Accelerating HEP data analyses in massively parallel platforms

Design hints from Hydra

A. Augusto Alves Jr

October 11, 2019

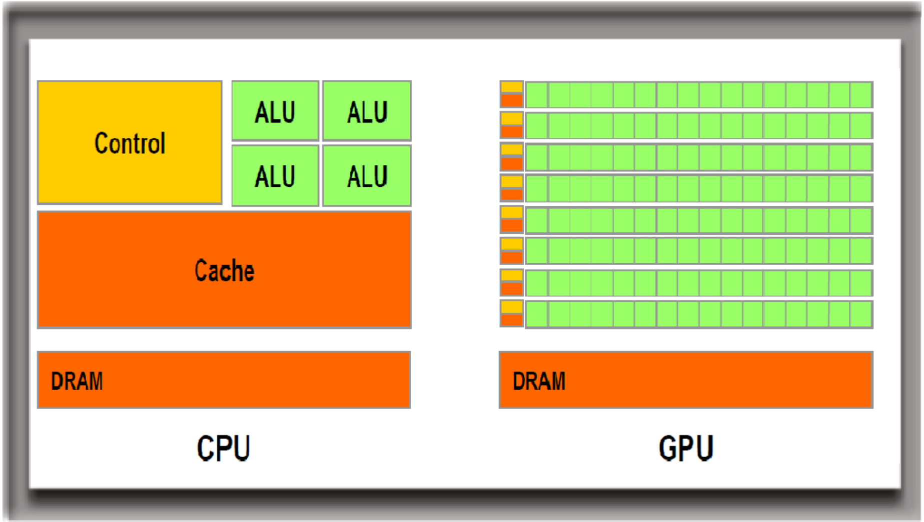
HYDRA
Multithreaded Data
Analysis Framework

- **CPU, GPU and parallelism.**
- **Hydra**
- **Examples and performance**
- **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.

Preliminars: CPUs and GPUs



Concurrency

The ability to execute 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 maximise throughput.
- Problems: race conditions, deadlocks, resource starvation etc....

Motivation to deploy 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 very often needs re-tune.
- Processors will not increase clock frequency any more. The current road-map to increase overall performance is to deploy concurrency.
- Multi-platform environments became very popular among data-centers, but HEP software is not completely prepared yet to deploy opportunistic computing strategies.

Hydra

Hydra proposes a computing model to address this situation. The framework provides collection of parallelized high-level algorithms and optimized containers, through a modern and functional interface, to enhance HEP software productivity and performance, keeping the portability between GPUs and multicore CPUs.

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.

Design

- Static polymorphic structure.
- Optimized containers to 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 or double-coding.

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 histogramming.
- Support to C++11 lambdas, filters, smart-ranges,... etc.

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

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:

```
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) { /*...code...*/ }

// implement the Evaluate() method for tuples
template<typename T> __hydra_dual__  inline double Evaluate(T x) { /*...code...*/ }

};
```

Arithmetic operations and composition with functors

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

```
...
//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.

Support for C++11 lambdas

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()` :

```
...
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, const double* x){

      return two*sin(x[0]);
  } );

...
```

Support for C++11 lambdas

It is also possible to add named parameters to C++11 lambdas. In Hydra's jargon: "parametric lambdas"

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

//
auto my_lamba_wrapped  = hydra::wrap_lambda(
  [] __hydra_dual__ (unsigned nparams, const hydra::Parameter* param, unsigned n, const double* x){

        return param[0]*sin(x[0]);

    }, multiplier);

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

This feature is very usefull 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*:

```cpp
auto P1 = hydra::Parameter::Create().Name("P1").Value(5.291).Error(0.0001).Limits(5.28, 5.3);
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*

```cpp
//name, value, error, minimum, maximum
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( functor, 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.

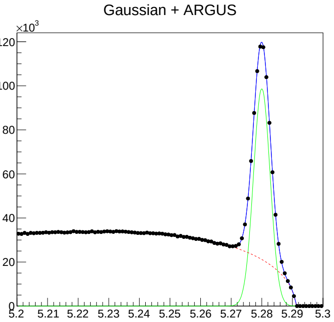
Example 1: Gaussian + Argus

