Overlap fermions on GPUs

Nigel Cundy, Weonjong Lee
Seoul National University

Abstract:
We report on our efforts to implement overlap fermions on NVIDIA GPUs using CUDA, commenting on the algorithms used, implemetation details, and the performance of our code.

- The overlap Dirac operator [1] is the only known practical lattice Dirac operator with exact chiral symmetry.
- It has various advantages over other discretisations: exact chiral symmetry, no additive mass renormalisation, solid definition of the topology, automatic O(a) improvement, good topological properties ....
- It also has two major disadvantages: computational cost and algorithmic complexity.
- A GPU cluster provides a powerful way to have a low (finacial) cost, energy efficient supercomputer.
- GPUs are ideally suited for lattice QCD, constrained only by memory requirements on larger volumes.
- We here report on our attempt to write an overlap production code for the GPU architecture, based on the CPS library.
- Our first physics goal is to test this code with a computation of the quark condensate (see talk by Hwancheol Jeong).
- Our code will later be used in computations of B_K and ϵ_K.
- See [2] for another project implementing overlap fermions on GPUs.

- The overlap Dirac operator at mass parameter μ is defined as

$$D[\mu] = \frac{1}{2}(1+\mu+(1-\mu)\gamma_5 \text{sign}(K)). \tag{1}$$

- We use the Wilson Kernel for K (with $m_W = 1.5$).

$$\begin{aligned} K =& \gamma_5(D_W - m_w) \\ (D_W\psi)(x) =& 4 - \frac{1}{2}\sum_\mu \left[(1-\gamma_\mu)U_\mu(x)\psi(x+a\hat{\mu}) + (1+\gamma_\mu)U^\dagger_\mu(x-a\hat{\mu})\psi(x-a\hat{\mu})\right]. \end{aligned} \tag{2}$$

- For our tests, we apply 3 steps of over-improved stout smearing [3] with parameters $\rho = .1$, $\epsilon = -.25$.
- We calculate the matrix sign function using a Chebyshev Polynomial approximation and deflation of the smallest eigenvector/eigenvalues pairs, ϕ_i, λ_i, of K

$$\text{sign}(K) = \sum_m \alpha_m T_m(K) + \sum_i \phi_i \phi_i^\dagger \left[\text{sign}(\lambda_i) - \sum_m \alpha_m T_m(\lambda_i)\right]. \tag{3}$$

- T_m are the Chebyshev polynomials of the first kind, and α_m are the appropriate polynomial coefficients.
- We chose to use the Chebyshev Polynomial because it is efficient and requires less memory than other approaches.

- We pre-compute a number of eigenvectors ϕ of the Kernel operator K (the number calculated is limited by the system memory; the number used depends on the desired precision of the matrix sign function approximation) using a polynomial preconditioned Implicitly restarted Lanczos routine. These eigenvectors are stored in CPU memory.
- We compute the polynomial term on the GPU while simultaneously using the CPU to perform the deflation.
- Our tests are performed on a Desktop Computer, with a quad core Intel Xeon (2.5 GHz), and two NVIDIA GK110 (GeForce GTX Titan) GPUs.

GPU details:

Memory (MiB)	6144
Memory Bandwidth (Global Memory) (GB/s)	288
Single Precision Processing Power (GFlops)	4500
Double Precision Processing Power (GFlops)	1500

- The performance of our code is primarily limited by memory Bandwidth.
- We test our code on two sets of lattices, at two different (partially quenched/mixed action) masses

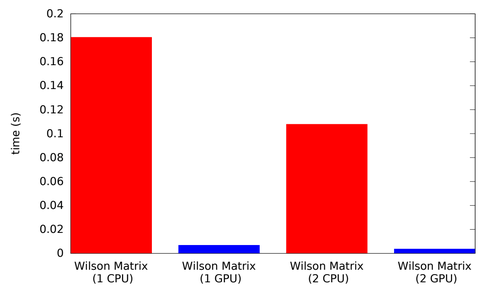
Lattice Size	Action	Sea Mass	Valance Mass	a^{-1} (GeV)	$m_\pi a$
$8^3 \times 32$	Overlap, Lüscher-Weisz	0.01	0.05	1.7	0.662(19)
$8^3 \times 32$	Overlap, Lüscher-Weisz	0.01	0.01	1.7	0.366(72)
$20^3 \times 64$	Asqtad (MILC [4]), Lüscher-Weisz	$0.01 + 0.05$	0.01	$\sim$1.6	0.276(2)
$20^3 \times 64$	Asqtad (MILC [4]), Lüscher-Weisz	$0.01 + 0.05$	0.0033	$\sim$1.6	0.173(2)

