Summer Student Program
at Joint Institute for Nuclear Research

Laboratory of information technology

GPU acceleration of the computation of cross sections of particle processes in high energy physics interactions

Marius Hromnik

Supervisors: Alexander Ayriyan, Ovik Grigorian

Dubna 2015

Contents

Abstract

The computation of cross sections of processes in high energy physics often requires one to solve complex systems of multidimensional integrals numerically. Traditionally, the approach to solving multidimensional integrals is to use Monte Carlo integration using an effective sampling technique to try and improve the error estimates of this non-deterministic approach. In this project the deterministic approach to the solution of the equation computing the pion damping width from the SU(2) x SU(2) Nambu-Jona_Lasinio model.

Computing multidimensional integrals directly using a quadrature scheme such as a trapezoidal method or a Gaussian integration scheme is computationally prohibitive using the traditional approach. It is an $O(n^m)$ calculation for the m dimensional integral and to obtain any kind of accuracy, at least 64 points must be sampled and it is clear that the computation explodes. In this project the problem was tackled using the machinery of the graphics processing unit (GPU) which offers orders of magnitude higher processing performance for suitable problems than CPU-based solutions. Using the Nvidia Compute Unified Device Architecture (CUDA) to control the graphics card hardware, one has access to what is these days referred to as personal supercomputing – TFLOPs of computing power and orders of magnitude faster memory transactions. The problem of solving the pion damping with of a pion in a hot pion gas is well suited to GPU computing. As the problem is commonly attacked using distributed computing systems, so the solution presented here scales up naturally to computing clusters equipped with multiple GPUs per blade such as the Hybrid Computing Cluster at the Joint Institute for Nuclear Physics.

Acknowledgement

I would like to thank Alexander Ayriyan and Ovik Grigorian, my supervisors during my time at the Joint Institute of Nuclear Research. Sasha is an open minded talented scientist and his guidance during my time at the institute was invaluable. The operating procedure at this prestigious institution in Russia is quite different to what I am used to and Sasha bridged the gap seamlessly providing the ideal environment to work productively in. In every aspect of the problem we tackled that I struggled with, his expertise allowed obstacles to be surmounted without delay.

Ovik Grigorian has a lot of experience in the field of High Energy physics, as well as physics in general. With his insight, we were able to tackle a problem very relevant to the field today, rather than a demonstration of some kind. Discussions with Ovik were inspiring and reminded me very much of a professor in my home university department, whose understanding of physics has always been an inspiration.

Discussions with Gary Tupper from the department of Physics at UCT were invaluable to getting some understanding of the QCD theory used in this project. My expertise is in computing, and Gary's open door policy and perpetual willingness to provide explanations to my questions helped to make this project a very enjoyable one for me.

I would like to thank the director of University Centre of Joint Institute for Nuclear Research, S. Z. Pakuliak, for giving me an opportunity to participate in Student Summer Program and for providing the funding for this project.

Introduction

Computing

With the advent of high powered vector processing units, the possibilities for supercomputing, both personal and large-scale distributed supercomputing, exploded. TFLOPS (trillions of floating point operations per second) became available to shared memory vector processing architectures and PFLOPS became available to distributed vector processing architectures. High Energy physics has provided the motivation for the development of computing capabilities over the decades. The IBM Blue Gene architecture is a famous one of these high performance computing architectures developed originally for solving large-scale grid-based physical problems such as Lattice Quantum chromodynamics, and the host of Computational Fluid Dynamics (CFD) problems.

In recent years the idea of using graphics processing units to solve physical problems has been realized with a huge success. Since the 1980s GPUs have been used in bitmap operations - mathematical functions applied to determine the colour value of each pixel in a visual display. Herein lies the heart of vector processing units. A vector processing unit applies the same instruction to multiple units of data simultaneously. In 2001, with the release of the Nvidia Geforce 3 series of graphics cards and the Microsoft DirectX 8.0 standard that required that pixel and vertex shading pipelines be programmable, the task of phrasing a physical problem in such a way that a GPU could solve it, became easier, opening the field to a wider variety of problems.

In a visual display, millions of pixels need to be rendered simultaneously at least as often as 75 times every second. Consider a 1080p Full HD monitor with a response time of 5ms which equates to 2.1M pixels refreshed at up to 200 times per second. This cannot be done by a CPU serially, and is achieved by having an assistant processor, a GPU vector processing unit that can assign many (millions) of threads to compute (calculate the RGBA value) a single pixel each. As the architecture is intrinsically parallel, there is no overhead cost of spawning or collapsing these threads. The architecture is divided into cores that interact with memory, each containing several logic units and registers for performing operations on each thread. More about the hardware will be discussed in the explanation of the GPU implementation.

