Fine-grained parallelization of lattice QCD kernel routine on GPUs

Khaled Z. Ibrahim, François Bodin, Olivier Pène

1. Introduction

Quantum Chromodynamics (QCD) is a theory that describes the strong interactions that bind quarks and gluons together to form hadrons. Nuclear matter such as protons and neutrons are constituted of these hadrons. QCD is one of the grand challenge computing problems that require peta-FLOPS capable machine or better to obtain believable answers for some fundamental physics questions. Modeling the interactions described by this theory is often done through discretizing the space-time domain into a four dimensional hypercube, forming what is called Lattice QCD. Hybrid Monte Carlo (HMC) method is used to simulate this theory [1]. In this study, we consider the HMC implementation code of the ETMC collaboration [20,19].

Specialized computing machines are usually built to simulate this challenging problem in physics [4,5]. Exploring the use of accelerators to speedup floating-point computation is currently under investigation [9,3,13]. Graphic processing units (GPUs) are proving to be cost-effective architectures for many computationally demanding applications [16,2,10,12]. Graphical processing units have recently grasped the attention of scientific computing community as possible hardware accelerators for kernel routines that have high density of floating-point operations [1, 17,11,18,6]. GPUs have started attracting the interest of the LQCD community [9], as well. This type of architecture exhibits different resource constraints compared with CPU-based machines, for instance, the small on-chip cache, the large register file, the low overhead of thread synchronization and the limited communication between threads. These new constraints motivate for exploring new dimensions of parallelizing applications, targeting the best performing application decomposition for this class of architectures.

In this paper, we describe our experience in parallelizing the most time-consuming kernel routine of LQCD-HMC computation, named as the Hopping_Matrix, into GPUs. This routine computes the actions of the Dirac operator on a field of spinors. For realistic lattice sizes, it contributes almost 90% of the total execution time [21,7]. We present techniques to decompose sequential code for computing an iteration of the kernel routine into eight to sixteen threads of execution. The process involves breaking dependencies between the code fragments. The resulting parallelized code normally contains conditional functional execution. We show how to apply code transformations to avoid having diverging control-flow instructions that can severely impact the
performance on GPUs. The expensive communication of arrays between the GPU memory and the host memory is optimized based on observing the calling pattern to the Hopping_Matrix routine. The rest of this paper is organized as follows: Section 2 introduces the Hopping_Matrix routine and its functionality. Section 3 describes how to decompose an iteration of the Hopping_Matrix into multiple fine-grained threads. Experimental setup is briefed in Section 4. The search for optimal performance for the GPU-based parallel code is detailed in Section 5. In Section 6, we analyze the performance and the limitations observed for the Lattice QCD on GPUs. We conclude in Section 7.

2. Introducing the Hopping_Matrix kernel routine

The main kernel routine for the Lattice QCD, called the Hopping_Matrix, is responsible for computing the actions of the Dirac operator on a spinor field. It computes the interactions for half the field of spinors and gauge field links. Eq. (1) outlines the computation of the actions of a Dirac operator. This computation involves a sum over quark field spinors ($\psi_i$) multiplied by gauge field links ($U_{i,\mu}$) through the spin projector ($I \gamma_{\mu}$).

$$\chi_i = \sum_{\mu=x,y,z,t} \kappa_\mu \left[ U_{i,\mu} \left( I - \gamma_\mu \right) \psi_{i+\hat{\mu}} + U_{i,\mu}^\dagger \left( I + \gamma_\mu \right) \psi_{i-\hat{\mu}} \right].$$

(1)

The representation of each gauge field is an SU(3) matrix (3 $\times$ 3 complex variables). SU(3) refers to a matrix with three colors of quarks that are of special unitary, i.e., unit determinant. The gauge field links go in the four dimensions of the problem space. The spinors are represented by four SU(3) vectors composed of three complex variables. The runtime implementing this computation is based on the code provided by the HTMC collaboration [20,19].

The computation of one spinor involves 1608 floating-point (FP) operations based on the code provided by the HTMC collaboration [20,19]. The total FP operations for all directions slightly supersede 1600 FP operations. Most of these FP operations have a little reuse of the memory referenced (references per memory locations ranges from one to six); almost 60 doubles (about 800 bytes) are referenced in each direction, with little overlap between data referenced in each direction. This abundance of computation is not intervened by control-flow instructions, forming extremely large basic blocks. This good characteristic cannot be easily preserved for fine-grained parallel version as will be discussed in Section 3.2.

The half-spinor version of the computation performs half the computation in one loop and stores the intermediate results in a temporary array that is used later to complete the computation. The type of computation in each loop is more homogeneous (similar) compared with the full-spinor version.

With the advent of multicore chip, the chance for fine-grained parallelism is possible with little associated overheads for communication and synchronization. GPUs architectures allow a large number of fine-grained threads to cooperate in solving bigger calculations.

