# Profiling

## Intel VTune Amplifier

To analyze which kind of kernels have been called, and from where these kernels have been invoked (call stack), the library allows profiling its JIT code using Intel VTune Amplifier. To enable this support, VTune's root directory needs to be set at build-time of the library. Enabling symbols (SYM=1 or DBG=1) incorporates VTune's JIT Profiling API:

```
source /path/to/vtune_amplifier/amplxe-vars.sh
make SYM=1
```

Above, the root directory is automatically determined from the environment (VTUNE_AMPLIFIER_*_DIR). This variable is present after source'ing the Intel VTune environment, but it can be manually provided as well (`make VTUNEROOT=/path/to/vtune_amplifier`). Symbols are not really required to display kernel names for the dynamically generated code, however enabling symbols makes the analysis much more useful for the rest of the (static) code, and hence it has been made a prerequisite. For example, when "call stacks" are collected it is possible to find out where the JIT code has been invoked by the application:

```
amplxe-cl -r result-directory -data-limit 0 -collect advanced-hotspots \
          -knob collection-detail=stack-sampling -- ./myapplication
```

In case of an MPI-parallelized application, it might be useful to only collect results from a "representative" rank, and to also avoid running the event collector in every rank of the application. With Intel MPI both of which can be achieved by adding

```
-gtool 'amplxe-cl -r result-directory -data-limit 0 -collect advanced-hotspots \
                  -knob collection-detail=stack-sampling:4=exclusive'
```

to the `mpirun` command line. Please notice the `:4=exclusive` (unrelated to VTune's command line syntax), which is related to mpirun's gtool arguments; these arguments need to appear at the end of the gtool-string. For instance, the shown command line selects the 4th rank (otherwise all ranks are sampled) along with "exclusive" usage of the performance monitoring unit (PMU) such that only one event-collector runs for all ranks.

Intel VTune Amplifier presents invoked JIT code like functions, which belong to a module named "libxsmm.jit". The function name as well as the module name are supplied by LIBXSMM using the afore mentioned JIT Profiling API. For instance "libxsmm_hsw_dnn_23x23x23_23_23_23_a1_b1_p0::mxm" encodes an Intel AVX2 ("hsw") double-precision kernel ("d") for small dense matrix multiplications ("mxm") which is multiplying matrices without transposing them ("nn"). The rest of the name encodes M=N=K=LDA=LDB=LDC=23, Alpha=Beta=1.0 (all similar to GEMM), and no prefetch strategy ("p0").

## Linux perf

With LIBXSMM, there is both basic (`perf map`) and extended support (`jitdump`) when profiling an application. To enable perf support at runtime, the environment LIBXSMM_VERBOSE needs to be set to a negative value.

- The basic support can be enabled at compile-time with PERF=1 (implies SYM=1) using `make PERF=1`. At runtime of the application, a map-file ('jit-*pid*.map') is generated ('/tmp' directory). This file is automatically read by Linxu perf, and enriches the information about unknown code such as JIT'ted kernels.
- The support for "jitdump" can be enabled by supplying JITDUMP=1 (implies PERF=1) or PERF=2 (implies JITDUMP=1) when making the library: `make JITDUMP=1` or `make PERF=2`. At runtime of the application, a dump-file ('jit-*pid*.dump') is generated (in perf's debug directory, usually `$HOME/.debug/jit/`) which includes information about JIT'ted kernels (such as addresses, symbol names, code size, and the code itself). The dump file can be injected into 'perf.data' (using `perf inject -j`), and it enables an annotated view of the assembly in perf's report (requires a reasonably recent version of perf).

