FSSpMDM—Accelerating Small Sparse Matrix Multiplications by Run-Time Code Generation

F.D. Witherden

Department of Ocean Engineering
Texas A&M University
Motivation

• Small matrix multiplications (SMM’s) are a key building block of high-order finite element methods.
Motivation

• Canonical example of this is polynomial evaluation.
Motivation

• The most efficient SMM library for CPUs is libxsmm.

• It employs run-time assembly code generation and has support for AVX2, AVX-512, NEON, and SVE.
Motivation

• However, it is also possible for the operator $A$ to be *sparse*.

• The prototypical example of this are elements with a *tensor-product construction*: quads, hexes, and prisms.
Motivation

• Sparse operators can also arise through the factorisation of dense operators.
Motivation

- Heretofore, the standard approach for handling such sparse operators on CPUs has been GiMMiK.
Motivation

• GiMMiK works by unrolling the operator as a C kernel.

• As such performance is unpredictable and compiler dependent.

• Moreover, for larger operators kernel compilation can take several minutes and require gigabytes of memory.
Approach

• In this talk we consider adding a Fixed Size Sparse Matrix Dense Matrix (FSSpMDM) multiplication routine to libxsmm for performing:

\[ C \leftarrow AB + \beta C, \]

where \( A \) is \( M \) by \( K \) and \( B \) is \( K \) by \( N \) and \( \beta \) is zero or one.
Approach

- $A$ is sparse and invariant.
- The multiplication will be performed repeatedly.
- $B$ and $C$ are stored in row-major order.
- $B$ and $C$ are small enough to reside in L1/L2 cache.
- $N$ is a multiple of the SIMD vector length $v_l$. 
Approach

• Our overall approach is to follow GiMMiK and fully unroll the multiplication, eliding multiplications through by zero.

```python
for j in 0:(N / vl)
    for i in 0:M
        jj = j:(j + vl)
        dp = sparse_dot(A[i, :], B[:, jj])
        C[i, jj] = dp + β*C[i, jj]
```
Approach

• One potential issue with this strategy is that we are only accumulating a single dot product at a time.

• When run on an in-order CPU which can dual issue FMAs with 6 cycle latency (e.g., the original Intel Xeon Phi) we are limited to ~8% of peak FLOPs.
Approach

• Thankfully, all recent CPUs incorporate out-of-order execution and so are able to look ahead in the code stream and find independent dot products to work on.

• However, this only works if a reasonable number of complete dot products fit within the out-of-order execution window.
Approach

• Beyond this, one solution is $n$-blocking where we **unroll and interleave** iterations of the outer loop.

```python
for j in 0:(N / vl)
    for i in 0:M
        jj = j:(j + vl)
        dp = sparse_dot(A[i, :], B[:, jj])
        C[i, jj] = dp + β*C[i, jj]
```
Approach

- This provides a simple means of doubling or quadrupling the number of dot products in flight.

- However, it also results in a commensurate increase in code size.

- Furthermore, $N$ must now be divisible by $nv_l$ where $n$ is the $n$-blocking factor.
Approach

• A better solution is $m$-blocking where we **interleave the iterations of the inner loop** together.

```python
for j in 0:(N / vl)
    for i in 0:M
        jj = j:(j + vl)
        dp = sparse_dot(A[i, :], B[:, jj])
        C[i, jj] = dp + β*C[i, jj]
```
Approach

• Since we are just rearranging existing instructions \( M \) blocking not result in an increase in code size.

• However, to be effective we need to group together rows with similar numbers of non-zero entries.

• The best case is grouping rows with identical non-zero structures as we can reuse values from \( B \).
Approach

• The biggest factor when generating a kernel is register allocation.

• Once registers have been allocated synthesising the assembly code itself is a trivial exercise.

• As such, from here on we’ll be focusing on registers.
AVX2 provides us with 16 256-bit vector registers.

We have instructions for evaluating:

\[ c \leftarrow a \cdot b \quad \text{and} \quad c \leftarrow \pm c \pm a \cdot b, \]

where \( a \) and \( c \) are registers and \( b \) is a register or memory operand of the form \([g + \langle \text{imm} \rangle]\) where \( g \) is a general purpose register and \( \text{imm} \) a 32-bit displacement.
x86-64: AVX2

• If we have 15 or fewer unique absolute values in $A$ then the simplest approach is to store one value per register.