In this paper, we explore new design space dimensions of extracting parallelism for GPUs because they have a set of attributes different from those associated with general-purpose processors. GPUs enjoy high number of physical threads, low overhead for synchronizing threads, large number of registers and the ability to execute in SIMD fashion. On the other hands, they have small caches, constrained communication between threads and high latency to communicate with the host machine.

The GPU compiler is usually under the pressure of two contradicting requirements, increasing parallelism and improving the performance of each thread. Performance can usually be improved by allocating more resources per thread which reduces the number of threads for a given limited shared resources. Programmer intervention is of crucial importance in this case to explore the tradeoffs of the different ways of expressing the problem to the compiler. In the rest of this paper, we will specifically concentrate on task granularity impact on performance for the spinors’ calculation.

3. Parallelizing the Hopping_Matrix routine

The obvious way for parallelizing the Hopping_Matrix routine is to parallelize the outer loop, i.e., to do the computation of each spinor independently. The computation at each spinor is independent of the others because of the even–odd decomposition. Each thread or process can be assigned one or more spinors.

The amount of computation for one spinor is independent of the lattice size and requires a considerable number of FP operations (approximately 384 FP operations for each dimension of the 4D space). The total FP operations for all directions slightly supersede 1600 FP operations. Most of these FP operations have a little reuse of the memory referenced (references per memory locations ranges from one to six); almost 60 doubles (about 800 bytes) are referenced in each direction, with little overlap between data referenced in each direction. This abundance of computation is not intervened by control-flow instructions, forming extremely large basic blocks. This good characteristic cannot be easily preserved for fine-grained parallel version as will be discussed in Section 3.2.

The half-spinor version of the computation performs half the computation in one loop and stores the intermediate results in a temporary array that is used later to complete the computation. The type of computation in each loop is more homogeneous (similar) compared with the full-spinor version.

With the advent of multicore chip, the chance for fine-grained parallelism is possible with little associated overheads for communication and synchronization. GPUs architectures allow a large number of fine-grained threads to cooperate in solving bigger calculations.

In this paper, we explore new design space dimensions of extracting parallelism for GPUs because they have a set of attributes different from those associated with general-purpose processors. GPUs enjoy high number of physical threads, low overhead for synchronizing threads, large number of registers and the ability to execute in SIMD fashion. On the other hands, they have small caches, constrained communication between threads and high latency to communicate with the host machine.

The GPU compiler is usually under the pressure of two contradicting requirements, increasing parallelism and improving the performance of each thread. Performance can usually be improved by allocating more resources per thread which reduces the number of threads for a given limited shared resources. Programmer intervention is of crucial importance in this case to explore the tradeoffs of the different ways of expressing the problem to the compiler. In the rest of this paper, we will specifically concentrate on task granularity impact on performance for the spinors’ calculation.

3.1. Decomposition of the Hopping_Matrix routine

The calculation for each spinor of the output involves the following two groups of operations:
Group 1: Simple operations on SU(3) vectors, for instance, add, subtract, add conjugate, etc. These operations reference 12 doubles (96 bytes) and have 6 FP operations. These operations are due to applying the spin projection operator on spinor vectors, as introduced in Eq. (1).

Group 2: Complex operations, for instance, multiply a \(3 \times 3\) complex matrix (or its conjugate transpose) by an SU(3) vector. These operations reference 44 doubles (352 bytes) and have 84 FP operations, mostly due to multiplying a gauge field matrix and count of FP operations but slightly differs depending on the direction in space. For instance, in one direction we may have to add vectors while for the other we may have to add the conjugate vectors (which involves different ordering for accessing memory locations and replacing some additions with subtractions).

Fig. 2 shows the computation of one spinor in the Hopping_Matrix routine, for the full-spinor version. The original code serializes the computation of a spinor in the four direction of the space. As shown in Fig. 2, the computations in the positive direction are mostly alike, and similarly the computation in the negative direction.

The half-spinor version of the code does the computation in one loop for the positive direction and then in another loop for the negative direction. The intermediate results are stored in additional data structures. The indexing for global structures is serialized because relative indexing is used. The indexing is also pre-computed in arrays that involves multiple levels of indirection.

Fig. 3 shows restructuring the dependency of computation for computing one spinor considering the full-spinor version of the code. The computation in one direction can be further decomposed into two half-spinors (not shown in figure).

The indexing of the computation needed to be replaced by absolute indexing such that accessing data for one dimension of the space is independent of accessing data for the other dimensions of the space. Indexing is also performed through computation rather than accessing pre-computed indices. In our experiment we noticed no performance difference when using repeated computations of array indices. The tradeoffs are between repeating accessing memory and repeating calculation. Formally, this decomposition corresponds to computing half a spinor in one direction of the space.

The basic work unit (thread assignment) explored in this work is composed of one simple operation on SU(3) vectors of the first group of operations and one complex operation of the second group. This work unit has 90 FP operations and 56 memory references. Formally, this decomposition corresponds to computing half a spinor in one direction of the space.

