FIAS Frankfurt Institute for Advanced Studies

Portable generic applications for GPUs and multi-core processors:

An analysis of possible speedup, maintainability and verification at the example of track reconstruction for ALICE at LHC

David Rohr

Frankfurt Institute for Advanced Studies

Perspectives of GPU computing in Science 2106

Rome, 28.9.2016

Why GPUs?

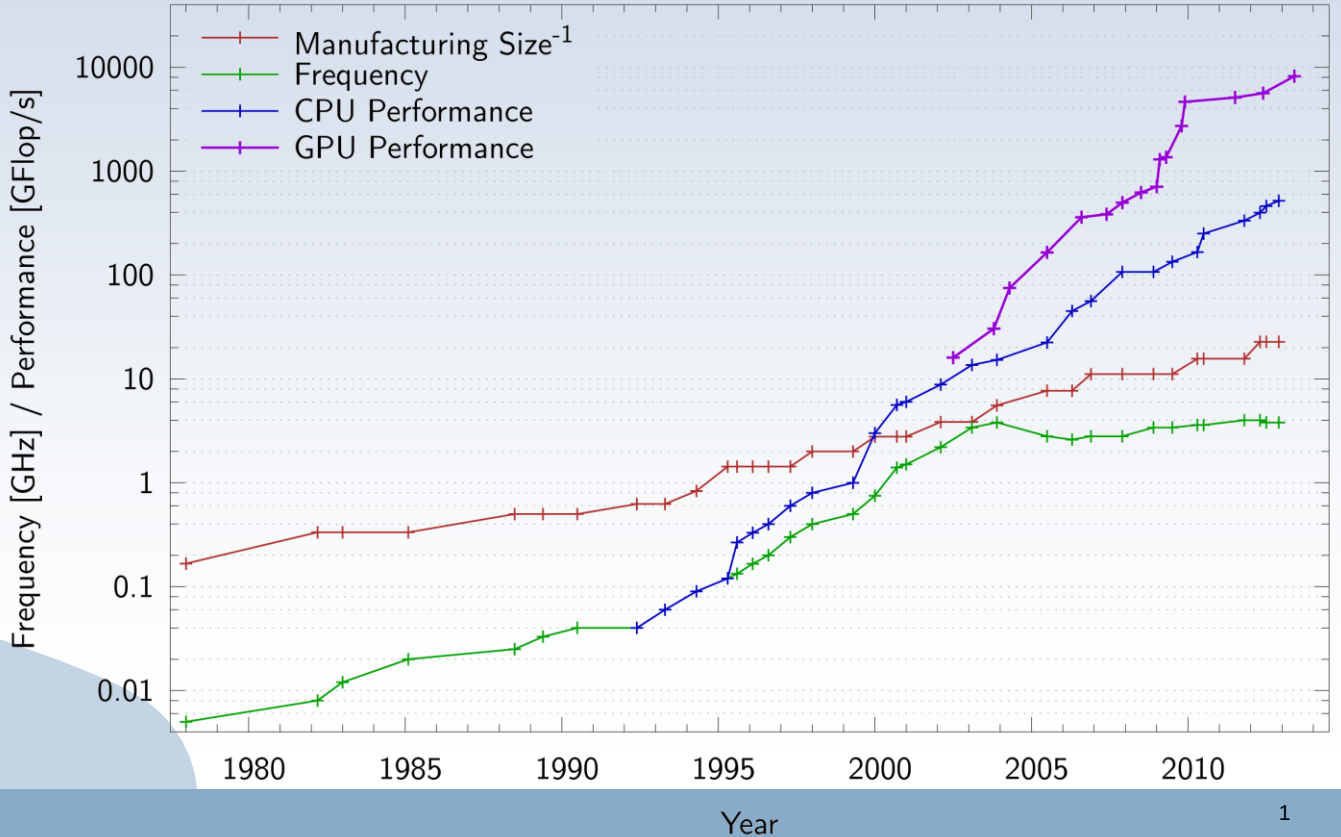
FIAS Frankfurt Institute for Advanced Studies

Transistors become smaller and smaller.

CPU clocks have stagnated!

Higher Performance through parallelism.

GPUs have higher peak performance than traditional processors.



Why GPUs?

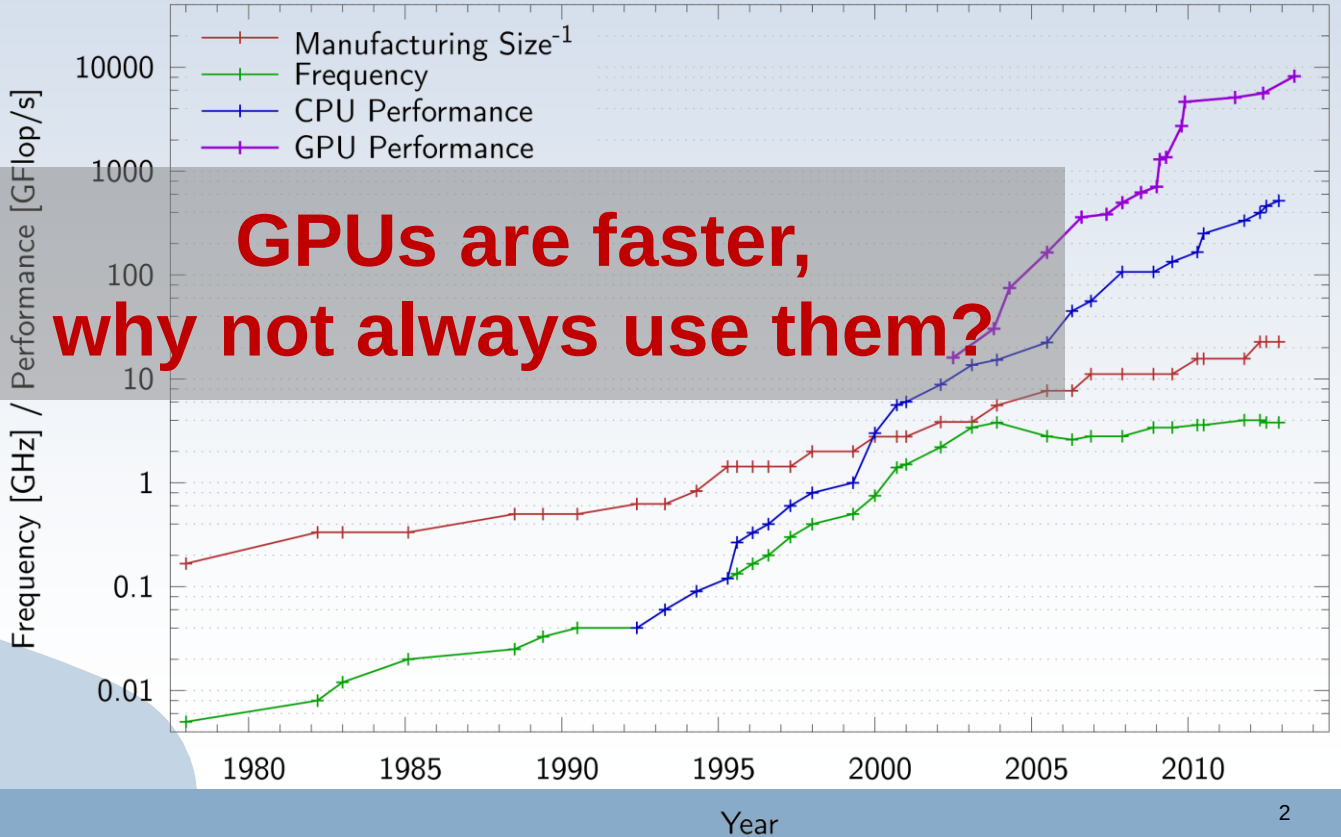
FIAS Frankfurt Institute for Advanced Studies

Transistors become smaller and smaller.

CPU clocks have stagnated!

Higher Performance through parallelism.

GPUs have higher peak performance than traditional processors.



GPUs are faster, why not always use them?

Why GPUs?

