Hydra: A framework for data analysis in massively parallel platforms

A. Augusto Alves Jr

Presented at CHEP 2018 Conference
9-13 July 2018, Sofia, Bulgaria
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.
Design

- Static polymorphic structure.
- Optimized containers to store polymorphic and multidimensional data-sets using SoA layout.
- Enforced type and thread-safeness and strong separation between algorithms and data.
- 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 or double 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.
- Dense and sparse multidimensional histogramming.
- Support to C++11 lambdas, filters, smart-ranges, etc.
Functors and C++11 lambdas

- Hydra calls user’s code using functors.
- The framework adds features and type information to generic functors using the CRTP idiom.
- All functors derive from `hydra::BaseFunctor<Func,ReturnType,NPars>` and needs to implement the `Evaluate(...)` method.
- C++11 lambdas are supported via `hydra::wrap_lambda()`.
- Some of the built-in functors:
  - ArgusShape
  - BifurcatedGaussian
  - BreitWignerLineShape
  - Chebychev
  - ChiSquare
  - CosHelicityAngle
  - CrystalBallShape
  - Exponential
  - Gaussian
  - M12PhaseSpaceLineShape
  - PlanesDeltaAngle
  - Polynomial
  - WignerDFunctions
  - ZemachFunctions
PDFs representation

- PDFs are represented by the `hydra::Pdf<Functor, Integrator>` class template and can be conveniently built using the function `hydra::make_pdf(functior, integrator)`.
- The PDF evaluation and normalization can be executed in different back-ends.
- PDF objects cache the normalization integrals results. The user can inspect the cached values and corresponding errors.
- It is also possible to represent models composed by the sum of two or more PDFs.
- 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 types of log-likelihood FCNs are covered specializing the class template `hydra::LogLikelihoodFCN<PDF, Iterator, Extensions...>`, using the function template `hydra::make_likelihood_fcn(...)`.
Example 1: Gaussian + Argus

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

// Gaussian: parameters definition using ‘named parameter idiom’
auto mean = Parameter::Create("Mean").Value(5.28).Error(0.0001).Limits(5.27, 5.29);
auto sigma = Parameter::Create("Sigma").Value(0.0027).Error(0.0001).Limits(0.0025, 0.0029);

// Gaussian: PDF definition using analytical integration
auto Signal_PDF = make_pdf(Gaussian<>(mean, sigma), GaussianAnalyticalIntegral(min, max));

// Argus: parameters definition
auto m0 = Parameter::Create("M0").Value(5.291).Error(0.0001).Limits(5.28, 5.3);
auto slope = Parameter::Create("Slope").Value(-20.0).Error(0.0001).Limits(-50.0, -1.0);
auto power = Parameter::Create("Power").Value(0.5).Fixed();

// Argus: PDF definition using analytical integration
auto Background_PDF = make_pdf(ArgusShape<>(m0, slope, power), ArgusShapeAnalyticalIntegral(min, max));

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

// Make model
auto Model = hydra::add_pdfs({N_Signal, N_Background}, Signal_PDF, Background_PDF);
```
Example 1: Gaussian + Argus

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

// generator
Random<> Generator();

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

// Make model and fcn
auto fcn = make_loglikehood_fcn( model, data_range );

// Execute the fit 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(500, 5));
...
```
Example 1: Gaussian + Argus

Unbinned fit with 2 million 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
Example 2: \( D^+ \to K^- \pi^+ \pi^+ \)