```
//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);
```

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());
// or using range semantics: Generator.Sample(data, min, max, model.GetFunctor());

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

Comments

- Same code compiled and executed on hardware with different architecture, providing numerically identical results and showing consistent scaling over the available resources.
- Observed speed-ups by a factor O(10-1000) depending on the operations.
- 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.

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

Design hints: interface

- Rely on template parameter deduction and avoid open template instantiation
- Use named parameter semantics: `auto par = Parameter::Create().Name("par").Value(0.0)` has the same effect of `auto par = Parameter::Create().Value(0.0).Name("par")`.
- Implementation of *convenience functions* for instantiate templates via argument deduction. For example, instead of doing `ClassObj<ParType> obj(par);` it can be better to do `auto obj = make_obj(par);`. Remember that C++17 supports template argument deduction from constructors as well.
- Behavior of the interface needs to be defined and tested via unit-tests(ex. Catch2).
- Deploy range-based semantics. It is useful for nesting algorithms, to express lazy evaluation, to simplify syntax and to reach higher levels of abstraction. Compare this

```
...
ns::sort(input.begin(), input.end());
ns::transform( input.begin(), input.end(), output.begin(), some_functor );
auto result = ns::reduce(output.begin(), output.end());
...
```

with this `auto result = ns::reduce(ns::sort(input) | some_functor);`

Design hints: algorithms and data handling

- Extensive use of RAII (resource acquisition is initialization).
- Implementation of policy based design for algorithms and data-storage, for management of concurrency, resource allocation and release.
- Thrust implements STL-like algorithms abstracting away the parallel back-end. C++17 added support for parallel algorithms to the standard library. In both cases, the algorithms behavior are controlled by policies. It is wise to implement as much as possible the computing intensive routines on top of STL (or Thrust in case of Hydra), decomposing all calculations in terms of transforms, reductions etc.
- Taking advantage of the processor's cache prefetching mechanisms (SoA vs AoS).
- Using patterns that favor automatic vectorization or deploy it explicitly (can be tricky!)
- Using static polymorphism when performance has priority over run-time flexibility. Example: calling user's code using CRTP.

Integrating Hydra with ROOT's ACLIC and CLING

- 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.

Step-by-step installation in LXPLUS

This will compile for `OMP` and `CPP` backends:

1. `git clone https://github.com/MultithreadCorner/Hydra.git`
2. `cd Hydra`
3. `mkdir build` then `cd build`
4. `cmake -DTCLAP_INCLUDE_PATH=~/opt/tclap/include BUILD_DOXYGEN_DOCUMENTATION=FALSE ../`
5. `make -j12`

- You can install `TBB` somewhere and `export TBB_INSTALL_DIR=<your tbb>`
- TCLAP is distributed here: http://tclap.sourceforge.net/

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 in some analyses in LHCb, like the Measurement of the Kaon mass.
- Also for simulation of three-dimensional silicon sensor response at TIMESPOT collaboration.

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	$1.03 \pm 0.30 \pm 0.16$	$7.4 \pm 0.1 \pm 0.6$
	$\phi(°)$	$-11 \pm 14 \pm 8$	$-18.4 \pm 0.5 \pm 8.0$
	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)
	$\phi(°)$	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	$1.01 \pm 0.10 \pm 0.08$	$3.00 \pm 0.06 \pm 0.14$
	$\phi(°)$	$48 \pm 7 \pm 10$	$49.7 \pm 0.5 \pm 2.9$
	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	$0.20 \pm 0.05 \pm 0.04$	$0.962 \pm 0.026 \pm 0.050$
	$\phi(°)$	$-54 \pm 8 \pm 7$	$-29.9 \pm 2.5 \pm 2.8$
	FF (%)	$0.5 \pm 0.1 \pm 0.2$	$0.38 \pm 0.02 \pm 0.03$
$\bar{K}^*(1680)\pi^+$	a	$0.45 \pm 0.16 \pm 0.02$	$6.5 \pm 0.1 \pm 1.5$
	$\phi(°)$	$28 \pm 13 \pm 15$	$29.0 \pm 0.7 \pm 4.6$
	FF (%)	$2.5 \pm 0.7 \pm 0.3$	$1.28 \pm 0.04 \pm 0.28$
$\kappa\pi^+$	a	$1.97 \pm 0.35 \pm 0.11$	$5.01 \pm 0.04 \pm 0.27$
	$\phi(°)$	$-173 \pm 8 \pm 18$	$-163.7 \pm 0.4 \pm 5.8$
	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).
- Mimics 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:

```
//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.

