LIBXSMM

LIBRARY TARGETING INTEL ARCHITECTURE (X86)

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

Hans Pabst
Intel High Performance and Throughput Computing
Switzerland

Alexander Heinecke
Parallel Computing Lab
Intel Labs
USA

Greg Henry
SSG Pathfinding
USA

November 28th 2016

https://github.com/hfp/libxsmm

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.

Copyright © 2015, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

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

Motivation – Why Improving Small GEMMs Speed

Many applications rely on many and (in)dependent small LA / GEMM operations:

- Element-local operations for high-order methods:
  - Element updates in DG
  - Tensor contractions in SEM
- Block Sparse matrices in Quantum Chemistry
- Skewed GEMMs in machine learning
- Tiling of mediums sized calls
- H-matrices
- Deep Learning

→ Batch GEMM operations are often not possible, or destroy optimal cache usage on CPUs.

→ We need the best possible small GEMM routine to speed-up these codes to push our application to the right in the roofline model

→ For CPUs this is LIBXSMM

LIBXSMM: Overview

Highly efficient Frontend

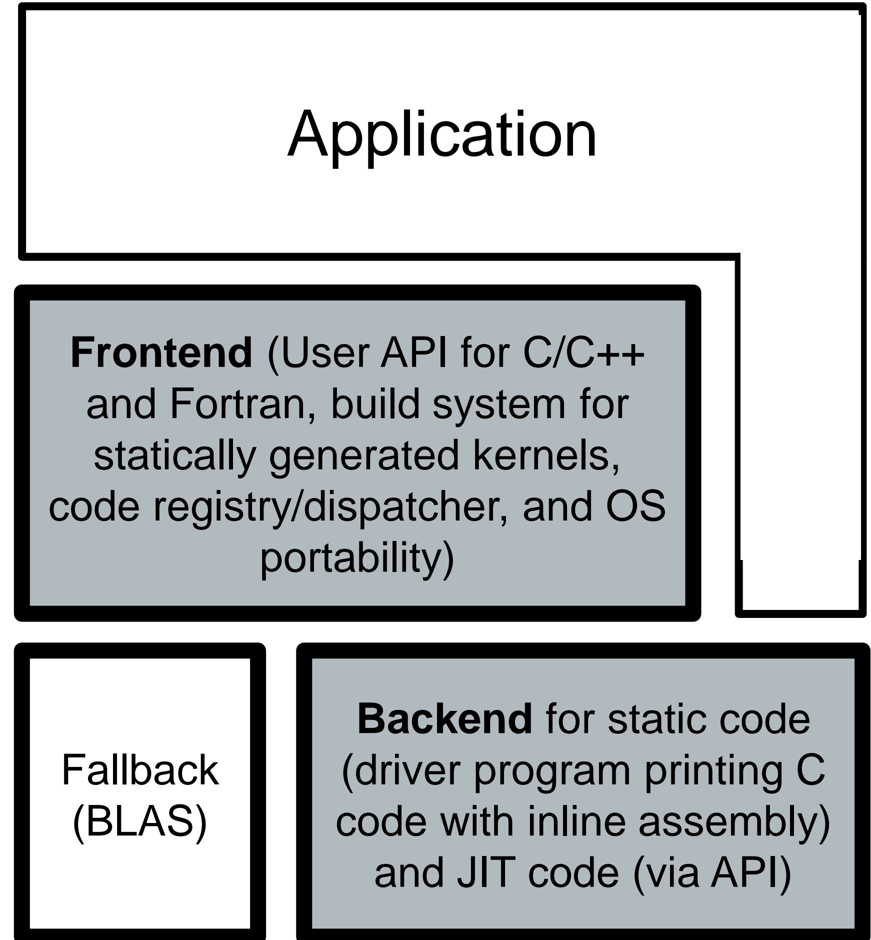
- BLAS compatible (DGEMM, SGEMM) including LD_PRELOAD
- Support for F77, C89/C99, F2003, C++
- Zero-overhead calls into assembly
- Two-level code cache

