Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Extending the QUDA library for Domain Wall and Twisted Mass fermions

Alexei Strelchenko[1], Marcus Petschlies[2] and Giannis Koutsou[3]

CaSToRC, Nicosia 2121, Cyprus

Abstract

We extend the QUDA library, an open source library for performing calculations in lattice QCD on Graphics Processing Units (GPUs) using NVIDIA's CUDA platform, to include kernels for non-degenerate twisted mass and multi-gpu Domain Wall fermion operators. Performance analysis is provided for both cases.

1. Introduction

Major improvements in the calculation and prediction of hadronic observables require large amounts of computer resources, of the order of hundreds of Tflop/s of sustained performance. The main objective of this project was to develop the necessary tools, so that the calculation of some key hadronic observables, which up to now where too demanding to be computed, will be made feasible. One example of this kind is disconnected diagrams, or fermion vacuum loops, that have typically been omitted from lattice QCD calculations due to their large computational cost, and the systematic uncertainties introduced by this omission is still an open issue. This computational intensive task could be harnessed, however, using Graphics accelerators (GPUs) which make it possible to investigate various numerical techniques in LQCD [1].

QUDA is the most well established community code for carrying out the time-consuming components of an LQCD computation. The QUDA library, a package of optimized CUDA kernels and wrapper code [2], has already attracted a wide developer community and is currently being used in production at U.S. national laboratories, as well as in Europe. QUDA includes optimized implementations of a number of different discretizations of the continuum QCD fermion operator, such as the Wilson and Staggered fermion actions. It also implements a range of iterative solvers for these fermion actions, i.e. CGNE (for normalized equations), BiCGStab and recently the domain decomposition solver.

[1] a.strelchenko@cyi.ac.cy
[2] m.petschlies@cyi.ac.cy
[3] g.koutsou@cyi.ac.cy

For the first part of the project we implemented the non-degenerate twisted mass fermion operator [3], in the context of the preparatory access project “Lattice QCD at the Peta-flops scale” which was awarded computer time on Curie in the first round of preparatory access calls.

The second part of the project consisted of MPI-parallelization of the Domain Wall fermion [4] operator, an operator, which was already available in QUDA albeit only for a single GPU.

2. QUDA Overview

The QUDA library consists of GPU code with a number of implemented fermion operators (and other helper kernels) written in 'C for CUDA' and a host interface written in C++, which employs an Object-Oriented Programming paradigm. In particular, each type of Dirac operator as well as host and device spinor fields are encapsulated into separate classes with all the necessary functionality for any third party client code such as, e.g., the Chroma package etc. QUDA uses also a special interface designed for message-passing frameworks – MPI or QMP [5]. The latter “QCD message-passing” standard was developed to provide a simplified subset of communication primitives most used by LQCD codes, allowing for optimized implementations on a variety of architectures, including purpose-built machines that lack MPI.

In this section we present an overview of the key ideas lying behind fine-grid (single GPU) and coarse-grid (multi-GPU) parallelizations in the QUDA library.

a) QUDA kernels.

The main kernel operation of QUDA is sparse matrix-vector multiplication (which represents a re-casting of various fermion operators). The general approach here is to assign a single GPU thread to each site of the lattice, when each thread performs all operations required to update that site given the stencil operator. The most essential optimizations are related to corresponding memory operations to diminish effects of relatively low arithmetic intensity of sparse matrix-vector multiplications on overall performance. These include several schemes [2].

- To benefit memory coalescing, where this is possible, it is necessary to re-oder the spinor and gauge field arrays using double2 or float4 built-in structures. It's also possible to reduce effects on non-coalescing access (e.g., on lattice boundaries) using the GPU texture cache. Note that the NVIDIA Fermi architecture provides with L1/L2 caches that may serve the same purpose.

- The SU(3) gauge link matrices, which are effectively 3×3 unitary matrices with positive unit determinant, can be parametrized from 18 real numbers to 12 or 8 real numbers. In this way memory traffic can be reduced for additional floating point computations required for reconstruction the matrices into their full representation of 18 real numbers.

