Technische Universität München
Department Physik

# IceCube: neutrinos on graphics cards
# IceCube: Neutrinos auf Graphikkarten

Themensteller:

Prof. Dr. Elisa Resconi

Zweitkorrektor:

Prof. Dr. Peter Fierlinger
Department Physik
Technische Universität München
Arcisstraße 11, 80333 München

Bearbeitet von:

Kevin Abraham
Matrikelnummer: 03627986 , Fachsemester: 6

Abgabetermin: 28 August 2014

## Contents



## List of Figures

# 1. Introduction

## 1.1. IceCube

IceCube is a neutrino detector located at the geographic South Pole (Aguilar 2013) (The IceCube Collaboration 2008). IceCube instruments 1 $km^3$ of ice in a depth from 1450 to 2450 meters with 5160 digital optical modules (DOMs), deployed on 86 strings. Among other particles, IceCube detects atmospheric muons, neutrinos and their antiparticles. These are produced in the decay of pions, which are produced by collisions of cosmic rays with gas nuclei in the upper atmosphere. As atmospheric muons cannot pass through the earth, these are only registered from the southern hemisphere. Additionally, IceCube registers when neutrinos interact with nuclei in the ice, producing either - in the case of electron neutrinos - an electron shower, or, for muon neutrinos, a muon, which is registered as a long track, often passing through the entire detector.
The original IceCube can detect muons with an energy above 100 GeV, the DeepCore extension, with six additional strings, was built to lower the threshold to 10 GeV events (Geisler 2010, p38). Figure 1 shows a schematic view of IceCube, DeepCore and IceTop.
Additionally, on the surface there is a 1 $km^2$ large air shower detector, IceTop, consisting of a pair of tanks placed about 25 meters from each hole cable with two IceCube DOMs each (The IceCube Collaboration 2008).
The IceCube DOMs use photo-multiplier tubes to detect photons from Cerenkov radiation. Cerenkov radiation is emitted when a charged particle travels through a medium at a speed faster than the speed of light in that medium. Charged particles polarize the atoms in the medium. If the particle were traveling slower than the speed of light in the medium, radiation from these atoms returning to their equilibrium state would interfere destructively. However, if the particle is traveling faster than the speed of light in the medium, the atoms can no longer interfere. The radiation is emitted at an angle specific to the speed of the particle and the medium, in the case of ice and a particle traveling at speeds near the speed of light, this is about $41°$ (Geisler 2010, p. 33). The DOMs record timing and number of photons. A series of reconstructions is then applied to get the most likely track and energy. Reconstructions range from fast but inaccurate reconstructions like linefit, and more precise but more time-consuming reconstructions like millipede (Whitehorn 2012). See chapter 3 for more details on reconstruction.

## 1.2. Graphics Cards for General Purpose Computing

Graphics cards are highly specialized devices, originally built to accelerate the computation of 3D computer graphics, mainly for PC games. Due to the very parallel nature of graphical calculations, graphics processing units, called GPUs, were built to handle a large amount of threads. Unlike CPUs, which are built to execute a

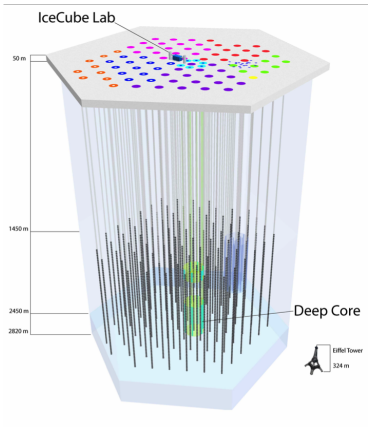


Fig. 1: The IceCube detector with 86 strings and 5160 DOMs in a depth from 1450m to 2450m

small number of threads at a high speed, graphics cards gain their speed by parallelism. The nvidia GK110 GPU can handle 2880 threads in parallel (nvidia 2012). To be able to accommodate this amount of threads, a GPU uses a reduced reduced the amount of control logic per thread. On the nvidia Kepler architecture, threads are grouped into groups of 32, warps, named after the group of strings "threads" used in weaving. All threads in one warp share the same control logic, therefore all threads must execute the same series of commands on different data sets. This is called a SIMD architecture: same instruction, multiple data. This allows the GPU to cut down on control logic and the associated power consumption, at the cost of general-purpose applicability. Therefore, GPUs should not be used in cases where code is highly divergent, meaning that the threads quickly split into non-predictable branches.
Originally, programming graphics cards was very difficult, requiring knowledge of the use of the graphics drivers. As graphics cards became more general-purpose devices, vendors began to market their graphics cards specifically for high performance computing. In 2006, nvidia released the compute unified device architecture (CUDA), allowing direct programming of the GPU and giving methods for communication between CPUs and GPUs (Jason Sanders 2012). Another option for programming GPUs is openCL, an open standard that supports devices from a large variety of vendors, unlike CUDA, which only works on nvidia graphics cards.

In my thesis, I analyze the use of GPUs for physical computations in the IceCube collaboration the case of two different programs, described in the following chapters.

## 2. Small-Scale Anisotropies

### 2.1. The Analysis

The Small-Scale Anisotropies analysis searches for faint sources in the IceCube data that are too weak to be detected in a point source search. The standard point source search needs a strong flux to be sensitive, while the diffuse analysis needs a large amount of sources. The Small-Scale Anisotropy search is in between these two and does not need any prior information about the potential sources (M.G. Aartsen et al. 2014). The analysis cannot identify individual sources, it can only prove or disprove the existence of sources in general. Due to detector uncertainties, two neutrinos originating from the same point in space may be reconstructed at slightly different angles. To overcome this, the analysis evaluates the Two-Point Autocorrelation Test, called 2-pt, at angles comparable to the detector resolution. It works by calculating the angle $\Psi$ between each pair of events, and counting how many pairs are separated by an angle smaller than a specified angle $\theta$. This is done for a range of angles $\theta$, called angle bins, each with individual counters. The result is then compared with what would be expected from an isotropic distribution. In addition, to reduce background and increase sensitivity to high-energy sources, a minimum energy $E_{min}$ is introduced, only counting pairs where both events have a reconstructed energy larger than $E_{min}$.

With $\Psi$ being the spatial distance between two events, the so-called test statistic (TS) for an angle $\theta$ and a minimum energy $E_{min}$, is defined as:

$$TS(\theta, E_{min}) = \frac{\text{obs.no.of pairs with}\Psi \leq \theta, \mathrm{E_{i,j}} \geq \mathrm{E_{min}}}{\text{avg.no.of bg.pairs with}\Psi \leq \theta, \mathrm{E_{i,j}} \geq \mathrm{E_{min}}} \tag{2.1}$$

The observed number of pairs is calculated by the equation

$$N(\theta_{max}) = \sum_{i=0}^{N} \sum_{j=i+1}^{N} \Theta(\theta_{i,j} - \theta_{max}) \tag{2.2}$$

Assuming a $\mathrm{E}^{-2}$ neutrino spectrum, and more than 20 sources in the northern sky, the 2-pt analysis could detect a signal that the point source likelihood search would not (M.G. Aartsen et al. 2014, p.19).

To know how likely the resultant test statistic would have happened by chance, the distribution of the test statistic without any sources must be known. This is calculated by repeatedly replacing the azimuth component of the sources by random numbers, called scrambling, and evaluating equation 2.2 again. Then, a distribution like in fig. 3 is obtained. Only the azimuth component of the events is scrambled, as there is a zenith dependency in IceCube data: while atmospheric neutrinos will reach IceCube almost unobstructed from any point, atmospheric muons can not travel trough the earth, so only the muons from the southern hemisphere will reach the detector. For this reason, a different event selection mechanism is used for the southern hemisphere, and the analysis is carried out separately for the southern and northern hemispheres.

Currently, the analysis groups pairs into 20 angular bins, from an angle of $0.25^\circ$ to $5^\circ$ in $0.25^\circ$ steps. Figure 2 shows a schematic of the pair counting algorithm, illustrated by four different colors representing four angles. In addition, there are four energy bins, the analysis is carried out separately for all the events, the top 10 $\%$, the top 1$\%$ and the top 0.1 $\%$ of the events sorted by energy. All together, there are 80 bins, each containing the respective number of pairs.