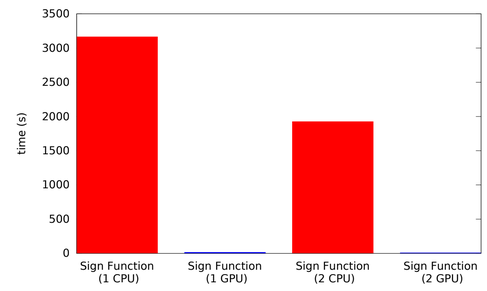
- Our code was written in the CUDA language, built on a C++ code which extends the CPS library.
- CUDA is an extension to C++ that incorporates kernels, to be run in a threaded environment on the GPU, as well as routines to transfer memory from the GPU to the CPU.
- The GPU serves as a very highly threaded co-processor for the CPU.
- The memory on the GPU is stored in various locations. Shared memory and registers have the fastest access, then the cached texture (permament) and constant memory, followed by the global memory, which holds most of the data.
- There are two primary memory bottlenecks: transfer from the GPU to the CPU (very slow) and transfer from global memory to the registers (very important).
- The GPU/CPU bottleneck is not so important for our code, since most calculations are performed entirely on the GPU, and we can overlap communication and computation. The exception to this is the deflation of small eigenvectors, where the GPU/CPU bandwidth was significant when we tried to use the GPU for this calculation. We resolved this problem by using the CPU for deflation, while the GPU computed the polynomial part of the matrix sign function.
- One important optimization of the code was to merge numerous CUDA Kernels together. This saves on memory bandwidth between global memory and the registers, and reduces the kernel launch overhead. The most important gain was by merging together the Wilson matrix and polynomial algebra for the matrix sign function into a single GPU Kernel.
- Because of our unique needs, it was difficult to integrate our code into the QUDA library [5]. We therefore wrote our own Wilson Matrix Kernel, linear algebra routines, and GPU/CPU transfer routines, though based on the implementation in QUDA. Unlike QUDA, we do not use odd/even preconditioning.
- We achieve a similar performance for the Wilson Matrix in our own code and the QUDA implementation. The table below gives a comparison between QUDA and our code (OC) for various routines for 1 and 2 processors.
- The improvement in the time for the sign function between our code and the QUDA code are due to the optimisations we were able to perform by merging the Wilson operator and the rest of the sign function code.

Lattice	$8^3 32$ QUDA 1 GPU	$8^3 32$ QUDA 2 GPU	$8^3 32$ OC 1 GPU	$8^3 32$ OC 2 GPU
Wilson Matrix (10^{-3}s)	0.236	0.195	0.248	0.172
Sign Function (s)	0.2282	0.1716	0.1448	0.1396
Lattice	$20^3 64$ QUDA 1 GPU	$20^3 64$ QUDA 2 GPU	$20^3 64$ OC 1 GPU	$20^3 64$ OC 2 GPU
Wilson Matrix (10^{-3}s)	7.092	3.842	6.759	3.625
Sign Function (s)	16.58	9.17	15.31	8.71

The following table gives a comparison between CPU and GPU performance for various key routines. The peak performance is the time compared to the expected time for processing and optimal memory bandwidth. It excludes kernel start up time, inter-processor Communication, cost because we can't use all the threads due to lack of registers, etc.

