Hydra

A. Augusto Alves Jr

Presented at 33th ROOT Parallelism, Performance and Programming Model,
CERN, March 1, 2018

- Hydra: design and features
- Basic interfaces, algorithms and performance
- Integration with ROOT
- Summary

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, CUDA and TBB on the suitable devices.
- It is focused on portability, usability, performance and precision.

The package has been presented in several computing conferences:

- **Hydra: Accelerating Data Analysis in Massively Parallel Platforms**- University of Washington, 21-25 August 2017, Seattle
- **Hydra: A Framework for Data Analysis in Massively Parallel Platforms** - NVIDIA's GPU Technology Conference, May 8-11, 2017 - Silicon Valley, US
- **Hydra** - HSF-HEP analysis ecosystem workshop, 22-24 May 2017 Amsterdam
- **MCBooster and Hydra: two libraries for high performance computing and data analysis in massively parallel platforms**- Perspectives of GPU computing in Science September 2016, Rome
- **Efficient Python routines for analysis on massively multi-threaded platforms-Python bindings for the Hydra C++ library** -Google Summer of Code project 2017

Design

The main design features are:

- Static polymorphic structure.
- Optimized containers that can store polymorphic and multidimensional data-sets using SoA layout.
- Enforced separation between algorithm and data. Data handled using iterators and all classes manages resources using RAII.
- Enforced type and thread-safeness.
- 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`
  - `hydra::device::sys`

The source files written using Hydra and standard C++ compile for GPU and CPU just exchanging the extension from .cu to .cpp and one or two compiler flags. There is no need to re-factory code.

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.

Functors

- Hydra adds features and type information to generic functors using the CRTP idiom.
- A generic functor with N parameters is represented like this:

```
1   struct SomeFunctor: public hydra::BaseFunctor<MyFunctor,double,N>
2   {
3   ...
4   // Evaluate() method for multidimensional data with same type
5   template<typename T> __hydra_dual__
6   inline double Evaluate(unsigned int n, T* x) const { /*actual calculation*/ }
7
8   // implement the Evaluate() method for polymorphic multidimensional (tuple)
9   template<typename T> __hydra_dual__
10  inline double Evaluate(T& x) const { /*actual calculation*/ }
11  };
```

- All functors deriving from `hydra::BaseFunctor<Func,ReturnType,NPars>` can be cached, used to perform fits and to compose more complex mathematical expressions.
- All the basic arithmetic operators are overloaded. Composition is also possible:

```
auto compose_functor = hydra::compose(C, A, B,...)
```

Support for C++11 lambdas

The user can define a C++11 lambda function and convert it into a Hydra functor, with or without parameters, using `hydra::wrap_lambda()`:

```
1  double two = 2.0;
2  //define a simple lambda and capture "two"
3  auto lambda = [=] __hydra_dual__ (unsined int n, double* x) {
4            return two*sin(x[0]);
5        };
6  //convert into a Hydra functor
7  auto Lambda_I  = hydra::wrap_lambda(lambda);
8
9  //define a parameter
10 auto multiplier = hydra::Parameter::Create().Name("multiplier").Value(2.0);
11
12 //define a simple lambda and capture "two"
13 auto Lambda_II = [] __hydra_dual__ (size_t np, hydra::Parameter* params, unsined int n,  double* x){
14           return params[0]*sin(x[0]);
15       };
16
17 //convert into a Hydra functor
18 auto Lamba_II  = hydra::wrap_lambda(my_lambda, multiplier);
19
20 //set the multiplier to a different value
21 my_lamba_II.SetParameter(0, 3.0);
```

SoA friendly containers: `hydra::multivector` and `hydra::multiarray`

`hydra::multivector` and `hydra::multiarray` allow to store multidimensional data efficiently using structure of arrays layout.

- `hydra::multivector` is optimized to store data with different types. Example:

```
1  ...
2  //3D data containers storing N {unsigned, int, double} entries.
3  hydra::multivector<hydra::tuple<unsigned, int, double>, hydra::device::sys_t> data_d(N); // Device
4  hydra::multivector<hydra::tuple<unsigned, int, double>, hydra::host::sys_t>   data_h(N); // Host
5  ...
```

- `hydra::multiarray` is optimized to store data of same type. Example:

```
1  ...
2  //3D data containers storing N {double, double, double} entries.
3  hydra::multiarray<double, 3, hydra::device::sys_t> data_d(N); // Device
4  hydra::multiarray<double, 3, hydra::host::sys_t>   data_h(N); // Host
5  ...
```

In the following slides I will discuss how to use these container classes to store objects of different types, saving memory and with efficient access pattern. I will use `hydra::multivector`, but everything is valid for `hydra::multiarray`.

