Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Generating Small Sparse Matrix Multiplication Kernels for Knights Landing

Nathan W. Brei

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Generating Small Sparse Matrix Multiplication Kernels for Knights Landing

Author:	Nathan W. Brei
1st examiner:	Univ.-Prof. Dr. Michael Bader
2nd examiner:	Univ.-Prof. Dr. Thomas Huckle
Assistant advisor:	Carsten Uphoff, M.Sc.
Submission Date:	February 15, 2018

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

February 15, 2018 Nathan W. Brei

Acknowledgments

I would like to thank:

My family, my advisor, Carsten Uphoff, and the Chair of Scientific Computing

for giving me the opportunity to start this project,

and the strength to finish it.

"Science is what we understand well enough to explain to a computer.

Art is everything else we do."

-Donald Knuth

Abstract

High-performance seismic wave simulators based on ADER-DG, such as SeisSol, have an inner compute kernel consisting of a chain of small sparse and dense matrix multiplications. Due to their excellent scaling characteristics, these methods benefit from any improvement to single-core performance. In the past, two code generators have been employed to produce routines optimized for each matrix product. The sparse generator unrolls the sparsity pattern into the instruction stream, while the dense generator fills in the matrix and makes optimal use of vectorization and register blocking. This work combines ideas from each in order to design a family of generators, focusing on the dense-by-sparse case, which can outperform their predecessors. It introduces UnrolledSparse, a generator which combines sparsity pattern unrolling with register blocking, along with GeneralSparse, which bypasses a number-of-nonzeros limit inherent in earlier sparse kernels. Other generators are developed to take advantage of regularities in the sparsity pattern. An experiment demonstrates a speedup of 1.83, over the current dense kernel, for SeisSol's 'star' matrix product. Scaling studies show that the speedup of UnrolledSparse is linear relative to the number of nonzeros in the sparse matrix. To implement these, tools for manipulating syntax trees of assembly code and traversing abstract matrix blocks were developed.

Contents

1 Introduction

1.1 Motivation

SeisSol solves the seismic wave equations over an unstructured tetrahedral mesh, including dynamic rupture and viscoelastic attenuation. [13] It is able to scale to petaflop performance due to its numerical method, ADER-DG. [10] ADER-DG combines the Discontinuous Galerkin method, a finite element method which allows for discontinuities at element interfaces, with the Arbitrary high-order DERivatives time integrator. Notably, it avoids a global system matrix, using element-local matrix chains instead. Because communication only happens between elements which share a face, and each element experiences a high computational load, SeisSol benefits from painstaking optimization of its innermost compute kernels. These compute kernels consist of repeated (and recursive, in the case of ADER) products of small sparse-dense-sparse matrix chains. One example of such a matrix chain is shown in Figure 1.1. Because these sparsity patterns are known at compile time, there is room for creativity in eking out performance gains.

1.2 Hardware constraints

While the motivation is straightforward, it is challenging in practice because the problem is working against current trends in computer architecture. A growing number of the Top500 computers are using manycore processors such as GPUs or Xeon Phis. This trend is likely to continue because it allows greater parallelism, throughput, and scaling at a lower

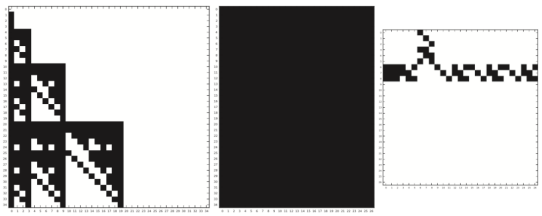


Figure 1.1: Common sparsity patterns in SeisSol

energy cost. For this particular work, the goal is to further optimize the kernels for LRZ's CoolMUC3 cluster, which runs on Knights Landing.

Knights Landing (KNL) is the second-generation Intel Xeon Phi, a manycore processor that offers greater parallelism and less power consumption in exchange for a slower clock speed. An overview is given by [12] and a more comprehensive resource is given by [9]. Unlike its predecessor, Knights Corner, KNL is a host processor rather than a coprocessor, which means that it can run the entire x86-64 instruction set, albeit with a performance penalty for some operations. KNL has two kinds of memory, a high-bandwidth MCDRAM and a high-capacity DDR4. Each processor contains between 64 and 72 cores with 4 hyperthreads per core, resulting in at least 256 logical CPUs. The instruction-level parallelism is similarly impressive due to the AVX-512 instruction set extensions. These provide 32 vector registers, each of which holds 64 bytes, allowing 8 double-precision floating point numbers to be operated on simultaneously in each vector processing unit (VPU). Instruction-level performance is further improved by an enhanced fused multiply add (FMA) instruction, which performs $c := c + a * b$ in a single cycle, optionally including a broadcast and a mask. Thus the theoretical peak performance is given as:

$$2\ \frac{\text{VPUs}}{\text{core}} \times 16\ \frac{\text{double flops}}{\text{VPU} \cdot \text{cycle}} \times 1.3\ \frac{\text{GCycles}}{\text{sec}} = 41.6\ \frac{\text{double GFlop}}{\text{sec} \cdot \text{core}}$$

$$36\ \frac{\text{tiles}}{\text{chip}} \times 2\ \frac{\text{cores}}{\text{tile}} \times 41.6\ \frac{\text{double GFlop}}{\text{sec} \cdot \text{core}} = 3.0\ \frac{\text{double TFlop}}{\text{sec} \cdot \text{chip}}$$

This peak performance is in practice merely an upper bound, as it assumes a steady-state throughput of 2 FMAs per cycle. The algorithms which approach this are mainly dense matrix multiplication or LU/Cholesky factorization. If the algorithm cannot be vectorized at all, the attainable performance drops by a factor of 8, and if it cannot use the FMA instructions, it drops by a further factor of 2. Hence inherently scalar algorithms perform dramatically worse. This may be a reasonable tradeoff: since scalar algorithms are much more likely to be memory-bound, the extra flops might be wasted. Nevertheless this provides a strong incentive to find creative ways to make the most of the vector registers.

Another key constraint is cache bandwidth. With only one store unit, KNL can store at most 64B/cycle, corresponding to one vector register. This means that it is essential to accumulate all possible changes to that vector register before storing it. It also makes loop unrolling quite important for amortizing the cost of loads and stores.

A number of related bottlenecks encompass the instruction pipeline. Because Knights Landing was derived from the older Silvermont architecture, which was optimized for low power consumption, the instruction pipeline is simpler and narrower. The instruction cache provides 16B/cycle, and the decoder handles 2 instructions/cycle. For comparison, a single

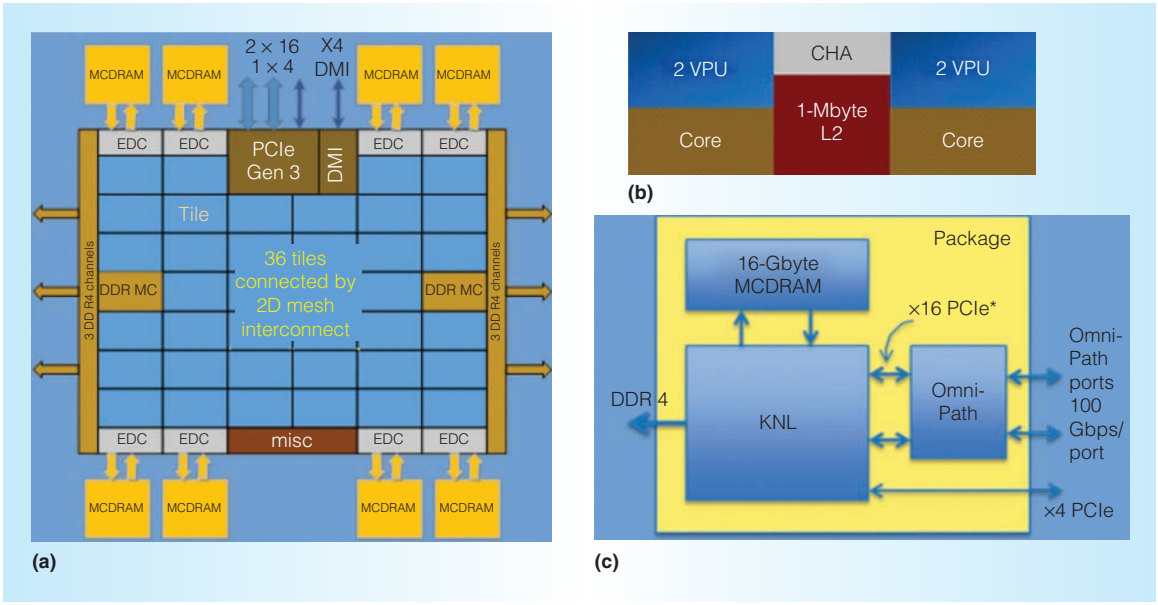


Figure 1.2: Block diagram of the Knights Landing architecture, taken from [12]. As shown in (a), each processor is divided into 36 tiles connected in a 2D mesh. As shown in (b), each tile contains 2 cores and a shared L2 cache. Each core has two VPUs and its own L1 caches.

FMA instruction takes up between 7 and 10 bytes. The least expected constraint, however, is that the out-of-order execution pipeline can only issue 2 instructions per cycle. This means that any instruction which is not vector arithmetic (e.g. an FMA) *displaces* a vector arithmetic instruction, halving the potential computational efficiency of one cycle.

In summary, there is a variety of architectural constraints which make Knights Landing well-suited for large dense operations, and correspondingly less well-suited for small sparse operations. SeisSol still performs quite well on KNL, because of its scalability across threads and nodes.[**?**] Nevertheless its performance responds to optimizations of its inner matrix kernels.

1.3 Previous approaches

Significant work has already been undertaken to optimize the performance of the sparse matrix kernels. There are three main approaches which have been explored. Originally an off-the-shelf sparse matrix routine was used. Later, this was replaced by a custom kernel generated for each specific sparsity pattern, introduced by Breuer in [2]. Finally, this was

replaced by a dense kernel optimized for small matrices and wide registers, introduced by Heinecke in [7]. The decision to use the dense kernel on KNL is discussed in depth in [9].

There are several properties of the SeisSol matrix chains which can be leveraged to improve performance. Firstly, these matrices remain quite small: the largest matrix dimension under consideration is 56 and the smallest is 9. This means that they can be assumed to fit in cache. Secondly, the sparsity patterns are constant. This is because they are derived from the basis functions, which themselves depend on the desired order of accuracy and the choice of viscous damping model. These properties do not appear to play to the strengths of Knights Landing, but they do provide the programmer something to work with.

There are also several constraints. SeisSol is already carefully parallelized via MPI and OpenMP, and the kernels are expected to run within each thread. This means that only instruction-level parallelism shall be considered in this work. Likewise, the strategy for utilizing KNL's high-bandwidth memory is determined at a much higher level, and shall not be considered here.

1.3.1 Off-the-shelf sparse matrix routines

The most straightforward approach to handling SeisSol's small sparse matrix kernels is to use off-the-shelf dense-by-sparse multiplication routines such as those provided by MKL [8]. This approach benefits from being very general: it places no constraints on the matrix size or sparsity, and relegates knowledge of the sparsity pattern to runtime. The matrix is stored in a common format such as CSC, which store indexes so that no information is lost and other routines may interact in a relatively abstract manner. The routines themselves take up a small, fixed amount of memory which does not depend on the matrix sparsity pattern.

