PDF examples: Composite

```
CompositeThrustFunctor::CompositeThrustFunctor
(std::string n, FunctorBase* core, FunctorBase* shell)
  : ThrustPdfFunctor(0, n)
{
  std::vector<unsigned int> pindices;
  pindices.push_back(core->getFunctionIndex());
  pindices.push_back(core->getParameterIndex());
  pindices.push_back(shell->getFunctionIndex());
  pindices.push_back(shell->getParameterIndex());

  // Add as components so that observables and
  // parameters will be registered.
  components.push_back(core);
  components.push_back(shell);

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

```
__device__ fptype device_Composite (fptype* evt,
                                    fptype* p,
                                    unsigned int* indices) {
  unsigned int coreFcnIndex  = indices[1];
  unsigned int coreParIndex  = indices[2];
  unsigned int shellFcnIndex = indices[3];
  unsigned int shellParIndex = indices[4];

  // NB, not normalising core function, it is not being used as a PDF.
  fptype coreValue = callFunction(evt, coreFcnIndex, coreParIndex);

  unsigned int* shellParams = paramIndices + shellParIndex;
  unsigned int numShellPars = shellParams[0];
  unsigned int shellObsIndex = shellParams[2 + numShellPars];

  fptype fakeEvt[10]; // Allow plenty of space in case events are large.
  fakeEvt[shellObsIndex] = coreValue;

  // Don't normalise shell either, since we don't know what composite
  // function is being used for. It may not be a PDF. Normalising at
  // this stage would be presumptuous.
  fptype ret = callFunction(fakeEvt, shellFcnIndex, shellParIndex);
  return ret;
}
```

GooFit global variables

- `__constant__ fptype cudaArray[maxParams]`: **Holds fit parameters - in effect** `Variables`.
- `__constant__ unsigned int paramIndices[maxParams]`: **Holds the parameter indices.**
- `__constant__ fptype functorConstants[maxParams]`: **Non-integer constants. Note: The zeroth entry is reserved for the number of events.**
- `__constant__ fptype normalisationFactors[maxParams]`: **Holds the *inverse* normalisation integrals f the PDFs.**
- `__device__ void* device_function_table[200]`: **Function lookup table.**
- `fptype* cudaDataArray`: **Stores events.**
- `int host_callnumber = 0`: **MINUIT iteration - host side.**
- `int totalParams = 0`: **Registered parameters.**
- `int totalConstants = 1`: **Registered constants.**
- `cpuDebug,gpuDebug,debugParamIndex` **- explained below.**

PDF example: Products

```
ProdThrustFunctor::ProdThrustFunctor
  (std::string n, std::vector<FunctorBase*> comps)
  : ThrustPdfFunctor(0, n)
  , varOverlaps(false)
{
  std::vector<unsigned int> pindices;

  for (std::vector<FunctorBase*>::iterator p = comps.begin();
       p != comps.end(); ++p) {
    assert(*p);
    components.push_back(*p);
  }

  getObservables(observables); // Gathers from components

  FunctorBase::obsCont observableCheck;
  // Use to check for overlap in observables

  // Indices stores
  // (function index)(function parameter index)(variable index)
  // for each component.
  for (std::vector<FunctorBase*>::iterator p = comps.begin();
       p != comps.end(); ++p) {
    pindices.push_back((*p)->getFunctionIndex());
    pindices.push_back((*p)->getParameterIndex());
```

```
    if (varOverlaps) continue; // Only need to establish this once.
    FunctorBase::obsCont currObses;
    (*p)->getObservables(currObses);
    for (FunctorBase::obsIter o = currObses.begin(); o != currObses.end(); ++o) {
      if (find(observableCheck.begin(), observableCheck.end(), (*o)) ==
          observableCheck.end()) continue;
      varOverlaps = true;
      break;
    }
    (*p)->getObservables(observableCheck);
  }

  if (varOverlaps) { // Check for components forcing separate normalisation
    for (std::vector<FunctorBase*>::iterator p = comps.begin();
         p != comps.end(); ++p) {
      if ((*p)->getSpecialMask() & FunctorBase::ForceSeparateNorm)
        varOverlaps = false;
    }
  }

