IXPUG
INTEL EXTREME PERFORMANCE USERS GROUP

LIBxSMM86 What‘s Inside?

Intel High Performance and
Throughput Computing

Hans Pabst

https://github.com/hfp/libxsmm

March 5th 2018, CINECA, Bologna

LIBXSMM LIBRARY TARGETING INTEL ARCHITECTURE (X86)

FOR SMALL, DENSE OR SPARSE MATRIX MULTIPLICATIONS, AND SMALL CONVOLUTIONS.

Intel High Performance and Throughput Computing

Hans Pabst

https://github.com/hfp/libxsmm

Parallel Computing Lab
Intel Labs, USA

Alexander Heinecke
Evangelos Georganas

Software and Services Group,
Pathfinding, USA

Greg Henry

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Modern HPC….

$$\boldsymbol{Speedup} = \left(\frac{1}{Serial_{frac} + \frac{1 - Serial_{frac}}{\boldsymbol{NumCores}}}\right) * \left(\frac{1}{Scalar_{frac} + \frac{1 - Scalar_{frac}}{\boldsymbol{VectorLength}}}\right)$$

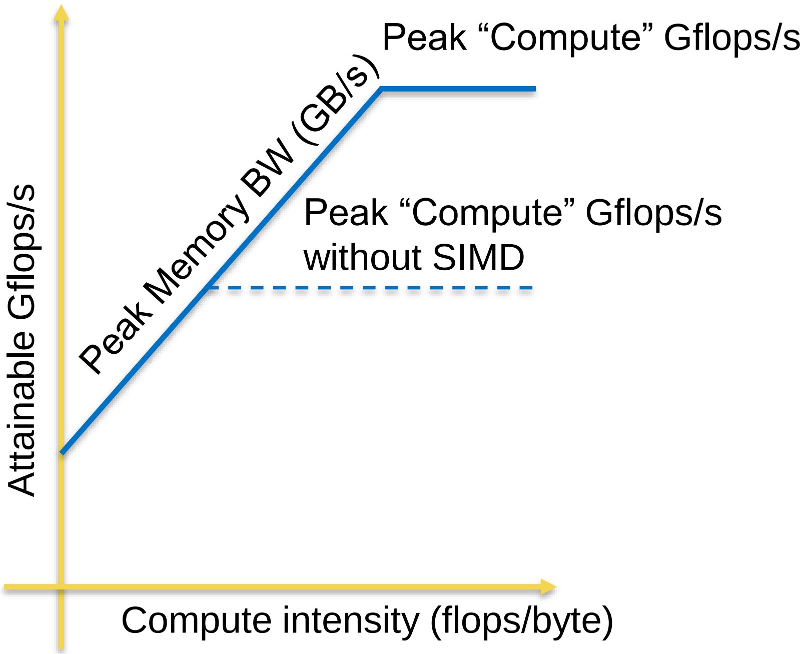
Goal: *Reduce* ***Serial Fraction*** and *Reduce* ***Scalar Fraction*** of Code

Ideal Speedup: **NumCores * VectorLength (requires zero scalar, zero serial work)**

Bandwidth Bound Performance

Kernels are often memory BW bound i.e., not enough compute intensity (flops/byte). *But: modern chips are unbalanced (high flops/low byte/s)*

Can we make better "use" of data after we have moved it into the chip?



Optimizations for bandwidth-bound codes?

Valuable Optimizations

- Maximize use of the register file
- Maximize instruction throughput
  - Take maximum number of instructions per cycle into account (FE bound).
  - Consider maximum number of Bytes, which can be decoded (FE bound).
- Limited instruction mix (BE bound)
  - Address calculations incl. prefetch

Less important

- Loop unrolling; only the load/store vs. FMA/compute-mix matters

Benchmark Primer: model the real case accurately and measure accordingly!

Example of "false measurement"

```
for (i = 0; i < NREPEAT; ++i) {
   DGEMM(&transa, &transb, &m, &n, &k,
         &alpha, a, &lda, b, &ldb,
         &beta, c, &ldc);
}
```

Correct*: A, B, or C need to be streamed

```
for (i = 0; i < NREPEAT; ++i) {
   DGEMM(&transa, &transb, &m, &n, &k,
         &alpha,   a[i*lda*k], &lda,
                   b[i*ldb*n], &ldb,
         &beta,    c,          &ldc);
}
```

* C is accumulated (Beta!=0), A and B are loaded

LIBXSMM Function Domains

Main function domains in LIBXSMM

(S)MM	(Small) Matrix Multiplication Kernels (original library)
DNN	Deep Neural Network Kernels for CNNs (v1.5)
SPMDM	Sparse Matrix Dense Matrix Multiplication for CNNs (v1.6)
AUX	Mem. allocation, synchronization, debugging, profiling, …

There is more functionality…

- Tiled GEMM routines based on SMM kernels (also parallelized)
- Stand-alone out-of-place matrix transpose routines (non-JIT, and JIT)
- Matrix-copy kernels (JIT)
- Other “sparse routines”

LIBXSMM, for small, dense or sparse matrix multiplications, and small convolutions.

Highly efficient Frontend

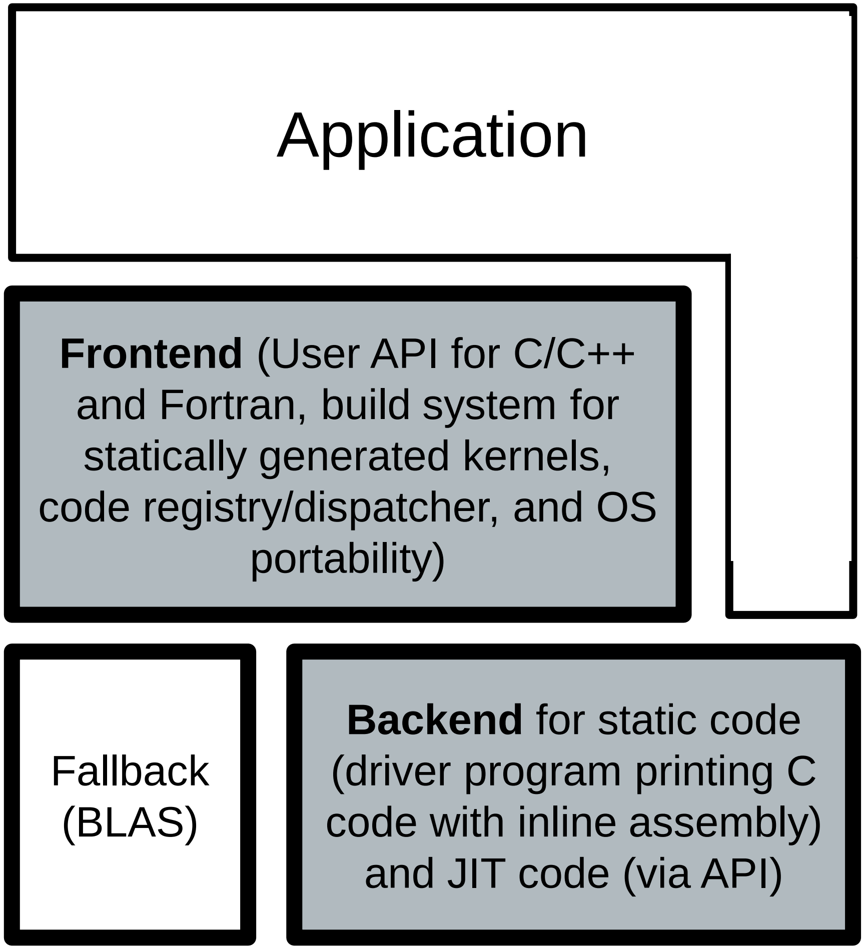
- BLAS compatible (DGEMM, SGEMM) including LD_PRELOAD
- Support for F77, C89/C99, F2003, C++
- Zero-overhead calls into assembly
- Two-level code cache