The fine-grained parallelism available varies for the two versions of the Hopping_Matrix implementations explored in this study. For the full-spinor version of the code, we can extract four threads of computation for each dimension of the space. This allows having 16 threads in total for computing one output spinor. Although all these threads have identical amount of work (FP operations and memory references), the types of computation are not exactly similar. Each thread needs to determine the type of work based on its position (thread ID) in the space.

For collective operations, some threads are assigned the job of computing the total output spinor. The collective operations require synchronization to ensure consuming data after being produced from different threads.

For the half-spinor version, the computation is split into two phases (two main loops in the original code). The first phase can be decomposed into 8 work units, fortunately homogeneous. The results of the computation are stored in an intermediate array of structures. The second phase of computation can also be split into 8 phases of computation which are slightly less homogeneous. In this phase, the output spinor are computed based on processing

![Fig. 2. Computation of one spinor in the Hopping_Matrix kernel routine.](image-url)
the results stored on the intermediate data structures. The second phase is also appended with reduction operations.

As we target GPUs as a platform for parallelization, we should recall that GPUs perform better if all threads are exactly executing the same instruction on possibly different data (SIMD model). Control-flow dependent computations are usually serialized if the outcomes of a control-flow instruction are different for different threads, which lead to lower performance. Parallelizing an iteration of spinor computation does not always lead to identical instructions performed on different data. We will explore this effect in Section 5.1.

GPUs also benefit from large number of threads grouped into blocks. Threads within the same block can communicate, but no inter-block communication is allowed. NVIDIA documentation reports that thread blocks are executed in wraps that are multiples of 32 threads.

Only 8 threads are needed to cooperate for the calculation of a spinor for the half-spinor version (in two steps), while 16 threads are needed for the full-spinor version (in one step). Although the intrinsic dependence leads to small number of threads per block, we can form larger group of threads by assigning the computation of multiple spinors to the same block of threads. We explored this idea as will be detailed in Section 5.2.

4. Experimental setup

All the results reported involve the average execution time for 1000 call for Hopping_Matrix routine toggling between the computation of the odd and the even subfields. A warm-up of 16 iterations is performed before taking the timing measurements.

The used GPU is NVIDIA GeForce 8800 GTX [14] hosted by a server of 8 quad-processor Intel Xeon processor at 1.86 GHz, 4 MB L2-cache. The programming environment is Cuda 1.1 [15].

For the GPU performances, we report the execution time relative to the execution time using SSE2 optimized version of the half-spinor version of the code running on a dual-core machine 2.8 GHz Intel Xeon with 512 KB cache. This reference machine outperformed the machine hosting the NVIDIA card. On the GPU all computations were done in a single precision (32 bit) rather than on double precision as with the SSE2 version on the CPU.

The execution time is split into the following components:

- Computation while assigning all threads the same type of FP operations and referencing memory locations similar to the original code. This allows measuring the potential performance advantage of GPU SIMD mode of execution. We do not violate the dependencies needed for the collective operations and synchronizations are added to respect the dependencies imposed by the original computation.

- The additional effect of control-flow on performance.

- The additional effect of considering the communication of spinor arrays between the host machine memory and the GPU memory. This effect assumes that the input and the output spinor arrays need to be communicated on each call of the Hopping_Matrix.

We run the parallelized versions for the half-spinor and the full-spinor versions for three lattice sizes. The first size is with all dimensions of the lattice equal to 8 and the time dimension is set to 8, denoted as $8^3 \times 8$. The second and the third lattices have the sizes $16^3 \times 16$ and $32^3 \times 32$, respectively.

5. Parallelization of the Hopping_Matrix routine for performance on GPUs

This section addresses the performance of the fine-grained parallelization of the Hopping_Matrix routine. We illustrate various obstacles that can substantially reduce the GPU throughput and how to overcome them. We specifically address the conditional execution of the code and the memory transfer of arrays between the GPU and the host as the two major causes of reduced performance.

We also show the advantage of using fine-grained parallelization in contrast with coarse-grained parallelization.

5.1. Tuning the Hopping_Matrix parallel version for GPUs

GPUs perform at its utmost speed when all threads of a wrap (group of threads executing simultaneously) execute in SIMD fashion. Control-flow instructions can significantly affect the performance of GPUs if they create divergent threads of execution for the same wrap and the execution of the same wrap of threads is serialized instead of being executed in parallel.

Ideally all threads of a wrap need to take the same decision on a control-flow instruction. To extract fine-grained parallelism from one iteration of the Hopping_Matrix, each thread may have slightly different code to execute. The problem with doing the calculation for a spinor in parallel is that it is not straightforward to make control decision the same for a wrap of threads.

In this section, we show two transformations we used to avoid inserting divergent control-flow instructions. Both transformations can be adopted with control-flow instructions that have control variables with limited number of values, for instance those dependent on thread IDs.

The first transformation replaces a control-flow instruction with some additional computations. Fig. 4 shows two versions of the same code. The code in Fig. 4(a) uses control-flow instructions with the thread ID as the control variable. Because we know that the threads in the x-direction (referred to as threadx) are limited to four possibilities, we transform the code to the one shown in Fig. 4(b). It is shown that we can replace the control-flow
Fig. 4. (a) Code with divergent control-flow. (b) The transformed version of the same code without control-flow.

instruction with additional computations that can be done by all threads and each thread reaches the appropriate part of the array to access.

