Service Functions

Getting and Setting the Target Architecture

There are ID based and string based functions to query the code path (as determined by the CPUID), or to set the code path regardless of the presented CPUID features. The latter may degrade performance (if a lower set of instruction set extensions is requested), which can be still useful for studying the performance impact of different instruction set extensions. This functionality is available for the C and Fortran interface, and there is an environment variable which corresponds to `libxsmm_set_target_arch` (LIBXSMM_TARGET).
NOTE: There is no additional check performed if an unsupported instruction set extension is requested, and incompatible JIT-generated code may be executed (unknown instruction signaled).

```
int libxsmm_get_target_archid(void);
void libxsmm_set_target_archid(int id);

const char* libxsmm_get_target_arch(void);
void libxsmm_set_target_arch(const char* arch);
```

Getting and Setting the Verbosity

The Verbose Mode (level of verbosity) can be controlled using the C or Fortran API, and there is an environment variable which corresponds to `libxsmm_set_verbosity` (LIBXSMM_VERBOSE).

```
int libxsmm_get_verbosity(void);
void libxsmm_set_verbosity(int level);
```

Timer Facility

Due to the performance oriented nature of LIBXSMM, timer-related functionality is available for the C and Fortran interface ('libxsmm_timer.h' and 'libxsmm.f'). This is used for instance by the code samples, which measure the duration of executing various code regions. Both "tick" functions (`libxsmm_timer_[x]tick`) are based on monotonic timer sources, which use a platform-specific resolution. The xtick-counter attempts to directly rely on the time stamp counter instruction (RDTSC), but it is not necessarily counting real CPU cycles due to varying CPU clock speed (Turbo Boost), different clock domains (e.g., depending on the instructions executed), and other reasons (which are out of scope in this context).
NOTE: `libxsmm_timer_xtick` is not directly suitable for `libxsmm_timer_duration` (seconds).

```
unsigned long long libxsmm_timer_tick(void);
unsigned long long libxsmm_timer_xtick(void);
double libxsmm_timer_duration(unsigned long long tick0, unsigned long long tick1);
```

Meta Image File I/O

Loading and storing data (I/O) is normally out of LIBXSMM's scope. However, comparing results (correctness) or writing files for visual inspection is clearly desired. This is useful with image-processing and the DNN domain in particular. The MHD library domain provides support for the Meta Image File format (MHD). Tools such as ITK-SNAP or ParaView can be used to inspect, compare, and modify images (even beyond two-dimensional images).

Writing an image is per `libxsmm_mhd_write`, and loading an image is split in two stages: (1) `libxsmm_mhd_read_header`, and (2) `libxsmm_mhd_read`. The first step allows to allocate a properly sized buffer, which is then used to obtain the data per `libxsmm_mhd_read`. When reading data, an on-the-fly type conversion is supported. Further, data that is already in memory can be compared against file-data without allocating memory or reading this file into memory.

Memory Allocation

The C interface ('libxsmm_malloc.h') provides functions for aligned memory one of which allows to specify the alignment (or to request an automatically selected alignment). The automatic alignment is also available with a `malloc` compatible signature. The size of the automatic alignment depends on a heuristic, which uses the size of the requested buffer.
NOTE: Only `libxsmm_free` is supported to deallocate the memory.

```
void* libxsmm_malloc(size_t size);
void* libxsmm_aligned_malloc(size_t size, size_t alignment);
void* libxsmm_aligned_scratch(size_t size, size_t alignment);
void libxsmm_free(const volatile void* memory);
int libxsmm_get_malloc_info(const void* memory, libxsmm_malloc_info* info);
int libxsmm_get_scratch_info(libxsmm_scratch_info* info);
```

The library exposes two memory allocation domains: (1) default memory allocation, and (2) scratch memory allocation. There are service functions for both domains that allow to change the allocation and deallocation function. The "context form" even supports a user-defined "object", which may represent an allocator or any other external facility. To set the default allocator is analogous to setting the scratch memory allocator as shown below. See include/libxsmm_malloc.h for details.

```
int libxsmm_set_scratch_allocator(void* context,
  libxsmm_malloc_function malloc_fn, libxsmm_free_function free_fn);
int libxsmm_get_scratch_allocator(void** context,
  libxsmm_malloc_function* malloc_fn, libxsmm_free_function* free_fn);
```

There are currently no claims on the properties of the default memory allocation (e.g., thread scalability). The scratch memory allocation is very effective and delivers a decent speedup over subsequent regular memory allocations. In contrast to the default allocation technique, the scratch memory establishes a watermark for buffers which would be repeatedly allocated and deallocated. By establishing a pool of "temporary" memory, the cost of repeated allocation and deallocation cycles is avoided when the watermark is reached. The scratch memory is scope-oriented, and supports only a limited number of pools for buffers of different life-time. The verbose mode with a verbosity level of at least two shows some statistics about the populated scratch memory.

```
Scratch: 173 MB (mallocs=5, pools=1)
```

NOTE: be careful with scratch memory as it only grows during execution (in between `libxsmm_init` and `libxsmm_finalize` unless `libxsmm_release_scratch` is called). This is true even when `libxsmm_free` is (and should be) used!