CL²QCD - Part 1

M. Bach‡, O. Philipsen†, C. Pinke† and A. Sciarra†

† Institut für Theoretische Physik, Goethe Universität, Frankfurt am Main, Germany
‡ Frankfurt Institute for Advanced Studies / Institut für Informatik, Goethe-Universität, Frankfurt am Main, Germany

We present the **Lattice QCD (LQCD)** application CL²QCD, which is based on OpenCL and can be run on Graphic Processing Units (GPUs) as well as on common CPUs. We focus on implementation details as well as performance results of selected features. CL²QCD has been successfully applied in LQCD studies at finite temperature and density and is available at http://code.compeng.uni-frankfurt.de/projects/clhmc.

Lattice QCD

- **Importance Sampling:** Generate gauge configurations $\{U_m\}$ using Boltzmann-weight $p[U,\phi] = \exp\{-S_{\text{eff}}[U,\phi]\}$ with **Hybrid-Monte-Carlo (HMC)** algorithm [Duane et al., 1987].
  - Observables:

$$\langle K \rangle \approx \frac{1}{N} \sum_m K[U_m] \, ;$$

  - Effective action S_{eff} proportional to inverse fermion matrix D^{-1}.
- Most expensive ingredient to current LQCD simulations: **fermion matrix inversion**

$$D\phi = \psi \quad \Rightarrow \quad \phi = D^{-1}\,\psi \, .$$

  - Carried out with Krylov subspace methods, e.g. conjugate gradient (CG);
  - Matrix-vector product $D\phi$ has to be carried out multiple times.
- LQCD functions local (depend on a number of nearest neighbours only) ⇒ very well suited for parallelization.
- LQCD operations **limited by memory bandwidth**. Most expensive part: Derivative part of D, so-called $\not{D}$.

$$\text{Numerical density } \rho = \frac{\text{Number of FLOPs}}{\text{Number of Bytes to read and write}} \quad \Rightarrow \quad \begin{array}{l} \text{– } \textbf{Wilson fermions: } \rho(\not{D}) \sim 0.57 \\ \text{– } \textbf{Staggered fermions: } \rho(D_{KS}) \sim 0.35 \end{array}$$

⇒ LQCD requires hardware with a high memory-bandwidth to run effectively
⇒ Meaningful measure for efficiency is achieved bandwidth

OpenCL and Graphic Cards

	Chip	Peak SP {GFLOPS}	Peak DP {GFLOPS}	Peak BW {GB/s}
AMD Radeon HD 5870	Cypress	2720	544	154
AMD Radeon HD 7970	Tahiti	3789	947	264
AMD FirePro S10000	Tahiti	2×3410	2×850	2×240
NVIDIA GeForce GTX 680	Kepler	3090	258	192
NVIDIA Tesla K40	Kepler	4290	1430	288
AMD Opteron 6172	Magny-Cours	202	101	43
Intel Xeon E5-2690	Sandy Bridge EP	371	186	51

Table 1: *Theoretical peak performance of current GPUs and CPUs. SP and DP denote single and double precision, respectively. BW denotes bandwidth.*

	LOEWE -CSC
GPU nodes	786
GPUs/node	1 × AMD 5870
CPUs/node	2 × Opteron 6172
	SANAM
GPU nodes	304
GPUs/node	2 × AMD S10000
CPUs/node	2 × Xeon E5-2650

Table 2: *AMD based clusters where CL²QCD was used for production runs.*

- Graphics Processing Units (GPUs) surpass CPUs in peak performance as well as in memory bandwidth (see Table 1);
- GPUs can be used for general purposes;
- Many computing clusters are accelerated by GPUs, for example LOEWE -CSC in Frankfurt [Bach et al., 2011] or SANAM [Kalcher et al., 2013] (see Table 2);
- GPUs constitute inherently parallel architecture;
- LQCD applications are always memory-bandwidth limited ⇒ they can benefit from GPUs tremendously;
- In recent years the usage of GPUs in LQCD simulations has increased, mainly relying on CUDA as computing language, applicable to NVIDIA hardware *only*[1].