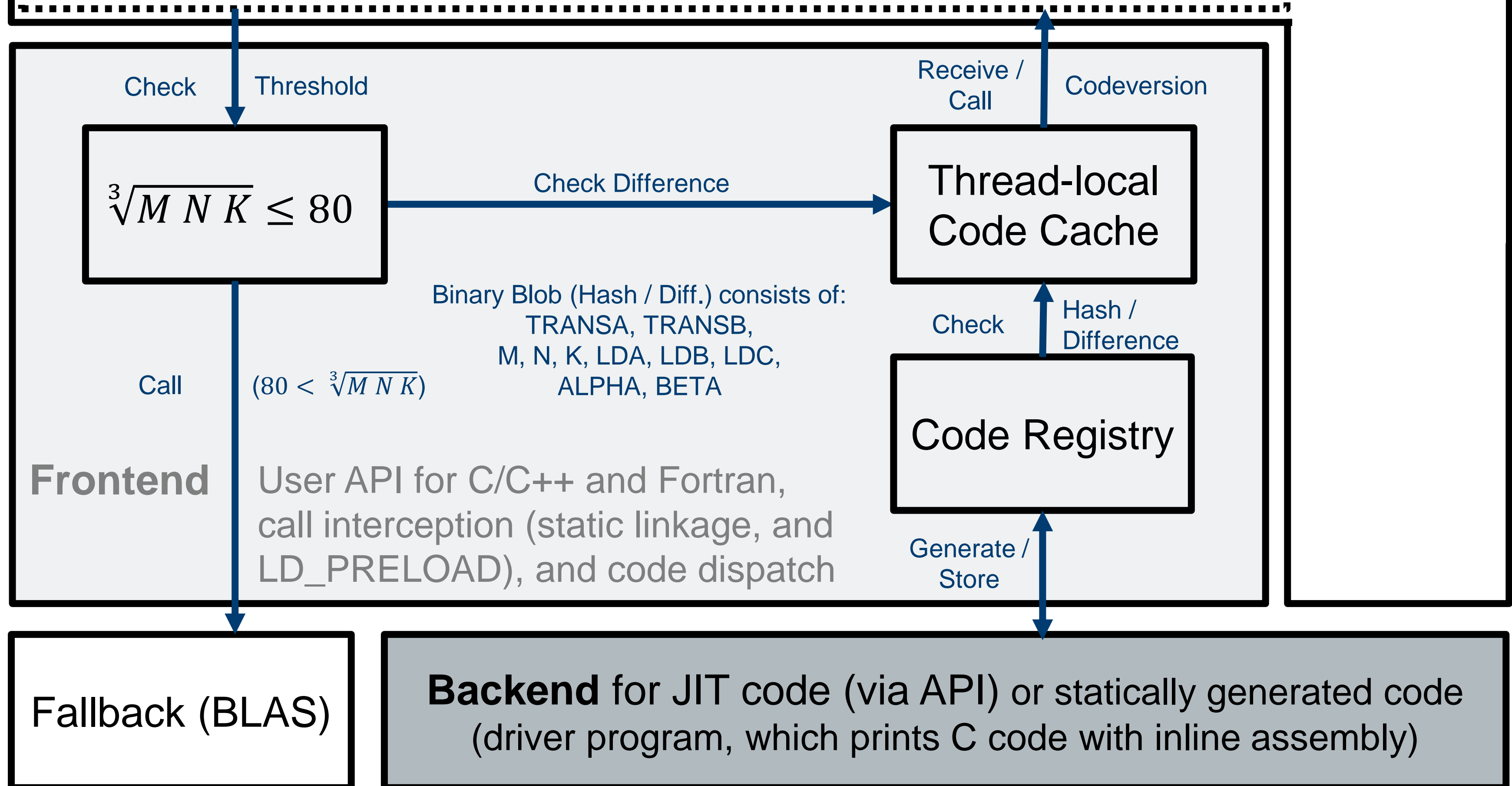
Code Generator

- Supports all Intel Architectures since 2005, special 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 instruction based on basic blocks
- Very fast as no compilation is involved