Data is accessed using subscript operator and iterator idioms. A comprehensive interface is provided:

- STL-like interface to access and modify the whole container: `operator[]`, `begin()`, `end()`, `pop_back()`, `shrink_to_fit()...` and so on. 32 methods in total.
- STL-like interface to access columns, using `hydra::placeholders::{_0, _1, _2,...,_100}`:

```
1   ...
2   hydra::multivector<hydra::tuple<unsigned, int, double>, hydra::device::sys_t> v(10);
3
4   //Accessing column 2 (double) using iterators. Very efficient.
5   for(auto c = v.begin( _2 ); c !=v.end( _2 ); c++)
6    std::cout << *(c) << std::endl;
7
8   //Accessing column 2 (double) using subscript operator and placeholders. Very efficient.
9   for(size_t i = 0; i < v.size();i++)
10   std::cout << v[_2][i] << std::endl;
11
12   //Accessing column 2 (double) using subscript operator and hydra::get. Less efficient.
13  for(size_t i = 0; i < v.size();i++)
14   std::cout << hydra::get<2>(v[i]) << std::endl;
15  ...
```

- Direct and reverse constant access using **caster lambdas**.

hydra::multivector: storing objects using caster lambdas

It is possible to convert entries to objects on-the-fly, using the direct and reverse caster iterators and the subscript operato, or push-back objects into the container using a caster lambda operating the opposite direction

```
1   //the  caster
2   struct ComplexCaster{
3       // tuple -> complex
4       template<typename T>  __hydra_dual__
5       inline hydra::complex<T> operator()(hydra::tuple<T, T>& v) {
6           return hydra::complex<T>(hydra::get<0>(v), hydra::get<1>(v));
7       }
8       //complex -> tuple
9       template<typename T>  __hydra_dual__
10      inline hydra::tuple<T, T> operator()( hydra::complex<T>& v) {
11          return hydra::tuple<T, T>(v.real(), v.imag() );
12      }
13  };
14
15  hydra::multivector<hydra::tuple<double, double>, hydra::device::sys_t> v;
16
17  for(size_t j = 0; j<10; j++)
18    v.push_back(ComplexCaster(), hydra::complex(1.0*j, 2.0*j) );
19
20  for(auto x=v.begin(ComplexCaster()), x!=v.end(ComplexCaster()); x++ )
21    std::cout << *x << std::endl;
22  ...
```

Fitting data-sets with Hydra: PDFs

- PDFs are represented by the `hydra::Pdf<Functor, Integrator>` class template and can be conveniently built using the function `hydra::make_pdf( functor, integrator)`.
- The PDF evaluation and normalization are can executed in different back-ends.
- PDF objects cache the normalization integrals results. The user can monitor the cached values and corresponding errors.
- It is also possible to represent models composed by the sum of two or more PDFs. Such models are represented by the class templates
  - `hydra::PDFSumExtendable<Pdf1, Pdf2,...>`
  - `hydra::PDFSumNonExtendable<Pdf1, Pdf2,...>`

  and can be built using the function `hydra::add_pdfs({yield1, yield2,...}, pdf1, pdf2,...);`

Fitting data-sets with Hydra: FCN

The FCN is defined binding a PDF to the data the PDF is supposed to describe.

- Hydra implements classes and interfaces to allow the definition of FCNs suitable to perform maximum likelihood fits on unbinned and binned data-sets.
- The different use cases for Likelihood FCNs are covered specializing the class template `hydra::LogLikelihoodFCN<PDF, Iterator, Extensions...>` .
- Objects representing likelihood-based FCNs are conveniently instantiated using the function templates:
  - `hydra::make_likelihood_fcn(data.begin(), data.end() , pdf)`
  - `hydra::make_likelihood_fcn(data.begin(), data.end() , weights.begin(), pdf)`

  where `data.begin()`, `data.end()` and `weights.begin()` are iterators pointing to the data-set range, its weights or bin-contents.