In this transformation, all possible indexing components are computed and then we select among them using multiplications that nullify some of the computed components and take the right component. The multiplier works as a selector and is computed based on thread ID. Certainly the new code involves more computation compared with the original code but it provides better chance for SIMD execution.

The second transformation relies on using indirection arrays to avoid conditioning the selection of the correct memory address to access.

Fig. 5(a) shows another fragment of code with control-flow dependent on the thread ID. To remove control-flow from this code we have two problems; first the direction of subtracting vectors \( t \) and \( \phi \) changes with the thread ID; second, the indexing of the output arrays also depends on the thread ID.

To solve these problems we introduced an array, \( sb \), which can be indexed by the thread ID and returns the correct indexing for the output thread. To solve the subtraction direction problem, we introduced a new function, \( sub\_mul\_su3\_v \), that after doing the subtraction, it multiplies the result by a factor. The factor is assigned to \( 1 \) if the direction of the subtraction is to be preserved, or \(-1\) of the subtraction direction is to be reversed. We then introduced an array that is indexed by the thread ID to give the appropriate factor for each thread. Fig. 5(b) shows the equivalent branch-free code.

Combinations of the above two transformations were used to remove as many branches as possible from the fine-grained parallel version of the code. We did not apply these transformations when severe code expansion was likely to result from applying them.

Fig. 6 shows that the effect of branch instructions can severely decrease the performance. The transformed codes have the execution time reduced to 60% of the unoptimized execution time for the half-spinor version and 53% for the full-spinor version.

The proposed transformations almost removed the control-flow effect on the execution time for the half-spinor version while removing 70% of its effect on execution time for the full-spinor version.

In this experiment the GPU threading is underutilized with the small number of threads which gives the most stressful scenario for the effect of branching on execution. We will show that assigning more spinor per block of threads will further reduce the effect of control-flow.

It is notable that we cannot generate enough number of threads to reach 32 threads making the same control-flow decision because each thread has a large need for the shared registers and shared memory. A block of threads will fail to launch if their combined requirements of registers exceeded what is available by hardware (8K registers per block of threads). Additionally, the code will not compile if the shared memory required by all threads exceeded the hardware limit (16 KB per block of threads). We need to hold shared data needed for the reduction operations on all the spinors in computation simultaneously.

5.2. Hopping Matrix performance and overheads

Fig. 7 shows the relative performance on a GPU compared with the optimized serial version for a lattice of the size \( 8^3 \times 8 \). In this experiment we merged the computation for multiple spinors within one block of threads on the GPU. We noticed that the performance improve for both the half-spinor and the full-spinor version by increasing the number of spinors assigned to a block. Recalling that each spinor requires 8 threads for the half-spinor version a large number of threads and each thread reaches the appropriate part of the array to access.

In this transformation, all possible indexing components are computed and then we select among them using multiplications that nullify some of the computed components and take the right component. The multiplier works as a selector and is computed based on thread ID. Certainly the new code involves more computation compared with the original code but it provides better chance for SIMD execution.

The second transformation relies on using indirection arrays to avoid conditioning the selection of the correct memory address to access.

Fig. 5(a) shows another fragment of code with control-flow dependent on the thread ID. To remove control-flow from this code we have two problems; first the direction of subtracting vectors \( t \) and \( \phi \) changes with the thread ID; second, the indexing of the output arrays also depends on the thread ID.

To solve these problems we introduced an array, \( sb \), which can be indexed by the thread ID and returns the correct indexing for the output thread. To solve the subtraction direction problem, we introduced a new function, \( sub\_mul\_su3\_v \), that after doing the subtraction, it multiplies the result by a factor. The factor is assigned to \( 1 \) if the direction of the subtraction is to be preserved, or \(-1\) of the subtraction direction is to be reversed. We then introduced an array that is indexed by the thread ID to give the appropriate factor for each thread. Fig. 5(b) shows the equivalent branch-free code.

Combinations of the above two transformations were used to remove as many branches as possible from the fine-grained parallel version of the code. We did not apply these transformations when severe code expansion was likely to result from applying them.

Fig. 6 shows that the effect of branch instructions can severely decrease the performance. The transformed codes have the execution time reduced to 60% of the unoptimized execution time for the half-spinor version and 53% for the full-spinor version.

The proposed transformations almost removed the control-flow effect on the execution time for the half-spinor version while removing 70% of its effect on execution time for the full-spinor version.

In this experiment the GPU threading is underutilized with the small number of threads which gives the most stressful scenario for the effect of branching on execution. We will show that assigning more spinor per block of threads will further reduce the effect of control-flow.

