HISQ inverter on Intel® Xeon Phi™ and NVIDIA® GPUs

O. Kaczmarek[(a)], Swagato Mukherjee[(b)], C. Schmidt[(a)], P. Steinbrecher[(a)] and M. Wagner[(c)]

[(a)] Universität Bielefeld - Germany, [(b)] Brookhaven National Laboratory - USA, [(c)] Indiana University - USA

I. Introduction

Conserved charge fluctuations: For the analysis of QCD simulations one often needs to perform many inversions of the Fermion Matrix for a constant gauge field. In finite temperature QCD the calculation of the fluctuation of conserved charges, brayon number (B), electric charge (Q) and strangeness (S) is such an example. Their calculation is particular interesting as they can measured in experiments at RHIC and LHC and also be determined from generalized susceptibilities in Lattice QCD:

$$\chi^{BQS}_{mnk}(T) = \frac{1}{VT^3}\frac{\partial^{m+n+k}\ln\mathcal{Z}}{\partial\left(\mu_B/T\right)^m\partial\left(\mu_Q/T\right)^n\partial\left(\mu_S/T\right)^k}\bigg|_{\vec{\mu}=0} . \quad (1)$$

The required derivatives w.r.t. the chemical potentials can be obtained by stochastically estimating the traces with a sufficiently large number of random vectors η, e.g.

$$\mathrm{Tr}\left(\frac{\partial^{n_1}M}{\partial\mu^{n_1}}M^{-1}\frac{\partial^{n_2}M}{\partial\mu^{n_2}}\ldots M^{-1}\right) = \lim_{N\to\infty}\frac{1}{N}\sum_{k=1}^{N}\eta_k^\dagger\frac{\partial^{n_1}M}{\partial\mu^{n_1}}M^{-1}\frac{\partial^{n_2}M}{\partial\mu^{n_2}}\ldots M^{-1}\eta_k . \quad (2)$$

For each random vector we need to perform several inversions of the Fermion Matrix, depending on the highest degree of derivative we calculate. Typically we use 1500 random vectors to estimate the traces on a single gauge configuration. To also reduce the gauge noise we need between 5000 and 20000 gauge configurations for each temperature. The generated data have been used for several investigations in the last years [1, 2, 3, 4].

For reasons of the numerical costs, staggered fermions are the most common type of fermions for thermodynamic calculations on the lattice. We use the highly improved staggered fermion (HISQ) action. It reduces the taste-splitting as much as possible. The HISQ action uses two levels of fat7 (+ lepage) smearing and a Naik term. In terms of the smeared links X and Naik links N the Dslash operator reads

$$w_x = D_{x,x'}v_{x'} = \sum_{\mu=0}^{4}\left[\left(X_{x,\mu}v_{x+\mu} - X^\dagger_{x-\mu,\mu}v_{x-\mu}\right) + \left(N_{x,\mu}v_{x+3\mu} - N^\dagger_{x-3\mu,\mu}v_{x-3\mu}\right)\right] . \quad (3)$$

Accelerators: In the Krylov solvers used for the inversion the application of the Dslash operator is the dominating term. It typically consumes more than 80% of the runtime. It has a low arithmetic intensity (Flop/byte $\sim$0.73 for single precision). Hence the performance is bound by the available memory bandwidth. These types of problems are well suitable for accelerators as these currently offer memory bandwidths in the range of $200-400$ GB/s and with the use of stacked DRAM are expected to reach 1 TB/s in the next years. Still the most important factor when tuning is to avoid memory access. A common optimization is to exploit available symmetries and reconstruct the gauge links on the fly from 8 or 12 floats instead of loading all 18 floats. For improved actions these symmetries are often broken. For the HISQ action only the Naik-part can be reconstructed from 9 or 13/14 floats.

In the following we will discuss our implementation of the CG inverter for the HISQ action on NVIDIA® GPUs and the Intel® Xeon Phi™ . The GPUs are based on the Kepler architecture as also used in the Titan supercomputer in Oak Ridge. The Xeon Phi is based on the Knights Corner architecture as also used in Tianhe-2.

	Phi™ 5110P	K20	K40	GTX Titan
Cores / SMX	60	13	15	14
(Threads/Core) / (Cores/SMX)	4	192	192	192
Clock Speed [MHz]	1053	706	745/810/875	837
L1 Cache / Core [KB]	32	$16-48$	$16-48$	$16-48$
L2 Cache [MB]	30	1.5	1.5	1.5
Memory Size [GB]	8	5	12	6
peak fp32 [TFlop/s]	2.02	3.52	4.29	4.5
peak fp64 [TFlop/s]	1.01	1.17	1.43	1.5
Memory Bandwidth [GB/s]	320	208	288	288
TDP [W]	225	225	235	250

Tab. 1: Summary of the important technical data of the accelerators we have used in our benchmarks.

