Plan for the day

- **Learn to create new GooFit PDF classes.**
- **Look at some internals of existing PDFs, try to figure out what the devil the coder was thinking.**

Creating new PDF types

- **Another look at `GaussianThrustFunctor`:**

```
__device__ fptype device_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]];

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

__device__ device_function_ptr ptr_to_Gaussian = device_Gaussian;
```

- **Side note: '`fptype`' is just a typedef for `double` - this allows quick switching between `double` and `float` precision.**

Organising the indices

- **The more interesting work is done in the constructor:**

```
__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);
}
```

- **Notice `registerParameter` calls. This function returns the index of the `Variable` being registered, assigning a new index if necessary.**

Observables

- **The independent `Variable _x` was passed to the `ThrustPdfFunctor` constructor - why?**
- **So it can do this:**

```
if (_x) registerObservable(_x);
```

- **Works very similarly to `registerParameter`, but observable indices get added after all the variable parameters, as we did manually in lecture 3.**
- **Why not do this in the `GaussianThrustFunctor` constructor?**
- **Because when this was written, GooFit only supported single-dimensional PDFs, and it never got refactored...**
- **For multiple observables, we do put the second and subsequent `registerObservable` calls in the PDF constructor.**

```
__host__
CorrGaussianThrustFunctor::CorrGaussianThrustFunctor
  (std::string n,
   Variable* _x, Variable* _y,
   Variable* mean1, Variable* sigma1,
   Variable* mean2, Variable* sigma2,
   Variable* correlation)
: ThrustPdfFunctor(_x, n)
{
  registerObservable(_y);

  std::vector<unsigned int> pindices;
  pindices.push_back(registerParameter(mean1));
  pindices.push_back(registerParameter(sigma1));
  pindices.push_back(registerParameter(mean2));
  pindices.push_back(registerParameter(sigma2));
  pindices.push_back(registerParameter(correlation));

  cudaMemcpyFromSymbol((void**) &host_fcn_ptr,
                       ptr_to_CorrGaussian,
                       sizeof(void*));
  initialise(pindices);
}
```

Tricks with constants

- **Many ways to get constants, starting with fixed `Variables`:**

```
Variable* fixed_mean = new Variable("fixed_mean", 0);
Variable* fixed_sigm = new Variable("fixed_sigm", 1, 0.5, 1.5);
fixed_sigm->fixed = true;
```

- **For integer constants, it is convenient to just drop them into the indices array:**

```
// Constructor for single-variate polynomial, with optional zero point.
__host__ PolynomialThrustFunctor::PolynomialThrustFunctor
  (string n,
   Variable* _x,
   vector<Variable*> weights,
   Variable* x0,
   unsigned int lowestDegree)
// (...)
  pindices.push_back(lowestDegree);
```

- **Since it's up to the PDF writer to interpret the indices, we can interpret them any way that's convenient!**
- **For non-integer constants, we can use the `functorConstants` array.**
- **Why is there more than one way to do this? Because using CPU-to-GPU bandwidth to transfer numbers that don't change offends my sense of neatness. Additionally, it is sometimes useful to be able to set some constants without reference to `Variables`, as in `ConvolutionThrustFunctor::setIntegrationConstants`.**

- **`KinLimitBWThrustFunctor` constructor needs some cleanup, but will serve as an example:**

```
pindices.push_back(registerConstants(2));
fptype constants[2];
constants[0] = 1.8645;
constants[1] = 0.13957;
cudaMemcpyToSymbol(functorConstants,
                   constants,
                   2*sizeof(fptype),
                   cIndex*sizeof(fptype),
                   cudaMemcpyHostToDevice);
```

- **`registerConstants` method allocates some room in the `functorConstants` array, and sets the member variable `cIndex`, which we then store in the index array.**
- **To look up the constants, use double-indirection starting with indices:**

```
__device__ fptype device_KinLimitBW (fptype* evt,
                                     fptype* p,
                                     unsigned int* indices) {
  fptype x = evt[indices[2 + indices[0]]];
  fptype mean  = p[indices[1]];
  fptype width = p[indices[2]];
  fptype d0mass = functorConstants[indices[3]+0];
  fptype pimass = functorConstants[indices[3]+1];
```

- **Again, it's up to the coder to interpret his constants. `cIndex` just tells you where they are.**

Quick note on Makefile magic

- **Why are all these PDFs called `FooThrustFunctor`, anyway?**
- **That makes it easy for gmake to pattern-recognise them:**

```
SRCDIR = $(PWD)/FPOINTER
FUNCTORLIST = $(SRCDIR)/ThrustPdfFunctor.cu
FUNCTORLIST += $(wildcard $(SRCDIR)/*ThrustFunctor.cu)
FUNCTORLIST += $(wildcard $(SRCDIR)/*Aux.cu)
HEADERLIST = $(patsubst %.cu,%.hh,$(FUNCTORLIST))
WRKFUNCTORLIST = $(patsubst $(SRCDIR)/%.cu,wrkdir/%.cu,$(FUNCTORLIST))
```

