GPU applications within HEP

Liang Sun
Wuhan University
2019-07-19

The 6th International Summer School on TeV Experimental Physics (iSTEP), Guangzhou

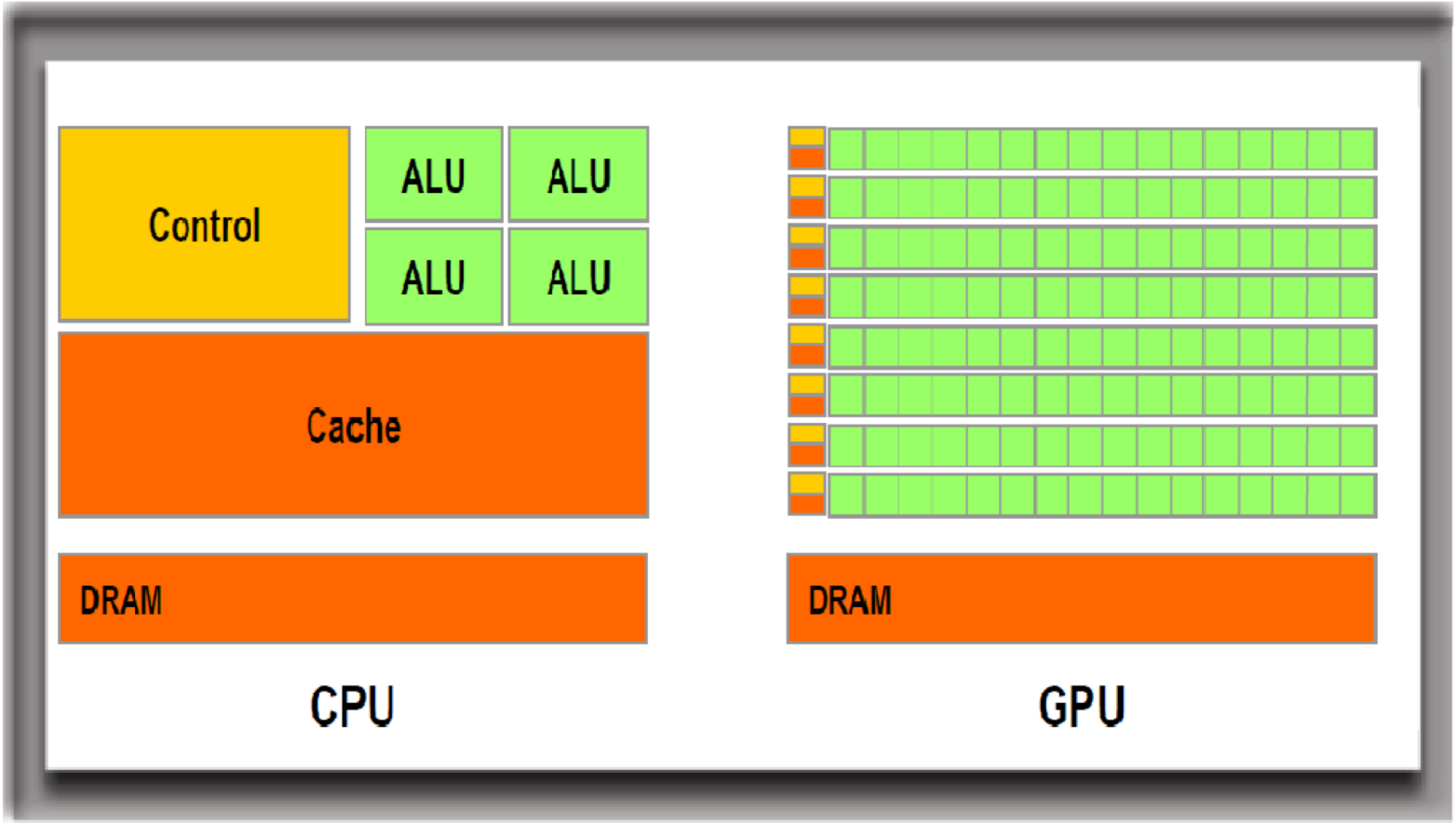
Outline

- Basic concepts
  - GPU, CUDA, Thrust
- Introduction on Dalitz-Plot Analysis
- Overview of HEP toolkits for amplitude analyses
- GooFit introduction
- Hydra introduction
- Summary

CPUs and GPUs

- The CPU (central processing unit) carries out all the arithmetic and computing functions of a computer. Principal components of a CPU: arithmetic logic unit (ALU),registers and a control unit
- The GPU (graphics processing unit) is specialized processor designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer. Modern GPUs have a highly parallel structure and are more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel

CPUs and GPUs



Concurrency

- The ability to execute or solve different parts of a program, an algorithm or a problem in out-of-order or in partial order, without affecting the final outcome
- Concurrent routines can be executed in parallel
- Significant improvement in the overall performance of the execution in multi-processor, multi-core and multi-thread systems
- Design of concurrent programs and algorithms requires reliable techniques for coordinating instruction execution, data exchange, memory allocation and execution scheduling to minimize response time and maximize throughput
- Issues: race conditions, deadlocks, resource starvation etc....

Motivation for massively parallel platforms in HEP

- A large fraction of the software used in HEP is legacy. It consists of libraries of single threaded, Fortran and C++03 mono-platform routines
- HEP experiments keep collecting samples with unprecedented large statistics.
- Data analyses get more and more complex. Not rarely, a calculation spend days to reach a result, which often needs re-tuning
- Processors will not increase clock frequency any more. The current road-map to increase overall performance is to deploy concurrency

NVidia GPUs

Geforce GTX 1080 Ti

GTX TITAN Z

GPU Architecture: Kepler
CUDA Cores 5760
Base Clock (MHz) 705
Single-Precision Performance 4.3 - 5.0 TeraFLOPS
Double-Precision Performance 1.4 - 1.7 TeraFLOPS
Memory Interface 12GB GDDR5

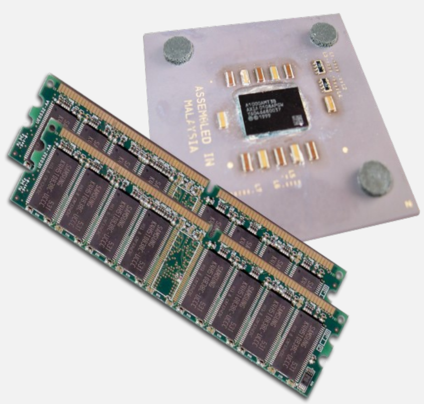
GPU Architecture: Pascal
CUDA Cores 3584
Base Clock (GHz) 1.126
Double-Precision Performance 4.7 TeraFLOPS
Single-Precision Performance 9.3 TeraFLOPS
Memory Interface 16GB CoWoS HBM2 at 732 GB/s

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host

Device

Heterogeneous Computing

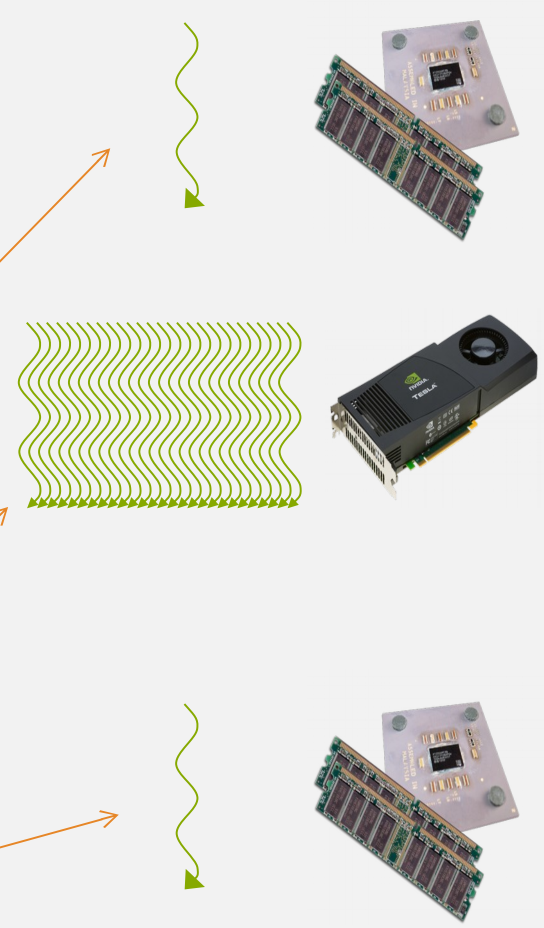
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
        __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
        int gindex = threadIdx.x + blockIdx.x * blockDim.x;
        int lindex = threadIdx.x + RADIUS;

        // Read input elements into shared memory
        temp[lindex] = in[gindex];
        if (threadIdx.x < RADIUS) {
                temp[lindex - RADIUS] = in[gindex -
RADIUS];
                temp[lindex + BLOCK_SIZE] = in[gindex +
BLOCK_SIZE];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

        // Apply the stencil
        int result = 0;
        for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                result += temp[lindex + offset];

        // Store the result
        out[gindex] = result;
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

