CP2K Artificial Benchmark

The first code sample given for LIBXSMM was a performance reproducer exercising the same set of kernels usually generated for CP2K's SMM library. The code sample attempted to model the way "matrix stacks" are processed in CP2K, however there are two different code paths in CP2K: (1) the "main" code path used when processing stacks on the host-side, and (2) a code path targeting offload devices. Beside of the host-sided parallelization via MPI (and perhaps OpenMP), the secondly mentioned code path relies on an additional level of parallelization (which is obviously necessary in order to drive a potentially highly parallel offload device). Also, the additional level of parallelism is not exactly "nested" in the sense that it participates on sharing the same resources as the host-side. In fact, this "artificial benchmark" (cp2k code sample) is modeling a code path as utilized in the secondly mentioned case (offload device).

Dispatch (Microbenchmark)

This code sample attempts to benchmark the performance of the dispatch mechanism. This mechanism is relevant when replacing GEMM calls (see Call Wrapper section of the reference documentation), or generally when calling LIBXSMM's `libxsmm_?gemm` functions.

Command Line Interface (CLI)

- Optionally takes the number of dispatches to be performed
- Measures the duration needed to find the requested kernel
- Excludes the time needed to actually generate the kernel
- Shows time needed in relation to an empty function call

NEK Sample Collection

This directory contains kernels taken from Nek{Box,5000}. They aim to represent the majority of the matrix-matrix workload.

Please note that the mxm_std.f source code is protected by an (US) GOVERNMENT LICENSE, and under the copyright of the University of Chicago.

stpm

Small tensor-product multiple (stpm) replicates the axhelm kernel, which computes the Laplacian with spectral elements.
Usage:

```
./stpm m n k size1 size
```

The elements are m-by-n-by-k, mode picks the libxsmm interace used, and size scales the number of spectral elements.

rstr

Restriction operator transforms elements from one size to another. This occurs in multi-grid, the convection operator, and, when the sizes are the same, the local Schwarz solves. Usage:

```
./rstr m n k mm nn kk size1 size
```

The input elements are m-by-n-by-k and the output elements are mm-by-nn-by-kk. When m=mm, n=nn, k=kk, this half of a Schwarz solve.

SMM Sample Collection

This collection of code samples exercises different memory streaming cases when performing the matrix multiplication $C_{mxn} = alpha \cdot A_{mxk} \cdot B_{kxn} + beta \cdot C_{mxn}$: (1) streaming the matrices A, B, and C which is usually referred as batched matrix multiplication, (2) streaming the inputs A and B but accumulating C within cache, (3) streaming the A and C matrices while B is kept in cache, (4) streaming the B and C matrices while A is kept in cache, and

(4) not streaming any of the operands but repeating the very same multiplication until the requested number of matrix multiplications has been completed.

Beside of measuring the duration of a test case, the performance is presented in GFLOPS/s. As an alternative metric, the memory bandwidth is given (the artificial "cached" case omits to present the cache-memory bandwidth). The "pseudo-performance" given in FLOPS/cycle is an artificial scoring, it not only uses a non-standard formula for calculating the FLOPS (*2 * M * N * K - M * N* rather than *2 * M * N * K*) but also relies on pseudo clock cycles:

```
$ ./specialized.sh 32
m=32 n=32 k=32 size=87381 memory=2048.0 MB (DP)

Batched (A,B,C)...
        pseudo-perf.: 10.7 FLOPS/cycle
        performance: 23.9 GFLOPS/s
        bandwidth: 11.1 GB/s
        duration: 239 ms
Streamed (A,B)...
        pseudo-perf.: 13.4 FLOPS/cycle
        performance: 29.9 GFLOPS/s
        bandwidth: 7.0 GB/s
        duration: 192 ms
Streamed (A,C)...
        pseudo-perf.: 12.3 FLOPS/cycle
        performance: 27.4 GFLOPS/s
        bandwidth: 6.4 GB/s
        duration: 209 ms
Streamed (B,C)...
        pseudo-perf.: 14.8 FLOPS/cycle
        performance: 33.0 GFLOPS/s
        bandwidth: 7.7 GB/s
        duration: 173 ms
Cached...
        pseudo-perf.: 23.2 FLOPS/cycle
        performance: 51.8 GFLOPS/s
        duration: 111 ms
Finished
```

There are two sub collections of samples codes: (1) a collection of C++ code samples showing either BLAS, Compiler-generated code (inlined code), LIBXSMM/dispatched, LIBXSMM/specialized functions to carry out the multiplication, and (2) a Fortran sample code showing BLAS versus LIBXSMM including some result validation.

C/C++ Code Samples: Command Line Interface (CLI)

- Optionally takes the M, N, and K parameter of the GEMM in this order
- If only M is supplied, the N and K "inherit" the M-value
- Shows the performance of each of the streaming cases
- Example I: ./specialized.sh 16 8 9
- Example II: ./specialized.sh 16

Fortran Code Sample: Command Line Interface (CLI)

- Optionally takes the M, N, and K parameter of the GEMM in this order
- Optional problem size (in MB) of the workload; M/N/K must have been supplied
- Optional total problem size (in MB) implying the number of repeated run
- If only M is supplied, the N and K are "inheriting" the M-value
- Shows the performance of each of the streaming cases
- Example I: ./smm.sh 16 8 9 1024 16384
- Example II: ./smm.sh 16

SPECFEM Sample

This sample contains a dummy example from a spectral-element stiffness kernel taken from SPECFEM3D_GLOBE.