PHYSICAL REVIEW D 78, 052001 (2008)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Parameter</th>
<th>E791</th>
<th>CLEO-c</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bar{K}^+(892)\pi^+ )</td>
<td>( a )</td>
<td>1.03 ± 0.30 ± 0.16</td>
<td>7.4 ± 0.1 ± 0.6</td>
</tr>
<tr>
<td></td>
<td>( \delta(\pi) )</td>
<td>-11 ± 14 ± 8</td>
<td>-18.4 ± 0.5 ± 8.0</td>
</tr>
<tr>
<td></td>
<td>FF (%)</td>
<td>13.0 ± 5.8 ± 4.4</td>
<td>8.9 ± 0.3 ± 1.4</td>
</tr>
<tr>
<td>( \bar{K}^0(1430)\pi^+ )</td>
<td>( a )</td>
<td>1 (fixed)</td>
<td>1 (fixed)</td>
</tr>
<tr>
<td></td>
<td>( \delta(\pi) )</td>
<td>0 (fixed)</td>
<td>0 (fixed)</td>
</tr>
<tr>
<td></td>
<td>FF (%)</td>
<td>1.23 ± 1.0 ± 0.9</td>
<td>1.12 ± 0.2 ± 2.0</td>
</tr>
<tr>
<td>( K^0(1430)\pi^+ )</td>
<td>( a )</td>
<td>1.01 ± 0.10 ± 0.08</td>
<td>3.00 ± 0.06 ± 0.14</td>
</tr>
<tr>
<td></td>
<td>( \delta(\pi) )</td>
<td>48 ± 7 ± 10</td>
<td>49.7 ± 0.5 ± 2.9</td>
</tr>
<tr>
<td></td>
<td>FF (%)</td>
<td>1.25 ± 1.4 ± 0.5</td>
<td>10.4 ± 0.6 ± 0.5</td>
</tr>
<tr>
<td>( \bar{K}^0(1430)\pi^+ )</td>
<td>( m ) (MeV/c^2)</td>
<td>1459 ± 7 ± 12</td>
<td>1463 ± 0.7 ± 2.4</td>
</tr>
<tr>
<td></td>
<td>( \Gamma ) (MeV/c^2)</td>
<td>175 ± 12 ± 12</td>
<td>163 ± 2.7 ± 3.1</td>
</tr>
<tr>
<td>( \bar{K}^0(1680)\pi^+ )</td>
<td>( a )</td>
<td>0.20 ± 0.05 ± 0.04</td>
<td>0.96 ± 0.03 ± 0.05</td>
</tr>
<tr>
<td>( \kappa\pi^+ )</td>
<td>( \delta(\pi) )</td>
<td>-54 ± 8 ± 7</td>
<td>-29.9 ± 2.5 ± 2.8</td>
</tr>
<tr>
<td></td>
<td>FF (%)</td>
<td>0.5 ± 0.1 ± 0.2</td>
<td>0.38 ± 0.02 ± 0.03</td>
</tr>
<tr>
<td>( \bar{K}^0(1680)\pi^+ )</td>
<td>( a )</td>
<td>0.45 ± 0.16 ± 0.02</td>
<td>6.5 ± 0.1 ± 1.5</td>
</tr>
<tr>
<td>( \kappa\pi^+ )</td>
<td>( \delta(\pi) )</td>
<td>28 ± 13 ± 15</td>
<td>29.0 ± 0.7 ± 4.6</td>
</tr>
<tr>
<td></td>
<td>FF (%)</td>
<td>2.5 ± 0.7 ± 0.3</td>
<td>1.28 ± 0.04 ± 0.28</td>
</tr>
<tr>
<td>( \kappa\pi^+ )</td>
<td>( a )</td>
<td>1.97 ± 0.35 ± 0.11</td>
<td>5.01 ± 0.04 ± 0.27</td>
</tr>
<tr>
<td></td>
<td>( \delta(\pi) )</td>
<td>-173 ± 8 ± 18</td>
<td>-163 ± 0.4 ± 5.8</td>
</tr>
<tr>
<td></td>
<td>FF (%)</td>
<td>47.8 ± 12.1 ± 5.3</td>
<td>33.2 ± 0.4 ± 2.4</td>
</tr>
<tr>
<td>( \bar{K}^0(1680)\pi^+ )</td>
<td>( m ) (MeV/c^2)</td>
<td>797 ± 19 ± 43</td>
<td>809 ± 1 ± 13</td>
</tr>
<tr>
<td></td>
<td>( \Gamma ) (MeV/c^2)</td>
<td>410 ± 43 ± 87</td>
<td>470 ± 9 ± 15</td>
</tr>
</tbody>
</table>

- Phases and magnitudes from paper above (see page 12, table 7).
- Mimics the corresponding EvtGen’s DDalitz model.
$D^+ \rightarrow K^- \pi^+ \pi^+$: contributions and toy data

- Contributions for each $K\pi$ channel: N.R., $\kappa$, $K^*(892)^0$, $K_0^*(1425)$, $K_2^*(1430)$ 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`.
- Each entry of the dataset (toy data) contains the four-vectors of the three final states.
- Dataset generation is run in parallel using the class `hydra::PhaseSpace<3>`
- Hydra provides:
  - `hydra::BreitWignerLineShape<hydra::Wave L>`
  - `hydra::ZemachFunction<hydra::Wave L>`
  - `hydra::CosTheta`
  - `hydra::complex` ... etc.
$D^+ \rightarrow K^- \pi^+ \pi^+$: Dataset

