PROCEEDINGS OF SCIENCE

LatticeQCD using OpenCL

Owe Philipsen, Christopher Pinke[*], Christian Schäfer, Lars Zeidlewicz
Institut für Theoretische Physik - Johann Wolfgang Goethe-Universität
Max-von-Laue-Str. 1, 60438 Frankfurt am Main
E-mail: philipsen, pinke, cschaefer, zeidlewicz @th.physik.uni-frankfurt.de

Matthias Bach
Frankfurt Institute for Advanced Studies / Institut für Informatik - Johann Wolfgang Goethe-Universität
Ruth-Moufang-Str. 1, 60438 Frankfurt am Main
E-mail: bach@compeng.uni-frankfurt.de

We report on our implementation of LatticeQCD applications using OpenCL. We focus on the general concept and on distributing different parts on hybrid systems, consisting of both CPUs (Central Processing Units) and GPUs (Graphic Processing Units).

XXIX International Symposium on Lattice Field Theory
July 10 - 16 2011
Squaw Valley, Lake Tahoe, California

[*]Speaker.

1. Introduction

Graphic Processing Units (GPUs) offer a computing architecture well suited for LatticeQCD applications. Consequently, there is an on-going software- and algorithm development in order to incorporate GPUs effectively into lattice simulations. See for example [1, 2, 3, 4]. These applications are developed and carried out predominantly on NVIDIA hardware, consistently using the NVIDIA exclusive CUDA language [5] for the interaction with the GPU. Using GPUs is attractive because they have good price-per-flop (e.g. $\approx$ 2,0 €/ Gflop on a NVIDIA GTX 580 and even $\approx$ 0,4 €/ Gflop on an AMD 6970 [6]) and performance-per-watt ratios.

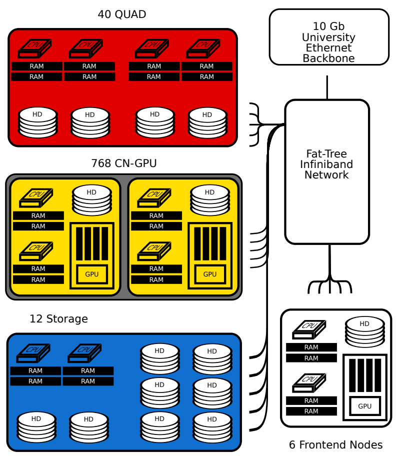

40 QUAD
10 Gb University Ethernet Backbone
RAM
HD
768 CN-GPU
GPU
Fat-Tree Infiniband Network
12 Storage
6 Frontend Nodes

Figure 1: LOEWE-CSC

Recently, a new cluster was introduced at Frankfurt University, the "LOEWE-CSC" [7]. It is dedicated to high-performance computing, but contrary to existing clusters it solely consists of AMD hardware. A sketch of its infrastructure is shown in Fig. 1. A striking feature is its heterogeneous architecture: The majority of its compute nodes each hold two 12-core AMD Magny-Cours Central Processing Units (CPUs) and one AMD RADEON 5870 GPU. The LOEWE-CSC is ranked 22 in the Top500 list of supercomputers [8] and rank 10 in the Green500 list of energy-efficient supercomputers (with 718 Mflops/Watt) [9].

However, presently existing GPU appplications are mostly suitable for NVIDIA hardware. Other than using graphic application programming interfaces (APIs) like OpenGL [10], the only tool available to use AMD GPUs for general purposes is OpenCL [11]. This is an open standard for parallel programming. Furthermore, it is explicitly designed for heterogeneous (or hybrid) systems, thus being well suited for the LOEWE-CSC as well as other, non-GPU platforms. Implementations of OpenCL can be found both from AMD (AMD Accelerated Parallel Processing (APP) [13], formerly ATI Stream SDK) and NVIDIA (as part of CUDA).

The first lattice simulations in OpenCL were performed in [1] with staggered fermions. On NVIDIA hardware, a significantly lower performance (25% on C1060 and 60% on S2050) of OpenCL was reported compared to CUDA for Hybrid Monte Carlo (HMC) updates. An AMD GPU was also considered. Here, OpenCL performance is better than on Nvidia hardware, but still below CUDA results (50% less performance on an AMD 5870 in OpenCL than on a S2050 in CUDA).