It is based on a 4th-order, spectral-element stiffness kernel for simulations of elastic wave propagation through the Earth. Matrix sizes used are (25,5), (5,25) and (5,5) determined by different cut-planes through a three dimensional (5,5,5)-element with a total of 125 GLL points.

Usage Step-by-Step

This example needs the LIBXSMM library to be built with static kernels, using MNK="5 25" (for matrix size (5,25), (25,5) and (5,5)).

1. In LIBXSMM root directory, compile the library with:

- general default compilation:

```
make MNK="5 25" ALPHA=1 BETA=0
```

additional compilation examples are:

- compilation using only single precision version & aggressive optimization:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3
```

- for Sandy Bridge CPUs:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1
```

- for Haswell CPUs:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=2
```

- for Knights Corner (KNC) (and also creating Sandy Bridge version):

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1 \
OFFLOAD=1 KNC=1
```

- installing libraries into a sub-directory workstation/:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1 \
OFFLOAD=1 KNC=1 \
PREFIX=workstation/ install-minimal
```

1. Compile this example code by typing:

- for default CPU host:

```
cd sample/specfem
make
```

- for Knights Corner (KNC):

```
cd sample/specfem
make KNC=1
```

- additionally, adding some specific fortran compiler flags, for example:

```
cd sample/specfem
make FCFLAGS="-O3 -fopenmp" [...]
```

Note that steps 1 & 2 could be shortened:

- by specifying a "specfem" make target in the LIBXSMM root directory:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1 specfem
```

- for Knights Corner, this would need two steps:

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1 OFFLOAD=1 KNC=1
make OPT=3 specfem_mic
```

Run the performance test:

- for default CPU host:

```
./specfem.sh
```

- for Knights Corner (KNC):

```
./specfem.sh -mic
```

Results

Using Intel Compiler suite: icpc 15.0.2, icc 15.0.2, and ifort 15.0.2

Sandy Bridge - Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz

- library compilation by (root directory):

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=1
```

- single threaded example run:

```
cd sample/specfem
make; OMP_NUM_THREADS=1 ./specfem.sh
```

Output:

```
============================================================
average over           15 repetitions
 timing with Deville loops    =   0.1269
 timing with unrolled loops   =   0.1737 / speedup =   -36.87 %
 timing with LIBXSMM dispatch =   0.1697 / speedup =   -33.77 %
 timing with LIBXSMM prefetch =   0.1611 / speedup =   -26.98 %
 timing with LIBXSMM static   =   0.1392 / speedup =    -9.70 %
============================================================
```

Haswell - Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz

- library compilation by (root directory):

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 AVX=2
```

- single threaded example run:

```
cd sample/specfem
make; OMP_NUM_THREADS=1 ./specfem.sh
```

Output:

```
============================================================
average over           15 repetitions
 timing with Deville loops    =   0.1028
 timing with unrolled loops   =   0.1385 / speedup =   -34.73 %
 timing with LIBXSMM dispatch =   0.1408 / speedup =   -37.02 %
 timing with LIBXSMM prefetch =   0.1327 / speedup =   -29.07 %
 timing with LIBXSMM static   =   0.1151 / speedup =   -11.93 %
============================================================
```

- multi-threaded example run:

```
cd sample/specfem
make OPT=3; OMP_NUM_THREADS=24 ./specfem.sh
```

Output:

```
OpenMP information:
  number of threads =           24

[...]

============================================================
average over           15 repetitions
 timing with Deville loops    =   0.0064
 timing with unrolled loops   =   0.0349 / speedup =  -446.71 %
 timing with LIBXSMM dispatch =   0.0082 / speedup =   -28.34 %
 timing with LIBXSMM prefetch =   0.0076 / speedup =   -19.59 %
 timing with LIBXSMM static   =   0.0068 / speedup =    -5.78 %
============================================================
```

Knights Corner - Intel Xeon Phi B1PRQ-5110P/5120D

- library compilation by (root directory):

```
make MNK="5 25" ALPHA=1 BETA=0 PRECISION=1 OPT=3 OFFLOAD=1 KNC=1
```

- multi-threaded example run:

```
cd sample/specfem
make FCFLAGS="-O3 -fopenmp -warn" OPT=3 KNC=1; ./specfem.sh -mic
```

Output:

```
OpenMP information:
  number of threads =          236

[...]

============================================================
average over           15 repetitions
 timing with Deville loops    =   0.0164
 timing with unrolled loops   =   0.6982 / speedup = -4162.10 %
 timing with LIBXSMM dispatch =   0.0170 / speedup =    -3.89 %
 timing with LIBXSMM static   =   0.0149 / speedup =     9.22 %
============================================================
```

Wrapped DGEMM

This code sample is calling DGEMM and there is no dependency on the LIBXSMM API as it only relies on LAPACK-/BLAS interface. Two variants are linked when building the source code: (1) code which is dynamically linked against LAPACK/BLAS, (2) code which is linked using `--wrap=`*symbol* as possible when using a GNU GCC compatible tool chain. For more information, see the Call Wrapper section of the reference documentation.

The code will execute in three flavors when running `dgemm-test.sh`: (1) code variant which is dynamically linked against the originally supplied LAPACK/BLAS library, (2) code variant which is linked using the wrapper mechanism of the GNU GCC tool chain, and (3) the first code but using the LD_PRELOAD mechanism (available under Linux).

Command Line Interface (CLI)

- Optionally takes the number of repeated DGEMM calls
- Shows the performance of the workload (wall time)