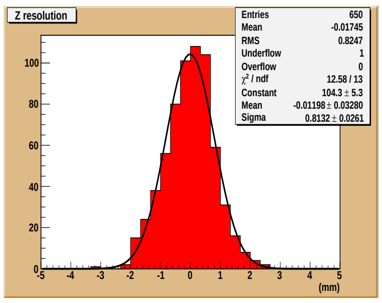


Figure 9.10: CPU Tracker Efficiency and Fake Rate (PP)

Figure 9.11: CPU Tracker Z Resolution (PP)

Figure 9.12: CPU Tracker Z Pull (PP)





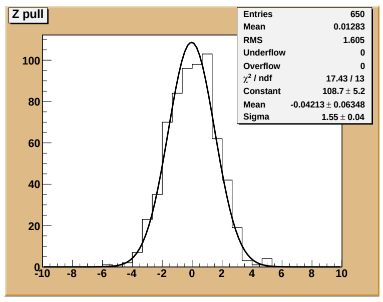


Figure 9.13: GPU Tracker Efficiency and Fake Rate (PP)

Figure 9.14: GPU Tracker Z Resolution (PP)

Figure 9.15: GPU Tracker Z Pull (PP)

9.1.2 Bitwise Comparison

Having shown that the GPU tracker quality is already satisfactory, in a second step the tracks found by the GPU and CPU tracker are compared directly. The problem is that the GPU tracker is not 100% deterministic due to parallelization: Clusters are assigned to the longest tracklet possible. But if two tracklets share the same length clusters they are assigned by the first come first serve principle. Since the tracklet order is not deterministic neither is the cluster assignment.[2] Additionally the floating point arithmetic is not completely IEEE conform.

For the CPU version, however, there is a well defined result as parallelization is done over the slices only. Fig. 9.16 shows the number of tracks found by the GPU tracker for multiple runs over one set of input data.

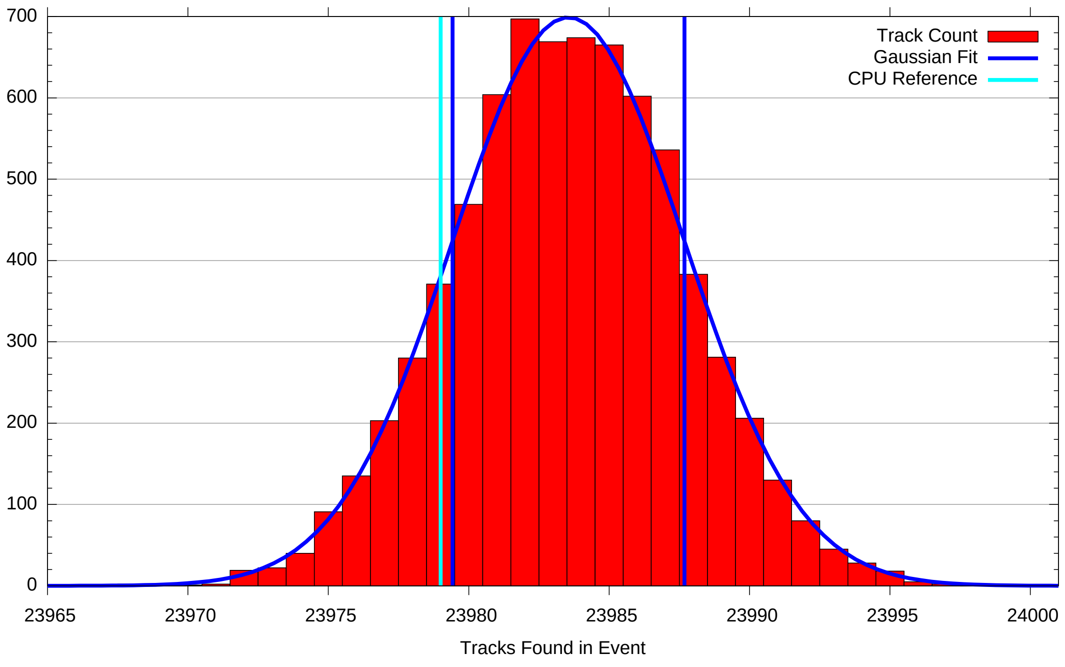


Figure 9.16: Histogram of Number of Tracks Found by GPU
(23979 Tracks Found by CPU Tracker)

Obviously, there is some distribution of the track count, resulting from the indeterministic processing. The distribution has a gaussian like shape, and the reference count of the CPU tracker lies slightly outside a one-σ deviation. It is clear that the tracks cannot be compared directly to the tracks found by the CPU tracker. To enable a meaningful comparison, the following method was used. Tracking steps I to III were run on the CPU and GPU in parallel. Start hits of the GPU and CPU tracker were found to be identical, but arranged in different orders. Now the CPU start hits were rearranged so the sequence of start hits was identical. Afterwards steps IV and V were executed. The final tracks were then compared bitwise, and were found to be close to perfect matches. The advantage is that even though the floating point arithmetic is not completely consistent, the Tracklet Constructor searches for the cluster next to the extrapolation point. As long as the GPU and CPU extrapolation points do not differ too much, the same cluster will be found. Tests show that in only about one out of $10,000$ cases, the cluster index differs. The distribution of the track count therefore results from differences of the start hit arrangement.

[2] Also the CPU trackers output changes, if the start hits are rearranged.

