

What is it?

Hydra is a header only templated C++ library designed to perform common 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 runs on Linux systems and uses OpenMP, CUDA and TBB back-ends.
- ▶ It is focused on portability, usability, performance and precision.

Design and features

- ▶ The library is structured using static polymorphism and the interfaces have a clean and concise semantics
- ▶ There is absolutely no need to write explicit back-end oriented code.
- ▶ All supported back-ends can run concurrently in the same program using the suitable policies: `hydra::omp::sys`, `hydra::cuda::sys`, `hydra::tbb::sys`, `hydra::cpp::sys`, `hydra::host::sys` and `hydra::device::sys`

The same source files written using Hydra and standard C++ compile for GPU, CPU or even both, just changing the extension from `.cu` to `.cpp` and one or two compiler flags.

Functionality

- ▶ Interface to ROOT::Minuit2 minimization package.
- ▶ Phase-space generation and integration.
- ▶ Multidimensional p.d.f. sampling.
- ▶ Parallel function evaluation on multidimensional datasets.
- ▶ Numerical integration: Monte Carlo and quadrature based methods.

Functors and containers

- ▶ Hydra adds features and type information to generic functors using the CRTP idiom:

```
1 struct MyFunctor:
2     public hydra::BaseFunctor<MyFunctor,double,N>{
3     ...
4     // implement the Evaluate() method
5     template<typename T> __host__ __device__
6     inline double Evaluate(unsigned int n, T* x)
7     { /*actual calculation*/ }
8     };
```

- ▶ The basic arithmetic operators are overloaded. Composition of functors is also possible. If A, B and C are Hydra functors, the code below is completely legal:

```
1 ...
2 //basic arithmetic operations
3 auto A_plus_B = A + B;
4 auto A_minus_B = A - B;
5 auto A_times_B = A * B;
6 auto A_per_B = A/B;
7 //any composition of basic operations
8 auto any_functor = (A - B)*(A + B)/(A/C);
9 // C(A,B) is represented by:
10 auto compose_functor = hydra::compose(C, A, B)
11 ...
```

- ▶ The user can define a C++11 lambda function and convert it into a Hydra functor using `hydra::wrap_lambda()`:

```
1 ...
2 double two = 2.0;
3 //define a simple lambda and capture "two"
4 auto my_lambda = [=] __host__ __device__
5     (unsigned int n, double* x){
6     return two*sin(x[0]); };
7 //convert is into a Hydra functor
8 auto my_lambda_wrapped = hydra::wrap_lambda(my_lambda);
9 ...
```

- ▶ Set of dedicated STL-like containers optimized to store multidimensional data using structure of arrays layout.

Examples and performance

System configuration:

- ▶ GPU model: Tesla K40c
- ▶ CPU: Intel Xeon(R) CPU E5-2680 v3 @ 2.50GHz (one thread)