- Sparsity of the Dirac matrix can be further increased by appropriately choosing the basis of the projection matrices such that the projectors connecting neighbours in the time dimension have non-zero elements in only two of their four columns. As a result, this will reduce the number of spinor components to two (out of four) when computing contributions from the hopping term in the time direction.

- To reduce memory traffic one can utilize a 16-bit fixed-point storage format which in conjunction with mixed-precision linear solvers allows one to achieve higher performance without loss in accuracy.

So the typical work-flow of the kernel consists in loading a spinor, loading the gauge links in each direction, applying projectors on the spinor, applying the gauge link matrix on the projected spinor and finally accumulating the result in device registers (or storing them in global buffer when all contributions from the hopping terms are computed).

b) Multi-GPU partitioning [6], [7].

One of the major disadvantages of contemporary GPUs is a relatively small amount of memory available for a single device (limited at best by 6GB). Therefore, to perform computations for large-size lattices one has to be able to distribute the job among multiple GPUs using, e.g., MPI.

In the QUDA library, parallelization is carried out by partitioning the lattice in all (four) dimensions where each device processes a local 4-dimensional subvolume while updating spinors on boundary sites (or "faces") relies on data transfer from adjacent GPUs. The received data is accumulated in separate buffers (the "ghost zones") on the device, which are placed in memory after the local spinor field. Accordingly, computational kernels fall into two categories. Namely, the so-called interior kernels that compute the spinors interior to the subvolume, as well as any contributions to spinors on the boundaries that do not require data from the ghost zones, and the exterior kernels that require data from ghost zones, respectively. Partitioning of the lattice is accomplished hierarchically starting from the direction corresponding to the slowest varying index (i.e., in the T-direction), and ending with the direction corresponding to the fastest index.

In the case of T-partitioning (and 4-dimensional fermion actions), the boundary sites are already contiguous in the memory, so sending them to host does not require any extra gathering operations. Contrary to this case, the ghost spinor data for the other three dimension must be additionally composed into contiguous buffers in GPU global memory by means of separate GPU kernels.

The QUDA library employs the CUDA Streaming API to overlap computation with MPI (as well as Device-Host) communications. For example, two streams per dimension are used for gathering and exchanging spinors in the forward direction and in the backward direction, respectively. One additional stream is used for executing the interior and exterior kernels, giving thus a total of 9 streams. The gather kernels for all dimensions are launched on the GPU immediately so that communication in all directions can begin. The interior kernel is executed after all other kernels finish, overlapping completely with the communication. QUDA also uses different streams for different dimensions so that the different communication components can overlap with each other. It should be noted, however, that while the interior kernel can be overlapped with communications, the exterior kernels must wait for arrival of the ghost data. As a result, the interior kernel and subsequent exterior kernels are placed in the same stream, and each exterior kernel blocks waiting for communication in the corresponding dimension to finish. The recent version of the package exploits GPUdirect technology eliminating overheads coming from unnecessary system memory copies.

2.1 Implementation of non-degenerate twisted-mass operator

The main peculiarity of the non-degenerate twisted mass fermion operator consists of the presence of off-diagonal matrix elements in flavour subspace. Since these components of the matrix mix spinor flavours one has to apply the operator on both flavours resulting in more complicated compute kernels.

The straightforward approach in the non-degenerate case is to re-use the gauge field to avoid an extra memory transaction while computing contributions from each spinor flavour. Thus, the single-gpu compute kernel's design can be described as follows:

1. Assign each GPU thread to a single lattice site (as in the case of plane Wilson-Dirac dslash matrix). Then, for each hopping direction (i.e. 8 directions in total for 4 dimensional lattice) we perform the operations:
2. Load a compressed gauge link connecting corresponding neighbouring sites and reconstruct it on the GPU (using either the 8- or the 12-reconstruction scheme);
3. Load the first spinor flavour component from the neighbour site and apply the projection operator and gaugelink on this component;
4. Accumulate the results in output registers;
5. Perform steps (3) and (4) with the second flavour. Finally, after collecting contributions from all hopping directions:
6. Perform twist rotation on the two flavours of the spinor;
7. Store results in the GPU global buffer;
8. All memory operations for flavor spinors from step (7) are performed via WRITE_FLAVOR_SPINOR_{DOUBLE2/FLOAT4/SHORT4} macros defined in *io_spinor.h* file.

