Journal of Physics: Conference Series **219** (2010) 042031 doi:10.1088/1742-6596/219/4/042031

Partial Wave Analysis using Graphics Processing Units

Niklaus Berger[1], Liu Beijiang[1,2] and Wang Jike[1]

[1]Institute of High Energy Physics, Chinese Academy of Sciences, 19B Yuquan Lu, Shijingshan, 100049 Beijing, China

[2]Hong Kong University and Chinese University of Hong Kong, c/o Institute of High Energy Physics[1]

E-mail: nberger@ihep.ac.cn

Abstract. Partial wave analysis is an important tool for determining resonance properties in hadron spectroscopy. For large data samples however, the un-binned likelihood fits employed are computationally very expensive. At the Beijing Spectrometer (BES) III experiment, an increase in statistics compared to earlier experiments of up to two orders of magnitude is expected. In order to allow for a timely analysis of these datasets, additional computing power with short turnover times has to be made available. It turns out that graphics processing units (GPUs) originally developed for 3D computer games have an architecture of massively parallel single instruction multiple data floating point units that is almost ideally suited for the algorithms employed in partial wave analysis. We have implemented a framework for tensor manipulation and partial wave fits called GPUPWA. The user writes a program in pure C++ whilst the GPUPWA classes handle computations on the GPU, memory transfers, caching and other technical details. In conjunction with a recent graphics processor, the framework provides a speed-up of the partial wave fit by more than two orders of magnitude compared to legacy FORTRAN code.

1. Introduction

The Beijing Electron-Positron Collider (BEPC) has recently been upgraded to a two-ring machine, providing the world's highest luminosity in the tau-charm energy region. The new Beijing Spectrometer (BES) III was designed and built to make the best possible use of this luminosity and is currently collecting its first major dataset. In the rich physics program of BES III [1], the spectroscopy of light hadrons is one of the most prominent topics. Besides the familiar mesons and baryons, Quantum Chromodynamics in principle also allows for states with valence gluons such as glueballs and hybrids. The search for these states is complicated by mixing and interference phenomena. Decays of charmonium provide an excellent laboratory for attempts to disentangle quark and gluon valence contributions in resonances; it is however required to determine their precise properties, i.e. spin, parity, mass and width. Partial wave analysis (PWA) is the tool of choice to achieve this aim [2]. The huge data sets expected from BES III and the computational demands of PWA fits required new techniques to be employed in order to allow for quick data analysis. In this paper we present a framework for PWA that uses the massively parallel floating point processors on modern graphics cards as accelerators.

2. Partial wave analysis as a computational problem

In a typical PWA (we use the radiative decay $J/\psi \longrightarrow \gamma X, X \longrightarrow K^+K^-$ [3] as an example), the decay is modelled by coherently summing the contributions from a variety of intermediate resonances X. The relative magnitudes and phases of these resonances are determined from a fit and the fit result is compared to the data. The set of resonances and their properties are changed until a sufficient agreement with the data is found.

The intensity I (relative number of events) at a particular point Ω in phase space can be expressed as

$$I(\Omega) = \left| \sum_{\alpha} V_\alpha A_\alpha(\Omega) \right|^2 , \tag{1}$$

where the sum runs over all intermediate resonances α, V_α is the complex production amplitude for α and $A_\alpha(\Omega)$ the complex decay amplitude at a particular point in phase space. The likelihood for a particular model is

$$\mathcal{L} \propto \prod_{i=1}^{N_{Data}} \frac{I(\Omega_i)}{\int \epsilon(\Omega) I(\Omega) d\Omega}, \tag{2}$$

where the product runs over all N_{Data} events in the sample and the integral is proportional to the total cross section, corrected for the detector efficiency $\epsilon(\Omega)$. The integral is usually performed numerically by summing over a large number N_{MC}^{Gen} of simulated events (Monte Carlo, MC) generated evenly in phase space. The limited acceptance and efficiency of the detector can be taken into account by summing only over simulated events that pass the reconstruction and analysis cuts. In a fit, a maximum of the logarithm of the likelihood, corresponding to the best set of parameters for the used model is searched for;