It is notable that we cannot generate enough number of threads to reach 32 threads making the same control-flow decision because each thread has a large need for the shared registers and shared memory. A block of threads will fail to launch if their combined requirements of registers exceeded what is available by hardware (8K registers per block of threads). Additionally, the code will not compile if the shared memory required by all threads exceeded the hardware limit (16 KB per block of threads). We need to hold shared data needed for the reduction operations on all the spinors in computation simultaneously.

5.2. Hopping Matrix performance and overheads

Fig. 7 shows the relative performance on a GPU compared with the optimized serial version for a lattice of the size \( 8^3 \times 8 \). In this experiment we merged the computation for multiple spinors within one block of threads on the GPU. We noticed that the performance improve for both the half-spinor and the full-spinor version by increasing the number of spinors assigned to a block. Recalling that each spinor requires 8 threads for the half-spinor version...
version and 16 threads for the full-spinor version, we noticed that the optimal performance corresponds to merging 8 spinors in the half-spinor version and 4 spinors in the full-spinor version which corresponds to having 64 threads per block. For the rest of the discussion we will concentrate on the best performing configuration.

Ignoring overheads (communication with the host memory and control-flow), the best speedup we can achieve is associated with the full-spinor version at 7.8x. The best performing half-spinor version lags the best performing full-spinor version by 22% with a peak of 6.1x speedup.

Control-flow reduces the attainable speedup to 7.5x (from 7.8x) and 5.5x (from 6.1x) for full-spinor and half-spinor, respectively. The transformation applied to remove the control-flow proved successful in reducing its impact on performance.

Considering memory transfers of the spinor arrays on each call of Hopping_Matrix can severely impact performance due to copying the input spinor array from the host memory to the target device and copying the output spinor array from the device to the host memory. The speedups are lowered to 4.1x for the full-spinor version and to 3.4x for the half-spinor version. The GPU-host communication overhead reaches 45% of the total execution time. In the next section we will describe how to reduce the effect of these transfers on performance.

Fig. 8 shows that the same trend observed for small lattice size continues with larger lattice sizes. The full-spinor version continues to outperform half-spinor version, and the optimal assignment of spinors per block of threads remained the same. Another observation is that the negative effect of control-flow decrease when we assign more spinors to the same block of threads for the full-spinor versions. This is compatible with the expected improvement of throughput when more threads are taking the same decision on a control-flow instruction and thus doing exactly the same computation for different spinors (different data). This alleviates the effect of deviation of control-flow within one spinor computations. We may have negative impact of increased parallelism if contention for memory occurs. Interestingly for the full-spinor version, the code without control-flow has less performance compared with the code with control-flow when assigning 16 spinors to a block. A possible reason for performance degradation of code without control-flow is that the conflicts for accessing memory locations increase with the large number of spinors assigned to a block of threads.

It is notable that the gain, while considering all overheads, improves from 4.1x for a lattice with the size $32^3 \times 8$ to 4.98x for a lattice with the size $16^3 \times 16$, and reaching 5.2x for a lattice with the size $32^3 \times 32$. One of the reasons for this improvement is the increase in the computation-to-communication ratio. The contribution to execution time of memory transfers, between the host and the GPU, decreases with the lattice size increase from 45% to 39% of the execution time.

Fig. 9 shows the results for the full-spinor version with a bigger lattice with the size $32^3 \times 32$. For this lattice size, the number of blocks (which relates to the number of spinors) exceeded the number of allowable blocks in one dimension. The number of blocks on a GPU should be less than 64k blocks in one dimension. We were to organize blocks in two-dimensional array of blocks to accommodate the large number of blocks. For smaller lattice sizes, we noticed that organizing blocks in a two-dimensional array have the same performance as organizing them in a one-dimensional array.

5.3. Reducing the effect of memory updates

Exchanging spinor arrays on each call of the Hopping_Matrix gives pessimistic view of the performance because some of the spinor arrays can be kept in the GPU memory and not communicated to the host memory. This involves moving other code from the host to the GPU and holding extra spinor arrays on the GPU memory.

For the HMC simulation, a function called Qtm_pm_psi contributes the most for calling the Hopping_Matrix routine (97.8% to 99.8%). This function calls the kernel four times and uses two intermediate spinor field copies. After each call a post-processing functions of the output spinors array is called.

We considered moving the whole computation for Qtm_pm_psi to the GPU. This involved allocating the two temporary spinor arrays on the GPU global memory and thus avoiding copying memory back and forth between the host machine memory and the GPU memory. The memory transfers are thus needed every four calls to the Hopping_Matrix routine.

We also tried two implementations for Qtm_pm_psi. The first implementation involved doing the post-processing of the spinors as part of the Hopping_Matrix routine. We appended the Hopping_Matrix function with conditional computation for post-processing. The conditional execution is independent from thread ID, i.e., the same branching decision will be taken by all threads, thus it should not affect the throughput of the GPU. The advantage of this implementation is the need for fewer calls from the host to the device. The disadvantage is the inflexibility due to the use of the same number of threads for computing the spinor and for post-processing.
The second implementation separated the calls for the Hopping_Matrix and the post-processing routines. The advantage of this technique is the ability to control computation load distribution more freely. Fig. 10 shows the performance of the Qtm_pm_psi routine compared with the base SSE2 version. The first implementation is referred to as “with fusion” while the second implementation is referred to as “no fusion.” The lattice size is \(32^3 \times 32\). For Qtm_pm_psi the memory transfer did not exceed 12% of the total execution time, compared with 39% for calling the Hopping_Matrix routine alone.