To claim a discovery, the test statistic should be outside of the $5\sigma$ range of the background distribu-

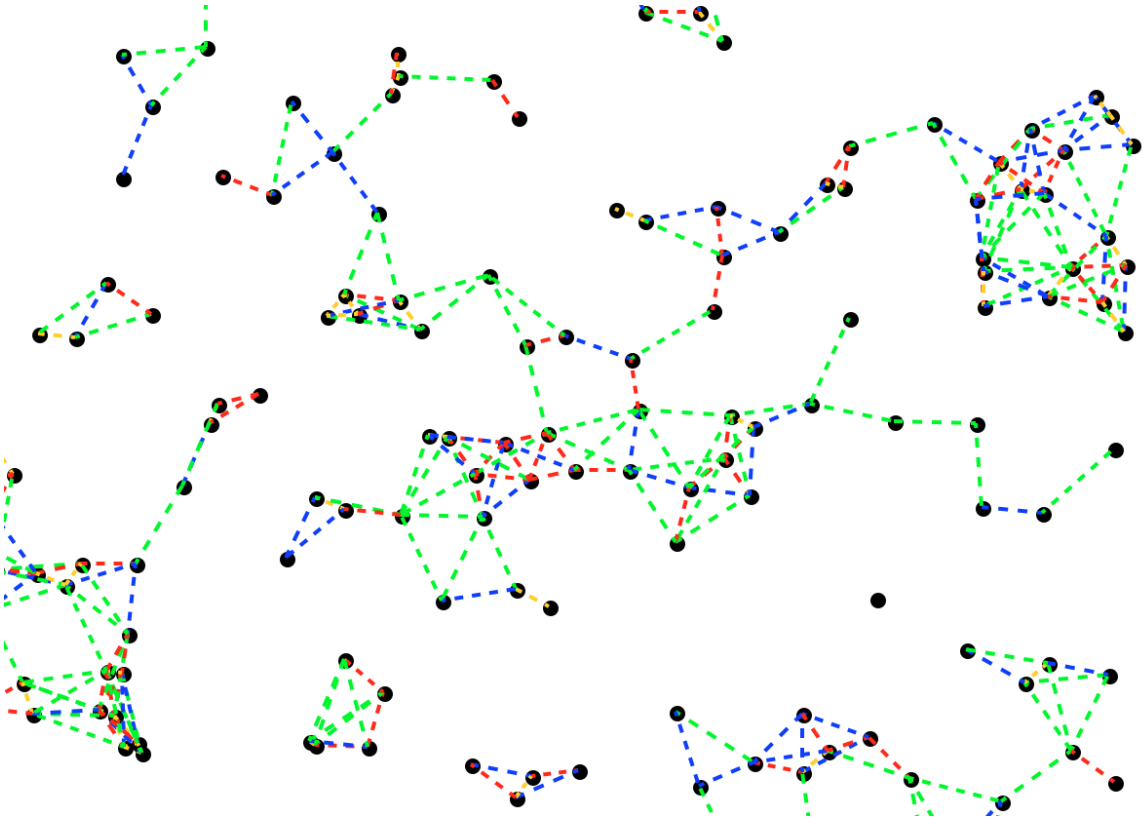

Fig. 2: A schematic representing the pair-counting algorithm on a segment of the sky. The dotted lines represent counted pairs. Here, four bins are represented by - from largest to smallest - the colors green, blue red and orange. Should there be sources, this would result in increased pair count.

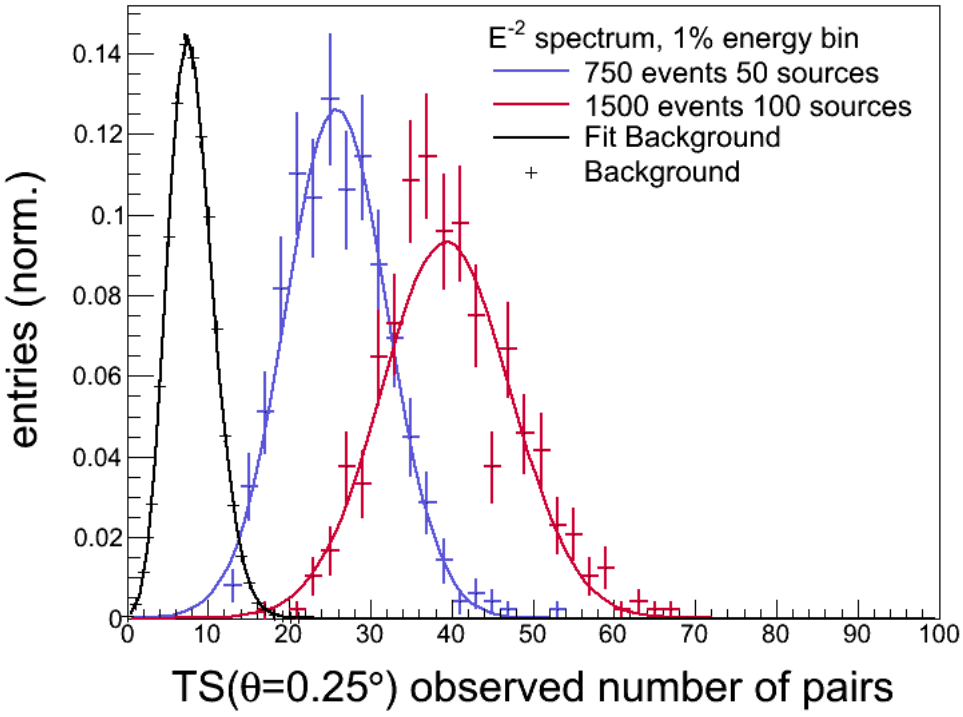


Fig. 3: normalized test statistic distribution assuming isotropic background (black) background plus 750 events from 50 sources (blue), and 1500 events from 100 sources (red)

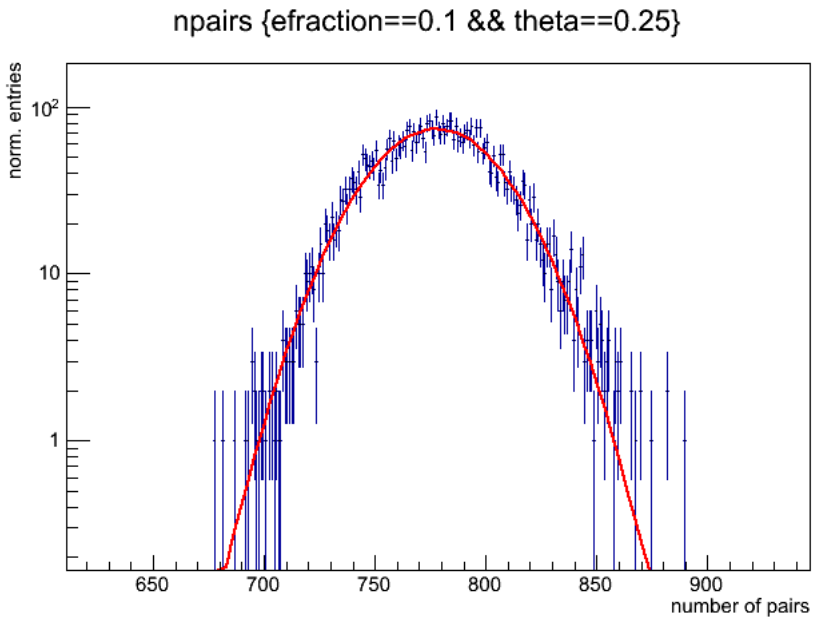


Fig. 4: distribution in the $0.25°$ angle bin and $10\%$ energy bin for random background after 5000 trials, fitted with a gamma function

tion. The likelihood for this to happen by chance is $2.78 \cdot 10^{-7}$. So, to obtain information about this range, it is necessary to repeat this process, called a trial, around $10^7$ times.
However, fitting the data with gamma functions for low energies and Gaussian distributions for high energies, this can be reduced to around 10.000 trials. Still, the more data points are known, the more accurate the fit becomes. Figure 4 shows the test statistic distribution for the 10% energy bin and the $0.25°$ angle bin, fitted with a Gaussian.

