Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Enabling CP2K Application for Exascale Computing with Accelerators using OpenACC and OpenCL

Mariusz Uchroński[a]*, Agnieszka Kwiecień[a], Marcin Gębarowski[a]

[a]*WCSS, Wroclaw University of Technology, Wyb. Wyspiańskiego 27, 50-370 Wrocław, Poland*

Abstract

CP2K is an application for atomistic and molecular simulation and, with its excellent scalability, is particularly important with regards to use on future exascale systems. The code is well parallelized using MPI and hybrid MPI/OpenMP, typically scaling well to ~1 core per atom in the system. The research on CP2K done within PRACE-1IP stated that due to heavy usage of sparse matrix multiplication for large systems, there is a place for improvement of performance. The main goal of this work, undertaken within PRACE-3IP, was to investigate the most time-consuming routines and port them to accelerators, particularly GPGPUs. The relevant areas of the code that can be effectively accelerated are the matrix multiplications (DBCSR library). A significant amount of work has already been done on DBCSR library using CUDA. We focused on enabling the library on a potentially wider range of computing resources using OpenCL and OpenACC technologies, to bring the overall application closer to exascale. We introduce the ports and promising performance results. The work done has led to the identification of a number of issues with using OpenACC in CP2K, which need to be further investigated and resolved to make the application and technology work better together.

Introduction

CP2K [1] is an open-source application designed for atomistic and molecular simulation of solid state, liquid, molecular and biological systems. CP2K proved to be a highly scalable code [2], what makes it a good candidate to use on the current petascale and future exascale systems. The code is written in FORTRAN 95, well parallelized with MPI and, in some parts, with hybrid MPI/OpenMP [15]. It is typically scaling well to ~1 core per atom in the system. CP2K supports the research of many science communities, as it is widely used for materials science, life sciences and computational chemistry. The code has been used by several projects on PRACE Research Infrastructure, including national Tier-1 resources, across the PRACE regular and DECI calls. It has been added to the Unified European Applications Benchmark Suite [3].

The main goal of this work, undertaken within PRACE-3IP, was to identify most promising routines in the CP2K code and port them to accelerators, particularly GPGPUs. The research on CP2K done within PRACE-1IP [2] stated that due to heavy usage of sparse matrix multiplication for large systems, there is still a place for improvement of the performance. Thus, the relevant areas of the code that could be effectively accelerated are the matrix multiplications in DBCSR library [16]. Based on previous suggestions and our analysis we decided to focus on the DBCSR library. A significant amount of work has already been done on the library using CUDA. The CUDA implementation of matrix multiplication in DBCSR has been refactored lately, and resulting improvements are included in the release of CP2K v2.5. Our aim was to enable it on a potentially wider range of computing resources, using OpenCL and OpenACC technologies. Both technologies have been identified by PRACE as important for exascale computing [4], and it is of interest to evaluate the possibility of using them to bring the application closer to exascale.

Within this task we have worked on CP2K v2.4. At the time of writing of this paper v2.5 is available for download.

Technology used

OpenACC [9] is a directive-based open standard supported by NVIDIA, PGI, Cray and CAPS, designed to simplify parallel programming of heterogeneous CPU/GPU systems [4]. The developer can annotate C, C++ and FORTRAN source code to identify the areas to be accelerated using `#pragma` compiler directives and additional functions. It is portable across operating systems, and multi-core processors such as NVIDIA and AMD GPUs, and Intel Xeon Phi [4], but the range of target options depends on the compiler used. The latest version of the standard, OpenACC 2.0a [5], was announced on August 31, 2013.

OpenCL [10] is an open standard for parallel programming of heterogeneous computing systems. It provides an API and a standard language to write portable code for multi-core CPUs, GPUs, APUs and other architectures, including latest Intel Xeon Phi accelerators. OpenCL kernels are written in a subset of the ISO C99 language that is compiled at runtime to target a particular computing device. An interesting feature of this technology is that the code prepared for Intel Xeon processors is also runnable on the Intel Xeon Phi with minimal changes [4], but it might result in sub-optimal performance. The latest version of the standard, OpenCL v2.0 [11], was announced in 18 March, 2014.

Hardware used

For development and testing we used the Supernova system, located at Wrocław Centre for Networking and Supercomputing (WCSS). The system was used to obtain most of the results presented in this paper.

Supernova serves as a Tier-1 machine within PRACE infrastructure, and is a cluster running the Scientific Linux operating system. The cluster is equipped with 404 thin nodes comprising two six-core Intel Xeon X5650 processors running at 2.67 GHz with 32 GB of memory per node, and 3 fat nodes with four sixteen-core AMD Opteron 6274 processors with 256 GB of memory and two NVIDIA Tesla M2075 (448 cores, 6 GB of memory) per node. For tests and development we used also two additional nodes: one with two NVIDIA GTX 480 and second with two AMD Radeon HD 6950 GPUs. The nodes have the GNU compilers, Portland Group's PGI Accelerator compilers and NVIDIA CUDA 5.0 (where necessary) installed.

Some part of the work was continued on the Zeus system, located at ACK Cyfronet AGH in Kraków, Poland. The system also serves as PRACE Tier-1 machine and has the PGI compilers installed. The GPGPU part of the cluster consists of nodes comprising Intel Xeon X5670 or E5645 processors and NVIDIA Tesla M2050 or M2090 GPUs. It is running the Scientific Linux 5 operating system.

For testing, the Fionn system located at Irish Centre for High-End Computing (ICHEC), has also been used. The system consists of several partitions, and the hybrid one contains 32 nodes, with two ten-core 2.2 GHz Intel Ivy Bridge processors and 64 GB of memory each. 16 nodes of this partition have 32 Intel Xeon Phi 5110P accelerators while the other 16 have 32 NVIDIA K20X's.

Porting and testing

OpenACC port

The work undertaken within this task started from a compilation of the CP2K v2.4 source code (release date: 2013-06-19) on Supernova using `gfortran` from the GNU compiler suite (v4.8), as it is known the code work well with it. The compilation of SOPT (pure serial) and POPT (pure MPI, with OpenMPI support) versions ended without any difficulty and the application tests from a test suite have been successfully run.