We found that controlling the threads-per-block for the Hopping_Matrix and separating the post-processing gives an advantage over fusing the computation. The best performance is associated with 64 threads (4 spinors). The achievable speedup considering all overheads is 8.3x.

For a lattice with the size \(32^3 \times 32\), the half-spinor version was not successful to launch on the GPU because of the larger memory requirement associated with the intermediate arrays needed. Fortunately, the performance of the full-spinor version outperformed the half-spinor version as shown in the previous section.

5.4. Scalability of the Hopping_Matrix on GPUs

Fig. 11 shows the scaling behavior for best performing parallelization (full-spinor version with 4 spinors per block) for different lattice sizes. In our computation for performance in GFLOPS, we considered workload intrinsic FLOPS (ignoring additional FLOPS introduced due to parallelization and transformations) while considering the time for total execution.

In Fig. 11 we notice that the peak GFLOPS achievable by the algorithm rises with the increase in the lattice size because of the chance of overlapping the computation from different thread blocks. This improvement saturates starting with the lattice size \(16^3 \times 16\) at 7.5 GFLOPS.

Considering the communication with the host CPU, the achievable GFLOPS improves with the increase of the lattice size because of the increase of the computation-to-communication ratio. The throughput rises from 3.6 to 4.6 GFLOPS. Unfortunately, we cannot continue indefinitely to increase the lattice size because it will not fit onto the GPU memory.

Fig. 11 shows that porting Qtm_pm_psi has a lower peak performance of 7 GFLOPS because the post-processing code ported to the GPU has lower computation-to-communication ratio compared with the Hopping_Matrix routine. Even with the lower efficiency of the additional code ported to the GPU, this helps in reducing the need to communicate the spinor arrays to one quarter of the instances of calling the Hopping_Matrix routine.

The achievable throughput of Qtm_pm_psi is 6.2 GFLOPS (compared with a peak of 7 GFLOPS), which is much higher than that attainable by making individual calls to the Hopping_Matrix routine.

In Fig. 11, it is shown that the SSE2 version of the code, which is enhanced with explicit software prefetching instructions, can run at 0.89 GFLOPS on the CPU and the performance goes down with increasing lattice size and reaches 0.74 GFLOPS for Qtm_pm_psi code.

The throughput achieved on the GPU represents a small fraction of its computational power. The main source of inefficiency is that the data reuse within the Hopping_Matrix is low. Bringing data to fast access memory (or cache) can be less fruitful if the data are not reused many times.

The GPU performance outperforms CPU performance because of the ability of the GPU to parallelize the costly memory accesses by the many threads it has. Up to 128 multiprocessors can issue memory requests concurrently. Memory requests are also costly on GPUs and require hundreds of cycles.

Allocating data in the cachable memory space on a GPU requires using textures address space. Unfortunately this address space can manage simple data types and cannot be used with complex data-structures needed by the Hopping_Matrix routine.

5.5. Fine-Granularity vs. Coarse-Granularity parallelism on GPUs

In this study, we focused on fine-grained parallelization of the Hopping_Matrix routine on GPUs. To compare the performance of fine-grained parallelization with coarse-grained parallelization, we considered assigning the whole spinor computation to one thread of computation, in case of the full-spinor version, and assigning a spinor to two consecutive threads in case of the half-spinor version.

This implementation is favored by the nonexistence of control-flow instructions, which suits well the SIMD execution model for GPUs. We also used a code with tree-like dependency between computations (thus reducing the height of the critical-path). The computation of the full-spinor version is done in one bulk of computation, while the computation for the half-spinor version is divided into two phases.

Fig. 12 shows the performance for lattice size \(32^3 \times 32\) relative to the best performing fine-grained full-spinor parallelization. The maximum number of threads that succeeded to launch were 64 threads per block. Higher number of threads failed for this code because of the pressure on the registers available per block of threads. Fig. 12 shows that coarse-grained half-spinor version performs better than the coarse-grained full-spinor version. The coarse-grained half-spinor version assigns almost half the amount of work per thread compared with the coarse-grained full-spinor version and uses two phases for computation. The best performing coarse-grained half-spinor version outperforms the...
best performing coarse-grained full-spinor version by 19%. It is notable that coarse-grained half-spinor code can be looked at as fine-grained version of coarse-grained full-spinor code.

As shown in Fig. 12, the best performing fine-grained version outperforms the best performing coarse-grained version by 28%. This shows that fine-grained parallelism is handled better by GPUs. We assert that the improvement of the fine-grained version is due to the granularity of the thread task rather than exposing more parallelism, as will be discussed in details in Section 6. Fine-grained parallelism provides a better chance for the compiler to apply optimizations when the resources per thread are critical to performance.