Code Generator

- Supports all Intel Architectures since 2005, focus on AVX-512
- Prefetching across small GEMMs
- Can generate assembly (*.s), inline assembly (*.h/*.c), and in-memory code

Just-In-Time (JIT) Encoder

- Encodes instructions based on basic blocks
- Very fast code generation (no compilation)



Application

Frontend (User API for C/C++ and Fortran, build system for statically generated kernels, code registry/dispatcher, and OS portability)

Fallback (BLAS)

Backend for static code (driver program printing C code with inline assembly) and JIT code (via API)

LIBXSMM (C API): Example

```
#include <libxsmm.h>

int main()
{
  const double alpha = 1.0, beta = 1.0;
  const int m = 23, n = 23, k = 23;      /* some problem size */
  double a[m*k], b[k*n], c[m*n];         /* init. not shown!   */
  libxsmm_dmmfunction xmm = NULL;        /* function pointer   */

  libxsmm_gemm(NULL, NULL, &m, &n, &k,   /* auto-dispatched */
    &alpha, a, NULL, b, NULL,
     &beta, c, NULL);

  /* like function interface for low-level JIT'ted kernel */
  libxsmm_dmm_23_23_23(a, b, c);         /* specialized */

  xmm = libxsmm_dmmdispatch(23, 23, 23, NULL, NULL, NULL,
                            &alpha, &beta, NULL, NULL);
  if (xmm) {                             /* specialized */
    for (int i = 0; i < some; ++i) {
      xmm(a, b, c);                      /* amortized   */
    }
  }
}
```

LIBXSMM (C API): Example

```
#include <libxsmm.h>

int main()
{
  const double alpha = 1.0, beta = 1.0;
  const int m = 23, n = 23, k = 23;        /* some problem size */
  double a[m*k], b[k*n], c[m*n];           /* init. not shown!  */
  libxsmm_dmmfunction x                      nction pointer   */

  libxsmm_gemm(NULL, NU                      to-dispatched */
    &alpha, a, NULL, b,
     &beta, c, NULL);

  /* like function interfac      low-level JIT'ted kernel */
  libxsmm_dmm_23_23_23(a,  , c);           /* specialized */

  xmm = libxsmm_dmmdispatch(23, 23,
                                &alp
  if (xmm) {
    for (int i = 0; i < some; ++i)
      xmm(a, b, c);
    }
  }
}
```

Explicit code generation is not subject to a problem-size threshold.

NULL-pointer is returned if the request is unsupported (alpha/beta, transpose).

Primer about GEMM…

GEneral **M**atrix **M**atrix routines (usually Real-SP, Real-DB, and Complex-SP/DP)

Original call/arguments	LIBXSMM (history)	Fixed/bound at
DGEMM('N', 'N', M, N, K,	Static compilation	JIT-compilation
ALPHA, **A**, LDA, →	JIT-compilation	JIT-compilation
B, LDB,	Call-time	Call-time
BETA, **C**, LDC)		

- LIBXSMM added relaxation for Alpha/Beta, TransA/B, and LDx later on

  JIT: only a subset of Alpha/Beta values is supported (can be exploited for optimization)

- JIT-GEMM descriptor: M, N, K, LDA, LDB, LDC, “Flags”, and “Prefetch”

  Flags: TransA & TransB, **Prefetch**: strategy (implies arity of JIT-code)

LIBXSMM: Dispatch Flow and Code Registry

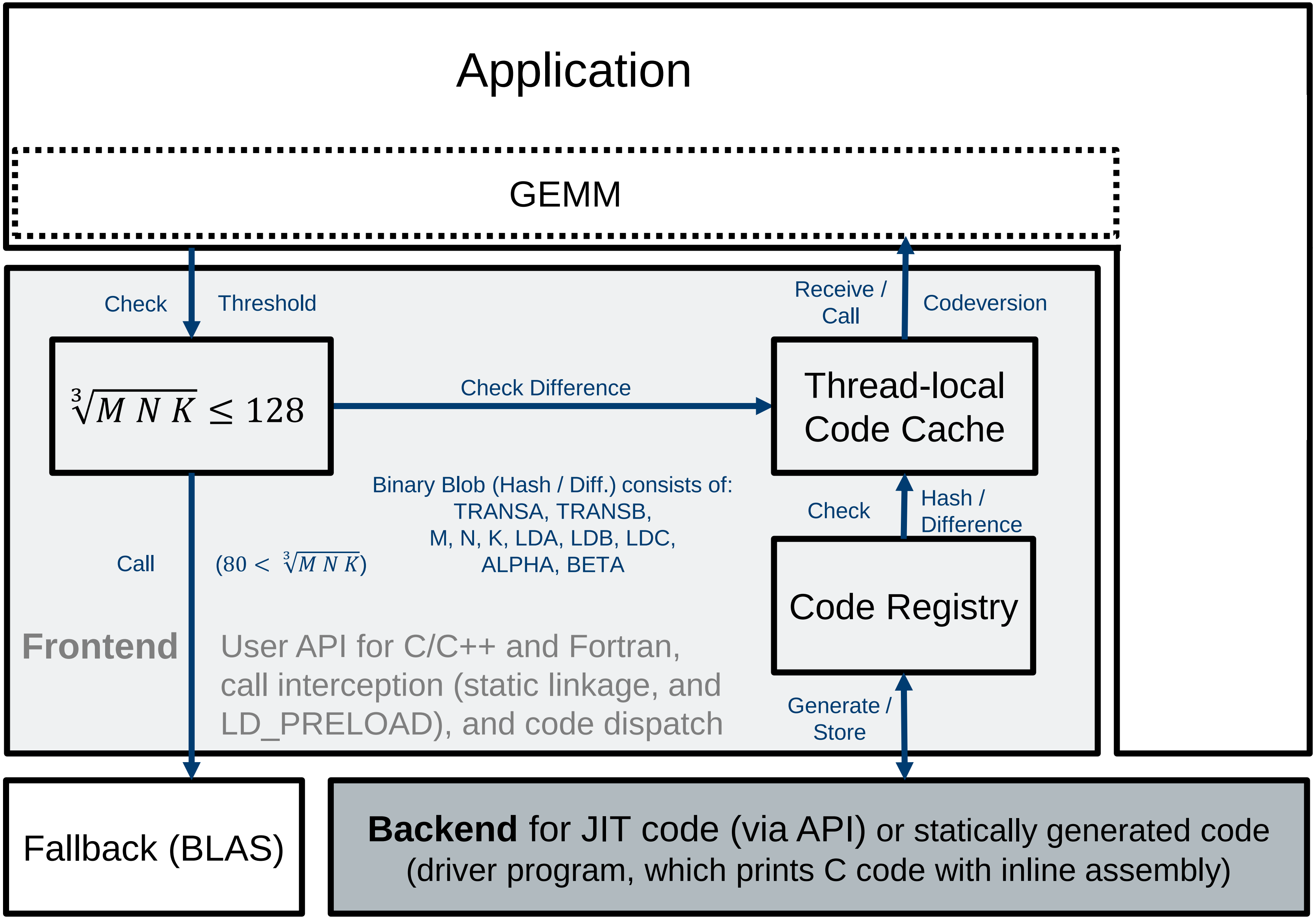
Dispatch flow: descriptor → CRC32 hash/index → [cache*] → code registry

- Registry: **custom data structure plus algorithm** for fast code retrieval
- Registry "hit" requires at least one comparison (two descriptors)
- Due to the hash-mechanism, collisions must be handled

Registry update: custom thread-safety

- Atomic reads are used, and locks are only used to protect updates/writes
- Multiple locks (POT number to simplify bitops) used to protect writes
  - Multiple updates are allowed for different locations
- Code duplication is possible due to readers not participating in locks
  - Mainly happens because of hash key collisions and multiple writer-locks
  - Only happens when code generation request is contended

