Project documentation:

HMC with OpenCL for hybrid manycore systems

Christopher Pinke Lars Zeidlewicz

February 2, 2011

Abstract

We want to write a complete HMC simulation programme with Wilson-type quarks (twisted mass action) using OpenCL for a hybrid structure consisting of CPUs and GPUs.

Contents

1 General thoughts – structure

This will be a lot of work...

Basically, the programme splits into two main parts: host-code and device-code. Host-code is standard C++ and provides the OpenCL-interface and master code for the OpenCL kernels. Furthermore, in the end, it is planned to have a completely working CPU-HMC, too. Since it is meant to run on a single node, the use of OpenMP is intended. The CPU-HMC might be needed to implement the hybrid approach in a later step.

The device code is the collection of all OpenCL-kernels. This should finally constitute a complete HMC. For now, we target at a heatbath implementation. The pure gauge heatbath will provide a good testing ground for a) benchmarks (and performance optimisation), b) the hybrid strategy. At this stage, we also want to learn what the optimal communication between host and device is, i. e. when do we need to transfer what information.

Data types for the C++ host and OpenCL device codes are not the same. Therefore, the C++ code also needs to provide transfer functions. For the gaugefield and SU(N_c) types this should be finished. For the spinor part, since its not needed yet, no OpenCL types have been defined so far.

On the host, there are plaquette and Polyakov functions that have been tested against existing configuration files. It is an open question whether we want to have those measurements on the device, too. The alternative would be to measure the gauge observables only when the gaugefield is transferred back to the host.

ILDG format configurations (as from Carsten Urbach's code [1]) can be read. We still need the according functions to write a config to a file.

2 Algorithm

In this section we describe the algorithm that is implemented in the programme. Ultimatively, we want to have an HMC with CG-solver (or maybe BiCGStab), even-odd preconditioning, leap-frog integration and possibly mass preconditioning **references_to_be_given_by_Lars**. That seems to be the standard choice of algorithm.

2.1 Heatbath

Right now only the heatbath algorithm is implemented. It corresponds to the standard description for SU(3) Gauge theories following Cabibbo & Marinari [2] and Creutz [3].

3 Options and I/O

3.1 Input-file

All informations on run-time options are passed to the programme in an input file. The name of that file is the only command line option to be given when calling the executable. This file contains one line per option/variable with the format
`OPTIONNAME =` *option*.
If one option name occurs several times, the last instance is used. It is possible to have comment lines starting with "#". The list of options and variables is given in table 1.

3.2 Configurations

Configuation files can be read from ILDG format. `host_readgauge.h` and `host_readgauge.cpp` provide a class `sourcefileparameters` which reads in all information from a given tmlqcd file concerning gauge-configurations. Not fully implemented is the read-in of fermion-propagators.

The reading-routine is based on a stand-alone-program that was written in C. The metainformations about the data are saved in XML- and XLF-format (see also appendix), while the actual data is saved binary. To read XLF and XLM files the LIME- (source) and XLM-libraries are used, which are only available for C. The routines are not optimized regarding speed or memory-usage since the reading of the data should only take place at the beginning of the programrun.

The reading of the metainforamtions was specifically tested for two tmlqcd-files created with different version of the hmc (5.1.1 and 5.1.5). Everything is read and saved into particular variables. If there are fermion-informations in the sourcefile, they are saved successively for each fermion. Here, only informations about the first are stored, all others are skipped. This has not been implemented because there is most likely no need to read in propagators. There were some problems in reading metainforamtions from files other than from the tmlqcd-files, revealing the no-optimized nature of the routine. So there might be the need to work in the specific layout of the file one wishes to read in.

The binary data is saved into a field which is a parameter of the routine (as a pointer). The precision in which the data was saved is extracted before during the metainformation gain. ILDG is always stored big endian, so the routine checks the local endianness with htons() to get the right order of the bytes.

In `host_writegaugefield.h` and `host_writegaugefield.cpp` functions to save Gaugefield-Configurations are contained. They are saved in the form `conf.XXXXX` with a five-digit number labeling the specific configuration. If a file of other name is read in, the program starts with `conf.00000`. Not

OPTIONNAME	description	possible values	default
kappa	hopping parameter κ	$0 < \kappa < \infty$	0.125
beta	lattice coupling β	$0 < \beta < \infty$	4
mu	twisted mass parameter μ	$0 < \mu < \infty$	0.006
cgmax	maximum number of CG iterations	$1 <$ cgmax $< \infty$	1000
prec	precision of gaugefield (from file)	32, 64	64
startcondition	start condition	hot, cold, continue	cold
thermalizationsteps	thermalisation steps	$0 \leq$ thermalizationsteps $< \infty$	0
heatbathsteps	heatbath steps	$0 <$ heatbathsteps $< \infty$	1000
sourcefile	name of gauge config file	string	not defined

Table 1: List of all possible options and variables that can be passed to the programme in the input file.

implemented so far is the ildg-checksum. Here, only a stubb is written into the limefile!!