9.2 Tracking Performance

9.2.1 CPU Performance

For obtaining the reference performance, first some CPU-only benchmarks will be shown to determine the optimal CPU tracker configuration. The parameters to find are: thread count, compiler, and operating system. Fig. 9.17 shows a plot using different thread counts. The Nehalem CPU employed offers 4 physical cores but 8 virtual cores (because of Hyperthreading). As expected, the performance increases strongly up to 4 threads, while the increase when going to 8 threads is still not negligible. Fig. 9.18 shows a more detailed plot for the high thread counts only. It can be seen, that 8 threads are still not optimal. This is probably related to the fact that 36 is not a multiple of 8. When distributing the slices among the threads, 4 threads will have to process 4 slices, while the remaining 4 threads will have to process 5 slices. This distribution works better for 12 threads. (Benchmarks up to here were run using GCC on a Linux system)

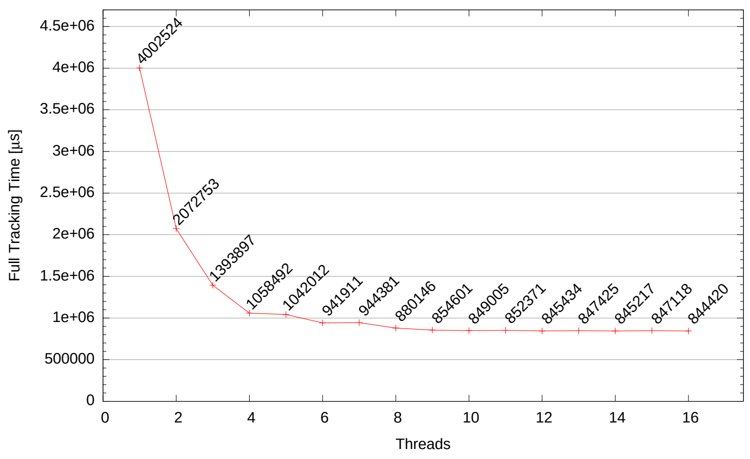


Figure 9.17: CPU Tracker Performance using Different Thread Counts

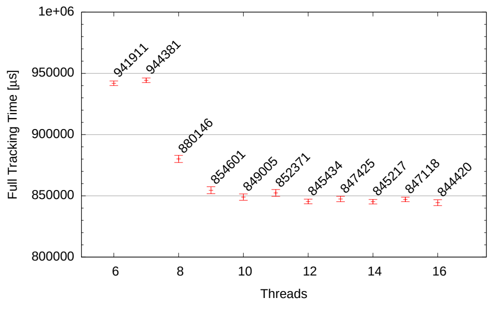


Figure 9.18: CPU Tracker Performance for Big Thread Counts

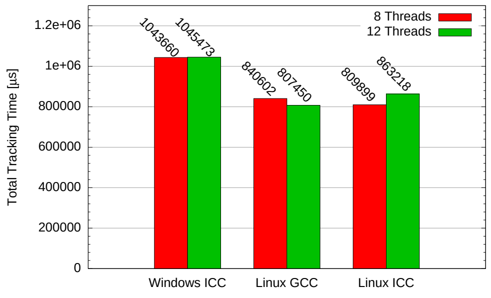


Figure 9.19: CPU Tracker Performance using Different Compilers

In a second step, the compiler and the operating system is analyzed. Fig. 9.19 shows the relevant data. The Linux platform is clearly faster than the Windows platform. This is good news as the HLT farm is based on Linux. An interesting fact is that 8 threads are optimal for ICC, in contrast to the GNU compiler. A possible reason is a more sophisticated OpenMP implementation within ICC, as the Intel OpenMP and TBB[3] multithread libraries (see [Int1]

[3]TBB stands for Intel Threading Building Blocks.

and [Int2]) offer a load balancing for the threads, but do not assign the slices to fixed threads. In this way, the ICC can already achieve optimal performance for 8 threads. In the end, it turned out that ICC and GCC performance are on a par (with a 0.3% margin for GCC). As GCC is also the compiler used in the HLT farm, the CPU benchmarks will be executed on a Linux system employing GCC (version 4.3.4) running with 12 threads.

9.2.2 GPU / CPU Comparison

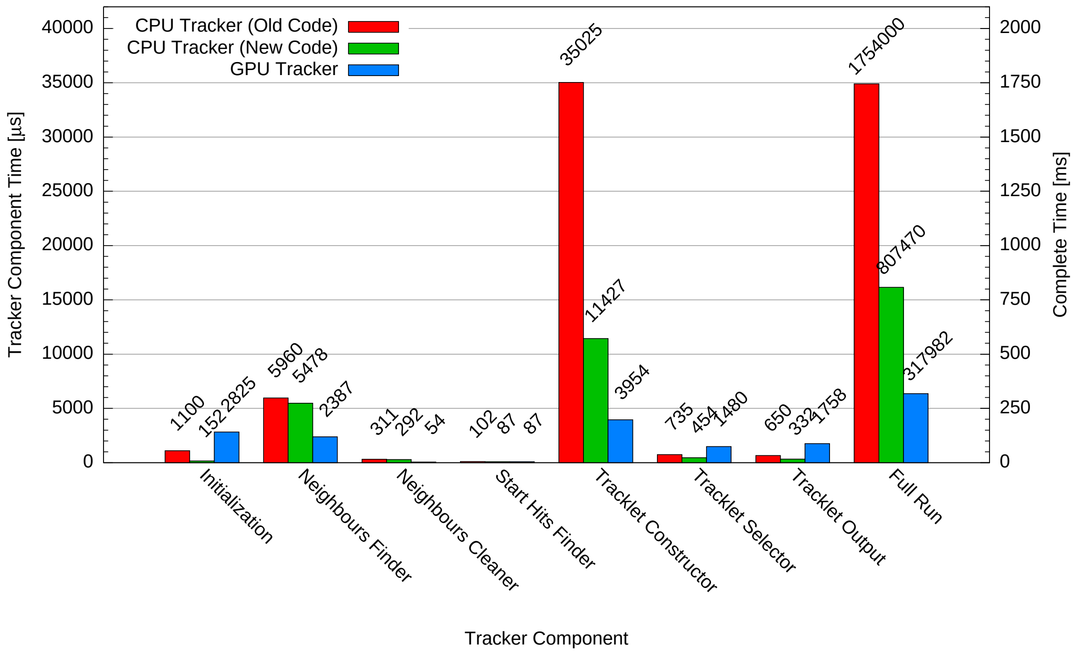


Figure 9.20: GPU Tracker Speedup

Fig. 9.20 shows the final speedup for the most recent GPU tracker version compared to the fastest CPU version available. The benchmark is done using the reference heavy-ion event with 23979 tracks. Because of the pipeline employed, the results for the single tracking steps do not sum up to the full tracking time. The result for the initial CPU code at the beginning of this work is also included. To get a fair comparison, the initial version was at least multithreaded over the slices. For the CPU code an almost twofold increase can be seen, originating from memory improvements. Tab. 9.21 shows the final GPU speedup.

