Lattice QCD simulations using the OpenACC platform

Pushan Majumdar
Department of Theoretical Physics, Indian Association for the Cultivation of Science, 2A & 2B Raja S.C. Mullick Road, Jadavpur, Kolkata 700032
E-mail: tppm@iacs.res.in

Abstract. In this article we will explore the OpenACC platform for programming Graphics Processing Units (GPUs). The OpenACC platform offers a directive based programming model for GPUs which avoids the detailed data flow control and memory management necessary in a CUDA programming environment. In the OpenACC model, programs can be written in high level languages with OpenMP like directives. We present some examples of QCD simulation codes using OpenACC and discuss their performance on the Fermi and Kepler GPUs.

1. Introduction
Lattice Quantum Chromodynamics (Lattice QCD) is a framework for computing properties of strong interactions through numerical simulations. In this formulation the continuum of space-time is replaced by a Euclidean lattice of the same dimension. In this formulation, matter fields such as quarks are placed on the lattice sites while the gauge fields are placed on the links which connect the sites. Space-time symmetries are sacrificed in this formulation, but gauge symmetry is maintained. The finite lattice spacing introduces an ultra-violet cutoff and the finite volume of the lattice provides an infra-red cutoff. Nevertheless to compare results obtained from the lattice with experiments one must extrapolate to zero lattice spacing and this is a very resource intensive process which goes as $a^{-7}$ [1] where $a$ is the lattice spacing.

A finite lattice means that the underlying path-integral has finite degrees of freedom and is amenable to evaluation using numerical simulations on computers. However the data size for such a problem is moderately large. Since we want to run our programs on GPUs which have very limited memory it is useful to make an estimate of the memory requirements. A rough estimate can be obtained as follows. For strong interactions we want to simulate a box with sides of 4 fermi ($10^{-15} m$) and $a$ between 0.1 and 0.01 fermi. With a lattice spacing of 0.1 fm we need $40^4 = 2.56 \times 10^8$ sites and $10.24 \times 10^6$ links. In terms of memory requirement this means about 1 GB for matter fields and 1.5 GB for gauge fields. Making the lattice spacing finer to $a = 0.01$ fm would increase the memory requirements by about 4 orders of magnitude to 10 TB for matter fields and 15 TB for gauge fields. Current state of the art simulations use $a \sim 0.05$ fermi.

With an ever-increasing need for higher accuracy for numerical results, increase in data volume coupled with saturation of single core performance, efficient parallelization has become more and more important for lattice QCD computations. Fortunately lattice QCD codes can be parallelized relatively easily as for a large fraction of the simulation time they perform repeated...
matrix vector operations with vector sizes of about $10^7$ elements. While parallelization between different nodes using MPI like platforms have been around for some time, the current hardware trend is to have many cores on a single chip or card. Our concern in this article is exploring parallelization on GPUs which have about 500 cores on a single card. Chip architectures which fall in this category are multicore CPUs $\sim$ 18 cores on a chip with about 2GB of memory per core, Xeon-PHIs with 60 cores on a chip with about 250 MB of memory per core and GPUs with 500 or 1000 cores on a chip with only about 12 MB per core.

The same OpenMP code can run on multi-core CPUs and the Xeon-PHI in its native mode although different organizations of the data is required for efficient operations as the memory available per core differs by about an order of magnitude. Running programs on GPUs require more effort. GPUs are programmed primarily with CUDA which is a C like language. It takes a reasonable effort to port existing codes to CUDA and run them efficiently. First attempts to run lattice simulations on GPUs was reported in [2]. Currently a repository of CUDA codes for lattice QCD simulations is maintained at https://github.com/lattice/quda [3]. Recently pure gauge theory codes have also been ported to GPUs [4]. Nevertheless a lot of existing lattice QCD codes have been written, tested and optimized in FORTRAN. CUDA FORTRAN exists but is not very useful. It only uses wrappers around the CUDA C functions for interfacing with FORTRAN routines. For individuals or small groups a much more viable option is to use a directive based programming similar to OpenMP. OpenACC provides one such option.