However, Breuer and Heinecke [2] demonstrated that this approach leaves a considerable amount of performance on the table. This is partially due to the fact the routines are usually optimized for much larger matrix sizes than are present here. There is also an overhead due to the index lookups, the innermost loops which can't be unrolled, and the prevalence of scalar arithmetic.

1.3.2 Sparse kernel generation

Breuer's solution to the inefficiencies in the previous approach was to generate a custom kernel for each sparse matrix product. The generator runs in a separate phase before compilation, and takes as inputs the matrix sizes, the target architecture, and a file describing the sparsity pattern. From this information it emits a C function to perform the multiplication. These functions differ from their predecessors in several key ways.

1. The sparsity pattern is expressed in a direct sequence of multiply-adds; indices are hardcoded. This means that the indirect lookups are eliminated completely, and the indices need not even be present. Thus the CSC format becomes a 'virtual' CSC format, effectively just an array of nonzero matrix entries ordered by columns, and the memory movement is reduced.

2. The code has far less control flow. Whereas the classic kernels must loop over individual elements of B, these kernels have in effect unrolled both inner loops, so that only the outermost loop remains. The compiler is instructed to vectorize this.

3. This unrolling emits one FMA per nonzero in the matrix B, which means that the size of this algorithm can run up against the instruction cache limit if B becomes too large or dense.

4. The generator for CSC dense-by-sparse is dramatically simpler and more performant than for CSC sparse-by-dense. This asymmetry comes from the vectorization, and will be discussed in more detail in Section 2.3.

This approach yielded a speedup of 2.2 over the off-the-shelf routines on Sandy Bridge using AVX-128 instructions. However, on Haswell and Knights Landing, it was shown to perform comparable to or worse than the dense approach discussed next. [9]

1.3.3 Dense kernel generation

The dense kernel generator is a library known as libxsmm. [7] Libxsmm specializes in small dense kernels, and is specifically tuned for Knights Landing, which shall be described later. Rather than emitting C code like Breuer's sparse generators, it emits assembly directly. The kernels it produces are specialized for the exact matrix sizes needed. It has both an offline generator and a JIT compiler which allows it to build desired kernels during runtime.

Despite its thoughtful design, libxsmm is fundamentally stymied by the fact that it has to fill in the sparse matrix with zeros and thereby perform large amounts of wasted computation. Somehow it still comes out ahead of the existing sparse kernels on Knights Landing. Quoted straight from [9]:

> As we are operating out of L1 without significant reuse in the case of the sparse variant, this limits us to at most 50% of peak performance of the core as we can only store one register per cycle. The dense variants have register accumulators and can therefore exploit both VPUs and catch up.

This is an algorithmic limitation of the current sparse matrix kernels, not an architectural one. If the code generator scheduled its loads, stores, and FMAs such that they accumulated

into a carefully-sized block of C, the performance of the sparse kernels would most likely outperform dense in some cases.

1.4 Goals, approach, and design space

Each of the existing algorithms discussed in the preceding section encounter different bottlenecks and constraints. Thus there appears to be room to create more complicated algorithms which combine the desirable features of each. The three existing algorithms can be thought of as poles in a barycentric design space, and the aim is to find points in the middle which finesse around the bottlenecks in the corners. One corner has high compute intensity and an optimal register blocking, but wasted flops; another has no wasted flops but many control dependencies and lower compute intensity; the last has no wasted flops and high compute intensity, but no register blocking. The center ought to have decent compute intensity, a decent register blocking, few control dependencies, and few wasted flops.

To keep the scope of this work bounded, we adhere to following assumptions:

- The sparse matrix has a fixed pattern which is known at compile time.
- The memory layout of the sparse matrix is not constrained elsewhere.
- All matrices fit in the L2 cache: $(mn + mk + kn) \cdot 8 \leq 1e6$

Within this framework, we consider variations of the following:

- Modifications of the sparsity pattern
- Modifications of the control flow
- The memory layout of the sparse matrix
- The block sizes, if a block decomposition is used
- The ordering and unrolling of the loop nest

Although interesting, we disregard prefetching, instruction alignment, and register assignment.

The goal of this work is to find algorithms which might outperform the status quo, and write a code generator which will emit an instance of each relevant algorithm for a given problem. The generator should be able to handle different matrix sizes, number of nonzeros, and sparsity patterns, while staying within the aforementioned assumptions.

Once implemented, the generators can be tested – using the actual SeisSol matrices – to determine if they yield a speedup. Equally interesting is exploring the algorithm's problem space constraints. Using both the observed and estimated performance, we can address the questions "which problems can this algorithm handle efficiently?" and "which algorithm is best for a given problem?".

Each algorithm requires additional tuning parameters, and the effect of these should also be estimated and tested, addressing the question "which parameters yield the best kernel for a given problem?". If possible, heuristics should be given in order to choose these parameters automatically.

1.5 Roadmap

This work is broken down as follows. Chapter 2 lays out the fundamental concepts and terminology. Chapter 3 covers a progression of different dense-by-sparse kernels in depth, and discusses strategies for sparse-by-dense and tensor kernels which might benefit from a similar treatment. Chapter 4 discusses the design of the code generator, and provides a brief manual illustrating its use. Chapter 5 walks through some experiment results, attempting to discern performance and scaling characteristics. Finally, Chapter 6 wraps everything up.

2 Background

2.1 Historical development

Dense and sparse matrix multiplication account for a surprising fraction of all computation, particularly in scientific and engineering fields. Since any optimization or algorithmic improvement has the potential to improve the performance of programs across the board, these algorithms have been organized under a series of common interfaces. Not only does this make it easy to reuse the best known implementation of many fundamental linear algebra operations, but it also confers other advantages: robustness is increased by the careful handling of edge cases, portability is increased because a programmer can link to the best implementation for their target architecture, and readability is increased because numerical algorithms can all work at the same abstraction level. The first such interface, basic linear algebra subprogram (BLAS), was introduced in 1979 by Lawson [11] and only supported dense vector and matrix operations. Later interfaces, such as LAPACK, build upon and extend BLAS to support increasingly complex operations; the history and design of such interfaces is covered by Dongarra in [4].

Most of the naming conventions used in this work follow those of BLAS and its successors. The arguments for each operation get quite complicated (see [8]), but for our purposes they can be simplified as shown in Figure 2.1. These conventions have propagated to libraries including MKL and libxsmm.

A sparse BLAS interface was not standardized until 2002, as described by Duff in [5]. Compared with the dense case, sparse matrix operations are considerably more difficult to pin down because the performance of each operation depends heavily – often asymptotically – on the memory layout, which in turn depends on both the chosen storage format and the sparsity pattern, which is often known only at runtime. The choice of storage format is tightly coupled with the implementation of the operations, making abstraction difficult. Thus there are a multitude of different storage formats, with not entirely consistent naming conventions. The formats used in this work are described in Figure 2.2 and introduced as they are needed in Chapter 3. Ultimately, a sparse BLAS works well when the matrices are large, are constrained to a common format such as CSC, and have unpredictable or nonconstant sparsity patterns; otherwise a custom implementation might be able to do much better.

```
// C = alpha*A*B + beta*C
dgemm(m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)
```

Variable	Meaning
dgemm	Double General Matrix Multiplication
alpha	A scalar constant
beta	A scalar constant
A	A matrix of doubles, in dense column-major format unless sparse
B	A matrix of doubles, in dense column-major format unless sparse
C	A matrix of doubles, in dense column-major format
m	The number of rows of C and A
n	The number of columns of C and B
k	The number of columns in A and rows in B
lda	The leading dimension (offset between columns) of A; zero indicates sparse
ldb	The leading dimension (offset between columns) of B; zero indicates sparse
ldc	The leading dimension (offset between columns) of C

Figure 2.1: Simplified BLAS naming conventions for general matrix multiplication

In 2008, Goto and Geijn [6] devised a general framework for designing optimized kernels for large dense matrix multiplication. They start by exploring the space of recursive block decompositions, and then they build up a model of the memory hierarchy which allows them to prune their design space, decide what data needs to be packed, and decide what block sizes are best. Thus they arrive at an algorithm for designing a performant matrix kernel. However, their reasoning only applies to matrices which are large enough to involve all the levels of the memory hierarchy. They only briefly discuss the design of the innermost kernel, which is small enough to fit in registers.

2.2 Outer-product formulation

Luckily, the state of the art innermost kernel is discussed in depth by Heinecke [7] with regards to libxsmm. It is known as the outer-product formulation. An excerpt from a kernel optimized for KNL is provided in Figure 2.3. The basic idea is to load a block of C, column by column, into the vector registers. We then stream loads of columns of A. We wish to perform the vector outer-product of each column of A with the corresponding row of B, and accumulate these in the C block. This is accomplished by repeatedly loading a *scalar* value of B, broadcasting it into a vector register, multiplying the broadcasted value of B with the current column of A, and adding it to the correct column of the C block. Once all of the columns of A have been processed, the C block is stored back down to memory

Dense (DNS) Column-major dense format, possibly padded

Coordinate format (COO) Entries are stored in an array alongside their row and column indices.

Compressed sparse column (CSC) Nonzero entries are stored top to bottom, then left to right. Row indices are used but column indices are not; pointers to the start of each column are used instead.

Compressed sparse row (CSR) The row-major equivalent of CSC

Block sparse column (BSC) The matrix is subdivided into blocks, each of which is either empty or dense. The blocks are ordered and indexed in CSC format.

Block sparse row (BSR) The row-major equivalent of BSC

Block compressed sparse column (BCSC) The matrix is subdivided into blocks, each of which might be zero, or might have its own sparsity pattern. There are effectively two levels of CSC.

This format is particularly important for us for two reasons: Firstly, all of the entries in one block are located together. Secondly, the memory offset from the first entry in the block to any other entry in the block is not influenced by neighboring blocks.

Block compressed sparse row (BCSR) The row-major equivalent of BCSC

Virtual... (V...) A sparse format where all of the index information has been omitted; all that is left is an array containing the nonzero values.

Figure 2.2: Descriptions of different sparse matrix formats *as used in this work*. Naming conventions vary between literature; pay attention to the distinction between BSC and BCSC.

and the algorithm moves on to the next block.