$D^+ \to K^-\pi^+\pi^+$: 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));
```

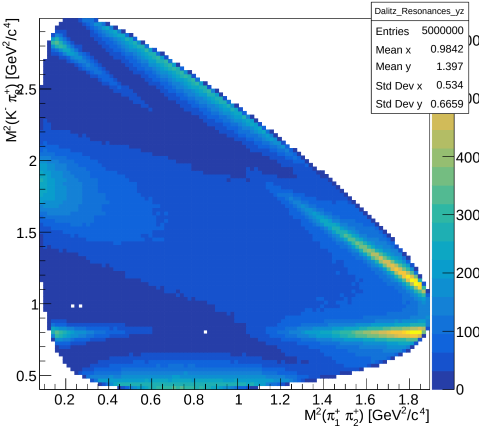
$D^+ \to K^-\pi^+\pi^+$: data generation, management and fit

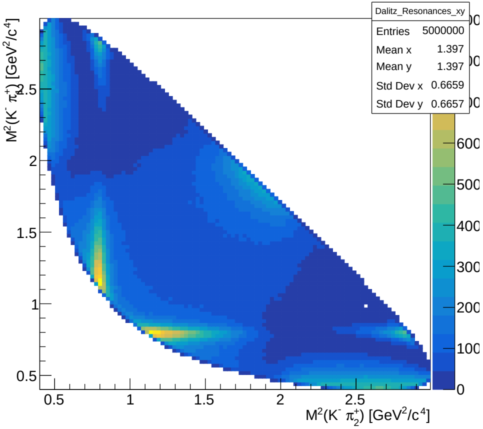
- Each entry of the dataset contains the four-vectors of the three final states.
- Dataset generation is managed by the template `class hydra::PhaseSpace<N>`
- The data is generated sampling the model on the device, in bunches of hundred of thousands events, which are then stored in a `hydra::Decays<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.