The program loops over the double sum, and then loops over all energy and angle bins, increasing the counter of the bin if both events have an energy above the energy bin's cut and are separated by an angle less than the maximum angle of the angle bin. Therefore ,the execution time of this algorithm increases with $\mathcal{O}(n^2)$, n being the number of events. Also, the execution time increases significantly with finer-grained binning.

As a single trial with 111,415 events selected from two years of data takes around 25 minutes on state-of-the art CPU hardware, and several thousand trials are needed, the time this analysis takes to compute becomes a problem. Due to the analysis scaling with $\mathcal{O}(n^2)$, this is especially true when considering running the analysis on a even larger data-set.
A possible solution to this problem is to use GPUs. In the course of this thesis, I ported the analysis to the GPU, and found and implemented further optimizations in both CPU and GPU versions that significantly reduced execution time and also made the execution time less dependent on the number of angle bins.

### 2.2. Moving the Analysis to the GPU

This method is optimal for a GPU, because the individual operations are not dependent of each other and memory access is predictable and linear. Moving to the GPU, I parallelized the outer loop, starting N threads in parallel, each thread picking one event and iterating over the second sum. This approach also has the advantage that all data that is specific to an event, like the space angles of that event, can be kept in local

registers of the respective threads.
Additionally, I implemented methods to copy all necessary data to and from the GPU.

Comparing a single 3.3 GHz Intel Ivy Bridge CPU to a nVidia Geforce GTX Titan, this approach yields a speed advantage of 70x, bringing the execution for one trial time down to 22 seconds from an original 25 minutes.

The resultant pair count is slightly different on the GPU compared to the CPU, this is due to differences in the implementation of trigonometric functions and different rounding. However, this effect is on the order of $10^{-5}$ and statistical.

### 2.3. Further Improvements to the Algorithm

I found a number of improvements to the algorithm, that will work both on the CPU and on the GPU.

#### 2.3.1. Sorting the Events by Zenith

Taking advantage of spacial locality in data structures, it is not necessary to compare every event with every other event. If the events are sorted by zenith, the second loop can be interrupted once the separation of the zenith angle alone between two events is larger than the angle of the largest bin, as the space angle is always at least as large as the relative difference in zenith.

This was already implemented in the code as I got it, however, there was a error that prevented the optimization from working.

This approach reduces the execution time on the CPU to 324 seconds from 1530 seconds, a speedup by a factor of 4.7. On the GPU, the execution time sinks form 22 seconds to 3.4 seconds, a speedup of 6.5x.

#### 2.3.2. Improved Binning

In the original code, the program loops over all bins, checking if the angle that was computed is smaller than the maximum angle of the respective bin every time. Because the bins are equally spaced, the last bin that the event pair is still in can also be obtained by multiplying the computed angle by 4 and rounding up. Once the two loops have finished, the total pair count for each bin can be obtained by adding the sum of all smaller bins.

Using this, one trial is approximately 11% percent faster on the CPU and 17% on the GPU, reducing execution times for one trial from 324s to 292s and 3.2s to 2.9s on the CPU and GPU, respectively.

As an interesting consequence of this, an almost arbitrary amount of angle bins can now be used without notably increasing execution time.

#### 2.3.3. Pre-Computing

A majority of the time spent by the program after these optimizations is spent calculating the space angle between two events. The the space angle $\Omega$ between two vectors in polar coordinates can be calculated by the equation:

$$\begin{aligned}\Omega = \arccos(&\sin(\theta_1)\cos(\phi_1)\sin(\theta_2)\cos(\phi_2)+\\ &\sin(\theta_1)\sin(\phi_1)\sin(\theta_2)\sin(\phi_2) + \cos(\theta_1)\cos(\theta_2))\end{aligned} \tag{2.3}$$

this equation is evaluated $\mathcal{O}(n^2)$ times. Trigonometric functions take a significant amount of time to compute. The recurring terms $\sin(\theta)\cos(\phi)$ , $\sin(\theta)\sin(\phi)$ and $\cos(\theta)$ do not depend on the position of the other particle, therefore, they only have to be calculated once for every event, instead of once for every pair. I implemented

an algorithm to pre-calculate these values, reducing the complexity of these calculations to $\mathcal{O}(n)$ and only leaving three multiplications, two additions and the inverse cosine to be done $\mathcal{O}(n^2)$ times. The three values for each event are computed and stored in arrays before the main loop begins.

This increases the execution speed by a factor of 6.5 on the CPU and 4.1 on the GPU, reducing execution times from 292s to 45s on the CPU, and 2.9s to 0.7s on the GPU.

This can also be ported to other applications, I have also seen my implementation being used in the MPS-HESE analysis, another analysis relying heavily on space angle calculation.

### 2.4. Conclusion

The execution time for one trial in the test-case was reduced from 1530 seconds to 45 seconds on the CPU, and to 0.7 seconds using one of the graphics cards. Using a single nvidia GTX Titan card, it is now possible to calculate 120.000 trials per day, opening the possibility of analyzing the $5\sigma$ range and beyond without using fit functions. Also, the analysis can now be carried out on even larger data-sets without the time being spent by computation becoming a limitation.

| Optimization/Execution time [s] | CPU | GPU |
|---|---|---|
| Original | 1530 | 22 |
| Zenith sorting | 324 | 3.4 |
| Improved Binning | 292 | 2.9 |
| Pre-computing | 45 | 0.7 |

Execution time in seconds for calculating one trial for the test statistic with 111415 events. CPU: Intel Ivy bridge, 3,3 GHz, GPU: nvidia GK110, 876 MHz (GTX Titan)

## 3. Millipede

Figure 5 shows an example of an event registered by IceCube. Red colors represent DOMs that were hit early in the time-frame, blue colors represent DOMs that were hit late in the time-frame. To reconstruct these events, IceCube uses a wide variety of algorithms. See (Whitehorn 2012) and (Geisler 2010) for a more detailed discussion on reconstruction methods. In short, there is the linefit method, which uses the least-squares algorithm for a first approximation of the track. As this is a analytic method, it is very fast. Then there is the pandel MPE fit, which optimizes a likelihood function based on the arrival time of the first photon and assumes uniform ice properties. Optimization is done in the coordinates of the origin and the angles of the muon track. Then there is millipede, which is used to reconstruct energy loss of high-energy muons, and will be discussed in more detail below.

In addition to the continuous Cerenkov radiation, high energy muons have a stochastic energy loss along their track due to statistical processes such as bremsstrahlung, photonuclear interactions and pair production. Millipede is a method to estimate how much energy a muon registered by IceCube lost along it's track. Millipede uses the DOMs responses to calculate the most likely energy deposition for a postulated muon track. This is done in discrete segments, usually 10-15 meters long. The output from millipede can be used to achieve a more accurate likelihood value for the initial postulated track. The most likely track can be estimated with a minimizing algorithm, or one can create a so-called full sky map, calculating the likelihood for a large amount of points in a mesh. Tests show millipede should result in up to 10 times better angular resolution than Pandel-function based approaches at high energies. (Whitehorn 2012, p.39) However, millipede takes significant computing time, especially when many repetitions are necessary, like in a full sky map. Using a GPU could allow more neutrino candidates to be reconstructed with millipede, and to use finer grained settings where millipede is already used.

A first functioning implementation of a part of millipede has already been written by IceCube PhD Jakob van Santen in openCL (van Santen, private communication). However, van Santen reported that his implementation was only about 10 times as fast as a single CPU thread in his test-case, not enough to justify the use of GPUs given the much higher price of a graphics card compared to a single CPU core.
In the course of my thesis, I studied ways to improve the GPU algorithm. The work is still on-going, however I will give a first report on the current status here.