$$\ln \mathcal{L} \propto \sum_{i=1}^{N_{Data}} \ln \left(\sum_{\alpha} \sum_{\alpha\prime} V_\alpha V_{\alpha\prime}^* A_\alpha(\Omega_i) A_{\alpha\prime}^*(\Omega_i) \right) - \ln \left(\sum_{\alpha} \sum_{\alpha\prime} V_\alpha V_{\alpha\prime}^* \left(\frac{1}{N_{MC}^{Gen}} \sum_{j=1}^{N_{MC}^{Acc}} A_\alpha(\Omega_j) A_{\alpha\prime}^*(\Omega_j) \right) \right). \tag{3}$$

The first sum runs over all data events, the second over all MC events. If the widths and masses of resonances are kept constant in the fit (i.e. the V_α's are the only free parameters), the last (inner) bracket and the $A_\alpha(\Omega_i) A_{\alpha\prime}^*(\Omega_i)$ term for each data event can be pre-calculated.

The number of floating point operations required is dominated by the sum over the data events and scales with $N_{iterations} \times N_{data} \times N_{waves}^2$, whilst the lookup table takes up storage space scaling with $N_{data} \times N_{waves}^2$. The storage space problem can be addressed by increasing the memory of the relevant machine ($\approx$1.5 GB are required per million events for a model with 20 partial waves), or with appropriate caching mechanisms for data samples with several million events. If the required floating point operations are performed sequentially, the time required can however become very long. There is no way of knowing whether the fit found just a local or the searched for global maximum of the likelihood. To gain confidence in the result, the fits are usually repeated with various sets of starting parameters. In addition, various models have to be tried out, especially in the study of possible new resonances and systematic effects. The thousands of fits needed in a typical partial wave analysis should thus be as fast as possible, especially as feedback from the physicist is required between the fits and they thus cannot easily be ran in parallel.

3. General purpose computing on graphics processing units

The large market for computer games has driven and continues to drive the development of powerful graphics processing units. In general, the CPU calculates the position of game objects

Table 1. Similarities between typical game graphics computations and PWA computations.

Graphics	PWA
Many independent pixels	Many independent events
Three colour and a transparency value	Lorentz vectors
Very floating point intensive	Very floating point intensive
Textures as lookup tables	Amplitude lookup tables
Few or no branches in shader code	No branches in PWA matrix multiplication

(consisting of many triangles) and the player viewpoint and forwards this information to the graphics processing unit (GPU). The GPU then first performs a projection to the screen view port and determines which triangles are visible to the player. These triangles are then assigned colour and transparency values by projecting so called textures onto them. In the quest for ever increasing realism, the last step now includes running small programs (*shaders*) for every pixel, which allows for various effects such as shadows or structure on the triangles making up the game objects. With the large number of pixels displayed by modern screens and the complexity of today's game worlds, powerful hardware is needed to run enough of theses shaders to produce an adequate frame rate.

Efforts to make this hardware available for general purpose computing are an obvious next step. Early GPGPU (general purpose computing on GPUs) efforts tried to transform the computational problem at hand to the rendering of an image and used graphics interfaces such as *OpenGL* [4]. The major GPU vendors and independent developers quickly realized the potential of GPGPU computing and started to provide frameworks giving direct access to the computing capabilities of the hardware, such as Nvidia's *CUDA* [5] or AMD/ATI's *Brook+* [6], which is in turn based on the Stanford University *Brook* project [7]. The *OpenCL* [8] (Open Computing Language, not to be confused with the *OpenGL* graphics language) is a hardware independent standard for data parallel computing created with the involvement of all major industry players. At the time of writing however, no implementation had been released.