Application
GEMM
Check
Threshold
Receive / Call
Codeversion
$\sqrt[3]{M\ N\ K} \leq 80$
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 (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 */
    }
  }
}
```

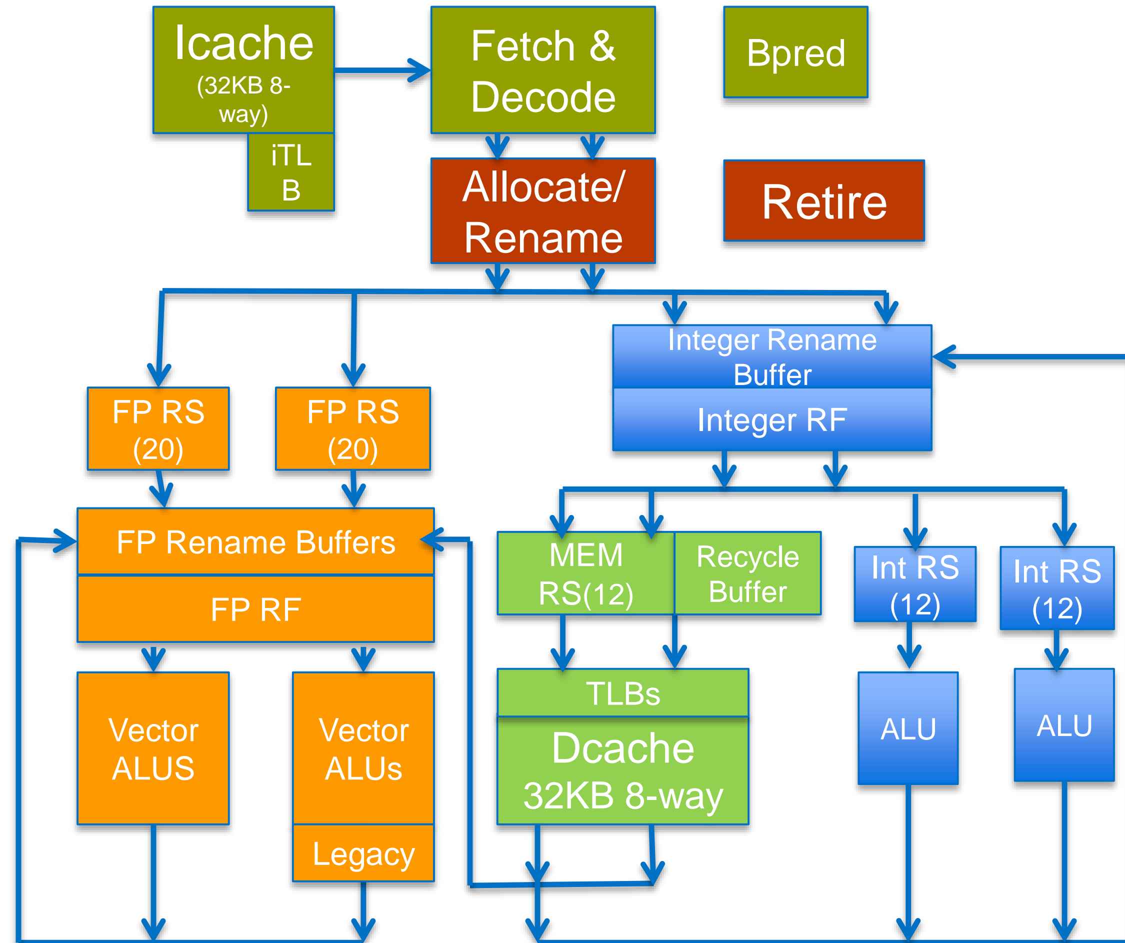
Intel Xeon Phi 72xx Core & VPU

Balanced power efficiency, single thread performance and parallel performance

2-wide Out-of-order core

4 SMT threads

- 72 in-flight instructions.
- 6-wide execution
- 64 SP and 32 DP Flop/cycle
- Dual ported DL1 → to feed 2 VPU
- Two-level TLB. Large page support
- Gather/Scatter engine
- Unaligned load/store support