• Any spare registers can then be used to facilitate $n$- and/or $m$-blocking.

• We term this the A-in-registers approach.
x86-64: AVX2

Broadcast A value

Loaded B value

C accumulator
x86-64: AVX2

• When the number of unique absolute values exceeds 15 we resort to *storing the unique values in an array.*

• They can then be loaded into a register as needed with the *vbroadcasts[sd]* instruction.

• We term this the *A-in-memory approach.*
x86-64: AVX2

• The idea is to treat all free registers, i.e. those not used for loaded $B$ values or $C$ accumulators, as a cache.

• However, as we have foreknowledge of future dot products we can be strategic about what values we evict.

• Specifically, we overwrite the register with the greatest distance between now and when its value is next used.
x86-64: AVX2

SIMD Register

- **Loaded B value**
- **C accumulator**
- **Temp broadcast A value**
AVX-512 is the most recent extension for x86-64.

It provides us with 32 512-bit vector registers.

This leads to straightforward extensions of the A-in-registers and A-in-memory strategies.
Additionally, the **permutation instructions** in AVX-512 also allow for a new approach: **storing multiple unique absolute values inside a single vector register**.

When a unique value is needed permutation instructions are used to broadcast it to a temporary register.

We term this **A-packed-in-registers**.
x86-64: AVX-512

- With this we can store 8 (double) or 16 (single) values inside a single register.

- However, broadcasting using (vpermd) requires up to 7 (double) or 15 (single) unique permutation constants which must be stored in registers.

- This enables up to 120 unique absolute values.
- Consider executing `vpermd zmm31, zmm8, zmm0`.

\[ \text{zzm31} \quad \text{zzm8} \quad \text{zzm0} \]

- \( \text{zzm0} = 0x4 \)
- \( \text{zzm31} = 0x5 \)
x86-64: AVX-512

SIMD Register

<table>
<thead>
<tr>
<th>u</th>
<th>m</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>42d</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>42d</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>43d</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>49d</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

- Green: Packed A values
- Yellow: Loaded B value
- Red: C accumulator
- Blue: Permutation constant
- Purple: Temp broadcast A value
AARCH64: NEON

• NEON provides us with 32 128-bit vector registers.

• We have instructions for evaluating:

\[ c \leftarrow a \cdot b \quad \text{and} \quad c \leftarrow c \pm a \cdot b, \]

but, being a RISC architecture all arguments must be registers.
AARCH64: NEON

• One unique feature is the availability of **indexed forms**:

\[ c \leftarrow a_i \cdot b \quad \text{and} \quad c \leftarrow c \pm a_i \cdot b, \]

where the suffix \( i \) denotes a specific lane of \( a \) to broadcast in advance of the operation.

• This enables us to implement the **A-packed-in-registers strategy for free**!
AARCH64: NEON

• Loading values of $B$ can be more involved than x86-64 as values are typically limited to 12-bits.

• Thus it is not unusual for a single AVX2/AVX-512 register-memory instruction to translate into four or five AARCH64 instructions: two or three adds to generate the address, then a load, and finally the FMA itself.
AARCH64: NEON

• One means of reducing this overhead is through **paired load instructions** (ldp), which enable us to load **256-bits** at a time.

• This enables us to appreciably increase FMA density and **strongly suggests using** $n = 2$ blocking.
AARCH64: NEON

SIMD Register

- Packed A values
- Loaded B value
- C accumulator
AARCH64: NEON

• Implementing the A-in-memory strategy requires more care for NEON due to restricted immediate displacements.

• Thankfully, NEON has two extremely powerful features which enable a highly efficient implementation.
AARCH64: NEON

• Firstly, using the ld1 instruction it is possible to load a constant into a specific lane of a vector.

• Combined with the aforementioned indexed forms this enables us to cache two (double) or four (single) times as many constants in registers as AVX-512.
Additionally, \texttt{ld1} has a \textit{post-indexed form}.