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

```
__host__ fptype ProdThrustFunctor::normalise () const {
  if (varOverlaps) {
    // Two or more components share an observable
    // and cannot be separately normalised, since
    // \int A*B dx does not equal int A dx * int B dx.
    recursiveSetNormalisation(fptype(1.0));
    cudaMemcpyToSymbol(normalisationFactors,
                       host_normalisation,
                       totalParams*sizeof(fptype),
                       0, cudaMemcpyHostToDevice);
    // Normalise numerically.
    fptype ret = ThrustPdfFunctor::normalise();
    return ret;
  }

  // Normalise components individually
  for (std::vector<FunctorBase*>::const_iterator c = components.begin();
       c != components.end(); ++c) {
    (*c)->normalise();
  }
  host_normalisation[parameters] = 1;
  cudaMemcpyToSymbol(normalisationFactors,
                     host_normalisation,
                     totalParams*sizeof(fptype),
                     0, cudaMemcpyHostToDevice);
  return 1.0;
}
```

```
__device__ fptype device_ProdPdfs (fptype* evt,
                                   fptype* p,
                                   unsigned int* indices) {
  // Index structure is nP | F1 P1 | F2 P2 | ...
  // where nP is number of parameters, Fs are
  // function indices, and Ps are parameter indices

  int numParams = indices[0];
  fptype ret = 1;
  for (int i = 1; i < numParams; i += 2) {
    int fcnIdx = indices[i + 0];
    int parIdx = indices[i + 1];

    fptype curr = callFunction(evt, fcnIdx, parIdx);
    curr *= normalisationFactors[parIdx];
    ret *= curr;
  }

  return ret;
}
```

Member variables of `FunctorBase`

- `parameters`: **Host-side parameter index for each PDF.**
- `cIndex`: **Host-side constants index - points into `functorConstants`.**
- `normRanges`: **Array storing lower limit, upper limit and step size of normalisation integral.**
- `observables`: **List of independent variables this PDF depends on. Superset of all component observables.**
- `parameterList`: **List of parameters of this PDF.**
- `components`: **List of 'nested' PDFs, eg the factors of a product.**
- `cachedParams`: **Parameter values from previous MINUIT iteration - used to check if, for example, normalisation integrals need to be recalculated.**

Tricks for debugging

- **If you look at the original (unsanitised) versions of the GooFit PDFs, they are full of commented-out statements like this:**

```
/*
  if (0 == blockIdx.x + threadIdx.x)
    printf("Conv debug: %i %i %i %i %i %i\n",
           numOthers,
           indices[8],
           numToLoad,
           workSpaceIndex,
           offsetInBins,
           numbins);
*/
```

- **Clearly, their development is not always entirely smooth. . .**
- **This is very nearly my only trick!**
- `ThrustPdfFunctor` **has a** `setDebugMask(int, bool)` **method, which I occasionally use to switch particular debug statements on and off, eg:**

```
if (gpuDebug & 1) printf("Whatever\n");
```

The method sets global variables `cpuDebug` **and** `gpuDebug`**. Additionally, if its boolean parameter is true, the variable** `debugParamIndex` **is set to equal the parameter index of the PDF object on which the method was called. I use**

this to print debug statements only from a specific PDF in cases when I have several of the same type in my fit; for example:

```
if ((gpuDebug & 1) && (paramIndices + debugParamIndex == indices))
  printf("Whatever\n");
```

- **Sometimes it is helpful to look at error codes; raw CUDA calls generally return an error, eg:**

```
cudaError_t err = cudaMemcpy(target, source, size, cudaMemcpyHostToDevice);
if (cudaSuccess != err) {
  std::cout << "Error near line " << __LINE__
            << " : " << cudaGetErrorString(err)
            << std::endl;
  abort(0);
}
```

- **It can in principle also be helpful to use `cuda-gdb`, which works much like ordinary `gdb`. Compile with `-G` instead of (or in addition to) `-g`.**