* Note: [cache] is skipped here for clarity



Application

GEMM

Check

Threshold

Receive / Call

Codeversion

$\sqrt[3]{M\ N\ K} \leq 128$

Check Difference

Thread-local Code Cache

Binary Blob (Hash / Diff.) consists of:
TRANSA, TRANSB,
M, N, K, LDA, LDB, LDC,
ALPHA, BETA

Check

Hash / Difference

Call

$(80 < \sqrt[3]{M\ N\ K})$

Code Registry

Frontend User API for C/C++ and Fortran, call interception (static linkage, and LD_PRELOAD), and code dispatch

Generate / Store

Fallback (BLAS)

Backend for JIT code (via API) or statically generated code (driver program, which prints C code with inline assembly)

LIBXSMM: Code Cache

Properties

- Plays well with optimized descriptor size (multiple of SIMD width, etc.)
- Small capacity (default: 4 entries)
- Fully associative cache
- LRU-style eviction

Purpose

- Accelerate calls via GEMM interface (a.k.a. auto-dispatched)

Cache Primer

- **Direct Mapped Cache:** simplest form of cache, check for a hit without search (only one possible place which may hold an address). Drawback: same cache location shared for many addresses (depends on cache / memory size ratio).
- **Fully Associative Cache:** best hit ratio since any cache location can hold any address. Drawback: full search needed (expensive).
- **N-Way Set Associative Cache:** addresses fall into bins of fixed capacity; search needed within hit-bin. Compromise between DMC and FAC.

JIT Technologies (2015)

Evaluation of suitable JIT code generators

- Numerous projects evaluated: jitasm, libgccjit, etc.
- Selection/rejection criterions
  - Support for recent Intel Architectures,
  - Active development
- Interesting candidates reviewed in 2015 (incl. comments)
  - LLVM Full-blown (with IR, phases, etc.), “slow” code gen., complex
  - Xbyak compiler and JIT-assembler, incomplete AVX-512 (2015!)
  - XED Closed source (Dec. 2016: https://github.com/intelxed/xed)
- Xbyak: adopted by MKL/MKL-DNN in July 2016

Decision in 2015: own development needed for LIBXSMM (“JIT Assembler”)

- Typically few instruction families are needed to cover a domain (still true)
- No legacy support (AVX/2 and beyond is fine), SSE added in 2018

LIBXSMM Backend: Runtime Code Generation (Very High Level Idea)

Idea*: leveraged GNU Compiler extension “Computed GOTO”

```
LABEL1:
  c = a + b;
LABEL2:

memcpy(code, &&LABEL1, &&LABEL2 - &&LABEL1);
```

Reality: LIBXSMM manually encodes all instructions needed

- Basic form is encoded with placeholder(s) for varying parts (immediates)
- Emitting an instruction: call a function (arguments may cover instruction variants and/or immediates), to write a whole kernel is like using a DSL (“assembly programming domain”)

* https://eli.thegreenplace.net/2012/07/12/computed-goto-for-efficient-dispatch-tables

LIBXSMM Backend: Code Generation (cont.)

Quick facts about LIBXSMM’s in-memory JIT code generation

- No intermediate representation
- No automatic register allocation
- No (compiler-)optimizations

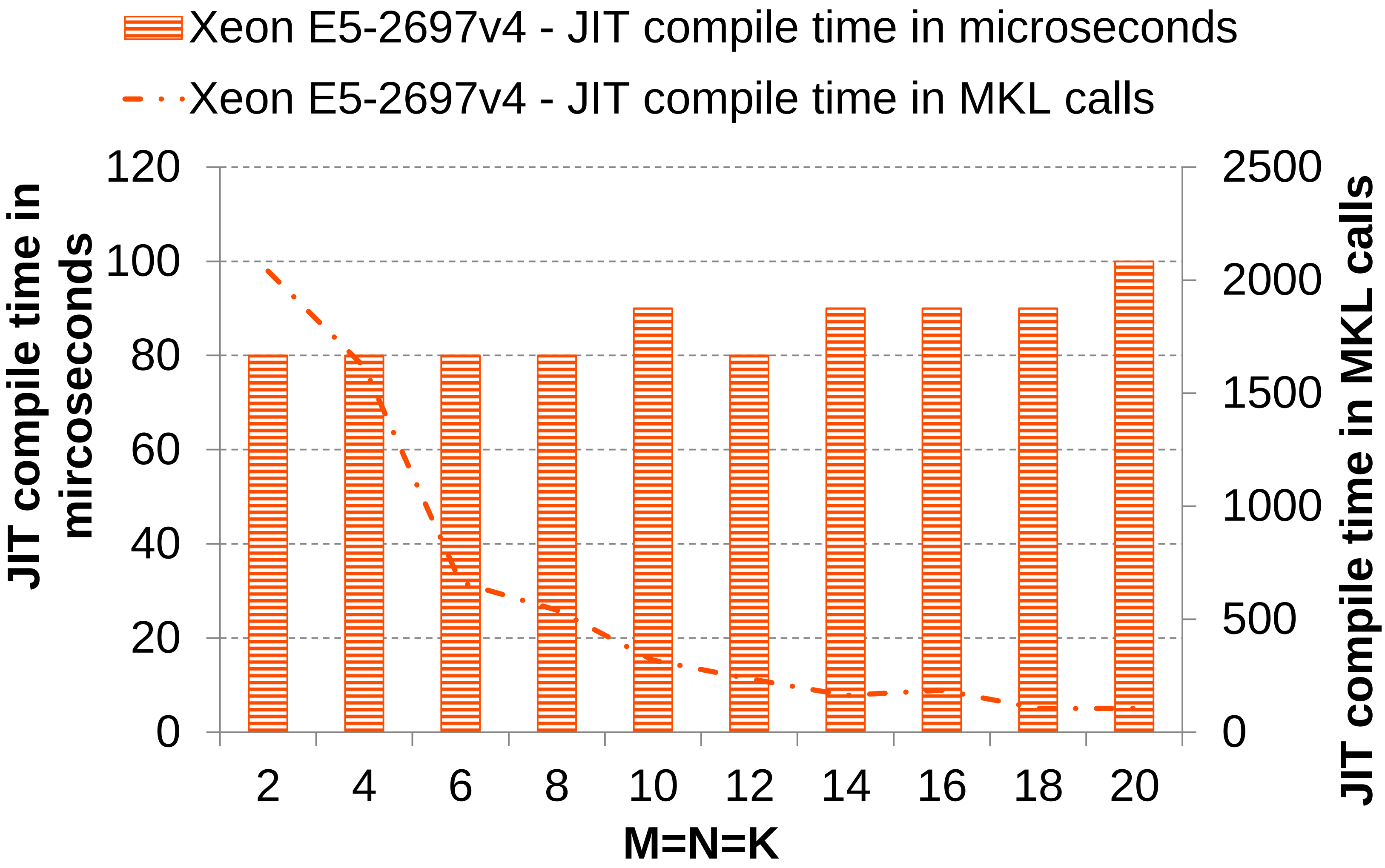
“JIT assembler”

What is the advantage of JIT code?

