Hydra

A library for data analysis in massively parallel platforms

A. Augusto Alves Jr and M.D. Sokoloff

University of Cincinnati
aalvesju@cern.ch

Presented at the
Workshop Perspectives of GPU computing in Science
September 2016, Rome

Outline

- Design, strategies and goals of Hydra
- Functionalities
- Functors
- Data containers
- Function evaluation
- Multidimensional numerical integration
- Multidimensional random number generation
- Phase-Space Monte Carlo
- Interface to Minuit2 and fitting
- Summary

Hydra

Hydra is a header only templated C++ library to perform data analysis on massively parallel platforms.

- It is implemented on top of the C++ Standard Library and a variadic version of the Thrust library.
- Hydra runs on Linux systems and can perform calculations using OpenMP, CUDA, 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.
- Type safety is enforced at compile time in order to avoid the production of invalid or performance degrading code.
- No function pointers or virtual functions: stack behavior is known at compile time.
- Static polymorphism also increases the implementation of optimizations by the compiler.
- No need to write explicit device code.
- Clean and concise semantics. Interface easy to use correctly and hard to use incorrectly.
- RAII, CRTP...

Ex.: The same source files written using Hydra components and standard C++ compiles on GPU or CPU just exchanging the extension from .cu to .cpp.

Functionalities currently implemented

- Generation of phase-space Monte Carlo Samples with any number of particles in the final states.
- Sampling of multidimensional PDFs.
- Data fitting of binned and unbinned multidimensional data sets.
- Evaluation of multidimensional functions over heterogeneous data sets.
- Numerical integration of multidimensional functions using Monte Carlo-based methods: flat or self-adaptive (Vegas-like) .

Many other possibilities can be implemented just combining the core functionalities.

Functors

In Hydra most of the calculations are performed using function objects.

- Hydra add features, type information and interfaces to generic functors using the CRTP idiom.
- For example, a Gaussian with two fit parameters is represented like this:

```
struct Gauss:public BaseFunctor<Gauss,double, 2>
{
...
//client need to implement the Evaluate(T ) method
//for homogeneous data.
template<typename T>
__host__ __device__
inline double Evaluate(T* x){}

//or heterogeneous data.
template<typename T>
__host__ __device__
inline double Evaluate(T x){}
...
 };
```

- For all functors deriving from `hydra::BaseFunctor<Func,ReturnType,NPars>`:
    - If the calculation is expensive and the functor will be called many times, the first call results can be cached.
    - Can be used to compose more complex mathematical constructs.

Arithmetic operations and composition with functors

Hydra provides a lot "syntax sugar" to deal with function objects.

- 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 plus_functor = A + B; auto minus_functor = A - B;
auto prod_functor = A * B; auto div_functor = A/B;

//composition of basic operation
auto any_functor = (A - B)*(A + B)*(A/C);

// C(A,B) is represented by:
auto compose_functor = 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++ lambdas

Lambda functions are a very precious C++ resources that allow the implementation of new functionalities on-the-fly. These objects can hold state, capture variables defined in the enclosing scope etc...

- Lambda functions are supported in Hydra.
- In the client code one can define a lambda function at any point 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 = wrap_lambda(my_lambda);
...
```

- CUDA 8.0 (in RC status right now) supports lambda functions on device and host code.
- Just a friendly advise: capture variables always by value!

Data containers

Hydra algorithms can operate over any iterable C++ container defined in the C++ Standard Library or Thrust. Hydra provides PointVector, a built-in generic container, that can represent binned or unbinned multidimensional datasets.

PointVector is an iterable collection of Point objects:

- hydra::Point represents multidimensional data points with coordinates and value, error of the coordinates and error of the value.
- hydra::Point uses conditional base class members and methods injection in order to save memory and stack size.
- hydra::Point objects can be streamed to std::cout (on the host of course)
- Coordinates can be of any type that make sense... not only real numbers!