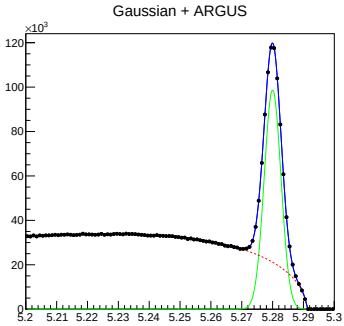
Example 1: Gaussian + Argus

```
1  //Analysis range
2  double min =  5.20, max = 5.30;
3
4  //Gaussian: parameters definition
5  hydra::Parameter  mean  = hydra::Parameter::Create().Name("Mean").Value( 5.28).Error(0.0001).Limits(5.27,5.29);
6  hydra::Parameter  sigma = hydra::Parameter::Create().Name("Sigma").Value(0.0027).Error(0.0001).Limits(0.0025,0.0029);
7  //Gaussian: PDF definition using analytical integration
8  auto Signal_PDF = hydra::make_pdf( hydra::Gaussian<>(mean, sigma),
9            hydra::GaussianAnalyticalIntegral(min, max));
10
11 //Argus: parameters definition
12 auto  m0     = hydra::Parameter::Create().Name("M0").Value(5.291).Error(0.0001).Limits(5.28, 5.3);
13 auto  slope  = hydra::Parameter::Create().Name("Slope").Value(-20.0).Error(0.0001).Limits(-50.0, -1.0);
14 auto  power  = hydra::Parameter::Create().Name("Power").Value(0.5).Fixed();
15 //Argus: PDF definition using analytical integration
16 auto Background_PDF = hydra::make_pdf( hydra::ArgusShape<>(m0, slope, power),
17           hydra::ArgusShapeAnalyticalIntegral(min, max));
18
19 //Signal and Background yields
20 hydra::Parameter N_Signal("N_Signal"        ,500, 100, 100 , nentries) ;
21 hydra::Parameter N_Background("N_Background",2000, 100, 100 , nentries) ;
22
23 //Make model
24 auto Model = hydra::add_pdfs( {N_Signal, N_Background}, Signal_PDF, Background_PDF);
```

Example 1: 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));

...
```

Example 1: Gaussian + Argus



Unbinned fit with 1,949,204 events.

- FCN calls: 789
- Intel® Core™ i7-4790 CPU @ 3.60 GHz (1 thread):150,72 s
- Intel® Core™ i7-4790 CPU @ 3.60 GHz (8 threads):28,875 s
- NVidia TitanZ GPU: 2,75 s
- Speed-up GPU vs CPU: $\sim$54x (1 thread) / $\sim$10x(8 threads)

Note: GPU scales better with problem size.

Usefulness of Hydra to ROOT, Roofit and RooStats

- Hydra provides some generic infrastructure components to handle multidimensional data with a light-weight, memory efficient and multi-thread friendly design. These components can be deployed already to improve ROOT algorithms, as it is.
- Hydra does not expose parallelism implementation details. It deploys parallelism to accelerate generic high-level. well documented algorithms. So, ROOT, Roofit and RooStats can invoke these algorithms to calculate numerical integrals, generate MC etc.
- Objects like parameters and PDFs can be made persistent without major problems, once a serialization scheme is chosen.

This level of integration would requires the implementation of a thin C++ interface layer, at maximum. During the GSoC-2017, we learned how to implement python bindings. The integrates the fit interface of Hydra with vanilla ROOT models can be accomplished using lambdas, and letting Hydra managing the FCN. The other way around, passing Hydra's FCN to ROOT, requires the implementation of a interface. Integration of the Hydra's fit facility with RooFit would be more involved.

Interaction with ROOT's intrinsic parallelism

- Most of the ROOT approach to parallelism is currently expressed in `ROOT::TDataFrame`. Hydra already implements many of the features currently supported by this class.
- Hydra algorithms run over data using iterators. I gave looked the design of `ROOT::TDataFrame`, and it seems to me that this class can easily be extended to provide iterators to bunches of data (loaded in memory) that can be processed by Hydra.
- By using Hydra with `ROOT::TDataFrame` is not necessary to implement "Actions" (in the `ROOT::TDataFrame` sense). Hydra smart-ranges, filter and `hydra::eval` already implements all the functionality of an "Action". Smart-ranges are intrinsically are lazy and can be nested.
- ROOT deploys TBB as parallel scheduler (Am I wrong ?). Hydra supports TBB as well, but does control the back-end run-time behavior. So, wherever Hydra algorithms are deployed inside, or aside, ROOT, Hydra will honor the caller's back-end setup.

Summary

- The project is hosted on GitHub:

  https://github.com/MultithreadCorner/Hydra

- A major release is under preparation. If you want use Hydra today, please fetch the code from the `hydra_rc2.0` branch

  https://github.com/MultithreadCorner/Hydra/hydrarc2.0

- The manual is available online: https://hydra-documentation.readthedocs.io/en/latest/
- The package includes a suite of 28 examples covering: fit, phase-space Monte Carlo, parallel and polymorphic containers (variant), numerical integration, PDF sampling and random number generation etc.
- It is being used on the measurement of the Kaon mass at LHCb.

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

Backup

Example 2: $D^+ \to K^-\pi^+\pi^+$

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).
- Did my best to code reproduce the corresponding EvtGen's DDalitz model.

$D^+ \to K^-\pi^+\pi^+$: contributions

- Contributions for each $K\pi$ channel: N.R., κ, $K^*(892)^0$, $K_0^*(1425)$, $K_2^*(1430)$ and $K_1(1780)$. The total number of parameters is 22: complex coefficients, masses and widths.
- Resonances are represented by the template `class Resonance<Channel, L>`, where $Channel = 1, 2, 3$ and L is a `hydra::Wave` object.
- Non-resonant contribution represented by `class NonResonant`.
- Hydra provides:
  - `hydra::BreitWignerLineShape<hydra::Wave L>`
  - `hydra::ZemachFunction<hydra::Wave L>`
  - `hydra::CosTheta`
  - `hydra::complex` ... etc.

$D^+ \to K^-\pi^+\pi^+$: contributions

Defining a contribution:

