Partial Wave Analysis on Graphics Cards: The GPUPWA package

Niklaus Berger

Outline

- Partial Wave Analysis as a computational problem
- Computing on Graphics Processors
- The GPUPWA framework
  - Design considerations and goals
  - Achievements and capabilities
  - To do list
- An example analysis: J/ψ→γKK: a walk through the code
- Advanced features
- Discussion

PWA as a computational problem

- Look at a defined decay channel, e.g. J/ψ→γKK
- Construct a model of the decay in the form of a partial wave decomposition, i.e. a coherent sum over partial waves
- The model will predict the intensity (number of events) I for every point in phase space Ω:

$$I(\Omega) = \left|\sum_{\alpha} V_{\alpha} A_{\alpha}(\Omega)\right|^2$$

where the V_{α} are the unknown production amplitudes (strength and phase of a partial wave)
the A_{α} are the decay amplitudes, including resonance shape (Breit-Wigner etc.) and angular structure

PWA as a computational problem

- We want to determine the V_α by a fit of the model to the data
- Need to construct a Likelihood
- Most straightforward (and computationally most expensive): Event by event likelihood (see 刘北江 s talk for alternatives)

$$\mathcal{L} = \frac{e^{-\mu}\mu^n}{n!}\prod_{i=1}^{n}\frac{I(\Omega_i)}{\int \eta(\Omega)I(\Omega)d\Omega}$$

where n is the number of events and η(Ω) the reconstruction efficiency as a point Ω in phase space

PWA as a computational problem

$$\mathcal{L} = \frac{e^{-\mu}\mu^{n}}{n!}\prod_{i=1}^{n}\frac{I(\Omega_i)}{\int \eta(\Omega)I(\Omega)d\Omega}$$

- How to solve the integral?
- Monte Carlo integration! Can also take into account efficiency η(Ω) by summing only over reconstructed MC events
- Generate lots of events evenly in phase space. Then:

$$\int \eta(\Omega)I(\Omega)d\Omega \approx \frac{1}{N_{gen}^{MC}}\sum_{j=1}^{N_{rec}^{MC}} I\left(\Omega_j\right)$$

PWA as a computational problem

$$\mathcal{L} = \frac{e^{-\mu}\mu^{n}}{n!}\prod_{i=1}^{n}\frac{I(\Omega_i)}{\frac{1}{N_{gen}^{MC}}\sum_{j=1}^{N_{rec}^{MC}} I(\Omega_j)}$$

- Take the negative logarithm of the Likelihood:

$$-\ln\mathcal{L} \propto -\sum_{i=1}^{n}\ln\left(\sum_{\alpha,\alpha\prime} V_\alpha V_{\alpha\prime}^{*} A_\alpha(\Omega_i) A_{\alpha\prime}^{*}(\Omega_i)\right) + \ln\left(\sum_{\alpha,\alpha\prime} V_\alpha V_{\alpha\prime}^{*}\left(\frac{1}{N_{gen}^{MC}}\sum_{j=1}^{N_{rec}^{MC}} A_\alpha(\Omega_j) A_{\alpha\prime}^{*}(\Omega_j)\right)\right)$$

- This is what we want to minimize in the fit

PWA as a computational problem

$$-\ln \mathcal{L} \propto -\sum_{i=1}^{n} \ln\left(\sum_{\alpha,\alpha\prime} V_\alpha V^*_{\alpha\prime} A_\alpha(\Omega_i) A^*_{\alpha\prime}(\Omega_i)\right) + \ln\left(\sum_{\alpha,\alpha\prime} V_\alpha V^*_{\alpha\prime}\left(\frac{1}{N^{MC}_{gen}} \sum_{j=1}^{N^{MC}_{rec}} A_\alpha(\Omega_j) A^*_{\alpha\prime}(\Omega_j)\right)\right)$$

- This needs to be evaluated for every iteration of the fit
- Optimize for speed:

  Precalculate and cache as much as possible

- If the decay amplitudes are left constant in the fit, we can cache the followoing:

$$-\ln \mathcal{L} \propto -\sum_{i=1}^{n} \ln\left(\sum_{\alpha,\alpha\prime} V_\alpha V^*_{\alpha\prime} A_\alpha(\Omega_i) A^*_{\alpha\prime}(\Omega_i)\right) + \ln\left(\sum_{\alpha,\alpha\prime} V_\alpha V^*_{\alpha\prime}\left(\frac{1}{N^{MC}_{gen}} \sum_{j=1}^{N^{MC}_{rec}} A_\alpha(\Omega_j) A^*_{\alpha\prime}(\Omega_j)\right)\right)$$