- **Current versions of `nvcc` like to have all CUDA code in One Big File. We like to have separate files for each class. So we cheat by globbing together all the separate files into what `nvcc` expects:**

```
wrkdir/CUDAglob.cu: $(WRKFUNCTORLIST) $(HEADERLIST)
@rm -f $@
@cat $(WRKFUNCTORLIST) > $@

wrkdir/ThrustPdfFunctorCUDA.o: wrkdir/CUDAglob.cu FunctorBase.cu
nvcc $(CXXFLAGS) $(INCLUDES) -I. $(DEFINEFLAGS) -c $< -o $@
@echo "$@ done"
```

- **Hope to improve this with future versions of CUDA.**

Tricks with caching

- **The obvious way to write the `ConvolutionThrustFunctor` evaluation function:**

```
__device__ fptype device_ConvolvePdfs (fptype* evt,
                                       fptype* p,
                                       unsigned int* indices) {
  fptype ret = 0;
  unsigned int modelIdx = indices[1];
  unsigned int modelPar = indices[2];
  unsigned int resolIdx = indices[3];
  unsigned int resolPar = indices[4];
  fptype loBound = functorConstants[indices[5]+0];
  fptype hiBound = functorConstants[indices[5]+1];
  fptype step    = functorConstants[indices[5]+2];
  fptype x0      = evt[indices[2 + indices[0]]];

  fptype fakeEvt[10];
  unsigned int* modelPars = paramIndices + modelPar;
  unsigned int modelXIndex = modelPars[2 + modelPars[0]];
  unsigned int* resolPars = paramIndices + resolPar;
  unsigned int resolXIndex = resolPars[2 + resolPars[0]];

  // integral M(x) * R(x - x0) dx
  for (fptype x = loBound; x < hiBound; x += step) {
    fakeEvt[modelXIndex] = x;
    fptype model = callFunction(fakeEvt, modelIdx, modelPar);
    fakeEvt[resolXIndex] = x - x0;
```

```
    fptype resol = callFunction(fakeEvt, resolIdx, resolPar);
    ret += model*resol;
  }
  ret *= normalisationFactors[indices[2]];
  ret *= normalisationFactors[indices[4]];
  return ret;
}
```

- **This works; but you do a lot of evaluations of the resolution and model functions. Then there's the ugly (and memory-consuming) `fakeEvt` array.**
- **Let's instead cache these numbers, only recalculating if necessary. First we need some space to work in:**

```
// Need multiple working spaces for the case
// of several convolutions in one PDF.
__constant__ fptype* dev_modWorkSpace[100];
__constant__ fptype* dev_resWorkSpace[100];

// Number which transforms model range (x1, x2) into
// resolution range (x1 - maxX, x2 - minX).
// It is equal to the maximum possible value of x0,
// ie maxX, in bins.
__constant__ int modelOffset = 0;
// NB: This should be an array so it can differ
// between components! Get away with it for the
// time being because all our physics fits have
// the same range for all components...
```

```
void ConvolutionThrustFunctor::setIntegrationConstants
  (fptype lo, fptype hi, fptype step) {

  // Copy integration constants to device
  if (!host_iConsts) {
    host_iConsts = new fptype[6];
    cudaMalloc((void**) &dev_iConsts, 6*sizeof(fptype));
  }
  host_iConsts[0] = lo;
  host_iConsts[1] = hi;
  host_iConsts[2] = step;
  cudaMemcpyToSymbol(functorConstants, host_iConsts, 3*sizeof(fptype),
                     cIndex*sizeof(fptype), cudaMemcpyHostToDevice);

  int numbins = (int) floor((hi - lo) / step + 0.5);
  modelWorkSpace = new device_vector<fptype>(numbins);
  // Working spaces are member variables.

  // Calculate number of bins and offset for
  // resolution function cache
  // We will do integral from x1 to x2 of M(x)*R(x - x0) dx.
  // So we need to cache the values of M from x1 to x2, which is given
  // by the integration range. But R must be cached from x1-maxX to
  // x2-minX, and the min and max are given by the dependent variable.
  // However, the step must be the same as for the model, or the binning
  // will get out of sync.
```

```
    Variable* dependent = *(observables.begin());

    // Notice that this will be copied to dev_iConsts
    // for use in integration, not functorConstants!
    host_iConsts[2] = numbins;
    host_iConsts[3] = (host_iConsts[0] - dependent->upperlimit);
    host_iConsts[4] = (host_iConsts[1] - dependent->lowerlimit);

    numbins = (int) floor((host_iConsts[4] - host_iConsts[3]) / step + 0.5);
    host_iConsts[5] = numbins;
    cudaMemcpy(dev_iConsts, host_iConsts, 6*sizeof(fptype),
               cudaMemcpyHostToDevice);
    resolWorkSpace = new device_vector<fptype>(numbins);

    fptype* dev_address[1];
    dev_address[0] = (&((*modelWorkSpace)[0])).get();
    // workSpaceIndex is set by constructor
    // - it equals the number of ConvolutionThrustFunctors
    // already created.
    cudaMemcpyToSymbol(dev_modWorkSpace, dev_address,
                       sizeof(fptype*), workSpaceIndex*sizeof(fptype*),
                       cudaMemcpyHostToDevice);
    dev_address[0] = (&((*resolWorkSpace)[0])).get();
    cudaMemcpyToSymbol(dev_resWorkSpace, dev_address,
                       sizeof(fptype*), workSpaceIndex*sizeof(fptype*),
                       cudaMemcpyHostToDevice);
}
```