Let us execute: \texttt{ld1 \{v5.s\}[1], [x5], #4}.
AARCH64: NEON

• Thus, by creating an array in memory which contains values **in the order they are needed** by the kernel (i.e., when values miss in the register cache) we can bring in constants with just a **single four-byte load instruction**.

• Although the array is larger than if we just stored unique values, this is not a problem in practice.
AARCH64: NEON

SIMD Register

- Loaded B value
- C accumulator
- Temp packed A values
AARCH64: SVE

• SVE is a newer vector instruction set for AARCH64.

• Its most notable feature is support for **varying vector lengths** from 128- to 2048-bits.

• The scalable functionality is not too useful for HPC but we can certainly benefit from **longer vectors**.
AARCH64: SVE

- Indexed forms are also carried forward from NEON but they now work on **128-bit chunks**.

\[ \begin{array}{c|c|c|c|c}
  a & a_0 & a_1 & a_2 & a_3 \\
\end{array} \]

SVE-256 single precision
AARCH64: SVE

• Constant register initialisation is aided through replicating loads instructions \((\text{ld}1\text{rqw})\).

Let us execute: \(\text{ld}1\text{rqw} \{ z1.s \}, \ p0/Z, [x26] \).
AARCH64: SVE

• Unfortunately, indexing is only supported for the first 16 (double) or 8 (single) registers.

• Thus when the number of unique values exceeds 32 we need to allocate a low-numbered temporary register.

• Thankfully, on modern architectures register-to-register move operations are zero latency.
AARCH64: SVE

SIMD Register

- **Packed A values**
- **Loaded B value**
- **C accumulator**
- **Temp packed A values**
The SVE implementation of A-in-memory is complicated by the absence of a lane-indexed replicating load.

It should be possible to emulate this using the predication support in SVE.

Unfortunately, ld1rqw only has support for zeroing masked lanes rather than merging them.
AARCH64: SVE

- As such we revert to an x86-64 type approach where we cache one unique value per register.

- Load instruction overhead is minimised through multiple pre-offset pointers into the constant array.

![Diagram showing memory layout with 64 values and pointers at x26, x27, x28, and x29.]
AARCH64: SVE

SIMD Register

- Loaded B value
- C accumulator
- Temp broadcast A value
Blocking Factor Selection

• We often have freedom around our choice of $m$- and $n$-blocking.

• Given the complexities associated with assessing the various tradeoffs, in FSSpMDM we adopt a simple auto-tuning strategy: generate a range of different kernels and see what works best.
Results

• In order to evaluate our routines we will consider several of the operator matrices which arise when solving an advection-diffusion problem with PyFR.

• Moreover, we will consider a quadrature-free numerical scheme employing a collocation type projection with Gauss–Legendre type solution/flux points.
Results

- Tetrahedron
  - Dense
- Pyramid
  - Uncommon
- Prism
- Hexahedron

• Let $N = 40$ (corresponds to **eight elements per block** with compressible Navier–Stokes).
Results

- **M0**: Volume-to-surface.
- **M132**: Local divergence.
- **M3**: Divergence correction.
- **M460**: Local gradient.
- **M6**: Gradient correction.
Results: AVX2

- i7-12700H (P-core).
- GCC 13.1.
- Hex.
Results: AVX2

- i7-12700H (P-core).
- GCC 13.1.
- Prism.
Results: NEON

- M1 Max (P-core)
- GCC 12.3.
- Hex.
Results: NEON

- M1 Max (P-core)
- GCC 12.3.
- Prism.
Conclusions

• Have described the new small fixed sized sparse matrix multiplication functionality (FSSpMDM) in libxsmm.

• Demonstrated how FSSpMDM is able to outperform GiMMiK on both Intel and Apple architectures.
Backup Slides
Hex M3 Matrix Performance

• GiMMiK often outperforms FSSpMMDM for the hex M3 matrix.

• This is due to the unique structure of M3 for tensor-product elements.
Hex M3 Matrix Performance

• GCC exploits these properties by:
  
  • Performing **common sub-expression elimination**.
  
  • Only using a single accumulator to save registers.
  
• Saving **multiple B values in registers** to save cache bandwidth.