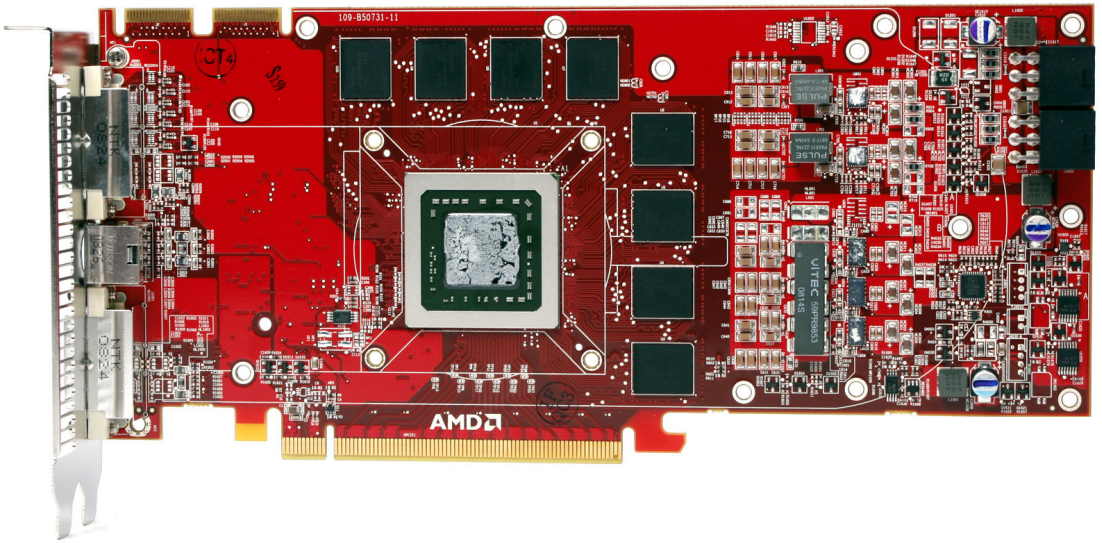
PWA as a computational problem

- This leaves us with the following calculations per iteration:
  - $(N_{wave})^2$ for the Monte Carlo part
  - $(N_{wave})^2 \cdot N_{data}$ for the data part
- If N_{wave} and N_{data} are “big”, this can still be a lot
- For many fitters, we also want to calculate derivatives of the likelihood with regard to fit parameters
- If we allow the decay amplitudes to change, it becomes even worse
- We either have to reduce N_{data} (e.g. by using bins) or calculate very fast
- Here we choose fast calculation

Computing on GPUs

GPUs

- Modern computer games need very powerful hardware to render game scenes in real-time – Graphics Processing Units (GPUs)
- This hardware is cheap because there is a large market
- The capabilities of the hardware increase faster than for CPUs because gamers always want the latest and best



Computing on GPUs

- GPUs can also be used for other things than games: General Purpose Computing on GPUs (GPGPU)
- Keep in mind that GPU architecture is very different from CPU architecture:

CPU	GPU
1-4 cores	Up to 1000 cores
Core can do lots of different things	Core does floating point calculation
Architecture optimized for predicting branches in the code	Architecture optimized for running short programs without branches
Memory access fast because of multi-level cache	Memory access fast because of high bandwith
Low latency, low throughput	High latency, high throughput

Computing on GPUs

- Partial wave analysis is very well suited for GPU computing:
  - Lots of events: can keep many cores busy
  - No branches in the calculation of the likelihood
  - Result of the calculation is a single number: not much bandwith between GPU and CPU needed
  - Access to lookup table similar to access to game textures
  - Use lots of four-vectors, corresponds to 3 colours and transparency
- GPUs are much faster for PWA than CPUs
- Writing GPU code is not as easy as C or FORTRAN

GPU Code

- Code for the GPU implements parallel computing
- Instead of a `for` loop, treating events sequentially, we have:
  - A kernel, specifying the operations on one event
  - Streams of event data
- Stream computing model

- Currently we use the ATI Brook+ stream computing language
- We want to change to the platform independent OpenCL as soon as possible
- You, as a user, should not have to care about this...

The GPUPWA framework

Why a framework?

- Allow easy access to the power of GPUs for all of you
- Develop a shared code base for partial wave analysis
  - More users = more testers; more robust code
  - Write the same code only once
  - Make mistakes only once
  - Share improvements and extensions
  - Share conventions, formats etc.

Some Design Considerations

- Allow the user to write C++ (no GPU specific code)

  Write Object-Oriented code,
  not just a FORTRAN translation

- Encourage declarative user code

  You should state what you want the code to do,
  implementation details handled by the framework

- If you want to change or add to the framework, do it the OO way

  Create derived classes, do not change the original code

- GPUPWA should also serve as an example for documentation

Functionality provided (v1.7)

- Covariant Tensor formalism with tensors up to rank 4
- Handling of in- and output files
- Handling of fit parameters
- MC Integration
- Fits with Minuit and Fumili, including analytical gradients and hessians in the Fumili approximation if needed
- Fits with free resonance parameters with Minuit
- Plotting of projections