Libxsmm performs a number of optimizations specifically for KNL. The most noticeable is that a 1D blocking of A and C is chosen, partly because of the convenience of that tile size, and partly because additional loads of A columns interfere with vector arithmetic due to the KNL's narrow issue width. A second problem is that B is stored by columns but accessed by rows, so the memory addresses will likely contain offsets greater than 128×8, causing the size of the FMA instructions to increase to 10 bytes, and hitting the instruction cache bandwidth bottleneck as discussed in Section 1.2. The solution is to switch from pointer-offset addressing, e.g. `rsi + 198`, to scale-index-base addressing, e.g. `rsi + 8 * r13 + 22`, which uses two registers and two hardcoded values to express the same address. This makes the assembly code much harder to read.

```
1  # Load C register block
2  vmovapd 0(%rdx),  %zmm16                              # load C[:,0]
3  vmovapd 64(%rdx), %zmm17                              # load C[:,1]
4  vmovapd 128(%rdx), %zmm18                             # load C[:,2]
5  # ...
6
7  # Load A register block
8  vmovapd 0(%rdi), %zmm0                                # load A[:,0]
9  vmovapd 64(%rdi), %zmm1                               # load A[:,1]
10 # ...
11
12 # Outer product of A[:,0] * B[0,:]
13 vfmadd231pd 0(%rsi) {1to8}, %zmm0, %zmm16             # C[:,0] += A[:,0] .* B[0,0]
14 vfmadd231pd 0(%rsi,%r15,1) {1to8}, %zmm0, %zmm17     # C[:,1] += A[:,0] .* B[0,1]
15 vfmadd231pd 0(%rsi,%r15,2) {1to8}, %zmm0, %zmm18     # C[:,2] += A[:,0] .* B[0,2]
16 # ...
17
18 # Outer product of A[:,1] * B[1,:]
19 vfmadd231pd 8(%rsi) {1to8}, %zmm1, %zmm16             # C[:,0] += A[:,1] .* B[1,0]
20 vfmadd231pd 8(%rsi,%r15,1) {1to8}, %zmm1, %zmm17     # C[:,0] += A[:,1] .* B[1,1]
21 vfmadd231pd 8(%rsi,%r15,2) {1to8}, %zmm1, %zmm18     # C[:,0] += A[:,1] .* B[1,2]
22 # ...
23 # More outer products ...
24
25 # Store C register block
26 vmovapd %zmm16, 0(%rdx)                               # store C[:,0]
27 vmovapd %zmm17, 64(%rdx)                              # store C[:,1]
28 vmovapd %zmm18, 128(%rdx)                             # store C[:,2]
29 # ...
```

Figure 2.3: Example outer-product formulation generated by libxsmm specifically for KNL

2.3 The dense-by-sparse vs sparse-by-dense asymmetry

The outer product formulation can be easily adapted into a column-major dense-by-sparse matrix product, because the dense A matrix is loaded column by column, and the sparse B matrix is loaded element by element. It can be vectorized perfectly, and zero entries in B eliminate exactly one FMA instruction. In contrast, the column-major sparse-by-dense matrix product vectorizes particularly badly. In order to load a full vector column of sparse A, the zero values must be filled in and the sparsity lost. For row-major formats, the reverse is true. The outer product formulation can simply load individual elements along a column of A, load rows of B, and stores back to rows of C. Thus the sparse-by-dense becomes easy and dense-by-sparse becomes hard.

This becomes less surprising when we remember that row-major and column-major formats are related to each other by the transpose operation, which has the property $(AB)^T = B^T A^T$. This hints at a solution: Transpose the matrices, perform the sparse operation, and transpose them back. Unfortunately this is too costly to do in an innermost loop. Breuer's solution to this problem was to identify contiguous entries and then mask everything else; noncontiguous entries were handled as scalars. A more elegant solution remains to be found.

2.4 Naming conventions

This work uses the BLAS naming conventions wherever possible. These conventions must be extended in several ways. Firstly, iteration variable names are derived from the size variable names by adding the suffix `-i`. Secondly, we wish to iterate not only over cells, but also blocks and vectors. The naming convention handles this via a prefix. In effect, `m,n,k` determine the iteration *direction*, and the prefix determines the units. If iterating over a matrix in isolation, `r,c` directions are preferred. Finally, we introduce the direction `l` to handle the third dimension, when working with tensors. A grammar is provided in Figure 2.4. This convention adds much-needed predictability and ease of comprehension inside deeply nested loops. It also helps ensure that iteration variables stay within the correct range. The most common pattern is to iterate over matrix blocks, and then inside each block iterate over cells (for sparse matrices) and vectors (for dense matrices). An example is given in Figure 2.5.

```
<var>       ::= [<units>]<direction>[i]
<units>     ::= b | B | v | V
<direction> ::= m | n | k | l | r | c
```

Units	Mnemonic	Meaning
none	Cell	Cells in the matrix
b	Intra-block	Cells in a block
B	Inter-block	Blocks in the matrix
v	Intra-vector	Cells in a vector
V	Inter-vector	Vectors in a block

Figure 2.4: Grammar for variable names.

```
for Bci in range(Bc):       # Iterate over blocks of columns
    for Bri in range(Br):   # Iterate over blocks of rows
        index = blocks[Bri, Bci]        # Get index of current block
        pattern = patterns[index]        # Get pattern for current block
        for bci in range(bc):               # Iterate over column entries inside block
            for bri in range(br):           # Iterate over row entries inside block
                if pattern[bri,bci]:
                    offsets[Bri*br + bri, Bci*bc + bci] = x   #
                    x += 1
```

Figure 2.5: Sample code demonstrating the variable naming convention. This code calculates the memory offset of each nonzero element in a sparse matrix when it is stored in VBCSC format.

3 Algorithms

The main focus of this thesis is on improving the performance of small matrix multiplication algorithms. It is necessary to finesse around the bottlenecks that constrain the approaches described in the previous chapter. The extra information which makes this possible is the sparsity pattern known at compile time. The focus is on dense-by-sparse multiplication because it stands to benefit the most from vectorization, although sparse-by-dense and tensor-by-sparse multiplication are also discussed. Figure 3.1 shows the different dense-by-sparse algorithms under consideration and the relationships between them.

The starting point is the dense outer-product formulation generated from `libxsmm`. Applying knowledge about the sparsity pattern yields MicroSparse, which avoids wasted computation but is restricted to very small matrix sizes. Decomposing a matrix into blocks, using MicroSparse for each block, and unrolling some of the outer loops yields UnrolledSparse, which is not restricted by matrix dimensions but is restricted by the number of nonzeros. In order to support matrices with many nonzeros, some regularity assumptions have to be made.

The simplest approach is TiledSparse, which assumes that a single block pattern is repeated exactly. This assumption is impractical because for most matrix patterns, the block pattern will fill in. Instead we can break the assumption into two parts and relax them separately. We can assume that there is only one block sparsity pattern, but some blocks are entirely zero, which yields the BlockedSparse algorithm. Alternatively, we can assume that there are multiple block sparsity patterns, but they all have the same number of nonzeros, which yields PatternSparse. Relaxing both assumptions yields GeneralSparse, which can evade the number of nonzeros constraint with the most flexibility, but at the cost of additional overhead. Variations can be explored which extract the diagonal and handle it separately; this would be especially helpful for PaddedSparse in the case of matrices with a full diagonal and random fill-in elsewhere.

3.1 Dense-by-sparse microkernel

The MicroSparse kernel is our first attempt to account for the sparsity pattern while building off of the state of the art for dense matrix multiplication. Our key assumption is that the

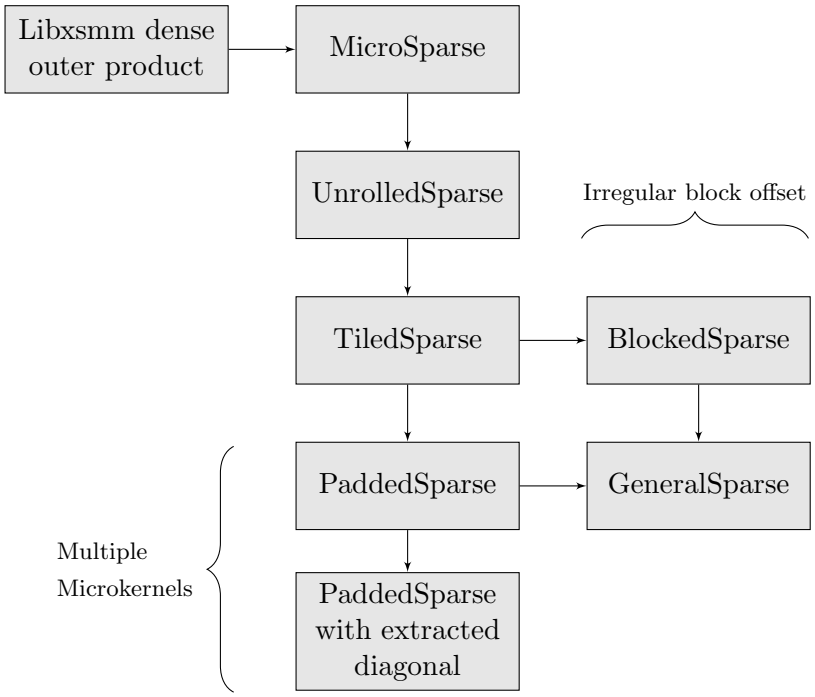


Figure 3.1: Relationships between different dense-by-sparse multiplication algorithms

matrices A and C are small enough to fit entirely in registers: $(n + k) * (m/8) <= 32$. Generating the dense GEMM for this problem effectively extracts the dense microkernel. This microkernel has no loops (otherwise it couldn't amortize the cost of loading and storing A and C) and this makes it easy to modify.

The sparse microkernel is a straightforward modification of the dense microkernel. All FMA instructions `C[:,ni] += A[:,ki] .* B[ki, ni]` where `B[ki, ni] = 0` due to the sparsity pattern are removed. This is tricky to do in a general way; an abstraction designed for the purpose is described at length in Section 4.4. Next, any loads of a column of A corresponding to an empty row of B are removed. Finally, any loads and stores of a column of C corresponding to an empty column of B are removed. This is described via pseudocode in Figure 3.2.

Before proceeding to more complex algorithms, it is worthwhile to consider the degrees of freedom available. The first parameter to consider is the format of the sparse matrix B. Because the memory offsets of each `B[ki,ni]` are hardcoded into the FMA, any format will work as long as the pointer to B never moves. However, the choice of format strongly affects memory access patterns. Keeping B dense is suboptimal because the algorithm will be streaming nonzeros which are unduly spread across different cache lines. Breuer and Heinecke used a virtual compressed sparse column format, where the nonzeros were laid

```
for each vector Vmi:
    for each cell ni:
        if B[:,ni] != 0:
            emit load C[Vmi, ni]

for each vector Vmi:
    for each cell ki:
        if B[ki,:] != 0:
            emit load A[Vmi, ki]

for each vector Vmi:
    for each cell ki:
        for each cell ni:
            if B[ki,ni] != 0:
                emit fma C[Vmi, ni] += A[Vmi, ki] .* B[ki, ni]

for each vector Vmi:
    for each cell ni:
        if B[:,ni] != 0:
            emit store C[Vmi, ni]
```

Figure 3.2: Pseudocode for MicroSparse

out in a plain 1D array ordered by columns. For MicroSparse, a virtual compressed row format might have better locality because the *ni* loop is innermost, but this is insignificant because few cache lines are involved on sparse matrices of this size.

Having chosen the format, the next step is to choose the addressing mode for the elements of the sparse matrix. For now we assume that B points to the first entry in this matrix and the entries are all adjacent. Because the maximum number of doubles in B is 256 and the cutoff for 7-byte FMAs is 128, it already makes sense to switch to scale-index addressing. Because this is complicated to do, it will be hidden behind the cursor abstraction described in Section 4.4.

Another question is alignment. The alignment of B is not particularly important because its elements are loaded one at a time, but very important for A and C, which are loaded in a vectorwise fashion. If not aligned on a 64-byte boundary, the vector load operation will suffer a delay because the data is split across two cache lines. Using the aligned vector load operation `vmovapd` will trigger a segmentation fault if the pointer is not actually aligned. On the other hand, the unaligned version `vmovupd` can handle both the aligned and unaligned case, and will not incur the penalty if the pointer is in fact aligned. Fortunately, this choice can be made independently of any other considerations.

The performance of MicroSparse is demonstrated in Section 5.2. It is clear that removing

the FMA instructions decreases the arithmetic intensity, and if the nonzeros are evenly distributed, there will be no accompanying decrease in memory bandwidth. Nevertheless the algorithm remains compute-bound and experiences an improved time-to-solution compared to its dense counterpart. Regardless of performance, MicroSparse has limited utility in real problems, since it can only be applied to extremely small matrices. It is valuable primarily as a building block for the more complex algorithms developed next.