CPU tracking time	807470	±359	[µs]
GPU tracking time	317982	±23	[µs]
Speedup	2.5394	±0.0011	%

Table 9.21: Final GPU Tracker Performance

The performance will be analyzed for a couple of different platforms and event sizes in the following section.

9.2.3 CUDA Version

Fig. 9.22 shows that performance for different CUDA versions is almost identical. Surprisingly the older version (2.2) is even slightly faster than the current version.

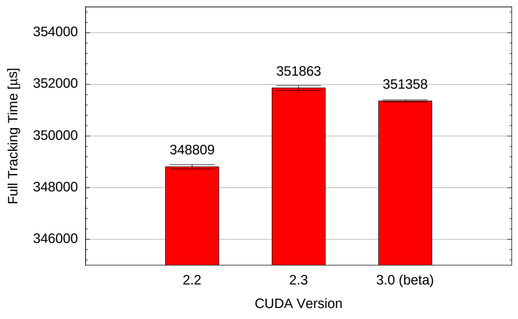


Figure 9.22: Performance for Different CUDA Versions

9.2.4 Performance on Different Architectures

Fig. 9.23 shows CPU and GPU performances for different clock speeds. It is not surprising that the GPU tracker is affected by the CPU clock speed, as the initialization and Tracklet Output steps are processed by the CPU and not the GPU.

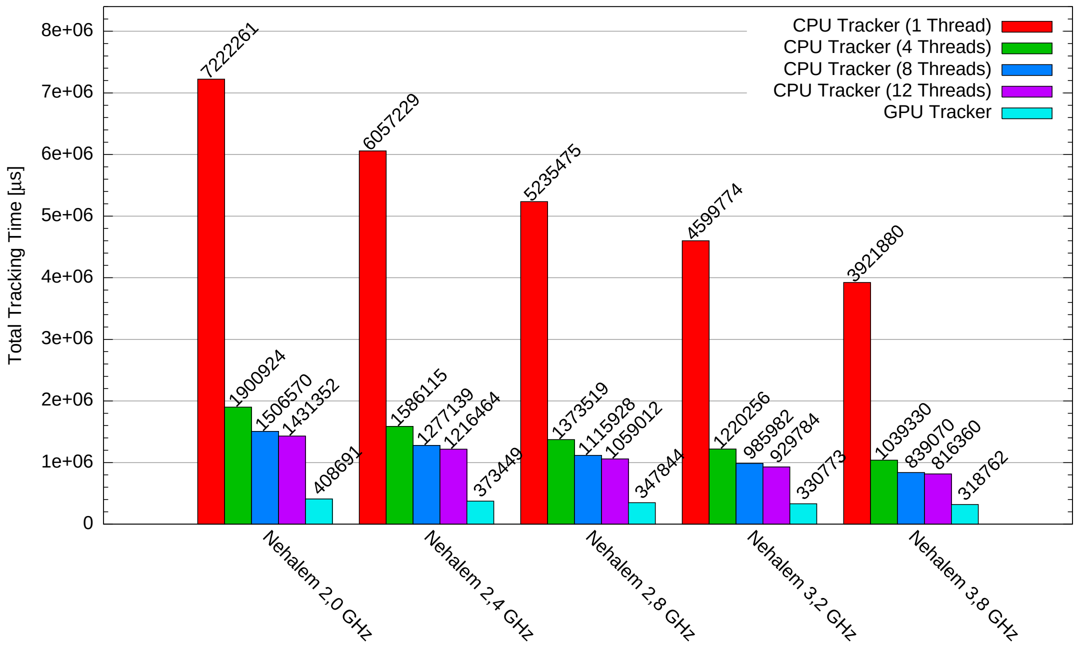


Figure 9.23: Performance Dependency on CPU Clock Speed

Fig. 9.24 shows a variety of benchmarks on different host systems, operating systems, and GPU chips. It can be seen that in every combination the GPU tracker is significantly faster. The fastest CPU version is running on 12 CPU cores which is very much in contrast to a single GPU chip.