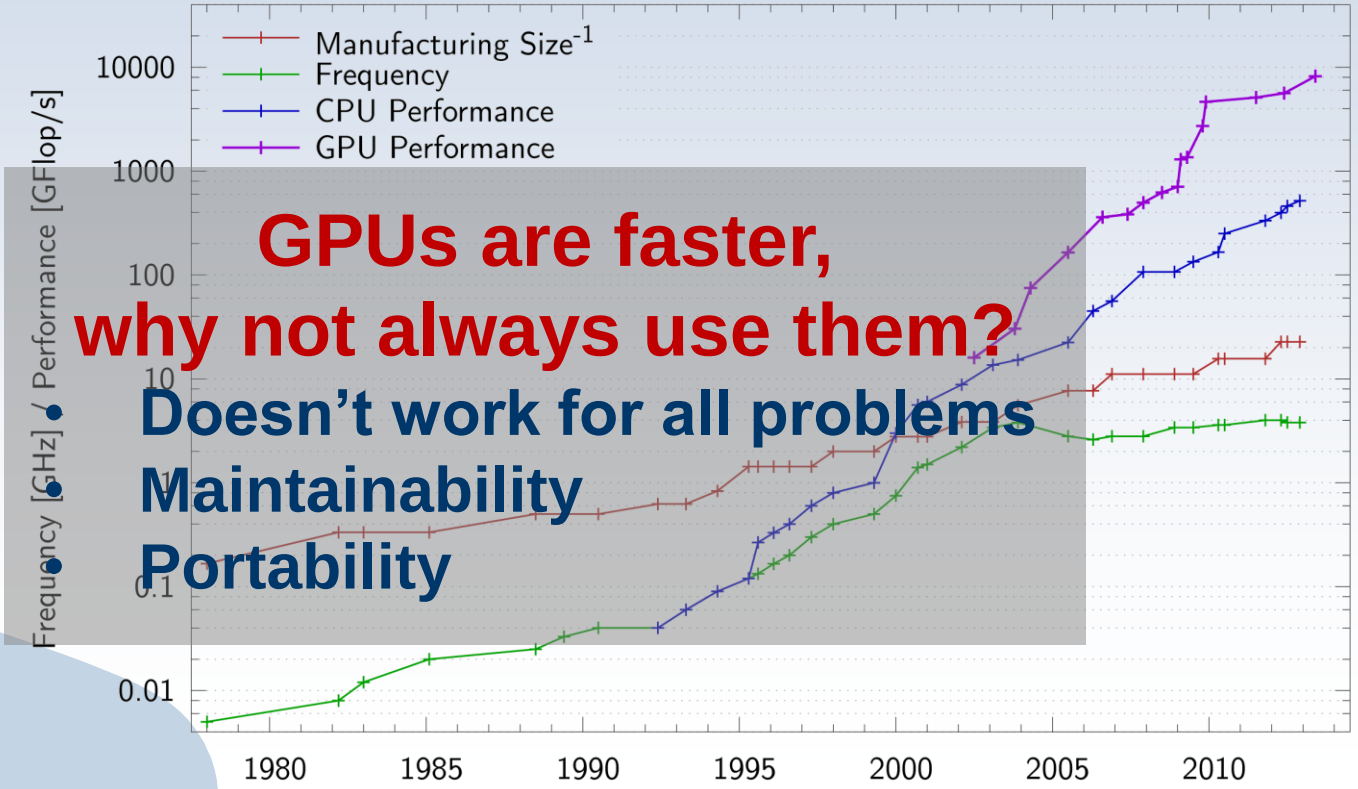
FIAS Frankfurt Institute for Advanced Studies

Transistors become smaller and smaller.

CPU clocks have stagnated!

Higher Performance through parallelism.

GPUs have higher peak performance than traditional processors.

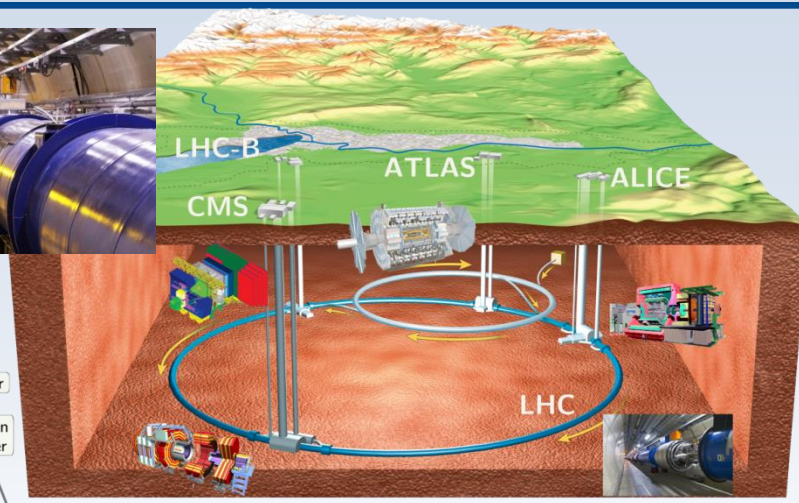


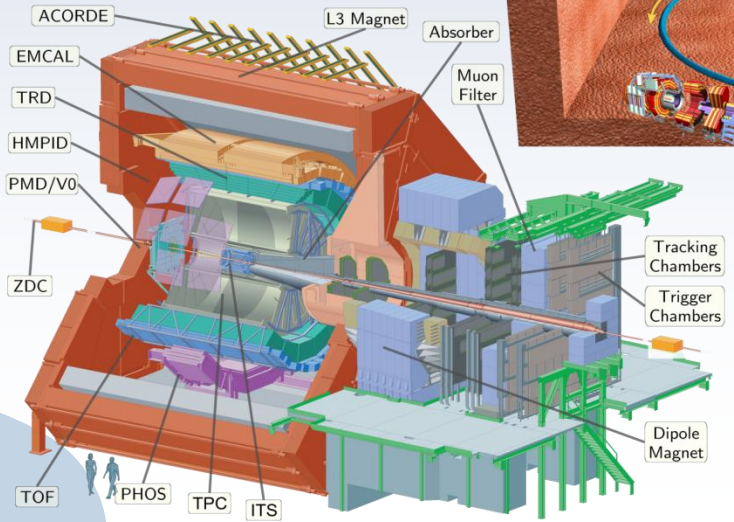
GPUs are faster, why not always use them?

- **Doesn't work for all problems**
- **Maintainability**
- **Portability**

ALICE at the LHC

- The Large Hadron Collider (LHC) at CERN is today‘s most powerful particle accelerator colliding protons and lead ions.
- ALICE is one of the four major experiments, designed primarily for heavy ion studies.
- The High Level trigger (HLT) is an online compute farm for real-time data reconstruction for ALICE.
- Most compute-intensive task is track reconstruction.