2. OpenCL

In the following, we will present the general ideas of OpenCL and explain important terms. For more information see [12]. The generic concept of an OpenCL application consists of a "host" program and several "compute devices" (see Fig. 2). They live together on a so called "platform". The host controls memory management and calculations carried out on the devices, while each device may either be a GPU, a CPU or any other kind of supported compute device. A single device then consists of a bunch of ”compute units“ (on a multicore CPU this would be a single core) which in turn can consist of one or more ”processing elements“ that carry out the actual computations. This of course reflects the architecture of GPUs. On CPUs a compute unit (i.e. a single core) and a processing element may also be the same, although this depends on the specific model and OpenCL implementation used. It should be noted that on a CPU it is possible to split up a multicore CPU into several devices, effectively grouping the cores suited for specific tasks.

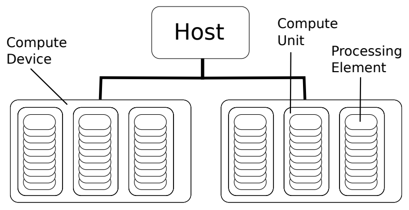


Figure 2: OpenCL Concept

Furthermore, execution commands for OpenCL functions (”kernels”) are scheduled in one or more “command queues” via the host. The queue launches execution of kernels on a specific device and also handles memory commands. Synchronization on this level is possible only within a command queue.

Central objects of any OpenCL application are the kernels. These have to be written in the `OpenCL C` programming language, which is based on a subset of `C99`, the `C` standard. Optionally, “native kernels“ can be included from libraries. Kernels are executed using an up to 3-dimensional index space, where each index can be mapped on an instance of the kernel, which in turn are called ”work items”. Several work items make up a ”work group“, which allows for synchronization between work items. Kernels can access memory on various levels, ranging from ”global” (e.g. the main memory on the GPU) to “private” (e.g. General Purpose Registers (GPRs) of a GPU stream-core). Besides the nomenclature, the setup is very similiar to CUDA.

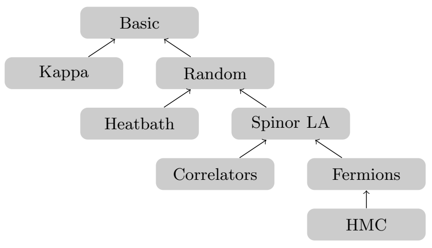
It is important to emphasize here that OpenCL allows for data- as well as for task parallel applications, meaning that it is on the one hand possible to perform SIMD computations and on the other hand to perform different (possibly independent) tasks in a parallel fashion, providing OpenMP [14] or UNIX’s pthreads functionality automatically.

In order to have a hardware-independent programming model, the actual OpenCL program is compiled and built at runtime of the (host) application. This is done using an OpenCL inherent compiler.

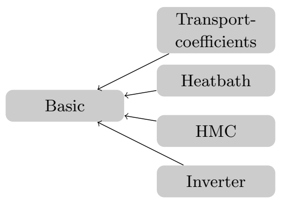
3. Implementation

The physical problems we are currently interested in are investigations of the quark gluon plasma (QGP) and the thermal transition of QCD with dynamical fermions [15, 16, 17] as well as in pure gauge theory (PGT). These have yet been carried out mainly relying on the `tmlqcd` program suite [18] and an application written with `QDP++` [19]. Several features shall thus be provided by the OpenCL application: On the PGT side we need a SU(3) heatbath algorithm [20], whereas on the fermionic side we require an HMC algorithm including standard features (even-odd preconditioning, 2MN integrator, multiple integration timescales) for $N_f = 2$ twisted-mass Wilson fermions [18, 21]. Also, ILDG-compatible I/O is required [22].

In order to account for the fundamentally different concept of OpenCL we decided to write an all new program. The implementation of the desired features mentioned above resulted in four executables, providing the possiblity to generate gauge configurations as well as calculate physical observables of interest for pure gauge theory and dynamical fermions. All calculations are performed in OpenCL. In the following we will go to some details describing the concrete implementation.



(a) OpenCL module classes



(b) Gaugefield classes

Figure 3: Structure of opencl modules- and gaugefield classes

The host program was set up in `C++` in order to implement independent program parts easily using `C++` classes and to have extension capabilities in a natural way.