Conjugate gradient for multiple right hand sides: At 1500 random vectors for the noisy estimators a large number of inversions are performed for a constant gauge field. Grouping the random vectors in small bundles one can exploit the constant gauge field and apply the Dslash for multiple right hand sides (rhs) at once:

$$\left(w_x^{(1)}, w_x^{(2)}, \ldots, w_x^{(n)}\right) = D_{x,x'}\left(v_{x'}^{(1)}, v_{x'}^{(2)}, \ldots, v_x^{(n)}\right) \quad (4)$$

This increases the arithmetic intensity of the HISQ-Dslash as the loads for the gauge field occur only once for the n rhs.

#rhs	1	2	3	4	5	6	8
Flop/byte (full)	0.73	1.16	1.45	1.65	1.80	1.91	2.08
Flop/byte (r14)	0.80	1.25	1.53	1.73	1.87	1.98	2.14

The numbers above are for single precision (fp32). The numbers for double precision (fp64) differ by a factor of 2. Throughout the following we will only discuss single precision and also neglect approaches like mixed precision. Increasing the number of rhs from 1 to 4 already results in an improvement by a factor of more than 2. For even higher n the relative effect is less significant. In the limit $n\to\infty$ the highest possible arithmetic intensity that can be reached is $\sim$2.75. At $n=8$ we have reached already $\sim$75% of the limiting peak intensity while for 1 rhs we only obtain $25-30$%. It is also obvious that for an increasing number of gauge fields the memory traffic caused by the loading of the gauge fields is no longer dominating and the impact of reconstructing the Naik links reduces from $\sim$10% for a single rhs to $\sim$ 3% for 8 rhs. The additional register pressure due to the reconstruction of the gauge links might then also result in a lower performance. For the full conjugate gradient the additional linear algebra does not allow for the reuse of any constant fields. The effect of the increased arithmetic intensity of the Dslash while therefore be less pronounced in the full CG.

II. GPU

Architecture: NVIDIA's current architecture for compute GPUs is called Kepler, the corresponding chip GK110. The latest compute card, the Tesla K40, comes with a slightly modified version GK110B and GPU Boost. The latter allows the user to run the GPU at a higher core clock. As memory-bandwidth bound problems usually stay well within the thermal and power envelopes the card is capable of constantly running at this higher clock for Lattice QCD simulations. The memory clock remains constant and thus a performance impact on bandwidth-bound applications is not obvious. However, the higher core clock allows to better saturate the available bandwidth. We will only show results with the highest possible core clock for the K40.

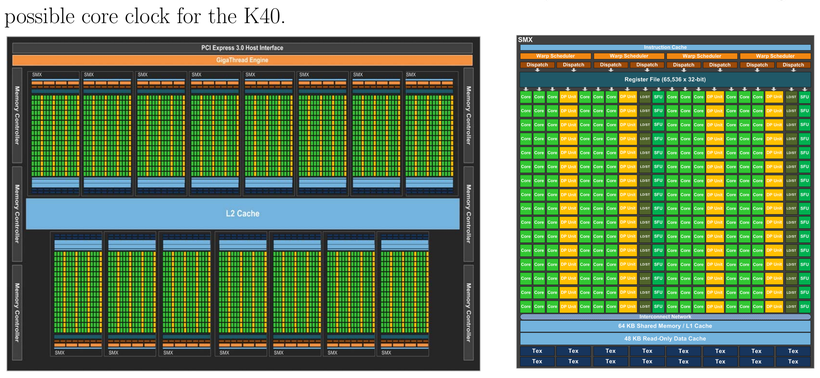

Fig. 1: The GK110 chip and one of its SMX processors [5].

The GK110 chip consists of several streaming multiprocessors ("SMX"). The number of SMX depends on the card. Each SMX features 192 CUDA cores. Within a SMX they share a configurable L1 cache / shared memory area and a 48KB read-only/texture cache. The shared L2 cache for all SMX is relatively small with only 1.5MB.

GPU-Implementation: For a bandwidth-bound problem the memory layout and access is crucial to achieve optimal performance. As GPUs have been used for several years to accelerate LQCD simulations a lot of the techniques we used may be considered as standard by now. We use a Structure of Arrays (SoA) memory layout for both the color vectors and the gauge links. We reconstruct the Naik links from 14 floats. We observed best results by loading the gauge links through the texture unit and the color vectors through the standard load path (L1).

