QCD on GPUs: cost effective supercomputing

M. A. Clark[*ab]

[a]*Harvard-Smithsonian Center for Astrophysics, 60 Garden St, Cambridge, MA 02138, USA*
[b]*Initiative in Innovative Computing, Harvard University School of Engineering and Applied Sciences, 29 Oxford St, Cambridge, MA 02138, USA*
E-mail: mikec@seas.harvard.edu

The exponential growth of floating point power in graphics processing units (GPUs), together with their low cost, has given rise to an attractive platform upon which to deploy lattice QCD calculations. GPUs are essentially many (O(100)) core chips, that are programmed using a massively threaded environment, and so are representative of the future of high performance computing (HPC). The large ratio of raw floating point operations per second to memory bandwidth that is characteristic of GPUs necessitates that unique algorithmic design choices are made to harness their full potential. We review the progress to date in using GPUs for large scale calculations, and contrast GPUs against more traditional HPC architectures.

---

[*]Speaker.

1. Introduction

The lucrative gaming market has lead to the exponential growth in the floating point performance of graphics processing units (GPUs) which has far outstripped the increase in performance of traditional CPUs. Coupled with a similarly increasing memory bandwidth, GPUs represent a very attractive platform upon which to deploy lattice QCD calculations. The combination of CPU with GPU represents an example of a *heterogeneous architecture*, which is rapidly becoming the norm for high performance computing.

This article focuses on how to utilize GPUs effectively for QCD calculations, reviews the status quo and looks to future opportunities for using GPUs. The outline is as follows: in §2 we review current GPUs, in particular the CUDA platform, §3 describes how to effectively implement the action of the Dirac operator upon GPUs, §4 focuses upon mixed precision solvers, §5 considers prospects for multiple GPU parallelization, §6 reviews effective price performance of current GPU solutions, and in §7 we present our conclusions.

2. Graphics Processing Units

There are currently two companies highly vested in the high performance GPU market: AMD [1] and NVIDIA [2]. Both companies offer highly programmable GPUs, and their current top of the range products offer similar bandwidth to their respective memories. Historically, programming GPUs required the use of graphics APIs, e.g., OpenGL, which are not suitable for deploying typical scientific applications. In a pioneering work by Egra *et al* [3] lattice QCD was shown to map well to GPU architectures, obtaining exceptional price performance. However, it was not until the introduction of NVIDIA's CUDA (Compute Unified Device Architecture) platform which lowered the barrier to GPU computing that resulted in an explosion of interest in harnessing GPUs for lattice QCD calculations [4, 5, 6, 7]. Since almost all current focus is upon the CUDA platform, we shall here on in consider NVIDIA's CUDA architecture only, except where noted.

Card	Generation	Cores	GBs^{-1} Bandwidth	Gflops 32-bit	Gflops 64-bit	GiB Device RAM
GeForce 8800 GTX	G80	128	86.4	518	-	0.75
Tesla C870	G80	128	76.8	518	-	1.5
GeForce GTX 280	GT200	240	142	933	78	1.0
Tesla C1060	GT200	240	102	933	78	4.0
Tesla C2070	Fermi	512	~ 200	1260	630	6.0

Table 1: Specifications of representative NVIDIA graphics cards. The G80 was the first GPU architecture to support CUDA, GT200 is the current generation which introduced double precision and Fermi is the next generation due for release early in 2010. The GeForce range represent the consumer gaming cards, and Tesla is the professional range which typically have a significantly increased device memory (at a significant price premium) [8].

GPUs are typified by massive single precision floating point performance and a very wide, and hence fast, bus to their on card memory (here on referred to as device memory). They consist