- **Now fill up these caches, only recalculating if necessary:**

```
__host__ fptype ConvolutionThrustFunctor::normalise () const {
  // First set normalisation factors to one so we can
  // evaluate convolution without getting zeroes
  recursiveSetNormalisation(fptype(1.0));
  cudaMemcpyToSymbol(normalisationFactors, host_normalisation,
                     totalParams*sizeof(fptype), 0,
                     cudaMemcpyHostToDevice);

  // Next recalculate functions at each point, in
  // preparation for convolution integral
  constant_iterator<fptype*> arrayAddress(dev_iConsts);
  constant_iterator<int> eventSize(1);
  counting_iterator<int> binIndex(0);

  if (model->parametersChanged()) {
    // Calculate model function at every point in integration space
    // Notice very subtle distinction from usual NLL sum to get
    // calculation of bin centers!
    MetricTaker modalor(model, getMetricPointer("ptr_to_Eval"));
    transform(make_zip_iterator(make_tuple(binIndex,eventSize,arrayAddress)),
              make_zip_iterator(make_tuple(binIndex + modelWorkSpace->size(),
                                           eventSize, arrayAddress)),
              modelWorkSpace->begin(),
              modalor);
    model->storeParameters();
  }
```

```
  if (resolution->parametersChanged()) {
    // Same for resolution function.
    constant_iterator<fptype*> arrayAddress2(dev_iConsts + 3);
    MetricTaker resalor(resolution, getMetricPointer("ptr_to_Eval"));
    transform(make_zip_iterator(make_tuple(binIndex,eventSize,arrayAddress2)),
              make_zip_iterator(make_tuple(binIndex + resolWorkSpace->size(),
                                           eventSize, arrayAddress2)),
              resolWorkSpace->begin(),
              resalor);
    resolution->storeParameters();
  }

  // Then return usual numeric integral
  fptype ret = ThrustPdfFunctor::normalise();
  return ret;
}
```

- **Finally change evaluation function accordingly:**

```
__device__ fptype device_ConvolvePdfs (fptype* evt,
                                       fptype* p,
                                       unsigned int* indices) {
  fptype ret     = 0;
  fptype loBound = functorConstants[indices[5]+0];
  fptype hiBound = functorConstants[indices[5]+1];
  fptype step    = functorConstants[indices[5]+2];
  fptype x0      = evt[indices[2 + indices[0]]];
```

```
  int workSpaceIndex = indices[6];

  int numbins = (int) FLOOR((hiBound - loBound) / step + 0.5);

  fptype lowerBoundOffset = loBound / step;
  lowerBoundOffset -= FLOOR(lowerBoundOffset);
  int offsetInBins = (int) FLOOR(x0 / step - lowerBoundOffset);

  // Brute-force calculate integral M(x) * R(x - x0) dx
  for (int i = 0; i < numbins; ++i) {
    fptype model = dev_modWorkSpace[workSpaceIndex][i];

    // Calculate x - x0 in bins.
    int resolIndex = i + modelOffset - offsetInBins;
    fptype resol = dev_resWorkSpace[workSpaceIndex][resolIndex];
    ret += model*resol;
  }

  ret *= normalisationFactors[indices[2]];
  ret *= normalisationFactors[indices[4]];

  return ret;
}
```

Summary

- **To make a new PDF, copy `GaussianThrustFunctor.cu` into `FooThrustFunctor.cu`, write your own device-side evaluation function, and keep careful track of what your indices mean.**
- **Automagic in the Makefile will make your new PDF available.**
- **There are three ways to get constants.**
- **Creating a fake event is ugly, but sometimes a useful trick.**
- **By the careful use of device-side caches, we can make our code *arbitrarily complex*. This great power must be used only for good, ie to speed things up.**