OpenACC directives are supported by only a few compilers, and the mostly used are Portland Group's PGI Accelerator compilers [13]. We decided to use PGI suite in our further work. Other compilers with OpenACC support, like CAPS [8] and Cray have not been initially considered, due to lack of access to the license or appropriate hardware. As a first step we planned to compile the existing source code and then introduce the OpenACC port, run tests and compare the performance. For debugging purposes we have used Allinea DDT [12] and PGDBG [14].

A number of different issues were identified when building CP2K with the `pgfortran` from the PGI compiler suite, including non-implemented FORTRAN functions and a segmentation fault. We mainly used PGI version 13.5, but next releases 14.1 and 14.3 were also tested, with no improvement in the problematic areas. Slow compilation when using PGI has been a disadvantage, limiting to some extent possibilities of exploring different compilation options and target architectures in a given time frame. The issues encountered and the proposed work-around are summarised below:

1. The compilation stopped with an error indicating that Error Function ERFC(X) from the FORTRAN 2008 standard [6] is not supported by the compiler. We investigated the PGI support for FORTRAN standards and while FORTRAN 2003 is officially stated as fully supported, the coverage for FORTRAN 2008 specific features is still low, resulting in the mentioned error. At first we replaced the function called `erfc` with a direct implementation done by Takuya Ooura from Research Institute for Mathematical Sciences at Kyoto University, Japan [7]. This work-around solved the problem resulting in a successful compilation, but was treated only as a temporary solution. Then we've found a direct implementation of the `erfc` function provided in CP2K code in `erf_fn` module (used several times in different parts of the code). Thus we've replaced the original FORTRAN 2008 function call with the one from the CP2K module. It required including the module in a `hfx_types.F` file. This will be no longer an issue as soon as PGI fully supports FORTRAN 2008 extensions. The newest release available to date (14.3) implements some further features, so there is progress in this direction.

2. Application ended with a segmentation fault error on all the tests from the test suite. An analysis reviled that the error is propagated in matrix multiplication nested operations (used in several files e.g. `atom_operators.F` and `atom_utils.F`). The erroneous line of code in `atom_operators.F` file is as follows:

   ```
   INT(1:n,1:n) = INT(1:n,1:n) +
   MATMUL(TRANSPOSE(cm(1:m,1:n)),MATMUL(omat(1:m,1:m),cm(1:m,1:n)))
   ```

   We have implemented a simple program performing the above calculations and it compiles and runs without errors, even for big matrices, so we suppose the problem is more likely linked to the application context in connection to the compiler. Such nested matrix operations may generate a big number of temporary data stored on the stack, and using the `ulimit -s unlimited` option might prevent the segmentation fault, but the unlimited stack size is already a default value on Supernova.

   We have compiled the program with the debug `-g` flag and run it within the Allinea DDT debugger [12] (v3.2.1) with the simple test `cp2k/tests/QS/C.inp`. The above error was duplicated and DDT reported:

   ```
   STOPED in SUBROUTINE contract2add ( int, omat, cm, error ) in a file:
   #1 contract2add () at /home/gensiub/cp2k-2.4.0/obj/Linux-x86-64-
   pgi/sopt/atom_operators.f90:1031 (at 0x000000000057beed)
   ```

   Setting a breakpoint at `atom_operators.f90:1031` and stepping into the operation resulted in displaying of a notice:

   ```
   Thread 1 stopped in pgf90_mmul_real8 with signal SIGSEGV (Segmentation fault).
   Reason/Origin: address not mapped to object (attempt to access invalid address)
   ```

   The memory debugging in DDT reported issues with the memory management on an earlier phase, in the function `create_cp2k_input_reading` (`input_cp2k.f90:133`), giving an error in `pgf90_ptr_alloc04` called by the `keyword_create` function (`input_keyword_types.f90:211`), and the DDT notice:

   ```
   Memory error detected in __memset_sse2 from /lib64/libc.so.6:
   read/write before start of allocation
   ```

   An analysis with the PGDBG (v14.3 on Zeus@Cyfronet) has given similar messages:

   ```
   Signalled SIGSEGV at 0x1EE1EB7, function pgf90_mmul_real8
    0x1EE1EB7:  F2 41 F 10 1F              movsd  (%r15),%xmm3
   ```

   The stack trace started and ended with memory errors:

   ```
   #16  cp2k line: "cp2k.f90"@331 address: 0x407688
   #15  run_input line: "cp2k_runs.f90"@1191 address: 0x6A94C3
        input_file_path = 0x7FFFECE58370, output_file_path = 0x7FFFECE58770,
        ierr = 0, ERROR: Cannot read value at address 0x0.
   (…)
   #1  err_matrix line: "atom_utils.f90"@1507 address: 0x56F549
       emat = 0xA3A8EC0, demax = 4.9406564584124654e-324, kmat = 0xA3A8CE0,
       pmat = 0xA39A4A0, umat = 0xA38CFD0, upmat = 0x60A0F50, nval = 0xA38B444,
       nbs = 0xA38FDF4, error = 0x7FFFECE54AA0,       = <unavailable>
   #0  pgf90_mmul_real8 address: 0x1EE1EB7
   ```

   We suspect some initialization errors, but to find out the possible reason, more investigation on the source code is needed, if further time allows.

Due to the issues mentioned above we decided to do some of the porting in parallel, expecting to find solutions of the problems at some point. We started with the source code analysis of the DBCSR library in order to introduce the OpenACC directives into the code. Because the segmentation fault error was not resolved in a reasonable timeframe, we needed to narrow our work. To check if using OpenACC in the library would give any performance improvement we ported an example `dbcsr_example_3.F` delivered with the application source code.