We introduced a new kernel generator in *tm_ndeg_dslash_def.h* header file to generate the complete set of CUDA kernels during the pre-processing phase of the code compilation.

To include the non-degenerate twisted-mass kernels in the whole framework the following modifications in the host code where made.

First, we added several extra constants in a number of enumerated types adduced in the *enum_quda.h* header file:

QUDA_NDEGTWISTED_MASS_DSLASH in QudaDslashType;
QUDA_NDEGTWISTED_MASS{PC}_DIRAC in QudaDiracType;
QUDA_TWIST_DOUBLET (= 2) in QudaTwistFlavorType.

In *dirac_quda.h*, in classes DiracParam, DiracTwistedMass and DiracTwistedMassPC, we added new data members that are responsible for flavour mixing, we also modified several methods to include the non-degenerate case functionality (defined in *dirac_twisted_mass.cpp*). These are:

- DiracTwistedMassPC::Dslash method that implements application of the pure non-degenrate twisted mass dslash operator on a parity spinor;
- DiracTwistedMassPC::DslashXpay method which is the same as above plus an extra addition operation on the parity spinor;

- DiracTwistedMassPC::M method that implements application of the preconditioned dslash operator on a parity spinor;
- DiracTwistedMassPC::prepare method which is responsible for the preparation of a source for the preconditioned solver;
- DiracTwistedMassPC::reconstruct method which is needed for retrieving the solution from the preconditioned solver.

In the multi-GPU version of the code we added new class constructors for DiracTwistedMass and DiracTwistedMassPC classes in order to enable ghost zones for spinor flavor doublets.

Next, in *dslash_quda.cu*, we modified the twistedMassDslashCuda and the twistGamma5Cuda interface functions to add an extra functionality for the non-degenerate twisted mass operator (with some modifications in appropriate methods of the TwistedDslashCuda template class that serves as a front-end to the GPU kernels for all client functions from *dslash_quda.cu*).

We made a number of minor modifications for several methods in the ColorSpinorParam, ColorSpinorField and CudaColorSpinor classes to take into the account 2-dimensional flavour subspace (needed for correct allocation of memory etc.).

Finally, for multi-GPU version we provided with packFaceNdegTM routine in *pack_face_def.h* and modified cudaColorSpinorField::packGhost method defined in *cuda_color_spinor_field.cpp* that implements ghost field packing for non-degenerate case.

2.2 Implementation of MPI Domain Wall operator

As in the case of twisted mass inverters, our improvements concerned both device and host components of the package. Our contribution here was to introduce MPI functionality to the existing Domain Wall routines (the single GPU version was written by Joel Giedt). Here we adopted an approach by which the splitting is performed in 4-dimensions, while the 5th dimension remaining untouched. There were two major reasons for this. On the one hand it is relatively easy to modify the code in this particular case, and on the other, one can further improve the performance of the compute kernels by employing gauge link re-use strategies similar to the non-degenerate twisted mass case described above (we did not investigate this possibility in the current version of the code though).

For the device, or the GPU code, we introduced exterior kernels for the domain wall dslash operator, while the interior kernel was left unchanged. As has been mentioned in section 4, these are responsible for computations of the ghost spinors (in each parallelized space-time direction).

The only consideration is the ordering used for assigning threads to sites that is slightly different than in the case of the 4-dimensional Wilson-Dirac compute kernels. For the interior kernel, the global one-dimensional CUDA thread index is assigned to sites of the 5-dimensional sublattice, in the same way that the spinor data is ordered in memory, with X being the fastest running index and the 5th-dimension axis the slowest. It is thus guaranteed that all spinor and gauge field accesses are coalesced. In the X,Y,Z,T exterior kernels, only the destination spinors are indexed in this way, while the ghost spinor and gauge field are indexed according to a different mapping. This makes it impossible to guarantee coalescing for both reads and writes. The index mapping functions as well as interface routines for ghost buffer packing are added in *pack_face_def.h* (packFaceDW routine).