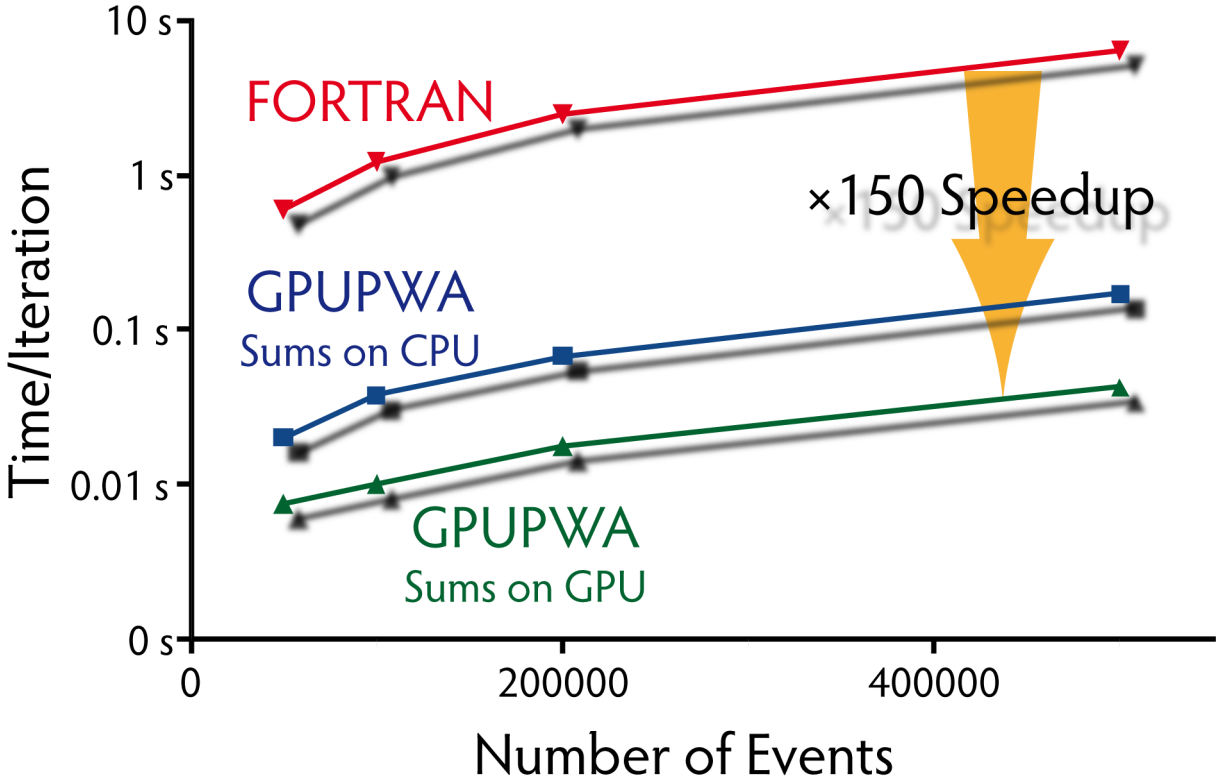
Functionality not yet provided

- Gradients and Hessian in fits with free resonance parameters
- Decay chains (multiple Breit-Wigners in one partial wave)
- Interface with external amplitude calculators (e.g. FDC), possible through text files
- Interface with BOSS to generate MC (will need some help from the BOSS side)

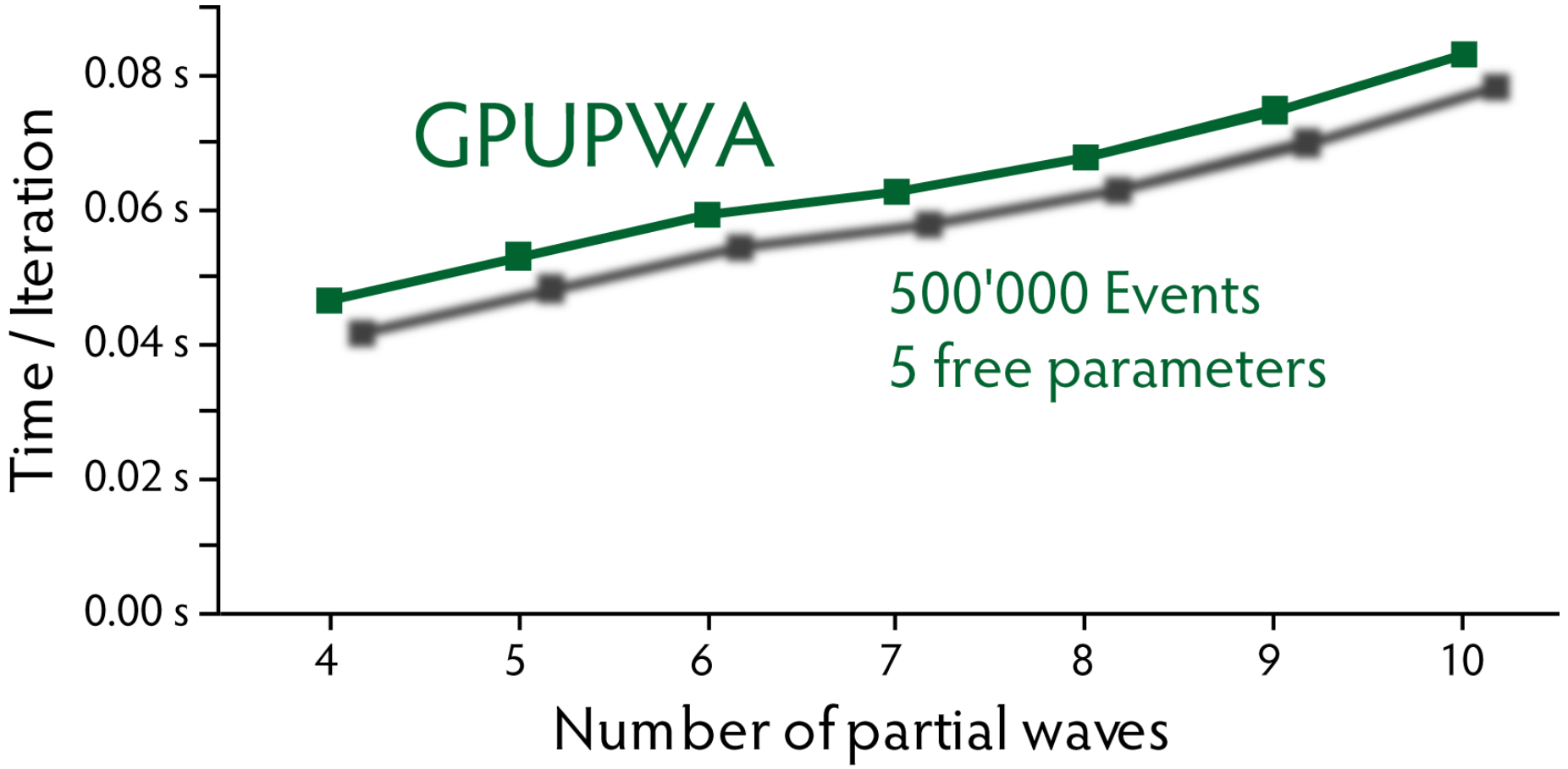
- New things that might come up this week