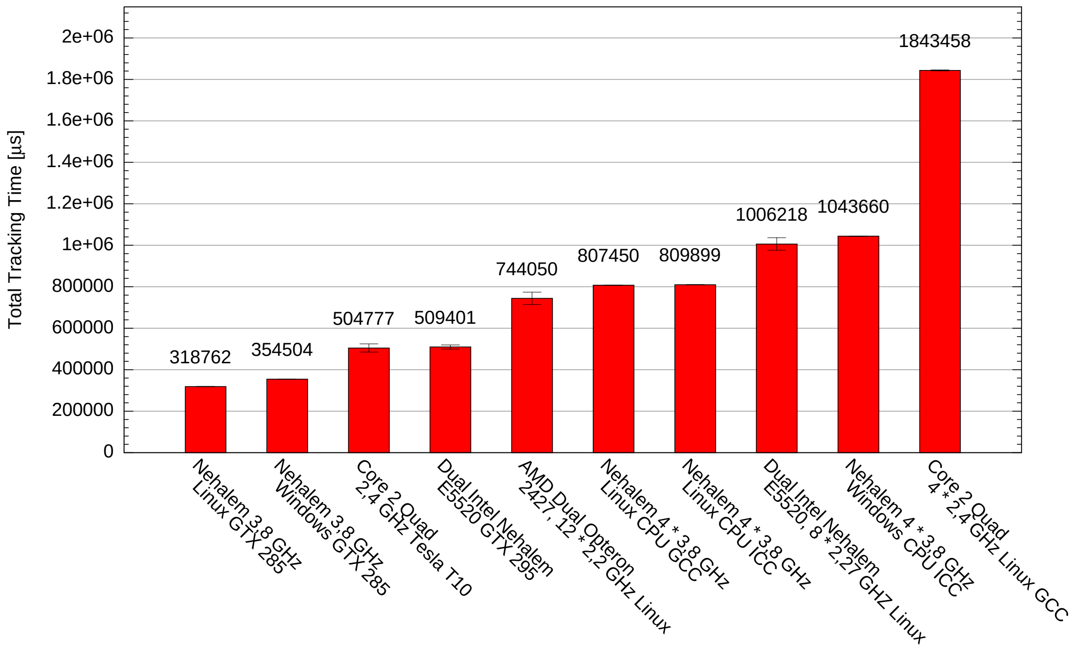


Figure 9.24: Performance on Different Architectures

9.2.5 Conclusion (for Heavy-Ion Events)

It was shown that the GPU easily outperforms every CPU available today. When doing a performance per price comparison, the GPU's advance even grows. The CPU version ran on a 12 Opteron cores and a Nehalem overclocked to 3.8 GHz, which is faster than any processor on the market today. The closest CPU is the 3.33 GHz Nehalem which costs about \$1000.[4] A more realistic CPU for clusters would run at 2.0 GHz or eventually 2.4 GHz costing about \$200. Then the GPU tracker advantage is even greater.

The compute nodes in the cluster possess two quad core CPUs each offering 16 CPU cores with Hyperthreading included. This is a problem as it was already stated that running too many slices in parallel on one node is undesirable because of the limited network bandwidth. In contrast, it was seen that the GPU tracker already shows good performance for 4 slices.

It can be argued that naturally a host is required in order to plug in the GPU. However, the cluster nodes are present anyway, and performance can be increased dramatically by plugging in a \$400 graphics card. Furthermore, the GPU tracker only causes a small load on the CPU which can therefore handle other tasks as well.

9.2.6 Performance for Different Event Sizes

Finally the dependency on the input cluster count will be analyzed. Here the pp-tracker which was specifically optimized for small cluster counts is again relevant. Fig. 9.25 shows the performance for the CPU tracker as well as both the pp and the heavy-ion version of the GPU tracker as a function of input cluster count. All three versions show a linear runtime with an offset. As expected the heavy-ion GPU tracker's offset is the highest one, since this tracker has many kernel calls and synchronization points. A further analysis will be done

[4]Cluster nodes usually use the server variant, which is even more expensive.

soon, but first it will be verified whether the cluster count really represents the right reference, or whether the track count would be more appropriate.

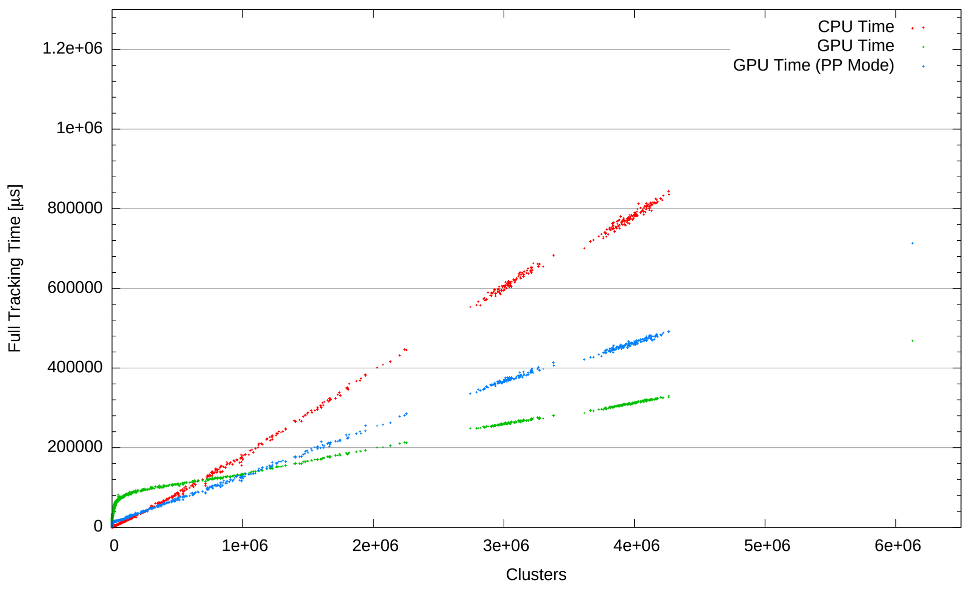


Figure 9.25: Tracker Performance compared to Number of Clusters

Fig. 9.27 shows the relation between cluster and track count. This relation appears to be almost linear, and in fact it is, as long as the data is produced using one particular AliRoot simulator version^4. In different versions, parameters for the cluster finder were changed, resulting in more clusters but the same number of tracks. The data shown in the plot was created in two major and two minor simulation runs. Therefore, different linear regions exist. Fig. 9.26 shows the same figure as Fig. 9.25, but as a function of the track count. It also looks mostly linear with the same exceptions as Fig. 9.27 displays. It becomes apparent, that neither cluster nor track count are a perfect reference, but the tracking time seems to depend on input cluster count rather than on track count.

This may sound confusing at first because the Tracklet Constructor time would usually be assumed to be proportional to the track count. However, this is not the case. More clusters will obviously result in more start hits and thus in more tracklets. In the end the number of tracklets representing one identical track will only be higher, but as they are combined in the merger (merging the track segments of the different slices but also tracks within one slice) the final track count is not influenced and thus the track count is no good reference.