3.2 Unrolled dense-by-sparse multiplication

The goal of UnrolledSparse is to support larger matrix sizes by building off of MicroSparse. The first key idea is to introduce a block decomposition: the generators traverse blocks (Bmi, Bni, Bki) and delegate the multiplication of each block to MicroSparse. The second key idea is to account for the fact that different blocks contain different subpatterns by unrolling the Bni and Bki loops. The sparsity pattern is obviously invariant with respect to Bmi, so it is not necessary to unroll that loop.

It is essential that Bki be the innermost loop because this lets us accumulate changes to the block `C[Bmi, Bni]`, and then store the entire block exactly once. Otherwise, the repeated storing of each block of C becomes the primary bottleneck. This also means that we only reuse the inner part of MicroSparse, and we handle the loads and stores of C ourselves.

Putting the Bmi loop on the outside is the best choice for several reasons. Firstly, it consolidates instructions that are free of control dependencies, which improves pipeline efficiency and gives the processor more room to amortize loads of A and C. Secondly, it dramatically reduces the number of branches. Even if branches are predicted perfectly, KNL's issue-width bottleneck ensures that the instructions to increment and compare the counter register would displace FMAs, reducing performance.

Thus there is only one reasonable ordering for our loop nest. At the innermost level we unroll over the Bki loop, laying down a sequence of sparse microkernels corresponding to a traversal of one panel of B, moving from top to bottom in short, fat slices. In the middle level, we unroll over Bni, corresponding to a traversal across panels of B from left to right. Finally, at the outermost level, we loop over Bmi: horizontal panels of A and C from top to bottom. This is described more precisely via pseudocode in Figure 3.4a.

This loop nesting looks very similar to what Kazushige Goto [6] calls GEPM_VAR2. Note that Goto rejects this algorithm within one paragraph, making the exact opposite argument as we have made above, regarding the repeated storing of the C block. This is because Goto is focusing on a much larger scale: his C blocks fit only in the L2 cache, whereas ours fit entirely in registers: we get our inner "GEPDot" for free whereas his is especially costly.

The UnrolledSparse algorithm has three additional tuning parameters which must be chosen. These are the blocksizes bm, bn, and bk. Section 5.3 explores how to choose these values heuristically. The choice of parameters and the set of problems which can be efficiently solved are subject to the following constraints:

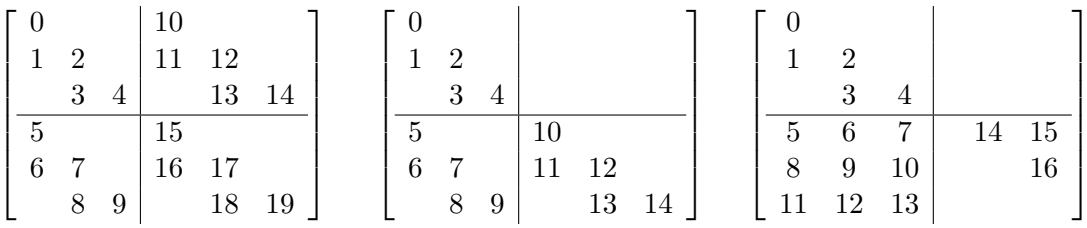
- The A and C blocks must fit in registers: $(bk + bn) * (bm/8) <= 32$
- Each nonzero in B produces exactly one FMA, and if there are too many of these, they will fill the instruction cache. This introduces a hard limit on the number of nonzeros, which shall be explored in depth in Section 5.2
- The instruction cache bottleneck can be hit with fewer nonzeros depending on the choice of blocksizes, since different blocksizes result in different numbers of `vmovapd` instructions. This is covered in Section 5.3.
- Choices of (m, n, k) which do not evenly divide into small blocks are feasible, but not fully supported at the time of this writing.

The experiments discussed later consistently show that UnrolledSparse strikes a good balance between performance and flexibility. It supports most available matrix sizes, it does not fill in any zeros or waste any flops, and it uses a dense-style register blocking. It continues to perform well as the B matrix becomes dense – the parallel efficiency increases – and it degenerates to something similar to libxsmm. For fully dense matrices it performs slightly worse than libxsmm, and if it reaches the instruction cache limit it performs dramatically worse. In order to move past this constraint, we find ways to compress the number of FMAs below the number of actual nonzeros.

3.3 Tiled dense-by-sparse multiplication

The basic idea behind TiledSparse is to scale the FullyUnrolledSparse algorithm to matrices with more nonzeros than the instruction cache permits by assuming that the sparsity pattern is perfectly regular. Specifically, we assume that there is a single sparsity pattern which tiles over the entire matrix and is small enough to be used by MicroSparse. Our tiling assumption can be made without loss of generality because we can always fill in zeros until we arrive at a regular pattern. One way to do this is to partition the original matrix with the desired blocksize, and then take the logical union of the sparsity patterns in each block. The assumption does result in a loss of practicality, however, as often the pattern will degenerate to dense. If the pattern becomes dense, the resulting algorithm will be more or less similar to libxsmm depending on the choice of block sizes.

We now decide the order in which the loops are nested, and also which loops are unrolled.



(a) TiledSparse (b) BlockedSparse (c) GeneralSparse

Figure 3.3: Sample matrices illustrating the blocking constraints for different GEMM algorithms. TiledSparse admits only a single block pattern which is tiled perfectly over the entire matrix. BlockedSparse admits a single block pattern along with empty blocks. GeneralSparse relaxes further to admit multiple different patterns. The numbers indicate the cell's index when using a block compressed-sparse-row format.

As with UnrolledSparse, the Bki loop should be the innermost in order to accumulate updates to C, and it is advantageous to unroll it unless the matrix is very large and not very sparse, in which case it may still be advantageous to partially unroll it (though this is not yet supported). The Bni loop can not be unrolled, however, because this would once again yield one FMA per nonzero and we would be back to UnrolledSparse. Unrolling the Bmi loop is similarly likely to curtail this algorithm's scaling to higher nnzs, albeit with less predictability. There is little expected benefit to interchanging the Bmi and Bni loops, but we are free to do so.

Pseudocode showing TiledSparse is provided in Figure 3.4b. Intriguingly, the only difference between UnrolledSparse and TiledSparse (at the abstraction level shown) is the Bni loop is no longer unrolled. However, this conceals another important difference. When the Bni loop is unrolled, the commands to move the B pointer gets completely absorbed into the memory addresses, whose offsets are absolute. When the Bni loop is not unrolled, we must move the B pointer to the start of the corresponding panel, in which case the offsets are relative. This is a simple add operation, but it requires that each block takes up the same, expected amount of space in memory. Depending on the origin of the data, it might be a challenge to enforce that the layout in memory is consistent with the tiled sparsity pattern.

Pattern-specific zero-padding of sparse matrices is not something numerical libraries typically allow, but a specialized class can be provided for that purpose. If this class stores the matrix in COO format, and does so using a structure of arrays, it can mimic any virtual storage format by simply reordering the entries. Given an unpadded matrix of numeric data alongside a matrix containing the sparsity pattern, it can construct a new matrix which contains both the numeric data and the correct zero padding in an arbitrary format.

```
loop over m blocks:

   unroll over n blocks:

      load C block into registers

      unroll over k blocks:

         load A block into registers
         blockwise C += A * B
         move A right 1 block
         move B down 1 block

      store C block from registers
      move A to far left
      move C right 1 block

   move A down 1 block
   move C far left, down 1 block
```

(a) Pseudocode for UnrolledSparse

```
loop over m blocks:

   loop over n blocks:

      load C block into registers

      unroll over k blocks:

         load A block into registers
         blockwise C += A * B
         move A right 1 block
         move B down 1 block

      store C block from registers
      move A to far left
      move B to top + right 1 block
      move C right 1 block

   move A down 1 block
   move C far left, down 1 block
```

(b) Pseudocode for TiledSparse

Figure 3.4: Pseudocode for UnrolledSparse and TiledSparse. UnrolledSparse unrolls two inner loops, emitting a microkernel for every block of B. TiledSparse unrolls only the innermost loop, emitting a microkernel for every (identical) block in one panel of B. Note that the 'unroll' command replaces 'move' statements with hardcoded memory addresses when possible.

There are several improvements possible. TiledSparse can be modified to support a non-perfect tiling with a fringe on the bottom and right. It is even possible to remove the block size constraint by replacing the MicroSparse kernel with an UnrolledSparse kernel. Nevertheless, it will take much bigger modifications to escape the problem of the sparsity pattern filling to dense.

3.4 General dense-by-sparse multiplication

The general dense-by-sparse algorithm comes from relaxing *both* of the key ideas of TiledSparse: There can be any number of different block patterns, and each pattern may contain any number of nonzeros. GeneralSparse contains a collection of MicroSparse GEMMs, and chooses the correct GEMM for each block of B. It is free to scale to larger matrix sizes than those supported by UnrolledSparse, because it can be compressed. Judiciously filling in zeros can

arbitrarily reduce the number of unique block patterns, which in turn reduces the number of GEMMs. Thus GeneralSparse degrades gracefully, similar to TiledSparse. As the matrices grow larger and denser, the algorithm will converge towards either BlockedSparse or dense, depending on the original sparsity pattern. In contrast to TiledSparse and BlockedSparse, however, the fill-in (and hence wasted FLOPs) can be effectively minimized.

There is of course a tradeoff: in order to redirect control flow to the correct block GEMM, one direct and one indirect jump are needed. This must happen for every non-empty block of B. Each indirect jump will add latency; this is analyzed and estimated in Section 5.4. It might be possible to reduce this latency by putting padding around the jump instructions; this has not yet been tried.

To efficiently handle the multiway control flow, we start with a low-level pattern called a *jump table*. Assume the control variable `x` is a small integer. The destination addresses are laid out in an array `A`, and control is transferred by jumping to the address stored in `A[x]`. This happens in constant time regardless of the size of `x`.

There are two different ways we can apply this to our problem. One approach involves putting all of the MicroSparse kernels inside of a loop nest over Bn and Bk. Somewhere else, we lay out a $Bk \times Bn$ array which contains the address of the kernel corresponding to each block. The loop nest updates pointers to the start of the active blocks of A and C, updates the pointer into the address matrix, and then performs a single indirect jump using that pointer. Note that either the MicroSparse kernel must be responsible for moving B to the start of the next block, or the starting locations of each block must be stored in another array.

The second approach involves unrolling the Bn and Bk loops completely. For each block, we move the A, B, C pointers to the start of the active block, and save a return address to a register. This is effectively a (non-ABI-compliant) function call, except it avoids the stack completely. We perform a direct jump to the correct block kernel. The kernel itself performs the indirect jump to the return address stored in the register. This approach is more performant because it avoids doing the jump when it knows that the current block is empty.

GeneralSparse is subject to following constraints:

1. The total number of indirect jumps should be minimized. This favors a larger block size.
2. The routine must fit in the 32kB instruction cache, which constrains the sum of nonzeros in each *unique* block.
3. The sparse matrix must be stored in *block* compressed sparse row or column format.

Otherwise, the offsets of the entries in a block (relative to the start of that block) are no longer constant, and the microkernel returns incorrect results.

4. Blocks should be uniformly sized. This is an implementation detail; the constraint may be removed in the future by implementing another MatrixCursor as described in Section 4.4.