Vegas multidimensional numerical integration

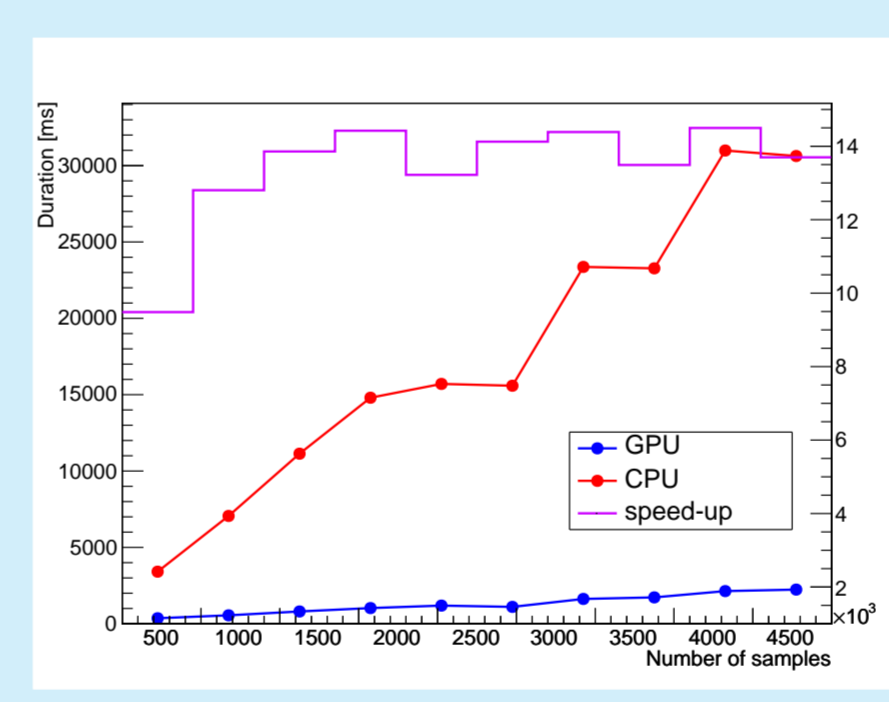
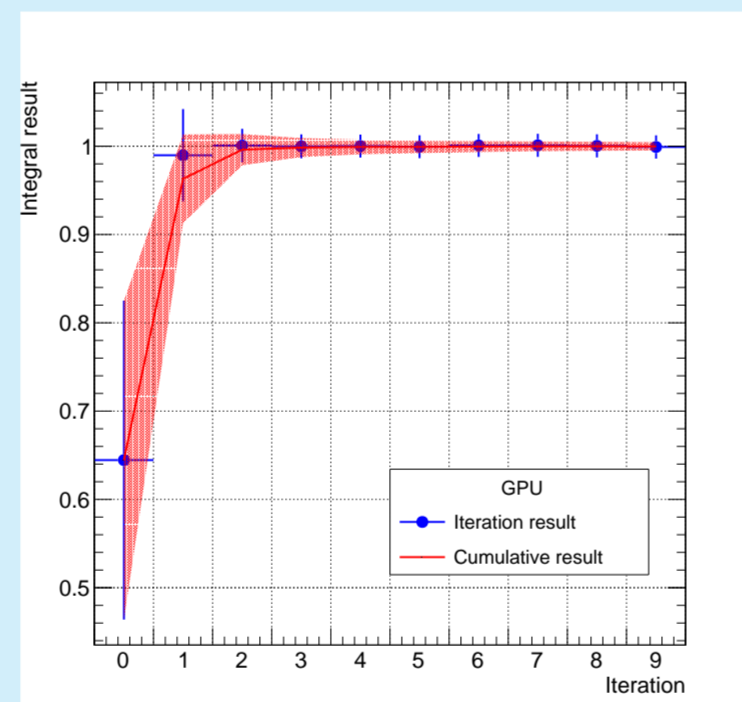
The VEGAS algorithm samples the integrand and adapts itself, so that the points are concentrated in the regions that make the largest contribution to the integral.

- ▶ Hydra implementation follows the corresponding GSL algorithm.
- ▶ No limit in the number of dimensions.

Example: integrating a normalized Gaussian distribution in 10 dimensions.

```
1 //VegasState hold resources and configurations
2 VegasState<10, hydra::device::sys> State_d(_min, _max);
3 //max number of iterations
4 State_d.SetIterations( 10 );
5 //max error
6 State_d.SetMaxError( 0.001 );
7 //number of calls
8 State_d.SetCalls( 5e5 );
9 //number of calls in training mode
10 State_d.SetTrainingCalls( 1e4 );
11 //number of training iterations
12 State_d.SetTrainingIterations(2);
13 //Vegas integrator object
14 Vegas<N, hydra::device::sys> Vegas_d(State_d);
15 //Gaussian10D is a functor describing a 10-dimensional
16 //Gaussian distribution.
17 Vegas_d.Integrate(Gaussian10D);
```

Results and performance:

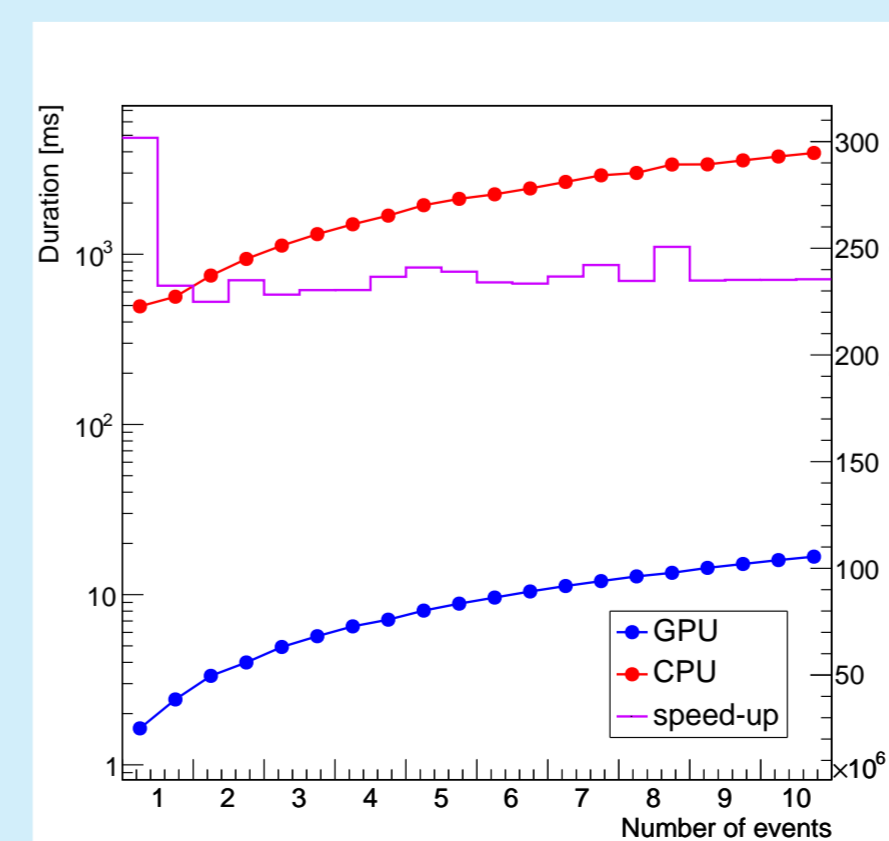
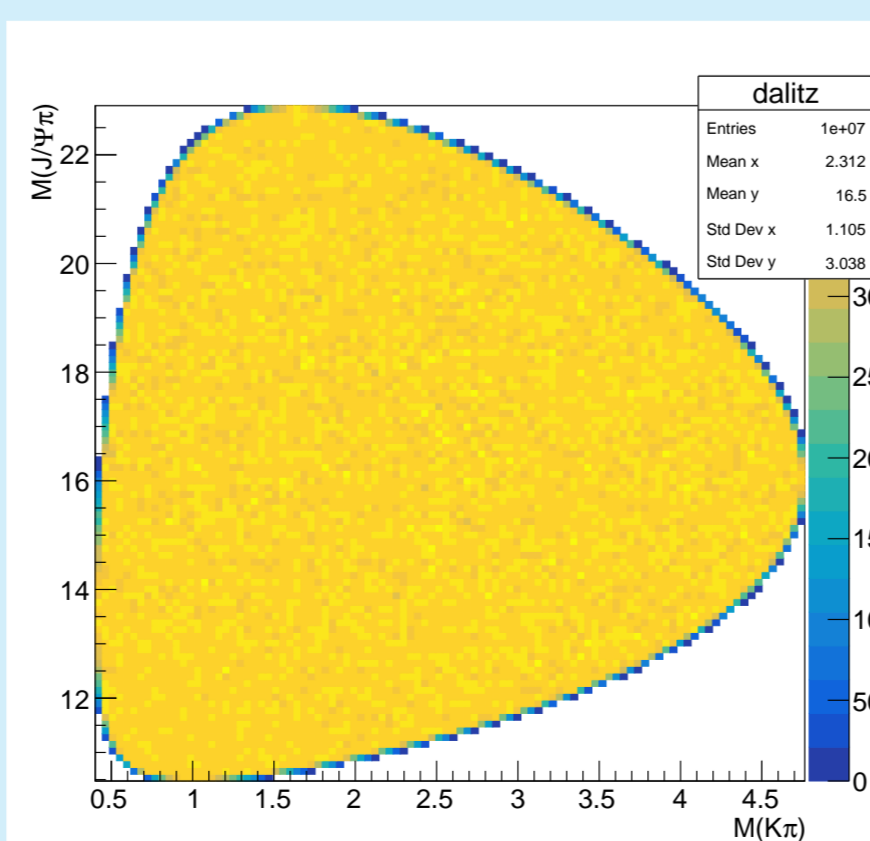


Phase-Space Monte Carlo

- ▶ No limitation on the number of particles in the final state.
- ▶ Support the generation of sequential decays and other features.

```
1 //Masses of the particles
2 hydra::Vector4R Mother(mother_mass, 0.0, 0.0, 0.0);
3 double Daughter_Masses[3]{daughter1_mass, daughter2_mass,
4     daughter3_mass };
5 //Create PhaseSpace object
6 hydra::PhaseSpace<3> phsp(Mother_mass, Daughter_Masses);
7 //Allocate the container for the events
8 hydra::Events<3, device> events(ndecays);
9 //Generate
10 phsp.Generate(Mother, events.begin(), events.end());
```