We have established that OpenACC can be introduced in the file `dbcsr_mm_stack_d.F` in the method `internal_mm_d_nn` which performs matrix multiplication:

```
PURE SUBROUTINE internal_mm_d_nn(&
               M,N,K,A,B,C)
  INTEGER, INTENT(IN)                :: M, N, K
  REAL(kind=real_8), INTENT(INOUT)   :: C(M,N)
  REAL(kind=real_8), INTENT(IN)      :: B(K,N)
  REAL(kind=real_8), INTENT(IN)      :: A(M,K)
  C(:,:) = C(:,:) + MATMUL (A, B)
END SUBROUTINE internal_mm_d_nn
```

The new method `internal_mm_d_nn_acc` has been created with OpenACC directives, and called in the `dbcsr_mm_stack_d.F` file instead of the original one. In order to use this method of matrix multiplication in CP2K some changes to the file `dbcs_config.F` had to be made. To enable the OpenACC port we have changed a value of the variables named `mm_driver` and `mm_host_driver` from `mm_driver_smm` to `mm_driver_matmul`.

```
PURE SUBROUTINE internal_mm_d_nn_acc(&
               M,N,K,A,B,C)
  INTEGER, INTENT(IN)                :: M, N, K
  INTEGER                            :: i, j, l
  REAL(kind=real_8), INTENT(INOUT)   :: C(M,N)
  REAL(kind=real_8), INTENT(IN)      :: A(M,K)
  REAL(kind=real_8), INTENT(IN)      :: B(K,N)
!$acc data copyin(A, B) copy(C)
!$acc kernels loop
  DO i = 1, M
     DO j = 1, N
        DO l = 1, K
           C(i, j) = C(i, j) + A(i, l) * B(l, j)
        ENDDO
     ENDDO
  ENDDO
!$acc end data
END SUBROUTINE internal_mm_d_nn_acc
```

All changes and tests were conducted on CP2K version 2.4. The code has been built in a pure serial version referred to as SOPT. For comparison reasons, the code was built with two compilers, GNU `gfortran` and PGI `pgf90`. The target architectures for PGI were: Intel Xeon CPU, AMD Opteron, NVIDIA GTX, NVIDIA Tesla, and AMD Radeon. We have used a standard set of `gfortran` options, as delivered with CP2K, with only the libraries locations (`LIBS`) modified. It is possible that other compiler options (e.g. `-O2`) could give better performance; this would need further investigation.

`Gfortran` options (`arch/Linux-x86-64-gfortran.sopt`):

```
CC       = cc
CPP      =
FC       = gfortran
LD       = gfortran
AR       = ar -r
CPPFLAGS =
DFLAGS   = -D__GFORTRAN -D__FFTSG -D__FFTW3
FCFLAGS  = -O1 -fstrict-aliasing -fbacktrace -g -fbounds-check -ffree-form $(DFLAGS)
LDFLAGS  = $(FCFLAGS)
LIBS     = -lstdc++ -lfftw3 /usr/lib/libblas/libblas.a -llapack
OBJECTS_ARCHITECTURE = machine_gfortran.o
```

`Pgf90` options for Supernova nodes with AMD CPUs and NVIDIA Tesla GPUs (`arch/Linux-x86-64-pgi-acc.sopt`):

```
CC       = pgcc
CPP      = cpp
FC       = pgf90 -Mfree
LD       = pgf90
AR       = ar -r
DFLAGS   = -D__PGI -D__FFTSG -D__FFTW3
CPPFLAGS = -traditional -C $(DFLAGS) -P
FCFLAGS  = -fastsse -acc -tp amd64
LDFLAGS  = $(FCFLAGS)
LIBS     = -llapack -lacml -lfftw3
OBJECTS_ARCHITECTURE = machine_pgi.o
```

The `-acc` option enables the OpenACC directives in the compiler. PGI compilers target the accelerator regions for the NVIDIA GPU, by default. The compilers provide the PGI Unified Binary technology, which outputs code for different target architectures. To enable it, the code should be built with the *target accelerator* option, e.g. `-ta=nvidia,host`. This option generates different versions of accelerated functions, and it is decided at runtime which one is used, depending on the hardware available. By default, the GPU version is used, if the GPU is available.

Performance results, discussed below, gathered for the DBCSR Example 3, illustrate the potential of OpenACC technology in CP2K, if possible to use with the whole application.

The `-Minfo` flag, if given to the PGI compilers, provides feedback on optimizations made by the compiler. We have observed that during compilation with `-fastsse` flag almost every loop in the program was optimized, giving with this automatic approach quite good performance. In the following section we show the performance results of different versions of the example code.

OpenCL port

Main goal of OpenCL work related to CP2K was porting DBCSR library to OpenCL. The same parts of the application identified as worth porting to OpenACC were considered candidates for OpenCL. The development work was divided into following stages:

- create C – Fortran interfaces in order to allow OpenCL code execution,
- device initialization,
- memory management – copy input to the device and then copy back results to the host,
- prepare OpenCL kernel and execute calculation on the device.

Development and testing was performed on AMD Radeon HD 7660G GPU with AMD Catalyst Driver 13.1 and OpenCL 1.2 AMD-APP 2.8, and on Supernova nodes with NVIDIA GTX 480. The code was compiled with the GNU compiler 4.6.3.

In order to call OpenCL DBCSR code from the FORTRAN interface the function `opencl_multiply` has been developed (fig. 1). The C function `dc_multiply_ocl` (bind with `ocl_multiply`), contains the code responsible for copying input data from a host to a device, executing calculations on the device and copying the results back to the host. The `opencl_multiply` function is called by the `dbcsr_opencl_multiply` subroutine (fig. 2).

```
#if defined (__DBCSR_OPENCL)
  INTERFACE
    FUNCTION opencl_multiply (params, stack_size, a_data, b_data, c_data,&
                              k, m, n)&
      RESULT (istat) BIND(C, name="dc_multiply_ocl")
      USE ISO_C_BINDING
      TYPE(C_PTR), INTENT(IN), VALUE           :: params
      INTEGER(KIND=C_INT), INTENT(IN), VALUE   :: stack_size
      TYPE(C_PTR), INTENT(IN), VALUE           :: a_data, b_data
      TYPE(C_PTR), VALUE                       :: c_data
      INTEGER(KIND=C_INT), INTENT(IN), VALUE   :: n, m, k
      INTEGER(KIND=C_INT)                      :: istat

    END FUNCTION opencl_multiply
  END INTERFACE
#endif
```

Figure 1. Fortran - C interface for OpenCL

DBCSR creates a stack with many small sparse matrixes which need to be processed to solve the large sparse matrix multiplication. In order to correctly multiply these small sparse matrixes the stack parameter is created.