The central object is the class `gaugefield`, which incorporates the initialization of OpenCL and holds an application-specific number of `opencl_device`-objects. As the name indicates, the latter class contains all compute device-related parts, such as kernels or memory objects, and eventually executes the kernels on a specific device. The class `gaugefield` is also dedicated to synchronize the physical gauge field when it is used on several devices.

The specific physical problems were implemented as child classes of `gaugefield` and `opencl_module`, containing the problem-related functionality (see Fig. 3). Furthermore, for each physical problem a number of “tasks” can be defined which then again can contain a number of device objects to carry out this task. For example, the `inverter` executable essentially performs two tasks, the inversion of the fermion matrix and the calculation of correlators. This concept will prove useful when looking at hybrid applications.

As was mentioned in section 2, the OpenCL environment has to go through certain initialization processes before the actual calculations can be carried out. A schematic flow of this process is shown in Fig. 4. The major part of the initialization is spent on generating the kernels. This is done in a couple of steps: First, the files needed for the kernel code are collected, after that an OpenCL “program“ is compiled and linked using the OpenCL compiler. This program can then be used to build kernels referring to functions declared with `__kernel` within the previously read-in source code.

We found it convenient to build every kernel as a stand alone program since the kernel is the object of interest. The compiler generates binary files which can be used to extract informations about the kernel, e.g. GPR usage, which is useful for benchmarking and optimization. They can also be reused for kernel generation at a later program run. This speeds up the initialization time significantly.

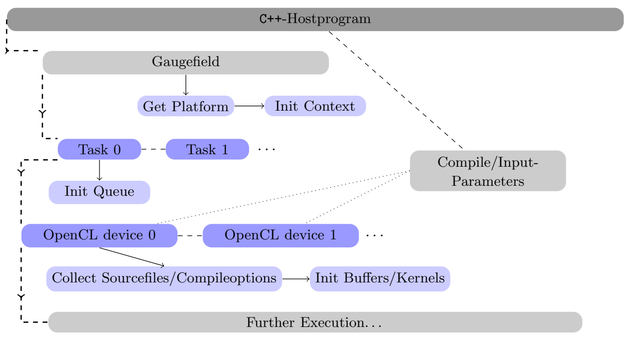
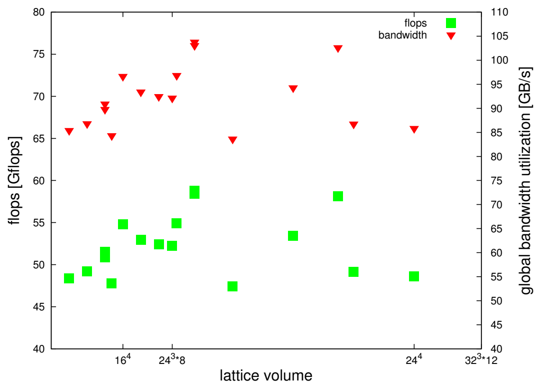


Figure 4: Schematic flow of program initialization

One can influence the whole process at this point by means of the simulation parameters. The latter are typically read in at runtime of the host, so one can e.g. switch between CPU and GPU simply via the input file. Since the kernels are compiled only at runtime, one can pass the simulation parameters (e.g. `NT`, `NS`, $\beta, \dots$) to them as compile time parameters. This is a nice way of avoiding many kernel arguments as well as to "hard code" the parameters into the kernel code.



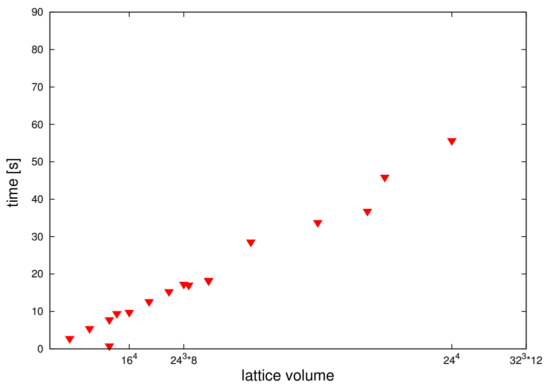


Figure 5: Dslash performance on AMD 5870 on various even-odd preconditioned lattices (statistics are of $\mathcal{O}(10k)$ for each volume).