of hundreds of cores (called stream processors) grouped together in many multiprocessors. With each passing generation the number of cores has historically doubled, and together with a widening of the memory interface, this has lead to GPUs quickly outpacing the performance of traditional CPUs (Table 1).
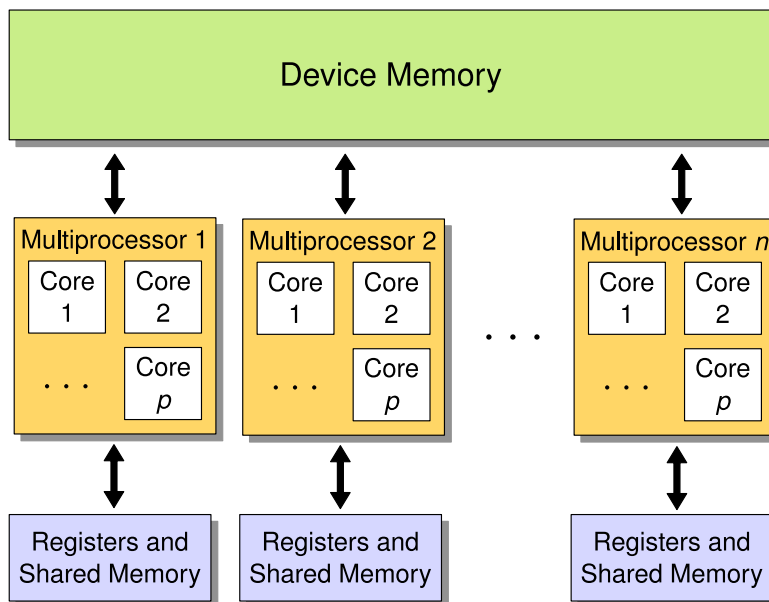

Figure 1: Architecture of a modern NVIDIA graphics card. In NVIDIA's nomenclature, cores called *stream processors* (or *scalar processors*), and in the GT200 generation each multiprocessor has eight such cores, 64 kB of registers and 16 kB of shared memory.

Current generation GPUs lack a traditional cache, and so hide latencies by running many threads concurrently:[1] if a group of threads encounters a stall due to memory latency, a new group of threads will be swapped in their place and run until either completion or indeed until they stall. GPUs are designed to run thousands of threads concurrently, and although they have a very large register file, in addition require fined grained parallelism to be fully exploited. The *occupancy* represents the fraction of the maximum number of possible threads that is achieved by a given *kernel* (a function that runs on the GPU). In general, a greater occupancy will lead to great performance. In addition, each multiprocessor has a small user managed cache, the shared memory, this allows communication between the cores within a multiprocessor. The size of the device memory varies from model to model: current gaming (GeForce) cards have typically 1 GB on board whereas the professional Tesla cards have up to 4 GB on board and are available in various suitable for high performance computing. In order to realize the peak bandwidth to device memory, reads and writes from consecutive threads on a given multiprocessor must access adjacent regions of memory, such transfers are known as being *coalesced*. For full coalescing each thread in a group of 16 threads must read data in either 32, 64 or 128 bit consecutive chunks. Although GPUs have extremely large memory bandwidth, the peak single precision floating point capacity to device memory band-

---

[1]768 threads per multiprocessor on current hardware.

width ratio means that they are extremely bandwidth limited and so efficient use of the registers and shared memory to minimize communication to device memory (even at the expense of introducing additional floating point work) is of paramount importance to obtain high performance.

Double precision capability to GPUs was introduced with NVIDIA's GT200 generation, however its use invokes a large penalty relative to single precision. This has motivated the use of mixed precision methods, specifically linear solvers, to circumvent this defect [3, 6, 7]. The next generation architecture from NVIDIA, *Fermi* [9], will address this issue (as as well as introducing ECC protection) with only a factor factor two difference between the peak single and double precision floating point arithmetic rates (Table 1).

The proverbial elephant in the room is the PCI express bus, with typical bandwidth 5 GB^{-1}, through which all communication with the host CPU must take place. Minimizing communication through this bus is of utmost importance in order to maximize performance. This typically requires that the GPU is not used as an *accelerator* for a single function, rather the complete algorithm must be deployed on the the GPU.

The most popular programming interface to CUDA is the C for CUDA interface. This presents the application programmer with a massively threaded C-like programming language from which the underlying graphics hardware can be effectively exploited. From a programming point of view, threads are grouped together in thread blocks, consisting of a minimum of 32 threads (though at least 64 are required for optimum performance), and thread blocks are arranged in a grid. Each of these thread blocks will run on a multiprocessor, and ideally (for optimum latency hiding) multiple thread blocks will be running concurrently on a multiprocessor. Communication between threads in a thread block is possible through the shared memory, however, no communication is possible between different thread blocks without going back to device memory (hence a global synchronization). Branching is possible by different threads in a thread block, however, if a group of 32 threads (called a Warp) diverge on the branch condition, the different routes of the branch will executed serially potentially drastically affecting performance.

While NVIDIA has the most momentum in GPU computing with CUDA, a new multi-platform standard called OpenCL [10], is intended to level the playing field: this brings C for CUDA like abstractions and compilers are available for both multi-core CPUs as well both NVIDIA and AMD GPUs [11]. At the time of writing OpenCL is still an immature product that lacks driver level performance, however, this will potentially be an important API in the future to allow for multi-platform deployment with minimal code development.