3.3 Compiler options

3.3.1 RECONSTRUCT_TWELVE

If `RECONSTRUCT_TWELVE` is defined, SU(3) matrices are stored using 12 floating point numbers of type `hmc_float`. They represent the first two rows of the matrix. The last row is reconstructed [4]. The 12 numbers are stored using a single index $n = i + (N_c - 1)j$ for the components U_{ij}. The reconstruction is done in `operations.cpp`. You have

$$j = \text{int}\left(\frac{n}{N_c - 1}\right)$$

and

$$i = n - (N_c - 1)j \ .$$

Thus the numbering works as follows:

$$\begin{pmatrix} u_{00} & u_{01} & u_{02} \\ u_{10} & u_{11} & u_{12} \\ u_{20} & u_{21} & u_{22} \end{pmatrix} = \begin{pmatrix} u_0 & u_2 & u_4 \\ u_1 & u_3 & u_5 \\ c_0 & c_1 & c_2 \end{pmatrix}$$

The reconstruction of the elements c_{ncomp} of the matrix `in` is done with the function `reconstruct_su3(hmc_su3matrix* in, int ncomp)`.

3.3.2 SOURCEDIR

`SOURCEDIR` is passed to the OpenCL `clprogram` in order to find the OpenCL kernel files.

3.3.3 _OPENMP

`_OPENMP` can be used to activate OpenMP.

3.3.4 _USEDOUBLEPREC_

`_USEDOUBLEPREC_` switches to double precision floating point numbers on the OpenCL device.

3.3.5 internal switches

- `_INKERNEL_` allows to check whether OpenCL kernel code or C++ host code is meant to be compiled (some `globalheaders.h` and `types.h` are included by both of them).

3.4 Output

Despite the possibiliy to write a gauge-configuration to a file (see above), a time-measurement is saved by default into the file `time_measurement_0/1`, where 1 and 0 is used in the case of `RECONSTRUCT_TWELVE` or not, respectivley. Furthermore, in `host_gaugeobservables` there is a function to print out gaugeobservables into a file. This is done in the following order:

```
Iteration number
Plaquette
Plaquette in time-direction
Plaquette in space-direction
Real(Polyakovloop)
Im(Polyakovloop)
Abs(Polyakovloop).
```

Here, the observables are saved with a precision of 15.

4 Functions and constants

4.1 `globaldefs.h`

Definitions in `globaldefs.h` should be available programme-wide.

- `NC`=3
- `NSPIN`=4
- `NDIM`=4
- `NSPACE`, `NTIME`: The spatial and temporal extent is defined at compile time in order to have fixed for-loop lengths. These two definitions are passed to the OpenCL kernels with their compile-time values.
- `VOLSPACE` = `NPACE*NSPACE*NSPACE`
- `VOL4D` = `VOLSPACE*NTIME`
- `PI` = 3.14159265358979
- `su2_entries` = 4
- `START_FROM_SOURCE`=2, `COLD_START`=0, `HOT_START`=1

4.2 hmcerrs.h

hmcerrs.h defines the error codes as typedef int hmc_error:

- HMC_SUCCESS
- HMC_STDERR
- HMC_FILEERROR
- HMC_OCLERROR
- HMC_XMLERROR
- HMC_UNDEFINEDERROR

4.3 types.h

types.h contains typdefs. There are differences between host and kernel code. For both, the basic floating point type is hmc_float

```
#ifdef _USEDOUBLEPREC_
typedef double hmc_float;
#else
typedef float hmc_float;
#endif
```

With this type hmc_one_f=1 is a global constant. For both host and kernel, there is a complex type hmc_complex with members re and im for real and imaginary part that is based on hmc_float. hmc_complex_one, hmc_complex_zero, hmc_complex_i are available. All according complex operations are defined in host_operations.h.

4.3.1 Spinor types

Spinor types have only been defined in the C++ part so far:

```
typedef hmc_complex hmc_full_spinor [NSPIN*NC];
typedef hmc_complex hmc_full_spinor_field [NSPIN*NC][VOLSPACE][NTIME];
```

All according operations are defined in host_operations.h.

