Hydra

A library for data analysis in massively parallel platforms

A. Augusto Alves Jr and Michael D. Sokoloff

University of Cincinnati
aalvesju@cern.ch

Presented at NVIDIA's GPU Technology Conference,
May 8-11, 2017 - Silicon Valley, US

Outline

- Design and goals of Hydra
- Basic functionalities and main algorithms
- Performance
  - Multidimensional numerical integration
  - Phase-space Monte Carlo generation
  - Interface to ROOT::Minuit2 and fitting
- Summary

Motivation

The Large Hadron Collider (LHC) and other facilities acquire 10's petabytes of data anually. The collective effort to analyze this amount data requires state-of-the-art software tools that:

- Scale efficiently to face the increasing statistics from the experiments.
- Meet the high precision requirements typically necessary to address High Energy Physics (HEP) problems.
- Are efficient and flexible enough to face the different conditions of specific HEP experiments.
- Are portable, scalable, compatible with existing software and hardware standards.

Hydra

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 is designed to run on Linux systems and to use OpenMP, CUDA and TBB enabled devices.
- It is focused on portability, usability, performance and precision.

Design and features

The main design features are:

- The library is structured using static polymorphism.
- There is absolutely no need to write explicit back-end oriented code.
- Clean and concise semantics.
- Interfaces are easy to use correctly and hard to use incorrectly.

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

Features

- Generation of Phase-space Monte Carlo samples.
- Sampling of multidimensional probability density functions.
- Data fitting using binned and unbinned multidimensional datasets.
- Evaluation of multidimensional functions over heterogeneous data sets.
- Numerical integration of multidimensional functions.

Functors

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

```
struct MyFunctor: public hydra::BaseFunctor<MyFunctor,double,N>
{
//MyFunctor constructor and other implementation details
...
//User always need to implement the Evaluate() method
template<typename T>
__host__ __device__
inline double Evaluate(T* x) { //actual calculation }
};
```

All functors deriving from `hydra::BaseFunctor<Func,ReturnType,NPars>` can be cached, used to perform fits and to compose more complex mathematical expressions.

Arithmetic operations and composition with functors

All the basic arithmetic operators are overloaded. Composition is also possible. 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)
...
```

- The functors resulting from arithmetic operations and composition can be cached as well.
- 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 simple lambda and capture "two"
auto my_lambda = [] __host__ __device__(double* x)
        { return two*sin(x[0]); };

//convert is into a Hydra functor
auto my_lamba_wrapped  = hydra::wrap_lambda(my_lambda);
...
```

- CUDA 8.0 supports lambda functions in device and host code.

Data containers

- `hydra::Point` represents multidimensional data points including its coordinates, value and errors.
- `hydra::PointVector` Looks like an array of structs, but data is stored in structure of arrays.

```
//two dimensional point
typedef hydra::Point<GReal_t, 2> point_t;
//two dimensional data set on the device
hydra::PointVector<point_t, device> data_d(1e6);
...
//get data from device
hydra::PointVector<point_t, host> data_h(data_d);
//fill a ROOT 2D histogram
TH2D hist("hist", "my histogram", 100, min, max);
for(auto row: data_h ){
  auto point(row);
  hist.Fill( point.GetCoordinate(0), point.GetCoordinate(1));
}
```

Functionalities

Data fitting and Monte Carlo generation

- Interface to ROOT::Minuit2 minimization package.
- Phase-space generator.
- Multidimensional p.d.f. sampling.
- Parallel function evaluation over multidimensional datasets

Numerical integration

- Flat Monte Carlo sampling.
- Vegas-like self-adaptive importance sampling (Monte Carlo).
- Gauss-Kronrod one-dimensional quadrature.
- Genz-Malik multidimesional quadrature.

Vegas-like multidimensional numerical integration

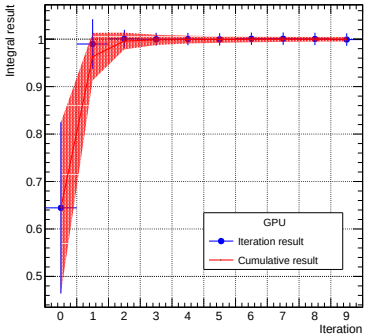
The VEGAS algorithm is based on importance sampling. It samples the integrand and adapts itself, so that the points are concentrated in the regions that make the largest contribution to the integral.

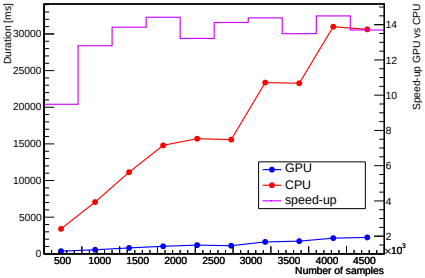
- Hydra implementation follows the corresponding GSL algorithm.
- No limit in the number of dimensions.

```
//VegasState hold resources and configurations
VegasState<N, device> State_d(_min, _max);
State_d.SetIterations( iterations );
State_d.SetMaxError( max_error );
State_d.SetCalls( calls );
State_d.SetTrainingCalls( tcalls );
State_d.SetTrainingIterations(1);
//Vegas integrator object
Vegas<N, device> Vegas_d(State_d);
//integrate a Gaussian
Vegas_d.Integrate(Gaussian);
```

Vegas-like multidimensional numerical integration

Processing a Gaussian distribution in 10 dimensions.