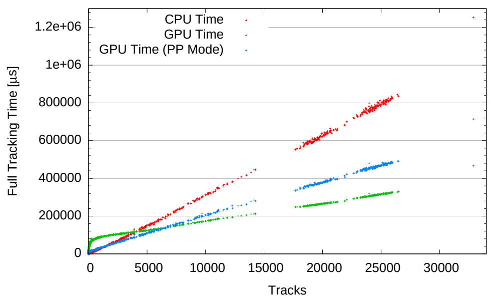


Figure 9.26: Tracker Performance compared to Number of Tracks

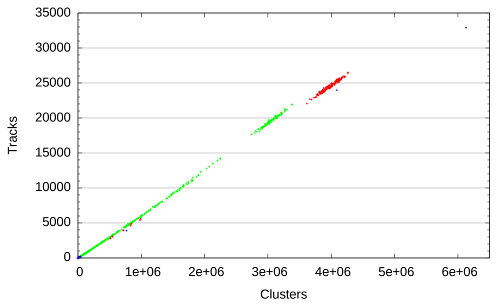


Figure 9.27: Clusters / Tracks Relation^4

For a closer look on small cluster counts, Figures 9.28 and 9.29 show the same diagrams again with logarithmically scaled axes. There is a small range in which the pp-tracker is the fastest version which is the interval between 2000 and 5000 tracks.

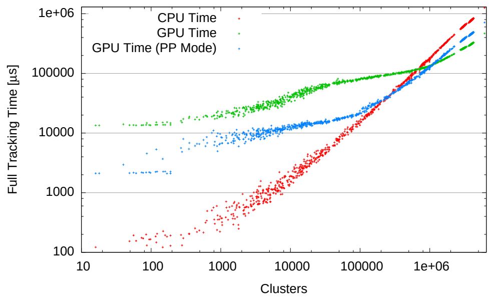


Figure 9.28: Tracker Performance compared to Number of Clusters (Logarithmic)

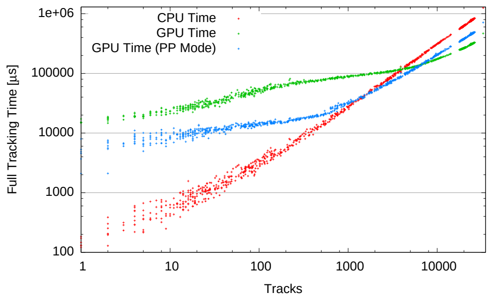


Figure 9.29: Tracker Performance compared to Number of Tracks (Logarithmic)

As all versions show a linear behavior, for each variant the performance is proportional to the reciprocal of the slope when looking at the limit for the cluster count going to infinity. Fig. 9.30 shows linear fits where only events with 500000 clusters or more are regarded as they represent the linear region. Interestingly the CPU fit has a negative offset. This can be deduced to cache effects slowing down big input data sets slightly. The fit results are shown in Tab. 9.31. By taking the ratio of the slopes, the theoretical speedup in the limit for a high cluster count is **3.3993 ± 0.0098**.

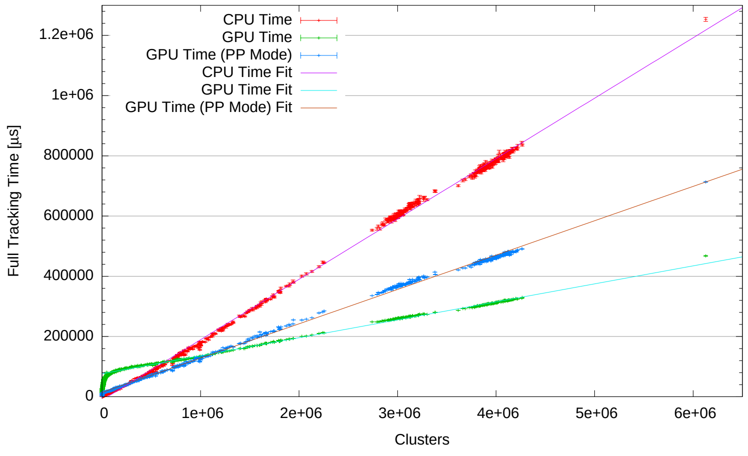


Figure 9.30: Linear Fit to Tracker Performance

-	Slope [µs / cluster]	σ_{Slope}	Offset [µs]	σ_{Offset}
CPU tracking time	0.210021	0.000397	−28589.8	868.1
GPU tracking time	0.061783	0.000134	73597.0	293.7
GPU tracking time (pp mode)	0.118078	0.000253	13219.4]	552.0

Table 9.31: Fit Parameters

Chapter 10

Summary and Perspectives for the Future

10.1 Summary

An adaptation of the GPU tracker algorithm was presented in this thesis. Many kinds of optimizations were discussed, some of rather general concern, while others are closely related to the GPU tracking algorithm. The critical parts turned out to be a good GPU utilization inside single kernels as well as a good overall utilization. The first aspect could by reached by the introduction of dynamic scheduling while the latter could benefit tremendously from the pipeline introduced. The tracking quality and efficiency were demonstrated to be in no way inferior to the CPU version.

A twofold speed increase for the CPU tracker was reached as a side effect and originated from memory optimizations. A new framework was developed to run both CPU and GPU tracker. Except for some wrapper functions, the CPU and GPU tracker share a common source code, greatly improving the maintainability. In most cases, changes to the original CPU code could be avoided.

The tracker has been integrated in the AliRoot and HLT framework with one machine currently installed at CERN for testing purposes. The GPU tracker will play to its strength when heavy-ions will start colliding in the LHC. Up to that time, the GPU node will be maintained in the farm for testing and validation so that when the time has come the cluster can be upgraded with the best performing cards then available.

10.2 NVIDIA Fermi

The next generation of NVIDIA GPUs is supposed to be presented to the public in March 2010. The number of shaders (ALUs) will be changed from 240 to 512 and memory bandwidth will also increase. At a first glance an increased ALU count seems ill-suitable for the GPU tracker, as was very difficult to keep all the ALUs in operation even for the current generation. However, NVIDIA announced an improved scheduling, so that only half the number of threads per ALU as before will be needed for fully exploiting the GPU potential.

Even more important than mere computing power is the fact that the architecture was updated with HPC applications in mind. The memory now supports ECC[1] and the double precision capabilities were greatly improved (which is a great step in general, but does not affect the tracker). The new CUDA framework officially supports C++. Finally, a general purpose L2 cache of 768 kB is introduced together with 64 kB of a mixed memory, which can be configured as shared memory or L1 cache.

