Mixed-precision GPU Krylov solver for lattice QCD

R. Babich[1], K. Barros[1], R. C. Brower[1], J. Chen[2], M. A. Clark[3], C. Rebbi[1]
[1] Boston University, [2] Thomas Jefferson National Accelerator Facility, [3] Harvard University

ABSTRACT

Using the CUDA platform we have implemented a mixed precision Krylov solver for the Wilson-Dirac matrix for lattice QCD. The matrix-vector product which accounts for the vast majority of the operations runs in excess of 130 Gflops in single precision on the GTX 280. We have developed a new approach for mixed-precision Krylov solvers that achieves in excess of 100 Gflops and achieves full double precision accuracy. We also explore the use of half precision in this context to further decrease time to solution. Finally we report on initial findings for extending the problem to multi-GPUs, where we find reasonable performance scaling.

1 Introduction

- Quantum chromo-dynamics (QCD) is the theory of the strong force: the force that binds together the protons and neutrons in the nucleus of atoms. The theory is non-perturbative which renders the theory inaccessible to analytic approaches, and a numerical approach must be used: lattice QCD.
- Lattice QCD involves discretizing the 4 dimensional spacetime onto a finite lattice with periodic boundary conditions. As the lattice spacing is reduced to zero and the size of the box is taken infinite, lattice QCD should exactly reproduce the continuum theory.
- This work concerns the development of a GPU implementation of Krylov solvers for the Dirac matrix in lattice QCD, e.g., solving

$$M(U)x = b,$$

where M is the discretized Dirac operator which is a sparse matrix whose elements are a function of a background field U, and b and x are the source and solution vectors respectively. This problem accounts for the majority of operations in lattice QCD.
- The dominant cost in such solvers is the *sparse matrix-vector product*. Thus all our effort is focused upon achieving high performance for this specific kernel. This is a very bandwidth intensive kernel, and so careful consideration of the problem is required.
- In general we require solutions to much greater precision than single precision can accommodate. Since there is a large disparity between single and double precision peak performance on current GPUs, special consideration must be given as to how to achieve the least time to accurate solution, i.e., *mixed precision solvers*.
- Large lattices are required to control discretization errors, e.g., $V = 128^4$. Each site on the spacetime lattice has 24 numbers (3 colors $\times$ 4 spins $\times$ 2 for complexity) $\implies O(10^{10})$ degrees of freedom in total. Thus we require the problem be spread onto multiple GPUs.

2 Wilson-Dirac Matrix

- The Wilson-Dirac matrix represents the central-difference approximation of a 1^{st} derivative operator acting in $4d$ spacetime. It has a regular sparsity structure with a diagonal constant (the mass parameter) and 8 off-diagonal 12×12 complex valued blocks.

$$\begin{aligned} M_{x,x'} &= -\frac{1}{2}\sum_{\mu=1}^{4}\left((I_4-\gamma_\mu)\otimes U^{\mu}_{x}\,\delta_{x+\hat{\mu},x'} + (I_4+\gamma_\mu)\otimes U^{\mu\dagger}_{x-\hat{\mu}}\,\delta_{x-\hat{\mu},x'}\right) + M\delta_{x,x'} \\ &= -\frac{1}{2}\sum_{\mu=1}^{4}\left(P^{-\mu}\otimes U^{\mu}_{x}\,\delta_{x+\hat{\mu},x'} + P^{+\mu}\otimes U^{\mu\dagger}_{x-\hat{\mu}}\,\delta_{x-\hat{\mu},x'}\right) + M\delta_{x,x'} \\ &= -\frac{1}{2}D_{x,x'} + M\delta_{x,x'}. \end{aligned}$$

- The background field U is a field of $SU(3)$ matrices that live on the links between the sites in the spacetime lattice. These represent the color (chromo) symmetry of the theory, and give rise to a color charge analogous to the electric charge of electro-magnetism.
- γ_μ are the 4×4 Dirac spin matrices, these are fixed and there is one per dimension.
- M is not Hermitian, but there is a symmetry relating super- and sub-diagonal elements.
- Solving systems involving this matrix usually requires the use of Krylov solvers, such as Conjugate Gradients (CG) on the normal equations, or BiCGstab directly.

3 Matrix-Vector Implementation

- One implementation option is to use a sparse matrix-vector product library. However, such libraries are ignorant of the structure of the matrix, so any physical symmetries to simplify the problem cannot be used, e.g.,
  - Diagonal symmetry between super- and sub- diagonal elements.
  - Regular sparsity pattern means indexing is not required.