```
...
//two dimensional data set
PointVector<device, double, 2> data_d(1e6);
...
//get data from device and fill a ROOT 2D histogram
PointVector<host> data_h(data_d);

TH2D hist("hist", "my histogram", 100, min, max);

for(auto point: data_h )
  hist.Fill(point.GetCoordinate(0), point.GetCoordinate(1));
...
```

Generic function evaluation

Functors can be evaluated over large data sets using the template function `hydra::Eval`

- `hydra::Eval` returns a vector with results.
- `hydra::Range` provides the flexibility to combine different pieces of the same container, for example:

```
//single functor
Eval(Functor const&, Range<Iterators> const&... );
//multiple functors
Eval(thrust::tuple<Functors...> const&, Range<Iterators>const&...)
```

- It is not necessary to explicitly set any template parameter:

```
// lambda to calculate sin(x)
auto sinL = [] __host__ __device__(double* x){ return sin(x[0]);};
auto sinW = wrap_lambda(sinL);
// lambda to calculate cos(x)
auto cosL = [] __host__ __device__(double*  x){ return cos(x[0]);};
auto cosW = wrap_lambda(cosL);
// evaluation
auto functors = thrust::make_tuple( sinW, cosW);
auto range    = make_range( angles_d.begin(), angles_d.end());
auto result   = Eval( functors, range);
```

Multidimensional numerical integration

Hydra provides two MC based methods for multidimensional numerical integration: Plain Monte Carlo and self-adaptive Vegas-like algorithm (importance sampling).

- Hydra implementations follow closely the corresponding GSL algorithms.
- Methods can be configured via template parameters (policies) to call the integrand on the host or on the device and use different random number engines.
- Both methods use RAII to acquire, initialize and release the resources.
- Example of Vegas usage:

```
//Vegas state hold the resources for performing the integration
VegasState<1> *state = new VegasState<1>( min, max);
state->SetVerbose(-1);
state->SetAlpha(1.75);
state->SetIterations(5);
state->SetUseRelativeError(1);
state->SetMaxError(1e-3);
//10,000 call (fast convergence and very precice)
Vegas<1> vegas( state, 10000);
```

Multidimensional PDF sampling

The template class `hydra::Random` manages the multidimensional function sampling in Hydra.

- Generic PDF sampling using accept-reject method.
- `hydra::Random` provides an increasing number of basic distributions: Uniform, Breit-Wigner, Exponential...
- Can be configured via template parameters (policies) to use different random number generators.
- Methods take the target's container iterators as input and engage the generation on the host or device backend.

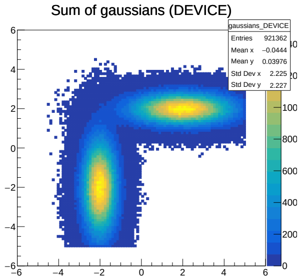
```
{
 //Random object with current time count as seed.
Random<thrust::random::default_random_engine>
Generator( std::chrono::system_clock::now().time_since_epoch().count() );
//1D host buffer
hydra::mc_host_vector<double>   data_h(nentries);
//uniform
Generator.Uniform(-5.0, 5.0, data_h.begin(), data_h.end());
//gaussian
Generator.Gauss(0.0, 1.0, data_h.begin(), data_h.end());
//exponential
Generator.Exp(1.0, data_h.begin(), data_h.end());
//breit-wigner
Generator.BreitWigner(2.0, 0.2, data_h.begin(), data_h.end());
 }
```

```cpp
{
//------------------------------------
// two gaussians hit-and-miss
//------------------------------------
//gaussian one
std::array<double, 2>  means1  ={2.0, 2.0 };
std::array<double, 2>  sigmas1 ={1.5, 0.5 };
//gaussian two
std::array<double, 2>  means2  ={-2.0, -2.0 };
std::array<double, 2>  sigmas2 ={0.5, 1.5 };
Gauss<2> Gaussian1(means1, sigmas1 );
Gauss<2> Gaussian2(means2, sigmas2 );

//add the pdfs
auto Gaussians = Gaussian1 + Gaussian2;