- Core resources **shared** or **dynamically repartitioned** between active threads
- General purpose IA core



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, and placeholder 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")

LIBXSMM Backend: Code Generation (cont.)

Quick facts about in-memory JIT code generation (JIT assembler)

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

What is the advantage of JIT code?

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

  Example: hard-coded stride for load instruction address (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

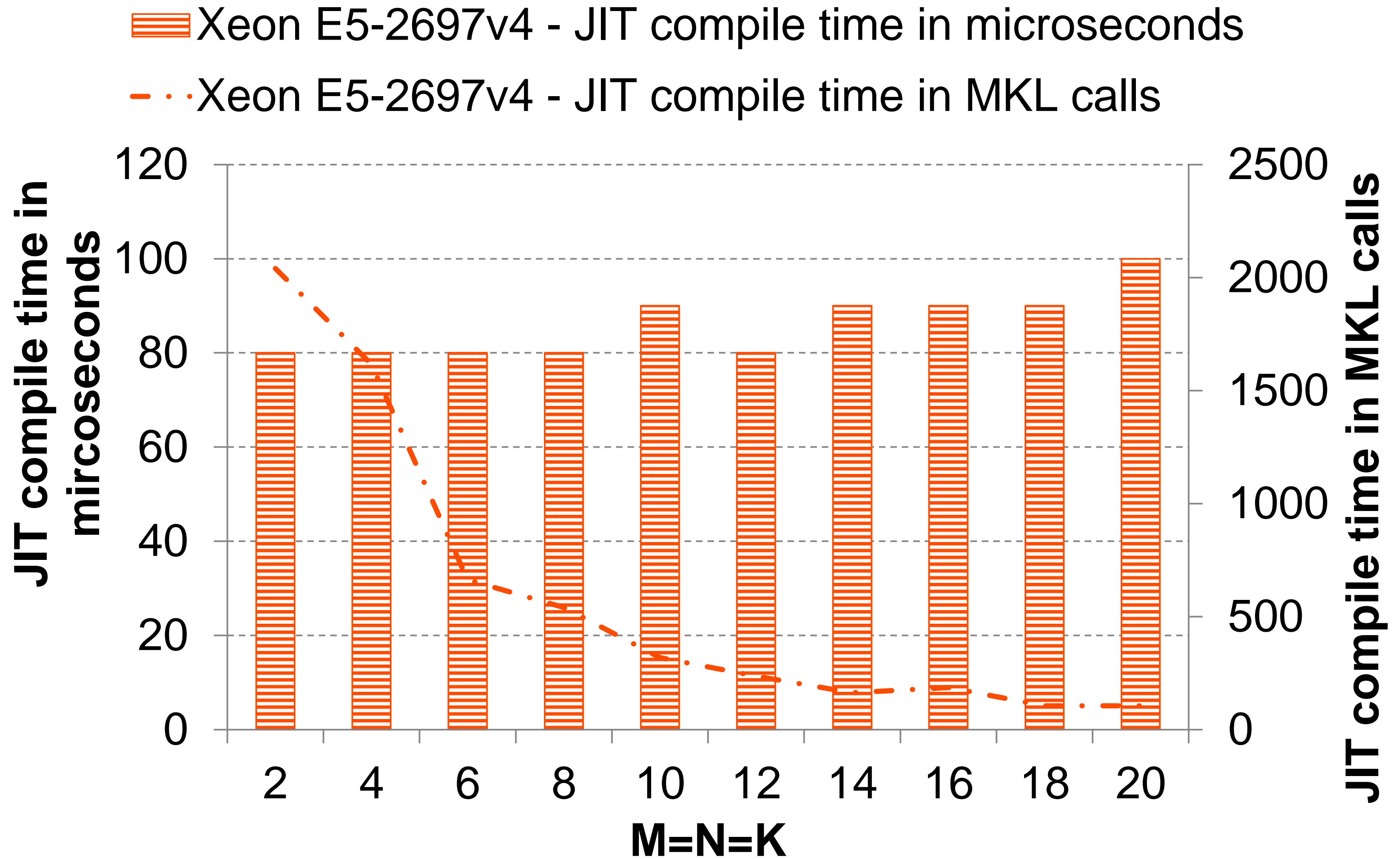
LIBXSMM AVX512 code for N=9