  Typical performance numbers from libraries are up to 16 Gflops depending on the sparsity pattern of the matrix and the packing method used.
- Thus we implement the matrix-vector product as a nearest neighbor gather operation. Exact load balancing is obtained by assigning one thread per spacetime point, where to update each lattice site, each thread has to
  - load each neighboring lattice site (vector of 24 numbers)
  - apply the relevant P matrix to this vector (result is a vector of 12 numbers)
  - load each $SU(3)$ matrix connecting that site (18 numbers)
  - SU(3) matrix-vector operation and accumulate over direction (vector of 24 numbers)
  - save the result to device memory
- For optimal performance there should be at least 192 threads active. The per-thread memory footprint and limited shared memory resources force us to store most data in registers. This precludes explicit index looping, but tedious code writing is avoided through the use of a scripted code generator.
- For each lattice site, this requires 1440 bytes in single precision, but involves only 1320 flops. Thus this routine is severely bandwidth limited, and we must reduce the bandwidth at all costs to improve performance.
- For memory coalescing the threads must read from consecutive memory addresses. We have found optimum performance ordering our data into fields of *float4* (or *double2* for double precision).

4 Reducing Memory Bandwidth Requirements

- An $SU(3)$ matrix is a 3×3 complex matrix with determinant =1 and whose Hermitian conjugate is its inverse. This imposes 10 constraints on the matrix, and thus a minimum of 8 real numbers are required to parametrize the matrix. Our strategy therefore is to store these parameters only and *reconstruct the full matrix in the registers on the fly*.
  - 12-number parametrization which requires 29% extra flops

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix} \Longrightarrow \begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{pmatrix}$$

  - 8-number parametrization which increases the flop count by 64%.

$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix} \Longrightarrow \begin{pmatrix} \arg(a_1) & \arg(c_1) & a_2 & a_3 & b_1 \end{pmatrix}$$

- We can also impose physically motivated *similarity transformations* to increase matrix sparsity and thus decrease the number of elements that must loaded.
  - Change the basis of the γ matrices to diagonalize them them in the T dimension.

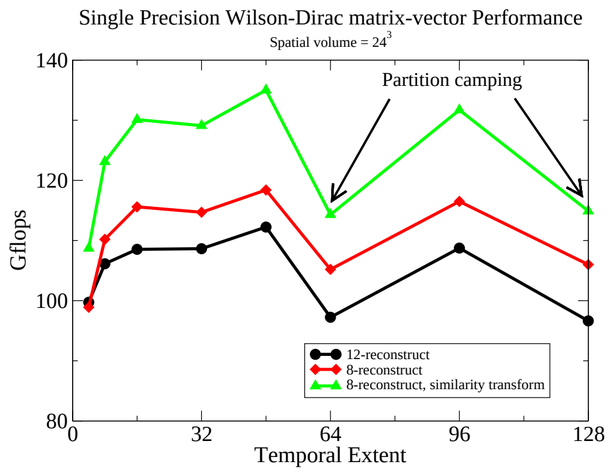
$$P^{\pm 4} = \begin{pmatrix} 1 & 0 & \pm 1 & 0 \\ 0 & 1 & 0 & \pm 1 \\ \pm 1 & 0 & 1 & 0 \\ 0 & \pm 1 & 0 & 1 \end{pmatrix} \Longrightarrow P^{+4} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, P^{-4} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0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

  - Perform similarity transformations on the U field such that the link matrices are the unit matrix in the T dimension.

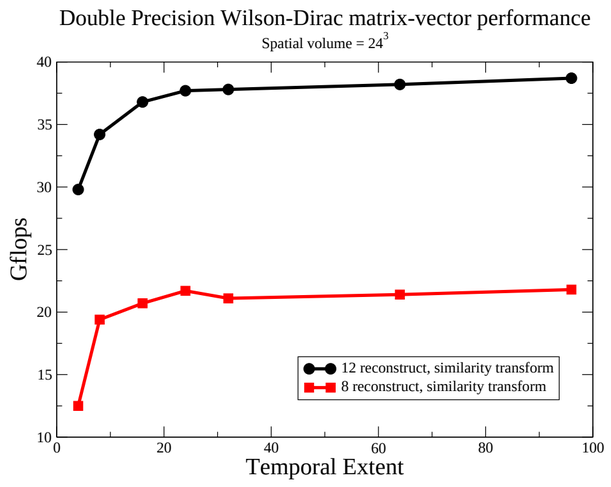
$$\begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{pmatrix} \Longrightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Altogether, transformations reduce the required memory throughput by 33%.

5 Matrix-Vector Performance



- In single precision the matrix-vector kernel achieves > 130 Gflops on a GTX 280 (here we of course only report useful Gflops, i.e., not including the cost doing the $SU(3)$ reconstruction).
- As the size of the problem is increased the performance improves, due to the increased numbers of threads available to tackle the problem. Notable, however, is the large reduction in performance at $T = 64$ and $T = 128$: this is due to *partition camping*, this is where memory conflicts occur when threads collide when reading from the device memory. This problem is analogous to shared memory conflicts, and can be overcome by padding the fields so that the memory read patterns are not divisible by the partition width (on current cards thisis 256 bytes) [1].
- GPU code performance is more than an order of magnitude greater than typical SSE optimized implementations (Wilson Matrix-Vector $O(10)$ Gflops on a Nehalem Xeon processor). In addition the scaling with increasing volume is relatively constant, compared to CPU implementations which drastically fall in performance as the local volume is increased out of the cache.