For the implementation of the Dslash for multiple rhs two approaches are possible. On the GPU the obvious parallelization for the Dslash for one rhs is over the elements of the output vector, i.e., each thread processes one element of the output vector. The first approach (register-blocking) lets each thread multiply the already loaded gauge link to the corresponding element of multiple right-hand sides. The thread thus generates the element for one lattice site for several output vectors. This approach increases the number of registers needed per thread and will result in a lower GPU occupancy or at some point spilling. Both effects will limit the achievable performance, while the latter is less likely as each thread can use up to 255 registers for the Kepler architecture.

The second approach (texture cache blocking) is to let each thread process one element of one output vector and group the threads into two-dimensional CUDA blocks with lattice site x and rhs i. As one CUDA block is executed on one SMX this ensures temporal locality for the gauge links in the texture cache. Ideally the gauge links only need to be loaded from the global memory for one rhs. When the threads for the other rhs are executed they are likely to obtain the gauge links from cache. This approach does not increase the register pressure. Furthermore the total number of threads is increased by a factor n and this may furthermore improve the overall GPU usage.

Both approaches can also be combined and the best possible solution is a question of tuning. For our benchmarks we determine the optimal configuration for a given lattice size and number of rhs for each GPU. Furthermore we employ an automatic tuning to select the optimal launch configuration for the Dslash operation also depending on the GPU, lattice size and number of rhs.

The remaining linear algebra-Kernels are kept separate for the different rhs. This allows us to easily stop the solver for individual rhs that have already met the convergence criterion. To hide latencies and allow for a better usage of the GPU we use separate CUDA streams for each right hand side. We keep the whole solver on the GPU and only communicate the set of residuals for all rhs at the end of each iteration.

III. MIC

Architecture: The Intel® Xeon Phi™ is an in-order x86 based many-core processor [6]. The accelerator runs a Linux μOS and can have up to 61 cores combined via a bidirectional ring (see figure 2). Therefore, the memory transfers are limited by concurrency reaching only 140 GB/s for a stream triad benchmark [7]. Each core has a private 32 KB L1 data and instruction cache, as well as a global visible 512 KB L2 cache. In the case of an local L2 cache miss a core can cross-snoop another's core L2 cache. If the needed data is present it is send through the ring interconnect, thus avoiding a direct memory access.

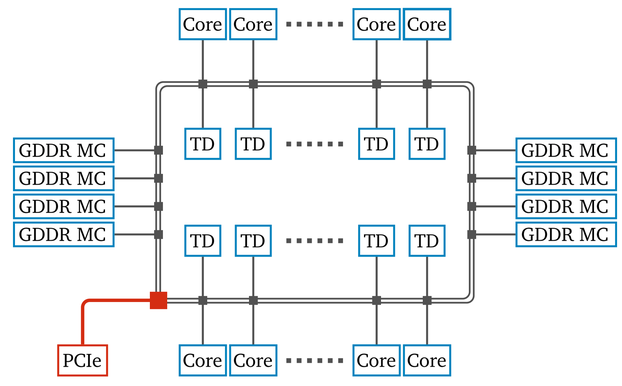

Fig. 2: Visualization of the bidirectional ring on the die. Each core has its own tag directory (TD) keeping the cache hierarchy fully coherent. The dotted line illustrates the missing cores.

One core has thirty-two 512 bit `zmm` vector registers corresponding to any multiple of a 32/64 bit floating-point number or integer (see figure 3). The Many Integrated Core (MIC) has its own SIMD instruction set extension `IMIC` with support for Fused Multiply Add (FMA) and mask operations. Each core has 4 hardware context threads scheduled with a round-robin algorithm delivering two executed instructions per cycle while running with at least two threads per core. In order to fully utilize the MIC it is mostly required to run with four threads per core. Especially for memory bound applications using four threads offers more flexibility to the processor to swap the context of a thread, which is currently stalled by a cache miss. The MIC has support for streaming data directly into memory without reading the original content of an entire cache line, thus bypassing the cache and increasing the performance of algorithms where the memory footprint is too large for the cache [6].