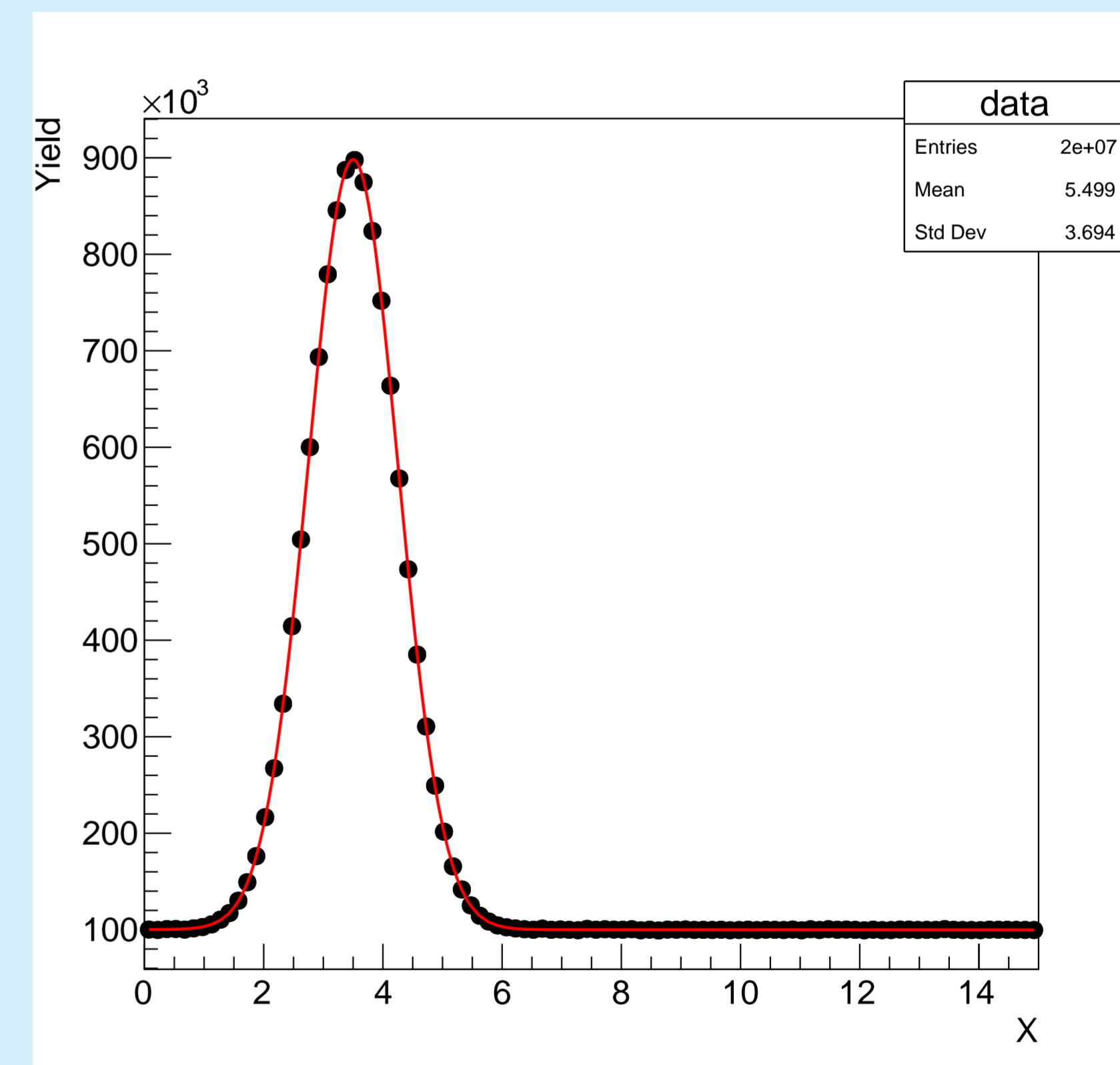
Results and performance:



Interface to Minuit2

- ▶ Hydra implements an interface to ROOT::Minuit2 that parallelizes the FCN calculation.
- ▶ This dramatically accelerates the calculation over large datasets.
- ▶ The pdfs are normalized on-the-fly using analytical or numerical integration algorithms provided by Hydra.
- ▶ Data is passed using iterators.

```
1 //Model = N_g * Gaussian + N_e * Exponential
2 //component pdfs
3 GaussAnalyticIntegral GaussIntegral(min, max);
4 ExpAnalyticIntegral ExpIntegral(min, max);
5 auto Gaussian_PDF =
6     hydra::make_pdf(Gaussian, GaussIntegral);
7 auto Exponential_PDF =
8     hydra::make_pdf(Exponential, ExpIntegral);
9
10 //add the pdfs to make a extended pdf model
11 std::array<hydra::Parameter*, 2>
12     yields{NGaussian, NExponential};
13 auto Model = hydra::add_pdfs(yields, Gaussian_PDF,
14     Exponential_PDF );
15 //get the FCN
16 auto Model_FCN = hydra::make_loglikelihood_fcn(Model,
17     data_d.begin(), data_d.end());
18
19 //pass the FCN to Minuit2
20 ...
```



Timing:

- ▶ Fit on GPU: 4.865 seconds
- ▶ Fit on CPU: 299.867 seconds
- ▶ Speed-up: ~62x

Summary



Hydra's development has been supported by the National Science Foundation under the grant number PHY-1414736.

- ▶ The project is hosted on GitHub: <https://github.com/MultithreadCorner/Hydra>
- ▶ The package includes a suite of examples.
- ▶ It is being used on the measurement of the Kaon mass at LHCb.
- ▶ A Google Summer of Code (GSoc) student has been working with the developers to add Python bindings. This is implemented for OpenMP and TBB back-ends.