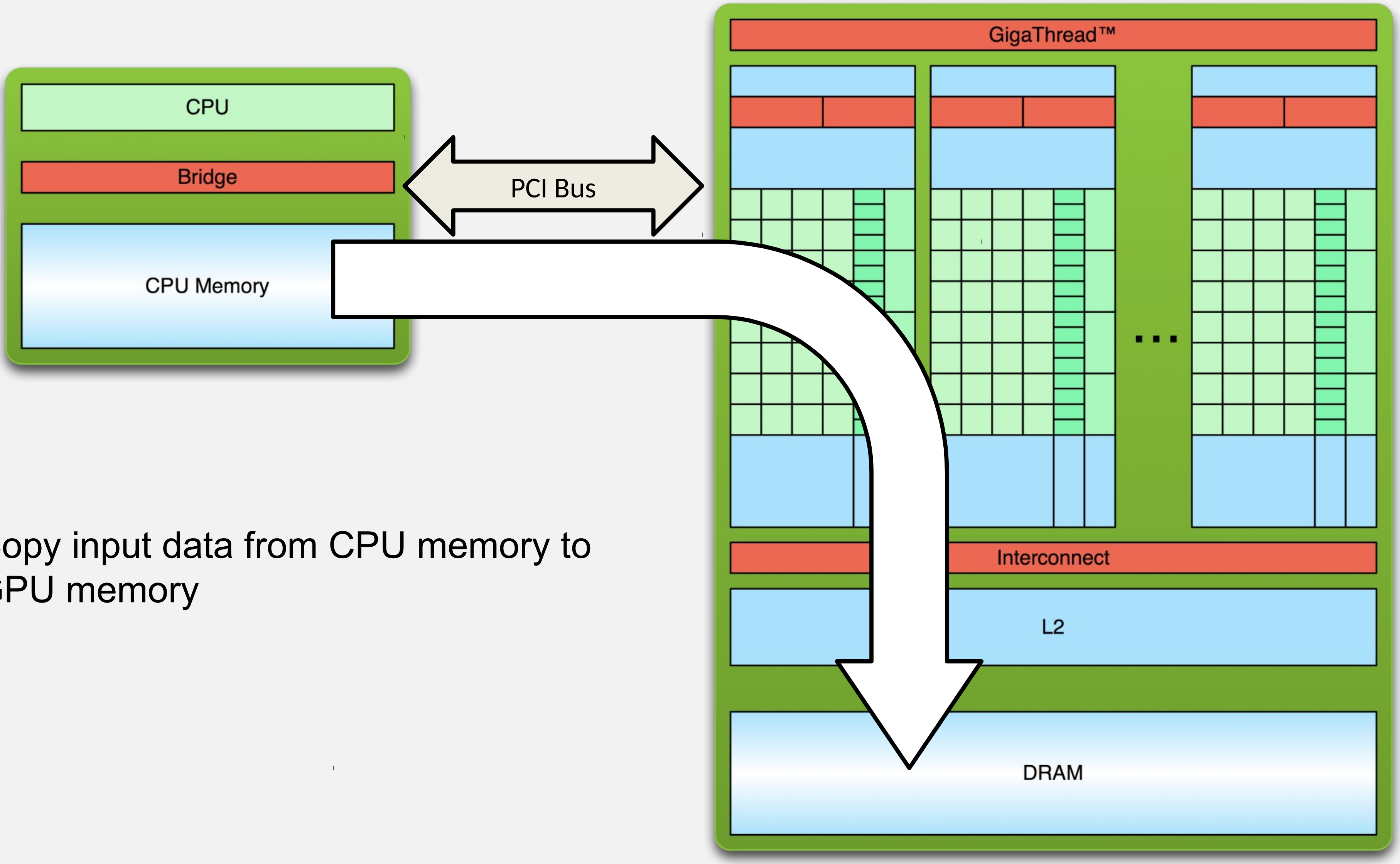
serial code

parallel code

serial code

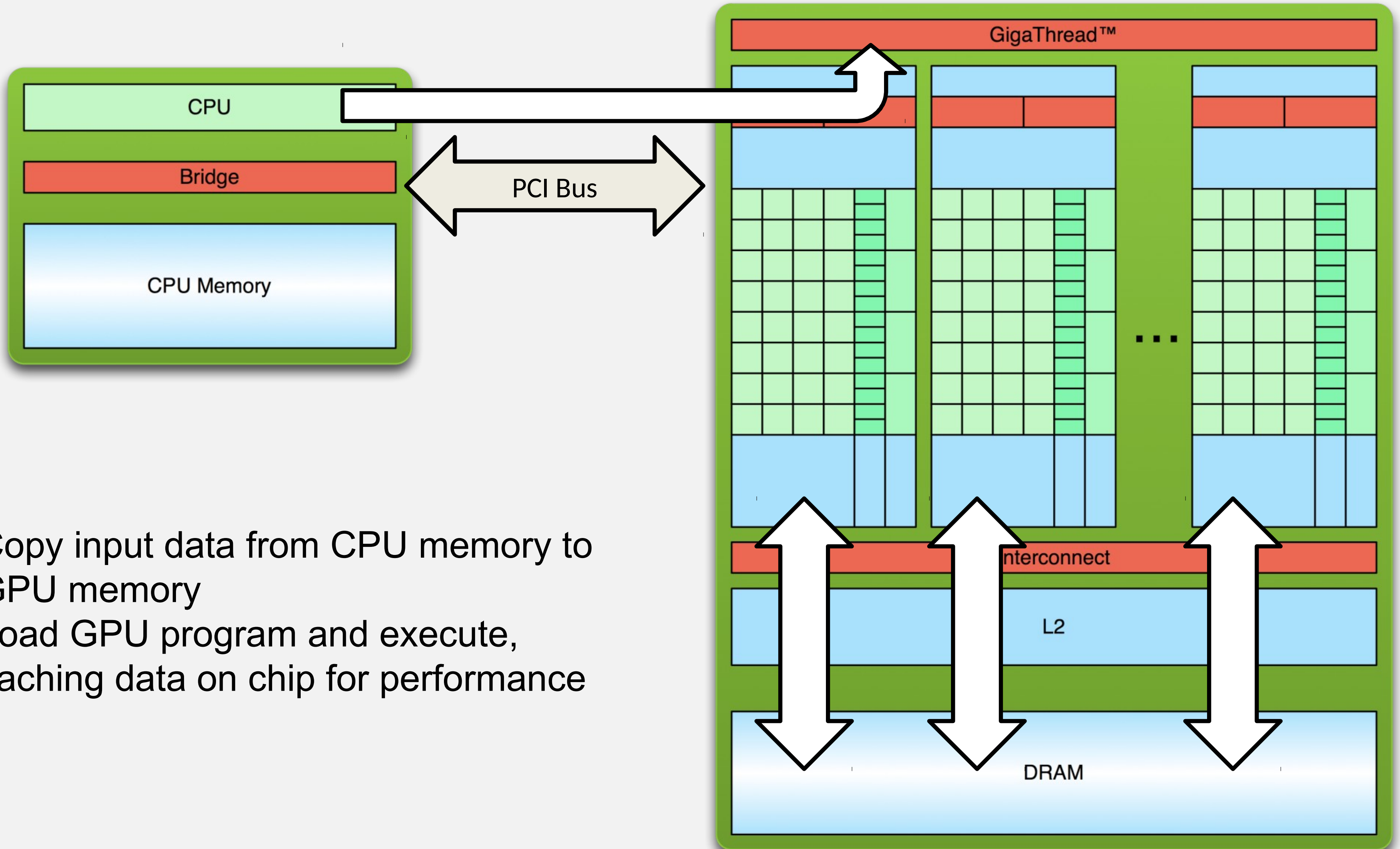


Simple Processing Flow



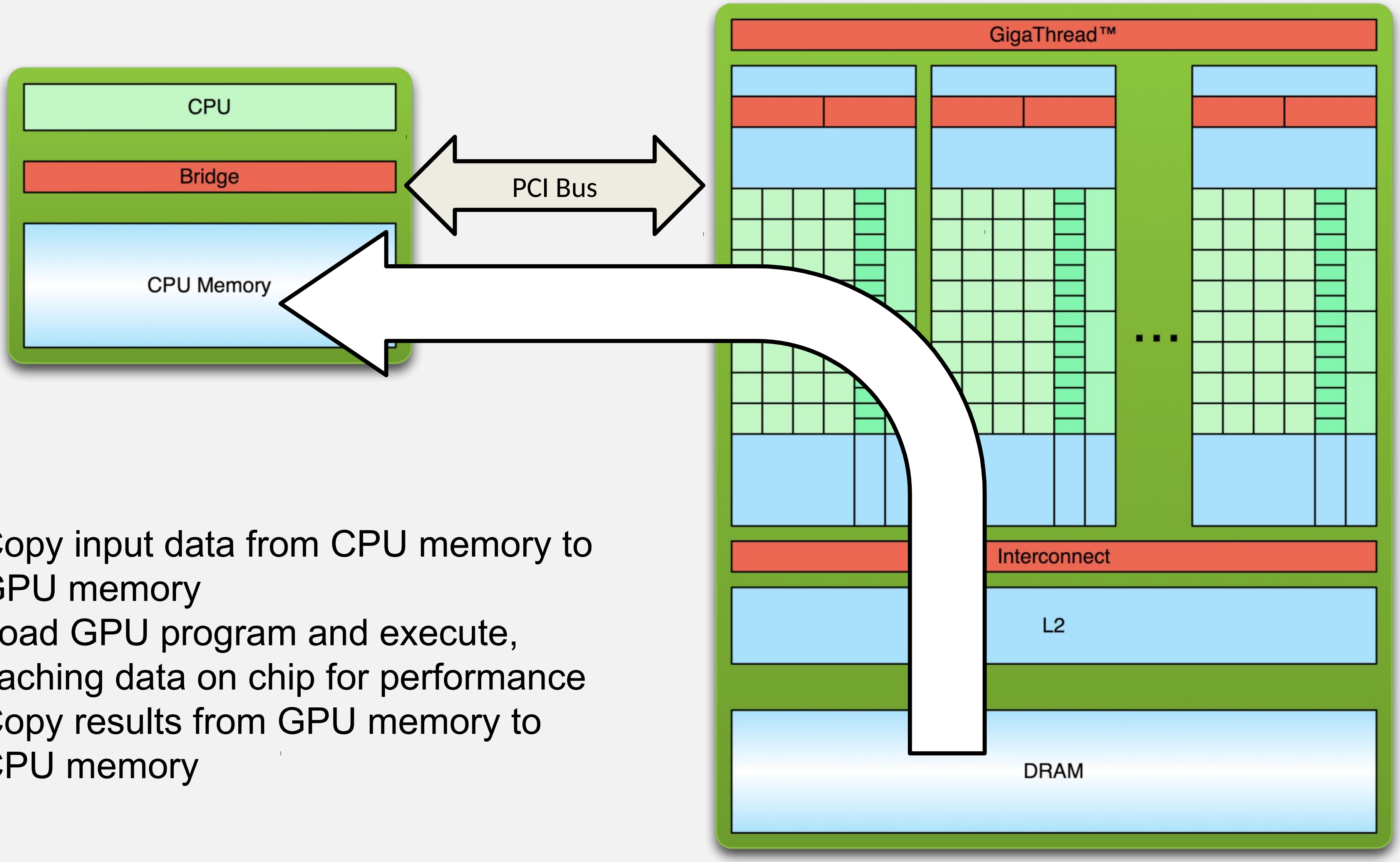
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

What is Thrust?