This increased amount of shared memory will directly address issues in the tracker, especially for the Neighbors Finder, for which benchmarks already showed a significant improvement if more shared memory was available. Clearly the tracker will benefit from the L2 cache, too.

10.3 SIMDization

With the introduction of the Pipeline it turned out that CPU performance is not negligible for the GPU tracker either. At the moment the initialization and Tracklet Output times on the CPU correspond to the tracking steps processed on the GPU in parallel. With further improvements on the GPU side, e.g., with an improved Neighbors Finder on the next GPU generation, these CPU steps will become time critical. As these CPU tasks principally only do data reformating and rearrangement they should be well suited for vector instructions. Unfortunately, at the moment only a vector implementation for the actual tracking steps I to V (see [Kre]) exists which are processed on the GPU already. Vectorization should be extended to the initialization and output step, especially since both CPU and GPU versions could benefit from this update.

10.4 Seed Merging

One idea that emerged but has not yet been analyzed is the possibility to merge seeds even before they are processed by the Tracklet Constructor. Because the Neighbors Finder skips one row, most tracks result in two seeds, with clusters in even and odd numbered rows respectively. A merge is clearly non trivial, but could halve the effort during Tracklet Construction, and even improve the fit as more information would be available.

10.5 Code Merge

During this work the code for the most recent version in AliRoot SVN and the GPU test version produced by Sergey Gorbunov earlier were merged. Especially since vectorization also should be added to the GPU tracker, a code merge of the CPU, GPU, and vectorized tracker seems reasonable. Bugs in the vector routines of the GCC compiler are a problem, which causes incorrect code before GCC version 4.1 (see [Kre]). On the other hand AliRoot is a framework widely distributed, therefore there is no way to integrate code which does not compile correctly on earlier compiler versions.

[1]Up to now, bit flips in memory were assumed tolerable for displaying graphics content.

10.6 Vector Classes

As noted in the GPU hardware chapter, and as can be seen in Appendix C, the GPU resembles a vector processor under some aspects. Vector classes were introduced in [Kre] as a general abstraction layer with SSE and Larrabee implementations currently available as well as a scalar fall back. When doing a code merge of the vectorized tracker and the GPU tracker, one should consider whether it is possible to write the GPU code as explicit vector-code. Two different vector types must at least be introduced then: the usual global vector for vectors in global memory, and a local vector which coincides with the global vector for real vector processors. On the GPU, however, the local vector would be implemented using scalar thread-local types.

10.7 Track Merger

During this whole work the track merger was not considered at all. The reason is that the slice tracker was not optimized well and not even multithreaded when the work was started. In benchmarks from that time, the slice trackers took up to 15 seconds, and thus merger performance was not relevant. Fig. 10.1 shows tracker and merger times.

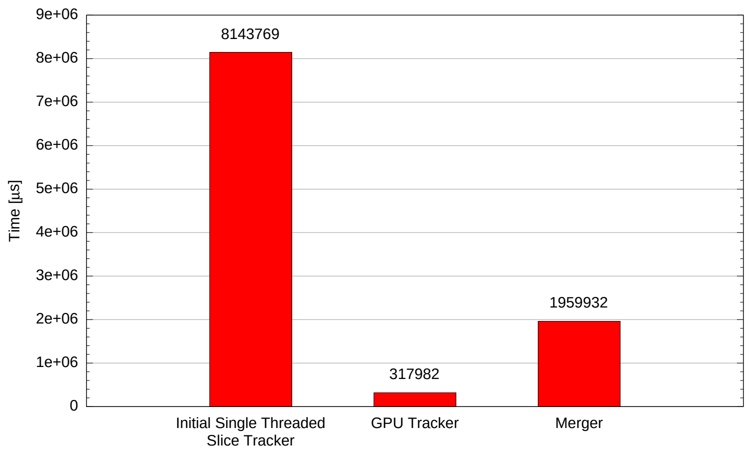


Figure 10.1: Tracker versus Merger Performance

After the tracker performance has been improved significantly, the merger is left as the most time consuming part that definitely is in need of improvement. The question is whether the track merging could also be done on a GPU. Then some other aspects should be considered. It would not make much sense to transfer tracks from GPU memory to the host if the data will be transferred back for the GPU merger afterwards. A scenario is imaginable where all slice trackers run on one GPU and the slice tracks are kept in the GPU memory. Then a GPU merger could directly process the tracks and only the final result would be transferred to the host.

Unfortunately, at least for now, this cannot be realized, as this would concentrate too much data from the HLT farm on one host. As the cluster was recently upgraded with QDR Infiniband, benchmarks will first have to show the practically available network bandwidth.

Appendix A

CUDA Assembler

This appendix provides a short introduction into the NVIDIA PTX code. PTX code is not the final assembler code executed by the device but intermediate code which is generated during the compilation process. More information about the PTX code can be found in [Nvi]. Unfortunately, there is no real reference for the final assembler code.

Assembler PTX code instructions look like:
(instruction).[memory specifier].[vector specifier].[data type 1].[data type 2]
(operand 1),(operand 2),[operant 3],[operand 4]
Square brackets are not obligatory.

Here **instruction** stands for the instruction itself like **ld** and **st** for memory loads and stores, **cvt** for type conversions or **add** and **mul** for arithmetic operations. The supported options in squared brackets above differ from instruction to instruction.

The **memory specifier** is available for memory operations in order to determine the memory type. Supported values are **global**, **local**, **shared**, and **const** for the corresponding memory types.

Memory load operations can be altered by a **vector specifier**. In such a case, up to 4 values of the corresponding **data type** can be read into 4 distinct registers. The destination operand must then be a list of registers.

Most instructions involve at least one **data type**. For arithmetic operations there is a common **data type** for the result and all source data values involved. For memory operations it defines the size of the memory access. Type conversions naturally involve two **data type** specifiers. Valid **data types** are **u16**, **s16**, **u32**, **s32**, **u64** and **s64** for signed and unsigned integer types of different bitwidth and **f32** and **f64** for single and double precision floats. Pointers are represented by the **u64** type.