2. QCD simulation algorithm and the GPU architecture

Computations in QCD boil down to estimating integrals over probability distributions. The expectation value of an operator $\mathcal{O}$ is

$$\langle \mathcal{O} \rangle = \int D\bar{\psi} D\psi DU \mathcal{O}[\bar{\psi}, \psi, U] e^{-S[\psi, U]}$$

(1)

where $e^{-S[\psi, U]}$ is the probability for the configuration $\psi$ and $U$. Here $\bar{\psi}$ and $\psi$ are grassman variables and $S$ has the structure $\bar{\psi} \mathcal{M} \psi + \mathcal{F}[U]$ where $\mathcal{M}$ is a lattice Dirac operator and $\mathcal{F}[U]$ is the lattice gauge action. Since there are no simple representations for grassman variables on the computer, they are integrated over and the resulting determinant re-exponentiated using auxiliary bosonic fields to give

$$\langle \mathcal{O} \rangle = \int D\phi \bar{\phi} D\phi DU \tilde{\mathcal{O}}[\bar{\phi}, \phi, U] e^{-(\phi^\dagger \mathcal{M}^{-1} \phi + \mathcal{F}[U])}.$$  

(2)

This is the form in which the expectation values are actually computed.

The algorithm of choice for performing these simulations is known as a Hybrid Monte Carlo (HMC) algorithm. The algorithm has three main components. The first step is the introduction of a Langevin noise by choosing random momentum variables from a gaussian distribution and a heat bath step in choosing random bosonic auxiliary fields. In the second step Molecular Dynamics (MD) equations are used to evolve the system through phase space for a given set of momenta and bosonic auxiliary fields. The third step is a Metropolis accept/reject step depending on the values of the Hamiltonian before and after the second step. This algorithm is exact in the sense that it has no systematic bias and satisfies the detailed balance condition.

The Langevin step requires between $10^6$ and $10^{10}$ random numbers at each step and one simulation requires about $10^4$ steps. Thus lattice QCD simulations require very high quality random number generators. The molecular dynamics step involves calculating the action of the inverse of a matrix on a vector and this is usually implemented through a conjugate gradient routine as $\mathcal{M}$ has positive definite eigenvalues. This is the most expensive step and takes up 85–95% of simulation time. Thus the conjugate gradient routine is the main target for parallelization. The accept/reject step is of course intrinsically serial.
The basic lattice QCD matrix vector operation requires 66 Flops and 120 bytes of data movement assuming single precision arithmetic. In our calculations, as in most others, double precision arithmetic is used. Thus lattice QCD computations are bandwidth limited. The GPU sits in a PCI slot on the motherboard. Thus the main bottle-neck for lattice QCD programs is the slow data movement between the CPU and the GPU. The maximum speed is about 5 GB/s or 8 GB/s depending on whether it is PCI 2.0 or 3.0. One possibility would be to avoid the CPU completely, but that is impossible as input-output and if-then clauses are evaluated on the CPU. Also with the Fermi GPU, BLAS functions and MPI calls are launched from the CPU. The main challenge in GPU programming for lattice QCD codes is how to get around these bottle-necks. To get any reasonable speed-up, at least the entire conjugate gradient must be on the GPU. Data must be copied from CPU to GPU at the start of the routine and the result copied back at the end of the routine.

The organization of the data on the GPU must be different from that of the CPU as the GPU memory is organized differently. As the CPU to main memory bandwidth is around 25 GB/s, the maximum performance without reusing data is about 13 GFLOPS. A similar calculation for the GPU would yield figures around 90 GFLOPS for a X2090 or 144 GFLOPS for a K40M. Therefore the main optimization would be to reuse data efficiently. Another big difference is that the caches on the GPU are much smaller than the CPU while there are a lot of registers. In fact the total memory offered by the registers is more than the cache. The bandwidth per core between the GPU main memory and the processor is also much smaller than that of the CPU. So it is much more beneficial to put variables directly into the register on the GPU. The CUDA compiler puts arrays into the main memory while scalar variables are put in registers. But our analysis shows that for lattice QCD calculations it would be more useful to put as many array elements into the registers as possible. One possible way to do that is to use code generators. Code generators can automatically convert arrays into lots of scalar variables. This can increase the performance of a lattice QCD CUDA code by a factor of several [5].