3. Dirac Operator

To date the focus has been upon implementing the most time consuming part of lattice QCD to GPUs: the linear equation solve of the Dirac operator. As noted in the previous section, the entire solver must be deployed on the GPU to prevent the onset of Amdahl's law. Specific issues relating the solver will be covered in §4; here we concentrate on the implementation of the Dirac operator.

As mentioned above, GPUs require fine grained parallelism for optimum performance. For any grid based computations, this typically equates to assigning a single thread to each point in the grid. This is equally true for the application of the Dirac operator to the spacetime lattice, where

typically a single thread is assigned to updating a single spacetime point.[2] Thus for a lattice of size $N_x N_y N_z N_T$, a grid of thread blocks with this total number of threads are created to each update their assigned lattice site. High register pressure will reduce the number of concurrent threads, so judicious use of both shared memory and registers is required to keep the occupancy high.[3] Since each site or link matrix consists of many entries, care must be taken with regards to field ordering in order obtain coalesced memory access: fields must be reordered with internal degrees of freedom (e.g., color, spin) running slowest since each adjacent thread can read at most 128 bits [3]. Some variations on these themes are possible, these will be mentioned where relevant below.

3.1 Wilson

The application of the Wilson dslash operator requires that, for every site in the lattice, we load the eight neighbouring sites (24 real numbers, forwards and backwards in four dimensions) and the link matrices connecting these sites (18 real numbers), perform the SU(3) matrix-vector multiplication and save the resultant (24 real numbers). For the even-odd preconditioned operator this equates to 1368 flops (using spin projection) and 1440 bytes of memory traffic (in single precision) per site. Given the ratio of peak single precision flops and the bandwidth to device memory, it is clear that this is a memory bandwidth bound operation. Thus to improve performance it is clear that any reductions in memory traffic will lead to higher throughput. Memory traffic reduction strategies for the Wilson-Dirac operator are thoroughly explored in [7]:

1. It is conventional in the lattice QCD community to use the DeGrand-Rossi basis (a chiral basis) for the Dirac matrices. In this basis, the spin projectors $P^{\pm\mu}$ for each of the 4 dimensions have 2 non-zeros per row. By changing to the non-relativistic basis, which diagonalizes γ_4, $P^{\pm 4}$ has only 2 non-zeros in the entire matrix; only a half spinor need be loaded when fetching the neighbouring sites in the temporal dimension, e.g.,

$$P^{+4} = \begin{pmatrix} 1 & 0 & \pm1 & 0 \\ 0 & 1 & 0 & \pm1 \\ \pm1 & 0 & 1 & 0 \\ 0 & \pm1 & 0 & 1 \end{pmatrix} \Longrightarrow P^{+4} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, P^{-4} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}.$$

2. In a similar vein, a gauge transformation can be applied to rotate all the links in one dimension to the unitary matrix (excluding a boundary term). Doing so means that the gauge field in this dimension need not be loaded (nor the connected spinor multiplied).

3. Given that $U \in SU(3)$, the link matrices can be simply parametrized using 12 [12] or 8 [13] numbers ($= N_c^2 - 1$ the number of generators of the group). Memory traffic can therefore be reduced by loading the parametrization of U and reconstructing the full matrix once in registers. Using such parametrizations increases the total number of operations to apply the Wilson operator by 384 flops (25%) and 856 flops (63%), however, as is a common

---

[2] As a gather operation, since a scatter formulation would cause a race condition between threads.

[3] A strategy that has not yet been explored is finer grained parallelization within either color, spin or complexity. Such extreme parallelization will likely be required on future architectures since the number of cores is expected to grow faster than the number registers and shared memory.

theme in GPU computing, reducing memory traffic at the expense of floating point operations increases overall throughput. Using a parametrization for the gauge field has the added benefit of reducing the memory footprint, which can be vital on the GPU where is at a premium.
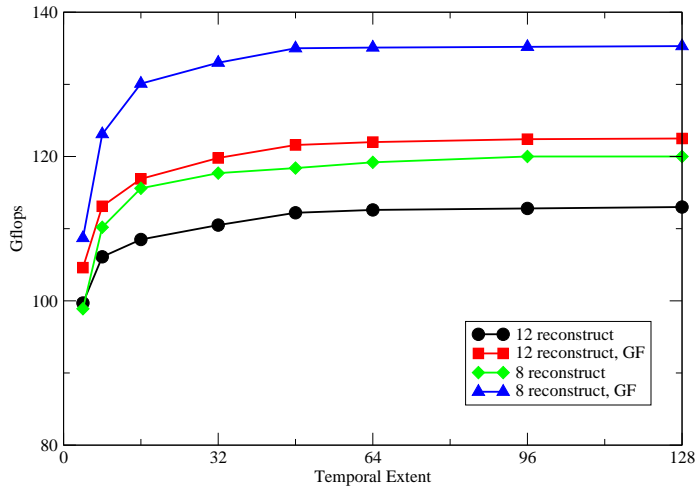