Toy data (5,000,000 events)
$D^+ \rightarrow K^- \pi^+ \pi^+$: Fit result

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

<table>
<thead>
<tr>
<th>Parallel system</th>
<th>Threads</th>
<th>Time (sec/min)</th>
<th>FCN Calls</th>
<th>Time/Call (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-4790 CPU @ 3.60GHz</td>
<td>1</td>
<td>5060.578 (1.4 hours)</td>
<td>1030</td>
<td>4.91</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>750.245 (12.50)</td>
<td>&quot;</td>
<td>0.73</td>
</tr>
<tr>
<td>Xeon(R) CPU E5-2680 v3 @ 2.50GHz</td>
<td>1</td>
<td>5128.480 (1.42 hours)</td>
<td>&quot;</td>
<td>4.98</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>784.252 (13.1)</td>
<td>&quot;</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>612.278 (10.2)</td>
<td>&quot;</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>371.838 (6.2)</td>
<td>&quot;</td>
<td>0.36</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>247.787 (4.1)</td>
<td>&quot;</td>
<td>0.24</td>
</tr>
</tbody>
</table>
The table below summarizes the time spent to perform a fit with 2.5 Million events.

<table>
<thead>
<tr>
<th>Parallel system</th>
<th>Threads</th>
<th>Time (s/min)</th>
<th>FCN Calls</th>
<th>Time/Call (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-4790 CPU @ 3.60GHz</td>
<td>8</td>
<td>746.684 (12.4)</td>
<td>1030</td>
<td>0.72</td>
</tr>
<tr>
<td>Xeon(R) CPU E5-2680 v3 @ 2.50GHz</td>
<td>48</td>
<td>184.779 (3.01)</td>
<td>&quot;</td>
<td>0.18</td>
</tr>
</tbody>
</table>
Performance: GPU with CUDA

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

<table>
<thead>
<tr>
<th>Parallel system</th>
<th>Time (s/min)</th>
<th>FCN Calls</th>
<th>Time/Call (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GTX Tesla P100</td>
<td>221.114 (3.68)</td>
<td>&quot;</td>
<td>0.21</td>
</tr>
<tr>
<td>GeForce GTX Titan Z (GPU 1)</td>
<td>336.672 (5.61)</td>
<td>&quot;</td>
<td>0.33</td>
</tr>
<tr>
<td>GeForce GTX 1050 Ti</td>
<td>729.165 (12.15)</td>
<td>&quot;</td>
<td>0.71</td>
</tr>
<tr>
<td>GeForce GTX 970M (video)</td>
<td>744.247 (12.40)</td>
<td>&quot;</td>
<td>0.72</td>
</tr>
</tbody>
</table>
From ROOT 6.13/03 and Hydra 2.1.0 it is possible to use Hydra interactively through ROOT, in both prompt and batch modes.

Configuration: `export ROOT_INCLUDE_PATH=/path-to-hydra/`

Example: `root -l -b my_macro_with_hydra.C++`

The code will parallelize using TBB instance controlled by ROOT.

Limitations: ROOT can’t deploy GPUs yet.
• Same code compiled and executed on hardware with different architectures, providing numerically identical results and showing consistent scale over the available resources.

• Observed speed-ups by a factor $O(10-100)$ on data fits. All other operations take maximum two or three dozens of milliseconds.

• It is 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 is required.

• Code is absolutely portable: you can run it on CERN’s lxplus machines, on your desktop, laptop, in summary, one can share its code or migrate calculations between different platforms without major concerns.
Summary

- The project is hosted on GitHub: https://github.com/MultithreadCorner/Hydra
- The manual is available online: https://hydra-documentation.readthedocs.io
- The package includes a suite of examples covering: ROOT integration, fit, phase-space Monte Carlo, parallel and polymorphic containers, 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 and by the European Research Council under the grant ERC-STG-639068.
Backup
Functors

- Hydra calls user’s code using functors.
- The framework adds features and type information to generic functors using the CRTP idiom.
- All functors derive from `hydra::BaseFunctor<Func,ReturnType,NPars>` and needs to implement the `Evaluate(...)` method.

A generic functor with N parameters is represented like this:

```cpp
struct MyFunctor: public hydra::BaseFunctor<MyFunctor,double,N>
{
    // constructors and assignment operator omitted
    ...

    // implement the Evaluate() method for arrays
    template<typename T> __hydra_dual__
    inline double Evaluate(T* x) { /*actual calculation*/ }

    // implement the Evaluate() method for tuples
    template<typename T> __hydra_dual__
    inline double Evaluate(T x) { /*actual calculation*/ }
};
```
Arithmetic operations and composition with functors

If $A$, $B$ and $C$ are Hydra functors, the code below is completely legal.

```cpp
... // basic arithmetic operations
auto A_plus_B = A + B;
auto A_minus_B = A - B;
auto A_times_B = A * B;
auto A_per_B = A/B;
... // any composition of basic operations
auto any_functor = (A - B)*(A + B)*(A/C);
... // $C(A,B)$ is represented by:
auto compose_functor = hydra::compose(C, A, B)
...```

These operations are lazy and there is no intrinsic limit on the number of functors participating on arithmetic or composition mathematical expressions.
Lambda functions are fully supported in Hydra.

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