3. The OpenACC standard and code examples
OpenACC is a programming standard for parallel computing developed by Cray, CAPS, NVIDIA and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems [6].

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators. The directives and programming model defined in this document allow programmers to create high-level host+accelerator programs without the need to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown [7].

The OpenACC codes look very similar to OpenMP codes in their directive structure. One major difference though is the creation of a data region which tells the compiler when to move certain data items to the GPU or bring them back to the CPU. We illustrate the various constructs below using the conjugate gradient routine as an example. To highlight the similarity to OpenMP we keep the commented out OpenMP directives along with the OpenACC directives.

The part of the code to be off-loaded to the accelerator begins with the !$ACC data command and ends with the !$ACC end data command. The !$ACC data accepts lists of variables with attributes such as copy, copyin, copyout and create. The attribute copy implies that the values of the variables must be copied from the CPU memory to the GPU memory at the start of the data region and copied back to the CPU memory once the data region ends. The attribute copyin implies that the variable must be copied into the GPU memory but the result need not be copied back. Similarly copyout means that the variables need not be copied into the
GPU, but the results must be copied back. Finally the attribute create means those variables are local to the GPU they do not require transfer between the CPU and the GPU memory.

subroutine congrad(nitcg)
   ... All kinds of definitions and declarations ...

   !$ACC data copy(nitcg,alpha,betad,betan) copyin(nx,iup,idn,u,r)
   !$ACC+ copyout(x,y) create(ud,ap,atap,p)

   *call linkc_acc
   !$OMP parallel do default(shared)
   !$ACC parallel loop collapse(2) reduction(+:betan) present(p,r,x)
   do l = 1, mvd2
     do ic=1,nc
       p(l,ic) = r(l,ic) ; x(l,ic) = (0.,0.)
       betan=betan+conjg(r(l,ic))*r(l,ic)
     end do
   end do

   ! betan=real(zdotc(mv3d2,r,1,r,1))
   !$ACC update host(betan)
   if (betan.lt.delit) go to 30

   !$ACC parallel present(beta,betan,betad,alphan)
   beta=betan/betad ; betad=betan ; alphan=betan

   !$ACC end parallel
   do nx = 1, nitrc ! Main loop of conjugate gradient begins
     nitcg ← nitcg+1 ; ap = 0
     call fmv(0,mvd2,ap,p) ! Matrix-vector multiplication

     alphad=⟨ap,ap⟩ + ⟨p,p⟩ ; alpha=alphan/alphad
     atap ← p ; x ← x + alpha * p
     call fmv(atap,ap) ! Matrix-vector multiplication

     r ← r - alpha * atap
     betan=(r, r)

   !$ACC update host(betan) ! Exit condition evaluated on CPU
   if (betan .lt. delit) go to 30
   beta=betan/betad ; betad=betan ; alphan=betan
   p ← r + beta * p
   end do ! Main loop of conjugate gradient ends

30 continue

   * y = 0 ! Solution on the second half lattice
   call fmv(mvd2,mv,y,x) ! Matrix-vector multiplication

   * !$ACC end data
   return

As one can see from the above example, the code requires no change from the original CPU code apart from the insertion of various directives. The nature of the directives is quite close
to that of OpenMP with a few extra attributes. One such attribute is present followed by a list of variables. This tells the compiler that the variables are already present in the GPU memory and the compiler need not copy it from the CPU. Another directive special to the accelerators is update host(). This tells the compiler to copy the value of the variable at that point in execution to the CPU from inside a data region. This is often used for evaluating if-then conditions on the CPU when the variable on which the condition depends is evaluated on the GPU. The directives parallel loop and parallel work in the same way as in OpenMP.

