GooFit: A GPU interface for MINUIT

Rolf Andreassen, Brian Meadows, Manjula de Silva, Mike Sokoloff
(Physics Department, University of Cincinnati)

Karen Tomko
(Ohio Supercomputer Center)

- **Reminder: How MINUIT works**
- **User-level code**
- **PDF code**
- **Performance**
- **Three levels of users:**
  - **End user**
  - **Advanced user**
  - **Engine developer**

MINUIT

- We have some data, $\vec{x}$, which we believe are drawn from a population described by a model $\mathcal{P}$ with parameters $\vec{\alpha}$. We want to find the values of $\vec{\alpha}$ such that the likelihood is a maximum.

- Given $\vec{\alpha}$, the probability of observing data $\vec{x}$ is

$$P(\vec{x}|\vec{\alpha}) = \prod_i \mathcal{P}(x_i; \vec{\alpha}). \tag{1}$$

which, for reasons of numerical accuracy, we transform to

$$\ln P(\vec{x}|\vec{\alpha}) = \sum_i \ln \mathcal{P}(x_i; \vec{\alpha}). \tag{2}$$

as we seek the parameters which maximise the likelihood.

- Notice that getting the probability of an event usually requires a normalisation integral:

$$\mathcal{P}(x) = \frac{\mathcal{F}(x)}{\int \mathcal{F}(x)\mathrm{d}x} \tag{3}$$

where $\mathcal{F}$ is the probability density function.

- Parallelise using the GPU in two places: Numerical normalisation integrals and sum over event probabilities.

Hello, GooFit: Trivial use case

```
int main (int argc, char** argv) {
  // Variable class stores name, upper and lower limit, optionally
  // number of bins and current error
  Variable* xvar = new Variable("xvar", -5, 5);
  xvar->numbins = 10000;

  // Generate data
  TRandom donram(42);
  UnbinnedDataSet data(xvar); // Stores events
  for (int i = 0; i < 10000; ++i) {
    fptype val = donram.Gaus(0.2, 1.1);
    if (fabs(val) > 5) {--i; continue;}
    data.addEvent(val);
  }

  // Create PDF
  Variable* mean = new Variable("mean", 0, 0.1, -10, 10);
  Variable* sigm = new Variable("sigm", 1, 0.1, 0.5, 1.5);
  // FooThrustFunctor classes are PDF objects.
  GaussianThrustFunctor gauss("gauss", xvar, mean, sigm);
  gauss.setData(&data);

  // PdfFunctor is glue between MINUIT and GooFit.
  PdfFunctor fitter(&gauss);
  fitter.fit();
}
```

Internals of Gaussian PDF

```
#include "GaussianThrustFunctor.hh"

__device__ fptype dev_Gaussian (fptype* evt, fptype* p, unsigned int* indices) {
  fptype x = evt[indices[2 + indices[0]]];
  fptype mean = p[indices[1]];
  fptype sigma = p[indices[2]];

  return EXP(-0.5*(x-mean)*(x-mean)/(sigma*sigma));
}

__device__ device_function_ptr ptr_to_Gaussian = dev_Gaussian;

__host__ GaussianThrustFunctor::GaussianThrustFunctor (std::string n,
                                                       Variable* _x,
                                                       Variable* mean,
                                                       Variable* sigma)
  : ThrustPdfFunctor(_x, n)
{
  std::vector<unsigned int> pindices;
  pindices.push_back(registerParameter(mean));
  pindices.push_back(registerParameter(sigma));
  cudaMemcpyFromSymbol((void**) &host_fcn_ptr, ptr_to_Gaussian, sizeof(void*));
  initialise(pindices);
}
```

Existing functions

- **Simple PDFs: Argus function, correlated Gaussian, Crystal Ball, exponential, Gaussian, Johnson SU, relativistic Breit-Wigner, polynomial, scaled Gaussian, smoothed histogram, staircase function, step function, Voigtian.**

- **Composites:**

  - **Sum,** $f_1 A(\vec{x}) + (1 - f_1) B(\vec{x})$.
  - **Product,** $A(\vec{x}) \times B(\vec{x})$.
  - **Composition,** $A(B(x))$ **(only one dimension).**
  - **Convolution,** $\int_{t_1}^{t_2} A(x - t) * B(t) \mathrm{d}t$.
  - **Map,**

$$F(x) = \begin{cases} A(x) & \text{if } x \in [x_0, x_1) \\ B(x) & \text{if } x \in [x_1, x_2) \\ \quad\ldots & \\ Z(x) & \text{if } x \in [x_{N-1}, x_N] \end{cases}$$

- **Specialised mixing PDFs: Coherent amplitude sum, incoherent sum, truth resolution, three-Gaussian resolution, Dalitz-plot region veto, threshold damping function.**

Performance

- **Fits used for testing:**
  - **Trivial Gaussian fit (with 10 million events).**
  - **"Zach's fit": Extracting the natural line width of the D^{*+}. Binned fit involving a convolution of a Breit-Wigner with the sum of three Gaussians.**
  - **Mixing fit: Time-dependent Dalitz-plot fit to extract $D^0 - \overline{D^0}$ mixing parameters.**
- **Several platforms:**
  - **Cerberus: 2.27 GHz Intel Xeon CPU, Fedora 14**
  - **Cerberus: nVidia C2050 GPU**
  - **Oakley: 2 C2070 GPUs in parallel, RedHat 6.3 (Santiago)**
  - **Starscream: Laptop with nVidia 650M GPU, Ubuntu 12.04**