This parameter contains information about input data location and location for storing multiplication results. A subroutine for processing DBCSR stack with OpenCL is shown in fig. 3.

```
SUBROUTINE dbcsr_opencl_multiply (params, stack_size, a_data, b_data,&
                                  c_data, k, m, n, error)
  TYPE(dbcsr_error_type), INTENT(INOUT)                :: error
  INTEGER, INTENT(IN)                                  :: stack_size
  INTEGER(KIND=C_INT), INTENT(IN), VALUE               :: n, m, k
  INTEGER, DIMENSION(dbcsr_ps_width,1:stack_size),&
           INTENT(IN), TARGET                          :: params
  REAL(kind=real_8), DIMENSION(*), INTENT(IN), TARGET  :: a_data, b_data
  REAL(kind=real_8), DIMENSION(*), TARGET              :: c_data

  CHARACTER(len=*), PARAMETER :: routineN = 'dbcsr_opencl_multiply', &
    routineP = moduleN//':'//routineN
  INTEGER                                    :: error_handle, istat

  CALL dbcsr_error_set (routineN, error_handle, error)
#if defined (__DBCSR_OPENCL)
  istat = opencl_multiply (C_LOC(params(1,1)), stack_size,&
                           C_LOC(a_data(1)), C_LOC(b_data(1)),&
                           C_LOC(c_data(1)), k, m ,n)
#else
  istat = -1
#endif
  IF (istat /= 0) THEN
       CALL dbcsr_assert (istat, "EQ", 0,&
            dbcsr_fatal_level, dbcsr_internal_error, routineN,&
            "Error during OpenCL dbcsr multiply.",&
            __LINE__, error=error)
  ENDIF
  CALL dbcsr_error_stop (error_handle, error)
END SUBROUTINE dbcsr_opencl_multiply
```

Figure 2. Subroutine for executing C/OpenCL code

```
SUBROUTINE opencl_process_mm_stack_d(params, stack_size,&
                                     a_data, b_data, c_data, error)
  INTEGER, INTENT(IN)                                  :: stack_size
  INTEGER, DIMENSION(dbcsr_ps_width,1:stack_size),&
           INTENT(IN)                                  :: params
  REAL(kind=real_8), DIMENSION(*), INTENT(IN)          :: a_data,&
                                                          b_data
  REAL(kind=real_8), DIMENSION(*), INTENT(INOUT)       :: c_data
  TYPE(dbcsr_error_type), INTENT(inout)                :: error

  CHARACTER(len=*), PARAMETER :: routineN = 'opencl_process_mm_stack_d',&
                                 routineP = moduleN//':'//routineN
  CALL dbcsr_opencl_multiply (params, stack_size, a_data, b_data, c_data,&
                              params(p_k,1), params(p_m,1), params(p_n,1),&
                              error=error)

END SUBROUTINE opencl_process_mm_stack_d
```

Figure 3. Subroutine for processing DBCSR stack with OpenCL

Fig. 4 shows the C function `dc_multiply_ocl` responsible for copying input data from a host to a device, executing calculations on the device and copying the results back to the host.

The DBCSR OpenCL kernel `dbcsr_kernel` code is illustrated on fig. 5.

In the OpenCL implementation for each small matrix multiplication (C=AxB) a thread block is created. Every element of the C matrix is computed by a single OpenCL thread. Threads from different thread block may update the same element of the C matrix in parallel. This situation can cause incorrect values in the C matrix. In order to prevent this issue a lock for the C matrix is required. The locking was implemented with OpenCL atomic function `atomic_cmpxchg` and global `c_locks` vector.

```
int dc_multiply_ocl(int *params, int stack_size,
                    double *a_data, double *b_data, double *c_data,
                    int k, int m, int n)
{

    // ...
    a_dev_data = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                sizeof(double)*a_data_size, NULL, NULL);
    b_dev_data = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                sizeof(double)*b_data_size, NULL, NULL);
    c_dev_data = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                sizeof(double)*c_data_size, NULL, NULL);
    c_dev_locks = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                 sizeof(int)*stack_size, NULL, NULL);
    dev_params = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                sizeof(int)*params_size, NULL, NULL);
    // ...
    clEnqueueWriteBuffer(queue, a_dev_data, CL_TRUE, 0,
                         sizeof(double)*a_data_size, a_data, 0, NULL, NULL);
    clEnqueueWriteBuffer(queue, b_dev_data, CL_TRUE, 0,
                         sizeof(double)*b_data_size,b_data, 0, NULL, NULL);
    clEnqueueWriteBuffer(queue, c_dev_locks, CL_TRUE, 0,
                         sizeof(int)*stack_size, c_locks, 0, NULL, NULL);
    clEnqueueWriteBuffer(queue, dev_params, CL_TRUE, 0,
                         sizeof(int)*params_size, params, 0, NULL, NULL);
    // ...
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_params);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &a_dev_data);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &b_dev_data);
    clSetKernelArg(kernel, 3, sizeof(cl_mem), &c_dev_data);
    clSetKernelArg(kernel, 4, sizeof(cl_mem), &c_dev_locks);
    // ...
    clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_threads,
                           &local_threads, 0, NULL, NULL);
    clEnqueueReadBuffer(queue, c_dev_data, CL_TRUE, 0,
                        sizeof(double)*c_data_size, c_data, 0, NULL, NULL);
}
```

Figure 4. OpenCL memory transfer and kernel execution.

```
#pragma OPENCL EXTENSION cl_amd_fp64 : enable
__kernel void dbcsr_kernel(__global int *params, __global double *a_data,
                           __global double *b_data, __global double *c_data,
                           __global int *c_locks)
{
  size_t sp, tn, psp;
  int m, n, k, mn, r, c, l;
  int c_id, sp_one, lock_owner;
  size_t a_begin, b_begin, c_begin;
  double cval;
  sp = get_group_id(0); tn = get_local_id(0);
  psp = 7 * sp;
  m = params[psp]; n = params[psp + 1]; k = params[psp + 2]; mn = m*n;
  a_begin = params[psp + 3] - 1; b_begin = params[psp + 4] - 1;
  c_begin = params[psp + 5] - 1;
  c_id = params[psp + 6] - 1;
  if (tn < mn)
  {
    r = tn % m; c = tn / m;
    cval = 0.0;
    for (l = 0; l < k; l++)
      cval = cval + a_data[a_begin + (l * m + r)] *
                    b_data[b_begin + (c * k + l)];
  }
  barrier(CLK_GLOBAL_MEM_FENCE);
  if(tn == 0)
  {
    sp_one = sp + 1;
    lock_owner = 0;
    while(lock_owner != sp_one)
      lock_owner = atomic_cmpxchg((volatile __global int *)&c_locks[c_id],
                                  0, sp_one);
  }
  if(tn < mn)
    c_data[c_begin + tn] += cval;
  barrier(CLK_GLOBAL_MEM_FENCE);
  if(tn == 0)
    c_locks[c_id] = 0;
}
```

