# Plan for the day

- **Engine internals**
- **Timing tools**
- **Hardware**
- **TODOs**
- **Discussion, wrapup**

# Engine internals

- **Three levels of GooFit code:**
  - **User level: Create `FooThrustFunctor` and `DataSet` objects and use in a fit.**
  - **Development: Write new PDF classes.**
  - **Framework: The core engine that ties together the classes.**
- **Engine code is mainly concentrated in `ThrustPdfFunctor.cu`, with some outliers in `FunctorBase.cu` and `PdfBuilder.cc`.**
- **Following the program flow, let's begin with `PdfBuilder.cc`, containing the class `PdfFunctor`.**

```
set<Variable*> vars;
FunctorBase* pdfPointer = 0;

void PdfFunctor::fit () {
  host_callnumber = 0;
  vars.clear();
  pdfPointer->getParameters(vars);

  numPars = vars.size();
  if (minuit) delete minuit;
  minuit = new TMinuit(numPars);
  int maxIndex = 0;
  int counter = 0;
  for (set<Variable*>::iterator i = vars.begin(); i != vars.end(); ++i) {
```

```
    minuit->DefineParameter(counter, (*i)->name.c_str(),
                            (*i)->value, (*i)->error,
                            (*i)->lowerlimit, (*i)->upperlimit);
    if ((*i)->fixed) minuit->FixParameter(counter);
    counter++;
    if (maxIndex < (*i)->getIndex()) maxIndex = (*i)->getIndex();
  }

  numPars = maxIndex+1;
  pdfPointer->copyParams();

  minuit->SetFCN(FitFun);
  if (0 < overrideCallLimit) {
    std::cout << "Calling MIGRAD with call limit "
              << overrideCallLimit << std::endl;
    double plist[1];
    plist[0] = overrideCallLimit;
    int err = 0;
    minuit->mnexcm("MIGRAD", plist, 1, err);
  }
  else minuit->Migrad();
}

void FitFun(int &npar, double *gin, double &fun, double *fp, int iflag) {
  vector<double> pars;
  // Notice that npar is number of variable parameters, not total.
  pars.resize(numPars);
```

```
  int counter = 0;

  for (set<Variable*>::iterator i = vars.begin(); i != vars.end(); ++i) {
    pars[(*i)->getIndex()] = fp[counter++];
  }

  pdfPointer->copyParams(pars);
  fun = pdfPointer->calculateNLL();
  host_callnumber++;
}
```

# `calculateNLL` function

```
__host__ double ThrustPdfFunctor::calculateNLL () const {
  normalise();
  cudaMemcpyToSymbol(normalisationFactors,
                     host_normalisation,
                     totalParams*sizeof(fptype),
                     0, cudaMemcpyHostToDevice);
  cudaDeviceSynchronize(); // Ensure normalisation integrals are finished

  int numVars = observables.size();
  if (fitControl->binnedFit()) {
    numVars += 2;
    numVars *= -1;
  }

  fptype ret = sumOfNll(numVars);
  if (0 == ret)
    abortWithCudaPrintFlush(__FILE__, __LINE__, getName() + " zero NLL", this);
  return 2*ret;
}

__host__ double ThrustPdfFunctor::sumOfNll (int numVars) const {
  static plus<double> cudaPlus;
  constant_iterator<int> eventSize(numVars);
  constant_iterator<fptype*> arrayAddress(cudaDataArray);
  double dummy = 0;
```

```
  counting_iterator<int> eventIndex(0);
  return transform_reduce(make_zip_iterator(
                           make_tuple(eventIndex,
                                      arrayAddress, eventSize)),
                          make_zip_iterator(
                           make_tuple(eventIndex + numEntries,
                                      arrayAddress, eventSize)),
                          *logger, dummy, cudaPlus);
}
```

# `MetricTaker` functor

- **Created by the `initialise` method:**

```
__host__ void ThrustPdfFunctor::initialise
  (std::vector<unsigned int> pindices, void* dev_functionPtr) {
  if (!fitControl) setFitControl(new UnbinnedNllFit());

  // MetricTaker must be created after FunctorBase
  // initialisation is done.
  FunctorBase::initialiseIndices(pindices);

  functionIdx = findFunctionIdx(dev_functionPtr);
  // Actually a setMetric call
  logger = new MetricTaker(this, getMetricPointer(fitControl->getMetric()));
}
```