- High-Level Parallel Algorithms Library
- Parallel Analog of the C++ Standard Template Library (STL)
- Performance-Portable Abstraction Layer
- Productive way to program CUDA

Example

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

Another example

- Containers

  `host_vector`

  `device_vector`

- Memory Management
  - Allocation
  - Transfers

- Algorithm Selection
  - Location is implicit

```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host data to device memory
thrust::device_vector<int> d_vec = h_vec;

// write device values from the host
d_vec[0] = 27;
d_vec[1] = 13;

// read device values from the host
int sum = d_vec[0] + d_vec[1];

// invoke algorithm on device
thrust::sort(d_vec.begin(), d_vec.end());

// memory automatically released
```

Easy to Use

- Distributed with CUDA Toolkit
- Header-only library
- Architecture agnostic
- Just compile and run!

```
$ nvcc -O2 -arch=sm_35 program.cu -o program
```

Portability

- Support for CUDA, TBB and OpenMP
  - Just recompile!

```
nvcc -DTHRUST_DEVICE_SYSTEM=THRUST_HOST_SYSTEM_OMP
```

NVIDA GeForce GTX 580

```
$ time ./monte_carlo
pi is approximately 3.14159

real    0m6.190s
user    0m6.052s
sys 0m0.116s
```

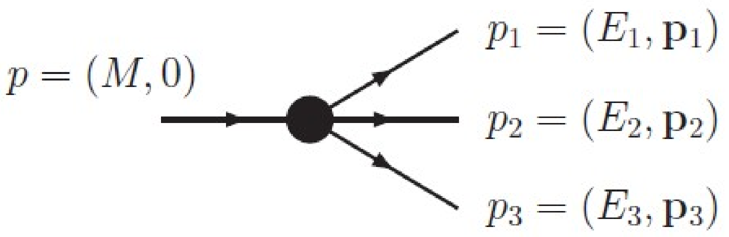
Intel Core i7 2600K

```
$ time ./monte_carlo
pi is approximately 3.14159

real    1m26.217s
user    11m28.383s
sys 0m0.020s
```

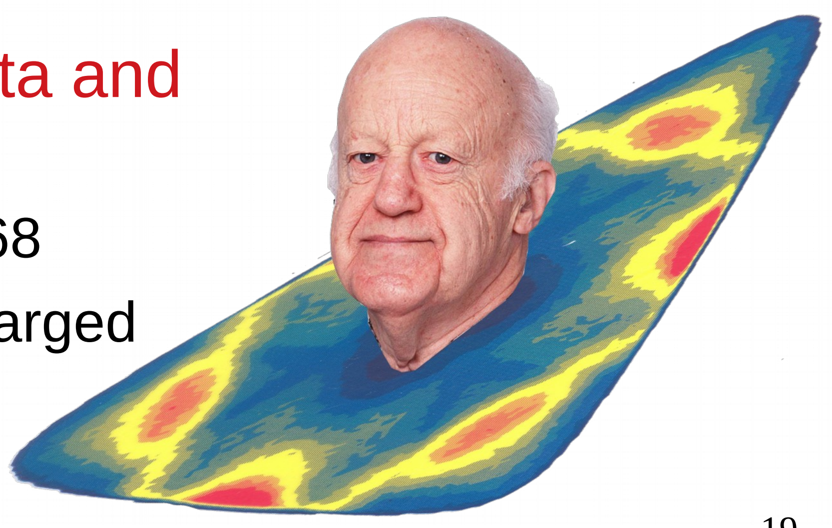
What is Dalitz-plot?

- Visual representation of the phase-space of a three-body decay: 0 → 123
  - Two independent Lorentz invariants:

$$m_{12}^2 + m_{13}^2 + m_{23}^2 = M^2 + m_1^2 + m_2^2 + m_3^2,$$

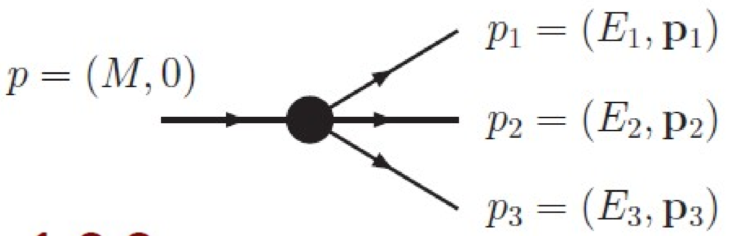


- Named after its inventor, Richard Dalitz (1925 – 2006)
  - "On the analysis of tau-meson data and the nature of the tau-meson."
    - R.H. Dalitz, Phil. Mag. 44 (1953) 1068
    - (historical reminder: tau meson = charged kaon)

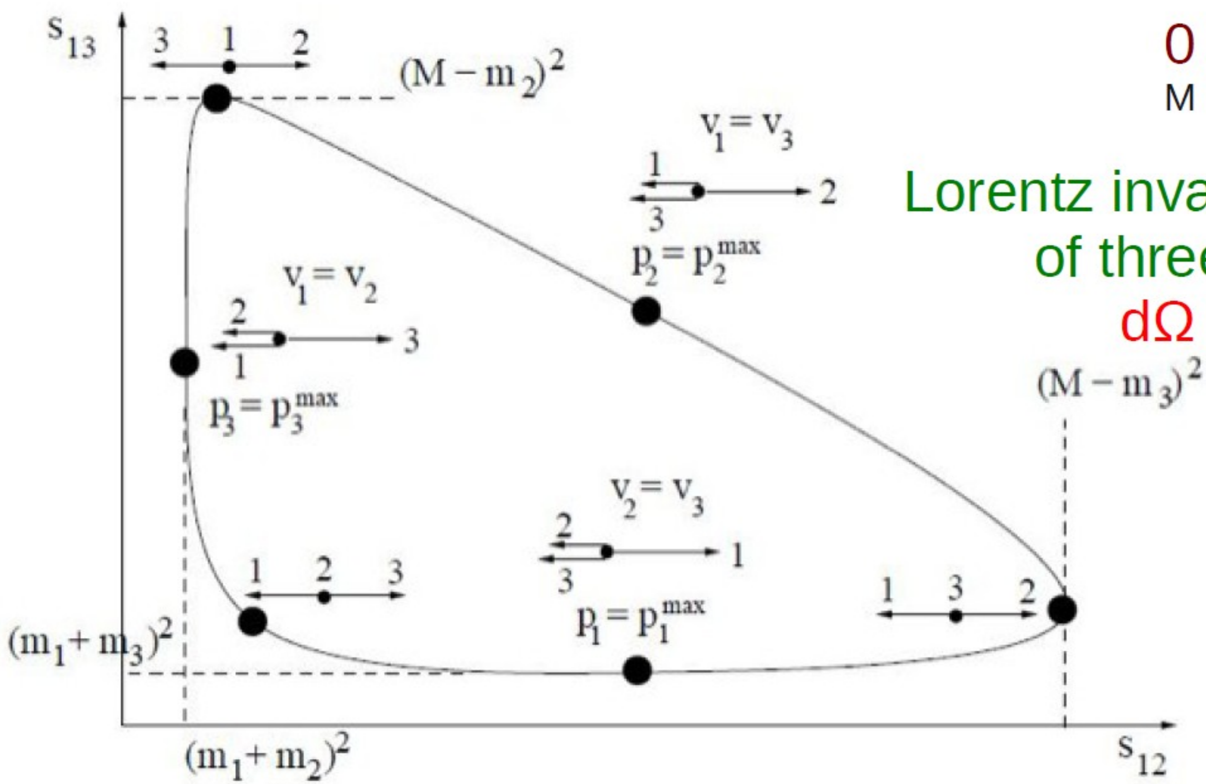


What is Dalitz-plot?

- Visual representation of the phase-space of a three-body decay: 0 → 123
  - Two independent Lorentz invariants:



$$m_{12}^2 + m_{13}^2 + m_{23}^2 = M^2 + m_1^2 + m_2^2 + m_3^2,$$



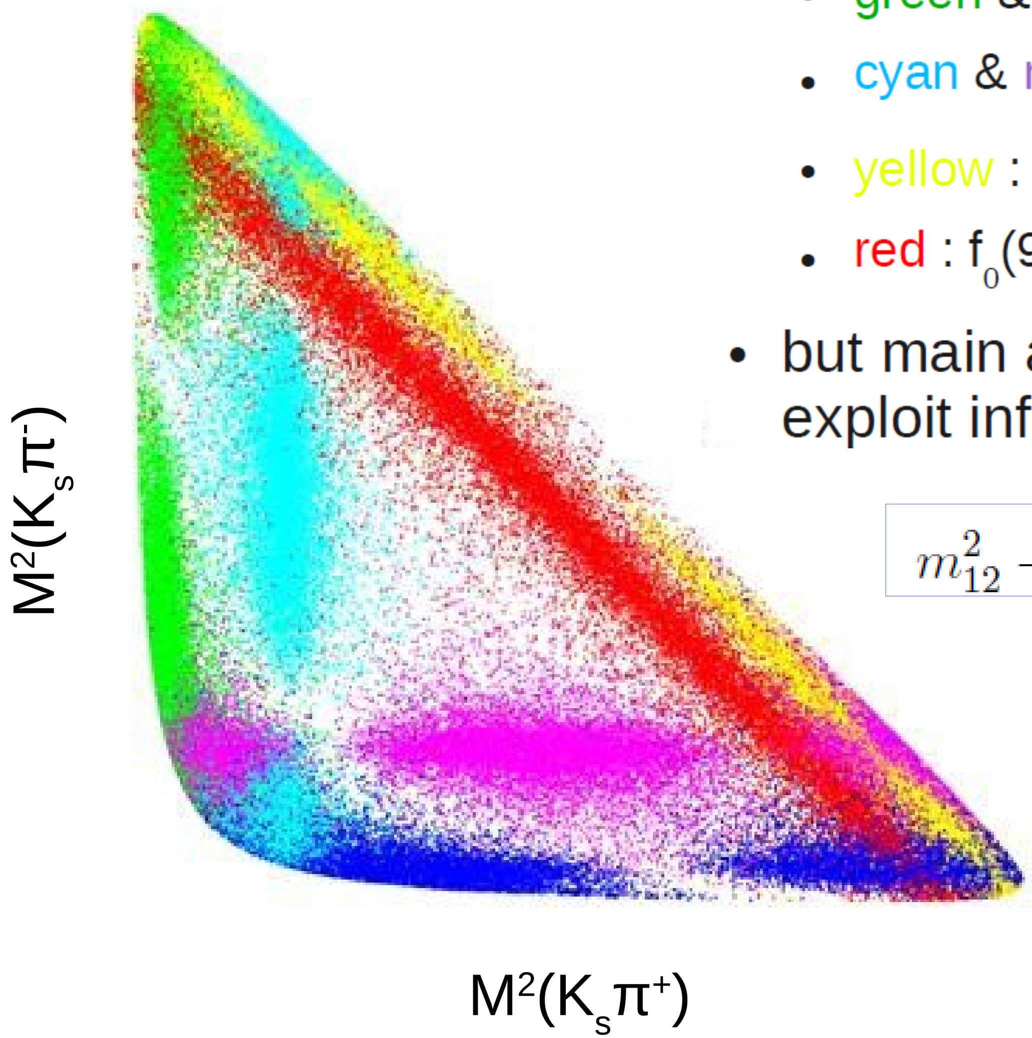
0 → 1 2 3

M m_1 m_2 m_3

Lorentz invariant phase-space of three-body decay

$d\Omega \sim ds_{12}ds_{13}$

Dalitz plots as visualizer of kinematics



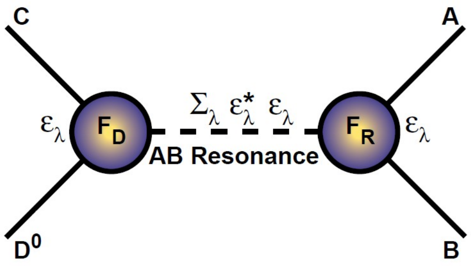
- Illustration for $D \to K_S \pi^+\pi^-$
  - green & blue: K*(892) (vector)
  - cyan & magenta : K_2*(1430) (tensor)
  - yellow : ρ(770) (vector)
  - red : f_0(980) (scalar)
- but main advantage of Dalitz plots is ability to exploit inference between different resonances

$$m_{12}^2 + m_{13}^2 + m_{23}^2 = M^2 + m_1^2 + m_2^2 + m_3^2,$$

Dalitz-plot analysis

- Amplitude analysis to extract directly information related to the phase at each Dalitz plot position
- Most commonly performed in the “isobar model”
  - Coherent sum of interfering quasi-two-body resonances: D → C R (→ AB):

$$\mathcal{A} = \sum_i c_i e^{i\phi_i} A_i + c_0 e^{i\phi_0}$$



  - Each described by Breit-Wigner (or similar) lineshapes, spin terms, etc.
  - Unbinned fit to determine lineshape parameters: inherent model dependence
- Alternative approaches aiming to avoid model dependence usually involve binning
  - Partial wave analysis

Overview of Amplitude analysis toolkits

AmpGen	github.com/GooFit/AmpGen/
cfit	github.com/cfit/cfit
GooFit	github.com/GooFit/GooFit
Hydra	github.com/MultithreadCorner/Hydra
Ipanema	gitlab.cern.ch/bsm-fleet/Ipanema/
Laura++	laura.hepforge.org
Mint2	github.com/jdalseno/Mint2
TFA	gitlab.cern.ch/poluekt/TensorFlowAnalysis
zFit	github.com/zfit/zfit

Overview of Amplitude analysis toolkits

	AmpGen	cFit	Craft	GooFit	Hydra	Ipanema	Laura++	Mint2	TFA	zFit
C++	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗
Python	✗	✗	✗	✓	✗	✓	✓	✗	✓	✓
GPU accelerated	✗	✗	✗	✓	✓	✓	✗	✗	✓	✓
$>$ 3-body	✓	✗	✗	✓		✓	✗	✓	✓	✓
s-dependent full width	✓	✗	✗	$\sim$		✗	✗	✓	✗	✗
Numerical dispersive mass	✓	✗	✗	✗		✗	✗	✓	✗	✗
Covariant spin	✓	✗	✗	✓		✗	✓	✓	$\sim$	✗
$S > 0$ initial/final state	✓	✗	✗	✗		✗	✗	✓	✓	WIP
$S >= \frac{1}{2}$ initial/final state	✓	✗	✗	✗		✗	✗	✗	✓	WIP
Photon polarisation	✓	✗	✗	✗		✗	✗	✓		✗
Simultaneous B/D mass fit	✗	✓	✓	✗		✗	✓	✗	✓	✓
Integral by MC	✓	✗	✗	✓		✓	✗	✓	✓	✓
Double Dalitz capable	✓	$\sim$	✗	✗		✗	WIP	✗	✓	✗
D Dalitz	✓	✓	✓	✓		✗	✓	✓	✓	✓
B Dalitz (SDP)	✗	✗	✓	✗		✗	✓	✗	✓	✗
B Amplitude (VV)	✓	✗	✗	✓		✓	✗	✓	✓	WIP
1D mass resolution	✗	✗	✗	✗		✗	✗	✗		
2D mass convolution map	✗	✗	✗	✗		✗	✓	✗		
Incoherent B_s^0 time	✗	✗	$\sim$	✗		✓	WIP	✓		
Coherent B^0 time	✗	✗	$\sim$	✗		✗	WIP	✗		
Missing energy	✗	✗	✗	✗		✗	✗	✗		

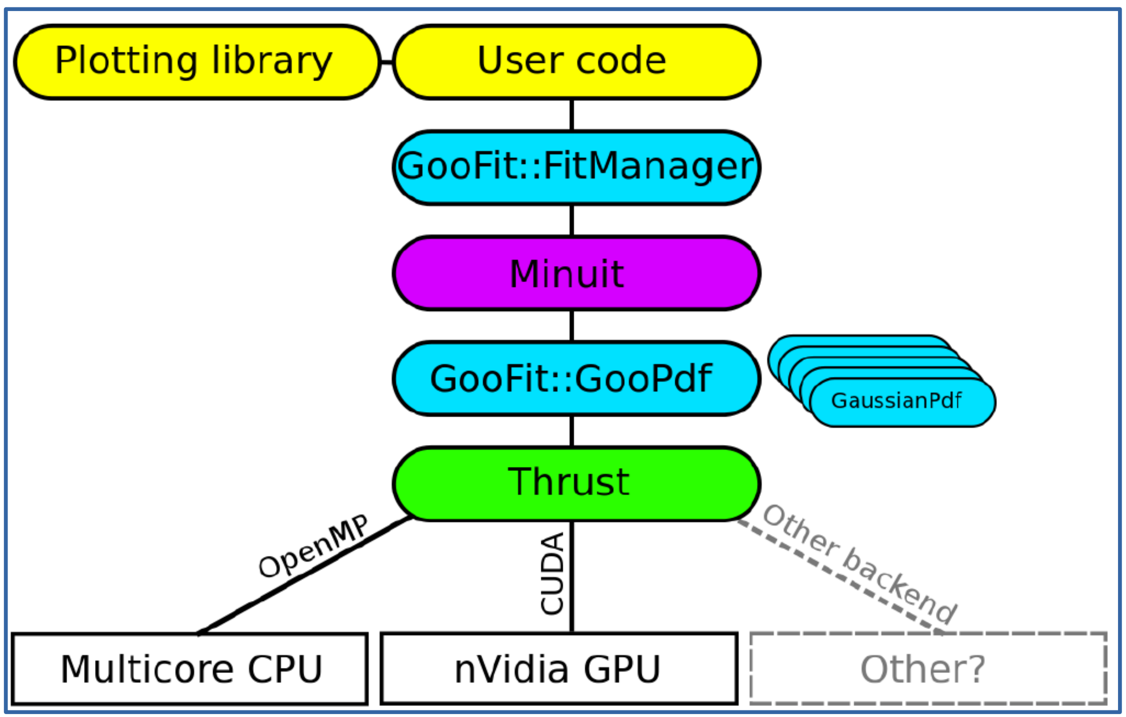
Table on features of different tools (source)

Disclaimer: not a complete list, GPU based fitters used in BESIII collaborator are not listed

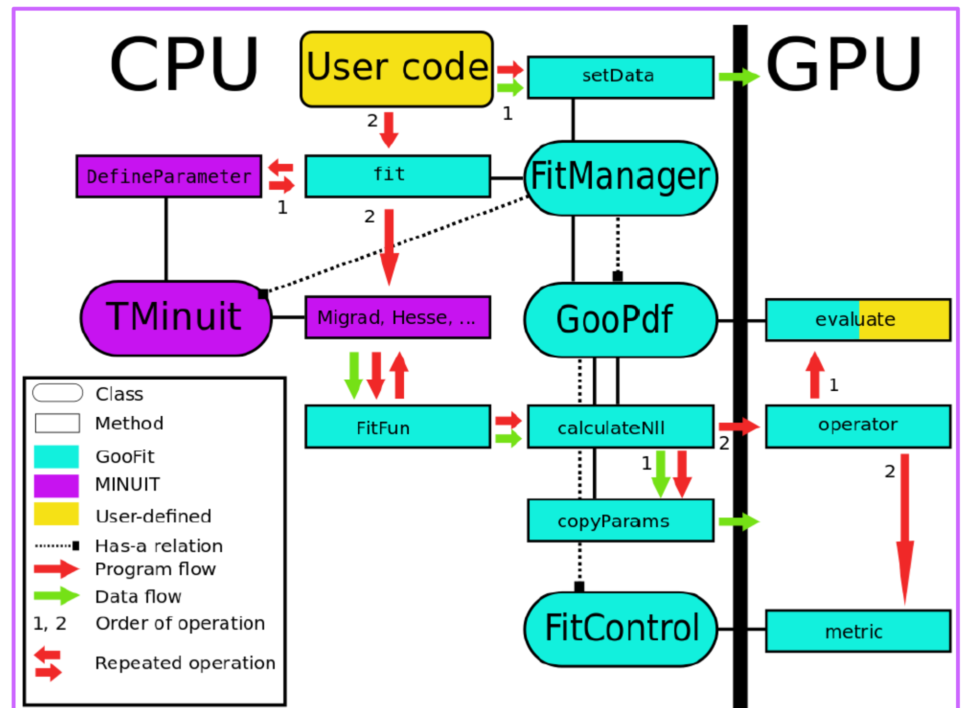
GooFit (v1) introduction

- GooFit: an open-source project originally developed by R. Andreassen and funded by NSF
  - **FitManager** object as the interface between MINUIT and a GPU which allows a PDF (**GooPDF** object) to be evaluated in parallel

Architecture:



Program flow:



"Implementation of a Thread-Parallel, GPU-Friendly Function Evaluation Library," R. Andreassen et al., IEEE Access, v.2, 2014.