Figure 5. DBCSR OpenCL kernel code

Performance results

OpenACC results

As stated before we have been unable to run CP2K compiled with PGI without a segmentation fault. Thus we used the example number 3 from the DBCSR library. The example has been compiled in several SOPT versions using `gfortran` and `pgf90`, and different `MM_DRIVER` keywords (select which routines to use for matrix block multiplications) for a comparison. The versions are as follows:

- gfortran internal – compiled with a driver pointing to the internal method of matrix multiplication (FORTRAN MATMUL());
- gfortran smm – a version with default settings, with matrix multiplication handled by the SMM library optimised for Small Matrix Multiplies (requires the SMM library at link time);
- pgi internal – compiled with PGI, without OpenACC support, matrix multiplication handled by the internal method;
- pgi smm – compiled with PGI, no OpenACC support, default settings with SMM library;
- openacc – compiled with PGI and OpenACC support.

Test runs were conducted on multiple problem sizes. We have changed the `nblkrows_total` parameter of input matrices, while a size of small block matrix remained constant: `rbs_size x cbs_size = 100 x 100`. We also modified the `mm_stack_size` parameter (in `dbcsr_config.F` file). We have measured the time spent by the program on preparing and computing all the small matrices.

Fig. 6 shows the results obtained for different number of block matrices (from 10k to 1000k blocks, and for additional 4000k blocks, with the constant block size set to 100x100, giving 10k elements in every block, and `mm_stack_size=1000`) with each compiled version of the code. Presented times are mean values gathered from 10 calls to `dbcsr_multiply`. We increased the problem size by changing the dimention of the sparse matrix – adding 100 blocks to each dimention in one step. The computation time increases linear along with the problem size for all five code versions. To check this linear trend we run the test for a bigger problem size: 4000k block matrices (2000x2000). The result obtained is aproximately two times bigger than for 1000k blocks (1000x1000), what confirms the trend.

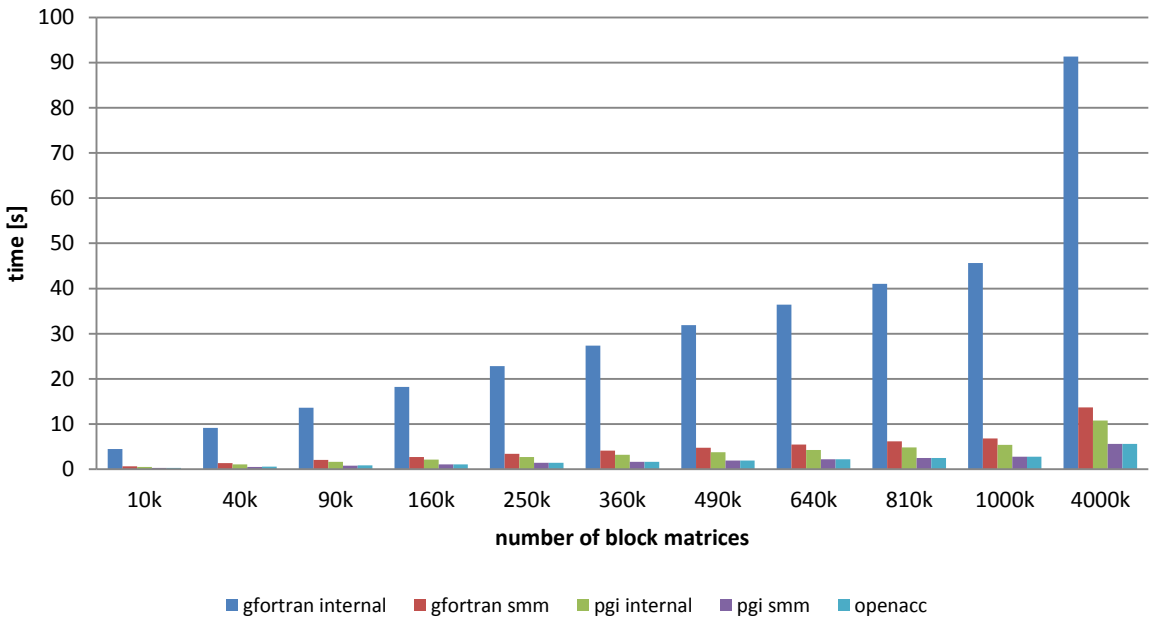

Figure 6. Tests results for different problem sizes

Figures 7 and 8 present the speedup of the particular version of the code for 250k and 1000k of block matrices respectively. The computation times of the non-optimized 'gfortran internal' version of the code have been used as a basis for the speedup calculation. The results show that PGI compilers provide very good automatic code omptimizations, which lead to reducing the computation time. The code compiled with `gfortran` and the SMM library for performing matrix operations, runs slower than the code compiled with PGI without the specialized

library. It must be mentioned, that using other compilation options for `gfortran` might indicate some performance improvements, but it requires further investigation. Using OpenACC gives very good results, although, the code compiled with PGI and SMM library shows similar performance. Such results for OpenACC suggest that the GPU is not fully utilized, and we suspected some of the library parameters may limit the performance.