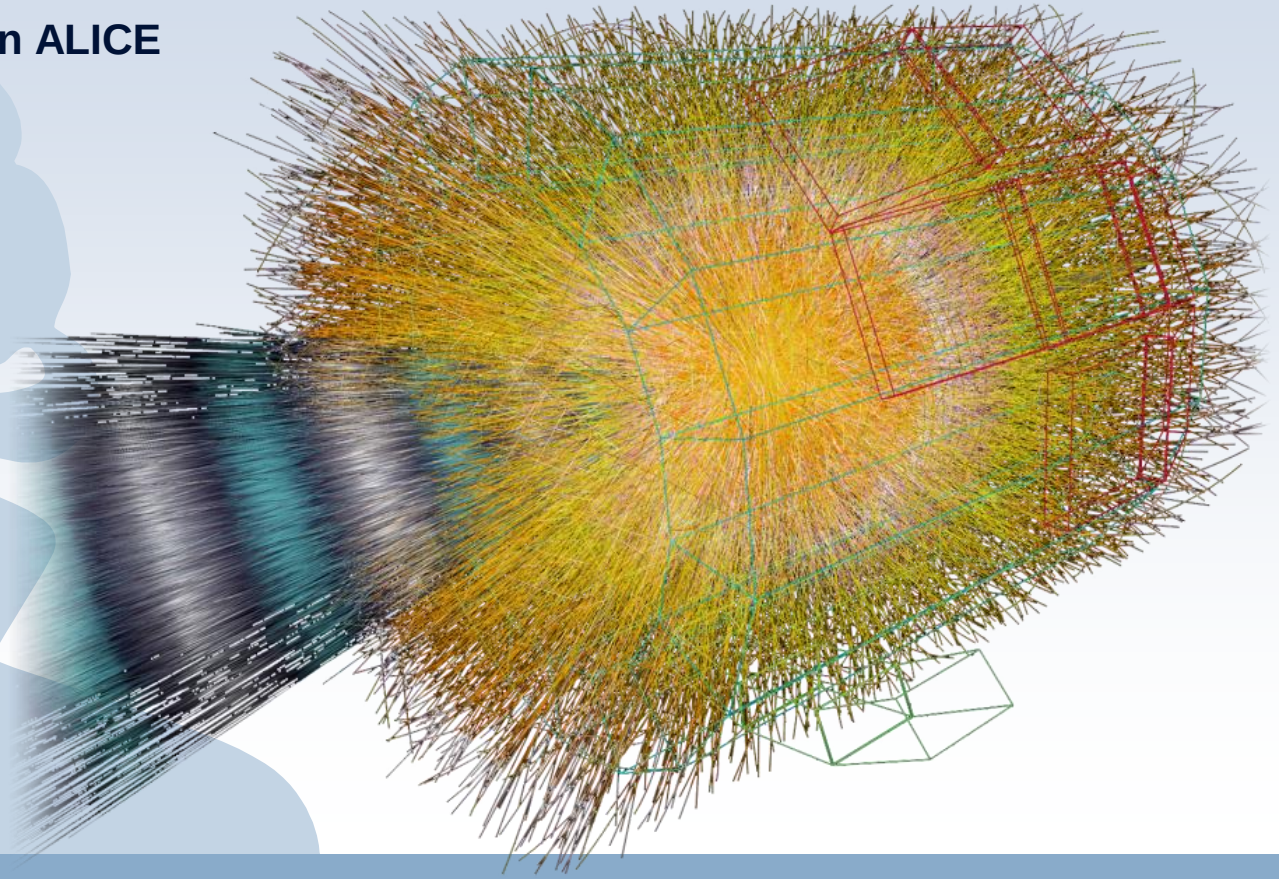




Track reconstruction in ALICE

FIAS Frankfurt Institute for Advanced Studies

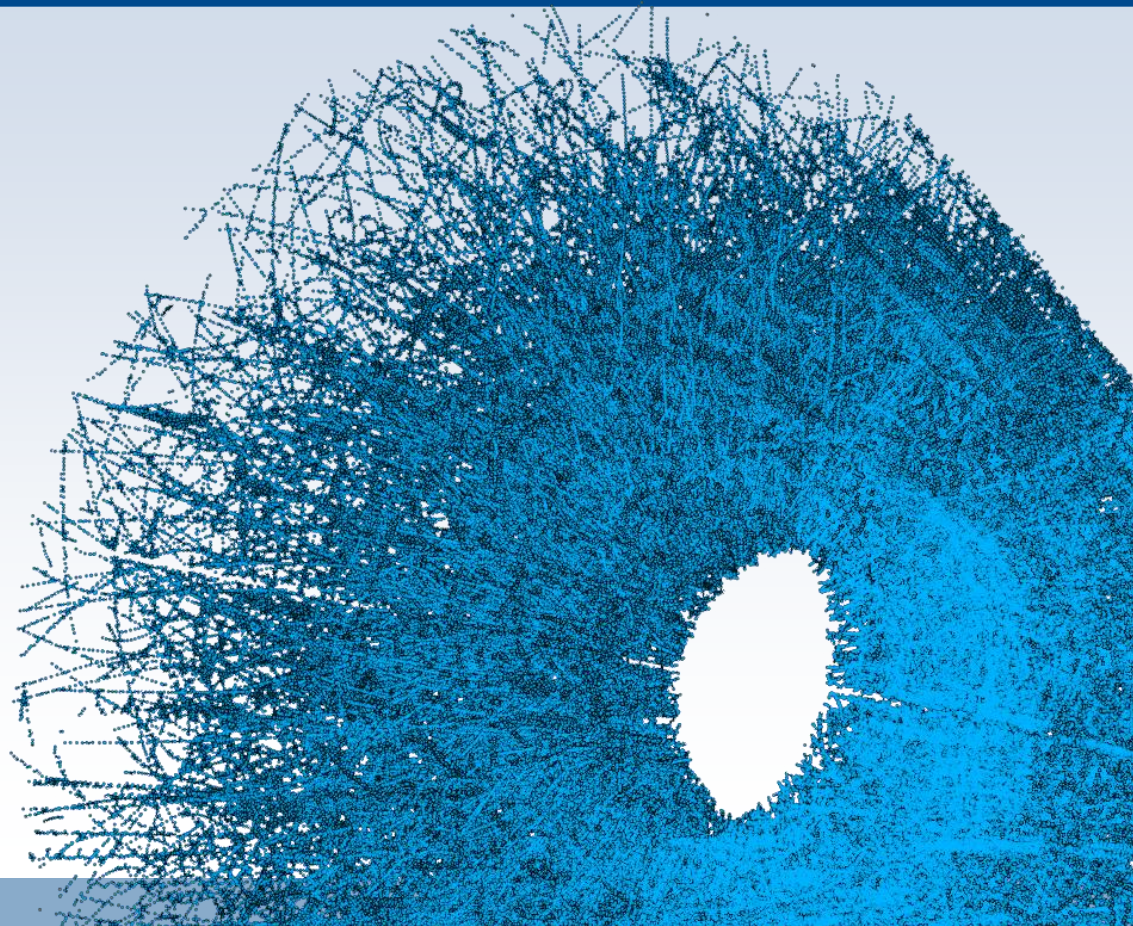
Lead-Lead event in ALICE



Track reconstruction in ALICE

FIAS Frankfurt Institute for Advanced Studies

Measured 3D space points in event.

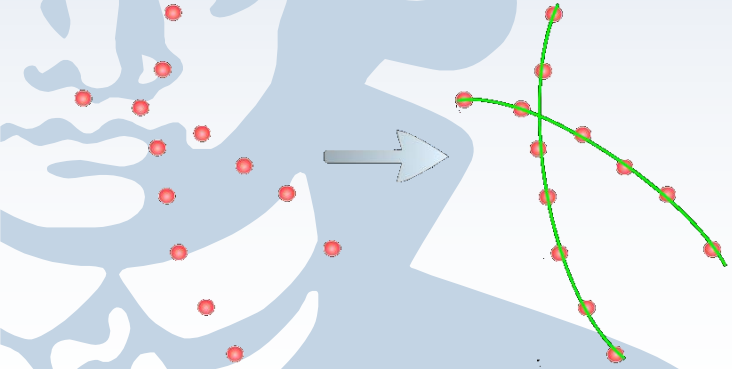


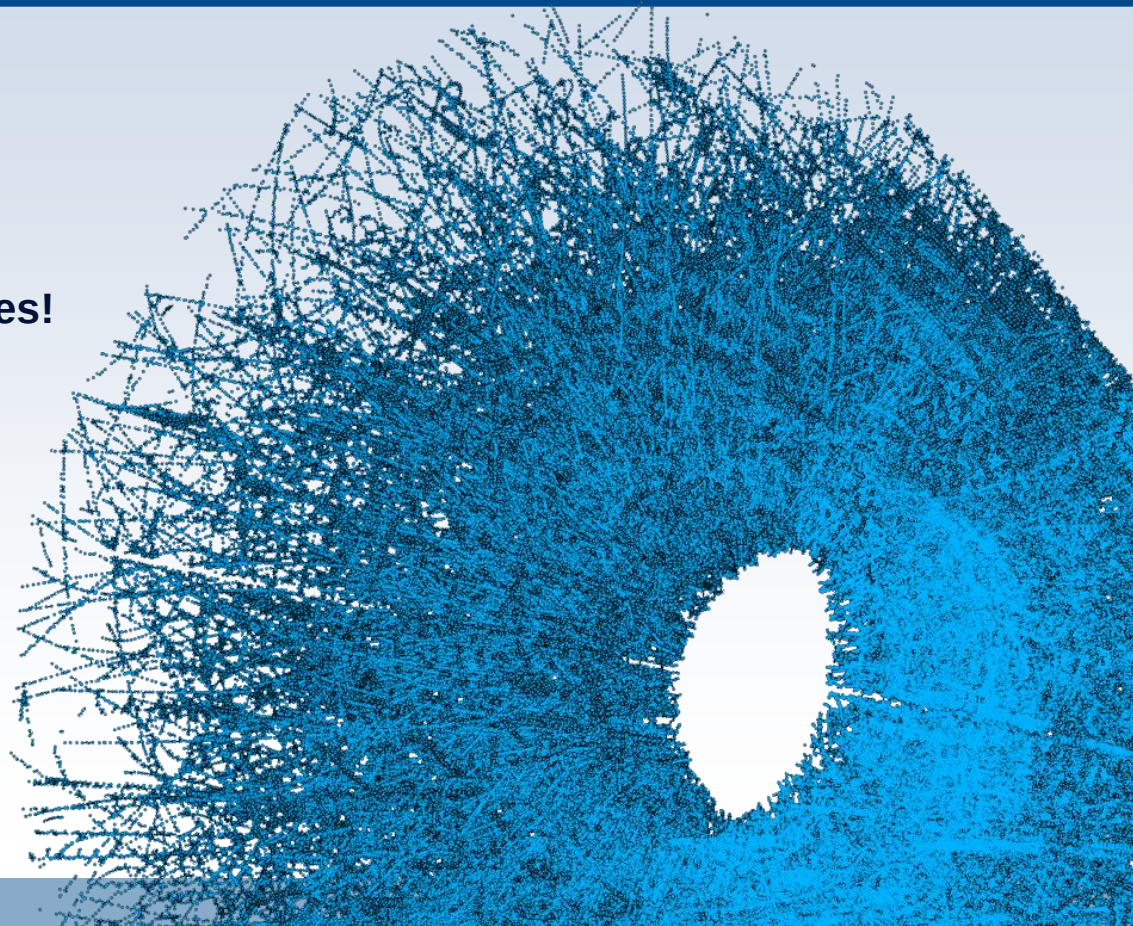
Track reconstruction in ALICE

FIAS Frankfurt Institute for Advanced Studies

Measured 3D space points in event.

Tracking needs to find the trajectories!