```
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
```

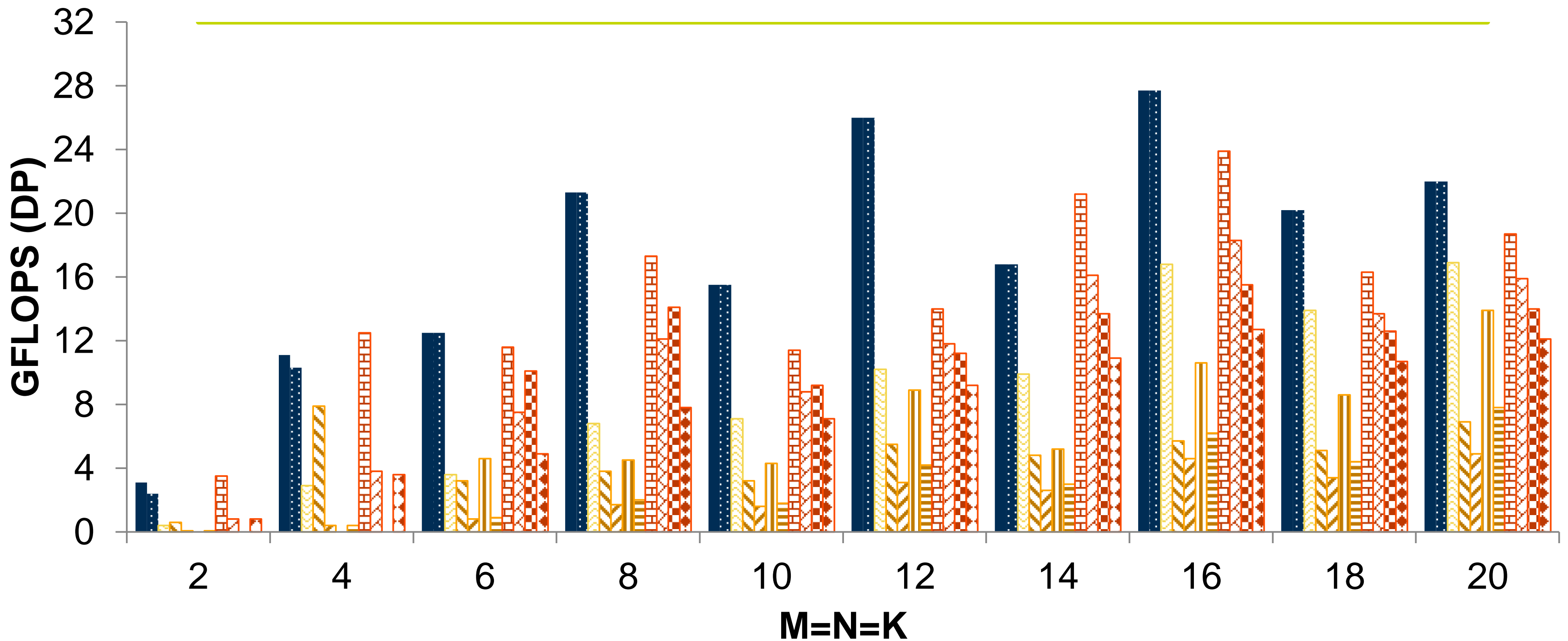
→ **Max. theoretical efficiency: 90%!**

- 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 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

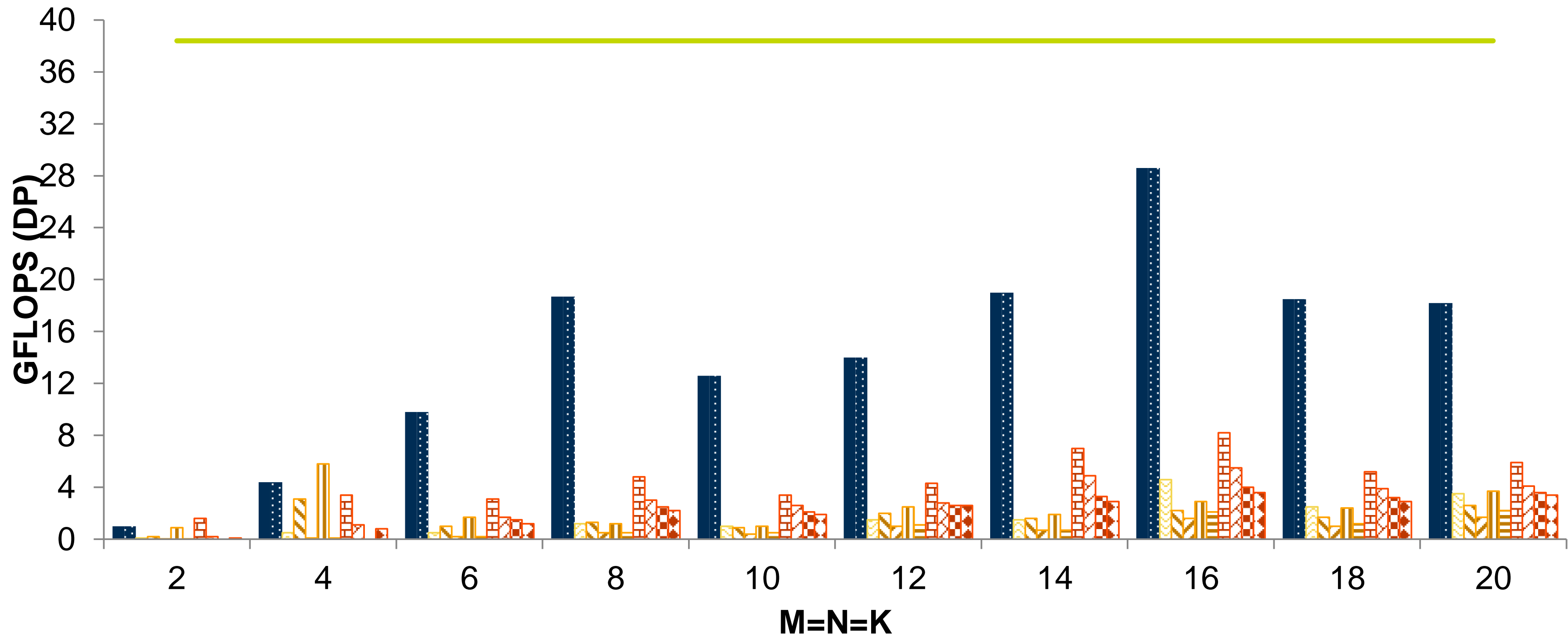
LIBXSMM Backend: JIT Overhead (incl. OS calls)



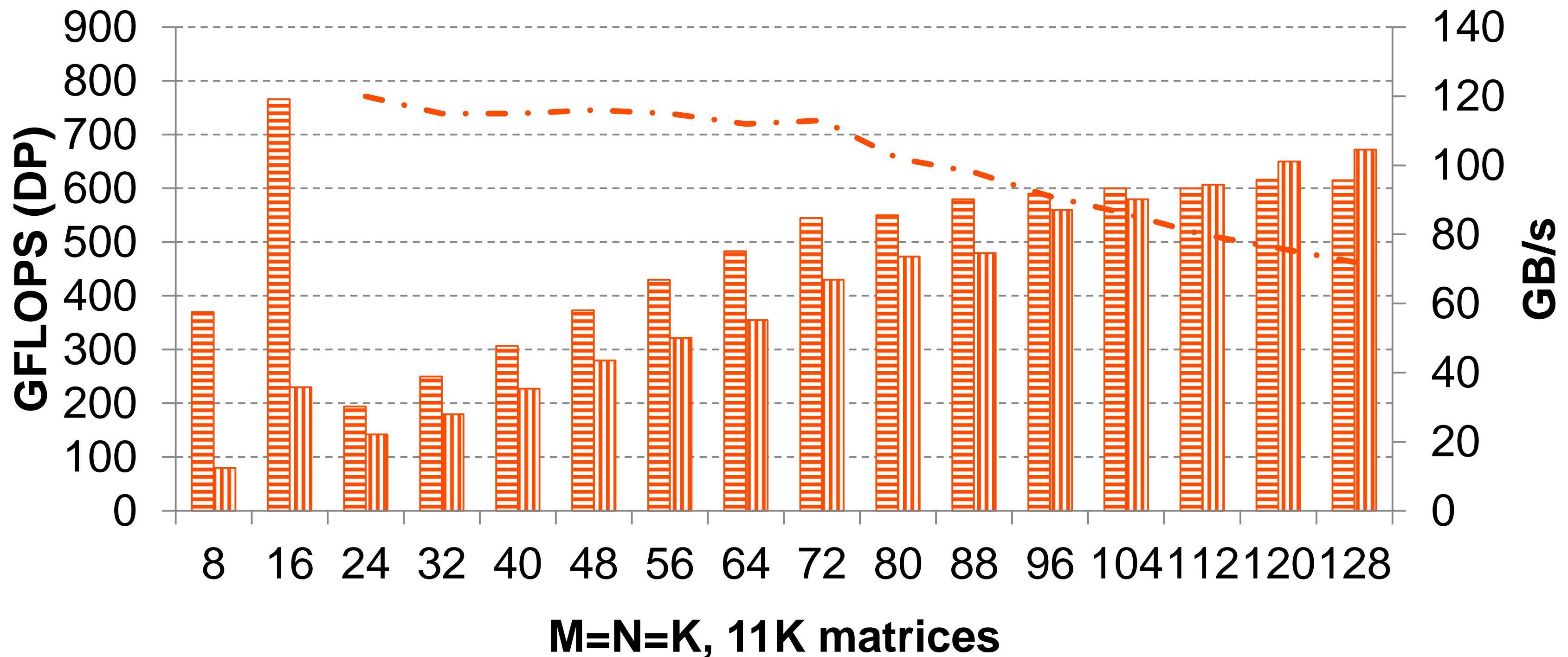
LIBXSMM Performance on 1c Xeon E5-2697v4 (BDX)



LIBXSMM Performance on 1c Xeon Phi 7250 (KNL)



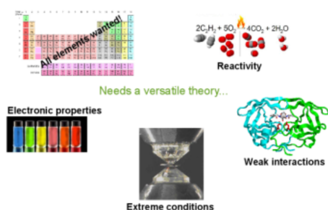
LIBXSMM vs. MKL DGEMM_BATCH 2x Xeon E5-2697v4 (BDX)



CP2K Small Matrix Multiplications

CP2K implements DFT* (Density Functional Theory) method