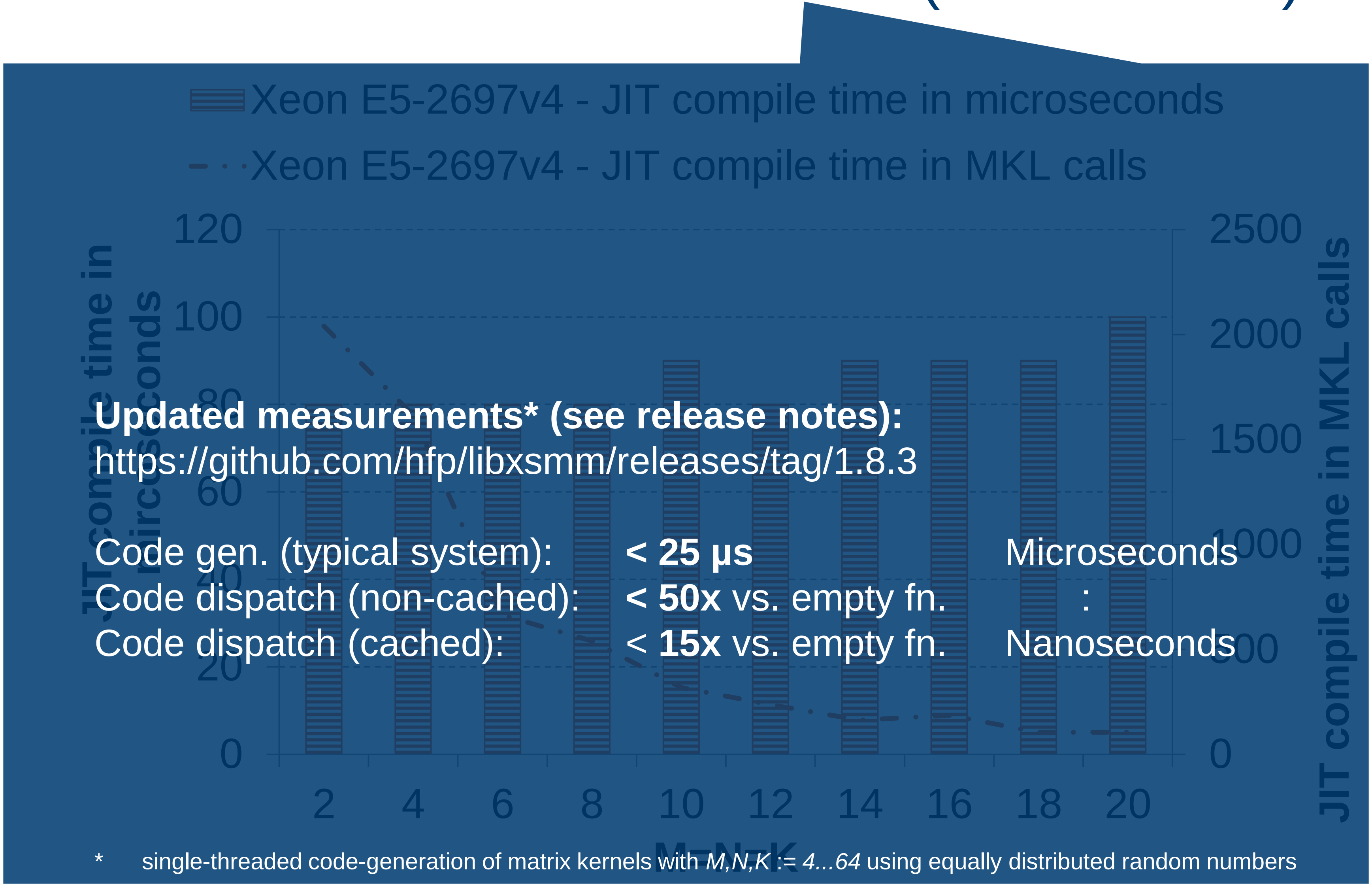
- It is able to leverage instruction variants/immediates to hardcode runtime knowledge (hard to statically compile equivalent code!)

  Example: hard-coded stride for load instruction address (e.g., broadcast-ld.)
- Why is there a particular focus on AVX-512? There is a lot of potential in the instruction set e.g., EVEX may also encode certain values into instruction
- All cases can be captured (avoids upfront code-generation and dispatch)

LIBXSMM Backend: JIT Overhead (incl. OS calls)



LIBXSMM Backend: JIT Overhead (incl. OS calls)

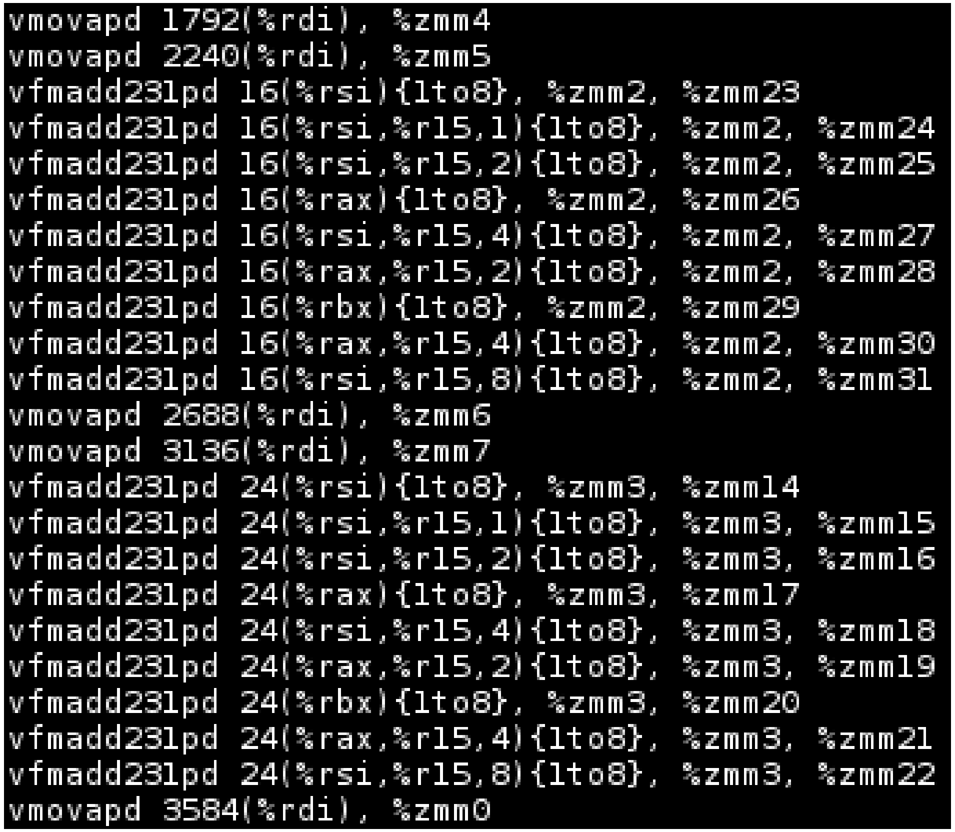


Updated measurements* (see release notes):
https://github.com/hfp/libxsmm/releases/tag/1.8.3

Code gen. (typical system):	**< 25 µs**	Microseconds
Code dispatch (non-cached):	**< 50x** vs. empty fn.	:
Code dispatch (cached):	< **15x** vs. empty fn.	Nanoseconds

* single-threaded code-generation of matrix kernels with *M,N,K* := *4...64* using equally distributed random numbers

LIBXSMM AVX512 code for N=9



```
vmovapd 1792(%rdi), %zmm4
vmovapd 2240(%rdi), %zmm5
vfmadd231pd 16(%rsi){1to8}, %zmm2, %zmm23
vfmadd231pd 16(%rsi,%r15,1){1to8}, %zmm2, %zmm24
vfmadd231pd 16(%rsi,%r15,2){1to8}, %zmm2, %zmm25
vfmadd231pd 16(%rax){1to8}, %zmm2, %zmm26
vfmadd231pd 16(%rsi,%r15,4){1to8}, %zmm2, %zmm27
vfmadd231pd 16(%rax,%r15,2){1to8}, %zmm2, %zmm28
vfmadd231pd 16(%rbx){1to8}, %zmm2, %zmm29
vfmadd231pd 16(%rax,%r15,4){1to8}, %zmm2, %zmm30
vfmadd231pd 16(%rsi,%r15,8){1to8}, %zmm2, %zmm31
vmovapd 2688(%rdi), %zmm6
vmovapd 3136(%rdi), %zmm7
vfmadd231pd 24(%rsi){1to8}, %zmm3, %zmm14
vfmadd231pd 24(%rsi,%r15,1){1to8}, %zmm3, %zmm15
vfmadd231pd 24(%rsi,%r15,2){1to8}, %zmm3, %zmm16
vfmadd231pd 24(%rax){1to8}, %zmm3, %zmm17
vfmadd231pd 24(%rsi,%r15,4){1to8}, %zmm3, %zmm18
vfmadd231pd 24(%rax,%r15,2){1to8}, %zmm3, %zmm19
vfmadd231pd 24(%rbx){1to8}, %zmm3, %zmm20
vfmadd231pd 24(%rax,%r15,4){1to8}, %zmm3, %zmm21
vfmadd231pd 24(%rsi,%r15,8){1to8}, %zmm3, %zmm22
vmovapd 3584(%rdi), %zmm0
```