Routine (1 GPU)	CPU $8^3 \times 32$	GPU $8^3 \times 32$	peak	CPU $20^3 \times 64$	GPU $20^3 \times 64$	peak
Wilson Matrix ($10^{-3}s$)	5.59	0.248	43.5%	180.4	6.759	49.9%
γ_5 ($10^{-3}s$)	0.555	0.105	19.4%	18.92	0.885	71.8%
Chebyshev sign function(s)	8.63	0.122	42.6%	3163	14.95	41.7%
$z = ax + by$ ($10^{-3}s$)	1.58	0.202	15.5%	58.2	1.50	65.0%
Routine (2 GPU)	CPU $8^3 \times 32$	GPU $8^3 \times 32$	peak	CPU $20^3 \times 64$	GPU $20^3 \times 64$	peak
Wilson Matrix ($10^{-3}s$)	3.43	0.172	32.4%	107.8	3.625	46.5%
γ_5 ($10^{-3}s$)	0.333	0.047	16.4%	1.21	0.454	70.0%
Chebyshev sign function (s)	4.67	0.090	30.2%	1925	7.869	40.2%
$z = ax + by$ ($10^{-3}s$)	0.441	0.288	6.2%	37.3	0.897	54.4%

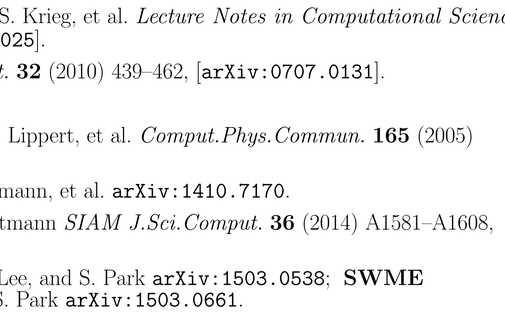




The times required for the Wilson operator (left) and the Chebyshev approximation to the matrix sign function (excluding deflation) (right) on the $20^3 \times 64$ lattice, for the CPU and GPU versions of the code.

- A optimal inversion routine for the overlap operator is the Shifted Unitary Minimal Residual algorithm (SUMR) [6, 7]. This can be combined with a computation of the eigenvectors and deflation [8] to form the eigSUMR algorithm [9].
- During the inversion, the accuracy of the matrix sign function can be relaxed as the inversion proceeds [7]. Furthermore, by incorporating the low accuracy eigSUMR inversion as a preconditioner for a flexible GMRES inverter [10], the bulk of the work can be done calling a low accuracy matrix sign function in single precision. Incorporating deflation, this gains roughly a factor of 20 performance gain over a naive CG inversion. This is the GMRES(eigSUMR) algorithm.
- However, the computation of the overlap eigenvalues requires a high accuracy matrix sign function, and cannot be efficiently relaxed [9].
- An alternative is to use a Wilson operator to precondition the overlap inversion [11]. One uses a low accuracy Wilson operator inverter as a preconditioner for a low accuracy overlap operator in a flexible GMRES algorithm, which in turn is used as a preconditioner in a second GMRES algorithm. This is the GMRES(αMGOv) algorithm.
- It is important to tune the κ value of the Wilson operator to get best performance. We do this numerically by finding the κ that minimizes the residual for a fixed number of GMRES(αMGOv) iterations.
- We use an adaptive multi-grid algorithm [12] (αMG) with ten inexact deflation vectors for the Wilson inverter. We have not yet fully tuned and optimized this inversion algorithm. The setup time for the inexact deflation is negligible for our application.
- Our αMG algorithm is not yet fully tuned or optimised. We use 10 inexact deflation vectors.
- We see more than a factor of 36 gain for αMG over a straight-forward CG inversion on our largest volume.
- We find that the GMRES(αMGOv) algorithm is superior to GMRES(eigSUMR) by about a factor of 5 on our largest volume and smallest quark mass. It has much better scaling with lattice volume than the eigSUMR routine.
- We use a relative inversion accuracy of 10^{-6} for Wilson fermions and 10^{-13} for overlap fermions.
- The Wilson inverters are in single precision; the overlap inverters use mixed precision.