The peak performance of a Kepler K40, the GPUs installed in the Laboratory of Information Technology's Hybrid Cluster at JINR, is quoted at 4.3 – 5 TFlops single precision. This frighteningly high number arises from 2880 cores clocked at around 810MHz. The peak performance on a GPU can only be obtained if several conditions are met. These include that each logic unit has the data it must compute available in registers at its disposal without having to wait, and that each logic unit is kept busy at all times. To address the memory issue, GDDR5 RAM was constructed with a throughput of 288GB/s, about 17x higher than the fastest DDR3 2133Mhz SD RAM.

Physics

To calculate the pion damping width of a pion in a hot pionic gas an approach is to use the Nambu-Jona-Lasinio (NJL) model. This is an effective model of calculating QCD (Quantum chromodynamics) point interactions. The behavior of the pion in a hot pionic gas has been studied in NJL type models [2 – 5] within the mean-field approximation [6 – 7]. The work in the project is to compute the correction to the pion width under extreme conditions due to collisions, as shown by Blaschke, Volkov and Yudichev [1].

A charged pion propagating in a hot pionic gas can be taken out of its state by several mechanisms. It decays by the weak interaction but in this project we are only concerned with considering collisions. A pion propagating in a hot pionic gas can be knocked out its state and effectively disappear or it can knock a pion in the gas into its state thus effectively creating a pion. The pion width is calculated using **equation (6)** taken from [1].

$$\tau^{-1} = \Gamma = \int \frac{d^3\mathbf{p}_1}{(2\pi)^3} \int ds_1 v_{\text{rel}} A_\pi(s_1) \times [n_\pi(\mathbf{p}_1, s_1, T)\sigma^{\text{dir}^*}(s; s_1, s_2) - \left(1 + n_\pi(\mathbf{p}_1, s_1, T)\right)\sigma^{\text{inv}^*}(s; s_1, s_2) \qquad \textbf{(6)}$$

As can be seen, the two directions of propagation must be combined. The second pion is taken into the rest frame and one integrates over the momentum of the other pion. $\boldsymbol{A_\pi(s_1)}$ is the spectral function of the pion and Brei-Wigner form of it is given using **Eqn (1)** taken from [1].

$$A(s) = \frac{1}{\pi} \frac{M\Gamma}{(s - M^2)^2 + M^2\Gamma^2}$$

Since $A_\pi(s_1)$ depends on Γ, **Eqn (6)** is an implicit integral equation. The technique used to solve this is iterative. An initial guess for Γ is crunched through the integral and the result is fed back into it until the required precision is reached.

The advantage of solving the equation directly using Gaussian integration, for example, is that this technique produces far more accurate results under normal circumstances. As long as there are no exceptional circumstances such as hidden poles nearby lurking off the real axis, integration by quadrature would be generally used if it were computationally reasonable.

Computational implementation

This multidimensional integral means that we're integrating a function that integrates a function…etc. Using **64**-point Gaussian integration, each function integrated is evaluated at **64** points within the integration boundaries. The result of these 64 evaluations are weighted according to a Gaussian distribution and summed to give the solution to the integral. Since we have **5** nested integrals, each evaluating a different function, we end up with **64** function calls to the outer most function and $\mathbf{64^5}$ function calls to the innermost function. In reality we end up quite a lot more work because the each function does a lot more than just integrating a function inside it. In addition, to get accurate results, some functions are integrated separately over different intervals, which results in doubling the number of function calls to such functions. Since these functions are nested, this propagates down the call stack multiplying the number of subsequent function calls.

Serial approach to *gaussianIntegrate()*

The serial solution to this problem was implemented by Ovik Grigorian. This was done to illustrate the unfeasibility of this approach, but was also necessary to facilitate the production of a parallel solution that could be implemented for the GPU. This approach took on average **38 minutes** to compute a single evaluation of **Eqn (6)**. The serial approach to the Gaussian integration is simply:

Use the integration bounds and pre-computed abscissas to generate positions to evaluate the function

Use pre-computed weights based on a Gaussian distribution to weight the function evaluations

Sum the results

This results in 64 function calls and 63 summations.