In the Fermi GPU, the MPI calls and BLAS functions are launched from the CPU. So we found it was faster to implement the vector operations using parallel loops rather than make BLAS calls. MPI calls unfortunately still need data transfer between the CPU and the GPU and that requires one more construct update device() which works like the update host() but in the reverse direction. We illustrate a basic MPI call below.

```c
ap_loc ← 0
!$ACC parallel loop present(u,ud,ap_loc,p,iup,idn)
do l = base+1, base+nvd2
v1 = ap_loc(1,l-base)
    ...
_lines identical to single node version
    ...
ap_loc(3,l-base) = v3
enddo
!$ACC update host(ap_loc)
call MPI_ALLGATHER(ap_loc,3*nvd2,MPI_DOUBLE_COMPLEX,
    + ap,3*nvd2,MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD,ierr)
!$ACC update device (ap)
```

In this example the local array ap_loc is combined to form the global array ap which is made available to all processes. For multi-GPU applications one must worry about synchronizing the data between the CPU and the GPU after each communication call. Usually in OpenACC this can be handled by compiler options.

4. Results
In this section we describe the performance we obtained for our programs. These are either a single node two flavour ($N_f = 2$) RHMC code for staggered or a single node two flavour Wilson fermion lattice QCD simulation program with OpenMP and OpenACC directives, on different kinds of hardware - compiler combinations. All calculations were performed in double precision.

In figure 1 we present the performance in GFLOPS [9] of a single node $N_f = 2$ RHMC code for staggered fermion lattice QCD simulation on three different hardware - software combinations. While the Ivybridge node with the Cray compiler is faster due to its higher clock speeds upto two threads, beyond that the Haswell node with the Intel compiler is faster. The Cray node with the K20X GPU turns out 3 to 4 times faster in this case compared to the node only with CPUs. For the GPU code almost all the parallel parts of the program have been pushed onto the GPU with very little data transfer required between the CPU and GPU. This results in the flat behaviour for the GPU code almost independent of the number of CPU threads.

The conjugate gradient routine achieves a higher performance and we present its performance in table 1. The performance of the conjugate gradient routine had to be estimated from a slightly slower version of the program as in the final version, most of the molecular dynamics trajectory
Effective GFLOPS

Figure 1. Performance of a single node \( N_f = 2 \) RHMC code for staggered fermion lattice QCD simulation on three different hardware - software combinations. The + corresponds to one Intel 10 core Ivybridge CPU at 2.8 GHz with 8 cores for computing on a Cray XC30 using the Cray compiler, the \( \times \) corresponds to a server with two 10 core Haswell CPUs at 2.3 GHz and the \( \ast \) corresponds to one node of a Cray XC30 with one CPU as in + and a K20X GPU using the Cray OpenACC compiler.

is inside a single ACC data region and timing estimates of a single routine without slowing down the program is difficult to get. Our best estimate is that on the K20X the conjugate gradient achieves about 55 GFLOPS.

| # of threads | 1    | 2    | 4    | 8    | 16   |
|--------------|------|------|------|------|------|
| Ivybridge (2.8 GHz) | 3.870 | 6.753 | 9.204 | 9.609 | -    |
| Haswell (2.3 GHz)   | 3.292 | 6.236 | 10.388 | 12.513 | 12.244 |

Table 1. Performance (in GFLOPS [9]) of the conjugate gradient routine in a two flavour RHMC code for simulation of staggered fermions for the Ivybridge and Haswell CPUs.

Gustafson’s law states that better parallelization can be achieved for larger data sizes. Figure 2 is an illustration of this. With the lattice size changing from \( 16^4 \) to \( 24^4 \), better parallelization is achieved. For lattice size \( 28^4 \), a position independent code has to be built and the program has to be linked dynamically. This is slower than the default statically linked executable on the Cray and very likely this results in the drop in performance.