Analogy with RooFit

- Code structure similar to RooFit framework, the overall fit set-up and running are familiar for RooFit users

RooFit	GooFit
RooRealVar	Variable
RooDataSet	UnbinnedDataSet
RooDataHist	BinnedDataSet
RooAbsPdf	GooPdf
RooGaussian	GaussianPdf
RooArgSet	`vector<Variable*>`
RooPlot	None! Use ROOT plotting.
`myPdf->plotOn(foo)`	`myPdf->getCompProbsAtDataPoints(points)`
RooAbsTestStatistic	FitControl

GooFit PDFs

- Simple PDFs: ARGUS, correlated Gaussian, Crystal Ball, exponential, Gaussian, Johnson SU, polynomial, relativistic Breit-Wigner, scaled Gaussian, smoothed histogram, step function, Voigtian

- Composites:
  - Sum, $f_1 A(\vec{x}) + (1 - f_1) B(\vec{x})$.
  - Product, $A(\vec{x}) \times B(\vec{x})$.
  - Composition, $A(B(x))$ (only one dimension).
  - Convolution, $\int_{t_1}^{t_2} A(x - t) * B(t) \mathrm{d}t$.
  - Map,

$$F(x) = \begin{cases} A(x) & \text{if } x \in [x_0, x_1) \\ B(x) & \text{if } x \in [x_1, x_2) \\ \ldots & \\ Z(x) & \text{if } x \in [x_{N-1}, x_N] \end{cases}$$

You can write your own PDFs based on the example PDF code with relative ease

- Specialized mixing PDFs: Coherent amplitude sum, incoherent sum, truth resolution, three-Gaussian resolution, Dalitz-plot region veto, threshold damping function
  - **TddpPdf** (**DalitzPlotPdf**) as the main engine for time-dependent (-integrated) Dalitz-plot (**DP**) fits, with a list of **ResonancePdf** objects as input to describe different (non-)resonance amplitudes

Gaussian PDF as an example

```
__device__ fptype device_Gaussian (fptype* evt,
                                   fptype* p,
                                   unsigned int* indices) {
  fptype x = evt[indices[2 + indices[0]]];
  fptype mean = p[indices[1]];
  fptype sigma = p[indices[2]];

  fptype ret = EXP(-0.5*(x-mean)*(x-mean)/(sigma*sigma));
  return ret;
}

__device__ device_function_ptr ptr_to_Gaussian = device_Gaussian;
```

Side note: ‘**fptype**’ is just a typedef for **double** - this allows quick switching between double and float precision

Time-dependent amplitude analysis of $D^0 \to \pi\pi\pi^0$

- **First** published physics analysis using GooFit
- Measurement on D^0 mixing parameters x and y using an unbinned maximum likelihood fit
- A total of **138k** data events from BABAR experiment
- Final results:

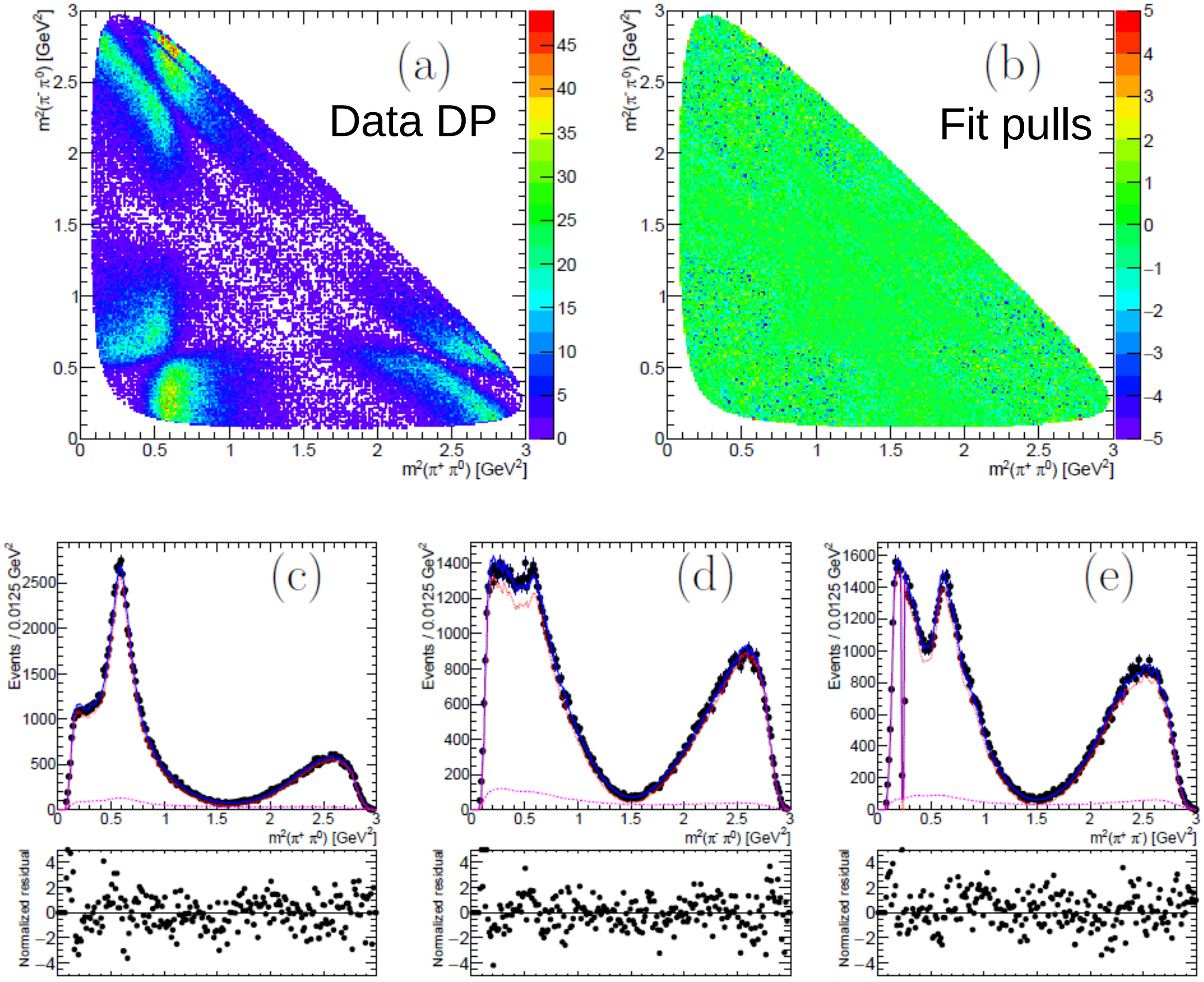
$$\boxed{\begin{aligned} x &= (1.50 \pm 1.17[\text{stat}] \pm 0.56[\text{syst}])\ \% \\ y &= (0.19 \pm 0.89[\text{stat}] \pm 0.46[\text{syst}])\ \% \end{aligned}}$$

PRD **93**, 112014 (2016)

Signal PDF (**TddpPdf**): Time-dependent mixing function involving coherent sums of Breit-Wigner amplitudes (**ResonancePdf**)

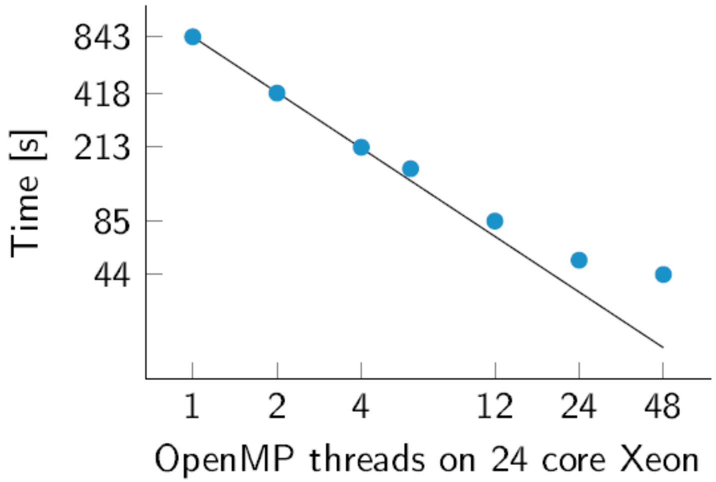
State	Resonance parameters		
	J^{PC}	Mass (MeV/c^2)	Width (MeV/c^2)
$\rho(770)^+$	1^{--}	775.8	150.3
$\rho(770)^0$	1^{--}	775.8	150.3
$\rho(770)^-$	1^{--}	775.8	150.3
$\rho(1450)^+$	1^{--}	1465	400
$\rho(1450)^0$	1^{--}	1465	400
$\rho(1450)^-$	1^{--}	1465	400
$\rho(1700)^+$	1^{--}	1720	250
$\rho(1700)^0$	1^{--}	1720	250
$\rho(1700)^-$	1^{--}	1720	250
$f_0(980)$	0^{++}	980	44
$f_0(1370)$	0^{++}	1434	173
$f_0(1500)$	0^{++}	1507	109
$f_0(1710)$	0^{++}	1714	140
$f_2(1270)$	2^{++}	1275.4	185.1
$f_0(500)$	0^{++}	500	400
NR			