```cpp
    double two = 2.0;

    //define a lambda capturing 'two' and convert it to a Hydra functor
    auto my_lamba_wrapped = hydra::wrap_lambda(
        [=] __hydra_dual__ (unsigned n, double* x){
            return two*sin(x[0]);
        });
```

It is also possible to add named parameters to C++11 lambdas. In Hydra’s jargon: “parametric lambdas”

```cpp
... //named parameter
auto multiplier = hydra::Parameter::Create().Name("multiplier").Value(2.0);

//set the multiplier to a different value
my_lamba_wrapped.SetParameter("multiplier", 3.0);
...
```

This feature is very useful for quickly prototyping new functors or to combine the existing ones.
Parameters representation

- Parameters are represented by the `hydra::Parameter` class and can hold name, limits and error.
- `hydra::Parameter` objects are thread safe and automatically tracked and managed by the `hydra::BaseFunctor<Func,ReturnType,NPars>` interface.
- Can be instantiated using the **named parameter idiom**:

```
1 auto P1 = hydra::Parameter::Create().Name("P1").Value(5.291).Error(0.0001).Limits(5.28, 5.3);
2 auto P2 = hydra::Parameter::Create("P3").Value(5.291).Limits(5.28, 5.3).Error(0.0001);
```

- Can be instantiated using the **parameter list idiom**

```
1 //name, value, error, minimum, maximum
2 hydra::Parameter P3("P3", 5.291, 0.0001, 5.28, 5.3);
```

Not all members in a functor are required to be represented by `hydra::Parameter` objects.
PDFs representation

- PDFs are represented by the `hydra::Pdf<Functor, Integrator>` class template and can be conveniently built using the function `hydra::make_pdf(functeur, integrator)`.
- The PDF evaluation and normalization 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,...);`
FCNs representation

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 typed of log-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.
\[ D^+ \rightarrow K^- \pi^+ \pi^+ : \text{contributions} \]

Defining a contribution:

```cpp
// 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.
$D^+ \rightarrow K^-\pi^+\pi^+:\text{model}$

```cpp
//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 $|N.R + \sum \text{Resonances}|^2$
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));
```
$D^+ \rightarrow K^- \pi^+ \pi^+$: 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 `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.

```cpp
... 
//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) );
```
$D^+ \rightarrow K^- \pi^+ \pi^+$: Projections
$D^+ \rightarrow K^- \pi^+ \pi^+$: Fit result
$D^+ \rightarrow K^-\pi^+\pi^+ : \text{Fit fractions}$

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
\[ D^+ \rightarrow K^- \pi^+ \pi^+ : \text{data generation} \]

```cpp
// Mother particle
hydra::Vector4R D(D_MASS, 0.0, 0.0, 0.0);

// create PhaseSpace object for D -> K pi pi
hydra::PhaseSpace<3> phsp{K_MASS, PI_MASS, PI_MASS};

// allocate memory to hold the final states particles
hydra::Decays<3, hydra::device::sys_t > Events( nentries );

// generate the final state particles
phsp.Generate(D, Events.begin(), Events.end());

// container hold the unweighted dataset on the host
hydra::Decays<3, hydra::host::sys_t > toy_data;

// unweighted on device
auto last = Events.Unweight(Model, 1.0);

// allocate memory to hold the unweighted dataset
toy_data.resize(last);

// copy
hydra::copy(Events.begin(), Events.begin()+last, toy_data.begin());
```
The package has been presented in several computing conferences and workshops:

- **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
Functor example: Gaussian

```cpp
template<unsigned int ArgIndex=0>
class Gaussian: public BaseFunctor<Gaussian<ArgIndex>, double, 2> {
    public:
        //copy constructor and assignment operator omitted
        Gaussian(Parameter const& mean, Parameter const& sigma):
            BaseFunctor<Gaussian<ArgIndex>, double, 2>({mean, sigma}) {}

        template<typename T>
        __hydra_host__ __hydra_device__ inline double Evaluate(unsigned int, T*x) const {
            double m2 = (x[ArgIndex] - _par[0])*(x[ArgIndex] - _par[0] );
            double s2 = _par[1]*_par[1];
            return exp(-0.5*m2/s2);
        }

        template<typename T>
        __hydra_host__ __hydra_device__ inline double Evaluate(T x) const {
            double m2 = ( get<ArgIndex>(x) - _par[0])*(get<ArgIndex>(x) - _par[0] );
            double s2 = _par[1]*_par[1];
            return exp(-0.5*m2/s2);
        }
};
```
NVidia GPUs

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
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)
System configuration:

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

System configuration:

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

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