It turns out that high energy physics applications, partial wave analysis in particular, have

```
float data[nevents];
float result[nevents];
...
for(index in nevents){
   result[index] = sqrt(data[index]);
}
```

```
kernel root(stream data,
            out stream result){
result = sqrt(data);
}
...
float4 data<nevents>;
float4 result<nevents>;
data.read(indata);
root(data, result);
result.write(outresult);
```

Figure 1. C code acting on an array of data (left) and the corresponding *Brook+* code (right). Note that GPU-optimised code will in general act on 4-tuples of floating point values (`float4`) and the data has to be explicitly transferred to (`data.read()`) and from (`data.write()`) the GPU memory.

```
...
GPUMetricTensor & g = *new GPUMetricTensor();
GPUStreamVector & f2 = k_plus + k_minus;
GPUOrbitalTensors f2orbitals(f2, k_plus, k_minus);
GPUOrbitalTensors psiorbitals(psi, gamma, x);
GPUStreamTensor2 & t_f2 = f2orbitals.Spin2OrbitalTensor();
GPUStreamScalar & B2 = psiorbitals.Barrier2();
GPUStreamTensor2 & U_f2 = g * (psi%psi)*t_f2 * B2;
...
```

Figure 2. GPUPWA code for the orbital part of an amplitude for the $J/\psi \longrightarrow \gamma f_2, f_2 \longrightarrow K^+K^-$ process: $U^{\mu\nu}_{(\gamma f_2)2} = g^{\mu\nu} p^{\alpha}_{\psi} p^{\beta}_{\psi} \tilde{t}^{f_2}_{\alpha\beta} B_2(Q_{\psi\gamma f_2})$, where $g^{\mu\nu}$ is the metric tensor, p^{α}_{ψ} the ψ four-momentum, $\tilde{t}^{f_2}_{\alpha\beta}$ the spin two orbital tensor and $B_2(Q_{\psi\gamma f_2})$ the appropriate Blatt-Weisskopf barrier factor. The `%` operator is used to denote the outer product.

many features in common with the calculations performed in 3D computer games, see table 1. The large memory bandwidth required to access the lookup tables and the branch-free, floating-point operation intensive code together with the large event numbers make GPUs an architecture better matched to the task at hand than e.g. multi-core CPUs.

As AMD/ATI was the first manufacturer providing double precision capable hardware, we choose the *Brook+* framework for development. *Brook+* implements a stream computing model, where essentially the contents of *for*-loops are replaced by kernels acting on the array of data represented in a stream; see figure 1 for example code. Currently we develop and test our code on workstations equipped with ATI Radeon 4870 GPUs, which provide 800 parallel floating point units (160 for double precision) for a theoretical performance above 1 TFlop/s at a price around 200 US$ for the card.

4. The GPUPWA framework

With the creation of a framework for partial wave analysis at BES III we intended to make the power of GPUs available to users without experience in GPU programming and at the same time create a reusable, documented, shared code base in C++ replacing the legacy FORTRAN programs. The framework was not very imaginatively christened GPUPWA.

4.1. General design considerations

GPUPWA was designed to allow GPU accelerated PWA without exposing the GPU specifics (i.e. writing *Brook+* code) to the users. The framework is inspired by the stream computing model; the user code basically describes the operations on a single event (a kernel), the parallelisation (and memory and cache management) is hidden from the user. Whilst the core functionalities of tensor manipulation and likelihood fits require custom GPU code, we rely on standard tools (namely *ROOT* [9] and its implementations of Minuit [10] and Fumili [11]) for tasks such as minimisation, I/O and plotting. The code is self-documenting using the *Doxygen* tool.