```
1  //K*(892)
2  //parameters
3  auto mass = hydra::Parameter::Create().Name("MASS_KST_892" ).Value(KST_892_MASS )
4                                        .Error(0.0001).Limits(KST_892_MASS*0.95,  KST_892_MASS*1.05 );
5
6  auto width = hydra::Parameter::Create().Name("WIDTH_KST_892").Value(KST_892_WIDTH)
7                                        .Error(0.0001).Limits(KST_892_WIDTH*0.95, KST_892_WIDTH*1.05);
8
9  auto coef_re = hydra::Parameter::Create().Name("A_RE_KST_892" ).Value(KST_892_CRe)
10                                       .Error(0.001).Limits(KST_892_CRe*0.95,KST_892_CRe*1.05).Fixed();
11
12 auto coef_im = hydra::Parameter::Create().Name("A_IM_KST_892" ).Value(KST_892_CIm)
13                                       .Error(0.001).Limits(KST_892_CIm*0.95,KST_892_CIm*1.05).Fixed();
14 //contributions per channel
15 Resonance<1, hydra::PWave> KST_892_Resonance_12(coef_re, coef_im, mass, width, D_MASS, K_MASS, PI_MASS, PI_MASS , 5.0);
16
17 Resonance<3, hydra::PWave> KST_892_Resonance_13(coef_re, coef_im, mass, width, D_MASS, K_MASS, PI_MASS, PI_MASS , 5.0);
18
19 //total contribution
20 auto KST_892_Resonance = (KST_892_Resonance_12 - KST_892_Resonance_13);
```

The other resonances are defined in a similar way.

$D^+ \to K^-\pi^+\pi^+$: model

