Hydra

A library for data analysis in massively parallel platforms

A. Augusto Alves Jr

HEP Analysis Ecosystem Workshop, Amsterdam 22-24 May 2017

Motivation to deploy massively parallel platforms on HEP

- A large fraction of the software used in HEP is legacy designed single threaded, C++03 and mono-platform routines.
- HEP experiments keep collecting samples with unprecedented large statistics.
- Data analyses get more and more complex... sometimes, 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 prepared yet to deploy opportunistic computing strategies.

Hydra purposes a model to address these issues. The library provides collection of high-level algorithms (typical computing bottlenecks) and optimized containers, through a modern interface, to enhance HEP software productivity and performance, keeping the portability between GPUs and multicore CPUs.

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.
- 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` and `hydra::device::sys`

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

Functionalities

Data fitting and Monte Carlo generation

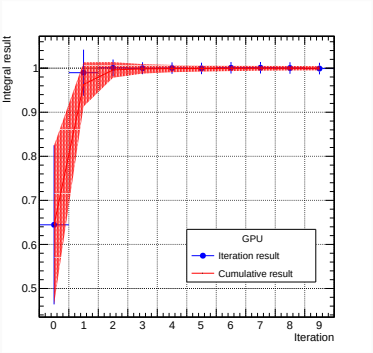
- Interface to ROOT::Minuit2 minimization package.
- Phase-space generator.
- Multidimensional p.d.f. sampling.
- Parallel function evaluation over multidimensional datasets

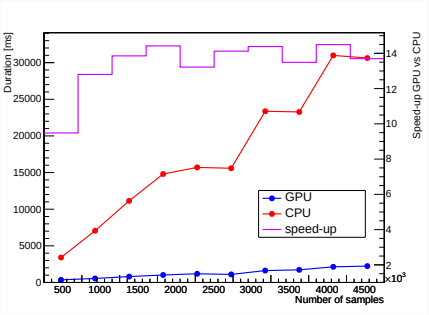
Numerical integration

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

Vegas-like multidimensional numerical integration

Integrating a normalized Gaussian distribution in 10 dimensions.

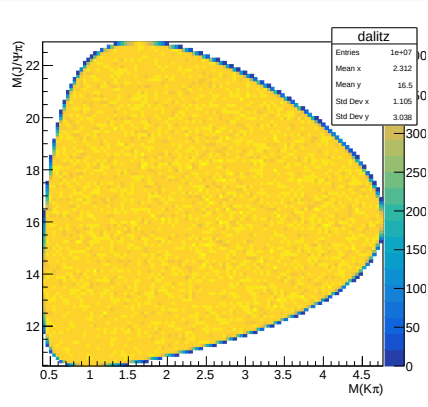


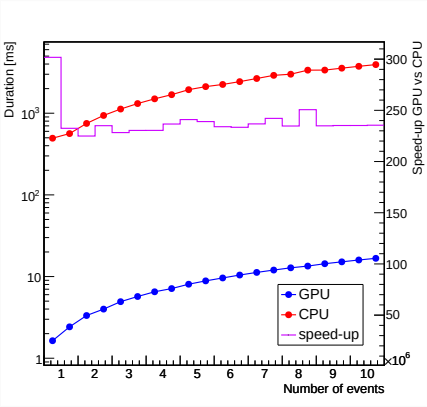


System configuration:

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

Phase-Space Monte Carlo



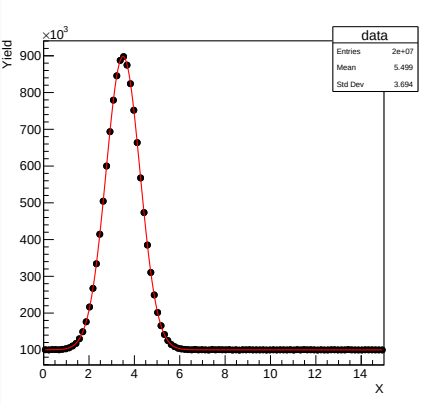


System configuration:

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

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

- Hydra provides high-level algorithms that run on NVidia GPUs and multi-core CPUs, transparently and concurrently.
- 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.
- Plans: kernel density estimation, convolution, multidimensional dense/sparse histograming of large datasets.

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 on the measurement of the Kaon mass at LHCb.
- Hydra was recently presented at NVidia's GTC 2017
- The project got a Google Summer of Code (GSoC) slot to port the library to Python.

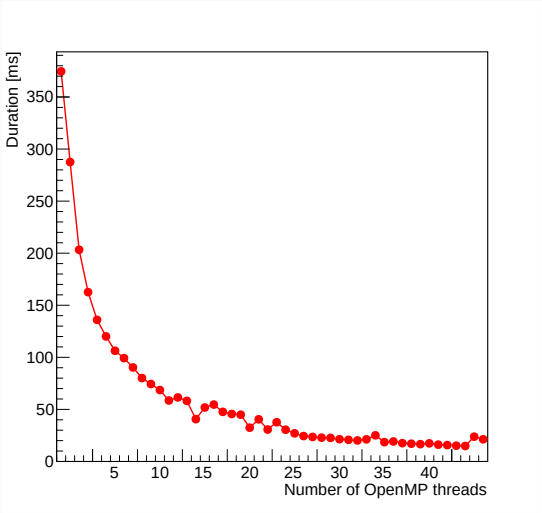
Please, visit the page of the project, try it out, report bugs, make suggestions... Thanks!