- **Constructor takes a `ThrustPdfFunctor` (to extract the function index) and a goodness-of-fit pointer:**

```
MetricTaker::MetricTaker (FunctorBase* dat, void* dev_functionPtr)
  : metricIndex(0)
  , functionIdx(dat->getFunctionIndex())
  , parameters(dat->getParameterIndex())
{

  std::map<void*, int>::iterator localPos =
    functionAddressToDeviceIndexMap.find(dev_functionPtr);
```

```
    if (localPos != functionAddressToDeviceIndexMap.end()) {
      metricIndex = (*localPos).second;
    }
    else {
      metricIndex = num_device_functions;
      host_function_table[num_device_functions] = dev_functionPtr;
      functionAddressToDeviceIndexMap[dev_functionPtr] = num_device_functions;
      num_device_functions++;
      cutilSafeCall(cudaMemcpyToSymbol(device_function_table,
                                        host_function_table,
                                        num_device_functions*sizeof(void*)));
    }
  }
```

- **Operator method calls the device-side function:**

```
// Notice that operators are distinguished by the order of the operands,
// and not otherwise! It's up to the user to make his tuples correctly.
// Main operator: Calls the PDF to get a predicted value, then the metric
// to get the goodness-of-prediction number which is returned to MINUIT.
__device__ fptype MetricTaker::operator () (tuple<int, fptype*, int> t) const
  int eventIndex = thrust::get<0>(t);
  int eventSize  = thrust::get<2>(t);
  fptype* eventAddress = thrust::get<1>(t) + (eventIndex * abs(eventSize));

  // Causes stack size to be statically undeterminable.
  fptype ret = callFunction(eventAddress, functionIdx, parameters);
```

```
  // Notice assumption here! For unbinned fits the 'eventAddress' pointer
  // won't be used in the metric, so it doesn't matter what it is. For
  // binned fits it is assumed that the structure of the event is
  // (obs1 obs2... binentry binvolume),
  // so that the array passed to the metric consists of (binentry binvolume);
  // unless the data has user-provided errors, in which case binvolume is
  // replaced by binError.

  device_metric_ptr metricFcn =
    reinterpret_cast<device_metric_ptr>(device_function_table[metricIndex]);
  ret = (*metricFcn)(ret, eventAddress + (abs(eventSize)-2), parameters);
  return ret;
}

// Operator for binned evaluation, no metric.
// Used in normalisation.
__device__ fptype MetricTaker::operator () (tuple<int, int, fptype*> t) const
  // Bin index, event size, base address [lower, upper, numbins]

  int evtSize = thrust::get<1>(t);
  assert(evtSize <= MAX_NUM_OBSERVABLES);
  int binNumber = thrust::get<0>(t);
  __shared__ fptype binCenters[1024*MAX_NUM_OBSERVABLES];
```

```
  // To convert global bin number to (x,y,z...) coordinates: For each
  // dimension, take the mod with the number of bins in that dimension.
  // Then divide by the number of bins, in effect collapsing so the
  // grid has one fewer dimension. Rinse and repeat.

  unsigned int* indices = paramIndices + parameters;
  for (int i = 0; i < evtSize; ++i) {
    fptype lowerBound = thrust::get<2>(t)[3*i+0];
    fptype upperBound = thrust::get<2>(t)[3*i+1];
    int numBins    = (int) FLOOR(thrust::get<2>(t)[3*i+2] + 0.5);
    int localBin = binNumber % numBins;

    fptype x = upperBound - lowerBound;
    x /= numBins;
    x *= (localBin + 0.5);
    x += lowerBound;
    binCenters[indices[indices[0]+2+i]+threadIdx.x*MAX_NUM_OBSERVABLES] = x;
    binNumber /= numBins;
  }

  // Causes stack size to be statically undeterminable.
  fptype ret = callFunction(binCenters+threadIdx.x*MAX_NUM_OBSERVABLES,
                            cudaArray, parameters);
  return ret;
}
```