### 3.1. How the Reconstruction works

The signal measured by one IceCube DOM is the result of light emitted along the entire length of the track. If we cut the track into discrete segments, the energy measured by a DOM can be seen as the sum over the product of the energies deposited in each segment with the likelihood of a photon from this segment arriving at the DOM. Additionally, all DOMs are subject to a random noise term $\nu$.
The signal $k_{jt}$ measured by a DOM $j$ at time $t$ can be written as:

$$k_{jt} = \sum_i E_i P_{ijt} + \nu \tag{3.1}$$

$E_i$ is the energy deposited by the muon in the segment i, consisting of the sum of the energy loss due to Cerenkov radiation $E_{Cerenkov}$ and the energy loss due to stochastic effects $E_{stoch}$. DOMs with many registered photons are split into multiple time bins. $P_{ijt}$ can be seen as an element of a Matrix $\Lambda$, containing the probability for photons from each segment being registered by each DOM. Also, $k_j$, $E$ and $\nu$ can be expanded to vectors containing all DOM signals, all energy depositions and all noise levels. The equation can then be written as:

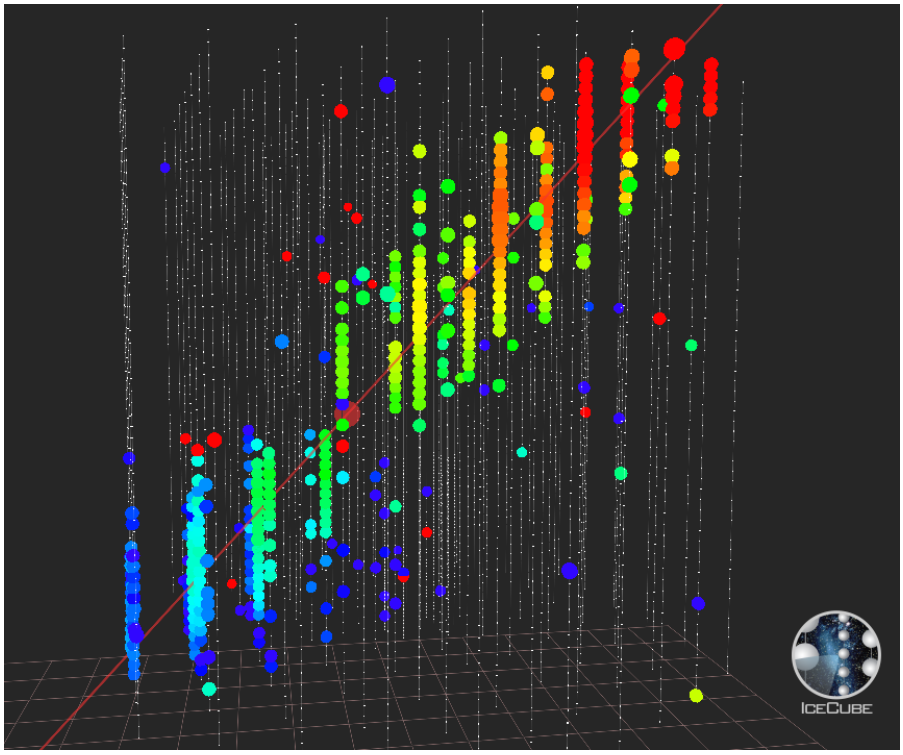


Fig. 5: An example of an event registered by IceCube, together with the track from the fastest reconstruction, linefit. Each point represents a DOM. Red points are among the first DOMs to register photons, blue colored DOMs registered photons towards the end of the time-frame. This specific event was triggered on 5th May 2011, 9:59:45 and was reconstructed by splineMPE to have an energy of 150 TeV

$$\vec{k} - \vec{\nu} = \Lambda \cdot \vec{E} \tag{3.2}$$

The coefficients of $\Lambda$ are calculated from a large set of pre-run simulations. The results of these simulations is tabulated, using splines to span a multi-dimensional curve.

Inverting the matrix $\Lambda$, one can calculate $\vec{E}$. This information can then be used to achieve a more precise likelihood for the initial track.

A significant amount of time is spent calculating $\Lambda$, the so-called response matrix. The GPU version parallelizes the calculation of these coefficients. Effectively, in this implementation the CPU does all necessary calculations until it is known where the segments of the track are in the coordinate system, and then passes this information on to the GPU, which builds the response matrix from the spline data.

The bulk ice on the south pole does not have uniform optical properties. The mean absorption and scattering length of a photon varies greatly by depth, owing to layers of volcanic dust throughout the ice. Tables containing the optical properties thus must consider the depth of the light source, and also the angle of emission. The ash layers are slightly tilted in the ice, but the optical properties until recently were not assumed to vary largely along the XY plane. Thus, the ice is assumed to be cylindrically symmetric around the z-axis to reduce the size of the tables.

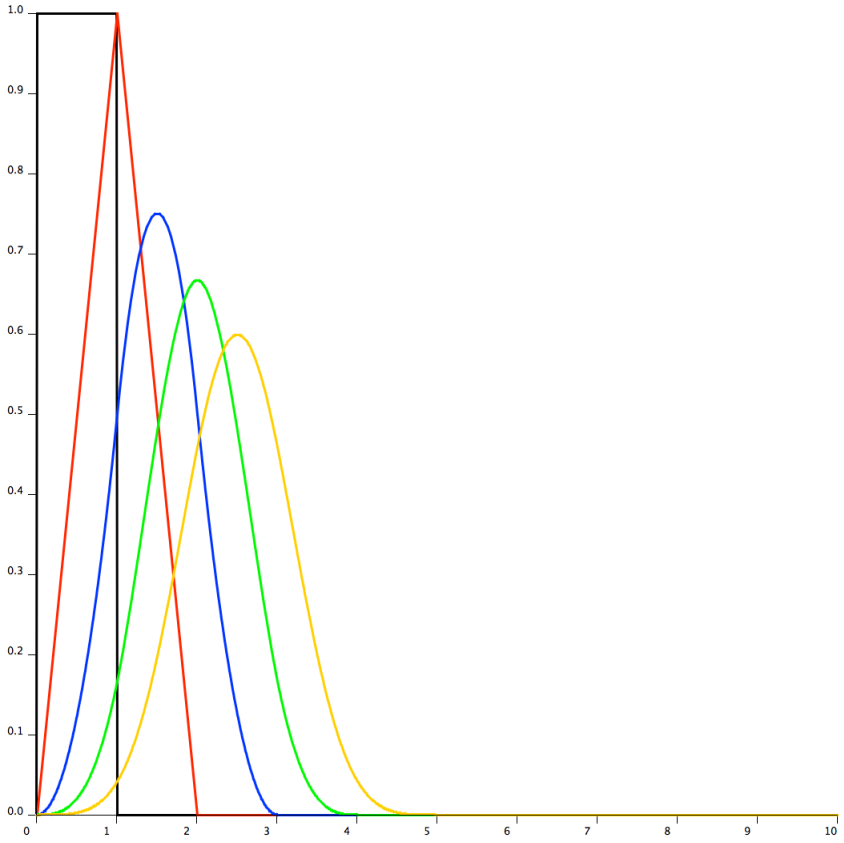


Fig. 6: B-Splines of order 1 (black) , 2(red) , 3 (blue) , 4 (green) , 5 (orange)

There are two types of tables: The five dimensional probability tables, which depend on the position of the observer in three space coordinates, the depth of emission, and the angle of emission; and the six dimensional timing tables, which also depend on the time between emission and detection.

### 3.2. B-Splines

The likelihood for a photon emitted from a section of the track to arrive at a DOM is calculated by simulation. In the original version, this simulated data is tabulated in a discrete way. In between two data points, linear interpolation is used to avoid sudden jumps in the data. However, this is problematic because the minimizing algorithm requires the derivative to be continuous. To overcome this issue, second and third order B-Splines are now used.

B-Splines are recursively self-convoluted functions, defined on a array of knots $k_0 ... k_n$. A spline of order n can be continuously derived (n-1) times. The $n^{th}$ order B-Spline at x is defined as : (Whitehorn 2012)

$$B_{k,1}(x) = \begin{cases} 1 & k_i \leq x < t_{k+1} \\ 0 & otherwise \end{cases} \tag{3.3}$$

$$B_{k,n}(x) = \frac{x - t_k}{t_{k+n-1} - k_i} B_{k,n-1}(x) + \frac{k_{i+n} - x}{k_{i+n} - k_{i+1}} B_{k+1,n-1}(x) \tag{3.4}$$

For a high order n, B-Splines converge towards Gaussian distributions. The splines can be used to approximate a function, by multiplying the splines by a vector $\vec{\alpha}$

$$f(x) \approx \sum_i \alpha_i B_{i,n}(x) \tag{3.5}$$

For multi-dimensional splines, the array of knots is expanded to a n-dimensional knot grid. The splines are then expanded to tensor product surfaces, by taking the tensor product of n one-dimensional splines. A n-dimensional function can then be approximated by multiplying every basis function by coefficients from an n-dimensional grid $\bar{\alpha}$. These coefficients are calculated to create the spline tables. At any given point $x$ in a curve approximated by splines of order n, the splines $B_i$ to $B_{i-n}$ are non-zero, with i being the index of the last knot $k_i$ that is smaller than $x$. All other splines are zero. For an n-dimensional spline, the amount of non-zero tensor product surfaces is the product of the amount of non-zero splines in each dimension. The values for the B-Splines can be calculated separately for each dimension, subsequently calculating the tensor product. The splines used by millipede are order two splines in every dimension except for time in the timing tables, which is order three. For the timing tables, this means that a six dimensional box with edge lengths of $3^5\mathrm{x}4$, in total 972 values, have to be calculated.