The result of an instruction is always stored in operand 1, which is the destination operand. Arithmetic instructions can have up to three source operands in the case of a multiply add. The operands can be constants, registers, and memory addresses. For a memory access the operand defines the memory address and must be enclosed in square brackets.

This appendix refers to PTX code only, in which case register assignment was not carried out before. All registers are virtual and might be merged in a later optimization step. They can also be swapped to local memory if the register file is insufficient. Registers are distinguished by the type (**r** for 32-bit integer, **rd** for 64-bit register, **f** for 32-bit and **fd** for 64-bit floats), and by an increasing number that is appended to the register type. Registers are prefixed by a %.

Some examples follow:

Instruction	Explanation				
ld.shared.f32	%f1	,[10];			Load a 32-bit float from shared memory address 10 into floating point register 1
st.local.s32	[%r1]	,%r2;			Store the 32-bit signed integer value in %r2 to the address pointed to by %r1 in local memory
ld.global.v2.u16	{%r1, %r2}	, [%rd1];			Load two 16-bit integers from global memory, pointed to by the register %rd1 to the registers %r1 and %r2
cvt.u32.f32	%r1	,%f1;			Convert the 32-bit float in register %f1 to a 32-bit integer and store to register %r1
madd.f32	%f1	,%f2	,%f3	,%f4;	Multiply add example with four operands

Table 1.1: Assembler Instruction Examples

Appendix B

Atomic Operations

When creating a programm in which multiple threads work in parallel, it is often difficult to regulate access of the worker threads to a shared resource. An atomic operation is a special instruction for such situations. For example two threads shall increment a variable by one, so finally the result should be an increment of two. From the software side the C++ code might look like:
var = var + 1;
However, this neither defines what assembler instructions are created from this by the compiler nor how they are processed by the hardware. An abstract realization could be:

- Load the value of the variable from memory into a register.
- Increment the register by one.
- Store the updated content in the register back to memory.

For concurrent threads this leads to a problem. In the case where both parallel threads have the variable already read into the register but not stored back, they will both increment the initial value by one and store it back, resulting in a total increase of just one, not two.

This can be solved by introducing atomic operations, which are by definition instructions executed by the hardware in one single step, without the possibility that another thread accesses the resource at the same time. By construction, atomic instructions require special hardware support, which is available for NVIDIA CPU as of compute capability 1.2. Generally an atomic instruction works on a memory address, and its return value is usually the data located at that address before the instruction was executed.

As a more complex example, it will be illustrated how the Tracklet Selector determines the final track ID. Every thread processes a well defined set of tracklets and forms a number of tracks. The tracks are stored in memory as an array. Enough memory to store all the tracks is allocated earlier. The challenge is to assign non overlapping memory segments to each thread to store its tracks.

A counter is initialized by zero. Every thread issues an atomic add instruction adding the number of tracks it would like to store to the counter. Given the case thread i created n tracks. The atomic add instructions issued by thread i returns with the value b (thus after the instruction the counter value is $b + n$). Thread i can now store its tracklets to the array elements b to $b+n-1$. As the addition was atomic, no other thread will attempt to store its tracklet to that position in the array. When all threads are done, the counter contains the number of tracks actually stored.

Appendix C

Gathers / Scatters

For scalar code, memory access is not very complex. In fact, accessing the memory means storing or reading one value from a register to memory and vice versa respectively. This gets more complex for vector computers with vector registers. Only the reading case will be dealt with, which will lead to a gather operation. The scatter operation in the write case is entirely analogous.

Let the vector width be n. In the simplest case the data read to the vector register is stored in memory consecutively. Loading data from address p to the register, will read the values at addresses p, $p + t$, $p + 2 \cdot t$, ..., $p + n \cdot t$, with t the size of the data type. In this case only a single start address p is used. For efficient implementation most processors require the address p to be aligned to $n \cdot t$ bytes.

This gets considerably more complex, when the addresses are not consecutive. In that case the address is not stored in a scalar register but many addresses are stored in another vector register itself. In an even more complex case it might not be desired to update the entire vector with data from memory but only a part of it. As a solution a mask register comes into play. A masked vector gather operation will update the value in the i^{th} component of the register r if the i^{th} component of the mask register m is set to 1 with the value at the address pointed to by the i^{th} component of the address register a. Fig. 3.1 shows an example.

The NVIDIA GPU is no real vector processor but it is very similar. The corresponding access to the vector gather above is a conditional scalar memory fetch from different memory locations for all threads in one warp. The coalescing rules describe almost exactly the case, which was found out to be the simplest possible gather, where with consecutive addresses. In that way writing CUDA code can be seen as implicit vectorization.

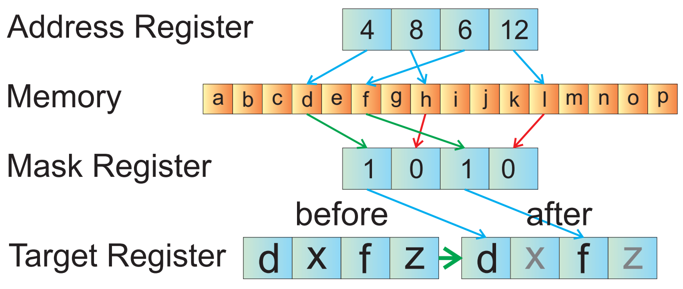


Figure 3.1: Example of Masked Vector Gather

Acknowledgements

I would like to thank all who contributed directly or indirectly to this thesis.

At first Prof. Dr. Volker Lindenstruth, who offered me many alternatives from which I chose the current topic and who always placed confidence in me. He gave me all the freedom I could have wanted and always had time for discussions and to answer my questions.

I also want to thank Prof. Dr. Thomas Ludwig, who offered to supervise the work as second reader.

Special thanks to Sergey Gorbunov, who already built a great basis that I could start from. He was always full of ideas and suggestions regarding a GPU tracker, and also introduced me to the tracking business, especially the theoretical parts, the Kalman filter, and the track fit.