4.2. Covariant tensor manipulation

Traditionally the most tedious part of writing a PWA program is coding the amplitudes in some formalism (we use the covariant tensor formalism) starting from a theory paper (e.g. [12], [13], [14]). GPUPWA allows the writing of amplitudes almost as on paper (see figure 2). Through operator overloading, function objects are created which perform the required operation on the data or a part of the data when their `operator()` method is called. These function objects

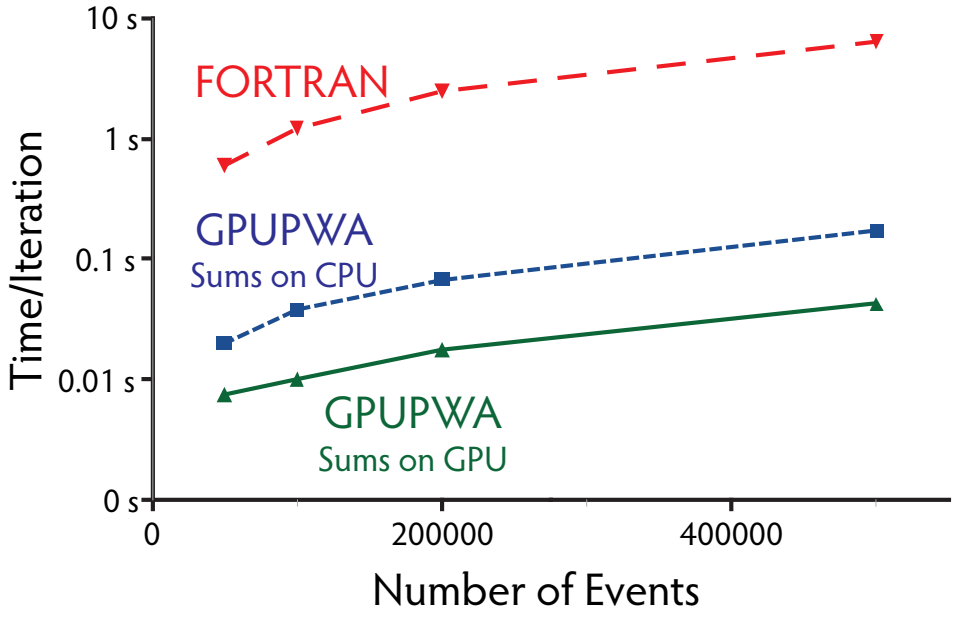

Figure 3. Execution times for one fit iteration of a $J/\psi \longrightarrow \gamma K^+K^-$ PWA with four waves and five free parameters for the legacy FORTRAN code ($\bigtriangledown$), an older GPUPWA version where the sums were performed on the CPU ($\square$) and GPUPWA performing the sums on the GPU ($\bigtriangleup$) versus the number of events used in the fit.

also implement a caching mechanism, which stores intermediate results until all the objects downstream in the calculation have made use of them. In C++, the tensor handling code can be written rather; on the GPU (accessed via *Brook+*) however, there is no mechanism for dynamic memory allocation or template support, so for every possible operation, a specific kernel has to be called[1]. The transition from the generic user side C++ to the specialized *Brook+* kernels is performed via template specialisation, thus allowing to check for illegal or non-implemented operations at compile- rather than at run-time. GPUPWA also provides optimized code for frequently used objects in the covariant tensor formalism such as orbital tensors and barrier factors.

4.3. Partial waves

GPUPWA treats a partial wave as consisting of a (possibly higher rank, but real) orbital part and a (complex, scalar) resonance shape, e.g. a Breit-Wigner. Each partial wave has associated fit parameters. A partial wave analysis object is a collection of partial waves that calls calculator classes to perform the Monte Carlo integration, fits and project the model to the data and generate the corresponding plots.

4.4. Likelihood fits

GPUPWA interfaces to the standard `ROOT::Minuit2` minimizers as well as to a minimal version of Fumili employed for the BES II analyses, all running on the CPU. Depending on the minimizer used, GPUPWA will calculate the logarithm of the likelihood, its gradients with regard to all free parameters and the Hessian matrix in the Fumili approximation for every fit iteration, almost

[1] Specifically, we have to differentiate between objects that are the constant for all events, such as the metric tensor $g`\mu\nu$ and objects that take different values for each event, such as particle momenta.