Performance

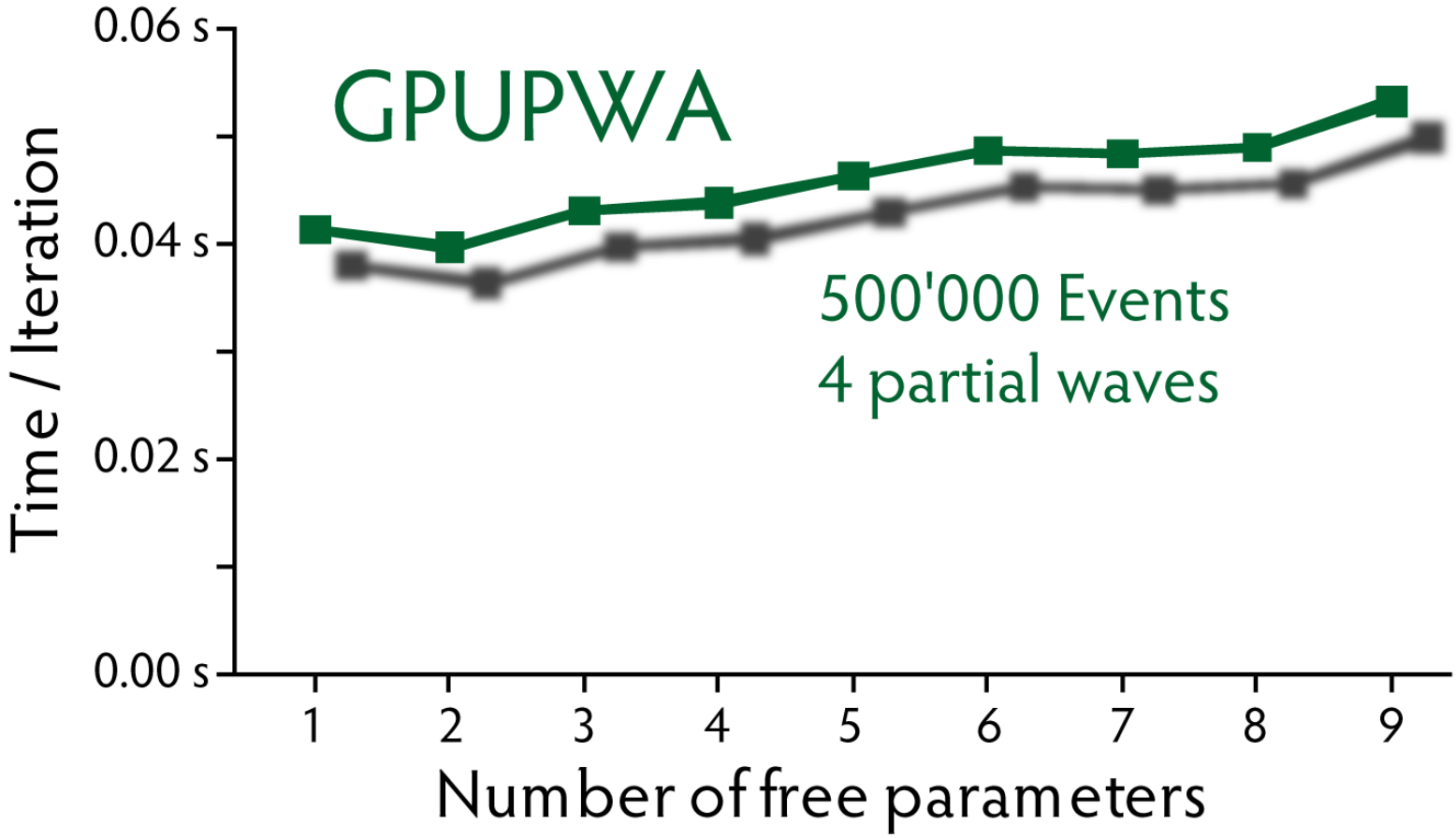
- Best optimized part is the likelihood, gradient and Hessian calculation in the fit: factor 150+ over FORTRAN version