Serial approach to evaluating *pionWidthDamping()*

The serial approach to computing **Eqn (6)** is:

Call guassIntegrate() on function F_0()

F_0() calls guassIntegrate() on function F_1() 2x

F_1() calls guassIntegrate() on function F_2() 4x

F_2() calls guassIntegrate() on function F_3() 2x

F_3() calls guassIntegrate() on function F_4() 1x

Each ***gaussIntegrate()*** call has different integration boundaries and the functions {***F_0(),F_1()…F_4()***} are unrelated. Each ***gaussIntegrate()*** spawns 64 calls resulting in ***F_0()*** being called **64** times and ***F_4()*** being called **1.75e10** times. The reason for ***guassIntegrate()*** to be called multiple time on some of the functions is due to the nature of the functions and to preserve accuracy. Note that each function call is a complex operation involving many parameters and complex mathematical operations.

Considerations before implementing and algorithm on the GPU

First a note on how GPU computation works, must be made. A GPU is an entirely separate processor from the CPU. It has its own memory which is far more limited than RAM but is far quicker, it has a lower clock speed than a CPU but it has many cores. While a high-end Intel Xeon may have 8 cores, a high-end commodity graphics cards has well over 2000 cores. This is where the enormous FLOP count arises from. (See Table 1 below for a comparison). The enormous memory bandwidth, the ability of the GPU to put massive amounts of data through it very quickly arises from its ability to transfer large chunks of data in a single cycle as well as at a much high speed. When a function is called to run on the GPU the number of threads spawned is specified. These threads are divided into blocks, called threadBlocks, and are further divided into groups of 32 threads called warps. For the performance of a GPU to be utilized, the following conditions must be preserved.

- All threads in a warp must execute the same instruction at the same time. If this is not the case, the instructions will be executed serially and disagreeing threads will be made dormant.
- threadBlocks are distributed between the cores on the GPU. If there are insufficient threadBlocks to occupy all the cores, which consist of streaming multiprocessors (SMs), some cores will be dormant.
- All threads must have the data they require to operate on available in registers on the streaming multiprocessor executing the thread. A thread is waiting for data from memory, the whole warp it is in is pushed out the queue to wait for the data required. If all warps on a SM are waiting for data, as often happens in memory bound codes, the SM will be dormant.
- If a warp requires more registers than are available to store that data it is operating on, the overflowing data is shuffled to global memory and has to be paged in and out as it is required. Global memory is many orders of magnitude slower than on chip memory and this almost certainly will result in SMs being dormant.
- To get performance from the memory bus, all data read by a warp must be stored in contiguous sequential blocks that are accessed in aligned reads. Hence all threads must read the same amount of data at a time. Eliminating unnecessary data transfer or partially occupied memory words requires care to be put into the data structures used and how they are stored. Clearly random accesses are to be discouraged.
- Threads in different thread blocks cannot communicate directly. Threads in different warps can communicate through a fast on-chip memory called shared memory, but this is limited to 49KB of data per thread block.

As can be seen, GPUs are sensitive device and problems have to be phrased very carefully to utilize their power. In addition to this, they have many limitations of which we will note a few relevant ones:

- The memory on a GPU is limited in size. It ranges between 12GB on a Kepler K40 GPU to 2GB on a Geforce mobile series graphics card.
- Data has to be manually transferred between RAM and the GPU. This is slow because it’s limited by the PCI express bus and is an overhead not seen in CPU computing.
- The dimensions of threads in a thread block are limited to a maximum of 1024 x 1024 x 64 threads but in addition to this, the maximum number of threads in a thread block is 1024 threads. 1024 threads will almost guarantee a shortage of memory registers as the number of registers on a single SM is limited and a thread block executes on a single SM.

Table 1: Comparison between CPU and GPU computing capability (Peak performance)

	CPU (Xeon Haswell E3 V5)	***GPU (Kepler K40)***
Cores #	*18 with 256bit AVX*	*2880*
Memory	*>512GB*	*12GB*
Clock speed	*3.2GHz*	*810MHz*
FLOPS	*700GFLOPS (using AVX)*	*4.3 – 5TFLOPS*
Memory bandwidth	*17GB/s*	*288GB/s*

Parallel GPU approach to *gaussianIntegrate()*

It was decided that 64 threads would be allocated per thread block. That's because of the following reason. A Gaussian distribution is symmetric so the abscissas and weights on either sides of the center are the same. As a result, each thread in the 2 warps composing the threadBlock, reads 1 abscissa and 1 weight per function call and since the data is stored contiguously, each read occurs simultaneously in a single cycle. The summation makes clever use of shared memory. The code for ***gaussianIntegrate()*** becomes: **Note:** No loops are used and each thread has an ID called **tid**.