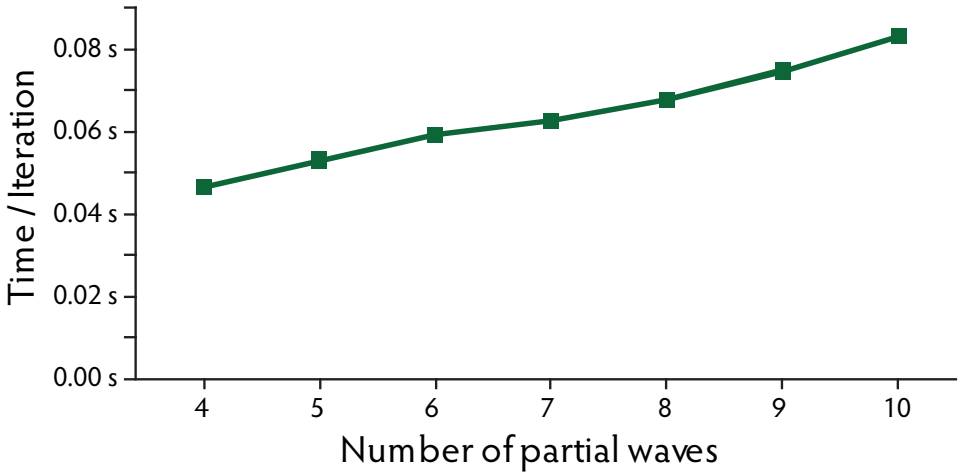

Figure 4. Execution times for one fit iteration of a $J/\psi \longrightarrow \gamma K^+K^-$ PWA with 500'000 events and five free parameters in GPUPWA versus the number of partial waves used in the fit.

completely on the GPU. Traditionally this is the most time-consuming part in any partial wave analysis program; we have thus taken great care to optimize this part of the code. The largest gains in speed were obtained by minimizing the number of data transfers over the PCIe-bus between main memory and GPU memory. In the current implementation the bottleneck is the access by the GPU to the lookup tables in GPU memory; it seems likely that optimizing memory access patterns in this area could produce an additional large speed-up. Efforts in this direction are currently hampered by the lack of profiling and debug tools.

The AMD/ATI GPUs work fastest on four-tuples of single precision floating point values, so most of the algorithms are implemented in single precision. As an input to the minimisers, single precision is however often not sufficient. Consequently all the sums over events are calculated in double precision. For a sufficiently large number of events, this mixed precision approach delivers results comparable to a complete double-precision implementation at much higher speed.

5. Performance

All the performance measurements quoted here were performed on an Intel Core 2 Quad 2.4 GHz workstation with 2 GB RAM running under Scientific Linux 5.2. The GPU is a ATI Radeon 4870 with 512 MB RAM. We compared a FORTRAN implementation used in BES II and the GPUPWA implementation of a $J/\psi \longrightarrow \gamma K^+K^-$ partial wave analysis, both minimizing using Fumili with analytical gradients and Hessian. The time for one fit iteration was measured with the system timer for various sizes of the data samples (see figure 3). For large enough samples, the GPUPWA code is more than 150 times faster than the FORTRAN code.

The execution times of the GPUPWA code also show benign behaviour when increasing the number of partial waves (figure 4) or the number of free parameters in the fit (figure 5). This shows that the speed is limited by memory accesses and data transfers rather than raw floating point power – in fact we reach about 9 GFlop/s (counting also transcendentals and double precision operations as 1 Flop), below 1% of the theoretical power of the card.

6. Outlook

GPUPWA enjoys a growing community of users within BES III. We continuously fix identified bugs and add functionality desired by the users without compromising the speed of execution.