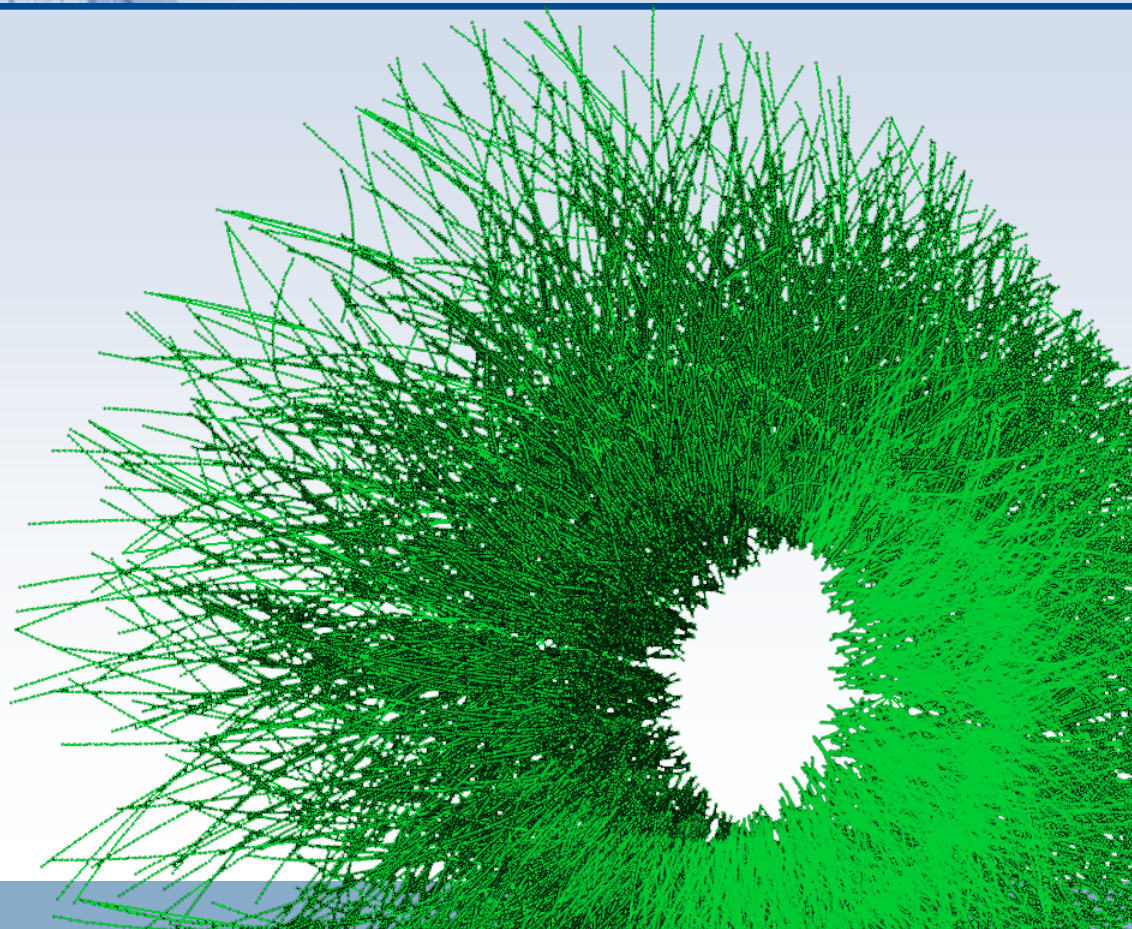




Track reconstruction in ALICE

FIAS Frankfurt Institute for Advanced Studies

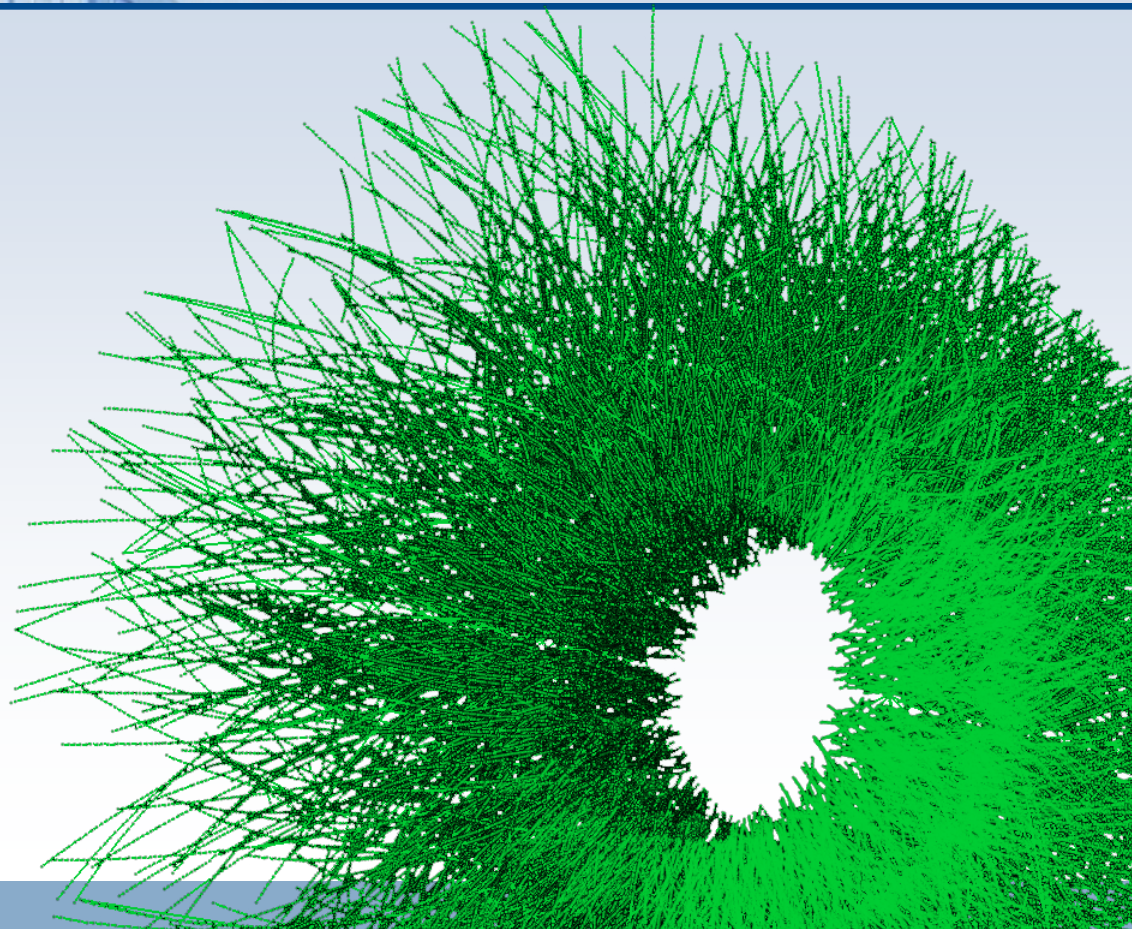
Trajectories found in event



Track reconstruction in ALICE

FIAS Frankfurt Institute for Advanced Studies

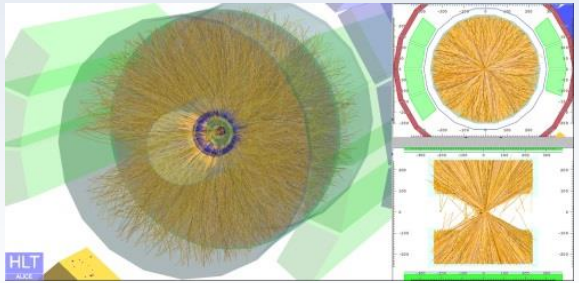
Trajectories found in event



There is plenty of parallelism,
So let‘s try GPUs.

Summary (current ALICE Tracking)

- **Tracking on GPU ca 3 times faster than full processor with all cores.**
  - But how to define speedup? See later!
- **GPU and CPU results consistent and reproducible.**
- **GPU Tracker runs on CUDA, OpenCL, OpenMP – one common shared source code.**
- **Now: 180 compute nodes with GPUs in the HLT as of 2015.**
  - First deployment: 2010 – 64 GPUs in LHC Run 1.
  - Since 2012 in 24/7 operation, no problems yet.
- **Cost savings compared to an approach with traditional CPUs:**
  - About 500.000 US dollar during ALICE Run I.
  - Above 1.000.000 US dollar during Run II.
  - Mandatory for future experiments, e.g.. CBM (FAIR, GSI) with >1TB/s data rate.



Requirements for CPU code

- **Portability**
  - Not all CERN GRID tier centers have GPUs (most do not!).
  - GPU model and vendor may vary.
  - CPU main / sole compute device in the GRID, GPUs used for real-time reconstruction

- **We want the CPU code as reference!**
  - Debugging should be possible on the CPU.
  - GPU results should match as closely as possible – but cannot be identical.

- **To reduce maintenance effort, a single source code is mandatory!**

Common tracker source code

- **CPU and GPU tracker (in CUDA and OpenCL) share common source files.**
- **Specialist wrappers for CPU and GPU exist, that include these common files.**

common.cpp:

```
__DECL FitTrack(int n) {
….
}
```

cpu_wrapper.cpp:

```
#define __DECL void
#include ``common.cpp``

void FitTracks() {
  for (int i = 0;i < nTr;i++) {
    FitTrack(n);
  }
}
```

cuda_wrapper.cpp and opencl_wrapper:

```
#define __DECL __device void
#include ``common.cpp``

__global void FitTracksGPU() {
  FitTrack(threadIdx.x);
}

void FitTracks() {
  FitTracksGPU<<<nTr>>>();
}
```

→ **Same source code for CPU and GPU version**

- The macros are used for API-specific keywords only.
- The fraction of common source code is above 90%.