Each thread computes f(x_{tid}), since 32 threads are computed simultaneously all 64 function calls are executed in 2 function calls.

The results are summed as shown in Fig 1. Hence summation is completed in **$\log_2 64$ = 6 steps**

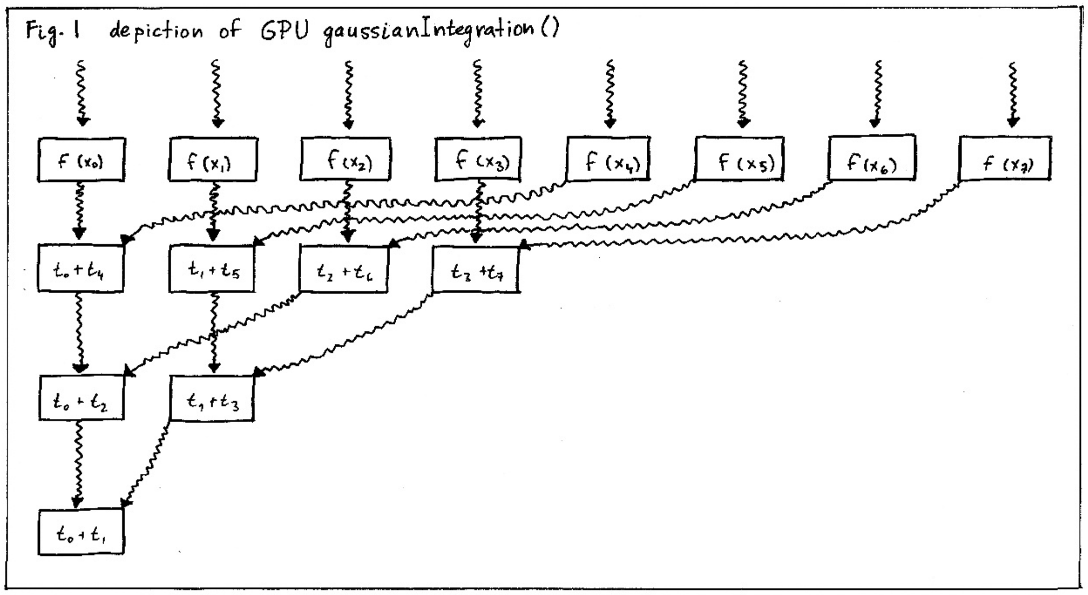

Fig. 1 Illustrates the procedure for the GPU parallel ***guassianIntegrate().*** There are numerous optimization to this algorithm but an explanation of these requires a too in-depth look at the GPU hardware for this report.