```
1  //NR
2  coef_re = hydra::Parameter::Create().Name("A_RE_NR" ).Value(NR_CRe).Error(0.001).Limits(NR_CRe*0.95,NR_CRe*1.05);
3  coef_im = hydra::Parameter::Create().Name("A_IM_NR" ).Value(NR_CIm).Error(0.001).Limits(NR_CIm*0.95,NR_CIm*1.05);
4
5  auto NR = NonResonant(coef_re, coef_im);
6
7  //Total model |N.R + \sum{ Resonaces }|^2
8  auto Norm = hydra::wrap_lambda(
9       []__host__  __device__ (unsigned int n, hydra::complex<double>* x) {
10            hydra::complex<double> r(0,0);
11            for(unsigned int i=0; i< n;i++) r += x[i];
12            return hydra::norm(r);}
13      );
14
15 //Functor
16 auto Model = hydra::compose(Norm, K800_Resonance, KST_892_Resonance,
17               KST0_1430_Resonance, KST2_1430_Resonance, KST_1680_Resonance, NR);
18
19 //PDF
20 auto Model_PDF = hydra::make_pdf( Model,
21                 hydra::PhaseSpaceIntegrator<3, hydra::device::sys_t>(D_MASS, {K_MASS, PI_MASS, PI_MASS}, 500000));
```

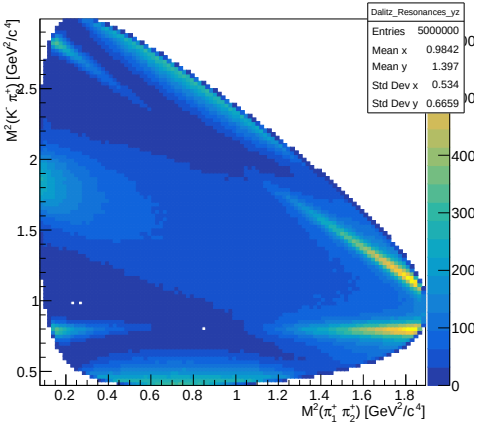
$D^+ \to K^-\pi^+\pi^+$: data generation, management and fit

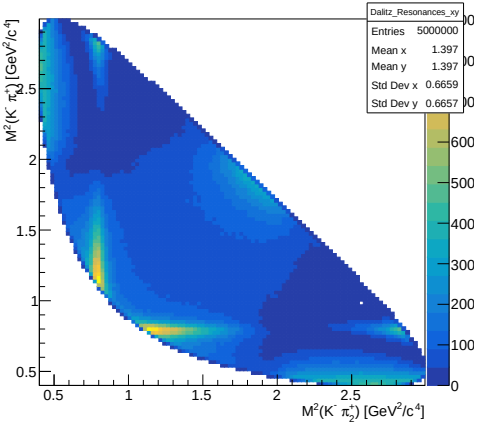
- Each entry of the dataset contains the four-vectors of the three final states.
- Dataset generation is managed by the template `class hydra::PhaseSpace<N>`
- The data is generated sampling the model on the device, in bunches of hundred of thousands events, which are then stored in a `hydra::Decays<N, Backend >` container allocated on the host memory space.
- When necessary, the data-set is transferred to the suitable device to perform the fit, histograming etc.