Figure 2: Performance of the padded single precision even-odd preconditioned Wilson-Dirac matrix-vector product on a GTX 280 as a function of temporal length (spatial volume 24^3; GF denotes that temporal gauge fixing has been applied) [7].

Figure 2 is a plot demonstrating the single precision performance using various combinations of the strategies highlighted above.[4] The conclusion is clear: reducing memory traffic leads to higher performance. Note that at smaller volumes the overall performance is reduced because there is less parallelism, and hence less threads to hide latency.

In double precision, because of the much lower floating point peak performance (on the GT200 generaion), the flop / bandwidth ratio is now much equally balanced. As such there is actually a net reduction in performance using the more flop intensive 8 parameter gauge field method, and the simpler 12 parameter strategy is preferred. The ratio between single and double precision actual performance is typically only a factor of 3-4 (see Table 2).

Another memory traffic reduction strategy is to decrease the precision with which the spinor and gauge fields are stored in. In particular, the use of half precision for reading and writing to device memory (while still performing the computation in single precision) halves the memory traffic, corresponding to an almost doubling in performance (see Table 2). When combined with a mixed precision solver, this can lead to a net reduction in the time to a double precision solution (see §4).

---

[4]The reported Gflops are effective Gflops, i.e., not including the cost of the $SU(3)$ matrix reconstruction, nor the savings from imposing gauge fixing.

3.2 Wilson-Clover Fermions

With a high performance Wilson dslash implemented, the extension to Wilson-Clover is straight-forward. It is worth pointing out, however, that in the non-relativistic Dirac basis the clover term is a fully dense matrix, hence the approach taken in the QUDA library [14] is to convert the spinor field back into the relativistic basis on the fly (and back again after the application of the clover term) to avoid incurring the extra bandwidth and storage. Performance is typically slightly greater ($\sim 10\%$) than the Wilson operator as the addition of the clover term increases the overall compute intensity.

3.3 Domain Wall Fermions

In the work by Chiu *et al* [6] they describe how to implement a $5d$ formulation of chiral fermions for CUDA. Here they use $4d$ even-odd preconditioning which results in the same gauge field along all sites of the 5th dimension (of length N_s). Thus, by sharing the gauge field load across all threads acting on a given $4d$ point and storing these elements in shared memory the memory traffic for the gauge field can be reduced by $O(1/N_s)$. In addition they create a grid of thread blocks consisting in total of only $N_x N_y N_z N_s$ threads, and each one these threads streams through the temporal dimension reusing the forward hop spinor from the previous site as the backward hop site for the current site. The resulting CG inverter performance is 120 Gflops sustained in single precision, and they use a mixed precision approach to achieve double precision accuracy (see §4).

3.4 Overlap Fermions

With a Wilson kernel implemented, the application of this to the Neuberger operator is a natural extension. On traditional architectures the sign function $\varepsilon(H)$ is usually best approximated using an optimal rational approximation (Zolotarev) evaluated using a multi-shift solver. However, such solvers map poorly onto GPUs because of limited memory and the bandwidth intensive nature of the additional linear algebra would significantly affect performance. The strategy taken by Wittig and Walk [15] is to evaluate $\varepsilon(H)$ on the GPU, and run the outer inverter on the host. Here $\varepsilon(H)$) is approximated using a Chebyshev polynomial using the Clenshaw recurrence relation: such an approach is well suited to the GPU because of small memory and linear algebra overhead. In single precision, they achieve 65 Gflops on a GTX 280, which represents a 20 fold speed up over their CPU implementation (left panel of Figure 3). To find the low modes required for projection, they are using the GPU simply as an accelerator for the eigenvector solver which runs on the host CPU, and as such the speedup over the CPU is limited (right panel of Figure 3).

3.5 Staggered Fermions

Obtaining high performance for staggered fermions on GPUs is more difficult than for Wilson fermions because of the decreased compute intensity, and since many improved staggered fermion variants use link matrices that are smeared such that $U \notin SU(3)$, meaning that no bandwidth reducing parametrizations are possible. Staggered fermions do have the advantage of involving less degrees of freedom, and so a larger spacetime lattice will fit on a single GPU.