Requirements for CPU code

- **The ALICE reconstruction and simulation framework AliRoot is based on C++.**
  - Track reconstruction must be C++.
  - OpenCL and C++ is a complicated story, which leaves (left) NVIDIA CUDA as sole alternative in the beginning.
- **Since last year, we use OpenCL with AMD C++ kernel extensions on AMD GPUs.**
  - CUDA still supported through common source code.
  - Unfortunately, this makes OpenCL single-vendor too – only AMD supports it.
  - Performance comparison inconvenient, because we cannut use the same API.
- **We are really hoping for C++ extensions in OpenCL 2.0.**
- **We support:**
  - AMD GPUs via OpenCL and C++ extensions
  - NVIDIA GPUs via CUDA
  - CPUs (via OpenMP if needed)

  } Common source code
  - Prototype for SSE / AVX / Xeon Phi via Vector library (Vc)

Which source code to have common?

FIAS Frankfurt Institute for Advanced Studies

- **In any case, the host code should be identical.**
  - Complicated long kernels codes could be shared.
  - Specialized hand-tuned short (assembler-like) kernels should be created for every device.
    - Also, special versions of “hot-spot” device functions can be used by a common kernel.
- **The host code should have a generic (or abstract) interface to use.**
  - We use templates or virtual classes here (only for the “management” code).
    - One derived class of virtual base class for every supported API.
    - A virtual function call to initiate a DMA transfer / start a kernel is **no overhead**!
    - **No virtual functions in performance critical device code**.
      - Anyway: limited availability for virtual functions in APIs (CUDA does it for some time now).
- **If a class that is used on the GPU shall have virtual functions (for management on the host):**
  - We have a non-virtual base-class used on the GPU (do not want virtual calls there anyway).
  - Virtual functions only added on derived class used on the host (data layout of base class remains).
  - This works even when the GPU API does not have virtual features.

Common tracker source code

- **Do not overdo it!**

Which API shall we use for our application?

Common tracker source code

- **Do not overdo it!**

Which API shall we use for our application?

Well, I just checked. There are 10 APIs. Each has pros and cons.

Common tracker source code

- **Do not overdo it!**

Which API shall we use for our application?

Well, I just checked. There are 10 APIs. Each has pros and cons.

So which do we use?

Common tracker source code

- **Do not overdo it!**

Which API shall we use for our application?

Well, I just checked. There are 10 APIs. Each has pros and cons.

So which do we use?

Hey, I got an idea. Let's create a new general API that abstracts all others

Common tracker source code

- **Do not overdo it!**

Which API shall we use for our application?

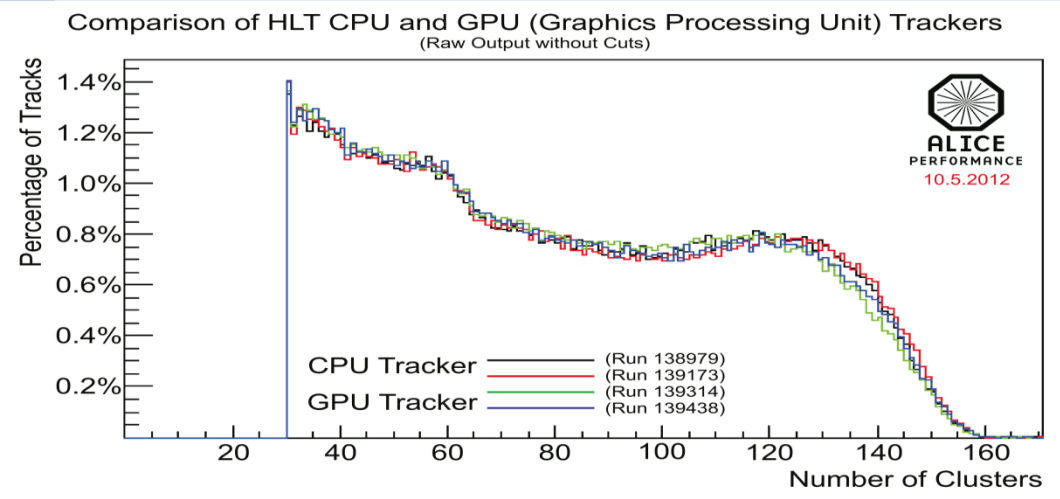
- **Use one that fits now!**
  - This is a rapidly changing field.
  - Keep your code generic so you can switch.
  - C and (restricted) C++ code can be executed everywhere, CUDA / OpenCL are not so different.
  - Do not use fancy features where not needed.
- **Try to start to have a common code for the CPU and for your API of choice!**

Consistency of Tracking Results

Even though the source code is identical, GPU and CPU yield different results.

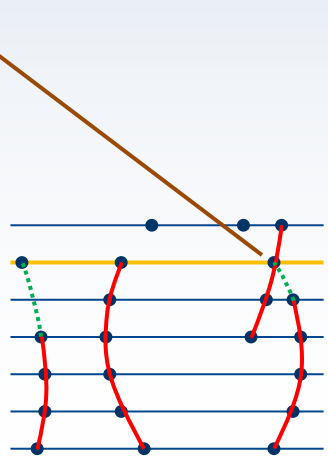
We identified three causes:

- Cluster to track assignment
- Variances during track merging
- Non-associative floating point arithmetic



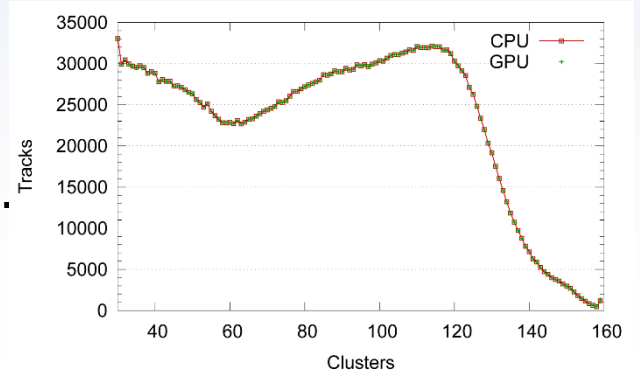
Consistency of Tracking Results

- **Cluster to track assignment**
  - Problem: Cluster to track assignment was depending on the order of the tracks.
    - Each cluster was assigned to the longest possible track. Out of two tracks of the same length, the first one was chosen.
    - Concurrent GPU tracking processes the tracks in an undefined order.
  - Solution: We need a continuous (floating point) measure of the track quality.
    - Two 32-bit floats can still be identical, but that is unlikely.
- **Similar problem in track merging, which depended on track order.**



Consistency of Tracking Results

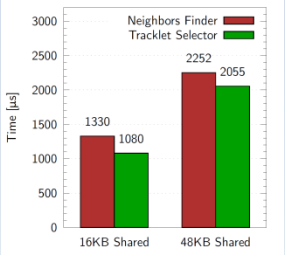
- **Non associative floating point arithmetic**
  - Problem: Different compilers perform the arithmetic in different order (also on the CPU).
  - Solution: Cannot be fixed, but...
    - Slight variations during the extrapolations do not matter as long as the clusters stay the same.
    - Inconsistent clusters: 0,00024%
- **Now, perfect match of CPU and GPU results in plots…**
- …But not binarily.



Generic Performance Optimization

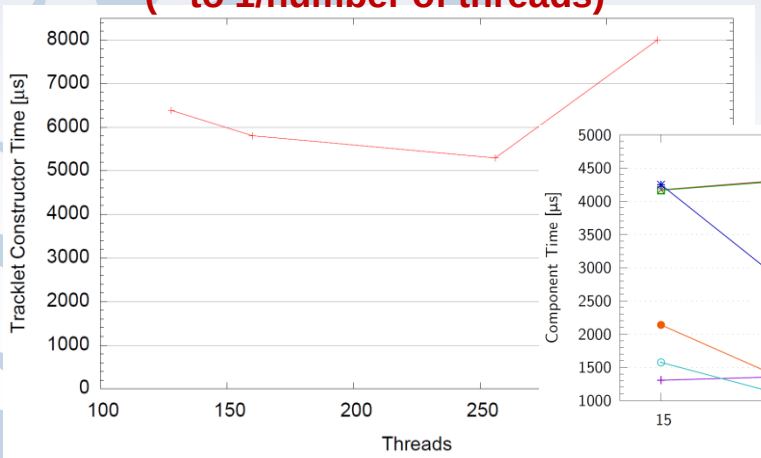
- **Not all processors are the same…**
  - how to optimize for all of them?
  - A code tailored for GPUs is not necessarily optimal for CPUs.
  - Many features can be parameterized.

Shared Memory Size

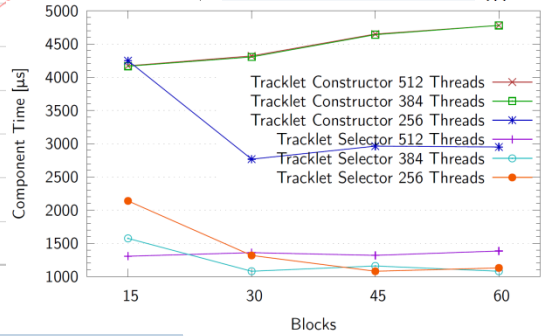


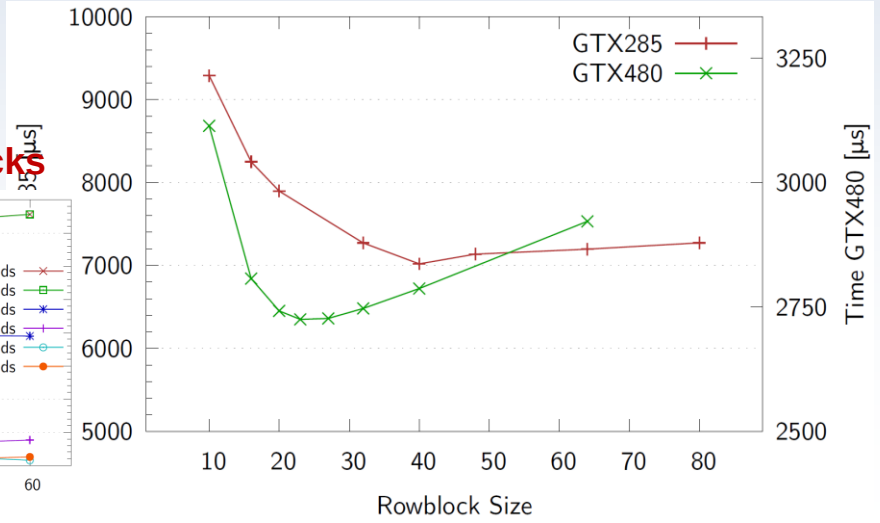
Algorithm internal parameters.