```
1  ...
2  //get the fcn
3  auto fcn = hydra::make_loglikehood_fcn(Model_PDF, particles.begin(), particles.end());
4  //minimization strategy
5  MnStrategy strategy(2);
6  //create Migrad minimizer
7  MnMigrad migrad_d(fcn, fcn.GetParameters().GetMnState() ,  strategy);
8  //fit...
9  FunctionMinimum minimum_d =  FunctionMinimum( migrad_d(5000, 5) );
```

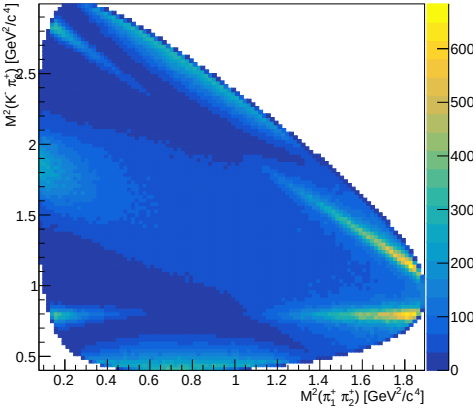
$D^+ \to K^-\pi^+\pi^+$: Dataset

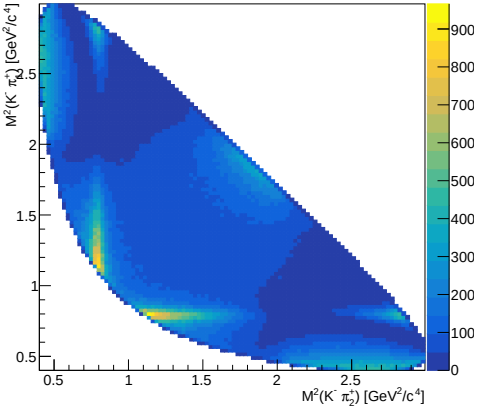
Toy data (5,000,000 events)



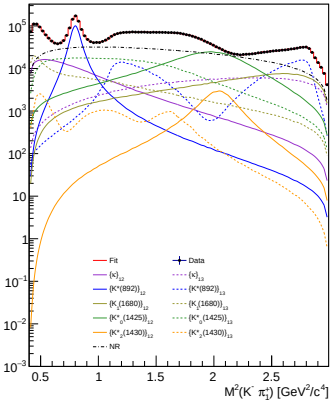


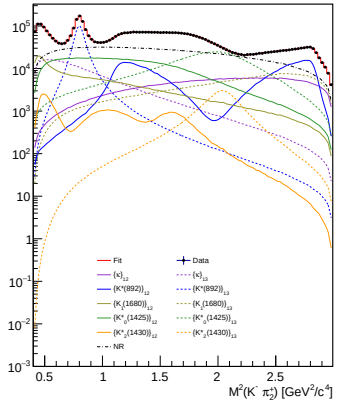
$D^+ \to K^-\pi^+\pi^+$: Fit result

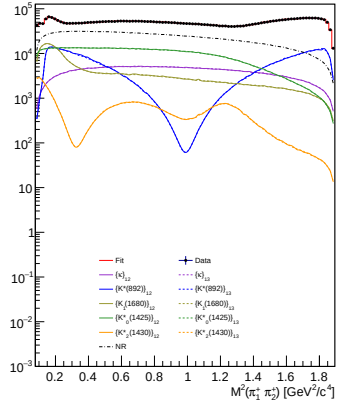




$D^+ \to K^-\pi^+\pi^+$: Projections







- 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.

Performance: CPU with OpenMP

The table below summarizes the time spent to perform a fit with 2.5 Million events.

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

Performance: CPU with TBB

The table below summarizes the time spent to perform a fit with 2.5 Million events.

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

Performance: GPU with CUDA

The table below summarizes the time spent to perform a fit with 2.5 Million events.

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

Obs.: Unfortunately, NVidia's Teslas (K40c and P100) of LHCb/Cincinnati server was unavailable when I performed this study.

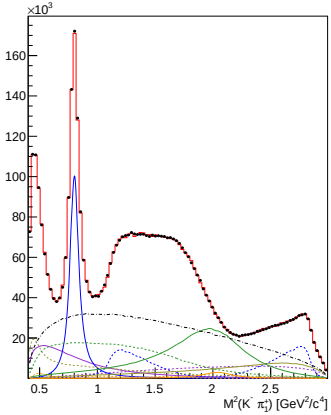
Comments

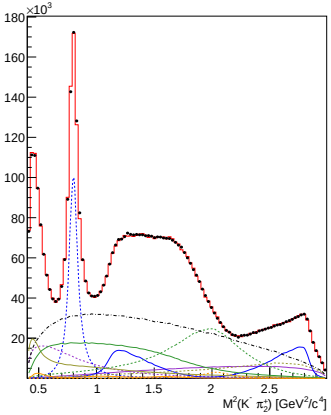
Yes! You can really perform quite complex amplitude unbinned fits in a matter of minutes.

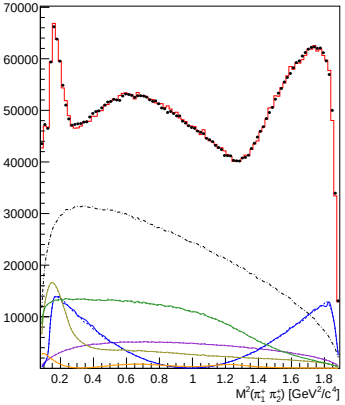
- Same code compiled and executed on hardware with different architecture, providing numerically identical results and showing consistent scale with the available resources.
- Observed speed-up by a factor 25x or higher on data fit. All other operations take maximum two or three dozens of milliseconds.
- I will profile later the scaling with the number of events. GPUs scales nicely with problem size... much better than CPUs.
- Not really a necessary to be a C++ expert to code your model on Hydra: no previous experience or specific knowledge on CUDA, OpenMP or TBB required.]
- Code is absolutely portable: you can run it on lxplus, on your gamer desktop, institutional laptop. You can share your code with the colleagues during the review of your analysis etc.

Hydra is not a sub-product of one the data analysis I performed. Since the beginning, Hydra has been designed to be a generic and open framework.

$D^+ \to K^-\pi^+\pi^+$: Projections







$D^+ \to K^-\pi^+\pi^+$:Fit fractions

```
KST800_12_FF :0.0782446
KST800_13_FF :0.0784398
KST892_12_FF :0.101073
KST892_13_FF :0.100459
KST1425_12_FF :0.17922
KST1425_13_FF :0.178935
KST1430_12_FF :0.00996452
KST1430_13_FF :0.00994939
KST1680_12_FF :0.0732225
KST1680_13_FF :0.0730777
NR_FF :0.44089
Sum :1.32348
```

$D^+ \to K^-\pi^+\pi^+$: data generation