→ **Max. theoretical efficiency: 90%!**

- Column-major storage; working on all 9 columns and 8 rows simultaneously
- Loads to A (vmovapd) are spaced out to cover L1$ misses; K-loop is fully unrolled
- B-elements are broadcasted within the FMA instruction to save execution slots (SIB)
- SIB addressing mode to keep instruction size <= 8 byte for 2 decodes per cycle (16 byte I-fetch per cycle)
- Multiple accumulators (zmm31-xmm23 and zmm22-zmm14) for hiding FMA latencies

The “Ninja Performance Gap”

Motivation “Architectural insight rather than Autotuning.”

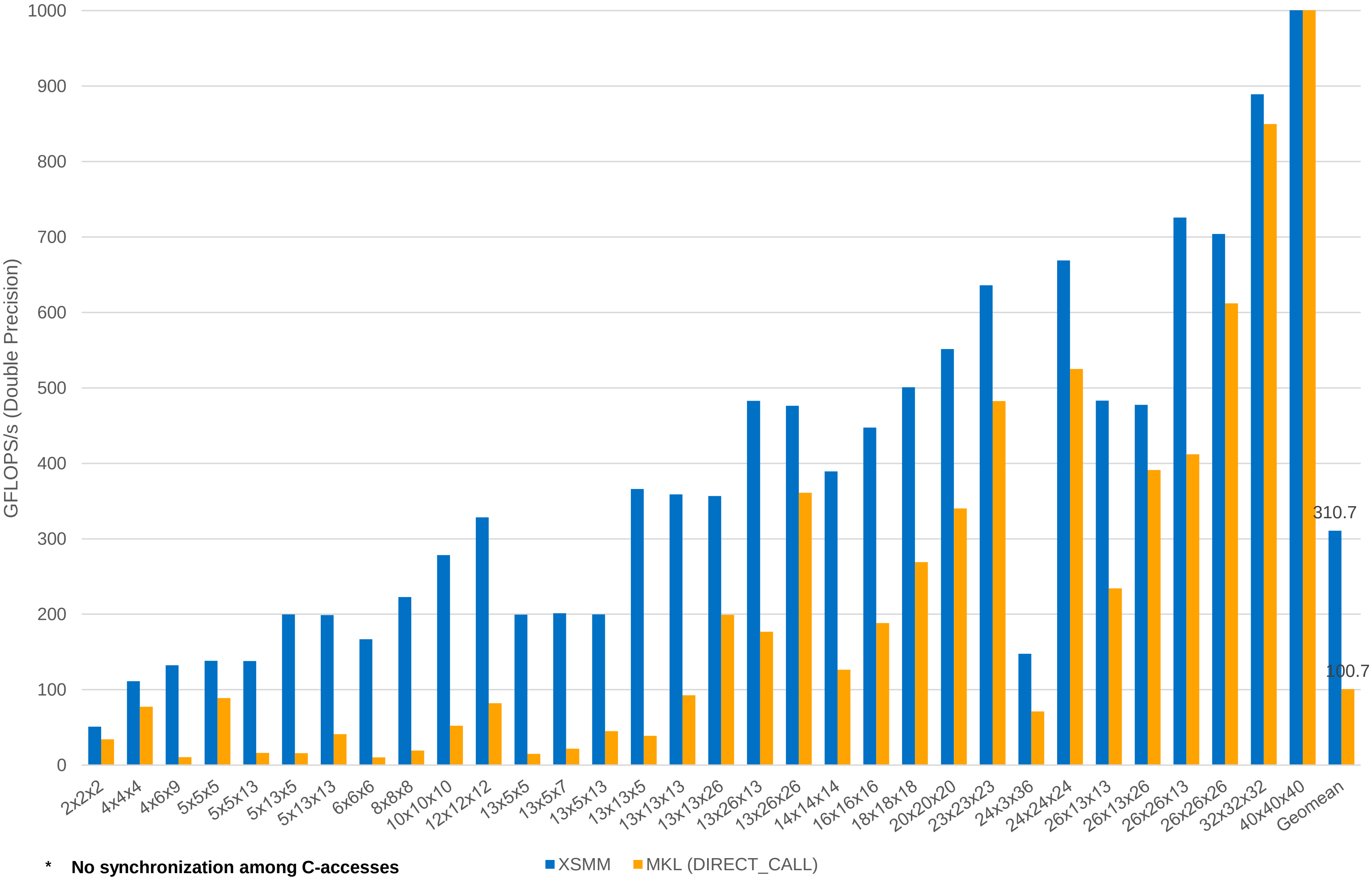
- Example for successful auto-tuning: CP2K (libsmm, libcusmm)
  → Large amount of code generated upfront (thousands of kernels)
    - CP2K‘s libsmm collects best GEMMs from various libraries; build-run-and-select process can take many CPU-hours (cluster job); last version (Haswell-only) ended up with more than 85% of the kernels taken from LIBXSMM
- Architectural insight with focus on AVX-512 instruction family
  - Insight is sourced from ahead-of-release HW knowledge (R&D) e.g., may be sourced from a “cycle accurate simulator”
  - Exploiting the register file to the maximum extent
  - Knowing about (x86-)addressing modes (SIB)
  → LIBXSMM

Ninja Gap

- Knowing about what’s possible, but also what’s left on the table…

LIBXSMM vs. Intel MKL-2017u5 (OpenMP loop / DGEMM)