Number of registers (~ to 1/number of threads)



Number of threads / blocks





Generic Performance Optimization

- **All new features we added can be switched off.**
- **Parameters can be changed easily - mostly at compile time.**
  - Via templates
  - or preprocessor directives.
- **Through runtime-compilation (CUDA / OpenCL), one can still easily run parameter scans.**

- **Essentially, it took us three iterations to add / parameterize features:**
  - NVIDIA GTX 285 (First version)
  - NVIDIA GTX 480 (New GPU Model)
  - AMD S9000 (OpenCL / Other Vendor / New GPU)

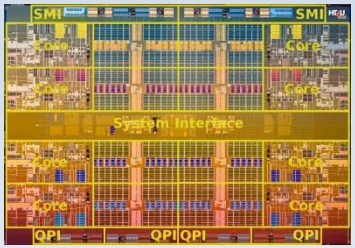
- **For new GPUs, we could find good parameters via parameter range scan.**
  - For instance, 140 ms → 50 ms switching from Keppler to Maxwell.
  - Of course, we do not really know whether this is optimal.

Generic Performance Optimization

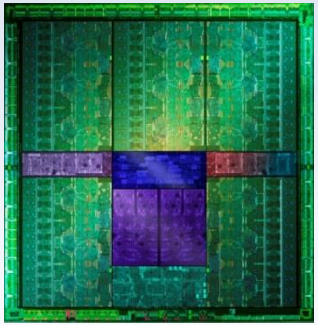
- **What about special GPU features: e.g. shared memory?**
  - Does not really matter.
  - Every memory on the CPU is “shared”.
  - Use (thread-local) normal memory for reductions etc.
  - Activate / deactivate explicit shared memory caches via pointer access.
- **Single or double precision?**
  - Only single or mixed, usually you don’t need double everywhere.

- **The biggest problem we face (with low-level APIs):**
  - SIMD v.s. SIMT
  - In fact, the Hardware is the same:
    - One instruction decoder.
    - Vector or vector-like processing.
    - Essentially, a GPU multiprocessor is a core.
    - A warp is a vector-processing unit.
  - But the programming is not the same.

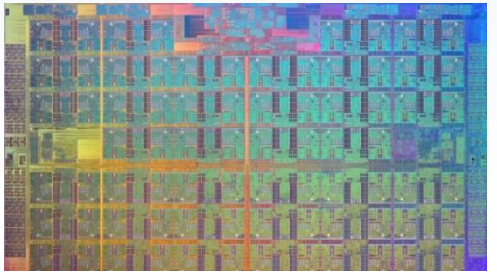
Xeon CPU



NVIDIA Kepler



Xeon Phi



SIMD v.s. SIMT

- **How to program for SIMD and SIMT, or how to support SSE/AVX/Xeon Phi and GPUs?**
  - Obviously no intrinsics.
  - Automatic parallelization / vectorization does not work (in a general scope).
  - Could use OpenCL (SIMD processors can run SIMT code using Masks, Gathers, Scatters). (Essentially, SIMT is SIMT + automatic masks, gathers, scatters.)
    - **No good experience with OpenCL on Xeon Phi KNC.**
    - **No OpenCL (yet?) for Xeon Phi KNL.**
    - OpenCL utilization of vector units of CPUs (SSE / AVX) suboptimal.
    - SSE / AVX lack SIMT instructions – getting better with AVX512 moving towards Xeon Phi ISA.
    - CPUs usually faster with OpenMP (+ vectorization) than with OpenCL.
  - Could use vector libraries (e.g. Vc) on GPU.
    - **Difficult to implement (in C++).**
    - **Vc C++ expects one thread, but SIMT "simulates" multiple threads.**

SIMD v.s. SIMT

- **This question is the reason why we currently have**
  - A common tracker for CUDA / OpenCL / OpenMP.
  - A forked prototype for Vc supporting AVX / Xeon Phi.

- **Could use both OpenCL / CUDA and Vc vector library togeter.**
  - For the GPU code, we use the scalar library version.

- **Then, Vc library could vectorize for AVX / Xeon Phi, SIMT would "vectorize" for GPUs.**
  - Vectorization should be efficient, because data structure guidelines are the same (SoA v.s. AoS).

→ More maintenance effort.

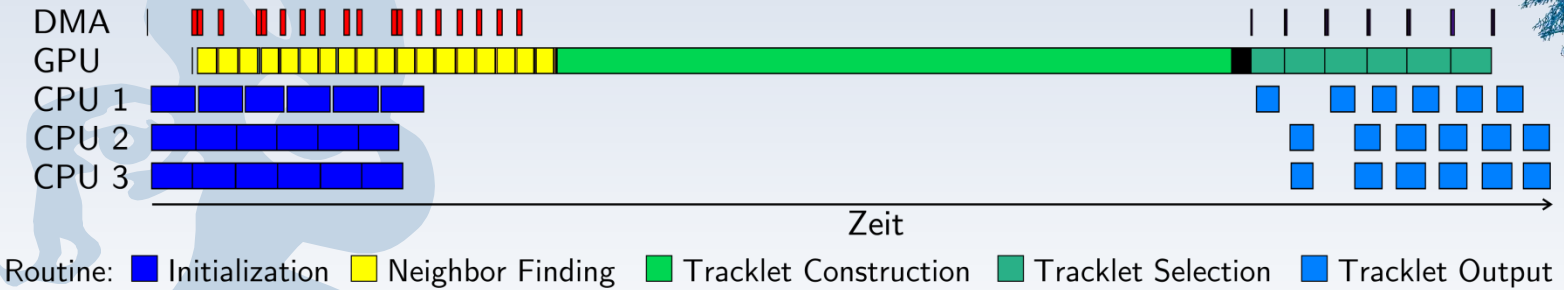
→ Possible compatibility issues.

→ Ugly!

- **No good solution yet.**

Optimizations (Performance)

- **Splitting of problems in parts, for us this means…**
- **separation of event in sectors enables the use of a Pipeline:**
  - Tracking on GPU, pre-/postprocessing on CPU, and data transfer run in parallel.



- **This is a very general concept which could apply to all GPU applications.**
  - (Not needed when data stays on the GPU all the time).
- **Splitting the workload usually simplifies processing a part on the CPU.**
  - However, in most cases we don't have a single workload.
  - We offload what runs efficiently on the GPU, and use the CPU for other tasks.
    - One can dedicate one core for scheduling with fast response.

Other Applications: Linpack

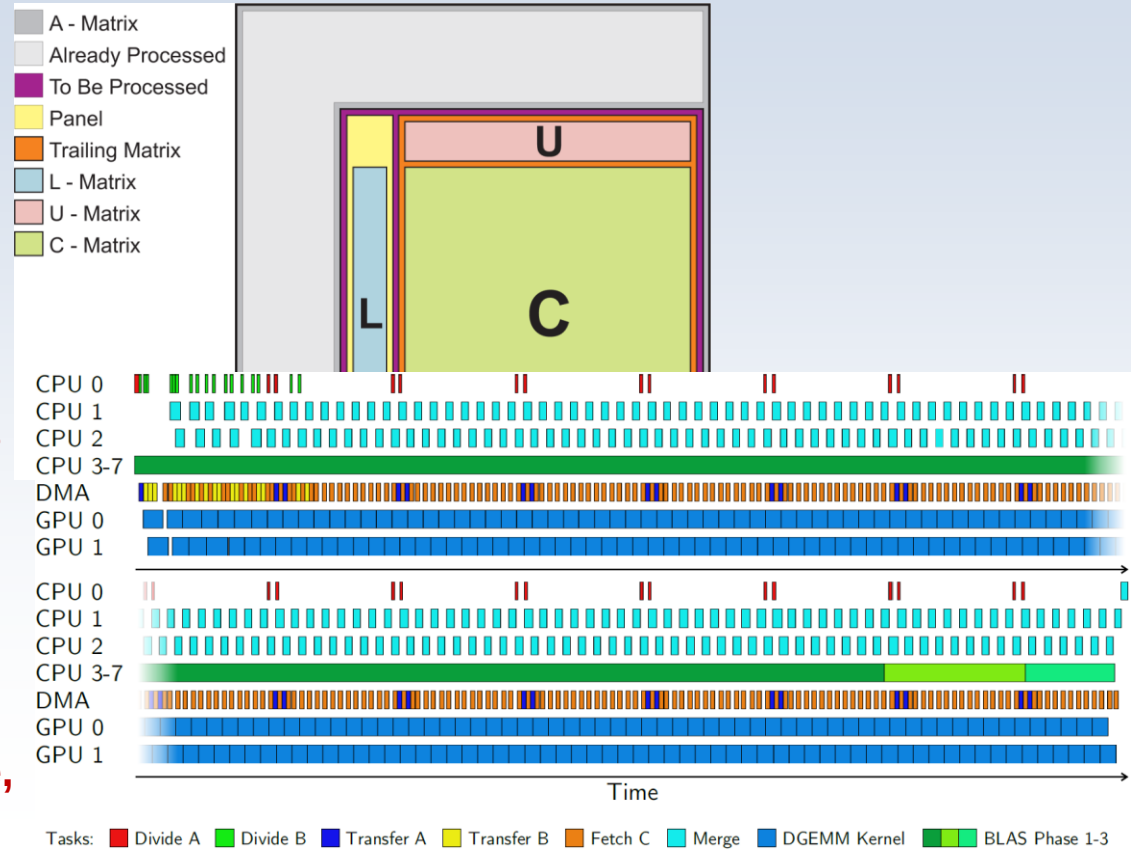
Linpack iteratively factorizes a dense system of linear equations.