6. Performance and limitations of GPUs for lattice QCD

Even though the design philosophy of most GPU manufactures intentionally hides the details of the hardware implementation and the details of the compiler, in this section we use some clues to predict the relation between the performance observed and the granularity of the task allocated per thread.

Table 1 summarizes a comparison between the different ways of parallelizing the Hopping_Matrix, introduced in this study. The number of threads for the coarse-grained version of the smallest lattice, considered in this study, is enough to saturate the GPU resources. For the experiments discussed in this paper, the number of threads for the coarse-grained version is 524,288 independent threads, while the count of GPU physical multiprocessors is merely 128. This shows that the advantage of fine-grained parallelism is not due to increasing the degree of concurrency in the algorithm.

To analyze the effect of granularity on performance, we used the “fail to launch” notifications given by GPUs to determine the maximum number of threads that exhausts the hardware resources. A GPU gives such notifications when the resource requirements for a block of threads running concurrently exceed what is architecturally available. In the following discussion, we denote the hardware resources allocated per block of threads as $R_{\text{block}}$ and the resources allocated to a spinor as $R_{\text{spinor}} = g \cdot R$, where $g$ represents the granularity and has the values of $2^f$ for coarse-grained full-spinor, $ch$ for coarse-grained half-spinor, $ff$ for fine-grained full-spinor, or $fh$ for fine-grained half-spinor.

In Table 1, we find that the maximum number of threads that succeed to launch depends on the granularity of the work assigned (multiples of 32). The number of threads that failed to launch for coarse-grained versions starts from 96 threads, while for the fine-grained versions it starts from 224 threads.

Based on the amount of work assigned for each thread of computation, we can deduce the following inequalities governing the resources that the compiler allocates per spinor:

$$\frac{R_{\text{block}}}{96} < \frac{R_{\text{spinor}}}{16} \leq \frac{R_{\text{block}}}{64}$$
$$\frac{R_{\text{block}}}{96} < \frac{R_{\text{spinor}}}{2} \leq \frac{R_{\text{block}}}{64}$$
$$\frac{R_{\text{block}}}{192} < \frac{R_{\text{spinor}}}{16} \leq \frac{R_{\text{block}}}{224}$$
$$\frac{R_{\text{block}}}{192} < \frac{R_{\text{spinor}}}{2} \times 8 \leq \frac{R_{\text{block}}}{224}$$

The denominator of the middle term in each of the above inequalities represents the number of threads involved in the computation of one spinor. Based on these inequalities and the task assignment, we know that $R_{\text{spinor}}$ is relatively similar. The comparison of the resources required by the $R_{\text{spinor}}$ and $R_{\text{spinor}}$ is then given by the following inequalities:

$$\frac{64}{14} < \frac{R_{\text{spinor}}}{R_{\text{spinor}}} < \frac{96}{12}$$
$$\frac{32}{14} < \frac{R_{\text{spinor}}}{R_{\text{spinor}}} < \frac{48}{12}$$

These two inequalities show that the spinor in the fine-grained version is allocated a ratio of 2.29 to 4 times the amount of resources allocated for the coarse-grained half-spinor version, while comparing with the coarse-grained full-spinor version ranges between 4.57 and 8. This provides us insights about how the resources affect the performance. The resources include, for instance, the number of registers allocated per thread (8192 for NVIDIA 8800 GTX).

Cuda compiler usually has conflicting constraints on the resources allocation per block of threads. It tries to maximize the level of parallelism while trying to improve the performance of each thread. Certainly, achieving higher degree of parallelism requires assigning fewer resources per thread thus possibly hurting performance. Apparently, Cuda compiler tries not to reduce the amount of parallelism below 64 threads. For the coarse-grained version of the code, the requirements for computation are high compared with the available resources, as detailed in Section 2. With fine-grained versions more resources are assigned to computing one spinor because each spinor is assigned the resources of multiple threads. Allocating more registers per spinors reduces register spilling and thus reduces memory references.
which are very expensive in GPUs. This results in the performance advantage we have noticed for fine-grained versions.

Comparing the fine-grained versions (full-spinor and half-spinor), the full-spinor version has fewer references to memory (no intermediate data structure) and smaller memory footprint compared with the half-spinor version which lead to the performance advantage noticed for the full-spinor version.

The challenges in porting an application like Lattice QCD can be summarized as follows:

- The need for SIMD kind of execution favor doing the computation in coarse-grained because the computation will not be intervened by control-flow and the same instructions can be applied on all spinors. Unfortunately, each thread computing a spinor will require a large number of registers to obtain optimal performance and to avoid frequent memory accesses which are very expensive in GPUs. Computing one spinor involves touching a large memory footprint with little reuse of data.

- The need for hiding long latency memory operations requires large number of threads which reduces the amount of resources available per thread, for instance registers, thus increasing the frequency of memory accesses. The optimal thread count vs. thread resources is not easily achievable because the Cuda compiler tries to achieve a certain predefined scheduling goals in terms of the number of threads.