A number of groups are using staggered fermions with GPUs: Cossu *et al* [16] are exploring the $N_f = 2$ staggered fermion phase transition for a variety of small masses exploiting trivial
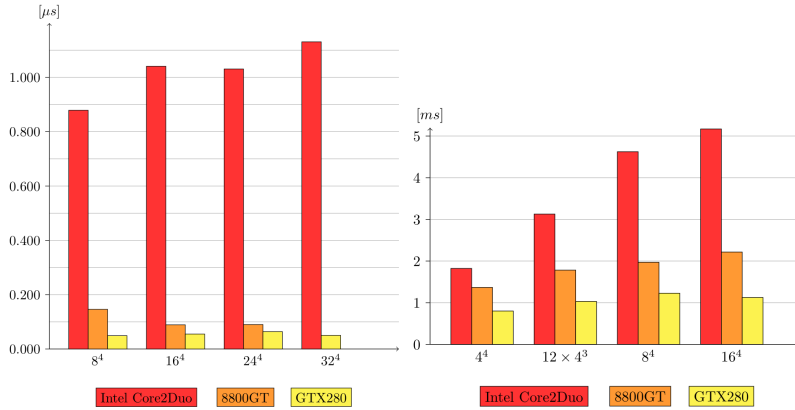
Figure 3: Left panel: runtime of the application of the Neuberger operator without low-modes. Right panel: runtime of low-mode solver. The runtime is normalized to the lattice volume and the degree of of the approximating polynomial (left) and the number of eigenvalues found (right).

parallelism to explore the parameter space. On the Tesla C1060 they achieve 60 Gflops in single precision for the complete CG inverter, this represents a 50 fold speed up over the CPU. However, when they consider HMC, only a 20 fold speedup is obtained: this is the onset of Amdahl's caused by the gauge force and momentum updates being done on the CPU and hence slowing down the entire calculation. To rectify this the entire HMC trajectory must be done on the GPU. In addition Shi [17] has ported variations of a staggered CG solver to the QUDA package [14] and observes 30, 86 and 135 Gflops for double, single and half precisions respectively on a GTX 280.

4. Mixed Precision Solvers

With the action of the Dirac operator upon a spinor field implemented, the implementation of a Krylov solver is straightforward. The additional linear algebra required can be found in the cuBLAS library, (included with the CUDA SDK) and provides a mostly complete implementation of BLAS [18]. However, since these operations are extremely bandwidth bound it is better to fuse such linear algebra operations into single kernels where possible to reduce memory traffic. Table 2 lists some typical performance numbers for CG and BiCGstab solvers at different precisions on the GTX 280, where it can be seen that actual solver performance is typically 20% slower than the equivalent "bare" kernel operation because of the additional linear algebra.

Kernel type	Precision	Kernel (Gflops)	CG (Gflops)	BiCGstab (Gflops)
Wilson	Half	202.2	170.6	152.5
Wilson	Single	134.1	110.1	105.1
Wilson	Double	35.4	33.5	29.3

Table 2: Performance comparison of the Wilson even-odd matrix-vector kernels with the associated CG and BiCGstab solvers on the GeForce GTX 280 (volume = $24^3 \times 48$, double precision global sums) [7].

Given the disparity in performance at different precisions, and that typically greater than single precision accuracy is required when solving the Dirac equation, mixed precision solvers have become a *de facto* tool when using GPUs.[5] The standard approach is defect-correction (also known as iterative refinement) [20] and allows the residual to be reduced in an inner solve using low precision, while the residual calculation and solution accumulation are done in high precision (see Algorithm 1). Such an approach is guaranteed to converge to the desired precision ε provided that the inverse of the spectal radius is bounded by the unit of least precision of the arithmetic used for the inner solve, i.e., $1/\rho(A) > \text{ulp}^{in}$, where A is the system matrix in question.

$r_0 = b - Ax_0$;
$k = 0$;
while $||r_k|| > \varepsilon$ **do**
    Solve $Ap_{k+1} = r_k$ to precision ε^{in};
    $x_{k+1} = x_k + p_{k+1}$;
    $r_{k+1} = b - Ax_{k+1}$;
    $k = k + 1$;
end

Algorithm 1: Defect-correction solver for $Ax = b$ (initial guess x_0, outer solver tolerance ε and inner solver tolerance ε^{in}).