### 3.3. Challenges

As shown in the last chapter, to evaluate one point in the probability tables, one has to evaluate 972 spline values on a six dimensional box with edge lengths of $3^5\mathrm{x}4$. The tensor product itself can be built relatively quickly, as the splines can be calculated separately for every dimension. However, each one of the 972 elements has to be multiplied by the respective coefficient from a 6-Dimensional coefficient array. At this point, we reach a limitation of many GPU architectures: a high reading speed from memory can only be achieved when the values are in order in the memory. However, it is not possible to store six-dimensional data in such a way that the coefficients from every possible rectangle are in order.
This calculation has to be done for every element of the response matrix, a matrix of the size (# of DOMs and time bins)x(#of track segments). With 5160 DOMs and in the order of 100 segments, this limitation becomes significant. Van santen claims this one of the major reasons why his GPU implementation does not perform as expected.

Just like a GPU sacrifices control logic for compute logic, the GPU's memory controller design also incorporates some fundamental tradeoffs between speed and general purpose usability. The architecture assumes that memory will usually be read in a linear fashion. Therefore, a memory controller will not simply load a single value from memory, but always loads a larger chunk.

GPUs, like many CPUs have two levels of on-chip cache, a small fast memory storing data that is likely to be used soon. There is a small L1 cache, with sizes in the order of some hundred kilobites, and a larger but slower L2 cache, with sizes ranging in the order of megabytes.
According to the nvidia cuda developers guide, a GPU with compute capability 2.0 or higher will always fill it's L2 cache with at least 32 consecutive bytes at a time. The even faster L1 cache will be filled in chunks no smaller than 128 bytes at a time. (nvidia 2014, G 4.1.2) This way, the GPU can load more data at a higher speed with less overhead from managing the memory transactions.

If the memory is not accessed in a linear fashion, this will result in a lower effective memory bandwidth, corresponding to the fraction of the pre-fetched line of memory that is actually used. This can significantly slow down a GPU program, as the random memory reading speed of a GPU is not much higher than a CPU's. Therefore, it is important to keep memory access patterns in mind when designing GPU programs. This design goal is called memory coalescence.

As the coefficients of the spline tables are stored in a 6-dimensional array, the maximum amount of consecutive reads is 4, the amount of coefficients in the lowermost dimension of the array. As the coefficients are 32-bit floating point values, this means we have 128 bits, or 16 bytes of consecutive data. This means that, if the data is loaded into L2 cache, only half of the theoretical bandwidth is used. If the L1 is used, one

can only obtain $\frac{1}{8}$ of the maximum bandwidth.

### 3.4. Redundant Memory

To overcome this limitation, I studied the use of redundant copies to increase spacial locality in the memory. The spline tables are only about 500 MB in size, and the graphics cards I use have 6 GB of memory. This led me to come up with a new approach, redundantly permutating the data in such a way to guarantee 12 consecutive coefficients in memory using timing tables, and 9 consecutive reads in probability tables. This comes at the cost of 3-4 times more memory use compared to using non-redundant tables.

If we consider only the lowest two dimensions of the array, the six dimensional box turns into a rectangle. If we consider the timing tables, as time has order 3 splines and in all other dimensions have order 2 splines, and time is the lowermost dimension, this yields a rectangle with side lengths 4x3. Usually, this would be stored in memory like this:

| | | | | | | |
|---|---|---|---|---|---|---|
| (1,1) | (1,2) | (1,3) | (1,4) | (1,5) | ... | (1,n) |
| (2,1) | (2,2) | (2,3) | (2,4) | (2,5) | ... | (2,n) |
| (3,1) | (3,2) | (3,3) | (3,4) | (3,5) | ... | (3,n) |
| (4,1) | (4,2) | (4,3) | (4,4) | (4,5) | ... | (4,n) |
| ... | ... | ... | ... | ... | ... | ... |
| (m,1) | (m,2) | (m,3) | (m,4) | (m,5) | ... | (m,n) |

(i,j) represents the coefficient with coordinates i and j in the 2nd and 1st dimension, respectively.

The red entries are those that would have to be read to calculate the spline value for a point that has 1,1 as the lowest point with non-zero splines. This means, that in a 2 dimensional projection, the point to be evaluated lies within the rectangle spanned by (1,1) (1,2) (2,1) and (2,2). The case where the top most point is (1,1) is shown for simplicity, usually the rectangle would lie somewhere in the middle. Here it can be seen that every jump to a new row is also a jump in memory, thus never having more than the said 4 coefficients next to each other.

The new permutation works like this: we take the first four coefficients from the first line, then the first four from the second line and so on, until we reach the end of the second dimension. Then we start again, this time taking the next four coefficients, and continue doing this until there are less than four coefficients left. After this sorting, the data is now stored in memory like this:

| | | | |
|---|---|---|---|
| [(1,1)(1,2)(3)(1,4)] | [(2,1)...(2,4)] | [(3,1)...(3,4)] | [(m,1)...(m,4)] |
| $[(1,5)(1,6)(1,7)(1,8)]$ | $[(2,5)...(2,8)]$ | $[(3,5)...(3,8)]$ | $[(m,5)...(m,8)]$ |
| ... | ... | ... | ... |
| $[(1,n-3)...(1,n)]$ | $[(2,n-3)...(2,n)]$ | $[(3,n-3)...(3,n)]$ | $[(m,n-3)...(m,n)]$ |

Higher dimensions are not affected by this process, and can simply be iterated over in the classical fashion.

As can be seen, all 12 coefficients of the box are now in order in memory. Of course, this only works when lowest dimension of the top left coefficient is a multiple of four. All other cases can however be accommodated by introducing a shift: We once again take four elements, but this time we start from the second element in the first dimension:

| [(1,2)(1,3)(1,4)(1,5)] | [(2,2)...(2,5)] | [(3,2)...(3,5)] | [(m,2)...(m,5)] |
|---|---|---|---|
| $[(1,6)(1,7)(1,8)(1,9)]$ | $[(2,6)...(2,9)]$ | $[(3,6)...(3,9)]$ | $[(m,6)...(m,9)]$ |
| ... | ... | ... | ... |
| $[(1,n-3)...(1,n)]$ | $[(2,n-3)...(2,n)]$ | $[(3,n-3)...(3,n)]$ | $[(m,n-3)...(m,n)]$ |

This process is then repeated two more times, with a shift of two and three, and then there are four redundant copies of the same data. When evaluating, the program calculates its first dimension modulus four, and picks the table with that offset. That table will have coefficients aligned for 12 consecutive reads in order.

It should be noted that n is no longer the number of coefficients in the first dimension, but rather the largest number for which it is still possible to select a full four elements. B-Splines are only fully defined when there are at least as many coefficients to the left of the point being evaluated as it's order in that dimension, and at least one to the right. There is a method that prevents requests from being too close to the edges, so the data in the space between n and the real edges can safely be discarded. In theory, this could be used to reduce the total amount of data. Ideally, the shortest dimension should be permuted to the last place, as here the relative fraction of cut-away data from the aforementioned process is the greatest, requiring somewhat less memory. However, in it's current state, my implementation adds 'white space' where the cut-away data would have been. This is due to the way the data is accessed by the program, this will be later explained in more detail.