Now let us overview the changes made in the host code. Since the QUDA library already incorporates an implementation for single GPU Domain Wall fermions, the only essential modifications done were for the MPI/QDP interface classes responsible for the message passing operations.

In particular, in class FaceBuffer, the class constructor defined in *face_mpi.cpp* and *face_qdp.cpp* files was changed to take into account an extra dimension (needed for correct allocation of memory etc.).

Next, in the cudaColorSpinorField::packGhost method defined in *cuda_color_spinor_field.cpp* we added code for packing ghost spinor fields for DW fermions (as a call for the function packFaceDW() defined in *pack_face_def.h* mentioned above). Finally, in the class cpuColorSpinorField, in the cpuColorSpinorField::allocateGhostBuffer and cpuColorSpinorField::packGhost methods, we added an extra dimension for allocating and packing ghost buffers on the host (used for the CPU dslash implementation only).

Also, corresponding changes where done in DiracDomainWall and DiracDomainWallPC classes to take into account MPI communications, as well as in domainWallDslashCuda(), the host front-end interface routine for Domain Wall CUDA kernels defined in *dslash_quda.cu*

3. Performance analysis

The kernels we adapted are available for double, single and half precision in order to exploit a novel mixed precision technique [2], which allows one to obtain the solution in full double precision accuracy while using only single or half precision arithmetics for the bulk of the computation. Additionally, even-odd (or red-black) preconditioning is used according to the problem at hand. Therefore our work in including the non-degenerate twisted-mass fermion operator consisted of implementing it in all three arithmetic precisions and for even-odd ordering.

Below we present performance results for the two operators we have implemented. We timed both the matrix-vector multiplication as well as the iteration time when using the operator in an iterative solver.

a) Twisted-mass non-degenerate operator (single-GPU execution)

The results are given for a 32×64 lattice with parameters κ=0.163272, μ-bar=0.19 and ε= 0.15. The runs were performed on NVIDIA M2090 GPUs. This means a peak floating point efficiency of ~1Tflop in single precision and about half of that in double precision. For the matrix-vector multiplication, we achieve:

- 33 Gflops/sec in double precision
- 136 Gflops/sec in single precision and
- 162 Glops/sec in half precision

To compare with typical performance on the BlueGene/P supercomputer (IBM PowerPC450). Since the GPU version uses a mixed precision solver, to compare fairly we quote the time to invert for the same right-hand-side for the same tolerance, which can in principle require a different number of iterations between the two architectures. On the GPU, for a tolerance of 10^{-10}, we require 174.3 seconds for 1116 iterations in mixed precision, compared to 184 seconds for 1383 iterations for the same tolerance on 256 cores of BG/P (64 nodes, with a theoretical peak performance of 870.4 Gflops in double precision). This means that the GPU accelerated inverter on a single device with a power consumption of less than 225W provides the same performance level as 64 BlueGene/P nodes which require about 0.34×13.6×64 = 2559W.

b) Parallelized Domain Wall operator

Strong scaling results for the MPI Domain Wall operator are given for a $16^3 \times 192 \times 8$ lattice on NVIDIA M2070 GPUs. We note that this is a rather unnatural choice for a problem size and is included hear purely to demonstrate the best case code scaling. We compare the performance of the double-half and double-single mixed precision CG solvers, presented in Fig. 1. Here we observe almost linear scaling for both cases.