```
...
//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) );
```

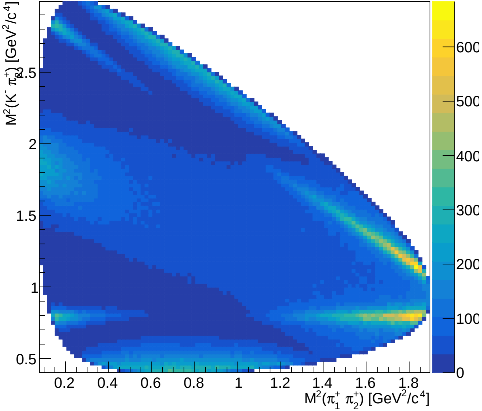
$D^+ \to K^-\pi^+\pi^+$: Dataset

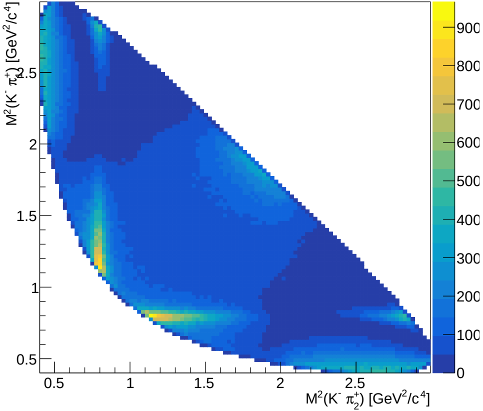
Toy data (5,000,000 events)



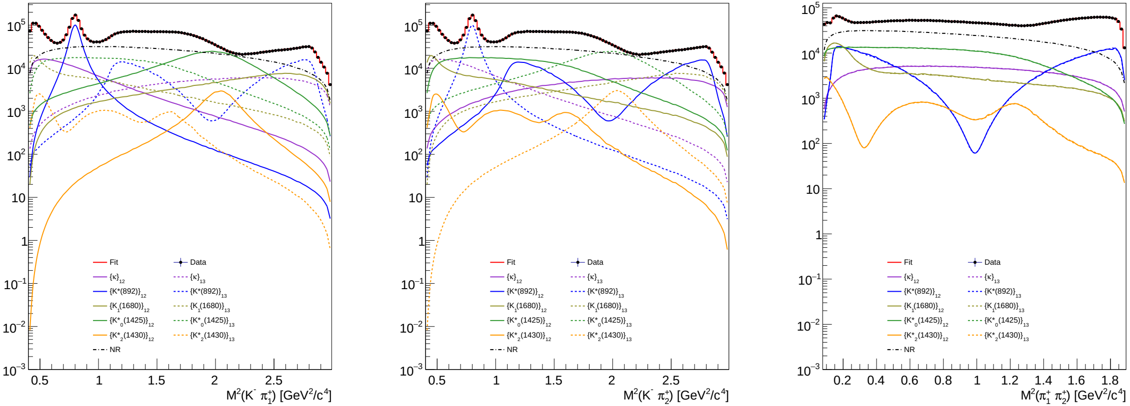


$D^+ \to K^-\pi^+\pi^+$: Fit result



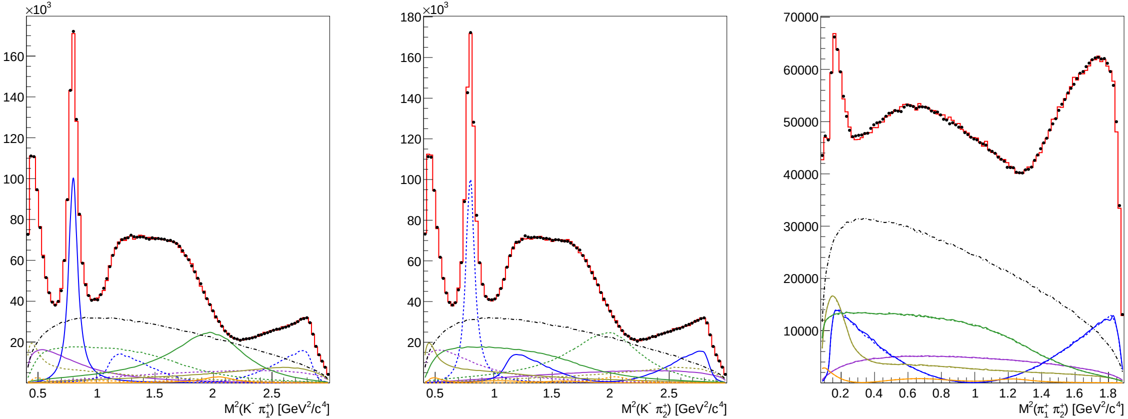


$D^+ \to K^-\pi^+\pi^+$: Projections



- 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.

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



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

.

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

```
//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());
```

Previous presentation

The package has been presented in several computing conferences and workshops:

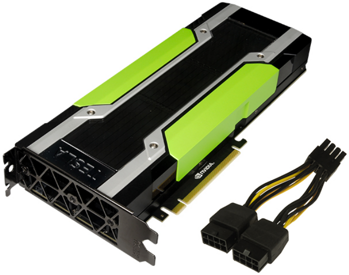
- **Hydra: Accelerating Data Analysis in Massively Parallel Platforms**- University of Washington, 21-25 August 2017, Seattle
- **Hydra: A Framework for Data Analysis in Massively Parallel Platforms** - NVIDIA's GPU Technology Conference, May 8-11, 2017 - Silicon Valley, US
- **Hydra** - HSF-HEP analysis 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