Fit projections



The data fit takes ~1 min to complete with Nvidia Tesla K40c, a speed-up of ~300 over the original CPU version

GooFit Performance [from ACAT 2017]

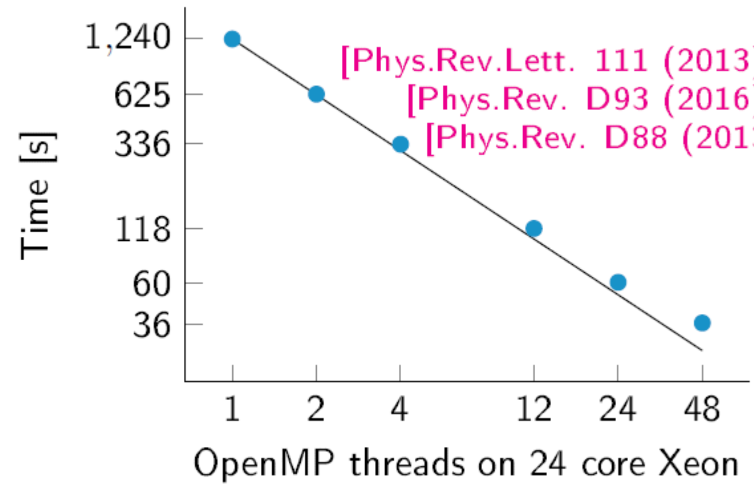


$\pi\pi\pi^0$, 16 time-dependent amplitudes

- Original RooFit code: 19,489 s single core

2 Cores	Core 2 Duo	1,159 s
GPU	GeForce GTX 1050 Ti	86.4 s
GPU	Tesla K40	64.0 s
MPI	Tesla K40 ×2	39.3 s
GPU	Tesla P100	20.3 s

GooFit Introduction



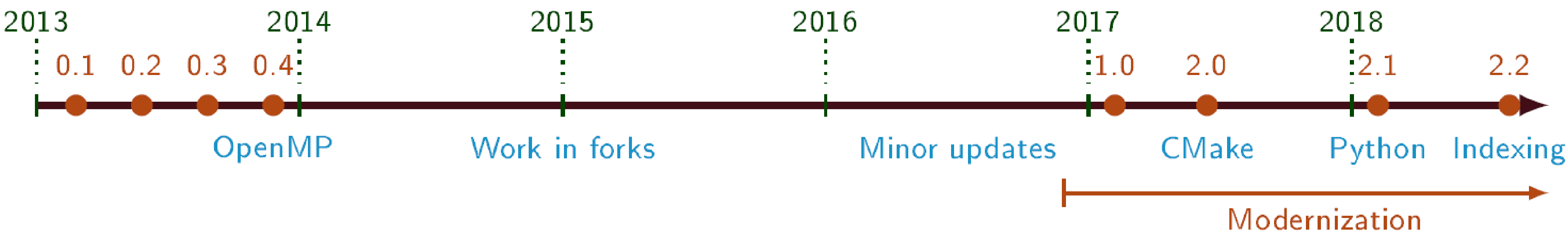
[Phys.Rev.Lett. 111 (2013) no.11, 111801]
[Phys.Rev. D93 (2016) no.11, 112014]
[Phys.Rev. D88 (2013) no.5, 052003]
[CHEP 2013]

ZachFit: $M(D^{*+}) - M(D^0)$

- 142,576 events in unbinned fit

2 Cores	Core 2 Duo	738 s
GPU	GeForce GTX 1050 Ti	60.3 s
GPU	Tesla K40	96.6 s
MPI	Tesla K40 ×2	54.3 s
GPU	Tesla P100	23.5 s

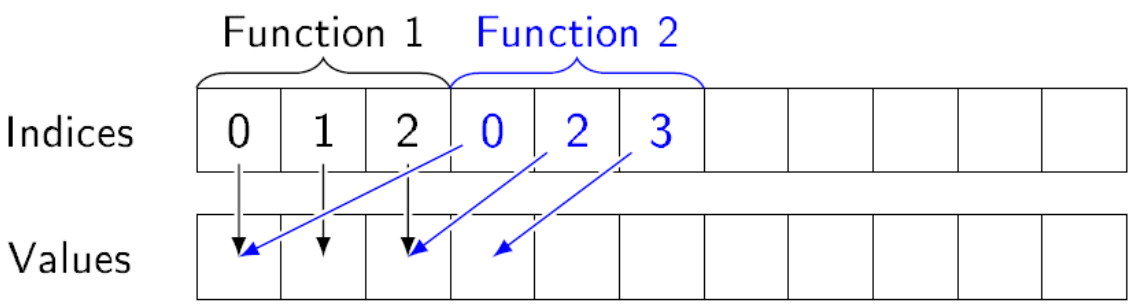
Recent GooFit developments



Recent History

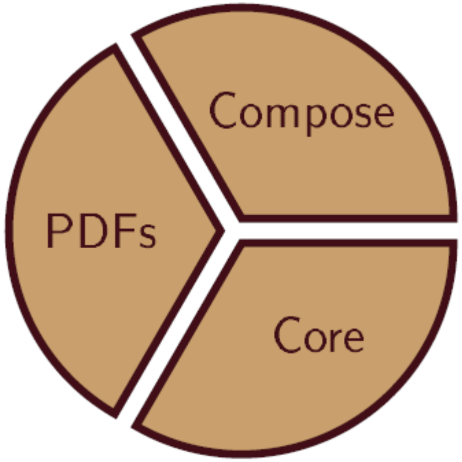
- 2.0: New build system, C++11, and 4-body time dependent analyses support
- 2.1: Python bindings using Pybind11
- 2.2: New indexing (and lots of Python improvements)

How GooFit v2 works?



How GooFit Works

- CPU classes: Variable, Observable, DataSet, GooPdf
- Functions in CUDA with pointers held by GooPdf
- Function and variable arrays populated by GooFit
- Evaluation runs through CUDA functions through pointers (one kernel)
- Launching is handled by Thrust



Code examples

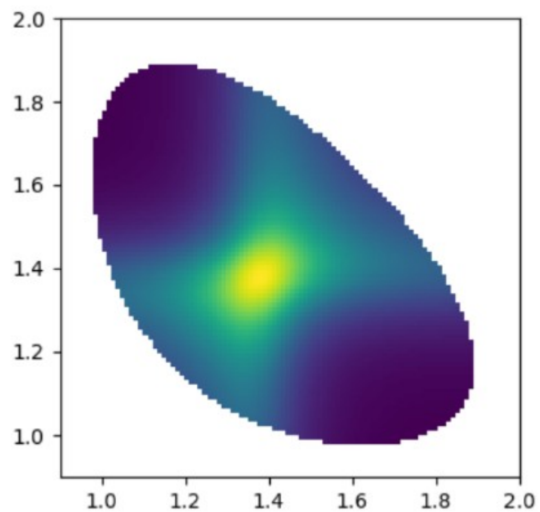
```cpp
#include <goofit/...>
using namespace GooFit;

Observable x{"x", 0, 10};
Variable mu{"mu", 1};
Variable sigma{"sigma", 1, 0, 10};
GaussianPdf gauss{"gauss", &x, &mu, &sigma};
UnbinnedDataSet ds{x};

std::mt19937 gen;
std::normal_distribution<double> d{1, 2.5};
for(size_t i=0; i<100000; i++)
    ds.addEvent(d(gen));

gauss.fitTo(&ds);

std::cout << mu << std::endl;
```

```python
from goofit import *
import numpy as np

x = Observable("x", 0, 10)
mu = Variable("mu", 1)
sigma = Variable("sigma", 1, 0, 10)
gauss = GaussianPdf("gauss", x, mu, sigma)
ds = UnbinnedDataSet(x)

data = np.random.normal(1, 2.5, (100000,1))
ds.from_matrix(data, filter=True)


gauss.fitTo(ds)

print(mu)
```

Evaluation Example

In [13]:
```python
fig, ax = plt.subplots(figsize=(6, 4))
v = ax.imshow(arr, extent=ext, origin='lower')
```

Figure 1



In [16]:
```python
@interact
def f(a=r_pi, b=r_pi, m1=r_mass, w1=r_width, m2=r_mass, w2=r_width):
    ar1.value, ai1.value = np.cos(a), np.sin(a)
    ar2.value, ai2.value = np.cos(b), np.sin(b)
    mass1.value, width1.value = m1, w1
    mass2.value, width2.value = m2, w2
    dplt = gf.DalitzPlotter(prod, dp)
    arr = dplt.make2D()
    arr = np.ma.array(arr, mask=arr==0)
    ax.clear()
    v = ax.imshow(arr, extent=ext, origin='lower')
```

a 1.57
b 1.57
m1 1.18
w1 0.11
m2 1.18
w2 0.11

GooFit (python) documentation

Documentation

- Documentation exported to Jupyter

Implementation details

- Generated by CMake from Doxygen style comments
  - Conversion to Jupyter style markdown for math
- Attached to class in PyBind11

In [2]:
```python
import goofit
```

In [3]:
```python
goofit.ExpGausPdf
```

Out[3]: An exponential decay convolved with a Gaussian resolution:

$$P(t; m, \sigma, \tau) = e^{-t/\tau} \otimes e^{-\frac{(t-m)^2}{2\sigma^2}}$$
$$= (\tau/2)\, e^{(\tau/2)(2m+\tau\sigma^2-2t)} \,\mathrm{erfc}\left(\frac{m+\tau\sigma^2-t}{\sigma\sqrt{2}}\right)$$

where erfc is the complementary error function. The constructor takes the observed time t, mean m and width σ of the resolution, and lifetime τ. Note that the original decay function is zero for $t < 0$.