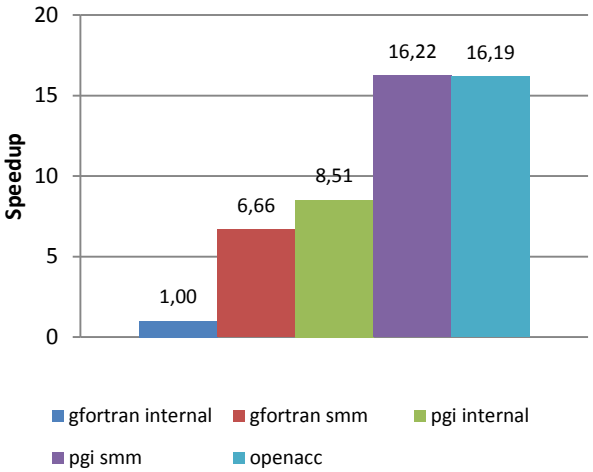

Figure 7. Speedup for problem size 250k blocks

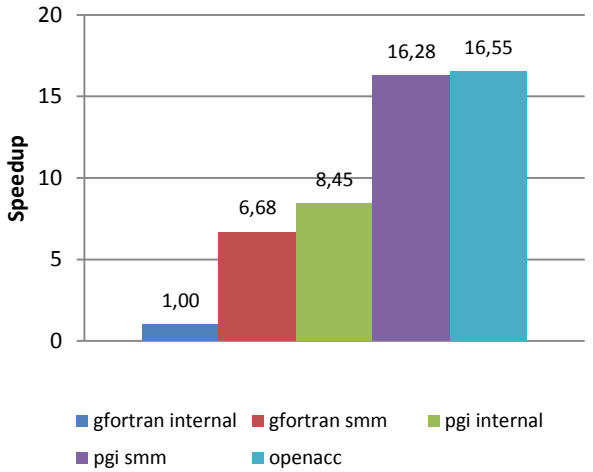

Figure 8. Speedup for problem size 1000k blocks

We looked closer on the data used in the computations and we have found that the multiplied stacks always had the maximum size of 1000, regardless from other settings. This caused an increased total number of function calls and data transfers to and from GPU for bigger problems. To get the most of GPU computations the size of the data transferred to an accelerator in one call should be maximized. The size of the stacks can be configured in the `dbcsr_config.F` file and we created two more versions of the code, with the maximum stack size set to 2048 and 10000 (noted as 10k), both with OpenACC support.

Figures 9 and 10 show the results for problem sizes 4000k and 25000k and different maximum stack sizes. When the problem size exceeded 4000k blocks the computation time increased drastically, and the speedup of all the code versions become significantly smaller, as it is shown for 25000k blocks. We have noticed that for each problem size there is no big difference between results gathered for OpenACC code variants for the three stack sizes. This suggests that the GPU is still not fully utilized, and the reason is more likely connected to the placement of the OpenACC pragmas.

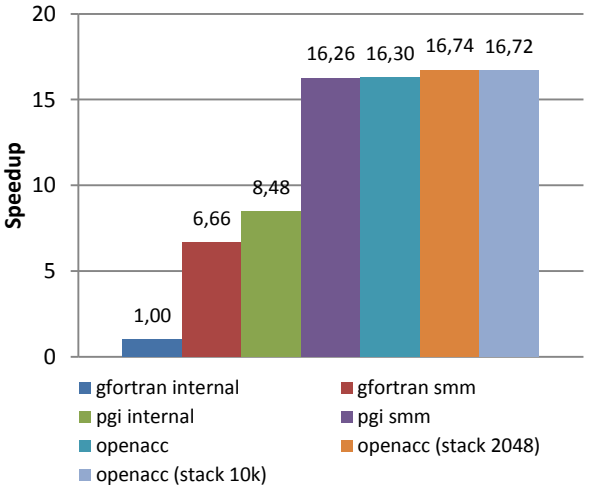

Figure 9. Speedup for problem size 4000k blocks

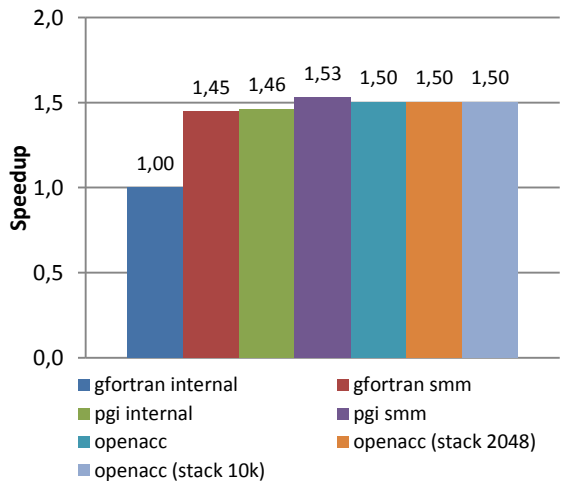

Figure 10. Speedup for problem size 25000k

After further analysis of the FORTRAN implementation we determined that we cannot gain a better performance without changing the FORTRAN code. It would require creating new methods for preparing bigger portions of data to be send to accelerators or modifying the multiplication method to be better adapted to the specific type of

data the program operates on. It could be realized by coping the data to the GPU before the multiplication, rather than right at the start of the OpenACC kernel. We have started to investigate this approach, and will continue if time allows.

OpenCL results

Similar as for OpenACC implementation example number 3 from the DBCSR library has been used for OpenCL performance tests. The tested problem sizes were also similar as for OpenACC. The ports have been developed and tested in parallel, but some of the findings could be reused, as both refer to the DBCSR library and consider accelerators as the target architecture. With the usage of OpenCL there is also a possibility to target the CPU, but it was not the main focus of this study.

The first set of test results for the OpenCL port is shown in the table 1. The testing was performed on a AMD Radeon HD 7660G GPU with the AMD Catalyst Driver 13.1 and OpenCL 1.2 AMD-APP 2.8. The size of the block matrices was set to 10x10 to initially evaluate the methods on a small problem size. The results show that OpenCL performed worse for bigger problems.

nblocks	OpenCL [s]	MATMUL [s]	SMM [s]
10k	0.006	0.034	0.007
40k	0.013	0.012	0.012
90k	0.021	0.017	0.018
160k	0.027	0.025	0.025
250k	0.035	0.032	0.032
360k	0.056	0.039	0.037
490k	0.059	0.046	0.044
640k	0.065	0.050	0.051
810k	0.071	0.059	0.056
1000k	0.079	0.067	0.065
4000k	0.171	0.136	0.127
25000k	0.441	0.338	0.310
100000k	1.473	0.657	0.608

Table 1. Test for the OpenCL, MATMUL and SMM methods (10x10 block size)

For better comparison with OpenACC additional tests of the OpenCL port have been run. The results are shown on fig. 11 and fig. 12. Tests were performed on a Supernova node with a NVIDIA GTX 480 GPU. Figure 1111 shows results for different number of block matrices (from 10k to 1000k blocks, with the constant block size set to 100x100, giving 10k elements in every block, and `mm_stack_size=1000`). OpenCL computation times are shorter than for the `gfortran` code. For problem sizes 10k, 40k, 90k and 160k the OpenCL implementation is faster than the OpenACC code and SMM code compiled with PGI.