```
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

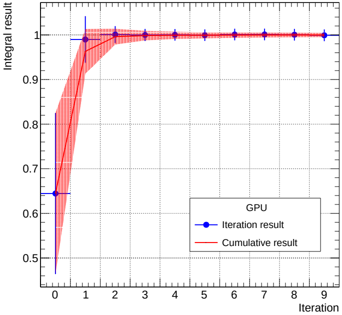
GPU Architecture: Kepler
CUDA Cores 5760
Base Clock (MHz) 705
Single-Precision Performance 4.3 - 5.0 TeraFLOPS
Double-Precision Performance 1.4 - 1.7 TeraFLOPS
Memory Interface 12GB GDDR5

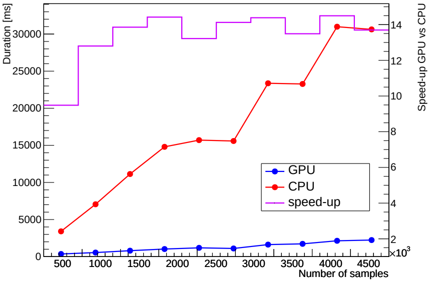


GPU Architecture: Pascal
CUDA Cores 3584
Base Clock (GHz) 1.126
Double-Precision Performance 4.7 TeraFLOPS
Single-Precision Performance 9.3 TeraFLOPS
Memory Interface 16GB CoWoS HBM2 at 732 GB/s

Vegas-like multidimensional numerical integration

Integrating a normalized Gaussian distribution in 10 dimensions.

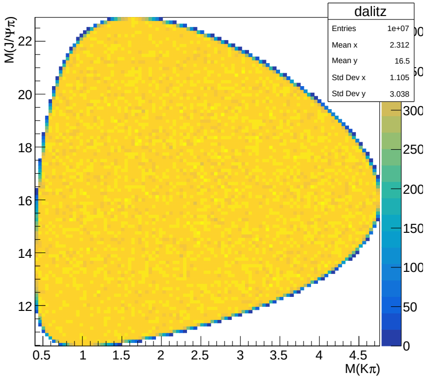


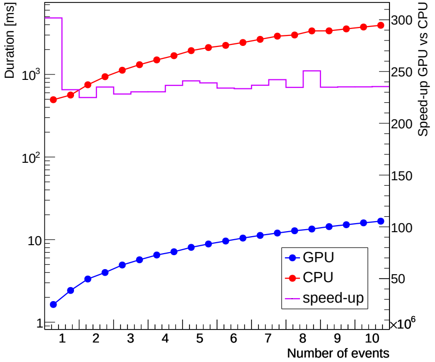


System configuration:

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

Phase-Space Monte Carlo





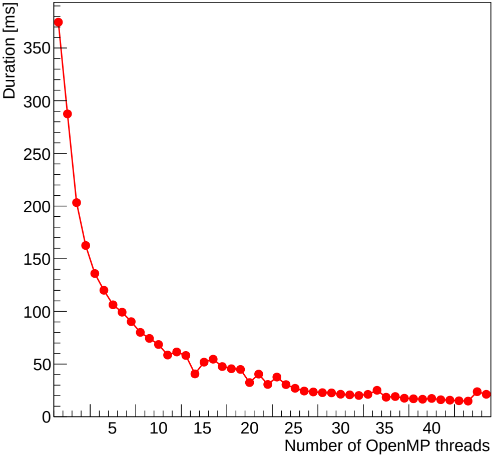
System configuration:

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

Phase-Space Monte Carlo

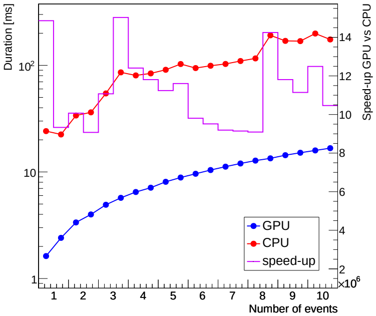
System configuration:

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

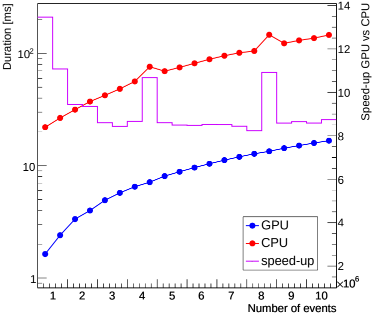


Phase-Space Monte Carl0

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