Global memory access in GPU is slow. Therefore an optimal ordering of array indices is
Figure 2. Performance of the Cray OpenACC compiler with varying data size. The x-axis denotes the length of one side so that the volume of the lattice is $L^4$.

| # of threads | 1   | 2   | 4   | 8   | 16  |
|--------------|-----|-----|-----|-----|-----|
| Order A      | 673.6 | 623.1 | 522.7 | 474.5 | 447.7 |
| Order B      | 651.7 | 505.9 | 325.2 | 243.2 | 194.7 |

Table 2. Timing in seconds of the conjugate gradient routine in a Wilson fermion simulation program for two different storage schemes of the vector on Fermi X2090 GPU. Order A corresponds to array(3,4,nsite) while Order B corresponds to array(nsite,4,3) with nsite = 20^4.

essential for arrays being declared on the GPU. In table 2 we present the timings of the conjugate gradient routine for the Wilson fermion simulation program for two different storage schemes of the vectors. There is a 3.5 times difference in speed depending how the array is defined. Order A is faster on the CPU while order B is faster on the GPU (on FORTRAN compilers). Thus while the conjugate gradient speeded up another routine got slowed down by a factor 2. Nevertheless since the conjugate gradient takes up most of the computation time, there was an overall speed-up by more than a factor 2 for storage order B.

5. Discussion
- In terms of both cost and power consumption, GPUs provide affordable supercomputing resources.
- OpenACC seems to be a comparatively easy path to port ones codes from the CPU to the GPU as it avoids a lot of the accelerator memory management issues and other associated...
issues such as initializing the accelerator and shutting it down or generating the individual accelerator kernels.

- The OpenACC coding effort is only marginally higher than OpenMP. Almost each OpenMP directive can be replaced with a OpenACC directive. Only additional directives are creation of a data region to demarcate the portion of the code to be off-loaded to the accelerator and updating the host or the device from within a data region.
- Real speedup is possible only when a comparatively large chunk of the computation (compared to the amount of data transferred between the CPU and the GPU) such as the whole conjugate gradient is performed on the GPU.
- Performance of single GPU is roughly equivalent to 128 cores of cluster with QDR infiniband interconnect [8].
- For the programs analyzed, the advantage of the GPU increases with increasing data size.
- There can be a huge penalty in performance (several times) for data access on the GPU if the data is not optimally organized in the GPU memory.
- Comparing between an optimized pure CPU MPI code and an OpenACC code on a single Cray node, the MPI code is faster for lattice sizes of about $20^4$. Beyond that the OpenACC code is faster.
- Further performance gains can be obtained by using mixed-precision routines and improved storage schemes.

Acknowledgments
We would like to acknowledge ILGTI-TIFR for funding the GPU portion of the Cray on which these studies were carried out. IACS for funding the rest of the Cray without which the GPU portion wouldn’t run as well as the Cray India team for help at various stages during the development of the OpenACC codes.

References
[1] Ukawa A for CP-PACS and JLQCD Collaborations 2002 *Nucl. Phys. Proc. Suppl.* **106** 195
[2] Egri G I, Fodor Z, Hoelbling C, Katz S D, Nogradi D and Szabo K K 2007 *Comput. Phys. Commun.* **177** 631
[3] Clark M A, Babich R, Barros K, Brower R and Rebbi C 2010 *Comput. Phys. Commun.* **181** 1517
[4] Cardoso N and Bicudo P 2013 *Comput. Phys. Commun.* **184** 509
[5] Chakrabarty A and Majumdar P 2012 Hybrid Monte Carlo with Wilson Dirac operator on the Fermi GPU Preprint arXiv:1207.2223 [hep-lat]
[6] http://en.wikipedia.org/wiki/OpenACC
[7] http://www.openacc-standard.org/
[8] Majumdar P 2014 Lattice Simulations using OpenACC compilers *PoS* **LATTICE2013** 031
[9] All FLOP measurements were done using the craypat utility on a Cray XC30. The precise command used was `pat_build -w -T function-name executable`. This measurement was done for a single thread version of the program and the effective GFLOPS were calculated by dividing the number of FLOP thus obtained by the run-time of the OpenMP or OpenACC program.