System configuration:

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

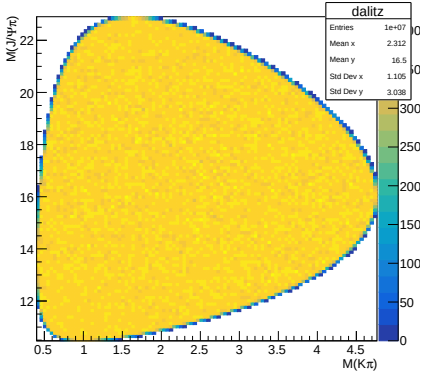
Phase-Space Monte Carlo

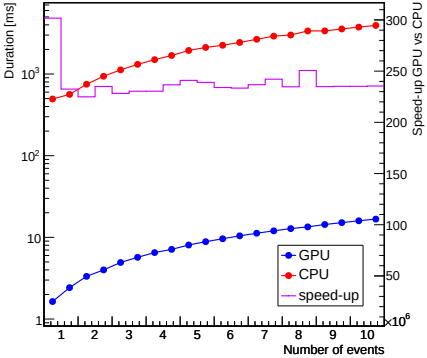
Describes the kinematics of a particle with a given four-momentum decaying to N-particle final state.

- No limitation on the number of particles in the final state.
- Support the generation of sequential decays.
- Generation of weighted and unweighted samples.

```
//Masses of the particles
hydra::Vector4R Mother(mother_mass, 0.0, 0.0, 0.0);
double Daughter_Masses[3]{daughter1_mass, daughter2_mass, daughter3_mass };

//Create PhaseSpace object
hydra::PhaseSpace<3> phsp(Mother_mass, Daughter_Masses);

//Allocate the container for the events
hydra::Events<3, device> events(ndecays);

//Generate
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

ROOT::Minuit2 is widely used in particle physics to find the minimum value of a multi-parameter function (FCN) and analyze the shape of the function around the minimum, and so to compute model's best-fit parameter values and uncertainties.

- 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 `hydra::PointVector`.

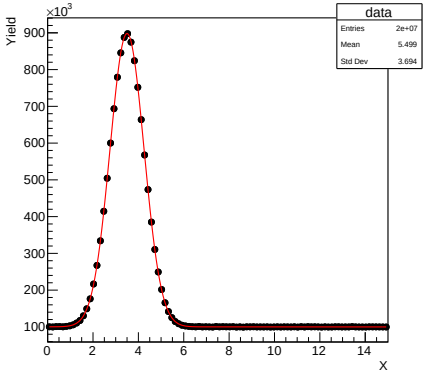
Interface to Minuit2

$$Model = N_g * Gaussian + N_e * Exponential$$

```
GaussAnalyticIntegral  GaussIntegral(min, max);
ExpAnalyticIntegral    ExpIntegral(min, max);

auto Gaussian_PDF   = hydra::make_pdf(Gaussian, GaussIntegral);
auto Exponentia_PDF = hydra::make_pdf(Exponentia, ExpIntegral);

//add the pds to make a extended pdf model
std::array<hydra::Parameter*, 3>  yields{NGaussian, NExponential};

auto Model = hydra::add_pdfs(yields, Gaussian_PDF, Exponentia_PDF );
model.SetExtended(1);

//get the FCN
auto Model_FCN = hydra::make_loglikehood_fcn(Model, data_d);

//pass the FCN to Minuit2
...
```

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)

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 at CERN on analyses aiming to measure the Kaon mass using large datasets.

Acknowledgments

- To Karen Tomko and Bradley Hittle from the Ohio Supercomputer Center.
- To the University of Cincinnati LHCb group.

Please, visit the page of the project, try it out, report bugs, make suggestions... Thanks!

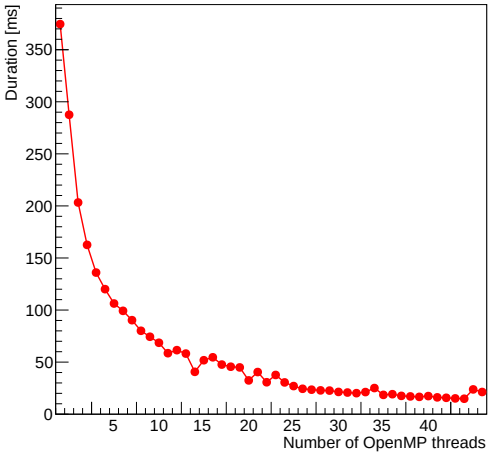
Backup

Phase-Space Monte Carlo

OpenMP: scalling with number of threads

System configuration:

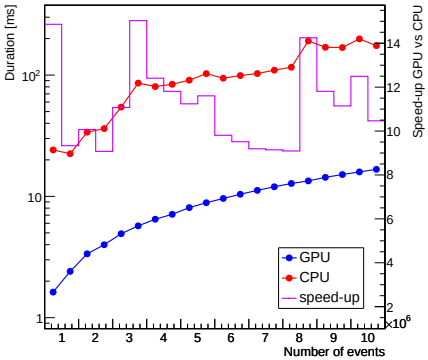
- CPU: Intel® Xeon(R) CPU E5-2680 v3 @ 2.50GHz x 48



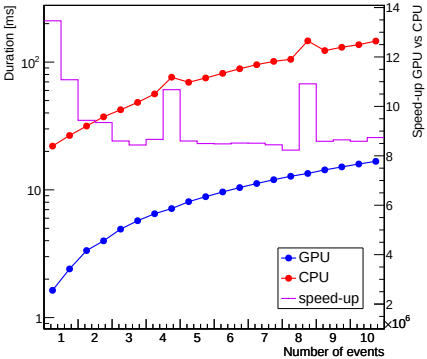
Phase-Space Monte Carlo

CUDA OpenMP, TBB

GPU vs OpenMP



GPU vs TBB



Vegas-like multidimensional numerical integration

OpenMP: scalling with number of threads

System configuration:

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