What is Hydra?

Hydra is a header-only, templated C++11 framework designed to perform common tasks found in HEP data analyses on massively parallel platforms.

- It is implemented on top of the C++11 Standard Library and a variadic version of the Thrust library.
- Hydra is designed to run on Linux systems and to deploy parallelism using
  - OpenMP. Directive-based implementation of multithreading.
  - TBB (Threading Building Blocks). C++ template library developed by Intel for parallel programming on multi-core processors.
  - CUDA. Parallel computing platform and application programming interface (API) model created by Nvidia for compatible GPUs.
- It is focused on portability, usability, performance and precision.

HYDRA
Multithreaded Data
Analysis Framework

Hydra features

- Interface to `ROOT::Minuit2` minimization package, to perform binned and unbinned multidimensional fits.
- Parallel calculation of S-Plots.
- Phase-space generator and integrator.
- Multidimensional p.d.f. sampling.
- Parallel function evaluation over multidimensional data-sets.
- Numerical integration: plain and VEGAS Monte Carlo, Gauss-Kronrod and Genz-Malik quadratures.
- Dense and sparse multidimensional histograming.
- Support to C++11 "parametric lambdas" for fits, filters, smart-ranges,... etc

All the algorithms can be invoked concurrently and asynchronously, mixing different back-ends.

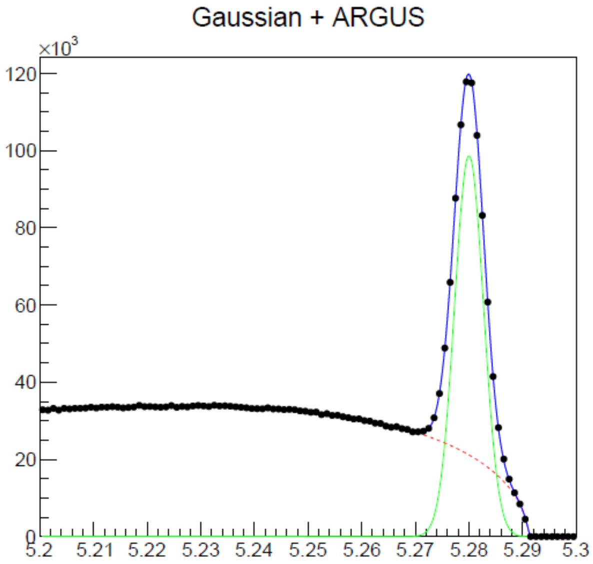
Hydra example I: Gaussian + Argus

```
//Analysis range
double min =  5.20, max = 5.30;

//Gaussian: parameters definition
hydra::Parameter  mean  = hydra::Parameter::Create().Name("Mean").Value( 5.28).Error(0.0001).Limits(5.27,5.29);
hydra::Parameter  sigma = hydra::Parameter::Create().Name("Sigma").Value(0.0027).Error(0.0001).Limits(0.0025,0.0029);
//Gaussian: PDF definition using analytical integration
auto Signal_PDF = hydra::make_pdf( hydra::Gaussian<>(mean, sigma),
          hydra::GaussianAnalyticalIntegral(min, max));

//Argus: parameters definition
auto  m0     = hydra::Parameter::Create().Name("M0").Value(5.291).Error(0.0001).Limits(5.28, 5.3);
auto  slope  = hydra::Parameter::Create().Name("Slope").Value(-20.0).Error(0.0001).Limits(-50.0, -1.0);
auto  power  = hydra::Parameter::Create().Name("Power").Value(0.5).Fixed();
//Argus: PDF definition using analytical integration
auto Background_PDF = hydra::make_pdf( hydra::ArgusShape<>(m0, slope, power),
          hydra::ArgusShapeAnalyticalIntegral(min, max));

//Signal and Background yields
hydra::Parameter N_Signal("N_Signal"        ,500, 100, 100 , nentries) ;
hydra::Parameter N_Background("N_Background",2000, 100, 100 , nentries) ;

//Make model
auto Model = hydra::add_pdfs( {N_Signal, N_Background}, Signal_PDF, Background_PDF);
```

Hydra example I: Gaussian + Argus

```
//1D device buffer
hydra::device::vector<double>  data(nentries);

//Generate data
auto data_range = Generator.Sample(data.begin(),  data.end(), min, max, model.GetFunctor());

//Make model and fcn
auto fcn = hydra::make_loglikehood_fcn( model, range.begin(), range.end() );

//Fitting using ROOT::Minuit2
//minimization strategy
MnStrategy strategy(2);

//create Migrad minimizer
MnMigrad migrad_d(fcn, fcn.GetParameters().GetMnState() ,  strategy);

//minimization
FunctionMinimum minimum_d =  FunctionMinimum(migrad_d(5000, 5));
```

Hydra example I: Gaussian + Argus



Unbinned fit with 1,949,204 events.

- FCN calls: 789
- Intel® Core™ i7-4790 CPU @ 3.60 GHz (1 thread):146,531 s
- Intel® Core™ i7-4790 CPU @ 3.60 GHz (8 threads):26,875 s
- NVidia TitanZ GPU: 3,75 s

Hydra example II: $D^+ \to K^-\pi^+\pi^+$ amplitudes

PHYSICAL REVIEW D 78, 052001 (2008)

Mode	Parameter	E791	CLEO-c
NR	a	$1.03 \pm 0.30 \pm 0.16$	$7.4 \pm 0.1 \pm 0.6$
	$\phi(^\circ)$	$-11 \pm 14 \pm 8$	$-18.4 \pm 0.5 \pm 8.0$
	FF (%)	$13.0 \pm 5.8 \pm 4.4$	$8.9 \pm 0.3 \pm 1.4$
$\bar{K}^*(892)\pi^+$	a	1 (fixed)	1 (fixed)
	$\phi(^\circ)$	0 (fixed)	0 (fixed)
	FF (%)	$12.3 \pm 1.0 \pm 0.9$	$11.2 \pm 0.2 \pm 2.0$
$\bar{K}_0^*(1430)\pi^+$	a	$1.01 \pm 0.10 \pm 0.08$	$3.00 \pm 0.06 \pm 0.14$
	$\phi(^\circ)$	$48 \pm 7 \pm 10$	$49.7 \pm 0.5 \pm 2.9$
	FF (%)	$12.5 \pm 1.4 \pm 0.5$	$10.4 \pm 0.6 \pm 0.5$
	m (MeV/c^2)	$1459 \pm 7 \pm 12$	$1463.0 \pm 0.7 \pm 2.4$
	Γ (MeV/c^2)	$175 \pm 12 \pm 12$	$163.8 \pm 2.7 \pm 3.1$
$\bar{K}_2^*(1430)\pi^+$	a	$0.20 \pm 0.05 \pm 0.04$	$0.962 \pm 0.026 \pm 0.050$
	$\phi(^\circ)$	$-54 \pm 8 \pm 7$	$-29.9 \pm 2.5 \pm 2.8$
	FF (%)	$0.5 \pm 0.1 \pm 0.2$	$0.38 \pm 0.02 \pm 0.03$
$\bar{K}^*(1680)\pi^+$	a	$0.45 \pm 0.16 \pm 0.02$	$6.5 \pm 0.1 \pm 1.5$
	$\phi(^\circ)$	$28 \pm 13 \pm 15$	$29.0 \pm 0.7 \pm 4.6$
	FF (%)	$2.5 \pm 0.7 \pm 0.3$	$1.28 \pm 0.04 \pm 0.28$
$\kappa\pi^+$	a	$1.97 \pm 0.35 \pm 0.11$	$5.01 \pm 0.04 \pm 0.27$
	$\phi(^\circ)$	$-173 \pm 8 \pm 18$	$-163.7 \pm 0.4 \pm 5.8$
	FF (%)	$47.8 \pm 12.1 \pm 5.3$	$33.2 \pm 0.4 \pm 2.4$
	m (MeV/c^2)	$797 \pm 19 \pm 43$	$809 \pm 1 \pm 13$
	Γ (MeV/c^2)	$410 \pm 43 \pm 87$	$470 \pm 9 \pm 15$

- Masses and widths from PDG-2017.
- Phases and magnitudes from paper above(see page 12, table 7).

Hydra example II: $D^+ \rightarrow K^-\pi^+\pi^+$ amplitudes

Defining a contribution:

```
//K*(892)
//parameters
auto mass = hydra::Parameter::Create().Name("MASS_KST_892" ).Value(KST_892_MASS )
                                    .Error(0.0001).Limits(KST_892_MASS*0.95,  KST_892_MASS*1.05 );

auto width = hydra::Parameter::Create().Name("WIDTH_KST_892").Value(KST_892_WIDTH)
                                    .Error(0.0001).Limits(KST_892_WIDTH*0.95, KST_892_WIDTH*1.05);

auto coef_re = hydra::Parameter::Create().Name("A_RE_KST_892" ).Value(KST_892_CRe)
                                    .Error(0.001).Limits(KST_892_CRe*0.95,KST_892_CRe*1.05).Fixed();

auto coef_im = hydra::Parameter::Create().Name("A_IM_KST_892" ).Value(KST_892_CIm)
                                    .Error(0.001).Limits(KST_892_CIm*0.95,KST_892_CIm*1.05).Fixed();
//contributions per channel
Resonance<1, hydra::PWave> KST_892_Resonance_12(coef_re, coef_im, mass, width, D_MASS, K_MASS, PI_MASS, PI_MASS , 5.0);

Resonance<3, hydra::PWave> KST_892_Resonance_13(coef_re, coef_im, mass, width, D_MASS, K_MASS, PI_MASS, PI_MASS , 5.0);

//total contribution
auto KST_892_Resonance = (KST_892_Resonance_12 - KST_892_Resonance_13);
```

The other resonances are defined in a similar way.