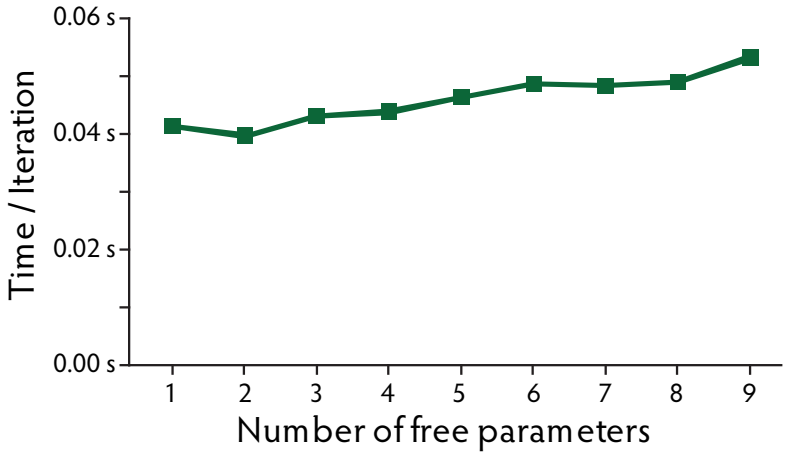

Figure 5. Execution times for one fit iteration of a $J/\psi \longrightarrow \gamma K^+K^-$ PWA with four waves and 500'000 events in GPUPWA versus the number of free parameters in the fit.

As soon as a Linux implementation of *OpenCL* becomes available, we plan to port GPUPWA to this hardware-independent standard. In the meantime, the capabilities of GPUs and the number of floating point units provided keep improving at high speed. Their architecture tailored to four-vectors should also make them great tools for many other applications in high energy physics.

Acknowledgments
We would like to thank the partial wave analysis group at BES III (the users of GPUPWA) for their efforts in testing the framework, their very helpful suggestions and bug reports.

The work presented was supported by the Chinese Academy of Sciences and the Swiss National Science Foundation.

Refernces

[1] Asner D M *et al.* 2008 Physics at BES-III (*Preprint* `arXiv:0809.1869 [hep-ex]`)
[2] Peters K J 2006 *Int. J. Mod. Phys.* A **21** 5618–5624 (*Preprint* `hep-ph/0412069`)
[3] Bai J Z *et al.* (BES) 2003 *Phys. Rev.* D **68** 052003 (*Preprint* `hep-ex/0307058`)
[4] Segal M and Akeley K 2008 The OpenGL graphics system: A specification (version 3.0) The Khronos Group URL `http://www.opengl.org/registry/doc/glspec30.20080811.pdf`
[5] Halfhill T R 2008 *Microprocessor Report* **01/28/08** 1–6
[6] Advanced Micro Devices 2007 AMD stream computing, software stack AMD white paper URL `http://ati.amd.com/technology/streamcomputing/firestream-sdk-whitepaper.pdf`
[7] Buck I 2003 Data parallel computing on graphics hardware Presentation at Graohics Hardware URL `http://graphics.stanford.edu/projects/brookgpu/GH03-Brook.ppt`
[8] Aaftab M 2008 The OpenCL specification, version 1.0 The Khronos Group URL `http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf`
[9] Brun R and Rademakers F 1997 *Nucl. Instrum. Meth.* A **389** 81–86
[10] James F and Roos M 1975 *Comput. Phys. Commun.* **10** 343–367
[11] FUMILI CERN Prorgram Library, D510
[12] Zou B S and Bugg D V 2003 *Eur. Phys. J.* A **16** 537–547 (*Preprint* `hep-ph/0211457`)
[13] Dulat S, Zou B S and Wu J M 2004 (*Preprint* `hep-ph/0403097`)
[14] Dulat S and Zou B S 2005 *Eur. Phys. J.* A **26** 125–134 (*Preprint* `hep-ph/0508087`)