Automatically choosing parameters for GeneralSparse is tricker than for the preceding kernels. Ironically, the only parameters needed are a valid choice of bn and bk. To minimize the jump penalty, they should be chosen as large as possible. However, for any particular choice, the algorithm cannot estimate its performance until it has compressed the matrix-wide sparsity pattern. This can become computationally expensive.

The aforementioned sparsity pattern compression can be accomplished with a greedy algorithm. It maintains a set of unique block patterns that together cover all of the blocks. As long as the sum of nonzeros in this set exceeds the instruction cache limit, the algorithm chooses the two which are most similar according to logical XOR, and then merges them using logical OR.

3.5 Algorithms not implemented

3.5.1 Blocked dense-by-sparse multiplication

The key idea behind BlockedSparse was to partially relax the TiledSparse assumption. Specifically, we would allow only one block sparsity pattern, but also allow empty blocks. This is reminiscent of the existing BSR format, except the blocks are internally sparse instead of dense.

This is ultimately just a simplified case of GeneralSparse. The additional assumptions mean that the generated code is much shorter because it contains only one MicroSparse kernel, and that the indirect jump can be replaced with a direct jump. Neither of these changes is expected to yield a substantial performance increase. The overwhelming majority of matrices will perform worse on BlockedSparse compared to GeneralSparse due to additional fill-in.

We can assume that there is only one block sparsity pattern, but some blocks are entirely zero, which yields the BlockedSparse algorithm. Alternatively, we can assume that there are multiple block sparsity patterns, but they all have the same number of nonzeros, which yields PatternSparse.

3.5.2 Padded dense-by-sparse multiplication

PaddedSparse partially relaxes the other TiledSparse assumption – there may be multiple unique block patterns, but there is always a constant offset between blocks in memory. This is achieved by padding zeros in memory similar to how TiledSparse fills in its block pattern. Unlike TiledSparse, this doesn't affect the flop count; it merely spreads the nonzeros out in memory. The principal advantage to this approach is that it compresses sparsity patterns like GeneralSparse, but is safe to use with the CSC and CSR storage formats.

One sparse matrix pattern which occasionally shows up in scientific and engineering applications has a full diagonal and a uniform random smattering of off-diagonals. This would be well-suited for PaddedSparse if the diagonal were to be extracted first and folded into the blocks of C using a vector dot product operation instead of a broadcast.

3.5.3 Tensor-by-sparse multiplication

Sparse-by-dense multiplication is significantly more difficult due to the asymmetry inherent in the outer-product formulation. One possible alternative is multiplication of sparse matrices with tensors. Breuer and Heinecke demonstrated with EDGE [3] that seismic simulations based on ADER-DG can be *fused*, where ensembles of forward simulations are run in parallel within one execution. This has some immediate advantages. Read-only data structures can be shared, dramatically reducing memory movement. Meanwhile, the computations can be vectorized along the third dimension.

This formulation has some useful consequences for sparse kernels. Firstly, it eliminates inefficiencies caused by awkwardly sized data structures, such as matrices with 9 rows, and effortlessly ensures that memory accesses are aligned with cache line boundaries. Secondly, it gets away from the outer-product formulation: along the m, n, k directions it resembles a naive matrix multiplication with no vectorization at all. This means that the sparse-by-tensor case can be unrolled just as well as the tensor-by-sparse case.

4 Implementation

This chapter covers the structure of the algorithm generators described previously and the resulting layout of the codebase. The first section discusses some key design decisions and the constraints driving them. The subsequent sections build up the machinery used for algorithm generation, moving from low-level to high-level. The foundation is a Python abstract syntax tree for assembly programs. These basic building blocks are aggregated and parameterized to create reusable components for subtasks such as manipulating register blocks. In order to manage the complexity of generating a traversal of a matrix with both a nontrivial sparsity pattern and a nontrivial memory layout, a *matrix cursor* abstraction is introduced. This raises the possibility of a more general approach to moving information from runtime to compile time. The chapter ends with a description of the various entry points into the codebase and a brief usage guide.

4.1 Design Decisions

The first major decision was the choice of output language. There were three options available. The first option was to emit plain C/C++ code annotated with pragmas, as Breuer did. The second option was to emit C/C++ code using intrinsics. The final option was to emit plain assembly. Plain assembly was chosen for the reasons discussed below:

1. The primary consideration was reducing the layers of indirection. Many of the necessary design decisions are not directly expressible using C semantics. While there exist pragmas to control optimizations such as loop unrolling and vectorization, they do not always combine in predictable ways. One is forced to simultaneously reason at multiple abstraction levels: the C semantics, in order to obtain a correct program; the effect of different pragmas on code generation, in order to obtain the desired assembly; and finally the performance of the assembly on the actual hardware. Since the goal of this work is to explore how low-level algorithm changes affect performance on a specific architecture, the higher-level tools – particularly, C with pragmas – introduce considerable noise without introducing expressiveness to compensate.

2. The second consideration was the ease of gaining knowledge through experimentation. Given a C program which heavily uses intrinsics, an optimizing compiler will most

likely produce a much more performant program compared to handwritten assembly. However, the extra optimization steps can obfuscate our experiment results, weakening our understanding of how the hardware reacts to small, precise changes. This understanding directly drives our algorithm design, which has a far greater potential upside than automatic optimization alone.

3. Another consideration was the principle of least power, as articulated by Tim Berners-Lee in [1]. Generating a proper C abstract syntax tree would require considerable effort, and generating templated strings would be very fragile. Neither representation could be conveniently used by tools other than the compiler. On the other hand, an assembly AST would be easy to implement, and it could be gradually extended to support higher-level semantics. Small tools may then independently interact with this AST. Such tools may range from prettyprinters, to reordering transformations, to full interpreters. Separating these concerns leads to a cleaner and more reliable architecture.

4. The final consideration was continuity with existing codebases. The `libxsmm` library already emits plain assembly, and it would be straightforward to reimplement its microkernel using an assembly AST. Each successive investigation would build upon this basic codebase, leading to a lower-risk software engineering process.

The second major design decision was the choice of language for the code generator itself. The two main requirements were that the language be amenable to rapid prototyping, be well-suited as a base for lightweight domain specific languages (DSLs), and have convenient access to numerical libraries. Two languages closely fit this niche: Python and Julia. Of the two, Python won due to its practicality – the current SeisSol code generation framework is already written in Python, the libraries are mature, and the code can be run directly on the cluster.

Julia, in comparison, has several notable advantages. Firstly, it has a much stronger type system, leading to both better performance and better correctness checking. Secondly, it is homoiconic, making it very amenable to metaprogramming. Thirdly, it is JIT-compiled on top of LLVM, which allows for features such as generating and displaying assembly code straight from the REPL.

Ultimately Julia was considered too risky because the language and libraries are changing quickly. Furthermore, the benefit of being able to mix high-level and low-level programming paradigms was cancelled out by the lack of support on the cluster: when run locally, the Julia interpreter must not emit AVX-512 instructions. The prospect of implementing the ideas explored in this work as Julia macros remains intriguing, but also well out of scope.

Two downsides to using Python emerged over the course of the work. Firstly, refactoring proved to be surprisingly time-consuming, given the lack of type safety. This was partially

mitigated by leveraging the new type annotation syntax along with the mypy type checker. Use of another new but irresistible syntax, 'f'-string interpolation, constrained the code's compatibility to Python 3.6, which is not installed on the cluster either.

These considerations lead to a strategic decision to keep a clear distinction between code generation and runtime, thus avoiding real metaprogramming. The code generation framework is architecture independent, written in pure Python, and emitting output in plain text. This plain text file happens to be a C++ program containing inlined AVX-512 instructions, which must be uploaded to the cluster, compiled, and run there. The upside of this approach is that it is simple and it allows the user to manually tinker with the generated C++/assembly. The downside is that it substantially increases the time and effort it takes to obtain feedback after making changes.

4.2 Abstract Syntax Tree

The most basic piece of machinery we need is a Python representation of the assembly code we emit. This is fundamentally an abstract syntax tree (AST) whether we recognize it or not, so we choose to recognize it. There is an inherent conflict between choosing data representations which are optimal for humans understanding the AST, optimal for custom tools modifying the AST, or optimal for producing textual assembly code. This is acerbated by Python's lack of constructor polymorphism and encapsulation. The human benefits from readability and a tower of abstractions which successively add specificity such as register length, precision, alignment, label names, and possibly even register names. The automated tools benefit from having a guarantee than any AST node is fully articulated, while still allowing select pieces to be abstract. Finally, the assembler needs every piece of information, repeated for every instruction, in a somewhat idiosyncratic format.

The solution to this conundrum is to decouple these representations. The AST is stripped down to be nothing more than a data container with a constructor that enforces some basic completeness invariants. The fields are chosen to be as *complete* and *orthogonal* as possible. The goals of being simple to work with, and of having a close mapping to the resulting assembly, are both disregarded with prejudice. Responsibility for the former is moved to the DSL, and responsibility for the latter is moved to the prettyprinter. This separation of concerns opens up long-term possibilities such as emitting C intrinsics instead of plain assembly, and in the short term it keeps the code legible.

The syntax tree is built up in stages of increasing complexity. All AST nodes inherit from the abstract base class `AsmStmt`, which enforces the existence of an `accept()` method, a `comment` string, and nothing else.

4.2.1 Operands

Operands are the primitive 'nouns' in this language. They consist of constants, labels, registers, and memory addresses. They can all be stringified into pseudocode or into GAS syntax. However, they are simpler than GAS's own concept of operands. For example, the FMA instruction expresses a broadcast operand like this: `16(%%rsi)%{1to8%}`, whereas our language makes the operand be a plain memory address `(rsi, None, 1, 16)` and passes responsibility for realizing that this operand needs to be broadcasted on to the `fma()` instruction. The weird GAS syntax for achieving this exists only in the GAS prettyprinter.

4.2.2 Instructions

An instruction is an AST node which can be mapped into one or more assembly instructions. This work currently only requires seven different instructions, which are described below.

Move Wraps `mov`, `vmovapd`, and `vmovupd`, using type information to choose.

Fma Wraps `vfmadd231pd`. Can optionally broadcast from a MemoryAddress. The choice of precision should eventually be determined from type information. Specifying mask registers will be needed as well.

Add Wraps `add` for now; should eventually handle the vectorized version as well.

Compare Compares two Operands, without getting confused over which is the left-hand side and which is the right.

Jump Jumps to an Operand; jump is indirect if the Operand is not a Label. Also supports local labels with direction specifiers. Unlike GAS, the direction specifier is part of the Jump node, not part of the Label. Whether or not a label is represented as local is independent of the underlying Label itself.

DeclLabel Declares a label.

DeclData Pastes a `Label` or a `Constant` directly into the code segment, wrapping the `db,dw,dd,dq` instructions. It is useful for creating jump tables or constant arrays. It could be modified to target other segments as well.

4.2.3 Blocks

Blocks are the first level of structure. A block is simply a list of AST nodes, along with a comment. Blocks are valuable because they make it possible to recover a stack trace *through*

the code being generated if the code generation fails. If the code generation succeeds, the prettyprinter indents based on block structure, making the code far more readable.

A common pattern inside the code generators is to have a function which takes certain parameters and returns a Block. This is used to generate reusable components, such as the inner part of MicroSparse, or register block movements. This is only slightly better than a preprocessor, because there is nothing controlling that the AST nodes inside the block make any sense together. Such functions usually benefit from having assert statements to enforce preconditions necessary for the generation to succeed.