```
1  //Mother particle
2  hydra::Vector4R D(D_MASS, 0.0, 0.0, 0.0);
3
4  // create PhaseSpace object for D-> K pi pi
5  hydra::PhaseSpace<3> phsp{K_MASS, PI_MASS, PI_MASS};
6
7  //allocate memory to hold the final states particles
8  hydra::Decays<3, hydra::device::sys_t > Events( nentries );
9
10 //generate the final state particles
11 phsp.Generate(D, Events.begin(), Events.end());
12
13 //container hold the unweighted dataset on the host
14 hydra::Decays<3, hydra::host::sys_t > toy_data;
15
16 //unweighted on device
17 auto last = Events.Unweight(Model, 1.0);
18
19 //allocate memory to hold the unweighted dataset
20 toy_data.resize(last);
21
22 //copy
23 hydra::copy(Events.begin(), Events.begin()+last, toy_data.begin());
```

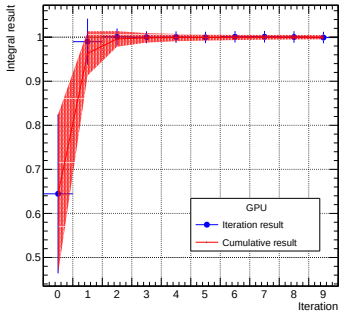
Vegas-like multidimensional numerical integration

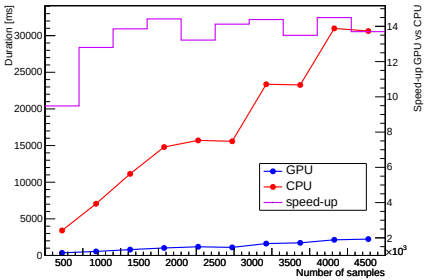
The VEGAS algorithm performs numerical integration using importance sampling:

```
1   constexpr size_t N=10;
2   //VegasState hold resources and configurations
3   VegasState<N, hydra::device::sys> State_d(_min, _max);
4   State_d.SetIterations( 10 );
5   State_d.SetMaxError( 0.001 );
6   State_d.SetCalls( 5e5 );
7   State_d.SetTrainingCalls( 1e4 ); //<-- set the number of training samples
8   State_d.SetTrainingIterations(2); //<-- number of training iterations
9
10  //Vegas integrator object
11  Vegas<N, hydra::device::sys> Vegas_d(State_d);
12
13  //integrate a 10D Gaussian
14  Vegas_d.Integrate(Gaussian);
```

Vegas-like multidimensional numerical integration

Integrating a normalized Gaussian distribution in 10 dimensions.





System configuration:

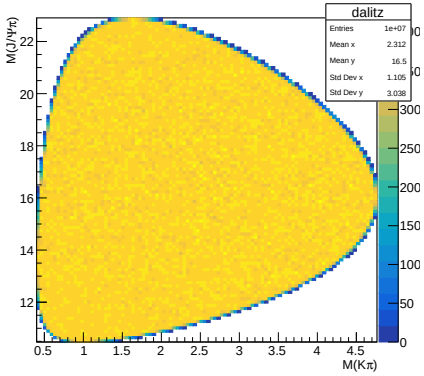
- GPU model: Tesla K40c
- CPU: Intel® Xeon(R) CPU E5-2680 v3 @ 2.50GHz (one thread)

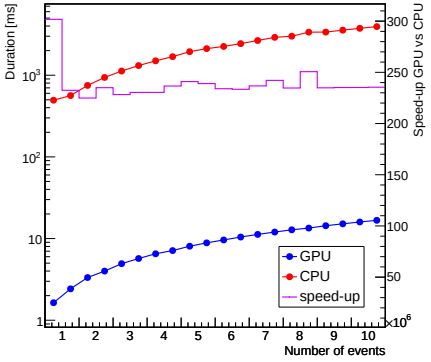
Phase-Space Monte Carlo

- Phase-Space Monte Carlo is used to simulate decay of particles.
- Beyond generate samples, the Phase-Space Monte Carlo generator of Hydra can calculates integrals and evaluate functions in-place to build datasets.