Scaling


Time / Iteration
0.08 s
0.06 s
0.04 s
0.02 s
0.00 s
GPUPWA
500'000 Events
5 free parameters
4
5
6
7
8
9
10
Number of partial waves

Scaling


GPUPWA
500'000 Events
4 partial waves
Time / Iteration
0.06 s
0.04 s
0.02 s
0.00 s
1
2
3
4
5
6
7
8
9
Number of free parameters

Performance Limitations

- Some things cannot be made much faster (reading and writing files, data transfers between GPU and CPU...)
- Some can (and will, if I find the time)
- Main limitation currently is not speed but memory (but much better than early GPUPWA versions)
  - MC samples can be arbitrarily big
  - Data lookup table has to fit into GPU memory (0.5 or 1 GB)
  - For fits with free resonance parameters, both data and MC lookup tables have to fit into GPU memory
- Plotting is also memory-sensitive, but can be done with a subset of the MC

Short Break

- GPUs can do fast PWA
- You can easily use GPUs for PWA with GPUPWA

- That was all a bit general: we will have take a walk through some code next

- Are there any general questions?

An example analysis

- We discuss the J/ψ→γKK analysis
- You also get this code when you check out GPUPWA from CVS

Introduction

```
/// GammaKK.cpp : Example partial wave analysis
/**
Example analysis for the channel J/psi -> gamma KK
Created by Liu Beijiang and Nik Berger
For Questions, please contact nberger@ihep.ac.cn
**/

// Include some headers from the GPUPWA package
// -> should we reduce this to a single header?
#include "../GPUPWA/GPUStreamTensor.h"
...
#include "../GPUPWA/GPUChi2FitConstraint.h"
```

More Headers

```
// We also need some stuff from root
#include "TFile.h"
#include "TRandom3.h"

// And some general C/C++ stuff
#include <ctime>
#include <iomanip>
#include <fstream>
#include <iostream>
#include <string>

using std::cout;
using std::endl;

// Switch on plotting
#define PLOT
```

And here we go

```
// Main program, can be called without any arguments
int main(int argc, char* argv[])
{
	// We will want to do some timing measurements, thus start the clock
	clock_t start = clock();

	// Say Hello to our user
	cout << "Gamma KK partial wave analysis huaning ni!" << endl;
```

Reading resonance parameters

```
// First, we read masses and widths of resonances from a configuration file
ConfigFile resconfig( "res.inp" );
// Values can be read from the file by name
ResCfg res_f2;
resconfig.readInto(res_f2 , "f2" );
cout << "f2 mass width " << res_f2 << endl;
ResCfg res_f0;
resconfig.readInto(res_f0 , "f0" );
cout << "f0 mass width " << res_f0 << endl;
```

Reading amplitude parameters

```
/* And now we read the inital values, error estimates and limits of the fit
parameters from another configuration file */
ConfigFile paraconfig( "para.inp" );
ParaCfg mag_f0,phase_f0;
ParaCfg mag_f20,phase_f20, mag_f21,phase_f21,mag_f22,phase_f22;
ParaCfg mag_bg;

paraconfig.readInto(mag_f0, "f0_mag");
paraconfig.readInto(phase_f0, "f0_phase");
cout << "f0 mag " << mag_f0 << endl;
cout << "f0 phase " << phase_f0 << endl;
...
```

Creating the PWA Object

```
/* Here we start: Create a GPU Partial Wave Analysis Object,
   in this case we are looking at a radiative decay to mesons,
   so it is a Radiative Meson Partial Wave Analysis. The type of the
   analysis determines the Rank of the orbital Tensors
   and the contraction of amplitudes. As parameters we give a name and
   the number of file types we want to use (in this case 2, namely data and the
   phase space Monte Carlo. If you need additional input, e.g.
   different MC for systematic studies, increase that number */
GPURadiativeMesonsPartialWaveAnalysis * myanalysis =
new GPURadiativeMesonsPartialWaveAnalysis("Gamma KK Analysis",2);


// For now we will store and use MC at index 1
myanalysis->SetMCIndex(1);
```

Introduction to the tensor formalism

```
// Next we will start building up the partial wave amplitudes from the particle
     momenta

 /* Convention used: Vectors (lower index) start in small letters, Covectors
     (upper index) with capitals
     For Tensors, the indices are given (T_mn), again capitals for upper indices,
     small letters for lower indices
     The metric Tensor stays the same, so is always called g */

 /* Note that all the following statements will not read files or perform
     calculations. They merey set up a framework such
     that as soon as some calculated quantity is needed, it can be computed using
     the power of the GPU */
```