It can be seen that for larger problem sizes the OpenCL code runs slower than SMM PGI and OpenACC. Following the analysis of OpenACC results we determined that a reason for this is dividing the data into smaller portions according to the `mm_stack_size` parameter. It resulted in many sequential executions of the OpenCL kernel on the GPU device. In this case the computational data is copied to/from GPU before and after each OpenCL kernel execution. The data transfer between CPU and GPU is a well known performance bottleneck, so we looked again into the algorithm parameters to find a better solution and avoid additional transfers.

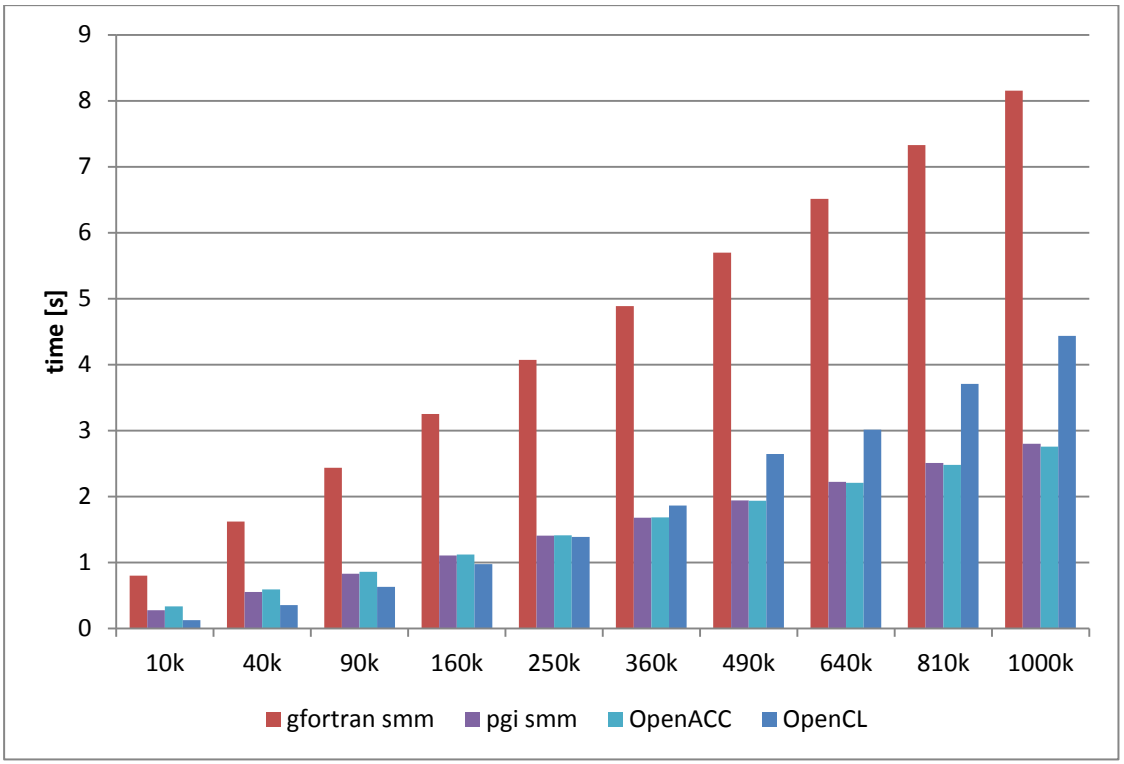

Figure 11. Test results for `mm_stack_size = 1000`

In order to avoid executing the OpenCL kernel multiple times the value of the `mm_stack_size` parameter has been increased. It results in less sequential OpenCL executions. In this case an impact of the memory transfer on the overall execution time has been significantly decreased. Results for `mm_stack_size = 10000` are shown on fig. 12. The OpenCL execution time is shorter than other implementations for all problem sizes. Changing the `mm_stack_size` parameter for implementations other than OpenCL has no impact on the execution time.