* among other methods

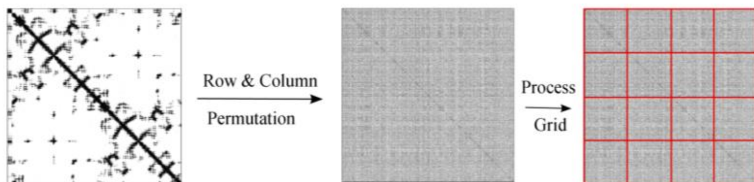


$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V_s(\vec{r})\right]\phi_i(\vec{r}) = \epsilon_i\phi_i(\vec{r})$$

$$V_s(\vec{r}) = V(\vec{r}) + \int \frac{e^2 n_s(\vec{r}\,')}{|\vec{r}-\vec{r}\,'|}\mathrm{d}^3r' + V_{\mathrm{XC}}[n_s(\vec{r})]$$

(Nobel Prize 1998: Walter Kohn and John A. Pople)

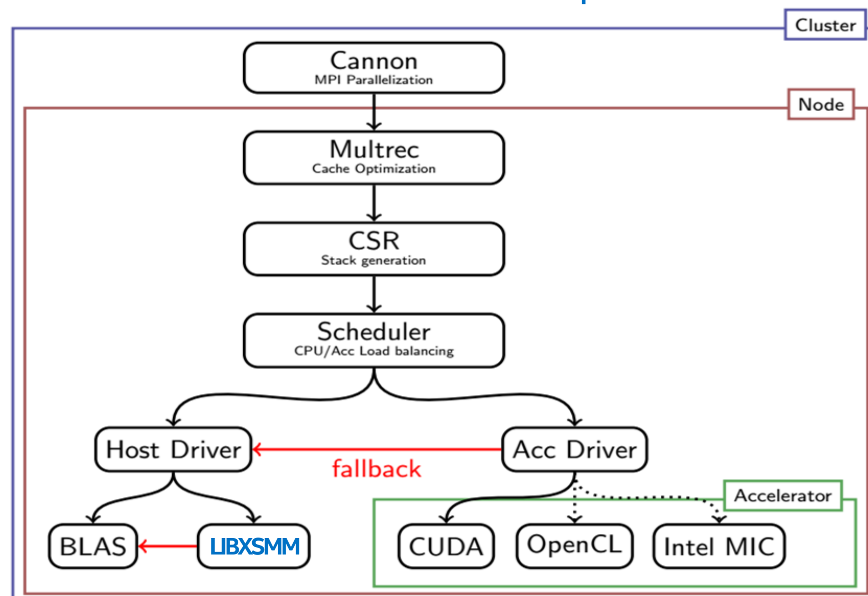
- DFT as gen. Eigenvalue problem solved via Self-Consistent Field (SCF) iterative method
- Sparsity can be exploited, and ends up with small dense blocks of natural structure (atoms)



- Recent CPUs (FMA) are doing very well (no GPU speedup)

* Pictures adapted from Speedup 2012 (Joost VandeVondele)