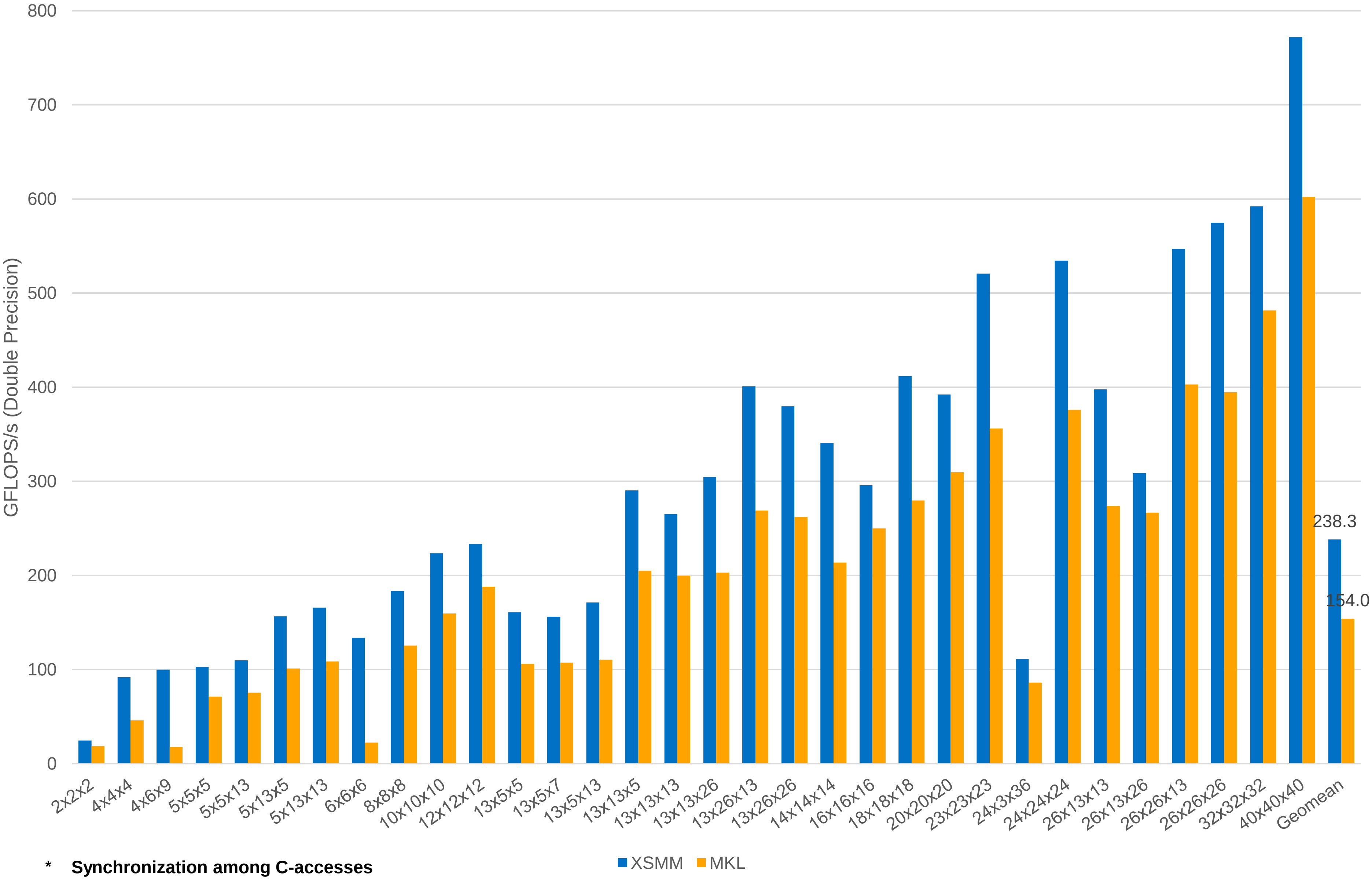
2 GB stream of A and B matrices* (C += A x B) on Intel Xeon-SP Platinum-8180



* **No synchronization among C-accesses**

LIBXSMM vs. Intel MKL-2017u5 (Parallel Batch Interface)

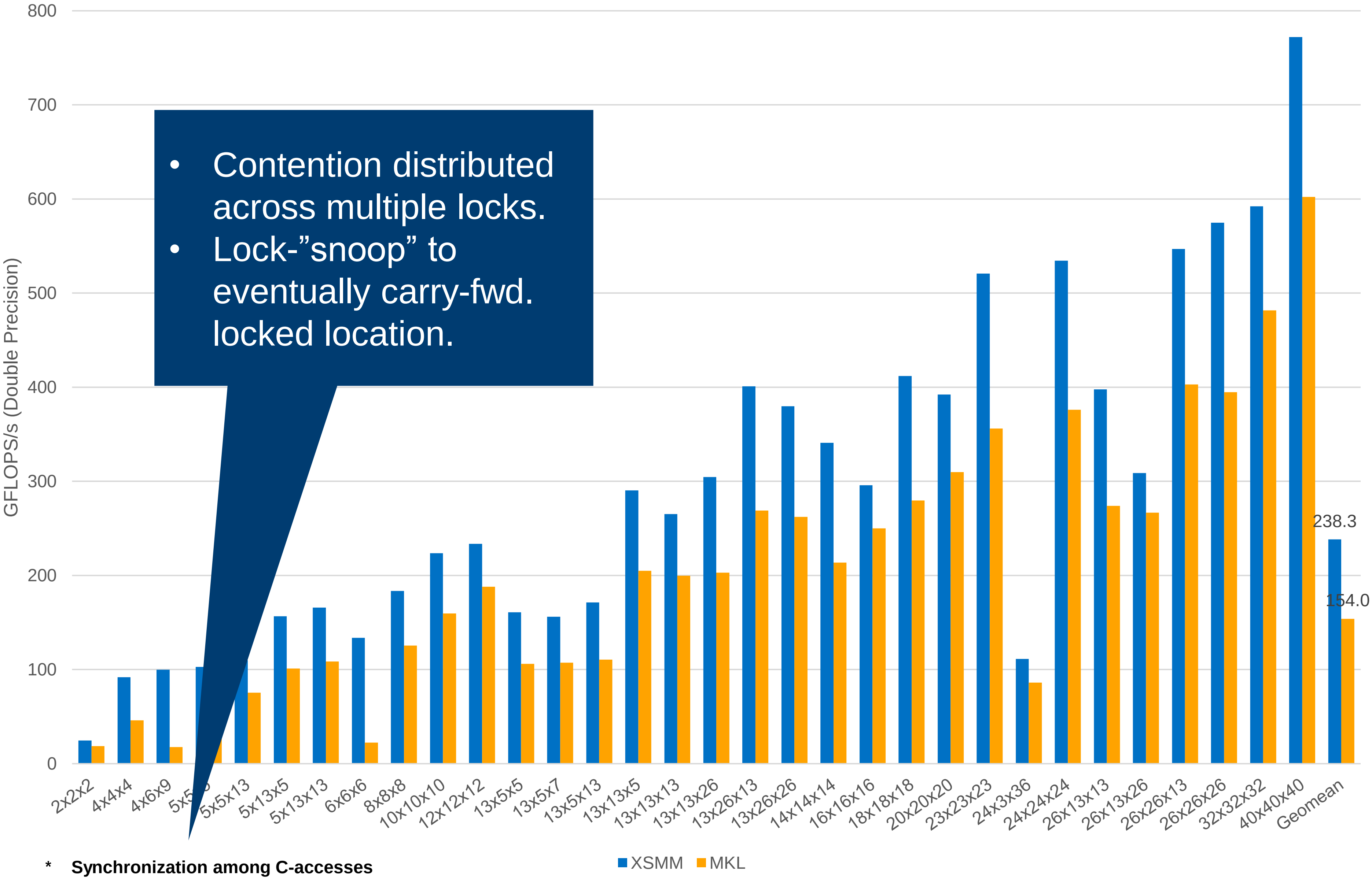
2 GB stream of A and B matrices* (C += A x B) on Intel Xeon-SP Platinum-8180



* **Synchronization among C-accesses**

LIBXSMM vs. Intel MKL-2017u5 (Parallel Batch Interface)

2 GB stream of A and B matrices* (C += A x B) on Intel Xeon-SP Platinum-8180



- Contention distributed across multiple locks.
- Lock-”snoop” to eventually carry-fwd. locked location.

* **Synchronization among C-accesses**

LIBXSMM: Applications HPC

[1] https://cp2k.org/: Open Source Molecular Dynamics with its DBCSR component processing batches of small matrix multiplications ("matrix stacks") out of a problem-specific distributed block-sparse matrix. Starting with CP2K 3.0, LIBXSMM can be used to substitute CP2K's 'libsmm' library. Prior to CP2K 3.0, only the Intel-branch of CP2K integrated LIBXSMM (see https://github.com/hfp/libxsmm/raw/master/documentation/cp2k.pdf).

[2] https://github.com/SeisSol/SeisSol/: SeisSol is one of the leading codes for earthquake scenarios, for simulating dynamic rupture processes. LIBXSMM provides highly optimized assembly kernels which form the computational back-bone of SeisSol (see https://github.com/TUM-I5/seissol_kernels/).

[3] https://github.com/NekBox/NekBox: NekBox is a highly scalable and portable spectral element code, which is inspired by the Nek5000 code. NekBox is specialized for box geometries, and intended for prototyping new methods as well as leveraging FORTRAN beyond the FORTRAN 77 standard. LIBXSMM can be used to substitute the MXM_STD code. Please also note LIBXSMM's NekBox reproducer.