4.2.4 Forms

Forms are the next level of structure. Whereas a block behaves like a simple container, forms behave more like a macro. They accept as arguments one or more blocks, and then transform these arguments however they see fit. This is how JumpTables are implemented. They accept Blocks containing each of the MicroSparse kernels, and they arrange them around the necessary labels and jumps. While this could be done using just a function that returns a Block, doing so would throw away the higher-level information about what is really going on. By keeping this information, we are essentially adding new syntax to our AST.

4.2.5 Commands

Commands are experimental and don't currently work. Their purpose is to express higher-level ideas which require knowledge of the program's state at runtime. This should be contrasted with Forms, which manipulate the AST without any such knowledge. Commands allow information – that was previously known only at runtime – to be moved to compile time. The motivating example is the matrix sparsity pattern. When we wish to move a pointer to B to the next block, and the block size is irregular, *and we are unrolling the loop*, we can do so. We can track the location of the pointer as each new instruction is emitted, and use the sparsity pattern to calculate the offset to the start of the next block. Commands generalize this idea, and shall be explored further in Section 4.5, once we have covered matrix cursors.

4.3 Interacting with an AST

The mechanisms for interacting with an AST were deliberately separated from the data structure itself in order to enforce a clean separation of concerns. We end up with two main

interfaces: a human-facing interface for assembling ASTs, and a machine-facing interface for traversing and modifying them.

The human interface is pure syntactic sugar. All of the KNL registers and datatypes are defined as global variables along with functions which are defined as conveniently as possible for constructing different AST nodes. A 'machine context' stores sensible defaults for choices like precision, alignment, and eventually perhaps vector length and available ISA extensions. An eventual goal is to make these swappable, so that the same generator may output AVX2 instead of AVX512 simply by loading a different module.

When it comes to organizing instructions into blocks, there are two options available. The first is a BlockBuilder class, which allows the programmer to add AST nodes to the current block, open a deeper block, or close the current block and move to its parent. This is particularly convenient when a loop is being manually unrolled. The second option is an interface inspired by S-expressions, where a block is merely a function call, and its contents the argument list. This is more convenient for laying out nested, static code such as non-unrolled loops.

The machine interface is a Visitor Pattern. A Visitor Pattern allows a method to be defined polymorphically for all AST nodes in an isolated, self-contained, extensible way while maintaining static type safety. In practice what this means is that the AST nodes all have an `accept(self, visitor)` method, and each new 'method' is in reality a Visitor class which contains specialized methods `visitMov(), visitFma(), visitAdd(),` etc. Thus every function which might wish to traverse the AST lives in its own separate module.

Visitors currently exist for prettyprinting the AST into valid GAS syntax, prettyprinting into pseudocode, counting FMA instructions, and determining which registers have been used or modified. In the future they could be used for static type checking, instruction reordering, or abstract interpretation.

4.4 Cursors

The *matrix cursor* is an abstraction that allows the code generators to cleanly traverse a sparse matrix and access its contents while only using logical coordinates. The name was chosen because of its similarities to the cursor in a text editor, which must translate movement and insert commands from 2D screen coordinates into a 1D memory location which depends on the lengths on the neighboring lines. The term 'cursor' has already been overloaded to describe a kind of iterator into a database; this is also a kind of iterator. MatrixCursor performs the following tasks:

Traversing a matrix by blocks. MatrixCursor translates logical block coordinates `(Bki,`

`Bni)` into a movement instruction. This instruction updates a register pointing to the start of the 'active' block. The logical block coordinates may be either absolute or relative to the active block.

Optionally tracking which block is active. Generating a GEMM is much simpler if the generator can model the state of the machine while deciding which code to emit in order to modify said state. Not all generators implemented currently do this; some emit loops which do not get simulated, and then assume that the loops maintain certain invariants. This is the start of a more theoretical discussion in Section 4.5.

Accessing cells within a block. MatrixCursor translates logical cell coordinates `(bki, bni)` into a `MemoryAddress` operand. Cell coordinates are relative to the top-left-corner of the active block. It also generates a comment string (which can be rendered alongside the assembly) indicating which logical coordinates lie behind the otherwise inscrutable memory address.

Handling the case of empty blocks and cells. The generators may directly query whether a given block or cell exists. If the generators attempt to move to an empty block or access an empty cell, the MatrixCursor will throw an exception because there is no meaningful memory address which can be associated with that logical location. Preemptively checking for these cases makes the generation code much cleaner.

Retrieving information about the active block. This information includes the block shape `bk, bn`, sparsity pattern, and pattern index. This is inessential, but has proven convenient for implementing nested block decompositions.

This is achieved through the interface in Listing 4.1.

The purpose of a matrix cursor is to separate concerns in order to keep the algorithm generation logic as clean and simple as possible. Otherwise there would be a proliferation of minor variations of the same few algorithms. The complexity which is encapsulated includes the following considerations:

Handling different sparsity patterns The sparsity pattern enters the code generator as either an MTX file or a `Matrix[bool]`, and immediately gets processed by a `Cursor`. The generation algorithm interacts with the sparsity pattern primarily by calling `cursor.has_nonzero_cell(...)`. On the other hand, if the generation algorithm needs a *recursive* block decomposition, it is free to extract the sparsity pattern for a given block and wrap that in a new Cursor.

Handling different block decompositions The generation algorithm is free to iterate over m, n, and k blocks without having to explicitly consider their shape or number. It can also cleanly handle the case of non-constant block sizes, but this option must always

be opt-in, as some generators, particularly TiledSparse, make regularity assumptions that this would violate.

Handling different matrix formats Various cursors currently support column-major dense, column-major sparse, CSC, CSR, BCSC, and BCSR matrix formats. Future work could readily extend this to support diagonal formats as well. It is also important here that the generators validate that they can handle the choice of format.

Handling different memory addressing schemes The choice of memory addressing scheme strongly affects instruction size, notably in the case of FMAs. Scale-index addressing is preferred, but this requires maintaining a collection of registers whose sole purpose is to be part of the memory address, and then choosing a combination of base and index registers such that the offset stays under 128.

The following cursor implementations currently exist. A future cursor could support variable block sizes.

TiledCursor assumes a single sparsity pattern which tiles perfectly.

DenseCursor supports column-major and row-major dense matrix formats including padding. It is used to traverse every dense matrix. Under the hood, it is a facade around TiledCursor.

MiniCursor handles the case of a sparse matrix which has not been divided into blocks. Under the hood, it is also a facade around TiledCursor.

BlockCursor handles the case

GeneralBlockCursor Note: While this has not yet been implemented, it is technically straightforward.

4.5 Symbolic execution and reification

In a very general sense, the GEMM generators implemented here all take information which was previously discovered at runtime and move it to compile time. Up until now, this activity has been denoted by the verb "unroll". However, this term is somewhat misleading. It comes from the concept of loop unrolling, a standard compiler optimization. Unrolling a loop is a path-independent transformation of an abstract syntax tree which has no access to new information. Unrolling a sparsity pattern is a path-dependent transformation of an (even more) abstract syntax tree which depends on the local program state space. A better word for this activity might be *reification*.

The pseudocode for UnrolledSparse and TiledSparse differ only by a single line, but their implementations differ substantially. This is because there is a deep asymmetry between how a looped block and an unrolled block interact with MatrixCursor. The critical question is "Is a MatrixCursor stateful or not?" An unrolled block which traverses a MatrixCursor needs to track the current CursorLocation one way or another. (Even a simple iteration variable is in effect simulating the state of the program's runtime.) On the other hand, a looped block knows nothing about this, and instead must rely on assumed invariants, such as a constant, perfectly tiled sparsity pattern.

In order to unify the concepts of a loop and an "unrolled" (reified) loop, so that we may automatically generate assembly code for both using the same tools, it is necessary to thread the machine state consistently throughout the code generation. This is effectively a symbolic execution of the code.

Because the AST can be extended to support higher-level concepts using Forms and Commands, this symbolic execution need not only concern itself with transformations of individual registers and memory addresses. It can work at an arbitrary abstraction level. In fact, pieces of a higher-level interpreter already exist: the Matrix, RegisterBlock, and MatrixCursor classes work together to capture most of the matrix-multiplication semantics. If the methods on these classes were recasted as Commands, we would end up with a high-level imperative language specifically for matrix multiplication. Executing this language on the interpreter would translate the commands down to assembly code.

4.6 User Manual

4.6.1 sparsemmgen

`sparsemmgen` is the primary entry point for the program. It requires a minimal set of parameters: the choice of algorithm generator, values of m, n, k, lda, ldb, ldc, and the path to an MTX file containing the sparsity pattern. It emits a C function containing inline assembly straight to standard out. For closer control over the generators it accepts optional parameters: values of bm, bn, bk, the output filename, the output function name, and output format. The latter allows the user to choose between GCC, GAS, or pseudocode.

If the user does not provide the optional parameters, the generator will attempt to choose values automatically based off of heuristics derived via experiment and a little bit of brute force.

4.6.2 libxsmmproxy

While `sparsemmgen` is designed to allow the user to control the algorithm generator precisely, it suffers from two drawbacks. Firstly, the burden of choosing the best algorithm and tuning it correctly is put entirely on the user. Secondly, if an existing codebase has been sufficiently optimized such that its developers would consider using the tools developed here, the integration would likely be very difficult, since the information `sparsemmgen` needs is unlikely to be present. Reducing the barrier to integration increases the likelihood of this tool being used.

`libxsmmproxy` addresses both of these points. It is a command-line program which matches the interface of `libxsmm_gemm_generator` exactly and can be integrated into an existing codebase by changing a path or adding a symbolic link. (No plans for creating a JIT version exist at this time.) The program analyzes its arguments and decides whether to use the `sparsemmgen` implementation or to simply delegate to the original `libxsmm_gemm_generator`.

Unfortunately, the information which the `libxsmm_gemm_generator` interface provides only partially overlaps with the information which `sparsemmgen` needs. On one hand, `libxsmm_gemm_generator` does not know about our generators or their parameters such as block sizes. On the other, it supports different microarchitectures, single precision, and different prefetching strategies, which this work currently does not. Thus the proxy must pattern-match for the KNL architecture, double precision, and no prefetching strategy, and delegate all other requests. Then it must conjure the generator choice and its parameters.

There are two approaches to supplying the missing information, one manual and one automatic. The manual approach is a whitelist, in pure Python, which maps specific MTX files to `Parameters` objects. It is advantageous insofar as it gives the user precise control, and disadvantageous insofar as it is repetitive and requires knowledge of the system. It is best combined with the automatic system described next.

In theory, automatically choosing the best generator involves determining, for each generator:

- Can the generator handle the posed problem at all?
- What is the best choice of parameters for this (problem, generator)?
- What is the expected performance of this choice?

and then choosing the (generator, parameters) which offer the best performance. In the general case this is a combinatorial optimization problem, but in practice the matrix sizes are very small and the parameter space is very constrained, particularly when only allowing perfect tilings. Thus a brute-force approach is satisfactory for the time being.

In the case of dense-by-sparse multiplication, a brute-force approach is not even necessary most of the time. If the number of nonzeros in the B matrix is below the limit imposed by the instruction cache, there is no reason to use any generator besides UnrolledSparse. If it is greater, UnrolledSparse is out of the question, and the controller need only choose between TiledSparse, BlockedSparse, GeneralSparse, and dense. TiledSparse and BlockedSparse are likely to perform very well on certain sparsity patterns but not in the general case, so the automatic system can default to using GeneralSparse. There will be a point at which the penalty of performing indirect jumps outweighs the saved FLOPs, in which point the system should simply use the dense algorithm. This we can determine experimentally.