It should also be noted that permuting the dimensions in such a way that time would be second-to-last, and putting one of the other dimensions with order 2 in last place could lead to the same result but only need three redundant copies instead of four.

In the course of the thesis, I implemented methods to do these permutations, copy the data to the GPU, and then evaluate the splines using the new memory structures. The implementation can be shown to return the same results as the original version.

With $\vec{\xi}$ containing the positions of one particle in all dimensions, $\vec{\Xi}$ the length of the data in each dimension, and $\vec{O}$ the order of the splines, the following equation can be used to calculate the position $Idx$ of a coefficient in the redundant array with stride $\chi$:

$$Idx = \xi_2 \cdot O_1 + (\xi_1 - \chi)\%O_1 \cdot O_2 \cdot \Xi_1 + (\xi_1 - \chi)\%O_1 + \sum_{i=2}^{ndim} (\prod_{j=0}^{i} O_j)\xi_i \tag{3.6}$$

The first term takes the position in the second dimension, which is identical to the number of boxes before it in one line, an multiplies it by the order of the first dimension, identical to the length of a box. The second term calculates which line of memory the point is in, and multiplies it by the length of a line. The third therm calculates the position within the box. The final term calculates the position with respect to the rest of the non-permuted dimensions.
While this equation may be complicated, it only has to be calculated once. There is a method in the original code, which I modified, that pre-generates code specifically for one spline table. This takes advantage of the fact that, once a starting position has been chosen, the relative distance to all following coefficients in memory remains the same. Therefore, once the initial position is calculated, the program needs only to iterate over the pre-calculated strides. This is also why I added 'white space' where I could have reduced the size of the tables. As the amount of data that is cut away differs between the permuted copies, this would require

a different branch of code for every table. Branching, however, would reduce the execution speed on a GPU.

### 3.5. Outlook

Preliminary results do not show significant improvement, however, I found that only a small fraction of the time is spent in the considered functions in my test-case. Investigation into differences in hardware, software environment and possible sources of overhead are ongoing. In theory, further coalescence could be gained by applying the permutation again, additionally permutating the third dimension, yielding 36 consecutive coefficients for the probability tables at the cost of 9 times the memory use. However, the additional memory in modern graphics cards could also be used for more accurate spline tables. As it often is, a compromise between speed and accuracy will have to be found.

## 4. Conclusion

During the work on my thesis, I found two very different programs from very different parts of the IceCube collaboration. The experiences with porting these programs onto GPUs was equally different. The 2-pt analysis had linear memory access, and could easily be split into multiple independent threads. Also, I could find many optimizations to the algorithm itself. The reconstruction method millipede on the other hand proved to be much more complicated, accessing memory in a highly non-linear fashion. Also, the CPU code was far more optimized to begin with. This does not mean that no improvement is possible, I will continue to work with the collaboration to finish my work on millipede.

**Bibliography**

Aartsen, M. G., Abbasi, R., Ackermann, M., Adams, J., Aguilar, J. A., Ahlers, M., Altmann, D., Arguelles, C., Auffenberg, J., Bai, X. & et al. (2014): Energy reconstruction methods in the IceCube neutrino telescope, Journal of Instrumentation **9**: 3009P.
**URL:** *arXiv: 1311.4767*

Aguilar, J. A. (2013): Neutrino searches with the IceCube telescope, Nuclear Physics B Proceedings Supplements **237**: 250–252.

Geisler, Matthias (2010): On the measurement of atmosperic muon-neutrino oscillations with icecube-deepcore.

Jason Sanders, Edward Kandrot (2012): CUDA by Example, Addison-Wesley.

M.G. Aartsen, M. Ackermann J. Adams et al. (2014): Searches for small-scale anisotropies form neutrino point sources with three years of icecube data.
**URL:** *arXiv:1408.0634*

nvidia (2012): the kepler gk110 whitepaper.
**URL:** "*http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf*"

nvidia (2014): cuda c programming guide.
**URL:** "*http://docs.nvidia.com/cuda/cuda-c-programming-guide/*

The IceCube Collaboration (2008): The IceCube Data Acquisition System: Signal Capture, Digitization, and Timestamping, ArXiv e-prints: 0810.4930 .

Whitehorn, Nathan (2012): A search for high-energy neutrino emission from gamma-ray bursts.

**Ehrenwörtliche Erklärung**

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht.

Garching bei München, 27.8.2014

Kevin Abraham

## A Appendix

In the following pages, the code for the original CPU version, optimized CPU version and GPU version of the Small-Scale Anisotropy test can be found. Also, the code for producing redundant copies of the data is in the appendix.

# Small-Scale Anisotropies, original Code:

```
//return the angle between event and source for signal pdf
inline double getSpaceAngle(double zenith1, double ra1, double zenith2, double ra2) //returns and
accepts angles in radians
{
 double x1=sin(zenith1)*cos(ra1);
 double x2=sin(zenith2)*cos(ra2);
 double y1=sin(zenith1)*sin(ra1);
 double y2=sin(zenith2)*sin(ra2);

 double space_angle=TMath::Abs(TMath::ACos(x1*x2+y1*y2+cos(zenith1)*cos(zenith2)));
 return space_angle;
}

vector <struct AutoResults> AutocorrelationFn(int N, vector<double> energy_cuts)
{
 // create entries in results vector for each desired energy cut
 vector <struct AutoResults> auto_results_energy_vector;
 for(int e=0; e<energy_cuts.size(); e++)
  {
   struct AutoResults my_auto_result;
   my_auto_result.energycut=energy_cuts[e];
   auto_results_energy_vector.push_back(my_auto_result);

  }

 // initialize result structures
 for(int e=0; e<auto_results_energy_vector.size(); e++)
  {
   auto_results_energy_vector[e].thetapairs.clear();
   auto_results_energy_vector[e].thetavalues.clear();
   for(int thk=0; thk<(int)(max_auto_scale/step_auto_scale); thk++)
    {
     auto_results_energy_vector[e].thetavalues.push_back(step_auto_scale+thk*step_auto_scale);
     auto_results_energy_vector[e].thetapairs.push_back(0);
    }
  }

 double distance;
 int localpairs[4][20];
 for(int i = 0;i<20;i++)
          localpairs[i] = 0;

 // calculate distances and count pairs
 for(int i=0; i<N-1; i++) // loop over all declination-sorted events
  {
   //     cout<<"event "<<i<<endl;
   for (int j = i+1; j<N; j++) // loop over all events which have not yet been counted
    {
     if(TMath::Abs(ScrambledEvents[i].declination-ScrambledEvents[j].declination)>=
max_auto_scale/TMath::RadToDeg()) break; // don't waste time by calculating large distances
          if(TMath::Abs(ScrambledEvents[i].ra-ScrambledEvents[j].ra)>= max_auto_scale*TMath::RadToDeg())
continue; // don't waste time by calculating large distances
      distance=getSpaceAngle(ScrambledEvents[i].declination+(pi/2), ScrambledEvents[i].ra ,
ScrambledEvents[j].declination+(pi/2) ,ScrambledEvents[j].ra)*TMath::RadToDeg();
```

```
            for(int e=0; e<auto_results_energy_vector.size(); e++)
          {
            // check if both events pass the energy cut
            if(ScrambledEvents[i].energy>=auto_results_energy_vector[e].energycut &&
ScrambledEvents[j].energy>=auto_results_energy_vector[e].energycut)
             {

               for(int thk=0; thk<auto_results_energy_vector[e].thetavalues.size(); thk++) // loop to check if we
found a close enough pair for each scale
                        {

                                    if(distance<=auto_results_energy_vector[e].thetavalues[thk])
auto_results_energy_vector[e].thetapairs[thk]+=1.0; // got one -> count it!
                        } // end loop theta
              } // end if energy
           } // end loop energy ranges
         //}//end loop declination>decmin
       } // end event loop j

    }//end event loop i



  return auto_results_energy_vector;
}
```