In fermionic applications, the most time-consuming part is the inversion of the fermion matrix and the non-diagonal part of the Dirac matrix ("dslash"), respectively. On GPUs, this problem is always bandwidth-limited and tuning is required to achieve a satisfiying amount of the maximum bandwidth (on an AMD 5870 this is e.g. 154 GB/s). Our (even-odd preconditioned) dslash implementation currently performs at 45 - 60 GFlops in double precision calculations (with a bandwidth utilization of up to 105 GB/s) over a wide range of lattice sizes (See Fig. 5). We will give more performance results for AMD hardware (also considering memory optimizations like `RECONSTRUCT TWELVE` [3]) in a future publication.

4. Hybrid strategies

Having a hybrid system as the LOEWE-CSC at hand, the question arises how one can use this infrastructure effectively. Both GPU and CPU hold several advantages, qualifying them for certain tasks. GPUs outperform CPUs when it comes to floating point operations, whereas a CPU can in general operate a bigger amount of memory and a bigger cache, just to name a few. OpenCL can be used quite easily to distribute computations over a hybrid system.

A typical scenario in lattice simulations is the iterative calculation of some observables out of a sequence of gauge configurations. The simplest case one can have is an observable that does not require any synchronization in between its calculation. Given 2 devices, one can distribute two generalized tasks among them, as depicted in Fig. 6. One task calculates the observable while in the meantime the other provides the ingre-

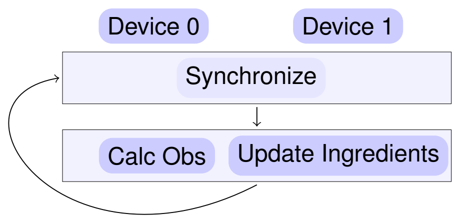


Figure 6: Simplest hybrid strategy.

dients required for the next iteration's calculation. Synchronization between the devices is carried out at the start of each iteration.

We implemented this concept for two observables, the $N_f = 2$ mesonic flavour doublet correlators with quantum number Γ,

$$C_\Gamma = -\operatorname{Tr}\left(S_u^\dagger(x_0,x)\gamma_5\Gamma S_u(x_0,x)\Gamma\gamma_5\right) , \tag{4.1}$$

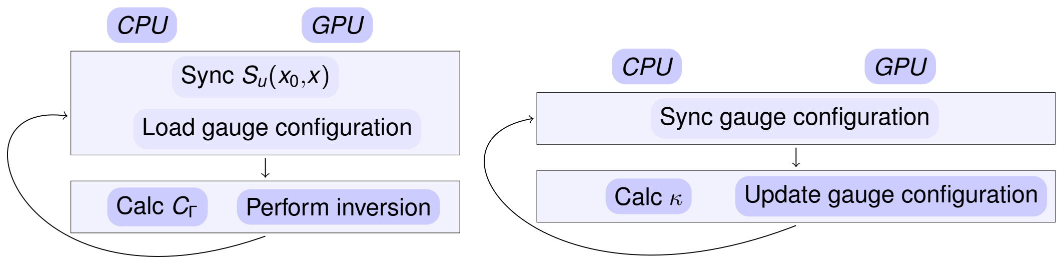
for which one has to provide the propagator $S_u(x_0,x) \sim M^{-1}b(x_0)$ (where M is the fermion matrix and b a point source at site x_0), and the second order transport coefficent κ of the Quark Gluon Plasma [23]. κ can be extracted from the retarted propagator G_R at zero Matsubara frequency,

$$G_R(\omega = 0,\vec{q}) = G(0) - \frac{\kappa}{2}|\vec{q}\,|^2 + \mathscr{O}(|\vec{q}\,|^3) , \tag{4.2}$$

which can be calculated on a given gauge configuration by the euclidean correlator G_E according to

$$G_R(\omega = 0,\vec{q}) = G_E(\omega = 0,\vec{q}) = N \sum_{x,y} \mathrm{e}^{q_3(x_3-y_3)} \langle T_{12}(x)T_{12}(y)\rangle . \tag{4.3}$$

$T_{\mu\nu}$ is a discretization of the energy-momentum tensor using clover-plaquettes [24].



(a) Mesonic correlator C_Γ (b) Transportcoefficient κ

Figure 7: Hybrid strategies for two observables of interest.