[4] https://github.com/Nek5000/Nek5000: Nek5000 is the open-source, highly-scalable, always-portable spectral element code from https://nek5000.mcs.anl.gov/. The development branch of the Nek5000 code incorporates LIBXSMM.

[5] http://pyfr.org/: PyFR is an open-source Python based framework for solving advection-diffusion type problems on streaming architectures using the flux reconstruction approach. PyFR 1.6.0 optionally incorporates LIBXSMM as a matrix multiplication provider for the OpenMP backend. Please also note LIBXSMM's PyFR-related code sample.

[6] http://dial3343.org/about/: The Extreme-scale Discontinuous Galerkin Environment (EDGE) is a solver for hyperbolic partial differential equations with emphasis on seismic simulations. EDGE optionally uses LIBXSMM, but highly recommends the library due to severe performance-limitations of the vanilla kernels.

Accelerating Eigen Math Library for Automated Driving Workloads

Steena Monteiro *Gaurav Bansal*

Automated Driving Engineering, Software & Services Group

Intel Corporation, Santa Clara, California *Intel Corporation, Hillsboro, Oregon*

Steena.Monteiro@intel.com *Gaurav2.Bansal@intel.com*

Small Matrix-Matrix Multiplication in Automated Driving Workloads—Extended Kalman Filter

Popular automated-driving workloads such as Extended Kalman Filter (EKF) comprise several small DGEMM operations.

EKF fuses RADAR- and LIDAR- sensor data and localizes tracked objects.

EKF comprises two stages:

Prediction

$X' = F * x + u$

$P' = F * P * F^T + Q$

Measurement update

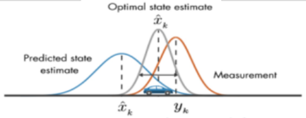
$y = z - H' * x$

$S = H * P' * H^T + R$

$K = P' * H^T * S^{-1}$

$x = x' + K * y$

$P = (I - K * H) * P'$



EKF for sensor fusion

EKF operates on matrices with specific dimensions:

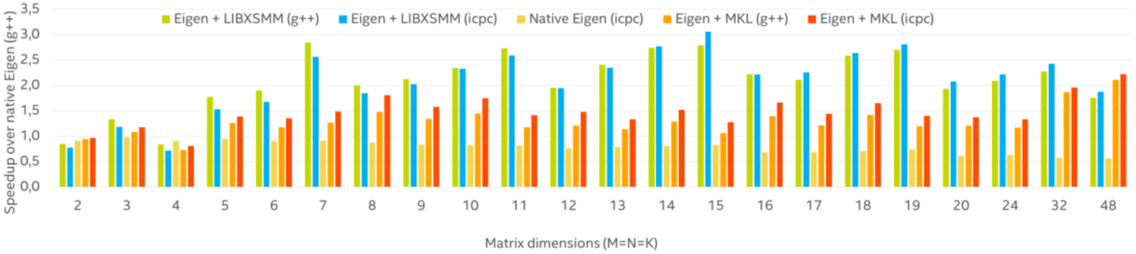
Matrix	rows	columns	Matrix	rows	columns
x	4	1	P	4	4
F	4	4	H	2 or 3	4
R	2 or 3	2 or 3			

Automated-driving community typically uses Eigen, a C++ math library, for DGEMMs on small matrices.

We benchmark DGEMM using Eigen, Eigen with Intel MKL, and Eigen with LIBXSMM with GNU and Intel compilers on Intel Xeon Gold 6148 CPU @ 2.40GHz.

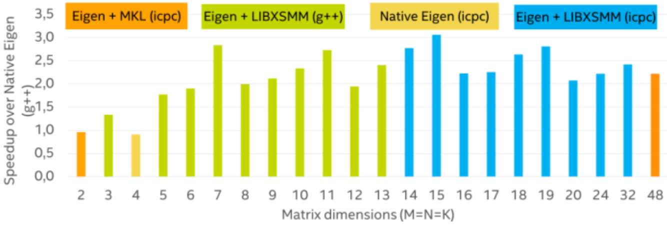
We accelerate EKF that uses native Eigen in its implementation by using MKL and LIBXSMM through Eigen.

Benchmarking DGEMM in Eigen, Eigen + Intel® MKL, and Eigen + LIBXSMM

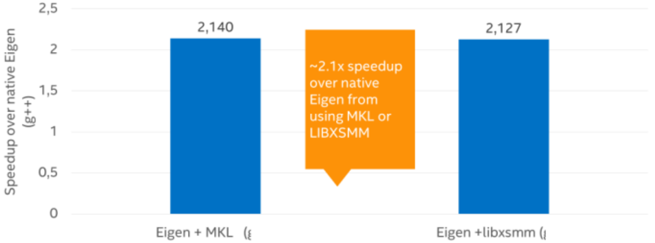


Self-driving car image source: http://blogs.intel.com/iot/files/2016/01/connectedCar2.png
Extended Kalman filter image: Udacity Self-Driving Nano Degree course

Best Eigen Variant per Matrix Dimension



Accelerating the Extended Kalman Filter



~2.1x speedup over native Eigen from using MKL or LIBXSMM

* https://software.seek.intel.com/accelerating-eigen-math-library

LIB x86

SMM

experience
what's inside

LIBXSMM: Portable Code-Multiversioning and Dispatch, Link-time BLAS-Wrapper, JIT-Profiling

CPUID-dispatched (critical) code paths

Makes LIBXSMM suitable for Linux distributions where the code path (target system) is unpredictable (1 package)

Link-time and Runtime Wrapper

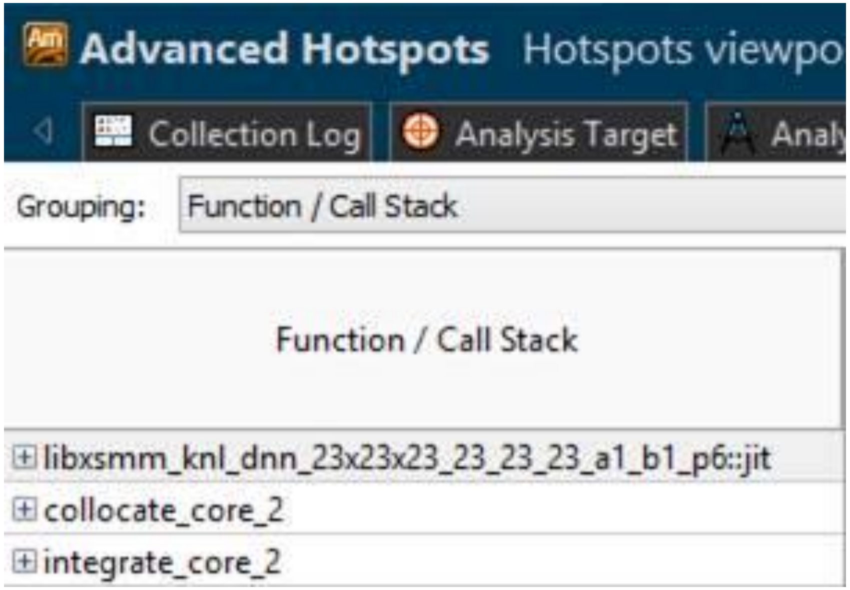
Intercepts existing xGEMM calls at runtime (LD_PRELOAD) or at link-time (LD's --wrap)

JIT Profiling

Support for ***Intel VTune Amplifier*** and ***Linux Perf*** (contributed by Google)

Self-introspection (TRACE)

Ability to print function name of caller, etc. Used to collect advanced statistics



```
⊞ libxsmm_knl_dnn_23x23x23_23_23_23_a1_b1_p6::jit
⊞ collocate_core_2
⊞ integrate_core_2
```

libxsmm_hsw_dnn_23x23x23_23_23_23_a1_b1_p0::jit

- Encodes an Intel AVX-512 (“knl") double-precision kernel ("d") which is multiplying matrices without transposing them ("nn"),
- Rest of the name encodes M=N=K=LDA=LDB=LDC=23, Alpha=Beta=1.0 (all similar to GEMM),
- No prefetch strategy ("p0").