Input vectors

```
// Input: Four vectors of the two Kaons
/* Here we create two GPUStreamInputRootFileVectors. The constructor
    takes the following arguments: The GPUPartialWaveAnalysis
    they will be used in. This is needed for the caching mechanism to work.
    Then we give the name of the root file for the data,
    the name of the root tree and the names of the branches in the tree
    conatining the momentum compnents and the energy.
    Alternatively, text files can be used*/
GPUStreamInputRootFileVector & k_plus =
    * new GPUStreamInputRootFileVector(myanalysis,
    "data/zeroplustwoplus_data_50k_01.root",
    "t","px1","py1","pz1","E1");
GPUStreamInputRootFileVector & k_minus =
    * new GPUStreamInputRootFileVector(myanalysis,
    "data/zeroplustwoplus_data_50k_01.root",
    "t","px2","py2","pz2","E2");
```

More input files

```
/* Add the filenames for the MC information (it is assumed it is saved at the
   same location in the trees as for the data).
   If you have additional files e.g. for systematics, just add them with a
   higher index */
k_plus.SetFilename("data/zeroplustwoplus_phsp_50k_01.root",1);
k_minus.SetFilename("data/zeroplustwoplus_phsp_50k_01.root",1);
```

Some bookkeeping...

```
/* We can use weights for the data events, e.g. to do a background subtraction.
Here we just set the weights to 1 for all data used*/
 myanalysis->SetEventWeights(1);

 // Set the number of generated MC events;
 // this is just a normalisation constant
 // - here we assume that all MC events have been accepted
 myanalysis->SetNumberMCGen(myanalysis->GetNumberMCAcc());
```

Some constant vectors

```
// Let's do some calculations

// First some constant ingredients: The metric Tensor g_mu_nu
// (we use a -1,-1,-1,1 metric)
GPUMetricTensor & g = * new GPUMetricTensor();

// And the four-vector of the J/psi...
float4 f_jpsi(0.0f,0.0f,0.0f,3.0969f);
// ... in covariant ...
GPUConstVector & jpsi = * new GPUConstVector(f_jpsi);
//  and contravariant form - in this case, the two vectors should be identical ...
GPUConstVector & Jpsi = moveindex(jpsi);
```

And some vector operations

```
// Four-Vector of the intermediate resonance x
GPUStreamVector & x = k_plus + k_minus;

// the rest of the momentum and energy is in the photon
GPUStreamVector & gamma = jpsi - x;
// and again the contravariant form
GPUStreamVector & Gamma = moveindex(gamma);

// We use the '|' charcter to denote contratctions - in the case of
// 4-vectors  such as here, this is the scalar product
GPUStreamScalar & x2 = x|x;
// ... and the square root of it is the mass of the intermediate state
GPUStreamScalar & mX = sqrt(x2);
// and here we contract the gamma and x four-vectors
GPUStreamScalar & xgamma = x|gamma;
```

Photon polarisation

```
// Prepare the g_perp_perp Tensor used in the contratctions
GPUStreamTensor2 & gPerpPerp_mn =
        g - (gamma%x +x%gamma)/(xgamma) +
        x2/(xgamma|xgamma)*(gamma%gamma);

// As this is needed in the contractions, we should tell the analysis object
myanalysis->SetGPerpStream(&gPerpPerp_mn);
```

Orbital Tensor Generators

```
/* Next we create two GPUOrbitalTensors objects. As it is rather tedious to
   calculate orbital tensors from the momenta, we do this for you. Again, you
   need to give the analysis object in order for the caching to work and in
   addition the four-momenta of the mother- and the two daughter particles.
   Here we expect the lower index (covariant) vectors*/

// Orbital tensors for x -> K+K-
GPUOrbitalTensors xorbitals(myanalysis, x, k_plus, k_minus);

// Orbital tensors for J/psi -> gamma x
GPUOrbitalTensors & jpsiorbitals =
    *  new GPUOrbitalTensors(myanalysis, jpsi, gamma, x);
```

Orbital Tensors

```
// Orbital tensors; for f0 we just have the metric tensor
 /* Here we have to cheat a little in order to create a stream object (with a
value for every event) from the constant metric tensor - so we just create a
stream that is one for all events...*/
 GPUStreamScalar & one = *new GPUStreamScalar(myanalysis,1.0f);
 GPUStreamTensor2 & Orbital_f0_MN = g * one;

 // Orbital tensors: three independent ones for f2s
 // Here we get the orbital tensor from the xorbitals object
 GPUStreamTensor2 &  t2_mn = xorbitals.Spin2OrbitalTensor();
 // And do some index lowering and raising gymnastics
 GPUStreamTensor2 &  t2_MN = moveindices(t2_mn);
 GPUStreamTensor2 &  t2_mN = movelastindex(t2_mn);
 GPUStreamTensor2 &  t2_Nm = trans(t2_mN);
```