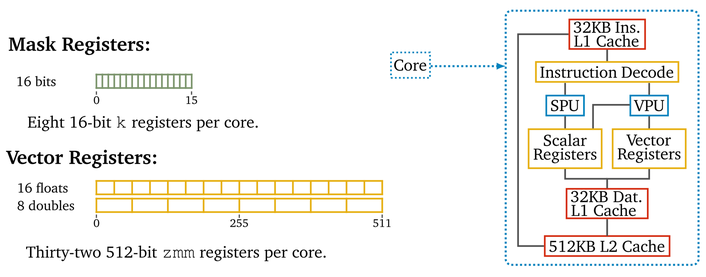

Fig. 3: The microarchitecture of one core (r.h.s.) showing the cache hierarchy, Scalar Processing Unit (SPU) and Vector Processing Unit (VPU). The l.h.s. visualizes the mask and vector register data types.

MIC-Implementation: We have parallelized our program with OpenMP over all HW threads and vectorized it using low-level compiler functions called intrinsics. Those assembly-coded functions are expanded inline and do not require explicit register management or instruction scheduling through the programmer as in pure assembly code. There are 512 bit intrinsics data types for single- and double-precision accuracy as well as for integer values. More than 32 variables of a 512 bit data type can be used simultaneously. With only 32 `zmm` registers available in hardware, the compiler is, in this case, forced to use spilling. Also using intrinsics, the software, therefore, has to be designed in a register aware manner; only the explicit management of the registers is taken over by the compiler. However, we found that the compiler is only able to optimize code over small regions. Thus, the order of intrinsics has an influence on the achieved performance, thereby making optimizations more difficult. We assume this issue is caused by the lack of out-of-order execution.

Site fusion: One problem of using 512 bit registers involving SU(3) matrix vector products is that one matrix does not fit into an integer number of `zmm` registers without padding. Because of that, it is more efficient to process several matrix vector products at the same time using a site fusion method. A naive single-precision implementation could be to create a "Struct of Arrays" (SoA) object for 16 matrices as well as for 16 vectors. Such a SoA vector object requires 6 `zmm` registers when it is loaded from memory. One specific register then refers to the real or imaginary part of the same color component gathered from all 16 vectors, thus each vector register can be treated in a "scalar way". These SoA matrix/vector objects are stored in an array with a site ordering technique. Our Dslash kernel runs best with streaming through xy-planes and is specifically adapted for inverting multiple right-hand sides. Therefore, we use a 8-fold site fusion method, combining 8 sites of the same parity in x-direction, which makes the vector arithmetics less trivial and requires explicit in-register align/blend operations. By doing so we reduce the register pressure by 50% compared to the naive 16-fold site fusion method, which is a crucial optimization for a multiple right-hand side inverter.

Prefetching: For indirect memory access, i.e. the array index is a non-trivial calculation or loaded from memory, it is not possible for the compiler to insert software prefetches. The MIC has a L2 hardware prefetcher which is able to recognize simple access pattern in such a way that it schedules a prefetch instruction before the data is actually needed. We found that it does a very good job for a linear memory access. Thus, there is no need for software prefetching by hand inside the linear algebra operations of the CG. However, the access pattern of the Dslash kernel is too complicated for the hardware prefetcher. Therefore, it is required to insert L2 and L1 software prefetches using intrinsics, since a cache miss on an in-order CPU is the worst-case scenario. The HISQ inverter runs 2× faster with inserted software prefetches.

IV. Comparison

We performed our benchmarks for a single accelerator. We used CUDA 6.0 for the GPU and the Intel compiler 14.0 for MIC. We used the default settings of MPSS 3.2 and a balanced processor affinity. First we discuss the performance as a function of the number of rhs. The maximum number of rhs is furthermore limited by the memory requirements.

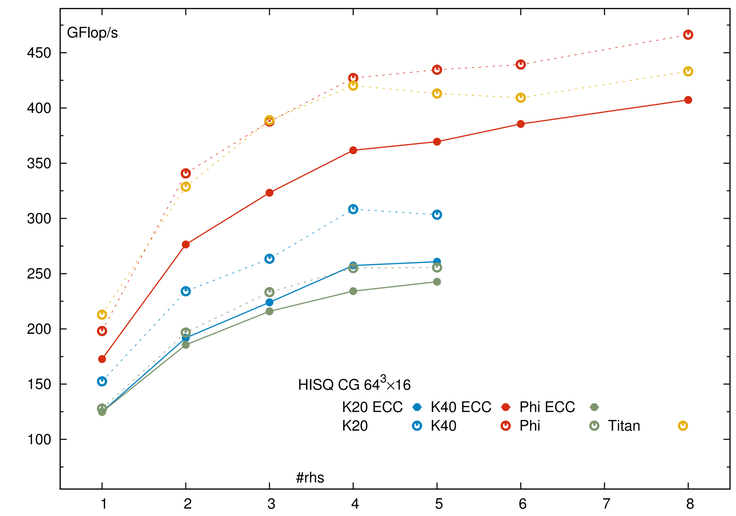

Fig. 4: Performance of the CG inverter on different accelerators for a $64^3\times16$ lattice as a function of the number of rhs. The dashed lines corresponds to ECC disabled devices.