# Small-Scale Anisotropies, Modified CPU Code:

```
inline double getSpaceAngleOpt( double x1 , double y1 , double x2 , double y2 , double cZ1 , double
cZ2) //returns and accepts angles in radians
{
        /*****************************************************************************
        this is the optimized version of getSpaceAngle, it uses pre-calculated values
        x1, x2, y1 and y2 are calculated once and saved in arrays, thus the compute-intensive
        sin and cosine must only be calculated once for every event, instead of once for every pair
        cZ1 and cZ2 are cosine of zenith1 and cosine of zenith2

        *****************************************************************************/

        double space_angle= abs(acos(x1*x2+y1*y2+cZ1*cZ2));
        return space_angle;
}

vector <struct AutoResults> AutocorrelationFn(int N, vector<double> energy_cuts)
{
 // create entries in results vector for each desired energy cut
 vector <struct AutoResults> auto_results_energy_vector;
 for(int e=0; e<energy_cuts.size(); e++)
  {
   struct AutoResults my_auto_result;
   my_auto_result.energycut=energy_cuts[e];
   auto_results_energy_vector.push_back(my_auto_result);

  }

 // initialize result structures
 for(int e=0; e<auto_results_energy_vector.size(); e++)
  {
   auto_results_energy_vector[e].thetapairs.clear();
   auto_results_energy_vector[e].thetavalues.clear();
   for(int thk=0; thk<(int)(max_auto_scale/step_auto_scale); thk++)
   {
        auto_results_energy_vector[e].thetavalues.push_back(step_auto_scale+thk*step_auto_scale);
        auto_results_energy_vector[e].thetapairs.push_back(0);
   }
  }

 double spAx[N]; //space Angle, component x, cached to save execution time;
 double spAy[N]; //-"- component y
 double cZ[N];  //cosine of Zenith


 for(int i = 0;i<N;i++){
        spAx[i] = sin(ScrambledEvents[i].declination+(pi/2))*cos(ScrambledEvents[i].ra);
        spAy[i] = sin(ScrambledEvents[i].declination+(pi/2))*sin(ScrambledEvents[i].ra);
        cZ[i]   = cos(ScrambledEvents[i].declination+(pi/2));
 }
```

```
 double distance;

 // calculate distances and count pairs
 for(int i=0; i<N-1; i++) // loop over all declination-sorted events
 {
        //cout<<"event "<<i<<endl;
        for (int j = i+1; j<N; j++) // loop over all events which have not yet been counted
        {
                if(TMath::Abs(ScrambledEvents[i].declination-
ScrambledEvents[j].declination)*TMath::RadToDeg()>= max_auto_scale) break; // don't waste time by
calculating large distances
                //  if(TMath::Abs(ScrambledEvents[i].ra-ScrambledEvents[j].ra)>=
max_auto_scale*TMath::RadToDeg()) continue; // don't waste time by calculating large distances
                distance=getSpaceAngleOpt(spAx[i], spAy[i] , spAx[j] , spAy[j] , cZ[i] ,
cZ[j])*TMath::RadToDeg();

                if(distance>max_auto_scale)
                        continue;

                for(int e=0; e<auto_results_energy_vector.size(); e++)
                {
                        // check if both events pass the energy cut
                        if(ScrambledEvents[i].energy>=auto_results_energy_vector[e].energycut &&
ScrambledEvents[j].energy>=auto_results_energy_vector[e].energycut)
                        {
        int bin = distance*4;
        if(bin < 20)
                auto_results_energy_vector[e].thetapairs[bin]++;
                        } // end if energy
                } // end loop energy ranges
                //}//end loop declination>decmin
        } // end event loop j

 }//end event loop i
 for(int e = 0;e<auto_results_energy_vector.size();e++){
        for(int i=1;i < auto_results_energy_vector[e].thetavalues.size();i++){
                auto_results_energy_vector[e].thetapairs[i] += auto_results_energy_vector[e].thetapairs[i-1];
        }
 }


 return auto_results_energy_vector;
}
```