- Creating fine-grained threads increases the heterogeneity in threads’ computations and makes SIMDizing the code more challenging because of the need to conditional executions. Conditional execution can be circumvented by transforming the code, but these transformations usually involve redundant computations that reduce the throughput of the GPU.

For the sake of portability, GPU makers allow little control for the application developer to decide the best tradeoffs between the above constraints. Programmer intervention is needed to explore coding alternatives to determine the best tradeoff between increasing concurrency and increasing resources per thread. In these cases, the application developer should express the problem in multiple alternative representations (task assignments) to the compiler.

Another important factor affecting the performance of the presented routine is the low density of computation compared with the memory accesses. The density of the computation to the memory access is at best 1.1 single precision FP per byte. In fact, data movement for the Lattice QCD computation bounds the amount of GFLOPS that can be achieved. The spinor field constitutes 25% of the static data accessed in the computation. For PCI express 16x interface, the overhead of exchanging data between the main memory and GPU memory makes the maximum possible performance 20 GFLOPS, assuming infinite computational power. If we consider the data transfer between the GPU memory and the GPU multiprocessors, then the maximum possible performance will be lowered to 16.5 GFLOPS. Improvements in the bandwidth between the main memory and the GPU memory or in the bandwidth between the GPU multiprocessors and its own memory will be greatly reflected in the performance for this kind of computations.

6.1. Limitations of GPUs for lattice QCD

The use of GPUs to accelerate the simulation of the Lattice QCD is challenged by the following factors:

- Floating-point exceptions are not available in the current generation GPUs. This problem can be partly handled by checking the SU(3) property of the results to check if this property is violated. The check can be done on the GPU because of its simplicity and the associated amount of parallelism. This check can be performed every multiple calls to reduce its associated overhead. Redundant computation is also possible to circumvent this problem.

- The computation in single-precision is usually more efficient on GPUs, while the computation requires double precision accuracy. This problem is usually handled using two phases of computation. The first phase involves computation with sloppy precision and when a certain error level is achieved, a new phase of precise computation is done. Fortunately, NVIDIA GPU manufacturer, as well as other manufacturers, announced efficient double precision arithmetic support in their new cards.

7. Conclusion

Deciding the granularity of the thread task versus increasing concurrency level is an important algorithmic tradeoff that affects the performance achieved in graphic processing units, especially for an application like Lattice QCD. Programmer intervention is needed to express the problem with multiple task assignments for threads of computation. In this paper, we explored four different implementations for the main kernel routine of the Lattice QCD simulation. We showed that parallelizing the kernel routine the Hopping_Matrix favors, in terms of performance, using fine-grained (with full-spinor computation) parallelism because it allows the allocation of more resources per spinor computations. Fine-grained full-spinor version consistently outperforms other versions in terms of GFLOPS throughput as well as memory requirements, achieving 8.3x over optimized SSE2 implementation.

Fine-grained parallelism requires careful code writing to avoid divergent control-flow that significantly impacts the throughput of the GPU. In this paper we presented two code transformations that can be used to eliminate control-flow instructions and trading them by additional computation. We illustrated the effectiveness of these transformations in reducing the negative impact of divergent control-flow.

We also showed that we may need to port code with less effective throughput in order to reduce communicating arrays between the host CPU memory and the GPU memory, thus improving the performance on GPUs.

Acknowledgments

We thank Thomas Chirappa, Karl Jansen and Carsten Urbach, main authors of the HMC code for having kindly allowed us to use it under the ETMC collaboration. We also thank Zhaofeng Liu who adapted the former code for the present work and Piotr Lesnicki for the insights to GPU programming.

References


Khaled Z. Ibrahim received his Ph.D. in Computer Engineering from North Carolina State University, USA, in 2003. He holds an assistant professor position in Suez Canal University, Egypt, since 2003. He is currently visiting the laboratory of IRISA/INRIA, Rennes, France. He conducted research in different aspects of parallel architectures and computing. His research interests include speculative execution on shared memory CMP-based systems, power-aware scheduling for multiprocessor systems, design space exploration for multiprocessor systems, and code optimization on accelerators for high performance computing.

François Bodin cofounded CAPS Entrepris in 2003 while he was a university professor at University of Rennes I and since January 2007, he joined CAPS Entreprise as CTO. His contribution includes new approach for exploiting high performance processors in scientific computing and in embedded applications. Prior to joining CAPS Entreprise, he held various research positions at University of Rennes I and INRIA. He has been appointed full professor in computer science in 1999. Professor François Bodin holds a Master's in CS and a Ph.D. in CS, both from University of Rennes I.

Olivier Pène received “Doctorat d’Etat en Sciences Physiques” in 1974 from Université d’Orsay, France. He is Directeur de Recherches 1ère classe in CNRS (France). His research interests are theoretical particle physics, especially lattice quantum chromodynamics (LQCD), and computer architectures adapted to LQCD. He was a member of the European collaboration which developed the LQCD dedicated ApENext computer.