4.6.3 runexperiment

The experiments discussed in the next chapter can all be reproduced by calling `runexperiment`. The experiments rely on generated data being in the correct directory, so it is best to clone the repository on the cluster and run it from there. A Python wrapper for interacting with SLURM is provided, and it supports remoting over SSH. If the SLURM setup breaks down, there are also recipes in the Makefile for building and running each experiment once the C++ code has been generated.

```
class Cursor:
    def has_nonzero_cell(self,
                         current_block: CursorLocation,
                         target_block: Coords,
                         target_cell: Coords) -> bool:
        pass

    def has_nonzero_block(self,
                          current_block: CursorLocation,
                          dest_block: Coords) -> bool:
        pass

    def move(self,
             current_block: CursorLocation,
             target_block: Coords) -> Tuple[AsmStmt, CursorLocation]:
        pass

    def look(self,
             current_block: CursorLocation,
             target_block: Coords,
             target_cell: Coords) -> Tuple[MemoryAddress, str]:
        pass

    def start_location(self, target_block: Coords) -> CursorLocation:
        pass

    def get_block(self, current_block: CursorLocation, target_block: Coords) -> BlockInfo
        pass
```

Source Code 4.1: MatrixCursor interface

5 Experiments

5.1 Performance comparison of practical examples

The goal of this first experiment is to show that the general approach can yield a speedup on matrices which are of practical interest. In the case of SeisSol, the compute kernels are recursively nested matrix multiplications of the pattern $Q' \mathrel{+}= K \cdot Q \cdot A$, where K and A are sparse and Q is dense. The A matrix always has the same shape and sparsity pattern, whereas there are many different matrices shaped like K. The performance of various kernels on the $Q \cdot A$ multiplication is depicted in Figure 5.1.

Unfortunately, the case of the $K \cdot Q$ multiplication can not be simply plugged in to SeisSol due to the asymmetry between dense and sparse multiplication discussed in Section 2.3. However, if the dense matrices were stored in row-major format instead of column-major, the situation would be reversed. Because the K matrix is larger, this might yield a greater speedup on SeisSol as a whole.

The experimental setup is straightforward. The problem size is set as $m = 40,\ n = 15,\ k = 9$, corresponding to a viscoelastic rheological model as described in [13]. The sparsity pattern used is shown on the right in Figure 1.1. The only variable measured is wall clock time, which was used to calculate speedup relative to `libxsmm`'s dense kernel. The two new kernels tested are UnrolledSparse and GeneralSparse. The only available degree of freedom for matrices this small is the m-block size; the two extremes, $bm = 8$ and $bm = 40$, were both considered. The TiledSparse and BlockSparse kernels, on the other hand, were not considered because they would either degenerate to UnrolledSparse or produce complete fill-in, depending on the chosen block sizes bn, bk. The performance of Breuer's sparse kernels was also tested for comparison.

The experiment shows that the UnrolledSparse kernel with $bm = 8$ has the best performance, corresponding to a speedup of 1.83 over dense. To put this in context, the upper bound on the speedup for this problem is $(n \cdot k)/nnz = 3.97$. The UnrolledSparse with the larger block size performs slightly worse, at 1.74. The GeneralSparse with the larger block size performed similarly worse than its UnrolledSparse equivalent, which was expected due to the indirect jump penalty. Meanwhile, the GeneralSparse with the small block size experienced a slowdown.

The automatically generated Breuer kernel also experienced a slowdown relative to dense, which was not surprising considering the architectural disadvantages it is working against. However, an examination of the code revealed a bug: the SIMD vector length was being constrained to 32 bytes instead of 64. Manually fixing this led to a speedup of almost 1.5 over dense, significantly better but still less than the UnrolledSparse kernel. The performance of the modified Breuer is expected to be similar for small matrices but then diverge as they get larger and denser, due to Breuer's lack of an accumulator for blocks of C.

This experiment can be repeated for different patterns of K, and also for the A-matrix sizes $n \in \{27, 36\}$.

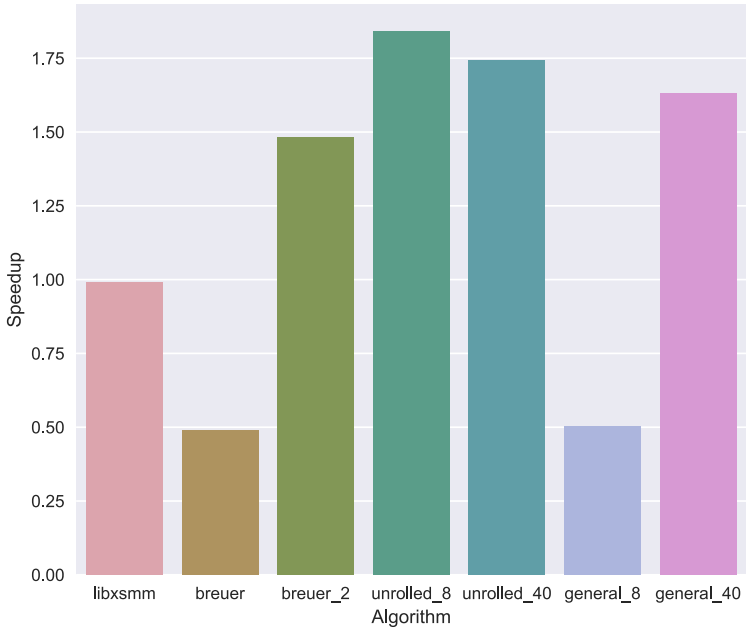


Figure 5.1: Speedup of different sparse GEMM kernels for the 9×15 SeisSol 'star' matrix, relative to the dense libxsmm implementation. Higher is better. UnrolledSparse outperforms the alternatives regardless of choice of block size. A minor modification of Breuer tripled its performance, but it remains below UnrolledSparse.

5.2 Scaling of UnrolledSparse

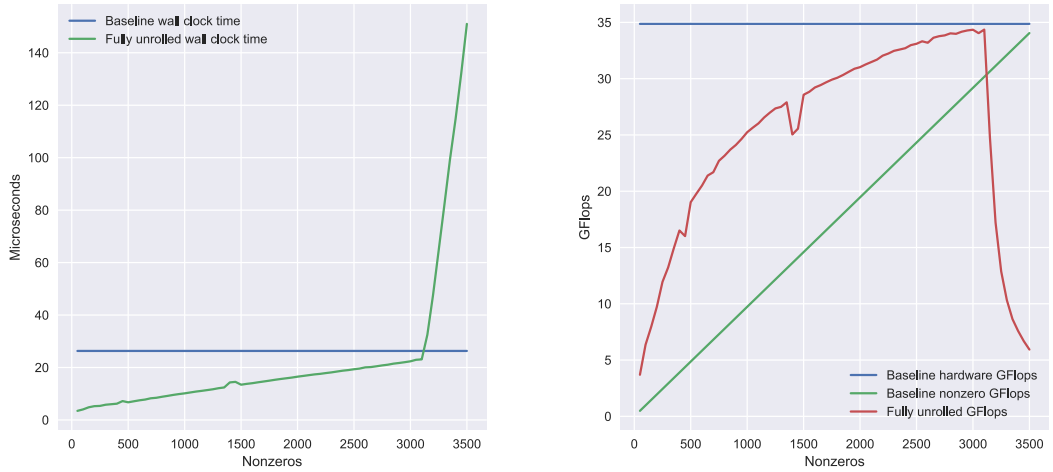
The goal of this experiment is to determine the range of problems which benefit from UnrolledSparse, and examine how its performance varies over this range. The focus is on matrix density rather than matrix size. UnrolledSparse is expected to have an upper limit on the number of nonzeros in B, and it is desirable to find this limit experimentally. Below it, the amount of computation is expected to scale according to $2 \cdot m \cdot nnz$, but because this also reduces the arithmetic intensity, it remains to be seen how the time to solution scales.

The experimental setup is as follows. The matrix sizes are taken to be $m = 128, n = 28, k = 128$, and they are blocked according to $bm = 8, bn = 28, bk = 4$. These numbers were chosen to try to match the blocking behavior of `libxsmm`. The B matrix uses CSC format and is filled with a varying number of nonzeros. Time-to-solution is measured and the achieved FLOP/S rate is calculated.

The results are shown in Figure 5.2. The instruction cache limit is reached at around 3100 nonzeros. Because the implementation tested does not use scale-index addressing, all but the first 128 FMA instructions take up 10 bytes apiece. Thus, for 3100 nonzeros, the 32KB instruction cache must contain 30616 bytes of FMAs. The remainder is presumed to be filled with load/store instructions (which are also larger than necessary). If scale-index addressing is used instead, the upper limit is estimated to be around 4500.

Below this limit, there is a clear linear relationship between nonzeros and time. There is a constant penalty of around 4 microseconds. If the instruction cache limit had not been reached, the dense kernel would have outperformed the sparse kernel starting around 3400 nonzeros, when B is around 95% dense.

Although the UnrolledSparse demonstrates an improved time to solution, it performs less overall computation than the dense version, paradoxically reducing the flop count. When considering the performance of the dense kernel, a distinction must be drawn between "hardware" GFlops $:= 2 \cdot m \cdot n \cdot k \cdot t^{-1}$ and "nonzero" GFlops $:= 2 \cdot m \cdot nnz \cdot t^{-1}$. The former reflects the work performed by the processor, whereas the latter reflects the work which was useful. Figure 5.2b shows that the UnrolledSparse GFlops are bounded above by the dense hardware GFlops and bounded below by the dense nonzero GFlops. Encouragingly, the UnrolledSparse GFlops starts close to the nonzero GFlops and asymptotically approaches the hardware GFlops as the nonzeros increase. This means that the parallel efficiency improves as the matrix becomes dense.



(a) Time to solution

(b) Performance

Figure 5.2: Scaling study of UnrolledSparse, varying the number of nonzeros. B is a 128×28 sparse matrix filled with a random sparsity pattern; it becomes dense at $nnz = 3584$ on the far right. The instruction cache limit is reached around $nnz = 3100$; below this limit the time-to-solution scales linearly and produces a speedup.

5.3 Choice of block sizes

The previous sections demonstrated the utility of UnrolledSparse kernel, and also showed that the choice of block sizes bm, bn, bk affects the overall performance. The KNL architecture places a number of constraints on the optimal block size, which is described in depth for the dense case by [7]. The goal of this experiment is to test the performance of UnrolledSparse for different block sizes and see to what extent the empirical sparse results match with the theoretical dense analysis. This can also lead to heuristics for automatically choosing the block sizes.

For the experimental setup, we chose $m = n = k = 96$ because it has a lot of convenient divisors and it is relatively large. This not only reduces measurement noise, but also stresses the data cache and possibly the instruction cache. Two cases were considered, $nnz = 500$ and $nnz = 2000$. The sparsity pattern was generated from a uniform random distribution. Kernels were generated for every choice of $bm \in \{8, 16, 24, 32\}$ and $bn, bk \in \{1, 2, 3, 4, 6, 8, 12, 16, 24\}$ satisfying the MicroSparse constraint $(bk + bn) \cdot (bm/8) \leq 32$.