	Cerberus (CPU)		Cerberus (GPU)		Oakley		Starscream	
Fit	Time [s]	Speedup	Time [s]	Speedup	Time [s]	Speedup	Time [s]	Speedup
Gaussian	78	1	0.35	220	0.21	371	3.1	25
Zach's fit	428	1	6	71	6	71	18.7	23
Mixing fit	24617	1	74	333	-	-	303	81

Data organisation

- **Storing events is easy. Just make One Big Array with events laid end-to-end:**

  `a1 b1 ... z1 | a2 b2 ... z2 | ... | aN bN ... zN`

  Then threads keep track of which event to look at, and PDFs keep track of within-event indices of the observables they depend on.

- **Constraints on how to store fit parameters:**
  - **We must be able to use the same parameter in different PDFs - eg two Gaussians with a shared mean.**
  - **A single PDF <u>type</u> may have an unknown number of parameters. For example, which degree is your polynomial? How many PDFs in your sum or product?**

- **Our solution: Store all parameters in one global array, '`cudaArray`'; the PDFs have indices into that array indicating which parameters they depend on.**

- **How to store the indices? We don't know how many a PDF has.**

- **Recurse the same pattern: Store an array of <u>indices</u>, '`paramIndices`', and then each PDF can be summed up as a function pointer plus an index into `paramIndices`!**

- **So, for each PDF, we store indices in a consistent pattern:**

```
numParams
p_idx1 p_idx2 p_idx3
numObservables
o_idx1 o_idx2 ...
```

- **For a single Gaussian, this looks like so:**

```
(# parameters = 2)
(index of mean = 0) (index of sigma = 1)
(# observables = 1)
(index of x = 0)
```

Hence the mysterious lines in the example:

```
__device__ fptype dev_Gaussian (fptype* evt, fptype* p, unsigned int* indices)
  fptype x = evt[indices[2 + indices[0]]];
  fptype mean = p[indices[1]];
  fptype sigma = p[indices[2]];
```

- **Notice that `evt` is a pointer into an array which stores all the event data:**

```
evt (thread 1)  evt (thread 2)  ...   evt (thread N)
x1 y1 z1        x2 y2 z2        ...   xN yN zN
```

- **The core engine's task in pseudocode:**

```
Calculate event address from thread number and event size
Call function with (event, parameters, start of PDF's index array)
Return logarithm of result
```

- **It is up to the function to interpret the numbers in its index array. In the case of `AddThrustFunctor`, we store triplets of function information: Function index, parameter index, index of weight parameter. Note that these are indices into three different arrays! So loop-over-components code looks like this:**

```
__device__ fptype dev_AddPdfs (fptype* evt, fptype* p, unsigned int* indices)
  int numParameters = indices[0];
  fptype ret = 0;
  fptype totalWeight = 0;
  for (int i = 1; i < numParameters-3; i += 3) {
    fptype weight                  = p[indices[i+2]];
    totalWeight                   += weight;
    unsigned int functionIdx      = indices[i];
    void* functionPtr             = device_function_table[functionIdx];
    unsigned int* functionParams = paramIndices + indices[i+1];

    fptype curr = (*(reinterpret_cast<device_function_ptr>
                    (functionPtr))) (evt, p, functionParams);
    ret += weight * curr * normalisationFactors[indices[i+1]];
  }
}
```

Notice that the `AddThrustFunctor` evaluation does not care which observables its components are looking at; that information is encoded in their index arrays. `AddThrustFunctor` just has to know what part of the global `paramIndices` it should pass to its target functions.

- **None of this is necessary to write user-level code!**
- **A PDF writer needs to know what his particular indices mean, but need not know anything about the core engine.**

Shovelling bytes

- **Data from host to device:**
  - **Parameter and function-pointer indices. Only at initialisation.**
  - **Parameter and normalisation values. Once per MINUIT iteration.**
  - **Events. Do once - unless the dataset is very large.**
- **What shall we do with a large data set?**
  - **Split it up so each part fits in a GPU.**
  - **If available, assign each part to a separate GPU!**
  - **If not, evaluate one part while another is being copied.**

Optimisation; what to do where

- **Three main tasks:**
  - **Decide what parameters to look at next - MINUIT's core algorithm. Always CPU.**
  - **Evaluate per-event PDFs. Always GPU.**
  - **Normalisation integrals. CPU if an analytic expression exists, GPU if done numerically.**
- **Lack of fine-grained profiling makes it hard to track down bottlenecks in execution.**
- **A useful trick for the mixing PDF: Cache the computationally-intensive RBW part of the calculation, which depends on masses and widths of the resonances. Tradeoff: More complicated PDF code.**

Summary and outlook

- **We have a great tool!**
- **We hope we can convince other people to use it.**
- **Still need to work on multiple GPUs, large data sets, fine-grained optimisations.**
- **Source code is available for download:**

  `http://www.physics.uc.edu/~rolfa/GooFit_16Jan2013.tar.gz`