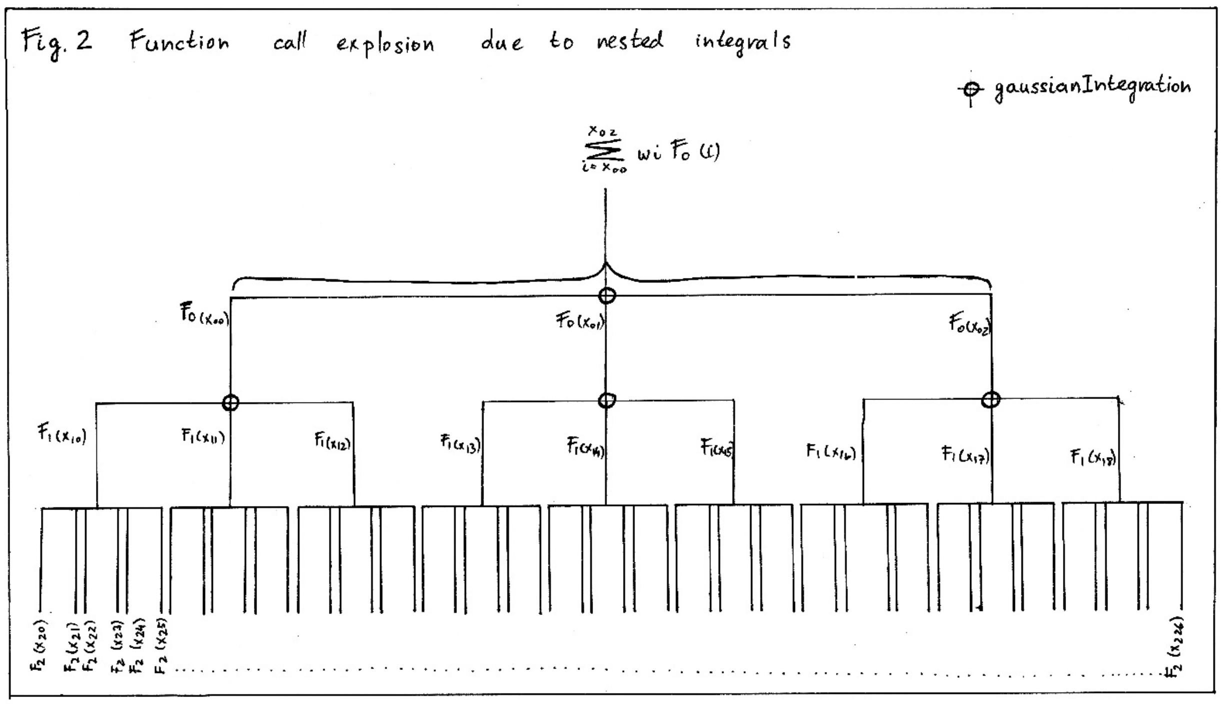

Fig. 2. Illustrates how the number of function calls explodes when pionWidthDamping() is called.

Parallel approach to evaluating *pionWidthDamping()*

Fig. 2 shows 3 way branching, rather than 64 way branching but the principle is the same.

Approach 1

The simplest approach, as implemented in the serial solution, does not work. If **64** threads were spawned in a function call ***pionWidthDamping()*** each thread would have so much work to do as the program proceeds down the tree that it would run out of resources. In addition to this, the work per thread diverges and one loses the property that all threads execute the same function simultaneously.

Approach 2

Spawn as many threads as there are leaves on the tree. This approach gets very complicated and there are many inactive threads during the high-level traversals of the tree.

Approach 3

Attempt a dynamic programming approach to the problem. Recursive code is to be avoided wherever possible in GPU computing. As one does not know the order in which threadBlocks are executed, recursive calls become extremely dangerous and are prone to causing hard to detect bugs. Despite the inclusion of dynamic parallelism which was implemented with the GK110 chipset on the Kepler K20 GPUs and above, most solutions can be optimized by avoided making use of this feature. Dynamic parallelism allows for very readable and user-friendly code to be written, and the GK110 chipsets and better, are well suited to launching micro kernels (a kernel is a launch of GPU threads to operate on a function. The threads are collapsed when the kernels finishes) but I cannot imagine launching **64^4** micro kernels. I guess a stack overflow exception would be thrown.

Approach 4

Launch different kernels to evaluate different levels in the tree. This is the approach that was implemented finally. There are **2^{30}** leaves on the tree each requiring 1 thread. One can argue for using fewer threads with each doing more work but that hasn't been experimented with here. The leaves are calls to $\{F_4(x_{4,0}), F_4(x_{4,1}), F_4(x_{4,2}) \ldots\ldots, F_4(x_{4,2^{30}-1})\}$. Each function call requires many independent parameters which are generated by the CPU. CPUs are better suited to diverging work and random memory accesses than GPUs are. The GPU is able to execute any kernels launched on it without holding up the CPU. Both perform operations independently. In addition to this, it is possible to copy data to the GPU while the GPU is operating on data already there. This is the way the overhead of having to transport all the data required by the kernels launched on the GPU, is hidden – this is referred to as latency hiding.

The data structures have to be examined closely. Before they can be finalized, however, the thread block distribution and structure has to be discussed. It was decided that optimal is to run 64 threads in a threadBlock. This is generally a very low number of threads per block, but in this case the functions called by each thread are resource heavy and we do not want to experience cache thrashing, when variables have to be paged between global memory and on-chip registers. The code is not, however, memory bound (meaning that warps spend significant time idling waiting for data) because coalesced memory reads are ensured through careful use of data structures. There are, of course, plenty of threadBlocks launched (**2^{24}**) but there is a limit on the number of threadBlocks that can be assigned to a SM (streaming multiprocessor). The next consideration is how to arrange the threadBlocks in the grid launched. There is a limit on the maximum number of threadBlocks in any direction (the grid of threadBlocks is 3-Dimensional and has a maximum dimension size of (**2^{16}-1**). It was decided to launch a grid of dimensions (**2^{12},2^{12}, 1**).