//2D range
std::array<double, 2>  min  ={-5.0, -5.0 };
std::array<double, 2>  max  ={ 5.0,  5.0 };

auto gaussians_data_d =
Generator.Sample<device>(Gaussians,min, max, ntrials );
}
```



Time for 10M events:

- GeForce Titan-Z: 0.063514s
- Intel i7 4 cores @ 3.0 GHz: 0.79484s

Phase-Space Monte Carlo

Hydra supports the production of phase-space Monte Carlo samples. The generation is managed by the class `hydra::PhaseSpace` and the storage is managed by the specialized container `hydra::Events`.

- Policies to configure the underlying random number engine and the backend used for the calculation.
- No limitation on the number of final states.
- Support the generation of sequential decays.
- `hydra::Events` is iterable and therefore is fully compatible with C++11 range semantics.
- Generation of weighted and unweighted samples.

The Hydra phase-space generator supersedes the library MCBooster (https://github.com/MultithreadCorner/MCBooster), from the same developer.

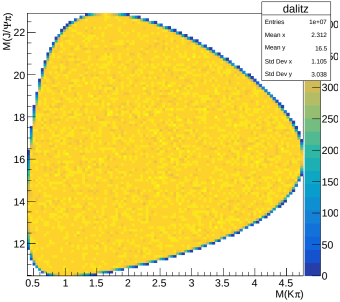
Phase-Space Monte Carlo

```
Vector4R B0(5.27961, 0.0, 0.0, 0.0);
vector<double>
massesB0{3.096916, 0.493677, 0.13957018 };

// PhaseSpace object for B0-> K pi J/psi
PhaseSpace<3> phsp(B0.mass(), massesB0);
// container
Events<3, device> B02JpsiKpi_Events_d(10e7);
// generate...
phsp.Generate(B0, B02JpsiKpi_Events_d);

//copy events to the host
Events<3, host>
B02JpsiKpi_Events_h(B02JpsiKpi_Events_d);

for(auto event: B02JpsiKpi_Events_h)
{... }
```



Time to generate 10M events:

- GeForce Titan-Z: 0.06896s
- Intel i7 4 cores @ 3.0 GHz: 0.53542s

Interface to Minuit2 and fitting

Hydra implements an interface to Minuit2 that parallelizes the FCN calculation. This accelerates dramatically the calculation over large datasets.

- The fit parameters are represented by the class `hydra::Parameter` and managed by the class `hydra::UserParameters`.
- `hydra::UserParameters` has the same semantics of `Minuit2::MnUserParameters`.
- Any positive definite Hydra-functor can be converted into PDF.
- The PDFs are normalized on-the-fly.
- The estimator is a policy in the FCN.
- Data is passed via iterators. Any iterable container can be used. I personally advise to use `hydra::PointVector`.

The FCN provided by Hydra can be used directly in Minuit2.

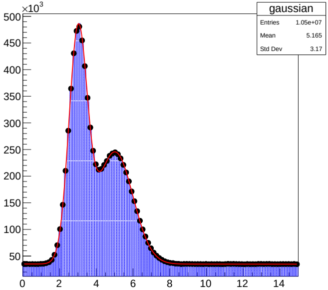
Interface to Minuit2 and data fitting

```
...
//Generate data
PointVector<device, double, 1> data_d(nentries);
//fill data container...
//get the FCN
auto modelFCN = make_loglikehood_fcn(model, data_d.begin(), data_d.end() );

//fit strategy
MnStrategy strategy(1);

//create Migrad minimizer
MnMigrad migrad(modelFCN,upar.GetState() , strategy);

//perform
FunctionMinimum minimum = migrad();
```



- The black dots represent ~10M event simulated datasample.
- The red line is the fit result
- The blue shadowed area is data sampled from the fitted model.

Summary and prospects

- The project is supported by NSF and is hosted on GitHub: https://github.com/MultithreadCorner/Hydra
- The package includes a suite of examples
- The next version will expand the range of options for data fitting and include histograming-related functionalities.

Please, visit the page of the project, give a try, report bugs, make suggestions... Thanks!