# Small-Scale Anisotropies, Modified GPU Code:

```
__device__ inline double GPUgetSpaceAngle( double x1 , double y1 , double x2 , double y2 , double
cZ1 , double cZ2) //returns and accepts angles in radians
{

        ////double s1s2 = sin(zenith1)*sin(zenith2);
        ////double c1c2 = cos(ra1)*cos(ra2);
        ////double sr1r2 = sin(ra1)*sin(ra2);
        //double x1=sin(zenith1)*cos(ra1);
        //double x2=sin(zenith2)*cos(ra2);
        //double y1=sin(zenith1)*sin(ra1);
        //double y2=sin(zenith2)*sin(ra2);
        ////double space_angle = abs(acos(s1s2*c1c2 + s1s2*sr1r2 + cos(zenith1)*cos(zenith2)));
        double space_angle= abs(acos(x1*x2+y1*y2+cZ1*cZ2));
        return space_angle;
}

__global__ inline void calcAngle(const int N, const double* __restrict eventsDec , const double*
__restrict eventsRa , double *spAx , double *spAy , double *cZ){
        int i = threadIdx.x + blockIdx.x*blockDim.x;

        if(i>=N)//only calculate if data set exists (due to cuda block size, some threads may have an i that is
larger than N
                return;

        spAx[i] = sin(eventsDec[i]+(Pi/2))*cos(eventsRa[i]);
        spAy[i] = sin(eventsDec[i]+(Pi/2))*sin(eventsRa[i]);
        cZ[i] = cos(eventsDec[i]+(Pi/2));
}
__global__ void gpuwork( const int N , const double* __restrict eventsDec , const  double* __restrict
eventsRa ,
                const double* __restrict eventsEnergy , const double* __restrict energyCut ,
                const float* __restrict resultsThetaValues ,unsigned int *resultsThetaPairs ,
                const double *spAx , const double  *spAy , const double *cZ){

        int i = threadIdx.x + blockIdx.x*blockDim.x;

        __shared__ int localPairs[NBINS];  //use shared memory to avoid atomic locking
        for(int idx = threadIdx.x;idx<NBINS;idx += blockDim.x){ //cooperatively set localPairs values
                localPairs[idx] = 0;
        }

        if(i>=N)//only calculate if data set exists (due to cuda block size, some threads may have an i that is
larger than N
                return;

        __syncthreads();

        double distance;

        double myspAx = spAx[i];
        double myspAy = spAy[i];
        double mycZ = cZ[i];
        for (int j = i+1; j<N; j++) // loop over all events which have not yet been counted
        {
                if(abs(eventsDec[i]-eventsDec[j])*RadToDeg >= ANGLE_BINS*STEPPING) break; // don't
waste time by calculating large distances
```

```
                distance= GPUgetSpaceAngle(myspAx , myspAy , spAx[j] , spAy[j] , mycZ , cZ[j])*RadToDeg;
                // if(distance>maxautoscale*RadToDeg) continue;

                int tBin = floor((double)(distance*multiply)); // calculate bin (this is why STEPPING must be
constant)

                if(tBin*STEPPING == distance) //filter out cases where event is on the edge (to give identical
results as <= on the CPU)
                        tBin--;
                if(tBin >= ANGLE_BINS)
                        continue;

                for(int e=0; e<ENERGY_BINS; e++)
                {
                        // check if both events pass the energy cut
                        if(eventsEnergy[i]>=energyCut[e] && eventsEnergy[j]>=energyCut[e])
                        {
                                atomicAdd(&localPairs[e*ANGLE_BINS + tBin] , 1);
                        } // end if energy
                } // end loop energy ranges
                //}//end loop declination>decmin
        } // end event loop j
        __syncthreads();
        for(int idx = threadIdx.x;idx<NBINS;idx += blockDim.x){  //cooperatively write results to global
memory
                atomicAdd(&resultsThetaPairs[idx] , localPairs[idx]);
                int a = idx;
                while(a%ANGLE_BINS != 0){ //add all values of lower bins
                        a--;
                        atomicAdd(&resultsThetaPairs[idx] , localPairs[a]);

                }
        }

        return;
}
vector <struct AutoResults> calculate(vector<double> *energy_cuts , vector<struct AutoResults> *results ,
vector<struct event> *ScrambledEvents , int N){


        //Copy and reformat data to work on GPU


        double *GPUeventsDec;
        double *GPUeventsRa;
        double *GPUeventsEnergy;

        double *eventsDec =   (double*) malloc(N*sizeof(double));  // allocate memory on CPU
        double *eventsRa =    (double*) malloc(N*sizeof(double));
        double *eventsEnergy= (double*) malloc(N*sizeof(double));


        for(int i = 0;i < N ;i++){                                  //fill values
                eventsDec[i] =  (double)  (*ScrambledEvents)[i].declination;
                eventsRa[i] =   (double)  (*ScrambledEvents)[i].ra;
                eventsEnergy[i] = (double) (*ScrambledEvents)[i].energy;
        }

        cudaMalloc((void**)&GPUeventsDec ,    N*sizeof(double));  //allocate memory on GPU
        cudaMalloc((void**)&GPUeventsRa ,     N*sizeof(double));
```

```
	cudaMalloc((void**)&GPUeventsEnergy , N*sizeof(double));
	//copy data to GPU


	cudaMemcpy(GPUeventsDec ,    eventsDec    , N*sizeof(double) , cudaMemcpyHostToDevice);
	cudaMemcpy(GPUeventsRa ,     eventsRa     , N*sizeof(double) , cudaMemcpyHostToDevice);
	cudaMemcpy(GPUeventsEnergy , eventsEnergy , N*sizeof(double) , cudaMemcpyHostToDevice);

	//initialize angle caches
	double *spAx; //saves values for angle calculation
	double *spAy;
	double *cZ;

	double *cpuA = (double*) malloc(N*sizeof(double));
	for(int i =0;i<N;i++){
		cpuA[i] = 0;
	}


	cudaMalloc((void**)&spAx , N*sizeof(double));
	cudaMalloc((void**)&spAy , N*sizeof(double));
	cudaMalloc((void**)&cZ , N*sizeof(double));
	cudaMemcpy(spAx , cpuA , N*sizeof(double) , cudaMemcpyHostToDevice);
	cudaMemcpy(spAy , cpuA , N*sizeof(double) , cudaMemcpyHostToDevice);
	cudaMemcpy(cZ   , cpuA , N*sizeof(double) , cudaMemcpyHostToDevice);


	calcAngle<<<N/(blockSize) + 1 , blockSize>>>(N , GPUeventsDec , GPUeventsRa , spAx , spAy , cZ);



	double *energyCut =  (double*) malloc(ENERGY_BINS*sizeof(double));
	double *GPUenergyCut;

	for(int i = 0; i < ENERGY_BINS ; i++){
		energyCut[i] = (*energy_cuts)[i];
	}


	cudaMalloc((void**)&GPUenergyCut , ENERGY_BINS*sizeof(double));
	cudaMemcpy(GPUenergyCut , energyCut , ENERGY_BINS*sizeof(double) ,
cudaMemcpyHostToDevice);

	unsigned int *resultsThetaPairs =  (unsigned int*) malloc(NBINS*sizeof(unsigned int));
	float *resultsThetaValues = (float*) malloc(NBINS*sizeof(float));

	unsigned int *GPUresultsThetaPairs;
	float *GPUresultsThetaValues;

	cudaMalloc((void**)&GPUresultsThetaPairs  , NBINS*sizeof(unsigned int));
	cudaMalloc((void**)&GPUresultsThetaValues , NBINS*sizeof(float));

	for(int i = 0; i < ENERGY_BINS ; i++){
		for(int j = 0;j<ANGLE_BINS;j++){
			resultsThetaPairs[i*ANGLE_BINS + j] = 0;
			resultsThetaValues[i*ANGLE_BINS + j] = (float) (*results)[i].thetavalues[j];
		}

	}
```

```
		cudaMemcpy(GPUresultsThetaPairs  , resultsThetaPairs  , NBINS*sizeof(unsigned int) ,
cudaMemcpyHostToDevice);
		cudaMemcpy(GPUresultsThetaValues , resultsThetaValues , NBINS*sizeof(float) ,
cudaMemcpyHostToDevice);









	cout<<"starting GPU kernel..."<<endl;
	gpuwork<<<N/blockSize + 1,blockSize>>>(N , GPUeventsDec , GPUeventsRa , GPUeventsEnergy ,
			GPUenergyCut , GPUresultsThetaValues , GPUresultsThetaPairs , spAx , spAy , cZ);
	cudaDeviceSynchronize();
	cout<<"...GPU kernel finished"<<endl;

	unsigned int *thResults = (unsigned int*) malloc(NBINS*sizeof(unsigned int));

	cudaMemcpy(thResults, GPUresultsThetaPairs, NBINS*sizeof(unsigned int) ,
cudaMemcpyDeviceToHost);

	cudaError  err=cudaGetLastError();
	if(err != cudaSuccess){
		cout<<"-------------------- Cuda Error ----------------------"<<endl;
		cout<<"cuda returned error code:"<<endl;
		cout<<cudaGetErrorString(err)<<endl;
	}

	for(int i = 0; i < ENERGY_BINS ; i++){

		for(int j = 0;j<ANGLE_BINS;j++){
			(*results)[i].thetapairs[j] = thResults[i*ANGLE_BINS + j];
		//	cout<<"Energy "<< i << " angle bin " << j << " has " << thResults[i*angleBins + j] << "
hits " << endl;
		}

	}

	cudaFree(GPUenergyCut);
	cudaFree(GPUeventsDec);
	cudaFree(GPUeventsRa);
	cudaFree(GPUeventsEnergy);
	cudaFree(GPUresultsThetaPairs);
	cudaFree(GPUresultsThetaValues);
	cudaFree(spAx);
	cudaFree(spAy);
	cudaFree(cZ);
	free(cpuA);
	free(energyCut);
	free(eventsDec);
	free(eventsRa);
	free(eventsEnergy);
	free(resultsThetaPairs);
	free(resultsThetaValues);
  //cudaDeviceReset();
	return *results;
}
```

# Millipede, create Redundancy:

```
void
splinetable_create_redundant_data(struct splinetable *table){


	int ndim = table->ndim;
	int redundancy = table->order[table->ndim - 1] + 1;
	unsigned long total_size = table->strides[0]*table->naxes[0];
	unsigned long redundant_size = total_size;
	unsigned long redPos = 0;
	float *redundantData = (float*) malloc(redundancy*redundant_size*sizeof(float));
	unsigned long **newStrides = (unsigned long**) malloc(redundancy*sizeof(unsigned long*));
	printf("creating redundancy... \n");
	for(int perm = 0;perm < redundancy ;perm++){
		/* boxes that are not full are ignored, as ndsplineeval ensures that full box is in range,
		 *  this makes strides for higher dimensions slightly shorter */
		int d1realL = table->naxes[ndim-1] - ((table->naxes[ndim-1] - perm) % redundancy);  // new length of 1st
dim
		int stride3 = table->naxes[ndim-2]*(d1realL -perm);
		newStrides[perm] = (unsigned long*) malloc(table->ndim*sizeof(unsigned long));
		newStrides[perm][ndim-1] = 1;
		newStrides[perm][ndim-2] = table->naxes[ndim-1];
		newStrides[perm][ndim-3] = table->naxes[ndim-2]*table->naxes[ndim-1];//stride3;

			for(int i = ndim-4; i >= 0 ;i--){
				newStrides[perm][i] = newStrides[perm][i+1]*table->naxes[i+1];
			}


		for(unsigned long pos = 0;pos<total_size;pos++){
			int dim_1 = (pos / table->strides[ndim-1] ) % table->naxes[ndim-1]; //position in  1st dim
			int dim_2 = (pos / table->strides[ndim-2] ) % table->naxes[ndim-2]; //position in  2nd dim
			/* ignore incomplete boxes */
			if(dim_1 >=  d1realL || dim_1 < perm)
				continue;

			redPos = 0; // position of coefficients[pos] in redundant array

			/*higer dimensions do not change */
			for(int dim = 0;dim<table->ndim-2;dim++){
				redPos += ((pos / table->strides[dim]) % table->naxes[dim])*newStrides[perm][dim];
			}

			redPos += dim_2*redundancy; //shift beween boxes
			redPos += ((dim_1 - perm) / redundancy)*redundancy*table->naxes[ndim-2]; //row
			redPos += (dim_1 - perm) % redundancy; //position in box

			redundantData[perm*redundant_size + redPos] = table->coefficients[pos];

		}
	}
	table->redundantData = redundantData;
	table->redundantStrides = newStrides;

	//for(int i = 0;i<100000;i++)
	//		printf(" %d %f \n " , i , table->redundantData[i]);

}
```