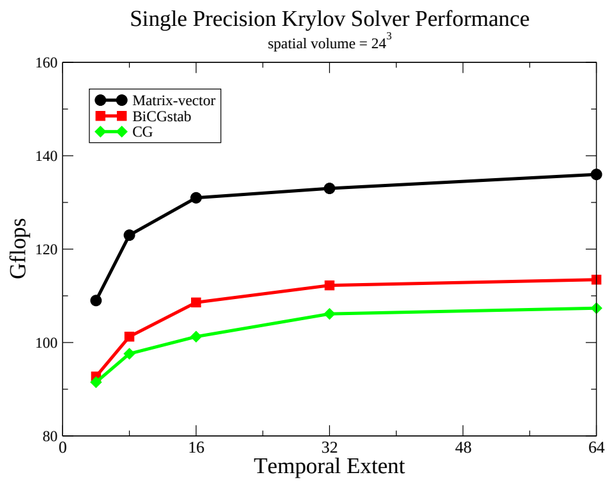
- For double precision, since the peak performance of the GPU is $12\times$ less than that of single, the kernels are no longer bandwidth bound. The extra flops overhead of the 8-parameter kernel results in significantly lower performance than the 12-parameter kernel. The double precision 12-parameter kernel achieves > 35 Gflops.
- Motivated by the fact that we are still bandwidth bound in single precision we have also implemented a pseudo-half precision kernel. Prior to version 2.3, CUDA did not support half precision in the runtime API. In our implementation we utilize the texture read mode called *cudaReadModeNormalizedFloat*, which loads in a 16 bit integer and scales it to a 32 bit float $\in [-1, 1]$. This is ideal for elements of the U matrices since they are exactly in this range, but for the vector field elements we also store one exponent per lattice site (per 24 numbers). The half precision kernel achieves in excess of 200 Gflops, at of course the price of reduced precision.

6 Krylov Solver Implementation

- Krylov solvers work by minimizing the residual vector r in an orthogonal sub-space $\mathcal{K} \in \{r_0, Ar_0, A^2r_0, \ldots\}$. Once the residual is within the prescribed tolerance ε the process is terminated and the solution is given by x.

$r_0 = b - Ax_0$;
$k = 0$;
while $||r_{k+1}|| > \varepsilon$ **do**
  $\beta_k = ||r_k||^2/||r_{k-1}||^2$;
  $p_{k+1} = r_k - \beta_k p_k$;
  $\alpha = ||r_k||^2/(p_{k+1}, Ap_{k+1})$;
  $r_{k+1} = r_k - \alpha A p_{k+1}$;
  $x_{k+1} = x_k + \alpha A p_{k+1}$;
  $k++$;
end

- We have implemented both CG (given above) and BiCGstab. The entire inverter must be implemented on the GPU, since transferring the vector across the PCIe bus would drastically reduce the overall performance.
- The required BLAS1 like operations are combined into single kernels to reduce memory traffic, e.g., the combing the AXPY ($y = ax + y$) and NORM2 ($||y||^2$) operations into a single AXPY_NORM2 kernel.
- For global sums we follow the approach of performing parallel reductions on the GPU until it becomes more efficient to complete the reduction on the CPU. Where possible reductions are combined so that only a single PCIe transfer is issued to reduce latency.

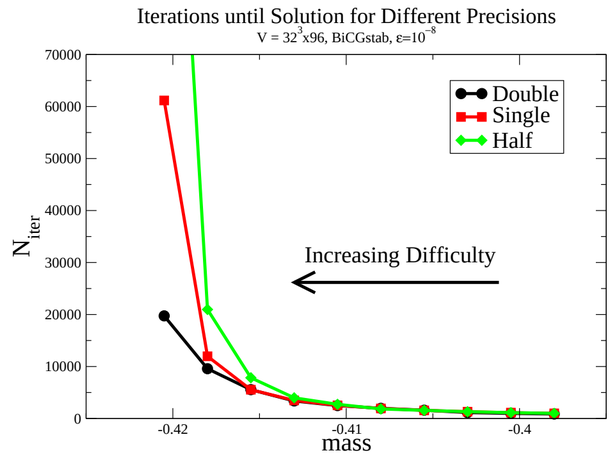


- The solvers run at well over 80% of the speed of just the matrix-vector kernel. This reduction in performance is unavoidable given the bandwidth limited BLAS1 kernels.