The disadvantage of using defect-correction is that each time the residual is recalculated in high precision, the low precision Krylov solver is restarted, thus losing the orthogonal sub-space that has been built up; thereby increasing the total number of iterations to reach convergence. The use of *reliable updates* [21] in the context of a mixed precision solver was introduced in [7], as a means to circumvent the restarting penalty. Here, the residual is recalculated and the solution accumulated in high precision more frequently, however, this is done *in situ*, without an explicit restart (see Algorithm 2). Although this makes little difference when compared to defect-correction in a single-double mixed precision solver, for half-double this was found to significantly decrease the number of iterations until solution [7].

Of course the only metric that is of real concern is the time to an accurate solution. Figure 4 is a plot comparing the time to solution of using a pure double precision solver with single and half precision solvers with double precision reliable updates. The desired final residual tolerance $\varepsilon = 10^{-12}$ is far the beyond the resolution of single precision. There is a significant reduction in the time to solution when using a mixed precision approach. Also shown is the speedup versus the pure double solver: here the speedup using single precision is around a factor 3, and for half precision it is a factor of 4. The decrease in speedup as the chiral limit is approached is caused by the inability half precision to correctly resolve the full eigenvalue spectra, but in the region of physical interest, the half-double approach is always the fastest.

---

[5]Mixed precision solvers are increasingly used on more traditional architectures such as BlueGene [19].

$r_0 = b - Ax_0$;
$\hat{r}_0 = r$;
$\hat{x}_0 = 0$;
$k = 0$;
while $||\hat{r}_k|| > \varepsilon$ **do**
  Low precision solver iteration: $\hat{r}_k \to \hat{r}_{k+1}, \hat{x}_k \to \hat{x}_{k+1}$;
  if $||\hat{r}_{k+1}|| < \delta M(\hat{r})$ **then**
    $x_{l+1} = x_l + \hat{x}_{k+1}$;
    $r_{l+1} = b - Ax_{l+1}$;
    $\hat{x}_{k+1} = 0$;
    $\hat{r}_{k+1} = r$;
    $l = l + 1$;
  end
  $k = k + 1$;
end

Algorithm 2: Mixed precision reliable update solver for $Ax = b$ (initial guess x_0, outer solver tolerance ε, $M(r)$ is the maximum of the norm of the residuals since the last residual update, (ˆ) denotes low precision) [7].
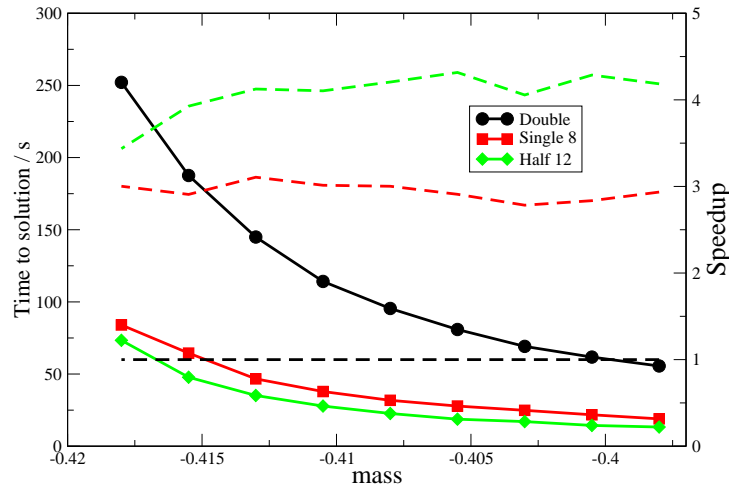

Figure 4: Time to solution (solid lines) for double precision and reliable update solvers and speedup versus double precision (dashed lines) (BiCGstab, $\delta = 0.1$, $\varepsilon = 10^{-12}$, volume $= 24^3 \times 64$) [7].

5. Multi-GPU

Although reasonably large lattice volumes can be accommodated for on current GPUs, the desire to go to larger volumes and perform the gauge generation on GPUs necessitates that the

calculation is spread over multiple GPUs. This is the next step to be taken for QCD on GPUs, and at the time of writing none of the QCD research groups have suitable multi-GPU production code.

There are two alternatives for multi-GPU operation:

- Place many GPUs with a node, and parallelize only within the node.

- Connect many nodes together, each with one or more GPUs, by a fast inter-connect.