And more orbital stuff

```
 // The orbital tensor object also conveniently provides barrier factors
GPUStreamScalar & B2_psi_gamma_f2 = jpsiorbitals.Barrier2();

// so we end up with the complete orbital part of the amplitudes
GPUStreamTensor2 & Orbital_f2_0_MN = t2_mn;
GPUStreamTensor2 & Orbital_f2_1_MN =
    -g * (((Jpsi%Jpsi)|t2_mn )* B2_psi_gamma_f2);
GPUStreamTensor2 & Orbital_f2_2_MN =
    (Gamma % (t2_Nm|jpsi))*  B2_psi_gamma_f2;
       /* The '%' charcter denotes the outer product of two tensors - you can
         find the complete catalogue of permitted operations on the GPUPWA
         Wiki*/
```

Propagators

```
/* Next we create the propagators, here assumed to have a Breit-Wigner
   form. Arguments are the mass squared at which it is to be evaluated and
   the resonance mass and width (here taken in a single object from the
   configuration file).*/

GPUPropagatorBreitWigner & propagator1 =
                    * new GPUPropagatorBreitWigner(x2,res_f2);
GPUPropagatorBreitWigner & propagator2 =
                    * new GPUPropagatorBreitWigner(x2,res_f0);
```

Partial Waves

```
// And now we build up partial waves from the orbital an propagator parts, the
// third argument names the wave

// A scalar
GPUTensorPartialWave & wave0 =
    * new GPUTensorPartialWave(Orbital_f0_MN,propagator2,"wave0");

// And a 2+ resonance, with three waves.
GPUTensorPartialWave & wave1 =
    * new GPUTensorPartialWave(Orbital_f2_0_MN,propagator1,"wave1");

GPUTensorPartialWave & wave2 =
    * new GPUTensorPartialWave(Orbital_f2_1_MN,propagator1,"wave2");

GPUTensorPartialWave & wave3 =
    * new GPUTensorPartialWave(Orbital_f2_2_MN,propagator1,"wave3");
```

Adding waves to the analysis

```
/* Add the waves to the partial wave analysis; for every wave added, four
   parameters (magnitude, phase, mass, width) are added to the list of fit
   parameters */
myanalysis->GetWaves()->AddPartialWave(wave1);
myanalysis->GetWaves()->AddPartialWave(wave2);
myanalysis->GetWaves()->AddPartialWave(wave3);
myanalysis->GetWaves()->AddPartialWave(wave0);
```

Initial parameters...

```
 /* Now we set the initial values, error estimates and limits of the parameters
from the configuration file. If the value of the error estimate is negative,
the parameter will be fixed. Limits with a value of 999 will be ignored, i.e.
the parameter will  be considered free in that direction
The 0th parameter is always the magnitude of the phase-space background*/
 myanalysis->SetParameter(0,mag_bg);

 /* Magnitude and phase parameters for the 4 resonances. Note that at least
one magnitude and one phase in the fit have to be fixed, otherwise there is
no unique solution and fits will generally not converge */
 myanalysis->SetParameter(1,mag_f20);
 myanalysis->SetParameter(2,phase_f20);
```

Sharing fit parameters

```
 myanalysis->SetParameter(5,mag_f21);
//myanalysis->SetParameter(6,phase_f21);
myanalysis->GetPartialWave(1)->SetPhaseParameter(2);
myanalysis->FixParameter(6,0);

myanalysis->SetParameter(9,mag_f22);
//myanalysis->SetParameter(10,phase_f22);
myanalysis->GetPartialWave(2)->SetPhaseParameter(2);
myanalysis->FixParameter(10,0);

myanalysis->SetParameter(13,mag_f0);

myanalysis->SetParameter(14,phase_f0);
```

Fix resonance parameters

```
myanalysis->FixParameter(3,res_f2.m);
myanalysis->FixParameter(4,res_f2.w);

myanalysis->FixParameter(7,res_f2.m);
myanalysis->FixParameter(8,res_f2.w);

myanalysis->FixParameter(11,res_f2.m);
myanalysis->FixParameter(12,res_f2.w);

myanalysis->FixParameter(15,res_f0.m);
myanalysis->FixParameter(16,res_f0.w);
```

Perform the MC integration

```
/* Here we perform the preparations for the Monte Carlo calculation.
This will read the MC file, compute and sum all amplitude and interference
terms and write them to a file. This has to be called only once for a constant
set of resonances, as long as their masses and widths are not changed */
myanalysis->MCIntegral();
```

Do the fit

```
  /* Now we can do the fit. Currently you can use either of the following
     fitters:
- FUMILI (the Minuit2 implementation,
- OLDFUMILI (the BES II implementation, in general requires fewest
                    iterations),
- MINUIT (with numerical gradients),
- MINUITGRAD (with analytical gradients,
- MINUITMINOS (MINUIT (numerical gradients) followed by a modified
                        MINOS error estimation)
*/
 myanalysis->DoFit(GPUPartialWaveAnalysis::OLDFUMILI);
```

Differential X-section from MC

```
/*And now we would like to plot some projections for the fit results. For this
  we need event-wise differential x-sections for the Monte Carlo, which we
  are going to generate with the following line. The boolean argument denotes
 whether the interference terms should also be plotted*/

float ** dcs = myanalysis->GetMCDcs(true);
```