In order to store all parameters required by function calls at each level of the tree, separate data structures for each parameter are required. This looks very messy in the code, but eliminates the need to worry about offsets during memory transactions. In addition to this, the order of the traversal of the tree is parsed with care.

When a leaf is processed a single value is left per thread block. This is then randomly written to the correct position in global memory where the next kernel launch uses it. The results of each kernel launch are not copied back to RAM. This is unnecessary as all reduction operations occur on the GPU.

Optimizations implemented and future work

It is not necessary to wait for the parameter arrays to be completely populated by finishing parsing the entire tree with the CPU before transferring the data to the GPU for integration of the functions. Experiments were performed staggering data transfer to the GPU performed as soon as sufficient leaves are prepared, with kernel execution. The problem of solving high dimensional integrals deterministically is so important in high energy physics that it will be worthwhile profiling the code to attain as close to peak performance as possible.

Another optimization that was implemented was that on all the levels of the tree besides the root and the leaves, multiple integrals are called. The parameter arrays are reused. This is safe because as soon as an array is completed for one integral it is copied to the GPU and is then repopulated for the subsequent integrals. These are then transferred to the GPU as soon as that portion of the tree has been processed. A possible optimization is to use a parameter buffer so that transfer to the GPU can occur as soon as the tree has been parsed. As the problem is phrased currently, the limitation on global memory size, is not a constraint.

An optimization that will be considered in the future will be to make use of dynamic parallelism available in GPUs with compute capability 3.5 and higher but it will have to be done in such a way to avoid the problems with this stated in this report. The great advantage of this is that it will allow for far greater flexibility in the code to accept different models or to be addressed to completely different problems.

The most natural extension of this code will be to distribute the computation first among several GPUs present on a single node, and then to distribute across many nodes.

Conclusions

The problem of calculating the pion damping width of a pion propagating in a hot pionic gas can be computed deterministically and serially using a traditional approach but as computation was seen to take on average **38 minutes** per iteration, this is not feasible. Since it is necessary to be able to solve for Γ at a series of temperatures and to a high level of precision, many iterations are required.

The problem is very well suited to GPU processing. It is not memory bound and does not exceed the limitations imposed by the available GPU global memory. The data structures implemented to hold all the parameters required by the unrelated functions called at each level in the tree built.

Unrolling the call stack was an absolute requirement for this implementation. The GPU's limitation on resources/thread makes this absolutely necessary causing one to abandon all possibilities at a naïve and straight forward approach.

Generalizing the solution presented in this report to the wide variety of problems requiring the solution of high dimensional integrals is not a simple task as many problem specific optimizations were implemented. The avoidance of dynamic parallelism in this project has led to it being very unmalleable.

The challenge of maintaining order in such an enormous and strangely built data tree, where branches are not related to either their roots or their leaves in terms of data quantity, type or requirements was quite a challenge. Attempts to solve the problem backward, though intuitive, proved fruitless. The fact that it is relatively small in memory made this implementation possible. An extension of this project to more complex equations will be an interesting challenge.

References

[1] D. Blaschka, M. K. Volkov, and V. L. Yudichev, Phys. Of Atomic Nucl. Vol. 66, **12**, 2233 (2003).

[2] M. Asakawa and K.Yazaki, Nucl. Phys. A **504,** 668 (1989).

[3] D. Ebert, L. Yu, L. Kalinovsky, L. Munchov, and M.K. Volkov, Int. J. Mod. Phys. A **8**, 1295 (1993).

[4] T.Hatsuda and T. Kunihiro, Phys. Rep. **247**, 221 (1994).

[5] D. B. Blaschke, G.R.G.Burau, M. K. Volkov, and V. L. Yudichev, Eur. Phys. J. A **11,** 319 (2001).

[6] M. K. Volkov, Firz. Elem. Chastits At. Yadra **17**, 433 (1986)[Sov. J. Part. Nucl. **17**, 186 (1986).

[7] D. Ebert, H. Reinhardt, and M. K. Volkov, Prog. Part. Nucl. Phys. **35,** 1 (1994).

[8] CUDA C Programming Guide, v.7.0 (2015)

[9] Y. Liu, CUDA performance tune, Graduate university of Chinese Academy of Sciences, (2009)