Backup

Phase-Space Monte Carlo

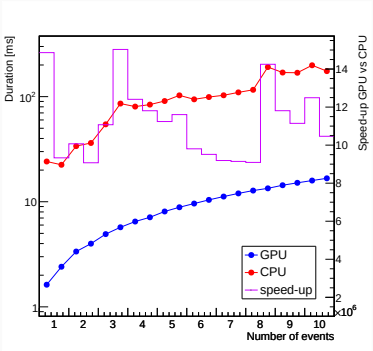
System configuration:

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

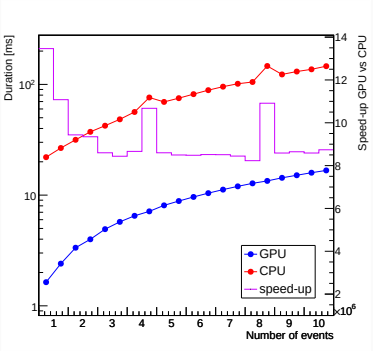


Phase-Space Monte Carl0

GPU vs OpenMP



GPU vs TBB



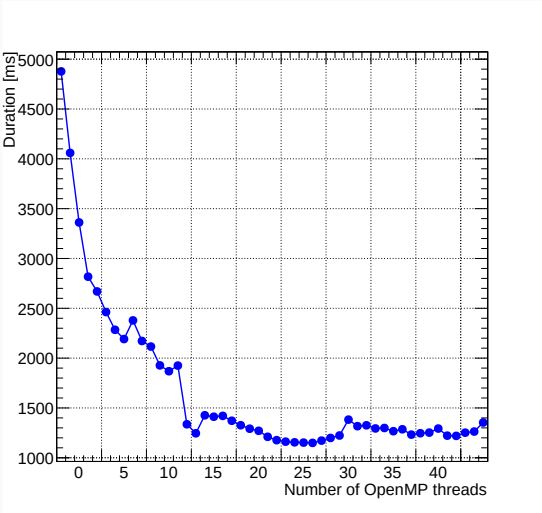
System configuration:

- 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



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 MyFunctor: public hydra::BaseFunctor<MyFunctor,double,N>
2  {
3  ...
4  // implement the Evaluate() method
5  template<typename T> __host__ __device__
6  inline double Evaluate(T* x) { /*actual calculation*/ }
7  };
```

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 I

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()`:

```
1  ...
2  double two = 2.0;
3  //define a simple lambda and capture "two"
4  auto my_lambda = [] __host__ __device__(double* x)
5          { return two*sin(x[0]); };
6  //convert is into a Hydra functor
7  auto my_lamba_wrapped  = hydra::wrap_lambda(my_lambda);
8  ...
```

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

Support for C++11 lambdas II

Now it is possible to add named parameters to C++11 lambdas.

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

//define a simple lambda and capture "two"
auto my_lambda = [] __host__ __device__(size_t n, hydra::Parameter* param, double* x)
        { return param[0]*sin(x[0]); };

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

//set the multiplier to a different value
my_lamba_wrapped.SetParameter(0, 3.0);
...
```

Data containers

- `hydra::Point` represents multidimensional data points.
- 

`hydra::PointVector` looks like an array of structs, but actually data is stored in structure of arrays layout.

```
1  //two dimensional point
2  typedef hydra::Point<GReal_t, 2> point_t;
3
4  //allocate a two dimensional data set on the gpu
5  hydra::PointVector<point_t, hydra::cuda::sys> data_cuda(1e6);
6  ...
7  //process data_cuda
8  ...
9  //copy the data to the CPU memory space
10 hydra::PointVector<point_t, hydra::host::sys> data_h(data_d);
```

Vegas-like multidimensional numerical integration

The VEGAS algorithm implemented in Hydra now supports training 


```
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);
```

Phase-Space Monte Carlo

- New version is about 4x faster.
- 

Method `hydra::PhaseSpace::AverageOn(...)` to calculate the average and variance of an arbitrary function over the phase-space.

```
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(Mother_mass, Daughter_Masses);
6  //Allocate the container for the events
7  hydra::Events<3, device> events(ndecays);
8
9  //Generate
10 phsp.Generate(Mother, events.begin(), events.end());
```

Interface to Minuit2

- 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`.
- New! More intuitive API. Not necessary to registry previously the parameters any more.

```
...
//get the FCN
auto fFCN = hydra::experimental::make_loglikehood_fcn(model, some_data);
ROOT::Minuit2::MnMigrad migrad_d(fFCN, fFCN.GetParameters().GetMnState(), strategy);
//-------------------------------------^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^-----------
  ...
```

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);
13
14 //pass the FCN to Minuit2
15 ...
```