JIT Backend

Overview

There might be situations in which it is up-front not clear which problem-sizes will be needed when running an application. To leverage LIBXSMM's high-performance kernels, the library implements a JIT (Just-In-Time) code generation backend which generates the requested kernels on the fly (in-memory). This is accomplished by emitting the corresponding byte-code directly into an executable buffer. The actual JIT code is generated per the CPUID flags, and therefore does not rely on the code path selected when building the library. In the current implementation, some limitations apply to the JIT backend specifically:

1. To stay agnostic to any threading model used, Pthread mutexes are guarding the updates of the JIT'ted code cache (link line with `-lpthread` is required); building with OMP=1 employs an OpenMP critical section as an alternative locking mechanism.
2. There is no support for the Intel SSE (Intel Xeon 5500/5600 series) and IMCI (Intel Xeon Phi coprocessor code-named Knights Corner) instruction set extensions. However, statically generated SSE-kernels can be leveraged without disabling support for JIT'ting AVX kernels.
3. There is no support for the Windows calling convention (only kernels with PREFETCH=0 signature).

The JIT backend can also be disabled at build time (`make JIT=0`) as well as at runtime (`LIBXSMM_TARGET=0`, or anything prior to Intel AVX). The latter is an environment variable which allows to set a code path independent of the CPUID (LIBXSMM_TARGET=0|1|sse|snb|hsw|knl|knm|skx). Please note that LIBXSMM_TARGET cannot enable the JIT backend if it was disabled at build time (JIT=0).

One can use the afore mentioned THRESHOLD parameter to control the matrix sizes for which the JIT compilation will be automatically performed. However, explicitly requested kernels (by calling `libxsmm_?mmdispatch`) fall not under a threshold for the problem-size. In any case, JIT code generation can be used for accompanying statically generated code.

Generator Driver

In rare situations, it might be useful to directly incorporate generated C code (with inline assembly regions). This is accomplished by invoking a driver program (with certain command line arguments). The driver program is built as part of LIBXSMM's build process (when requesting static code generation), but also available via a separate build target:

```
make generator
bin/libxsmm_gemm_generator
```

The code generator driver program accepts the following arguments:

1. dense/dense_asm/sparse (dense creates C code, dense_asm creates ASM)
2. Filename of a file to append to
3. Routine name to be created
4. M parameter
5. N parameter
6. K parameter
7. LDA (0 when 1. is "sparse" indicates A is sparse)
8. LDB (0 when 1. is "sparse" indicates B is sparse)
9. LDC parameter
10. alpha (1)
11. beta (0 or 1)
12. Alignment override for A (1 auto, 0 no alignment)
13. Alignment override for C (1 auto, 0 no alignment)
14. Architecture (noarch, wsm, snb, hsw, knc, knl, knm, skx)
15. Prefetch strategy, see below enumeration (dense/dense_asm only)
16. single precision (SP), or double precision (DP)
17. CSC file (just required when 1. is "sparse"). Matrix market format.

The prefetch strategy can be:

1. "nopf": no prefetching at all, just 3 inputs (A, B, C)
2. "pfsigonly": just prefetching signature, 6 inputs (A, B, C, A', B', C')
3. "BL2viaC": uses accesses to C to prefetch B'

4. “curAL2”: prefetches current A ahead in the kernel
5. “curAL2_BL2viaC”: combines curAL2 and BL2viaC
6. “AL2”: uses accesses to A to prefetch A’
7. “AL2_BL2viaC”: combines AL2 and BL2viaC
8. “AL2jpst”: aggressive A’ prefetch of first rows without any structure
9. “AL2jpst_BL2viaC”: combines AL2jpst and BL2viaC
10. “AL1”: prefetch A’ into L1 via accesses to A
11. “BL1”: prefetch B’ into L1 via accesses to B
12. “CL1”: prefetch C’ into L1 via accesses to C
13. “AL1_BL1”: prefetch A’ and B’ into L1
14. “BL1_CL1”: prefetch B’ and C’ into L1
15. “AL1_CL1”: prefetch A’ and C’ into L1
16. “AL1_BL1_CL1”: prefetch A’, B’, and C’ into L1

Here are some examples of invoking the driver program:

```
bin/libxsmm_gemm_generator dense foo.c foo 16 16 16 32 32 32 1 1 1 1 hsw nopf DP
bin/libxsmm_gemm_generator dense_asm foo.c foo 16 16 16 32 32 32 1 1 1 1 knl AL2_BL2viaC DP
bin/libxsmm_gemm_generator sparse foo.c foo 16 16 16 32 0 32 1 1 1 1 hsw nopf DP bar.csc
```

Please note, there are additional examples given in samples/generator and samples/seissol.