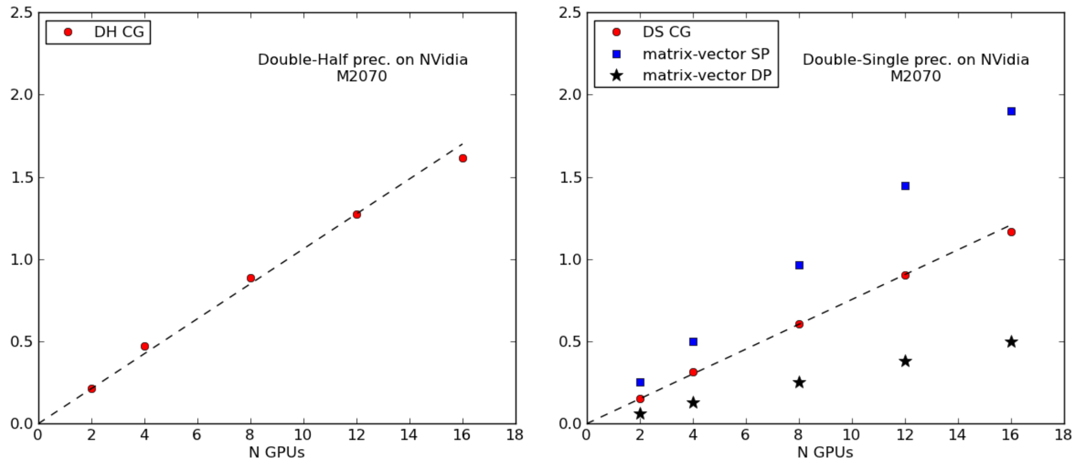

Figure 1: Performance results for double-half (left) and double-single (right) mixed precision Domain Wall CG solver

The double-half case gives for 16 GPUs ~1618 Gflops which is around 1.38 times the performance of the double-single solver (1170 Gflops). The main contribution to the solver in both cases comes from the low-precision matrix-vector products, namely from the half and single precision dslash operators respectively. In particular, on 16 GPUs, the half-precision dslash sustains ~3342 Gflops against 1900 Gflops of the single-precision dslash.

Results for our MPI Domain Wall operator are also given for a more natural choice of problem size. A $28^3 \times 64 \times 4$ lattice, with mass parameters: $M_5=-1.0$ and $M_0 = 0.0138$ was used, a lattice which has been employed for several hadron structure calculations in the past [8, 9]. The performance, compared with that obtained on a CPU system, is given in Fig. 2.

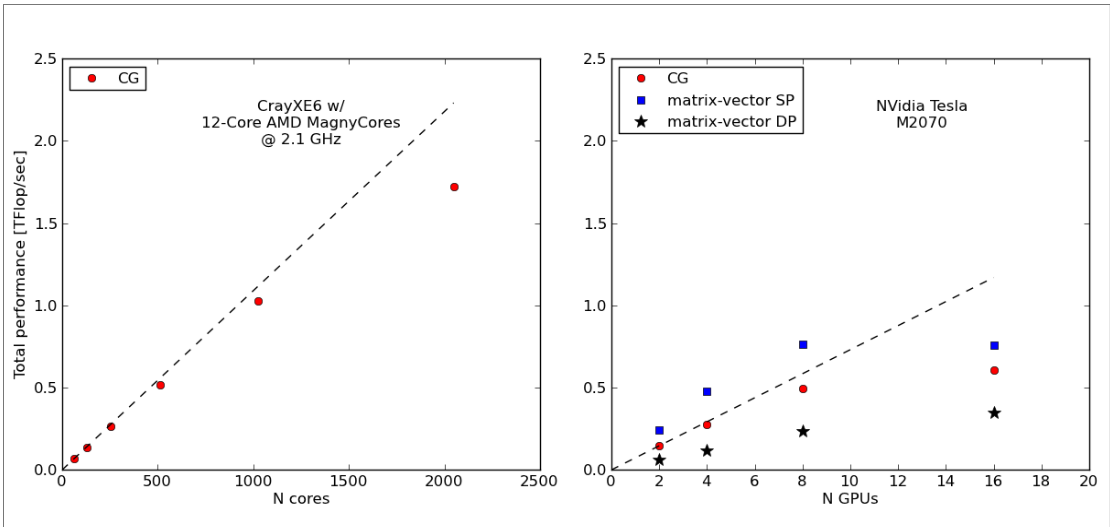

Figure 2: Domain Wall solver performance on CPU (left) compared to our QUDA implementation for GPUs (right). For the GPU, apart from the CG solver performance (red circles) we show the matrix-vector multiplication performance for double-precision (black asterisks) and single precision (blue squares).