PDF examples: Time-dependent mixing

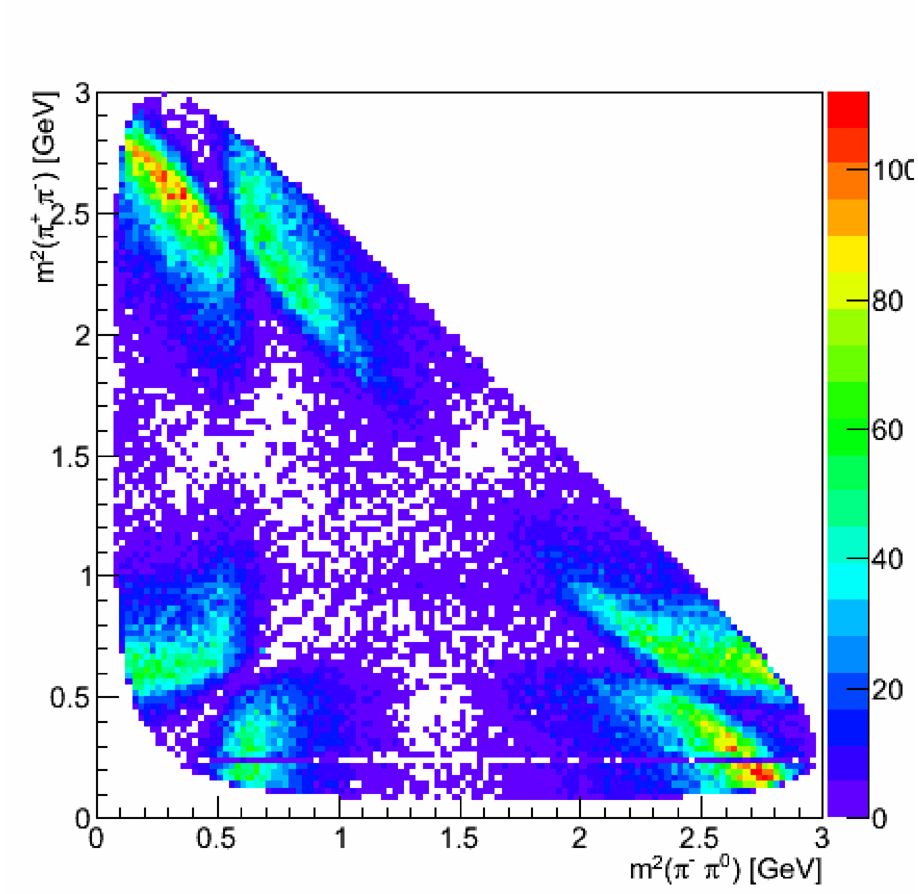
- **This is a very specialised PDF.**
- **Start with merely describing the quasi-two-body resonances that make up the total decay amplitude:**

```
struct ResonanceInfo {
  Variable* amp_real;
  Variable* amp_imag;
  Variable* mass;
  Variable* width;
  unsigned int spin;
  unsigned int cyclic_index;
  unsigned int eval_type;
};
struct DecayInfo {
  fptype motherMass;
  fptype daug1Mass;
  fptype daug2Mass;
  fptype daug3Mass;
  fptype meson_radius;
  Variable* _tau;
  Variable* _xmixing;
  Variable* _ymixing;
  std::vector<ResonanceInfo*> resonances;
};
```

- **The core of the function is a sum over Breit-Wigner resonances:**

$$\boldsymbol{\mathcal{A}(m_1, m_2) \ = \ \sum_{j=0}^{N} a_i e^{i\theta_j} B_j(m_1, m_2; m_j, \sigma_j)}$$

- **Amplitudes $a_j e^{i\theta_j}$ vary quickly; resonance masses and widths (m_j, σ_j) are kept fixed. Breit-Wigners are relatively costly to evaluate.**
- **Obviously: Cache the Breit-Wigner values at every point in the Dalitz plot!**



 Rolf Andreassen University of Cincinnati

Create the cache

- **Now we need to know the data size so the special *wave calculator* knows how far to jump, given that it's on event N.**

```
__host__ void TddpThrustFunctor::setDataSize (unsigned int dataSize,
                                              unsigned int evtSize) {
  // Default 5 is m12, m13, time, sigma_t, evtNum
  totalEventSize = evtSize;
  assert(totalEventSize >= 5);

  if (cachedWaves) {
    delete cachedWaves;
  }

  numEntries = dataSize;
  cachedWaves = new device_vector<WaveHolder>
               (dataSize*decayInfo->resonances.size());
  void* dummy = thrust::raw_pointer_cast(cachedWaves->data());
  cudaMemcpyToSymbol(cWaves, &dummy, sizeof(WaveHolder*),
                     cacheToUse*sizeof(WaveHolder*));
  setForceIntegrals();
}
```

Special normalisation

```
for (int i = 0; i < decayInfo->resonances.size(); ++i) {
  if (!redoIntegral[i]) continue;

  strided_range<device_vector<WaveHolder>::iterator> aragorn
              (cachedWaves->begin() + i,
               cachedWaves->end(),
               decayInfo->resonances.size());

  transform(make_zip_iterator(make_tuple(eventIndex, dataArray, eventSize)),
            make_zip_iterator(make_tuple(eventIndex + numEntries,
                                         arrayAddress, eventSize)),
            aragorn.begin(),
            *(calculators[i]));
  }
}
// End of time-consuming integrals.
```

- **Input iterator is just looping over indices.**
- **Output iterator does a *strided* loop over `cachedWaves`, jumping by the number of resonances every time input iterator jumps by one.**
- **Also cache integrals of each pair of resonances.**

Summary

- **We can write *very complex* PDFs and still get good speedups.**
- **Effort to complexity ratio is similar to RooFit's.**
- **Speed is way better!**
- **Practice makes perfect.**