4.3.2 Gaugefield and matrix types

There is hmc_su3matrix which is used as fundamental data type for SU(3) matrices. Depending on the switch _RECONSRUCT_TWELVE_ it is an array of length NC*(NC-1) or a field with [NC][NC] components. hmc_staplematrix is an array of length NC*NC (for reconstruct twelve) or simply a new name for hmc_su3matrix if all 18 components are stored anyways. The gaugefield ist stored as

```
#ifdef _RECONSTRUCT_TWELVE_
typedef hmc_complex hmc_gaugefield [NC*(NC-1)][NDIM][VOLSPACE][NTIME];
#else
typedef hmc_complex hmc_gaugefield [NC][NC][NDIM][VOLSPACE][NTIME];
#endif
```

All according operations are defined in `host_operations.h`.

On device, the typedefs are as follows:

```
typedef hmc_complex hmc_ocl_su3matrix;
typedef hmc_complex hmc_ocl_staplematrix;
typedef hmc_float hmc_ocl_gaugefield;
```

Thus, the necessary array length has to be allocated each time. Operations are given in `opencl_operations.cl`.

For the output file, there is a type `ildg_gaugefield`, which is just a big array of `hmc_floats`. `hmc_gaugefield` can be converted into this format to ensure combatibility.

4.3.3 Random number types

For the random numbers on the device there is the type `hmc_ocl_ran`, which is just a `cl_uint4`. Additionally there is a type `rndarra`, which is an array of `hmc_ocl_ran` with `VOL4D/2` entries. It is meant to store one random number for each thread.

4.4 Operations

Local and global operations on the types defined in `types.h` are contained in `host_-/opencl_operations.h` and `host_-/opencl_operations.cpp` for the host and device, respectivly. There are operations on complex types (conjugation, multiplication,...), matrix types (adjoin, trace, determinant,...), spinor types, and gaugefield types.

Besides arithmetic operations there are gaugefield functions to initialise cold and hot start as well as start from source. Furthermore interfaces to get and put SU(3) matrix elements of a gaugefield are provided and functions that copy a gaugefield from hmc type to ocl type.

4.5 Gaugeobservables

`host_gaugeobservables.h` and `host_gaugeobservables.cpp` provide

```
void print_gaugeobservables(hmc_gaugefield* field, usetimer* timer,
                            usetimer * timer2);
void print_gaugeobservables(hmc_gaugefield* field, usetimer* timer,
                            usetimer * timer2, int iter);
```

```
void print_gaugeobservables(hmc_gaugefield* field, usetimer* timer,
                            usetimer * timer2, int iter, std::string file);
void print_gaugeobservables(hmc_float plaq, hmc_float tplaq,
                            hmc_float splaq, hmc_complex pol, int iter);
void print_gaugeobservables(hmc_float plaq, hmc_float tplaq,
                            hmc_float splaq, hmc_complex pol, int iter,
                            std::string file);

hmc_float plaquette(hmc_gaugefield * field, hmc_float* tplaq,
                            hmc_float* splaq);
hmc_float plaquette(hmc_gaugefield * field);
hmc_complex polyakov(hmc_gaugefield * field);
hmc_complex spatial_polyakov(hmc_gaugefield * field, int dir);
```

The print functions print plaquette, temporal and spatial plaquette, real part, imaginary part and absolute value of Polyakov loop to standard out or a file in two versions: One can either give already calculated results or do this within the printing funcktion. Time-Measurement can be done. In the future there should be additional overload-function to also be able to do everything without time-measurement.

`host_gaugefieldoperations.h` and `host_gaugefieldoperations.cpp` provide

```
void print_info_source(sourcefileparameters* params);

hmc_error init_gaugefield(hmc_gaugefield* gaugefield,
                            inputparameters* parameters, usetimer* timer);
```

The print function prints xml-info from an input source file to standard out. `init_gaugefield` initialises a (previously allocated) gaugefield according to the start condition. Note the use of timer which needs to be improved.

4.6 Usetimer

`host_usetimer.h` defines an ugly wrapper around a timer class defined in `host_timer.h`. We need to improve this...