We observe roughly the expected scaling from the increased arithmetic intensity. When comparing the results for four right hand sides to one right hand side we see improvements by a factor of about 2, close to the observed increase in arithmetic intensity for the Dslash. For the full CG the linear algebra operations were expected to weaken the effect of the increased arithmetic intensity for the Dslash.

At four right hand sides we obtain about 10% of the theoretical peak performance. For the Xeon Phi it is a bit more ($\sim$12%), but its theoretical peak performance is also the lowest of all accelerators used here while its theoretical memory bandwidth is the highest. For the GPUs we observe a performance close to the naive estimate (arithmetic intensity) × (theoretical memory bandwidth). As previously report the theoretical memory bandwidth is nearly impossible to reach on the Xeon Phi. However, our performance numbers are in agreement with the estimate (arithmetic intensity) × (memory bandwidth from the stream benchmark).

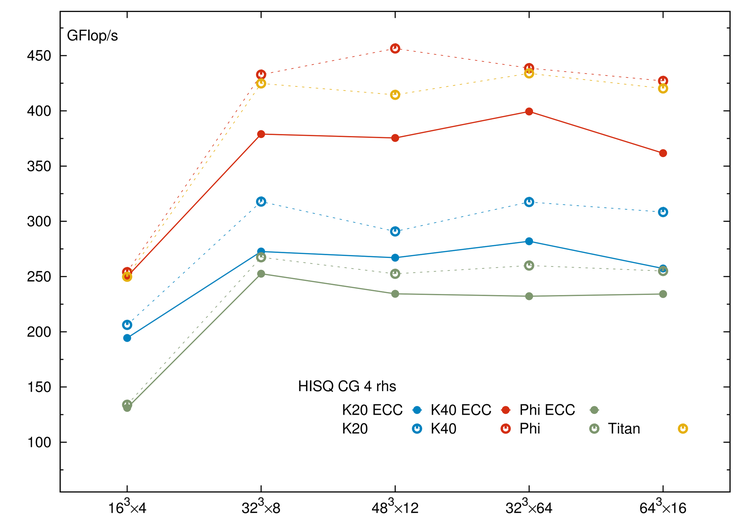

Fig. 5: Performance of the CG inverter for 4 rhs on different accelerators as a function of the lattice size. The dashed lines corresponds to ECC disabled devices.

If we consider the performance of the CG for a a fixed number of rhs (here $n=4$) we observe that the best performance is only obtained for lattice sizes larger than $32^3\times8$. With our code the Xeon Phi is slightly slower than a K20. The K40 is another $30-40$% faster. That is consistent with the increase in the theoretical memory bandwidth. For cases where disabled ECC is acceptable the significantly cheaper gaming / amateur card Titan GTX achieves a performance close to the K40.

Energy consumption: A further point that we quickly checked was the typical energy consumption of the accelerator. For four right-hand sides and lattice sizes between $32^3\times8$ and $64^3\times16$ we observed values $\sim$125 W for the K20 and $\sim$185 W for the K40 without ECC. The Xeon Phi consumed the most energy at about 200 W. These numbers have been measured using the system / accelerator counters and do not include a host system. The resulting efficiency for the Kepler architecture is hence about 2.25 (GFlop/s)/W. For the Xeon Phi we estimate $\sim$ 1.5(GFlop/s)/W at four right hand sides.

Acknowledgment: We acknowledge support from NVIDIA® through the CUDA Research Center program. We thank Mike Clark for providing access to a Titan GTX card for benchmarks. Furthermore, we would like to thank the Intel® Developer team for their constant support.

References

[1] A. Bazavov et al., `arXiv:1404.4043 [hep-lat]`.

[2] A. Bazavov et al., `arXiv:1404.6511 [hep-lat]`.

[3] A. Bazavov et al., Phys. Rev. Lett. **111**, 082301 (2013).

[4] A. Bazavov et al., Phys. Rev. Lett. **109**, 192302 (2012).

[5] *Nvidia GK110 whitepaper*, `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[6] *Intel Xeon Phi Coprocessor System Software Developers Guide.*

[7] J. D. McCalpin, `http://www.cs.virginia.edu/stream/`