Hydra example II: $D^+ \rightarrow K^- \pi^+ \pi^+$ amplitudes

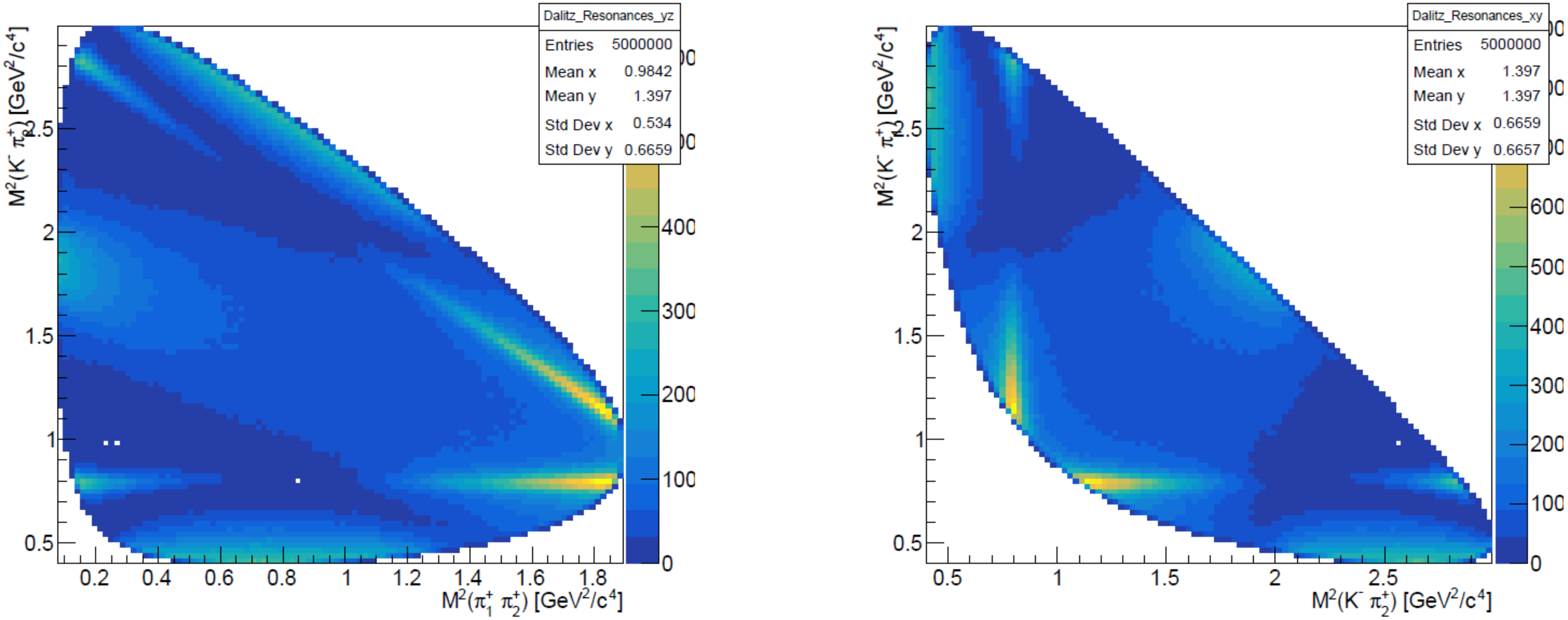
Now the fit model:

```
//NR
coef_re = hydra::Parameter::Create().Name("A_RE_NR" ).Value(NR_CRe).Error(0.001).Limits(NR_CRe*0.95,NR_CRe*1.05);
coef_im = hydra::Parameter::Create().Name("A_IM_NR" ).Value(NR_CIm).Error(0.001).Limits(NR_CIm*0.95,NR_CIm*1.05);

auto NR = NonResonant(coef_re, coef_im);

//Total model |N.R + \sum{ Resonaces }|^2
auto Norm = hydra::wrap_lambda(
     []__host__  __device__ (unsigned int n, hydra::complex<double>* x) {
            hydra::complex<double> r(0,0);
            for(unsigned int i=0; i< n;i++) r += x[i];
            return hydra::norm(r);}
     );

//Functor
auto Model = hydra::compose(Norm, K800_Resonance, KST_892_Resonance,
              KST0_1430_Resonance, KST2_1430_Resonance, KST_1680_Resonance, NR);

//PDF
auto Model_PDF = hydra::make_pdf( Model,
                hydra::PhaseSpaceIntegrator<3, hydra::device::sys_t>(D_MASS, {K_MASS, PI_MASS, PI_MASS}, 500000));
```

```
...
//get the fcn
auto fcn = hydra::make_loglikehood_fcn(Model_PDF, particles.begin(), particles.end());
//minimization strategy
MnStrategy strategy(2);
//create Migrad minimizer
MnMigrad migrad_d(fcn, fcn.GetParameters().GetMnState() ,  strategy);
//fit...
FunctionMinimum minimum_d =  FunctionMinimum( migrad_d(5000, 5) );
```

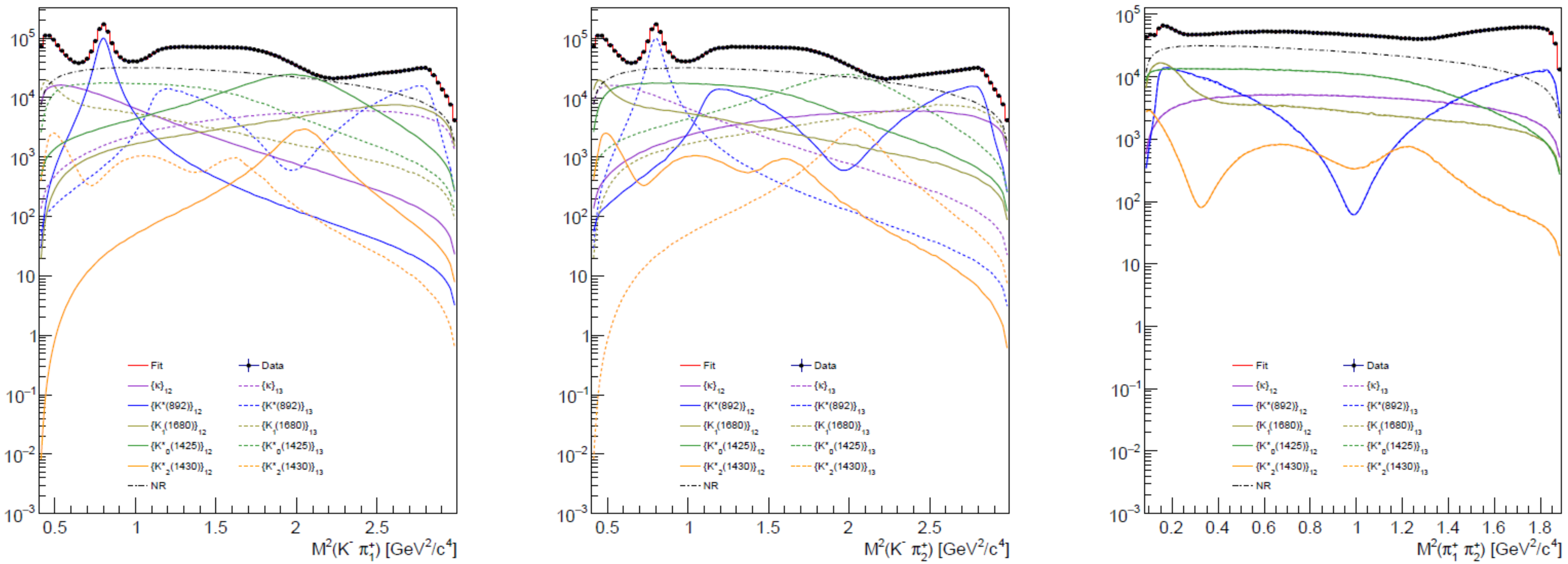
Hydra example II: $D^+ \rightarrow K^-\pi^+\pi^+$ amplitudes

Toy data (5,000,000 events)



Hydra example II: $D^+ \to K^-\pi^+\pi^+$ amplitudes

Fit projections:



- Resonances identified by color.
- Solid lines for $K\pi_1$-channel.
- Dashed lines for $K\pi_2$-channel.
- Lines are superposed in $\pi_1\pi_2$-channel.

Hydra example II: $D^+ \to K^-\pi^+\pi^+$ amplitudes

Performance: CPU with CUDA

Parallel system	Time (s/min)	FCN Calls	Time/Call (s)
GeForce GTX Tesla P100	221.114 (3.68)	"	0.21
GeForce GTX Titan Z (GPU 1)	336.672 (5.61)	"	0.33
GeForce GTX 1050 Ti	729.165 (12,15)	"	0.71
GeForce GTX 970M (video)	744.247 (12,40)	"	0.72

Hydra example II: $D^+ \rightarrow K^- \pi^+ \pi^+$ amplitudes

Performance: CPU with OpenMP

Parallel system	Threads	Time (sec/min)	FCN Calls	Time/Call (sec)
i7-4790 CPU @ 3.60GHz	1	5060,578 (1.4 hours)	1030	4.91
	8	750.245 (12.50)	"	0.73
Xeon(R) CPU E5-2680 v3 @ 2.50GHz	1	5128.480 (1,42 hours)	"	4.98
	8	784.252 (13.1)	"	0.76
	12	612.278 (10.2)	"	0.59
	24	371.838 (6.2)	"	0.36
	48	247.787 (4.1)	"	0.24

Hydra example II: $D^+ \to K^- \pi^+ \pi^+$ amplitudes

Performance: CPU with TBB

Parallel system	Threads	Time (s/min)	FCN Calls	Time/Call (s)
i7-4790 CPU @ 3.60GHz	8	746.684 (12.4)	1030	0.72
Xeon(R) CPU E5-2680 v3 @ 2.50GHz	48	184.779 (3.01)	"	0.18

Summary & resources

- GooFit and Hydra are two example tools which could make your fitting hundreds of times faster
- A number of physics analyses have benefited from GPU acceleration
- Growing interests in GPUs within HEP community
- Useful resources:
  - CUDA
    - https://docs.nvidia.com/cuda/index.html
  - Thrust:
    - https://docs.nvidia.com/cuda/thrust/index.html
  - GooFit:
    - https://github.com/GooFit/GooFit
  - Hydra:
    - https://github.com/MultithreadCorner/Hydra