OpenCL	Device	Compute Unit	Processing element	Local memory	Private memory	Work – group	Work item
CUDA	GPU	Multiprocessor	Scalar core	Shared (per-block) memory	Local memory	Block	Thread

Table 3: *OpenCL and CUDA terminology.*

- Hardware independent approach to GPU applications given by *Open Computing Language* (**OpenCL**)[2].
- OpenCL is an open standard to perform calculations on heterogeneous computing platforms ⇒ GPUs and CPUs can be used together within the same framework, taking advantage of their synergy and resulting in a high portability of the software. First attempts to do this in LQCD in [Philipsen et al., 2011].
- See Table 3 for a comparison of CUDA and OpenCL terminology.

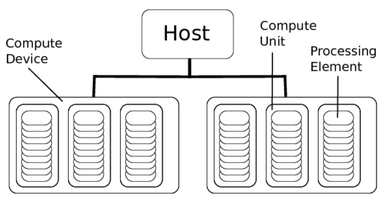

Figure 1: *OpenCL concept*

- An OpenCL application consists of a *host* program coordinating the execution of the actual functions, called *kernels*, on *computing devices* (Figure 1). A device can for instance be a GPU or a CPU.
- Although the hardware has different characteristics, GPU programming shares many similarities with parallel programming of CPUs. A computing device consists of multiple *compute units*. When a kernel is executed on a computing device, actually a huge number of kernel instances is launched. They are mapped onto *work-groups* consisting of *work-items*. The work-items are guaranteed to be executed concurrently only on the processing elements of the compute unit (and share processor resources on the device).

- Compared to the main memory of traditional computing systems, on-board memory capacities of GPUs are low, though increasing more and more[3]. This constitutes a clear boundary for simulation setups. Also, communication between host system and GPU is slow, limiting workarounds in case the available GPU memory is exceeded. Nevertheless, as finite T studies are usually carried out on moderate lattice sizes (in particular $N_\sigma \gg N_\tau$), this is less problematic for the use cases CL²QCD was developed for.

[1] See https://developer.nvidia.com/cuda-zone and the **QUDA** library: https://github.com/lattice/quda .
[2] See https://www.khronos.org/opencl .
[3] For instance, the GPUs given in Table 2 have on-board memory capacities of 1 GB and 3 GB, respectively.

CL²QCD Features

- First OpenCL application for Wilson fermions [Bach et al., 2013], focusing on **Twisted Mass Wilson fermions** [Frezzotti and Rossi, 2004; Shindler, 2008] (automatic $\mathcal{O}(a)$ improvement at maximal twist);
- **Staggered fermions** in standard formulation;
- Improved gauge actions;
- Standard inversion and integration algorithms;
- ILDG-compatible IO[4];
- RANLUX Pseudo-Random Number Generator (PRNG)[5] [Lüscher, 1994].

Executables:

- **HMC:** Generation of gauge field configurations for $N_f = 2$ (Twisted Mass) Wilson type fermions using the HMC algorithm [Duane et al., 1987];
- **RHMC:** Generation of gauge field configurations for staggered type fermions using the Rational HMC algorithm [Clark and Kennedy, 2007];
- **SU3HEATBATH:** Generation of gauge field configurations for $SU(3)$ Pure Gauge Theory using the heatbath algorithm [Cabibbo and Marinari, 1982; Creutz, 1980; Kennedy and Pendleton, 1985];
- **INVERTER:** Measurements of fermionic observables on given gauge field configurations;
- **GAUGEOBSERVABLES:** Measurements of gauge observables on given gauge field configurations.

[4] Via LIME, see http://usqcd.jlab.org/usqcd-docs/c-lime .
[5] See https://bitbucket.org/ivarun/ranluxcl .

CL²QCD Code Structure

- **Host program** of CL²QCD set up in C++ ⇒ allows for independent program parts using C++ functionalities and naturally provides extension capabilities.
- Cross-platform compilation provided using CMake framework[6].
- The code structure of CL²QCD is displayed in Figure 2. Two main components:
  - The physics package, representing high-level functionality;
  - The hardware package, representing low-level functionality.

  The meta package collects what is needed to control the program execution and IO operations.
- ***All parts*** of the simulation code are carried out using OpenCL kernels in double precision.
- **OpenCL kernels source files:**
  - Contain concrete implementations of basic LQCD functionality like matrix-matrix multiplication, but also more complex operations like the $\not{D}$ or the (R)HMC force calculation;
  - OpenCL language based on C99;
  - Compilation and execution is handled within the hardware package;
  - Kernels in a certain way detached from host part (host can continue independently of kernel execution status) ⇒ Clear separation into administrative part (host) and performance-critical calculations (kernels).