Here we see a degradation of the scaling in the CG, which is the effect of much higher surface-to-volume ratio when going up to16 GPUs (0.25 here, compared to 0.08 for the 'unnatural' lattice volume used in Fig.1). The surface-to-volume ratio is a quantization of the fact that the realistic volume becomes more communication-bound when we go beyond 8 GPUs, or 4 nodes. However, we can say that on average the GPU implementation offers a speed-up of around ×4.7 times when comparing the per GPU performance to the per CPU-socket performance. Namely, QUDA's performance is around 61.3 Gflops/sec per M2070 GPU, compared to the average of 13 Gflops/sec we obtain for a single CrayXE6 “MagnyCores” 12-core socket at 2.1 GHz.

4. Consideration and conclusion

Overall the project objectives where met; the QUDA library was extended to implement two extra fermion operators thus extending the potential user base of this software package. Implementation of the non-degenerate twisted-mass operator will allow utilizing GPUs for a wider set of problems than what was already possible with QUDA, problems which are relevant to the European Twisted Mass Collaboration, one of the larger lattice QCD collaborations in Europe.

For the case of the Domain Wall fermion operator, parallelizing the matrix-vector multiplication effectively *enables* use of GPUs for this branch of lattice QCD, since this operator is particularly demanding on memory, due to the fact that it is defined on a five-dimensional grid. At realistic lattice sizes the problem does not fit on a single GPU, meaning that if one is to use GPUs, a multi-GPU implementation is necessary.

Finally, the Domain Wall operator requires further optimizations that will include enabling peer-to-peer communications between GPUs on node as well as gauge field re-use in the fifth dimension.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557. The work is achieved using the PRACE Research Infrastructure resources.

References

1. Alexandrou, C. and Hadjiyiannakou, K. and Koutsou, G. and Cais, A. 'O and Strelchenko, A., “Evaluation of fermion loops for eta and nucleon scalar and electromagnetic form factors”, Comput. Phys. Commun., V **182**, 2012, p.1215
2. Clark, M. A. and Babich, R. and Barros, K. and Brower, R. C. and Rebbi, C, “Solving Lattice QCD systems of equations using mixed precision solvers on GPUs”, Comput. Phys. Commun., V **181**, 2010, p.1571
3. R. Frezzotti and G. C. Rossi, “Chirally improving Wilson fermions. II: Four-quark operators”, JHEP **0410** (2004) 070 [hep-lat/0407002]
4. D. B. Kaplan, “A Method for simulating chiral fermions on the lattice”, Phys. Lett. B **288** (1992) 342 [hep-lat/9206013]
5. http://usqcd.jlab.org/usqcd-docs/qmp/
6. Babich, R., Clark, M. A. and Joo, B., “Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics”, Proceedings of SC10, 2010
7. Babich, R., Clark, M. A., Joo, B., Shi, G., Brower, R. C. and Gottlieb, S., “Scaling Lattice QCD beyond 100 GPUs”, Proceedings of SC11, 2011
8. C. Alexandrou, E. B. Gregory, T. Korzec, G. Koutsou, J. W. Negele, T. Sato and A. Tsapalis, “The Δ(1232) axial charge and form factors from lattice QCD”, Phys. Rev. Lett. **107** (2011) 141601 [arXiv:1106.6000 [hep-lat]].
9. C. Alexandrou, G. Koutsou, H. Neff, J. W. Negele, W. Schroers and A. Tsapalis, “The nucleon to Delta electromagnetic transition form factors in lattice QCD”, Phys. Rev. D **77** (2008) 085012 [arXiv:0710.4621 [hep-lat]].
10. QUDA with non-degenerate twisted mass operator can be obtained on: https://github.com/lattice/quda/tree/quda-nondeg-twisted-mass