## Tuning

Specifying a code path is not really necessary if the JIT backend is not disabled. However, disabling JIT compilation, statically generating a collection of kernels, and targeting a specific instruction set extension for the entire library looks like:

```
make JIT=0 AVX=3 MNK="1 2 3 4 5"
```

The above example builds a library which cannot be deployed to anything else but the Intel Knights Landing processor family ("KNL") or future Intel Xeon processors supporting foundational Intel AVX-512 instructions (AVX-512F). The

latter might be even more adjusted by supplying MIC=1 (along with AVX=3), however this does not matter since critical code is in inline assembly (and not affected). Similarly, SSE=0 (or JIT=0 without SSE or AVX build flag) employs an "arch-native" approach whereas AVX=1, AVX=2 (with FMA), and AVX=3 are specifically selecting the kind of Intel AVX code. Moreover, controlling the target flags manually or adjusting the code optimizations is also possible. The following example is GCC-specific and corresponds to OPT=3, AVX=3, and MIC=1:

```
make OPT=3 TARGET="-mavx512f -mavx512cd -mavx512er -mavx512pf"
```

An extended interface can be generated which allows to perform software prefetches. Prefetching data might be helpful when processing batches of matrix multiplications where the next operands are farther away or otherwise unpredictable in their memory location. The prefetch strategy can be specified similar as shown in the section Generator Driver i.e., by either using the number of the shown enumeration, or by exactly using the name of the prefetch strategy. The only exception is PREFETCH=1 which is automatically selecting a strategy per an internal table (navigated by CPUID flags). The following example is requesting the "AL2jpst" strategy:

```
make PREFETCH=8
```

The prefetch interface is extending the signature of all kernels by three arguments (pa, pb, and pc). These additional arguments are specifying the locations of the operands of the next multiplication (the next a, b, and c matrices). Providing unnecessary arguments in case of the three-argument kernels is not big a problem (beside of some additional call-overhead), however running a kernel which is picking up more than three arguments and thereby picking up garbage data is misleading or disabling the hardware prefetcher (due to software prefetches). In this case, a misleading prefetch location is given plus an eventual page fault due to an out-of-bounds (garbage-)location.

Further, the generated configuration (template) of the library encodes the parameters for which the library was built for (static information). This helps optimizing client code related to the library's functionality. For example, the LIBXSMM_MAX_* and LIBXSMM_AVG_* information can be used with the LIBXSMM_PRAGMA_LOOP_COUNT macro to hint loop trip counts when handling matrices related to the problem domain of LIBXSMM.

### Auto-dispatch

The function `libxsmm_?mmdispatch` helps amortizing the cost of the dispatch when multiple calls with the same M, N, and K are needed. The automatic code dispatch is orchestrating two levels:

1. Specialized routine (implemented in assembly code),
2. BLAS library call (fallback).

Both levels are accessible directly, which allows to customize the code dispatch. The fallback level may be supplied by the Intel Math Kernel Library (Intel MKL) 11.2 DIRECT CALL feature.

Further, a preprocessor symbol denotes the largest problem-size ($M$ x $N$ x $K$) that belongs to the first level, and therefore determines if a matrix multiplication falls back to BLAS. The problem-size threshold can be configured by using for example:

```
make THRESHOLD=$((60 * 60 * 60))
```

The maximum of the given threshold and the largest requested specialization refines the value of the threshold. Please note that explicitly JIT'ting and executing a kernel is possible and independent of the threshold. If a problem-size is below the threshold, dispatching the code requires to figure out whether a specialized routine exists or not.

To minimize the probability of key collisions (code cache), the preferred precision of the statically generated code can be selected:

```
make PRECISION=2
```

The default preference is to generate and register both single and double-precision code, and therefore no space in the dispatch table is saved (PRECISION=0). Specifying PRECISION=1|2 is only generating and registering either single-precision or double-precision code.

The automatic dispatch is highly convenient because existing GEMM calls can serve specialized kernels (even in a binary compatible fashion), however there is (and always will be) an overhead associated with looking up the code-registry and checking whether the code determined by the GEMM call is already JIT'ted or not. This lookup has been optimized using various techniques such as using specialized CPU instructions to calculate CRC32 checksums, to avoid costly synchronization (needed for thread-safety) until it is ultimately known that the requested kernel is not yet JIT'ted, and by implementing a small thread-local cache of recently dispatched kernels. The latter of which can be adjusted in size (only power-of-two sizes) but also disabled:

```
make CACHE=0
```

Please note that measuring the relative cost of automatically dispatching a requested kernel depends on the kernel size (obviously smaller matrices are multiplied faster on an absolute basis), however smaller matrix multiplications are bottlenecked by memory bandwidth rather than arithmetic intensity. The latter implies the highest relative overhead when (artificially) benchmarking the very same multiplication out of the CPU-cache.