[6] See http://www.cmake.org .

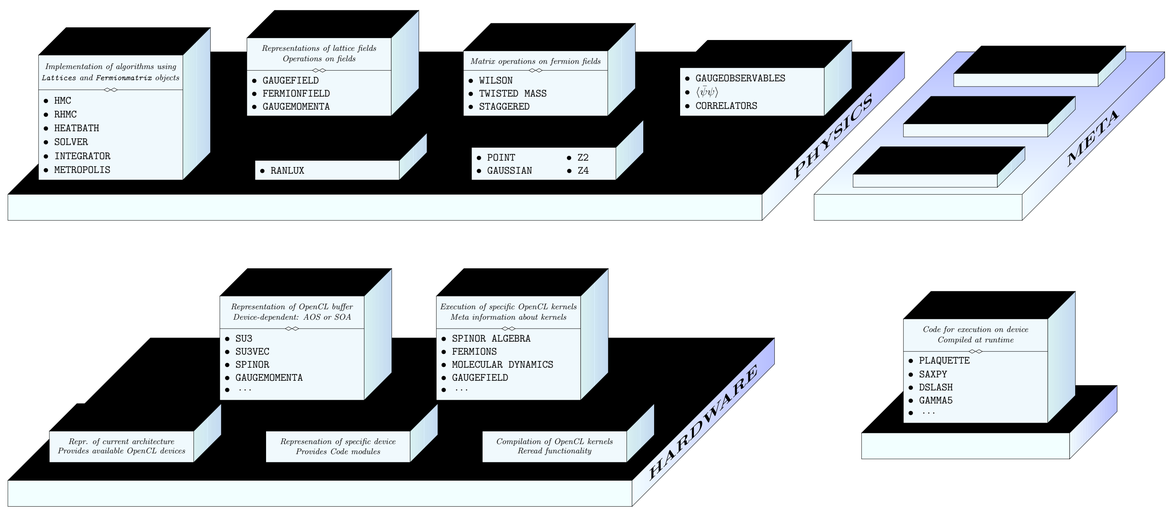

Figure 2: *CL²QCD code structure (illustrative). Packages and substructures are realized as namespaces.*

- The physics package provides representations of the physical objects like gauge fields or fermion fields. In addition, the corresponding classes provide functionality to operate on the respective field type. Moreover, algebraic operations like saxpy are provided. Similarly, the various fermion matrices are provided. This allows for the implementation of high-level functionality without knowing details of the underlying OpenCL structure. For example, the (R)HMC or the calculation of observables are completely independent of system or kernel specifics. In other words, the physics package works as an interface between algorithmic logic and the actual OpenCL implementation.
- In turn, the hardware package is destined to handle the compilation and execution of the OpenCL kernels. The hardware::System class represents the architecture available at runtime. The latter can provide multiple computing devices (i.e. CPUs and/or GPUs), which are represented by hardware::Device objects and initialized based on runtime parameters. Kernels are organized topic-wise within the hardware::code namespace, e.g. in the hardware::code::Fermions class. These classes take over the calling logic of the kernels and provide meta informations like the number of FLOPs a specific kernels performs. The hardware::Device class has each of the hardware::code classes as singleton objects, i.e. they are initialized the first time they are needed. During this process, the OpenCL kernels are compiled.
- Memory management is performed by the hardware::buffers classes, which also ensure that memory objects are treated in a *Structure of arrays (SOA)* fashion on GPUs, which on these is crucial for optimal memory access as opposed to *Array of structures (AOS)*.
- OpenCL kernels are compiled at runtime using the OpenCL compiler class. In OpenCL, this is mandatory as the specific architecture is not known a priori. On the one hand, this introduces an overhead, but on the other hand allows to pass runtime parameters (like the lattice size) as compile time parameters to the kernels, saving arguments and enabling compiler optimization for specific parameter sets. In addition, the compiled kernel code is saved for later reuse, e.g. when resuming an HMC chain with the same parameters on the same architecture. This reduces the initialization time. Kernel code is common to GPUs and CPUs, device specifics are incorporated using macros.