- Heavy use of linear algebra (BLAS) library.
- Most time consuming step is matrix-matrix multiplication (DGEMM).
- DGEMM is ideally suited for GPUs.

A similar asynchronous pipeline is used, in this case with multiple GPUs.

Few special, non-common linear algebra kernels for each GPU.

Only host code shared in this case, which is the majority of the code.



Measuring speedup

FIAS Frankfurt Institute for Advanced Studies

- **What speedup to we get / can we expect from GPUs.**
  - How to measure it.

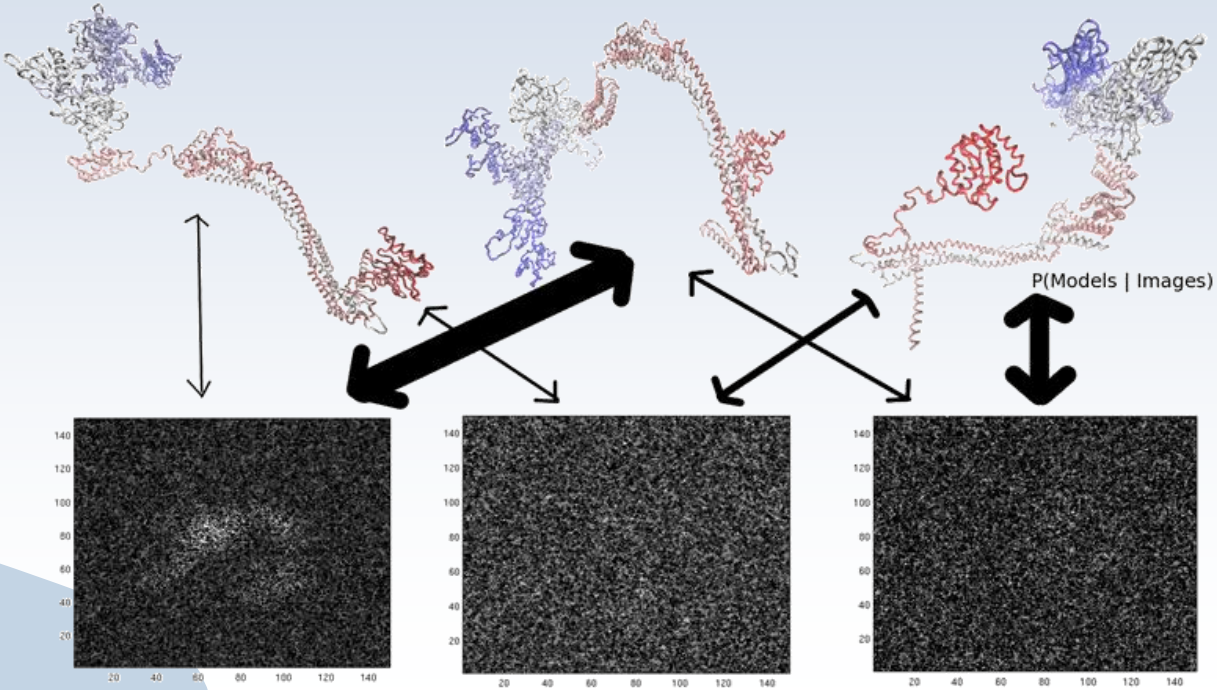
- **Stating only the time to solution might sound nice, but does not tell the whole store.**

Electron Microscopy Analysis

- **Question: Which model accounts best for the experimental EM image?**

Models

Images

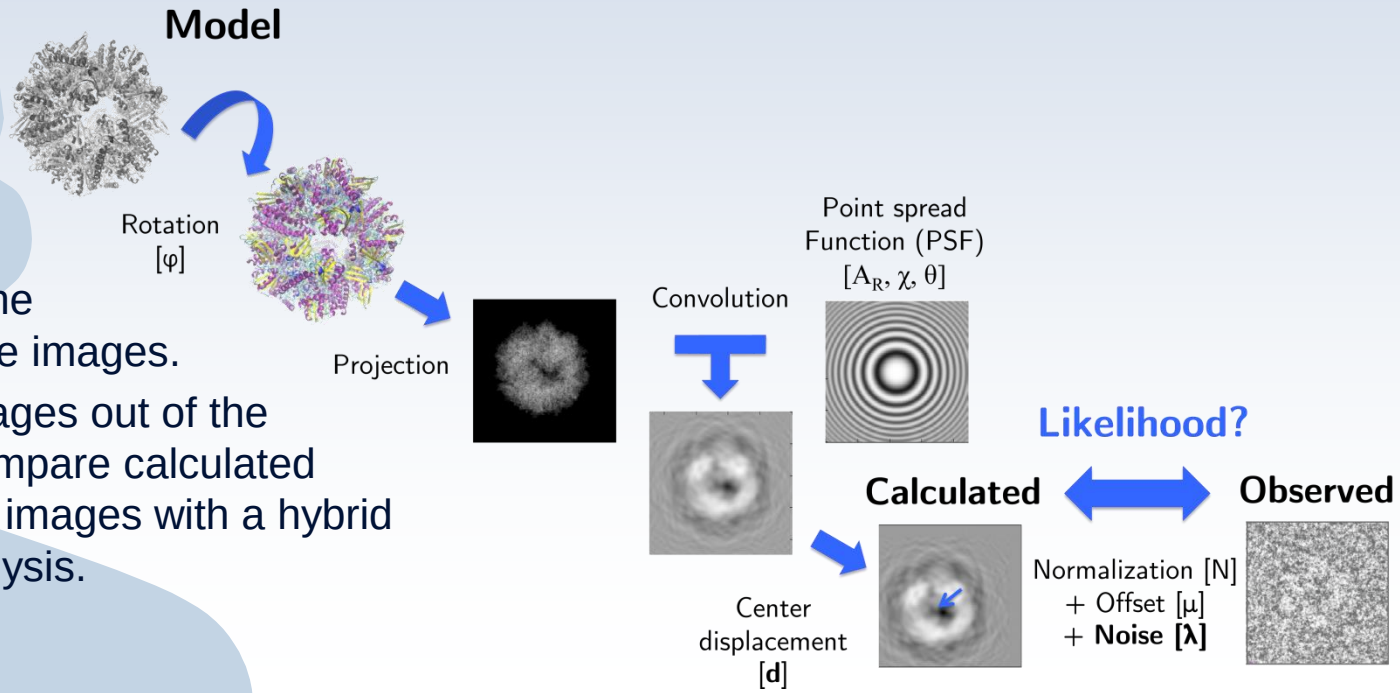


Electron Microscopy Analysis

- **Different approaches to this:**

a) Reconstruct the model from the images.

b) We create images out of the model and compare calculated and observed images with a hybrid Bayesian analysis.



Measuring speedup

FIAS Frankfurt Institute for Advanced Studies

- **What speedup to we get / can we expect from GPUs.**
  - How to measure it.