# Default numerical normalisation

```
__host__ fptype ThrustPdfFunctor::normalise () const {
  if (!fitControl->metricIsPdf()) {
    host_normalisation[parameters] = 1.0;
    return 1.0;
  }

  fptype ret = 1;
  if (hasAnalyticIntegral()) {
    for (obsConstIter v = obsCBegin(); v != obsCEnd(); ++v) {
      // Loop goes only over observables of this PDF.
      ret *= integrate((*v)->lowerlimit, (*v)->upperlimit);
    }
    host_normalisation[parameters] = 1.0/ret;
    return ret;
  }

  int totalBins = 1;
  for (obsConstIter v = obsCBegin(); v != obsCEnd(); ++v) {
    ret *= ((*v)->upperlimit - (*v)->lowerlimit);
    totalBins *= (integrationBins > 0 ? integrationBins : (*v)->numbins);
  }
  ret /= totalBins;

  fptype dummy = 0;
  static plus<fptype> cudaPlus;
  constant_iterator<fptype*> arrayAddress(normRanges);
```

```
  constant_iterator<int> eventSize(observables.size());
  counting_iterator<int> binIndex(0);

  fptype sum = transform_reduce(make_zip_iterator(
                                 make_tuple(binIndex,
                                            eventSize, arrayAddress)),
                                make_zip_iterator(
                                 make_tuple(binIndex + totalBins,
                                            eventSize, arrayAddress)),
                                *logger, dummy, cudaPlus);

  if (isnan(sum)) {
    abortWithCudaPrintFlush(__FILE__, __LINE__,
                            getName() + " NaN in normalisation", this);
  }
  ret *= sum;

  if (0 == ret) abortWithCudaPrintFlush(__FILE__, __LINE__, "Zero integral");
  host_normalisation[parameters] = 1.0/ret;
  return (fptype) ret;
}
```

# Timing tools

- **Old standby: Look at the clock!**

```
gettimeofday(&startTime, NULL);
fitter.fit();
gettimeofday(&stopTime, NULL);
```

- **For measuring device time more strictly, we have the `cudaEvent` class:**

```
cudaEvent_t start;
cudaEvent_t stops;
cudaEventCreate(&start, 0); // Int is stream index
cudaEventCreate(&stops, 0);

cudaEventRecord(start, 0); // Place start event in execution queue
some_kernel<<<blocks, threads>>>(args);
cudaEventRecord(stops, 0);
cudaEventSynchronize(stop); // Now safe to read results.

float timeUsed = 0;
cudaEventElapsedTime(&timeUsed, start, stop);
```

- **nVidia offers some profiling tools.**
- **We can also compile with profiling info; first set some environment variables:**

```
export COMPUTE_PROFILE=1
export COMPUTE_PROFILE_LOG=cuda_profile_%d.csv
```

```
export COMPUTE_PROFILE_CSV=1
export COMPUTE_PROFILE_CONFIG=./prof_config
```

**and create a profiler configuration:**

```
gridsize
threadblocksize
memtransfersize
memtransferdir
regperthread
dynsmemperblock
stasmemperblock
divergent_branch
local_load
local_store
```

**which tells the profiler which information to store about the kernel. Compile with debug flags:**

```
nvcc -G -g -o myBinary myCode.cu
```

- **Running the executable will now result in a csv file that you can load into Excel or otherwise mess with:**

```
kernel                 cpu time
_Z6kernelPh,911852.312, 911894.000,1000,1000,
            gpu time               grid size
```

```
registers
13,        0.167,
         occupancy
```

- **Can also run in visual mode. Login with `-X` option:**

```
ssh -X username@oakley.osc.edu
qsub -X -I -l nodes=1:ppn=6:gpus=1 -l walltime=01:00:00
module load cuda
nvvp&
```

- **Produces visual timeline and some helpful hints.**
- **Finally, latest versions of GooFit have inbuilt profiling option:**

```
#ifdef PROFILING
__constant__ fptype conversion = (1.0 / CLOCKS_PER_SEC);
__device__ fptype callFunction (fptype* eventAddress, unsigned int functionIdx
  clock_t start = clock();
  fptype ret = (*(reinterpret_cast<device_function_ptr>(device_function_table[
es + paramIdx);
  clock_t stop = clock();
  if ((0 == threadIdx.x + blockIdx.x) && (stop > start)) {
    // Avoid issue when stop overflows and start doesn't.
    timeHistogram[functionIdx*100 + paramIdx] += ((stop - start) * conversion)
  }
  return ret;
}
```

```
#else
// (Regular callFunction method)
#endif
```