Distributed **B**locked **C**ompressed **S**parse **R**ow
Distributed **B**locked **C**annon **S**parse **R**ecursive

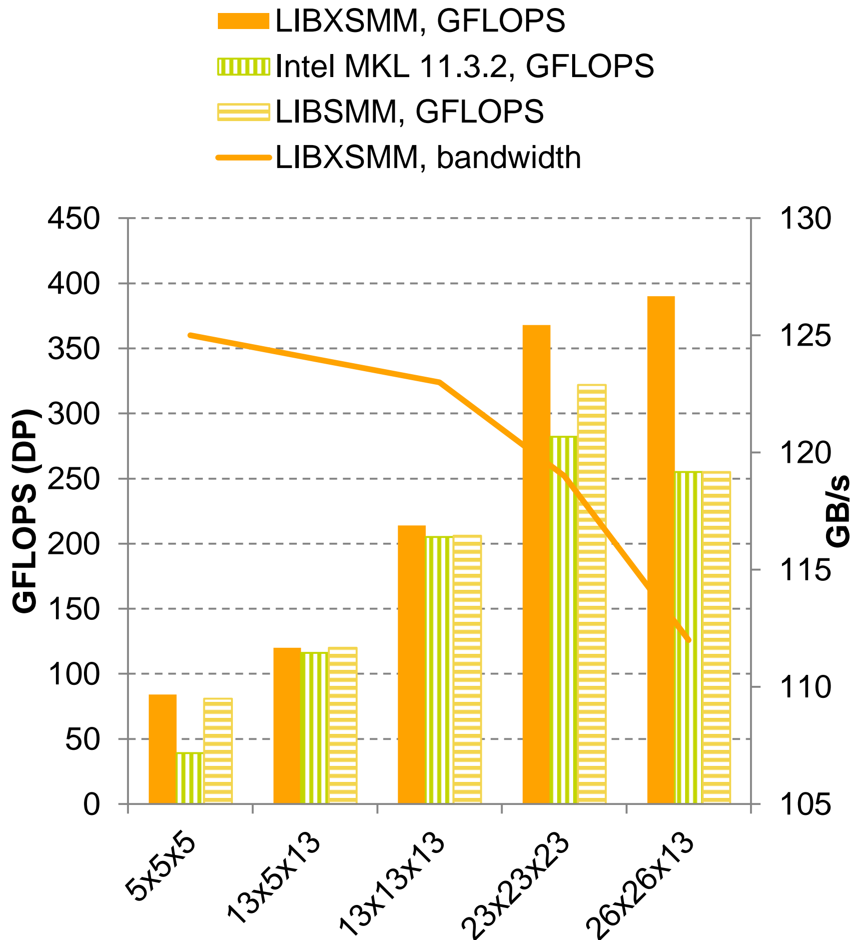


- DBCSR* is ubiquitously used by almost all algorithms in CP2K (not only for DFT)
- DBCSR generates matrix batches ("stacks") of ~1000 small matrices: C += A * B (accumulation)

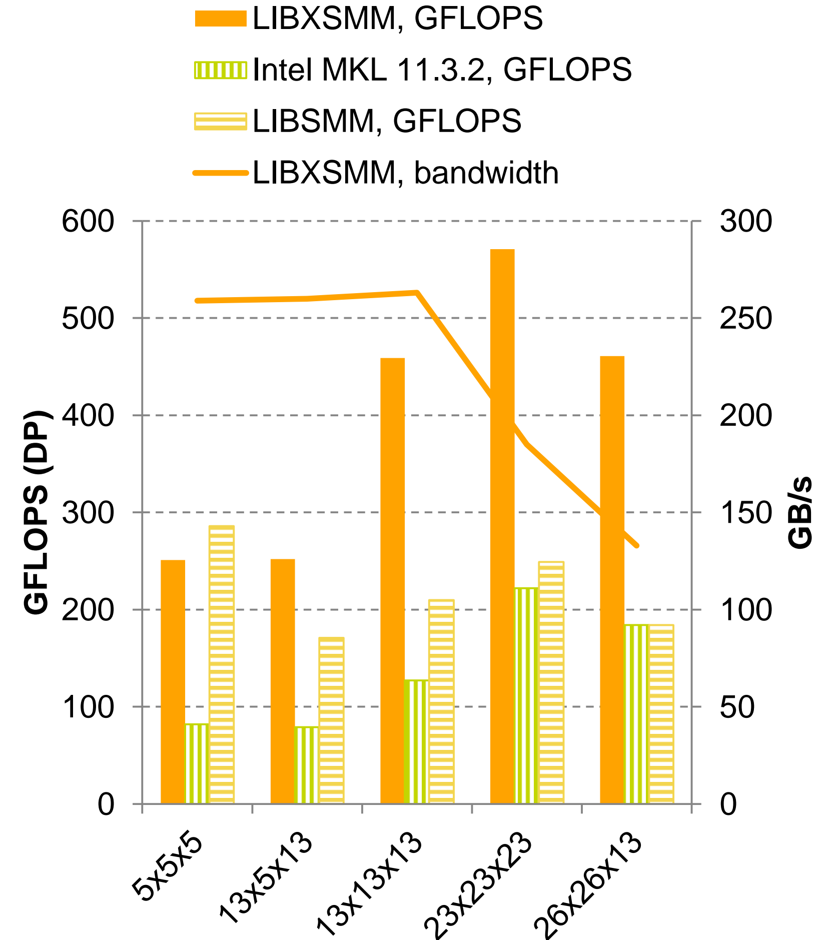
* CP2K's sparse matrix library: https://dbcsr.cp2k.org/

CP2K Kernel Performance for DBCSR Library

Dual Socket Xeon E5-2697v4



Single Socket Xeon Phi 7250



Conclusion

- Small GEMMs are an important kernel in real science applications
- They could be batched, but for best performance leveraging CPU features such as fast caches is essential
- Using LIBXSMM, we showed several factors in performance for small GEMMs compared to vendor BLAS or compile-time optimized libraries at kernel level
- At full application level and at scale up an to 50% performance increase is possible, this is even much higher in case of Xeon Phi where optimal code is highly important due to the architecture
- Current research is applying the proposed technology to direct convolutions for deep learning (11 input parameters instead of 6), early implementation is already available on GitHub for AVX2 and AVX512.

https://github.com/hfp/libxsmm

intel®
experience
what's inside™