Currently, it is not possible to copy data directly from GPU to GPU, and one is forced to go through the CPU host memory. However, it is possible to perform asynchronous transfers, i.e., overlap the communication between the GPU and CPU while the GPU is executing a kernel.[6] While going through the CPU memory does not affect the aggregate bandwidth it does increase latency, increasing the minimum local volume feasible on each GPU. Likewise, when sending and receiving messages through Infiniband, the data must be read through a CPU buffer. The use of threads over processes is preferred for multiple GPUs within a node to avoid the extra message sending overhead.
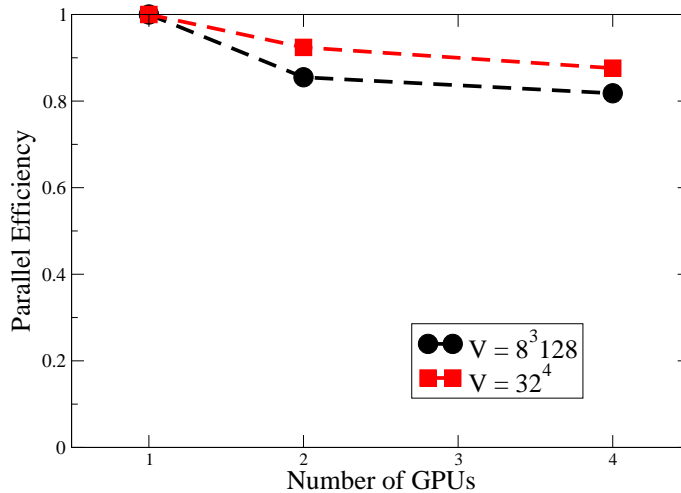

Figure 5: Weak scaling parallel efficiency of the Wilson Clover operator as function of number of GPUs (Tesla S1070, one process per GPU, temporal parallelization only) [23].

Initial attempts at limited multi-GPU operation within a node have been extremely promising. Even without the overlap of communication and computation, and the overhead of using MPI instead of threads to control each GPU, up to 90% parallel efficiency with 4 GPUs has been obtained (Figure 5). Clearly however, moving beyond the confines of a single node will be challenging, and alternative algorithms (e.g., Schwartz approaches [22]) may be required for parallelizing over many GPUs.

---

[6]In addition, the Fermi architecture introduces bi-directional transfers, so that the GPU can be sending and receiving while executing a compute kernel.

6. Price performance

The current generation of GPUs attain around 100 Gflops of sustained performance in single precision for Wilson like discretizations. This compares to a single rack of BG/P which sustains around 3 Tflops [19]. When we compare the upfront cost (Tesla C1060 $1200 and a host computer, BG/P a lot) and running costs (typical GT200 power consumption is 225W and probably the same again for the host, whereas a single BG/P rack requires 30kW [26]) this makes GPUs a very attrtactive proposition QCD calculations.

When designing GPU clusters for lattice QCD calculations, the nature of the target calculations will have an effect on the required hardware, and hence the price performance. For trivially parallelizable applications (e.g., those involving many solutions to the Dirac equation, with a constant gauge field), there is no requirement for a fast inter-connect between the nodes. Even for those calculations that require the use of multiple GPUs because of memory contraints, the most cost efficient approach is to use multiple GPUs within a node. As soon as one considers a multi-node cluster the cost will increaase significantly because of the added cost of Infiniband, and the lower Gflops per node.

There are an increasing number of large scale GPU clusters being used for lattice QCD calculations, e.g., Wuppertal, Thomas Jefferson Laboratory (Jlab) and TWQCD. These clusters have been built with the goal of multi-GPU within each node only (and currently using only a single single-GPU per Dirac inversion). Both the Jlab and TWQCD GPU clusters are built using the GT200 generation GPUs, consisting of 128 and 202 GPUs respectively (at the time of writing). For domain wall fermions and Wilson clover fermions, which these respective groups focus, both report a price performance around $0.02 per Mflop [6, 24].

We end this section by noting that in the era of heterogeneous architectures and mixed precision algorithms, comparing price performance is no longer as simple as $ per Mflop. Improved solver algorithms such as inexact deflation [25] and adaptive multigrid [27, 28], and their efficient mapping onto evolving target architectures further couple the algorithmic performance to the underlying hardware. The only fair comparisons will be those that take account for these variables: seconds per HMC trajectory and such like.

7. Conclusions

With the heterogeneous paradigm now in full swing, the lattice QCD community is well placed to embrace the increased computation available through GPUs. To take full advantage of the performance offered by GPUs, algorithms must be redesigned and must be ported completely else Amdahl's law will severely restrict any gains. GPUs additionally promise to vastly reduce the cost of entry into doing lattice QCD.