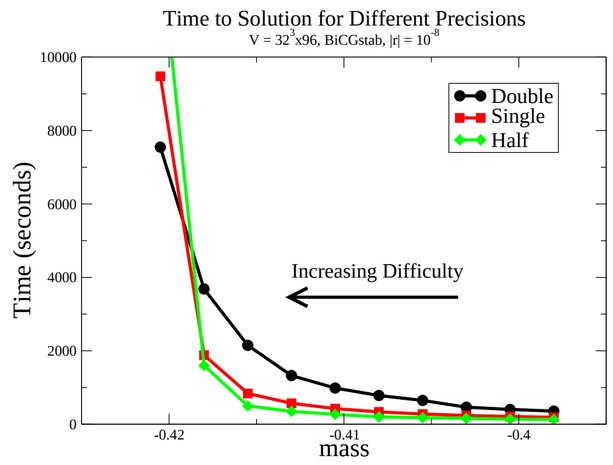
7 Mixed-Precision Krylov Solver

- The usual mixed-precision strategy used with linear equation solvers on GPUs is *defect-correction*[2]. While this strategy is sufficient for direct approaches, for Krylov subspace methods it is sub-optimal since every time the loop restarts, the Krylov sub-space is lost.
- As an alternative we apply the previously introduced *reliable updates*[3] in the context of mixed precision. Here the residual is periodically corrected to double precision, and high precision group-wise updates are applied to the solution *without explicitly restarting the Krylov process*. The frequency with which high precision updates take place is determined by the rate of decrease of the residual to ensure that the constructed (single or half precision) Krylov space does not drift from the full double precision space.

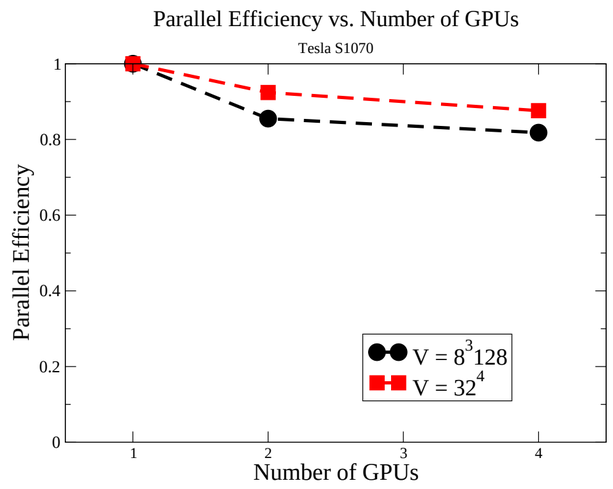
if *'compute_res'* = true **then**
  $r_0 = b' - Ax$;
  if *'update_app'* = true **then**
    $x' = x' + x$;
    $x = 0$;
    $b' = r_0$;
  end
end



- The plot above shows the number of iterations for the different precisions using the reliable update scheme (the target residual is smaller than the resolution of single precision). As the condition number gets very large (large negative mass) the total number of iterations increases, revealing there is a penalty to using the reliable update scheme.
- However, in terms of time until solution both single and half precision approaches are superior to using double precision alone.



8 Multi-GPU



- The weak scaling up to 4 GPUs is shown on the plot above. Here there is no overlap between communication and computation, so because of the surface to volume ratio, as the local problem size increases so does the parallel efficiency.
- Given the good scaling already, overlapping communication and computation should give near perfect parallel efficiency. Scaling up to 16 GPUs seems reasonable.
- These results used separate MPI processes to control each GPU. For multiple GPUs within a shared memory system threads would be preferable. Thus mixed-mode communication (threads within a box, MPI between boxes) will be required for optimum performance.
- The use of Schwarz methods to reduce communication between GPUs may be required to scale to many GPUs.

9 Future Work

- Implementation of other important kernels from lattice QCD to prevent the onset of Amdahl's law limiting the performance gain.
- Multigrid solver: this will drastically reduce the time to solution by using solutions on coarser grids to accelerate the solution finding process for the original (fine) problem.
- The actual CUDA code was produced by using a higher level code generator written in *Python* [4]. We intend to evolve this code generator into a more general toolkit for easy writing of CUDA kernels for field based operations. This will greatly accelerate the porting over of lattice QCD kernels to the CUDA environment, and would potentially be useful for other subject areas.

Acknowledgments

This work was supported in part by US DOE grants DE-FG02-91ER40676 and DE-FC02-06ER41440 and NSF grants 0221680, PHY-0427646, and OCI-0749300.

References

[1] Ruetsch and Micikevicius, "Optimizing Matrix Transpose in CUDA" (2009)
[2] Göddeke *et al*, Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique
[3] Sleijpen, *Computing* **56** 141-164 (1996)
[4] "Python Programming Language", http://www.python.org/