```
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, daughter3_mass };
4   //Create PhaseSpace object
5   hydra::PhaseSpace<3> phsp(Daughter_Masses);
6   //Allocate the container for the events
7   hydra::Decays<3, device> events(ndecays);
8
9   //Generate
10  phsp.Generate(Mother, events.begin(), events.end());
```

Phase-Space Monte Carlo





System configuration:

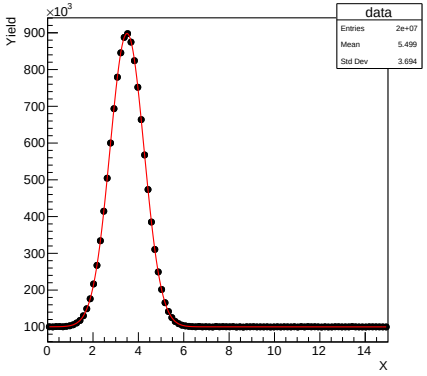
- GPU model: Tesla K40c
- CPU: Intel® Xeon(R) CPU E5-2680 v3 @ 2.50GHz (one thread)

Interface to Minuit2

```
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    = hydra::make_pdf(Gaussian, GaussIntegral);
6  auto Exponential_PDF = hydra::make_pdf(Exponential, ExpIntegral);
7
8  //add the pds to make a extended pdf model
9  std::array<hydra::Parameter*, 3>  yields{NGaussian, NExponential};
10 auto Model = hydra::add_pdfs(yields, Gaussian_PDF, Exponentia_PDF );
11 //get the FCN
12 auto Model_FCN = hydra::make_loglikehood_fcn(Model, data_d.begin(), data_d.end());
13
14 //pass the FCN to Minuit2
15 ...
```

Interface to Minuit2

20 million event maximum likelihood unbinned fit.



Timing:

- Fit on GPU: 4.865 seconds
- Fit on CPU: 299.867 seconds
- Speed-up: $\sim$62x

System configuration:

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

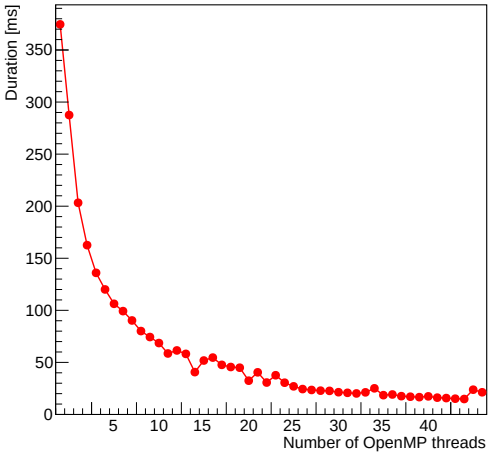
Comments

- As usual, the relative performance depends on many factors. The main are the problem size and the specificities of the hardware environment.
- Users can profile and deploy each algorithm on the more suitable back-end. Many interesting things can be done using `std::future` and `std::async` to dispatch algorithms asynchronously in different back-ends.
- The public interfaces are set using static arrays and C++11 Standard Library objects like `std::initializer_list` and `std::array`, so it is easy to integrate with existing software.

Phase-Space Monte Carlo

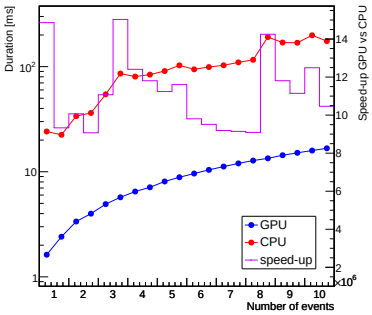
System configuration:

- CPU: Intel® Xeon(R) CPU E5-2680 v3 @ 2.50GHz x 48

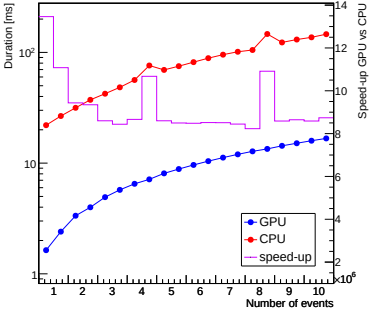


Phase-Space Monte Carl0

GPU vs OpenMP



GPU vs TBB



System configuration:

- GPU model: Tesla K40c
- CPU: Intel® Xeon(R) CPU E5-2680 v3 @ 2.50GHz x 48

Vegas-like multidimensional numerical integration

System configuration:

- CPU: Intel® Xeon(R) CPU E5-2680 v3 @ 2.50GHz x 48