The speedups of each kernel relative to dense are shown in Figure 5.3. For $nnz = 500$,

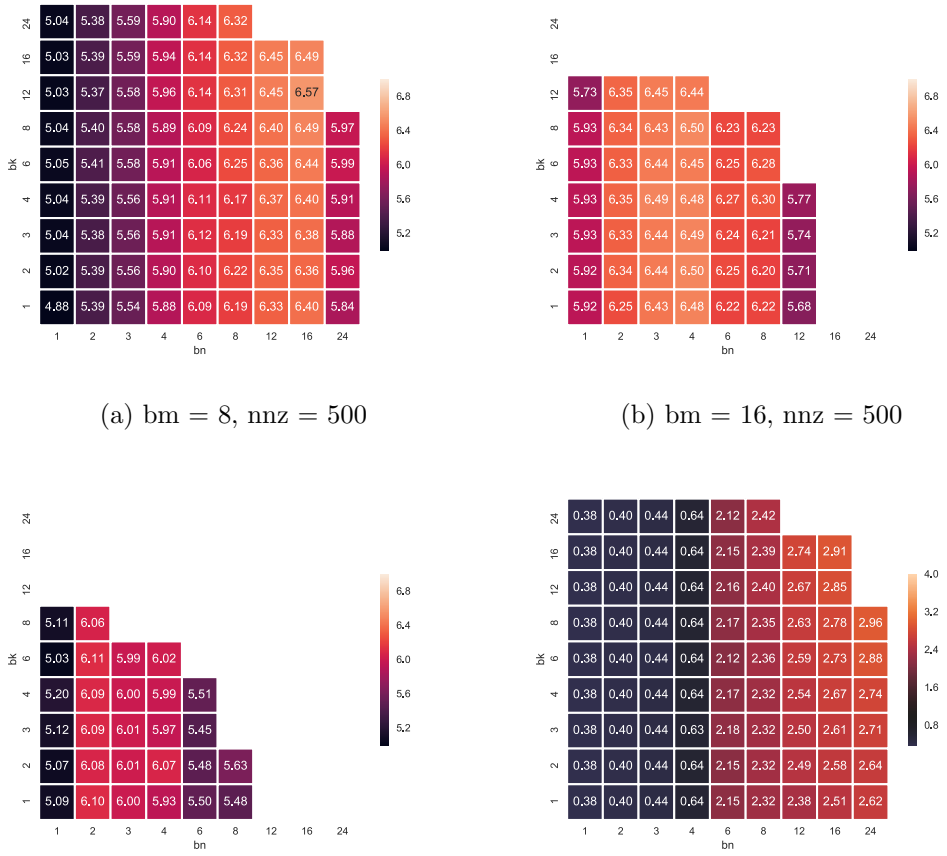
the theoretical max speedup is 18.4 and the observed max speedup is 6.57, implying an efficiency of 0.36. Meanwhile, for $nnz = 2000$, the theoretical speedup is 4.61 and observed is 2.96, an efficiency of 0.64. This is good news: not only did the efficiency *improve* as more nonzeros were added, but the best speedup happened at similar block sizes. The next step is to understand these block sizes.

The first observation is that for any choice of (bm, bn), the performance is effectively independent of bk. This makes sense because the unrolled Bk loop is the innermost, so the choice of bk ultimately only affects the ordering of FMAs instructions, relative to loads of columns of A, within the same panel. Inefficiencies here can be resolved by the reorder buffer. `libxsmm` approaches this differently: instead of unrolling a Bk loop of arbitrary size, it tackles the entire panel at once, keeping the current block of A in a ring buffer of 2-4 registers, and manually scheduling its load operations. An earlier experiment suggested that the choice of scheduling did not strongly affect performance, so the ring buffer approach was abandoned in favor of the conceptually simpler block unrolling.

The second observation is that the best choice of bn is directly tied to bm. For now consider only the $nnz = 500$ case. The maxima happen at $(bm, bn) \in \{(8, 16), (16, 4), (24, 2)\}$, and decrease as bm increases. Thus $bm = 8$ is preferred as long as the matrix dimensions support a tiling where $bn \in \{12, 16\}$. Otherwise a tiling with $bm = 16, bn \in \{1, 2, 3, 4\}$ is better.

This is consistent with libxsmm, which abandoned their 2D blocking of C (16×3) in favor of a 1D blocking (8×30) specifically for knights landing (KNL). They argue that the movement of columns of A into registers needs to be minimized relative to the number of FMAs in that block because of the narrow out-of-order execution issue width discussed earlier. Some of this reasoning applies to UnrolledSparse, since a smaller bn

Now let's turn to the $nnz = 2000$ case. The pattern looks similar for $bm = 8, bn \in \{8, 12, 16, 24\}$, but becomes qualitatively different for all $bm > 8$ or $bn < 6$. This is a plane in the problem space, beyond which performance falls off a cliff, again due to the instruction cache limit described previously.



(a) bm = 8, nnz = 500

(b) bm = 16, nnz = 500

(c) bm = 24, nnz = 500

(d) bm = 8, nnz = 2000

Figure 5.3: Speedup of a 96×96 random sparse matrix with different choices of block size. For smaller nnz, the speedup varies smoothly, every choice yields a speedup, and simple heuristics can be derived. For larger nnz, there are performance cliffs at $bn < 6$ and $bm > 8$.

5.4 Scaling of GeneralSparse

The UnrolledSparse approach is fundamentally unable to scale beyond a certain number of nonzeros, and the GeneralSparse approach seeks to ameliorate this by reintroducing some limited control flow and filling in zero entries. The extent of the fill-in depends solely on the distribution of the original sparsity pattern. On the other hand, the time to solution is expected to increase linearly with the number of jumps. We would like determine the value

of this *jump penalty*, and also determine whether it is sensitive to the number of nonzeros in B.

The experimental setup is as follows. The matrix sizes are kept fixed at $m = n = k = 64$ and the B matrix is randomly filled with $nnz \in \{400, 800, \ldots, 2800\}$. The block sizes are chosen as $bm = bn = 8$, with $bk \in \{4, 8, 16\}$ determining the number of jumps. For comparison, we test libxsmm and UnrolledSparse kernels as well. The results are shown in Figure 5.4.

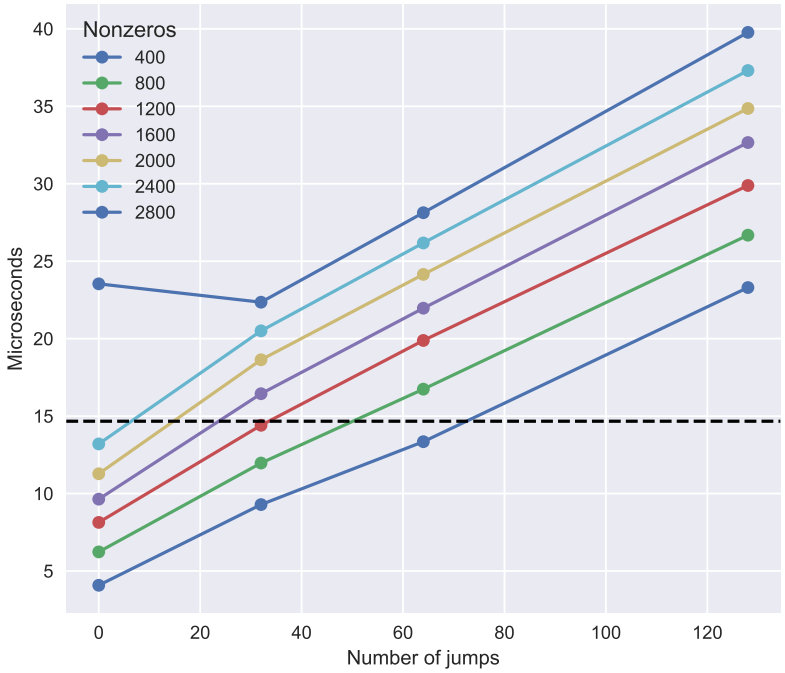


Figure 5.4: Scaling analysis for GeneralSparse. The number of jumps corresponds to the total number of nonzero blocks in the B matrix. The measurements in the $njumps = 0$ column came from UnrolledSparse, whereas the others came from GeneralSparse. The dashed line indicates the performance of the dense libxsmm kernel.

By averaging the slopes of each line, the jump penalty can be quickly estimated at around 0.2 microseconds/jump. (Recall that in this case, a 'jump' consists of a direct jump to a MicroSparse kernel, followed by an indirect jump from a register back to the calling location.) Though the penalty does become slightly higher for larger numbers of nonzeros, the two variables are close enough to be approximated as independent. This plot also suggests that the jumps are not hurting our pipeline performance. If they were, we would

expect the penalty to be steeper for lower nnz instead.

This new knowledge can be directly applied to auto-tuning GeneralSparse parameters. For each permissible block decomposition, the block patterns can be successively merged until they fit in the instruction cache. The jump count can be calculated from the block dimensions and number of empty blocks. The generator can then predict the performance of the block decomposition by estimating the UnrolledSparse performance for the new pattern, and adding the penalty for each jump.

It should be noticed that most of the measurements shown performed worse than libxsmm. Because GeneralSparse currently delegates to MicroSparse, the block sizes are small. GeneralSparse is only needed when B is relatively large, and $njumps = Bm \cdot Bk$ grows quickly. The solution is to allow for arbitrarily large blocks by delegating to UnrolledSparse instead. In its current condition, GeneralSparse is still useful in cases when B is tall and skinny or short and fat, or when B has many empty blocks.

6 Conclusion

This work approached the problem of speeding up a sparse matrix multiplication kernel on a variety of different levels. At one level, the goal was to make a very specific matrix multiplication faster. At a higher level, the goal was to create a generator which could produce a kernel for a wide range of input problems. Once this was done, it naturally followed to see how the kernel scaled across the space of input problems, and to analyze the effect of different parameter choices. As this analysis revealed flaws which could be fixed via small algorithmic changes, a family of different generators emerged. Though these generators made very different assumptions, they looked tantalizingly similar in pseudocode and quite different in implementation. This led to a search for a more powerful and abstract code generator, which could mix the concepts of compile time and runtime in a useful way.

This entire work is peppered with short remarks about work left for the future. The most important is probably to find a sparse kernel for column-major sparse-by-dense products.

Bibliography

[1] Tim Berners-Lee. Axioms of web architecture, 1998 (Accessed: 2018-02-10). http://www.w3.org/DesignIssues/Principles.html.

[2] A Breuer, Alexander Heinecke, M Bader, and Christian Pelties. Accelerating seissol by generating vectorized code for sparse matrix operators. 25:347–356, 01 2014.

[3] Alexander Breuer, Alexander Heinecke, and Yifeng Cui. Edge: Extreme scale fused seismic simulations with the discontinuous galerkin method. In Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes, editors, *High Performance Computing*, pages 41–60, Cham, 2017. Springer International Publishing.

[4] Jack J. Dongarra, Lain S. Duff, Danny C. Sorensen, and Henk A. Vander Vorst. *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[5] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, June 2002.

[6] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[7] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. Libxsmm: Accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 84:1–84:11, Piscataway, NJ, USA, 2016. IEEE Press.

[8] Intel. Mkl developer reference, (Accessed: 2018-02-10). https://software.intel.com/en-us/mkl-developer-reference-c-mkl-cscmm.

[9] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2016.

[10] Martin Kser, Michael Dumbser, Josep De La Puente, and Heiner Igel. An arbitrary high-order discontinuous galerkin method for elastic waves on unstructured meshes iii.

viscoelastic attenuation. *Geophysical Journal International*, 168(1):224–242, 2007.

[11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[12] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, March 2016.

[13] C. Uphoff and M. Bader. Generating high performance matrix kernels for earthquake simulations with viscoelastic attenuation. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 908–916, July 2016.