Things we want to plot

```
// Create the quantities to be plotted
GPUStreamScalar &ct_g=costheta(gamma);

/* .. yes, we can also rotate and boost vectors - this is of course meaningless
   for covariant amplitudes, but nice for plotting */
GPUStreamVector & kr= lorentzrotation(k_plus,x);
GPUStreamVector & xr= lorentzrotation(x,x);
GPUStreamVector & kb= lorentzboost(kr,xr);
GPUStreamScalar & ct_k=costheta(kb);
GPUStreamScalar & ph_k=phi(kb);
```

Create a set of plots

```
// So first we create a set of plots, which takes care of the formatting and
//file handling
GPUPlotset * plotset = new GPUPlotset();
// we also need the number of active partial waves
int nwaves =  myanalysis->GetWaves()->GetNActiveWaves();
```

Plotting

```
  /* And then we add plots.
 The GPUStreamScalar::Plot() function generates a vector of root histograms
(TH1F). Arguments are:
- Plot name (used for access in root files)
- Plot title (used for displaying). In the title, titles for the x and y axis can be
      given after semicolons
- Number of bins
- Axis low
- Axis high
- Array with the MC differential cross sections
- Number of waves
- Whether or not to plot the off diagonal (interference) elements. If true, the
      dcs array also has to contain  the interference terms */
 plotset->AddPlots(mX.Plot("mass","mass;mX [GeV]",50,1.7,2.3,dcs,
      nwaves,true));
...
```

Formatting and saving plots

```
// Nicely format the plots (root defaults are REALLY UGLY!)
plotset->Format();
// Write a postscript file with the plots. The arguments are currently ignored.
// -> Fix the argument issue
plotset->WritePsfile("testout1.ps",1,1);
// Write a rootfile with the plots
plotset->WriteRootfile("testout1.root");
```

Done!

```
    // Done
    return 0;

}
```

Recap

- Less than 500 lines in GammaKK.cpp, most of them comments and fit parameter handling for a complete PWA
- GPUPWA does all the work (it contains now > 30'000 lines of code, > 300'000 if you include the Brook+ generated code for the GPU)
- This code is not flawless, but many testers help us find many bugs – and it has to be debugged only once
- The code is written in a style so that it describes what you want to do: declarative programming
- Due to inheritance, polymorphism etc. in C++, you can easily change and extend GPUPWA functionality, without changing the core code
- GPUPWA is extremely fast

Other fun things we can do

- Scan a resonance
- Repeat fits with different initial parameters
- Do fits with free resonance parameters
- Constrain a parameter

Scan a resonance

Find out how the likelihood changes, if you change the mass or width of a resonance

Just replace:

```
myanalysis->DoFit(GPUPartialWaveAnalysis::OLDFUMILI);
```

with

```
TGraph * mygraph =
   myanalysis->ScanParameter(GPUPartialWaveAnalysis::OLDFUMILI,
                             15, 2.145,2.155,20);
```

and add the resulting TGraph to the Plotset:

```
plotset->AddGraph(mygraph);
```

Due to a root bug, do NOT fix the corresponding parameter!

Test initial parameters

If you want to make sure, whether your minimum is stable, repet the fit with different initial parameters:

Just replace:

```
myanalysis->DoFit(GPUPartialWaveAnalysis::OLDFUMILI);
```

with

```
myanalysis->DoMultiFit(GPUPartialWaveAnalysis::OLDFUMILI, 5, 20);
```

(vary fits within 5x the error given, do 20 fits) or

```
myanalysis->DoDynamicFit(GPUPartialWaveAnalysis::OLDFUMILI,
                         0.5, 5, 1000);
```

(perform fits until 5 of them have converged within 0.5 of the minimum likelihood)

Fits with free resonance parameters

- To do fits with free resonance parameters, do not fix all of them – the rest is taken care of by GPUPWA (sort of...)
- You have to use MINUIT as a fitter (we do not yet calculate analytical gradients for this case, but this is planned)
- The fit will take longer (we have to recalculate the MC integral for every iteration)
- You might run out of GPU memory with large MC samples

Constraining a parameter

Sometimes we have external knowledge about a parameter with some uncertainty (e.g. a badly measured mass and width from the PDG) that we want to add to the fit. GPUPWA can easily do this:

```
 myanalysis->SetParameter(15,res_f0.m);

myanalysis->SetParameterError(15,0.01);

// Constrain the parameter

GPUChi2FitConstraint * p15constraint =
    new GPUChi2FitConstraint(myanalysis, 15, 2.16, 0.001);

myanalysis->AddConstraint(p15constraint);
```

Additional Resources

- You can find installation instructions and the complete class documentation at:

  http://docbes3.ihep.ac.cn/twiki/bin/view/Main/GpuPwa

- If you find bugs, have questions or suggestions or do not understand the documentation: Ask us!

  Hypernews: http://202.122.32.197/HyperNews/get/pwa.html

  Email: nberger@ihep.ac.cn / Office A406

- Share your contributions with your colleagues: Add your extensions to the CVS repository (tell me beforehand)