I could learn a lot from Matthias Kretz, who was working on a similar subject and so we had lots of productive discussions on new algorithms. He helped me a lot by introducing me to all the people and computers. I could benefit a lot from his Linux knowledge and I especially thank him and his wife Flora for proofreading this document.

Furthermore, I thank Ivan Kisel, who was not directly working on the ALICE tracker but nevertheless helped me to find into the world of tracking.

I want to thank Katharina Hübner and Hans Hettmansperger for refreshing my knowledge of the Standard Model.

Matthias Bach already collected much experience with the NVIDIA cards. He was always there to help me, when I got stuck in my implementation and he also proofread this document.

Additionally I want to thank Dr. Timm Steinbeck, Jochen Thäder, Kenneth Aamodt, Oystein Haaland, Jan Buchholz and Timo Breitner who were a great help for integrating the tracker into the framework and doing the first tests at CERN.

Special thanks I owe to Collette Van Kerckvoorde, Andreas Ertelt, Oliver Thomas, Hamed Ali, Daniel Pfefferkorn, and my parents Angelika and Günther who invested a lot of time proofreading my thesis.

I want to thank Jörg and Nadine Körner, for helping me with many plots and with the 3D models.

In the end I would like to thank all my other friends who were always supporting me in my work.

Bibliography

[Ali1] ALICE Collaboration, "Technical proposal for A Large Ion Collider Experiment at the CERN LHC", Technical report, CERN, December 1995.
`http://cdsweb.cern.ch/record/293391/files/cer-000214817.pdf`.

[Ali2] ALICE Collaboration, "ALICE Home Page",
`http://aliceinfo.cern.ch/`.

[Ali3] ALICE Collaboration, "ALICE Time Projection Chamber. The homepage of the ALICE TPC"
`http://aliceinfo.cern.ch/TPC/index.html`.

[Ali4] ALICE Collaboration, "Alice Experiment Offline Project",
`http://aliceinfo.cern.ch/Offline/AliRoot/Manual.html`.

[Cbm] CBM Collaboration, "The CBM Experiment Introduction",
`http://www.gsi.de/fair/experiments/CBM/1intro.html`.

[Cer1] O. S. Brüning, P. Collier, P. Lebrun, S. Myers, R. Ostojic, J. Poole, P. Proudlock, "LHC Design Report", CERN, Geneva, 2004,
`http://lhc.web.cern.ch/LHC/LHC-DesignReport.html`.

[Cer2] Carlo Wyss, "LEP Design Report", CERN, Geneva, LEP2 Team, 1996.

[Cer3] CERN, "Overall view of LHC experiments",
`http://cdsweb.cern.ch/record/841555`.

[Fru$^+$] R. Früuhwirth et al., "Data analysis techniques for high-energy physics. Second edition", Cambridge Univ. Press (2000).

[Gor1] S. Gorbunov, "On-line reconstruction algorithms for the CBM and ALICE experiments", Dissertation Thesis, Frankfurth Institute for Advanced Studies, in preparation.

[Gor$^+$2] S. Gorbunov, Matthias Kretz, David Rohr "Fast Cellular Automaton tracker for the ALICE High Level Trigger", GSI Scientific Report 2009.

[Gor$^+$3] S. Gorbunov, U. Kebschull, I. Kisel, V. Lindenstruth, and W.F.J. Müller, "Fast SIMDized Kalman Filter based track Fit", Computer Physics Communications, 178:374 - 383, 2008.

[Hlt] Alice HLT, "Quarks and Bytes", submitted to Nature.

[Int1] Intel Corporation, "Intel C++ Compiler Professional Edition 11.1 for Linux - In-Depth",
`http://software.intel.com/sites/products/collateral/hpc/compilers/clin\_indepth.pdf`.

[Int2] Intel Corporation, “Intel Threading Building Blocks Reference Manual , July 2009“, http://www.threadingbuildingblocks.org/uploads/81/91/LatestOpenSourceDocumentation/Reference.pdf.

[Int3] Intel Corporation, “Larrabee: A Many-Core x86 Architecture for Visual Computing”, URL: http://software.intel.com/file/18198/.

[Kal] R.E. Kalman, “A new approach to linear Filtering and prediction problems”, Trans. ASME-Journal of Basic Engineering, 82 (Series D) (1960) 35-45.

[Kre] M. Kretz, “Efficient Use of Multi- and Many-Core Systems With Vectorization and Multithreading”, Diploma Thesis, University of Heidelberg.

[Lar] D.T.Larsen, “ALICE TPC control and read-out system”, TWEPP-09: Topical Workshop on Electronics for Particle Physics, Paris, France, 21 - 25 Sep 2009, pp.586-588.

[Man] R. Mankel, “Pattern recognition and event reconstruction in particle physics experiments”, Rep. Prog. Phys. 67 (2004) 553-622.

[Nvi] NVIDIA, “NVIDIA CUDA Reference Manual”, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/CUDA_Reference_Manual_2.3.pdf.

[Pen] F. Plentinger, “Discrete Flavor Symmetries”, Master Thesis, Technische Universität München.

[Pes+] Peskin, Schroeder, “An Introduction to Quantum Field Theory ”, 1997.

[Ret] F. Rettig, “Entwicklung der optischen Auslesekette für den ALICE-Übergangsstrahlungsdetektor am LHC (CERN)”, Diploma Thesis, University of Heidelberg.

[Roh$^+$] David Rohr , S. Gorbunov, Matthias Kretz“, Alice TPC Online Tracking on GPU”, GSI Scientific Report 2009.

[Roo] “ROOT - Architectural Overview”, http://root.cern.ch/drupal/content/architectural-overview.

[She] S. Sherman, “Non-Mean-Square Error Criteria“, Trans. IRE Prof. Group on Information Theory, IT-4, 1958.

[Sre] Mark Srednicki, “Quantum Field Theory”, 2009.

[Ste] T. Steinbeck, “A Modular and Fault-Tolerant Data Transport Framework”, Dissertation Thesis, University of Heidelberg, 2004.

Erklärung:

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 16.3.2010 ....