Inversion Routine	CG Wilson	αMG	GMRES(eigSUMR)	GMRES($\alpha MGOv$)
$8^3 \times 32$ 1 processor $\mu = 0.05$ (s)	0.123	0.108	9.34	19.51
$8^3 \times 32$ 1 processor $\mu = 0.01$ (s)	0.299	0.232	10.67	45.27
$8^3 \times 32$ 2 processors $\mu = 0.05$ (s)	0.114	0.091	7.09	12.93
$8^3 \times 32$ 2 processors $\mu = 0.01$ (s)	0.220	0.161	9.29	29.14
$20^3 \times 64$ 2 processors $\mu = 0.01$ (s)	75.3	1.97	4030	1086
$20^3 \times 64$ 2 processors $\mu = 0.0033$ (s)	65.76	1.83	12396	2571



The times required for the Wilson (left) and overlap (right) standard and multigrid inverters running on 2GPUs on the $20^3 \times 64$ lattice.

- We have implemented a code for overlap fermions on GPUs using the CUDA programming langauge.
- Our code runs a factor of 200 faster than the CPU code for the matrix sign function on our larger test lattices.
- Our code scales well with the number of processors in our production environment.
- Our Wilson operator and linear algebra routines are competitive with the QUDA library.
- We have implemented inversion, eigenvalue, and conserved current routines using the latest algorithms.
- We have confirmed that the GMRES(αMGOv) algorithm performs well on large volumes.
- We are able to perform an overlap inversion on 2 CPU/GPUs on a $20^3 \times 24$ lattice at $m_\pi \sim 280 MeV$ in $\sim$ 40 minutes.
- Eigenvalue routines for overlap fermions remain a bottleneck – we are working on it.
- This code will eventually be used for calculations of B_K and ϵ_K to check the ongoing SWME Staggered simulations [13].

References

[1] H. Neuberger *Phys. Lett.* **B417** (1998) 141–144, [hep-lat/9707022]; H. Neuberger *Phys. Rev.* **D57** (1998) 5417–5433, [hep-lat/9710089]; H. Neuberger *Phys. Rev. Lett.* **81** (1998) 4060–4062, [hep-lat/9806025]; R. Narayanan and H. Neuberger *Nucl.Phys.* **B443** (1995) 305–385, [hep-th/9411108].

[2] B. Walk, H. Wittig, E. Dranischnikow, and E. Schomer *PoS* **LATTICE2010** (2010) 044, [arXiv:1010.5636]; B. Walk, H. Wittig, and E. Schmer *The European Physical Journal Special Topics* **210** (2012), no. 1 189–199.

[3] C. Morningstar and M. J. Peardon *Phys. Rev.* **D69** (2004) 054501, [hep-lat/0311018]; P. J. Moran and D. B. Leinweber *Phys. Rev.* **D77** (2008) 094501, [arXiv:0801.1165].

[4] **MILC** Collaboration, A. Bazavov et al. *Rev.Mod.Phys.* **82** (2010) 1349–1417, [arXiv:0903.3598].

[5] M. Clark, R. Babich, K. Barros, R. Brower, and C. Rebbi *Comput.Phys.Commun.* **181** (2010) 1517–1528, [arXiv:0911.3191]; R. Babich, M. A. Clark, and B. Joo arXiv:1011.0024.

[6] Jagels and Reichel *Numerical Linear Algebra with Applications* **1(6)** (1994) 555.

[7] G. Arnold, N. Cundy, J. van den Eshof, A. Frommer, S. Krieg, et al. *Lecture Notes in Computational Science and Engineering* **47** (2003) 153–167, [hep-lat/0311025].

[8] A. Stathopoulos and K. Orginos *SIAM J.Sci.Comput.* **32** (2010) 439–462, [arXiv:0707.0131].

[9] N. Cundy and W. Lee arXiv:1501.0185.

[10] N. Cundy, J. van den Eshof, A. Frommer, S. Krieg, T. Lippert, et al. *Comput.Phys.Commun.* **165** (2005) 221–242, [hep-lat/0405003].

[11] J. Brannick, A. Frommer, K. Kahl, B. Leder, M. Rottmann, et al. arXiv:1410.7170.

[12] A. Frommer, K. Kahl, S. Krieg, B. Leder, and M. Rottmann *SIAM J.Sci.Comput.* **36** (2014) A1581–A1608, [arXiv:1303.1377].

[13] **SWME** Collaboration, J. A. Bailey, Y.-C. Jang, W. Lee, and S. Park arXiv:1503.0538; **SWME** Collaboration, J. A. Bailey, Y.-C. Jang, W. Lee, and S. Park arXiv:1503.0661.