The next generation GPU architecture Fermi from NVIDIA addresses nearly all concerns that lattice field theorists may have had when deploying their calculations on GPU platforms: fast double precision, ECC protected memory, C++ support (the only remaining one being peer to peer communication). It should be noted that even with fast double precision, mixed precision methods will still be important because there always be a factor of two difference in memory traffic between single and double precisions.

12

The remaining hurdle before GPUs will hit prime time is that of multi-GPU deployment. While limited parallelization is achievable with current technology, for large scale deployment (100s GPUs) improved inter-GPU communication or algorithmic developments will be required.

8. Acknowledgements

References

[1] AMD Corporation, `http://amd.com`

[2] NVIDIA Corporation, `http://nvidia.com`

[3] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi and K. K. Szabo, "Lattice QCD as a video game," Comput. Phys. Commun. **177**, 631 (2007) [arXiv:hep-lat/0611022].

[4] K. Barros, R. Babich, R. Brower, M. A. Clark and C. Rebbi, "Blasting through lattice calculations using CUDA," PoS **LATTICE2008**, 045 (2008) [arXiv:0810.5365 [hep-lat]].

[5] K. I. Ishikawa, "Recent algorithm and machine developments for lattice QCD," arXiv:0811.1661 [hep-lat].

[6] Ting-Wai Chiu, Tian-Shin Guu, Tung-Han Hsieh, Chao-Hsi Huang, Yu-Ying Lee, Yao-Yuan Mao, Kenji Ogawa and Po-Kai Tseng "TWQCD's dynamical DWF project" PoS **LATTICE 2009**, 034 (2009)

[7] M. A. Clark, R. Babich, K. Barros, R. C. Brower and C. Rebbi, "Solving Lattice QCD systems of equations using mixed precision solvers on GPUs," arXiv:0911.3191 [hep-lat].

[8] C2070 specifications estimated based on announced specifications at `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_Final_lowres.pdf`

[9] `http://www.nvidia.com/object/fermi_architecture.html`

[10] `http://www.khronos.org/opencl`

[11] `http://ati.amd.com/technology/streamcomputing/opencl.html`

[12] P. De Forcrand, D. Lellouch and C. Roiesnel, "Optimizing a lattice QCD simulation program," *J. Comput. Phys.* **59**, 324 (1985).

[13] B. Bunk and R. Sommer, "An eight parameter representation of SU(3) matrices and its application for simulating lattice QCD," *Comput. Phys. Commun.* **40**, 229 (1986).

[14] `http://lattice.bu.edu.quda`

[15] H. Wittig and B. Walk, private communication.

[16] G. Cossu, C. Bonati, M. D'Elia and A. D. Giacomo, private communication.

[17] G. Shi, private communication.

[18] `http://www.netlib.org/blas`

[19] P. A. Boyle "The BAGEL assembler generation library" *Comput. Phys. Commun.* **180**, 2739 (2009)

[20] R. S. Martin, G. Peters and J. H. Wilkinson, "Handbook series linear algebra: Iterative refinement of the solution of a positive definite system of equations," *Numerische Mathematik* **8**, 203âĂŞ216 (1966).

[21] G. L. G. Sleijpen, and H. A. van der Vorst, "Reliable updated residuals in hybrid Bi-CG methods," *Computing* **56**, 141-164 (1996).

[22] M. Luscher, "Solution of the Dirac equation in lattice QCD using a domain decomposition method," Comput. Phys. Commun. **156** (2004) 209 [arXiv:hep-lat/0310048].

[23] J. Chen, `http://www.jlab.org/~chen/lqcdgpu-09/lqcdgpu-09.ppt`

[24] W. Watson, private communication.

[25] M. Luscher, "Local coherence and deflation of the low quark modes in lattice QCD," JHEP **0707** (2007) 081 [arXiv:0706.2298 [hep-lat]].

[26] S. Alam *et al Conference on High Performance Networking and Computing archive Proceedings of the 2008 ACM/IEEE conference on Supercomputing* **00** 23 (2008)

[27] J. Brannick, R. C. Brower, M. A. Clark, J. C. Osborn and C. Rebbi, "Adaptive multigrid algorithm for lattice QCD," Phys. Rev. Lett. **100**, 041601 (2008) [arXiv:0707.4018 [hep-lat]].

[28] M. A. Clark, J. Brannick, R. C. Brower, S. F. McCormick, T. A. Manteuffel, J. C. Osborn and C. Rebbi, "The removal of critical slowing down," PoS **LATTICE2008** 035 (2008) [arXiv:0811.4331 [hep-lat]].