4.7 Geometry

`host_geometry.h` and `host_geometry.cpp` provide a mapping from from (x, y, z) to one int nspace. The naming scheme is as follows:
direction 1 = x; direction 2 = y; direction 3 = z; direction 0 = t. This differs from the ILDG-format!
Important: (x, y, z) are always used together and it should be possible to use

a different number of space dimensions (defined in `globals.h`). t is always apart.

```
//switch between (x,y,z) <-> nspace=0,...,VOLSPACE-1
int get_nspace(int* coord);
int get_spacecoord(int nspace, int dir);

int get_neighbor(int nspace, int dir);
int get_lower_neighbor(int nspace, int dir);
```

However, there is an important difference between host and opencl-device: the gaugefield on the device is one big array of hmc_floats! That is why one has to define an ordering there. It is done according to

- spatial position = x + y*NSPACE + z*NSPACE*NSPACE
- site position = pos + VOLSPACE*t
- link-index = mu + NDIM*site

For example, to acces one specific element while being on the device can be done through the function

```
int inline ocl_gaugefield_element(int c, int a, int b, int mu,
                          int spacepos, int t){
#ifdef _RECONSTRUCT_TWELVE_
  return c + 2*a + 2*(NC-1)*b + 2*NC*(NC-1)*mu + 2*NC*(NC-1)*NDIM*spacepos +
                          2*NC*(NC-1)*NDIM*VOLSPACE*t;
#else
  return c + 2*a + 2*NC*b + 2*NC*NC*mu+2*NC*NC*NDIM*spacepos +
                          2*NC*NC*NDIM*VOLSPACE*t;
#endif
}
```

with c being the complex index. On the host, this would simply be

```
gaugefield[a][b][mu][spacepos][t].re(.im)
```

or

```
gaugefield[a + NC*b][mu][spacepos][t],
```

respectivly.

4.8 Update

In `host_update_heatbath.h` and `host_update_heatbath.cpp` the heatbath and overrelaxing algorithms for the gaugefield are given in standard and checkerboard version, where the latter can be used with OpenMP (and perhaps needs one final test). Additionally there is the routine to calculate the staple of a given link.

```
void calc_staple(hmc_gaugefield * field, hmc_staplematrix * dest,
                 const int pos, const int t, const int mu_in);
void heatbath_update (hmc_gaugefield * gaugefield,
                 const hmc_float beta);
void heatbath_overrelax (hmc_gaugefield * gaugefield,
                 const hmc_float beta);
void heatbath_update_checkerboard (hmc_gaugefield * gaugefield,
                 const hmc_float beta);
void heatbath_overrelax_checkerboard (hmc_gaugefield * gaugefield,
                 const hmc_float beta);
```

4.9 Random

The files `host_random.h` and `host_random.cpp` provide a LCG random number generator for the host taken from Numerical Recipes [5] called `Random` which can be initialised with a seed that is currently set to be 50000. For the heatbath one needs a random order of 1,2 and 3 as well as random SU(2)-matrices, which is implemented here according to Kennedy & Pendleton [6]. Also, there is the init-function for a random-array that is used as the seed-array for the random-numbers on the device.

```
void random_1_2_3 (int rand[3]);
void init_random_seeds(Random random, clu_taus_state * hmc_rndarray,
                 const int NUM, usetimer * inittime);
void SU2Update(hmc_float dst [su2_entries], const hmc_float alpha);
```

On the device there is the function `ocl_new_ran` that provides a LCG PRNG for each thread (thanks to Matthias Bach).

4.10 Opencl

`opencl.h` and `opencl.cpp` provide a class `opencl` which can be used to initialise an OpenCL device and call the heatbath kernel. The kernels needed for the OpenCL-Implementaton are give in `opencl_xxx.cl` files (e. g. `opencl_operations.cl`). There is a testing kernel which can be called with the according test member function of the OpenCL class. Also, equivalent functions to basically every function defined on the host (except input- and output-operations) can be found in these files.

The kernel files that should be read in to build the OpenCL `clprogram` are listed in the `string` vector `cl_kernels_file`. When the programme is built, the complete source code is also written to the file `cl_kernelsource.cl` which is meant to allow for debugging.

4.11 Testing

`host_testing.h` and `host_testing.cpp` provide a playground for testing. Just have a look...

5 Hybrid strategy

Some bright ideas about how to make use of the hybrid architecture...

6 Some readings

6.1 ILDG

Here one can find infos about ILDG: `http://ildg.sasr.edu.au/Plone`

6.2 LQCD using OpenCL

To get the OpenCL specifications visit `http://www.khronos.org/opencl`.