Code-Multiversioning and Dispatch

Intel Compiler: automatic and manual dispatch

- Automatic: heuristic selects code worth to be retargeted
- Manual dispatch: still automatically dispatched (CPUID)
  - Allows to write processor-specific code (using intrinsics, or assembly), or allows to compile the same or different code for different targets
  - Mechanism: __declspec(cpu_dispatch(cpuid,cpuid…)

LIBXSMM: uses GCC’s function attributes (also available with Clang, and ICC)

Two groups of compilers can be identified wrt how **Intrinsics** are supported:

1. **Intel, CRAY, Microsoft, and others**: target flag (compiler’s command line) is independent of Intrinsic’s target requirements. In the past: intrinsics were treated like **WYSIWYG**. Intrinsics may be “uplifted” when command line target flag permits (e.g., AVX intrinsics may be recompiled to AVX2 code). Intel Compiler also supports function attributes (see below).
2. **GCC, Clang, and others**: static code path must **match** Intrinsic requirements. In the past (legacy) e.g., AVX code required -mavx. Later: target attribute allows to decorate functions with individual target. Intrinsics may be “uplifted”, however this is prevented by the target attribute (otherwise “AVX intrinsics” are dispatched but may cause an illegal instruction because of being uplifted to AVX2 code!)

Intel Compiler: Manual Dispatch Example

```
#include <stdio.h>
__declspec(cpu_dispatch(generic, core_5th_gen_avx))
void dispatch_func() {};

__declspec(cpu_specific(generic))
void dispatch_func() {
  printf("Code for non-Intel processors\and generic Intel\n");
}

__declspec(cpu_specific(core_5th_gen_avx))
void dispatch_func() {
  printf("Code for 5th generation Intel Core processors goes here\n");
}
int main() {
  dispatch_func();
  printf("Return from dispatch_func\n"); return 0;
}
```

Intel Compiler: Manual Dispatch Example

```
#include <stdio.h>
__declspec(cpu_dispatch(generic, core_5th_gen_a
void dispatch_func() {};

__declspec(cpu_specific(generic))
void dispatch_func() {
  printf("Code for non-Intel process    (and generic Intel\n");
}

__declspec(cpu_specific(core_5th_gen_avx))
void dispatch_func() {
  printf("Code for 5th generation Intel Core processors goes here\n");
}
int main() {
  dispatch_func();
  printf("Return from dispatch_func\n"); return 0;
}
```

One can essentially "overload" function (names) similar to C++ even with C-code!

Portable Code-Multiversioning and Dispatch

Using function attributes for target-specific dispatch (CPUID)

- Target-specific functions must be named different (C code)
- Target-specific functions must be decorated with target
- Dispatch function (using CPUID) calls target-specific versions

LIBXSMM implements portable Multiversioning and Dispatch

- Header-only: no need for target flags on command line;
  still best performance due to specific targets
- Enables easier code contributions (using Intriniscs);
  not all code requires JIT-code gen. (like assembly)
- Optimized Linux package*

* Most distributions do not accept target flags when building a “general package” (i.e., no SSE3, SSE4, AVX/2, or AVX-512 code).

LIBXSMM: Scratch Memory Allocation

Maybe useful for internal/special data formats

- Best possible code may need a specific data format. For example:
  - BLAS(-like) functions using an internal “compact format”.
  - Convolutions: NCHW (channel first) vs. NHWC (channel last).
  - AoS to SoA conversion (improved vectorization).
- Format-conversion shall be relatively rare, however copy-in (and copy-out) can be parallelized.
- Internal buffers may be required → “scratch memory”

Problems

- Frequent allocations may be unsuitable
- Thread-safety is burden (malloc/free)

LIBXSMM: Scratch Memory Allocation (cont.)

Scope oriented allocators are very fast

- Scratch memory buffer is kept around (no free) and allocation only atomically increments a “head” pointer (no thread-safe free list, etc.)
- Watermark only grows perhaps doubling the buffer size (if insufficient).
- Severe limitations: only **one consumer at a time** can use the allocator.

LIBXSMM: scope-oriented scratch allocator with “allocation sites”

- An allocation site might be a different thread or a different scope
- Different scopes usually have different lifetimes of req. buffer

Main-problem solved: different lifetimes usually prevent growing the scratch needed to serve a new site (at least one consumer still uses the scratch).

LIBXSMM: Scratch Memory Allocation (cont.)

How to interpret the performance results:

- OS impact can be high (different GLIBC versions; even settings such as huge pages served or not impact the memory allocation performance).
- Results are **not** valid for comparison between different CPUs, but rather to compare plain Malloc/free and LIBXSMM's scratch memory allocation.

LIBXSMM‘s scratch allocation is a specialized strategy, and not a general memory allocation; a high memory consumption (watermark) is possible.

	Concurrency (threads)	Malloc/Free* [kHz]	LIBXSMM [kHz]
Intel i7-4770T (Debian 9.3)	1	188	1057
	2	32	379
	4	8	103
Intel Xeon 8168 (RHEL 7.4)	1	6	99
	2	3	27
	4	2	9

* LIBXSMM_SCRATCH_POOLS=0 ./scratch.sh 100 4 2 (https://github.com/hfp/libxsmm/tree/master/samples/scratch#scratch-memory-allocation-microbenchmark). The best result out of three consecutive runs has been reported (“allocation+free calls/s”).

Transparent Threading and Tasking

Internal OpenMP region

```
libxsmm_dgemm_omp(
        &transa, &transb, &m, &n, &k,
        &alpha, a, &lda, b, &ldb,
        &beta,   c, &ldc);
```

OpenMP Tasking

```
# pragma omp parallel
# pragma omp single nowait
  libxsmm_dgemm_omp(
        &transa, &transb, &m, &n, &k,
        &alpha, a, &lda, b, &ldb,
        &beta,   c, &ldc);
```

Bonus: non-OpenMP threads

```
libxsmm_mmbatch(…,
        int tid, int nthreads);
```

Flow inside of libxsmm_dgemm_omp:

```
If (sufficient-problem-size)

    If (0 == omp_get_active_level())
    … execution is within a parallel region
    … and at top-most nesting level
    … tasks may be assumed.
    Test requires OpenMP 3.0.

    if (0 == omp_in_parallel())
    … execution is within a parallel region
    … still consider cache-blocking.
    Compiler is legacy (no OpenMP 3.0)

Else: use sequential implementation
          (no cache blocking needed)
```

LIBXSMM: References

[1] http://sc17.supercomputing.org/SC17%20Archive/tech_poster/tech_poster_pages/post190.html: Understanding the Performance of Small Convolution Operations for CNN on Intel Architecture (poster and abstract)**.** SC’17: The International Conference for High Performance Computing, Networking, Storage and Analysis, Denver (Colorado).

[2] https://software.intel.com/en-us/articles/intel-xeon-phi-delivers-competitive-performance-for-deep-learning-and-getting-better-fast: Intel Xeon Phi Delivers Competitive Performance For Deep Learning - And Getting Better Fast. Article mentioning LIBXSMM's performance of convolution kernels with DeepBench. Intel Corporation, 2016.

[3] http://sc16.supercomputing.org/presentation/?id=pap364&sess=sess153: LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation (paper). SC'16: The International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City (Utah).

[4] http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post137.html: LIBXSMM: A High Performance Library for Small Matrix Multiplications (poster and abstract). SC'15: The International Conference for High Performance Computing, Networking, Storage and Analysis, Austin (Texas).

intel®

experience
what's inside™