- **Total speedup of optimized BioEM (electron microscopy) program using GPU: 1000x – 13000x!!!**

Measuring speedup

FIAS Frankfurt Institute for Advanced Studies

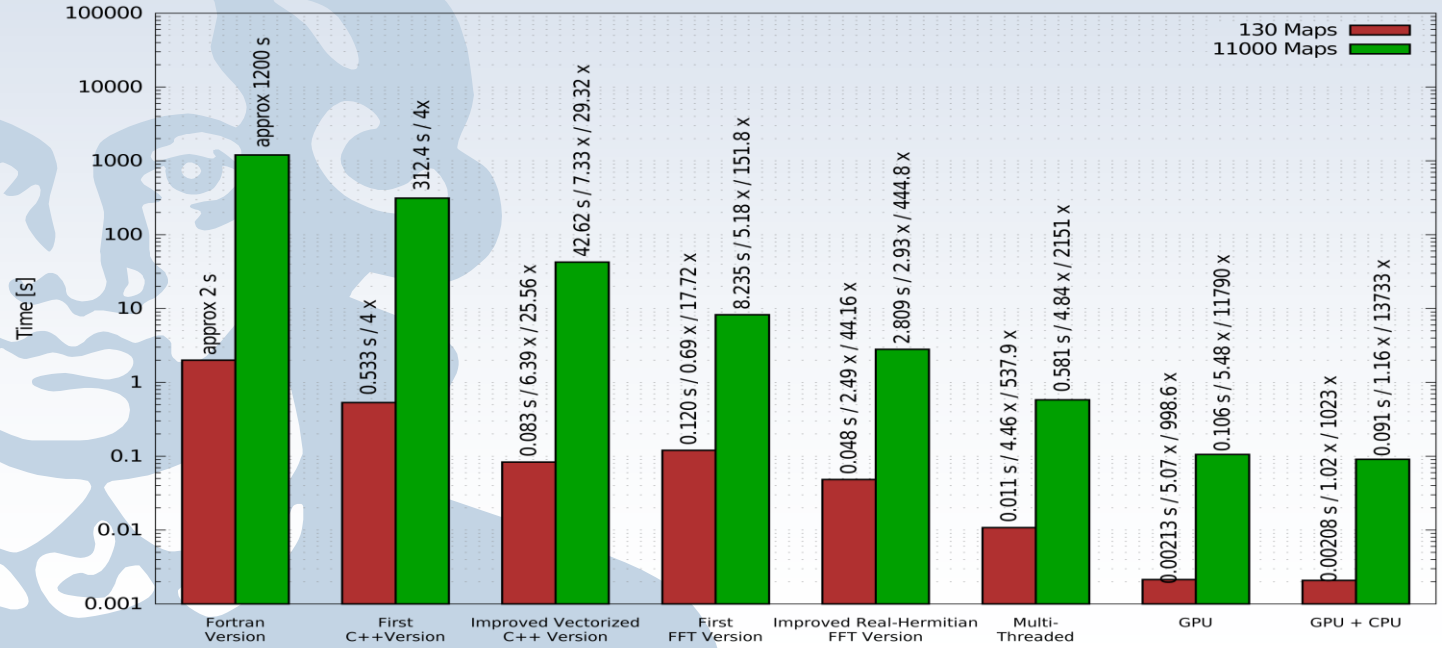
- **What speedup to we get / can we expect from GPUs.**
  - How to measure it.

- **Total speedup of optimized BioEM (electron microscopy) program using GPU: 1000x – 13000x!!!**
  - Is that apples compared to apples?
  - Which CPU, which GPU?
  - How many cores were used?
  - Was vectorization used?
  - Is it actually the same algorithm?
  - Is the result the same?

Electron Microscopy Analysis

FIAS Frankfurt Institute for Advanced Studies

- **Performance evolution over time:** **The plot shows execution time, speedup compared to previous version, and total speedup.**



- **Speedup depends on settings**
- **1000x – 13000x speedup**
- **1 year down to 40 min for 13000 maps**

Measuring speedup

FIAS Frankfurt Institute for Advanced Studies

- **What speedup to we get / can we expect from GPUs.**
  - How to measure it.

Measuring speedup

FIAS Frankfurt Institute for Advanced Studies

- **What speedup to we get / can we expect from GPUs.**
  - How to measure it.
- **Full GPU v.s. Full CPU (with all cores)**
  - I like this one, but how many cores?

Measuring speedup

FIAS Frankfurt Institute for Advanced Studies

- **What speedup to we get / can we expect from GPUs.**
  - How to measure it.
- **Full GPU v.s. Full CPU (with all cores)**
  - I like this one, but how many cores?
- **GPU v.s. one CPU core (and state it is one core)**
  - Can be misleading, but better.
  - But how does the CPU scale?

Measuring speedup

FIAS Frankfurt Institute for Advanced Studies

- **What speedup to we get / can we expect from GPUs.**
  - How to measure it.
- **Full GPU v.s. Full CPU (with all cores)**
  - I like this one, but how many cores?
- **GPU v.s. one CPU core (and state it is one core)**
  - Can be misleading, but better.
  - But how does the CPU scale?
- **Full GPU v.s. Full CPU / number of cores**
  - Even better, but
    - Scaling might depend on number of cores.
    - Which GPU after all?
    - Which CPU model, which frequency?

Comparing CPU / GPU Performance

FIAS Frankfurt Institute for Advanced Studies

- **The compute performance alone is no reasonable metric!**
  - The GPU is the faster chip – by construction.
  - The are many claims showing a 30x – 1000x speedup on GPU!
    → CPU code should be optimized before the comparison!

- **We consider the following relative efficiency:**

$$\varepsilon = \frac{\text{Efficiency on GPU}}{\text{Efficiency on CPU}} = \frac{(a_g/p_g)}{(a_c/p_c)} = \frac{a_g/a_c}{p_g/p_c}.$$

- **The advantage is of the second form is: achieved performance and theoretical peak performance can be measured in different units.**

Comparing CPU / GPU Performance

FIAS Frankfurt Institute for Advanced Studies

- **Overview of speedup in several applications:**

Benchmark	Type	Hardware	Performance	% of peak	Speedup	ε [%]
(old) HLT Tracker	Single	Nehalem 4C 3 GHz	1122 ms			
		GTX285 + CPU	312 ms		3.60	53
(new) HLT Tracker	Single	2×Magny-Cours 2.2 GHz	495 ms			
		GTX580 + CPU	155 ms		3.19	85
Track Merger	Single	Westmere 6C 4 GHz	65 ms			
		GTX580 + CPU	60 ms		1.10	13
DGEMM (Kernel)	Double	2×Magny-Cours 2.1 GHz	180 GFlop/s	89.3		
		5870	494 GFlop/s	90.8	2.74	102
		6970	624 GFlop/s	92.3	3.47	103
		7970	805 GFlop/s	84.4	4.47	95
DGEMM (System)	Double	2×Magny-Cours 2.1 GHz	180 GFlop/s	89.3		
		5870 + CPU	623.5 GFlop/s	83.6	3.46	94
		3×5870 + CPU	1435 GFlop/s	78.3	7.98	87
		2×6990	2292 GFlop/s	89.9	12.73	104
		2×S10000	2923 GFlop/s	79.8	16.24	89
One-Node HPL	Double	2×Magny-Cours 2.1 GHz	174.6 GFlop/s	86.6		
		5870 + CPU	563.2 GFlop/s	75.5	3.23	87
		3×5870 + CPU	1114 GFlop/s	60.7	6.38	70
		2×6990 + CPU	2007 GFlop/s	72.4	11.49	84
		2×S10000 + CPU	2679 GFlop/s	73.1	15.34	84
Erasure Codes (small n)	32-bit logical	Westmere 6 · 3.8 GHz	14.3 GB/s	74.7		
		GTX580	72.5 GB/s	75.3	5.32	102
		6970	51.1 GB/s	58.0	4.10	78
		Virtex 6 LX240 FPGA	2187.0 GB/s		152.94	
Erasure Codes (large n)	32-bit logical	Sandy Bridge 1 · 3.7 GHz	251.0 GAOp/s			
		Westmere 6 · 3.8 GHz	807.0 GAOp/s			
		GTX580	908.4 GAOp/s		1.13	19
		6970	1024 GAOp/s		1.27	26

Comparing CPU / GPU Performance

FIAS Frankfurt Institute for Advanced Studies

- **The compute performance alone is no reasonable metric!**
  - The GPU is the faster chip – by construction
- **We consider the following:**

$$\varepsilon = \frac{\text{Efficiency on GPU}}{\text{Efficiency on CPU}} = \frac{(a_g/p_g)}{(a_c/p_c)} = \frac{a_g/a_c}{p_g/p_c}.$$

- **Most of our applications reach about 70% or more in this metric.**
- **There are exclusions:**
  - PCI Express can limit the performance (track merger, encoding with small n / k).
  - CPU Compilers are better and allow more flexible core (JIT-compiled encoding).
  - CPU caches can better hide memory latencies (Electron Microscopy).

Suggestions

FIAS Frankfurt Institute for Advanced Studies

- **Compare specifications to requirements first, is the GPU suited for your program (PCIe limit etc.)**
- **Write generic source code, do not maintain multiple code bases!**
  - Simple code is easily portable. Use the when needed, not for fun.
- **Write fast code where it is critical, write “nice” code otherwise.**
  - Parameterize optimization features, to easily tune them for new hardware → portability.
  - Split problem in parts. Enables load balancing and pipelined processing.
- **Tell us exactly what you compare for the speedup.**
  - Results should include how it scales (to large GPUs / multiple cores).
  - Relative performance numbers can help to judge the efficiency (relative speedup often ca. 70%).
- **Optimizing “old” applications usually yields a great speedup on the CPU, too.**
  - Optimization strategies are not too different, the GPU and CPU architectures converge.
  - (Good code, optimized fort both CPU and GPU, often runs around 3x to 4x faster on GPU.)
- **Compiler optimizations will give you inconsistent floating point results on CPU.**
  - Do not expect this to be better on the GPU. Try to keep the algorithm consistent.
- **Use single precision where possible, double where needed, mixed is OK.**