- With OpenGL: Egri et al., Lattice QCD as a video game [7] (classic)
- Demchik and Strelchenko, SU(2) Monte Carlo [8]
- Demchik, Random Numbers on GPUs with OpenCL [9]

6.3 LQCD using CUDA

To get general CUDA informations visit `http://www.nvidia.com/object/cuda_home_new.html`.

- Cardoso and Bicudo, Lattice SU(2) on GPUs [10]
- Clark et al., Solving Lattice QCD systems of equations using mixed precision solvers on GPUs [4]
- Clark, QCD on GPUs: cost effective supercomputing [11]
- Hayakawa et al., Improving many flavor QCD simulations using multiple GPUs [12]
- TWQCD Collaboration: TWQCD's dynamical DWF project [13]
- Bonati, Cossu, D'Elia, Di Giacomo, Staggered fermions simulations on GPUs [14]

- Kim and Lee, Multi GPU Performance of Conjugate Gradient Algorithm with Staggered Fermions [15]
- Walk, Wittig, Dranischnikow, Schomer,Implementation of the Neuberger-Dirac operator on GPUs [16]
- Karimi et al., High-Performance Physics Simulations Using Multi-Core CPUs and GPGPUs in a Volunteer Computing Context [17], A Performance Comparison of CUDA and OpenCL [18]
- Osaki and Ishikawa, Domain Decomposition on GPU: [19]

References

[1] K. Jansen and C. Urbach. tmLQCD: A Program suite to simulate Wilson Twisted mass Lattice QCD. *Comput.Phys.Commun.*, 180:2717–2738, 2009.

[2] Nicola Cabibbo and Enzo Marinari. A new Method for updating SU(N) Matrices in Computer Simulations of Gauge Theories. *Physical Letters, Volume 119B, number 4,5,6*, 1982.

[3] Michael Creutz. Monte Carlo Study of quantized SU(2) gauge theory. *Physical Review D, Volume 21, Number 8*, 1980.

[4] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi. Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. *Comput. Phys. Commun.*, 181:1517–1528, 2010.

[5] W. H. Teukolsky, S. A. Vetterling, W. T. Press, and B. P. Flannery. *Numerical Recipes. The Art of Scientific Computing.* Cambridge University Press, Cambridge, 3. auflage edition, 2007.

[6] A. D. Kennedy and B. J. Pendleton. Improved Heatbath Method for Monte Carlo Calculations in Lattice Gauge Theories. *Physical Letters, Volume 156B, number 5,6*, 1985.

[7] Gyozo I. Egri, Zoltan Fodor, Christian Hoelbling, Sandor D. Katz, Daniel Nogradi, et al. Lattice QCD as a video game. *Comput.Phys.Commun.*, 177:631–639, 2007.

[8] Vadim Demchik and Alexei Strelchenko. Monte Carlo simulations on Graphics Processing Units. 2009.

[9] Vadim" "Demchik. Pseudo-random number generators for monte carlo simulations on graphics processing units. 2010.

[10] Nuno Cardoso and Pedro Bicudo. Lattice SU(2) on GPU's. 2010. * Temporary entry *.

[11] M.A. Clark. QCD on GPUs: cost effective supercomputing. *PoS*, LAT2009:003, 2009.

[12] M. Hayakawa, K.-I. Ishikawa, Y. Osaki, S. Takeda, S. Uno, et al. Improving many flavor QCD simulations using multiple GPUs. 2010.

[13] Ting-Wai Chiu et al. TWQCD's dynamical DWF project. *PoS*, LAT2009:034, 2009.

[14] Claudio Bonati, Guido Cossu, Massimo D'Elia, and Adriano Di Giacomo. Staggered fermions simulations on GPUs. *PoS(Lat2010)???*, 2010.

[15] Hyung-Jin Kim and Weonjong Lee. Multi GPU Performance of Conjugate Gradient Algorithm with Staggered Fermions. *PoS*, LAT2010:028, 2010. * Temporary entry *.

[16] Bjoern Walk, Hartmut Wittig, Egor Dranischnikow, and Elmar Schomer. Implementation of the Neuberger-Dirac operator on GPUs. 2010. * Temporary entry *.

[17] "Kamran Karimi, Neil G. Dickson, and Firas Hamze". "high-performance physics simulations using multi-core cpus and gpgpus in a volunteer computing context". *CoRR*, abs/1004.0023, 2010.

[18] "Kamran Karimi, Neil G. Dickson, and Firas Hamze". A performance comparison of cuda and opencl. *CoRR*, abs/1005.2581, 2010.

[19] Yusuke Osaki and Ken-Ichi Ishikawa. Domain decomposition method on gpu. 2010.