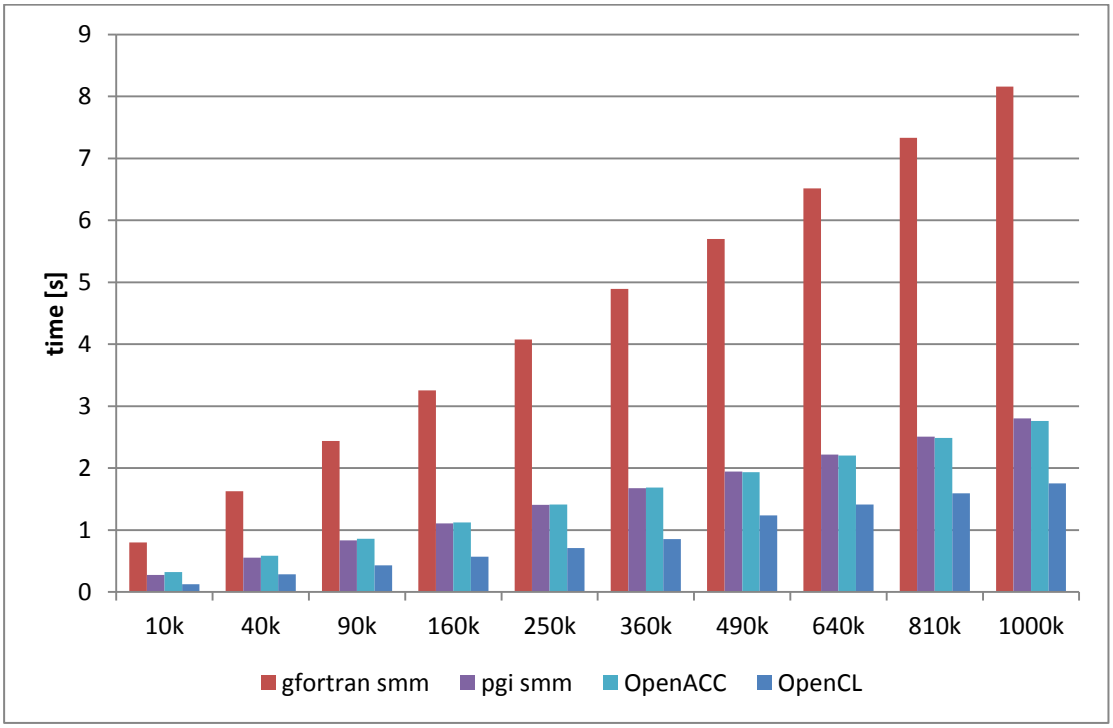

Figure 92 Test results for `mm_stack_size = 10000`

Conclusion

In conclusion, our tests demonstrated the potential of both OpenACC and OpenCL. With OpenACC we have been able to obtain a very good performance without many modifications to the source code. We added a new method with OpenACC directives and changed the code so the new method would be called. We tried to introduce as few changes as possible to the original code, to check how much work is required to introduce the technique into the existing application. The results show that, like we expected, in order to get the most of the GPUs some substantial modifications of the library would be needed. A comparison of PGI SMM results with OpenACC confirms our expectations. Lack of increase in performance after increasing the maximum stack size suggests that more changes in the code have to be made. Nevertheless, OpenACC as a whole deserves attention. We have shown that with minor code modifications one can achieve good results and we believe that with proper adjustments even better performance can be achived.

It must be noticed that the power of the OpenACC standard strongly depends on the compilers' support. The choice of proprietary compilers seems to be not sufficient, especially in a current landscape of scientific applications which often are open-source and developed using open tools and compilers like GCC. This may lead to compatibility issues, as described above for CP2K. However, OpenACC once introduced to the code, in connection with the compiler support for different targets, is a powerful technology. The OpenACC directives may be used by a compiler to generate kernels for new emerging architectures, as soon as they become supported.

One possible way for further evaluation of OpenACC in CP2K could be to copy the data on GPU before the multiplication, similar to the OpenCL and CUDA approach, rather than right at the strat of the OpenACC kernel. We have started to investigate this approach, and will continue if time allows. It requires some bigger changes in the original FORTRAN code, as in OpenACC we are opperating on the pragma level. Another direction worth considering is investigating the CAPS source-to-source compilers [8] together with GNU or Intel compilers, avoiding the PGI compilers. Combining CAPS and Intel compilers would give another possibility to target the Intel Xeon Phi as one of the accelerators, in addition to the work done with CP2K on this architecture so far [17].

The OpenCL results show a better performance to the other methods tested if proper value of `mm_stack_size` parameter is used. If time allows, further work will be focused on improving the performance and identifying additional areas of the code suitable for porting.

As an additional conclusion we may state that introducing OpenACC to an existing application is relatively simpler and requires less knowledge and time from the developer than OpenCL. However, it still requires a good understanding of the application, its data and algorithms, and may require refactoring of the original code to gain a performance as expected from the GPU acceleration.

References

[1] CP2K homepage, http://www.cp2k.org
[2] I. Bethune, A. Carter, X. Guo, P. Korosoglou: "Million Atom KS-DFT with CP2K", PRACE whitepaper, PRACE-1IP, pdf: http://www.prace-ri.eu/IMG/pdf/Million_Atom_KS-DFT_with_CP2K.pdf
[3] Unified European Application Benchmark Suite (UEABS), http://www.prace-ri.eu/ueabs
[4] PRACE Public deliverable, D7.2.1, "A Report on the Survey of HPC Tools and Techniques", 2013, pdf: http://www.prace-ri.eu/IMG/pdf/d7.2.1.pdf
[5] The OpenACC 2.0a Specification (Corrected), pdf: http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf
[6] The ERF (X) function in a draft of the Fortran 2008 Standard, http://nag.co.uk/sc22wg5
[7] Takuya Ooura: Ooura's Mathematical Software Packages – Error Functions, Institute for Mathematical Sciences, Kyoto University, http://www.kurmis.kyoto-u.ac.jp/~ooura/
[8] CAPS compilers, http://www.caps-entreprise.com/products/caps-compilers/
[9] The OpenACC homepage, http://www.openacc.org
[10] OpenCL Khronos Group homepage, http://www.khronos.org/opencl/
[11] The Khronos OpenCL Registry with OpenCL 2.0 specification, http://www.khronos.org/registry/cl/
[12] Allinea DDT homepage, http://www.allinea.com/products/ddt/
[13] Portland Group's PGI Accelerator compilers, http://www.pgroup.com/resources/accel.htm
[14] PGDBG Graphical Symbolic Debugger, http://www.pgroup.com/products/pgdbg.htm
[15] I. Bethune: "Improving the scalability of CP2K on multi-core systems. A dCSE Project", 2010, pdf: http://www.hector.ac.uk/cse/distributedcse/reports/cp2k02/cp2k02_final_report.pdf

[16] Urban Borštnik, Joost VandeVondele, Valéry Weber, Jrg Hutter: “Sparse Matrix Multiplication: The Distributed Block-Compressed Sparse Row Library”, Parallel Computing, Available online 1 April 2014, http://www.sciencedirect.com/science/article/pii/S0167819114000428
[17] Fiona Reid, Iain Bethune: “Evaluating CP2K on Exascale Hardware: Intel Xeon Phi”, PRACE whitepaper, PRACE-3IP, 2014

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-312763, and by the Polish Ministry of Science and Higher Education (funds for R&D in 2013-2014) under grant agreement no. 2890/7.PR/2013/2. The calculations were carried out on computing resources at Wrocław Centre for Networking and Supercomputing (Supernova), ACK Cyfronet AGH (Zeus) and Irish Centre for High End Computing (Fionn).