The concrete implementations can be seen in Fig. 7, where we considered the case of one CPU and one GPU device in the system (note that e.g. on one LOEWE-CSC node, one CPU device has by default 2*12 = 24 cores). The assignments of the different tasks to the devices came quite naturally, since κ's calculation requires much more memory ressources than the heatbath algorithm and the inversion of the fermion matrix is carried out faster on the GPU.

5. Conclusions

We presented to some detail our implementation of LatticeQCD applications using OpenCL. This is a quite different approach compared to other applications around, which mainly operate on NVIDIA hardware using CUDA. Since OpenCL is a hardware-independent programming model, any hardware can be used, which offer an optimal price-per-flop ratio. Especially, parallel calculations can be performed quite simply in OpenCL, allowing for applications suited for hybrid architectures. We gave two examples using GPU and CPU devices at the same time effectively. We are currently benchmarking and optimizing our code to exploit the compute powers of the LOEWE-CSC and AMD hardware in general, providing an alternative to NVIDIA hardware bound applications.

Acknowledgments

O. P. and C. P. are supported by the German BMBF grant *FAIR theory: the QCD phase diagram at vanishing and finite baryon density*, 06MS9150. L.Z. is supported by the DFG grant *phase transition and screening masses in $N_f = 2$ QCD*, PH 158/3-1. M. B., O. P, and C. S. are supported by the Helmholtz International Center for FAIR within the LOEWE program of the State of Hesse. M.B. is supported by the GSI Helmholtzzentrum für Schwerionenforschung. C.P. acknowledges travel support by the Helmholtz Graduate School HIRe for FAIR.

References

[1] C. Bonati, G. Cossu, M. D'Elia and P. Incardona, arXiv:1106.5673 [hep-lat].

[2] R. Babich, M. A. Clark and B. Joo, arXiv:1011.0024 [hep-lat].

[3] M. A. Clark, R. Babich, K. Barros, R. C. Brower and C. Rebbi, Comput. Phys. Commun. **181** (2010) 1517 [arXiv:0911.3191 [hep-lat]].

[4] R. Babich, M. A. Clark, B. Joo, G. Shi, R. C. Brower and S. Gottlieb, arXiv:1109.2935 [hep-lat].

[5] http://developer.nvidia.com/category/zone/cuda-zone

[6] http://geizhals.at/

[7] http://compeng.uni-frankfurt.de/index.php?id=86

[8] http://www.top500.org/list/2011/06/100

[9] http://www.green500.org/lists/2011/06/top/list.php

[10] http://www.opengl.org/

[11] http://www.khronos.org/opencl/

[12] Khronos Working Group, *The OpenCL Specification*, http://www.khronos.org/registry/cl/.

[13] http://developer.amd.com/SDKS/AMDAPPSDK

[14] http://openmp.org/wp/

[15] E. M. Ilgenfritz, K. Jansen, M. P. Lombardo, M. Muller-Preussker, M. Petschlies, O. Philipsen and L. Zeidlewicz, Phys. Rev. D **80** (2009) 094502 [arXiv:0905.3112 [hep-lat]].

[16] O. Philipsen and L. Zeidlewicz, Phys. Rev. D **81** (2010) 077501 [arXiv:0812.1177 [hep-lat]].

[17] F. Burger *et al.*, arXiv:1102.4530 [hep-lat].

[18] K. Jansen and C. Urbach, Comput. Phys. Commun. **180** (2009) 2717 [arXiv:0905.3331 [hep-lat]].

[19] http://usqcd.jlab.org/usqcd-docs/qdp++/

[20] M. Creutz, Phys. Rev. **D21** (1980) 2308-2315.; N. Cabibbo and E. Marinari, Phys. Lett. B **119** (1982) 387.; A. D. Kennedy, B. J. Pendleton, Phys. Lett. **B156** (1985) 393-399.

[21] A. Shindler, Phys. Rept. **461** (2008) 37 [arXiv:0707.4093 [hep-lat]].

[22] http://cssm.sasr.edu.au/ildg/

[23] P. Romatschke and D. T. Son, Phys. Rev. D **80** (2009) 065021 [arXiv:0903.3946 [hep-ph]].

[24] Y. Maezawa, H. Abuki, T. Hatsuda and T. Koide, PoS **LATTICE2010** (2010) 201 [arXiv:1012.2222 [hep-lat]].