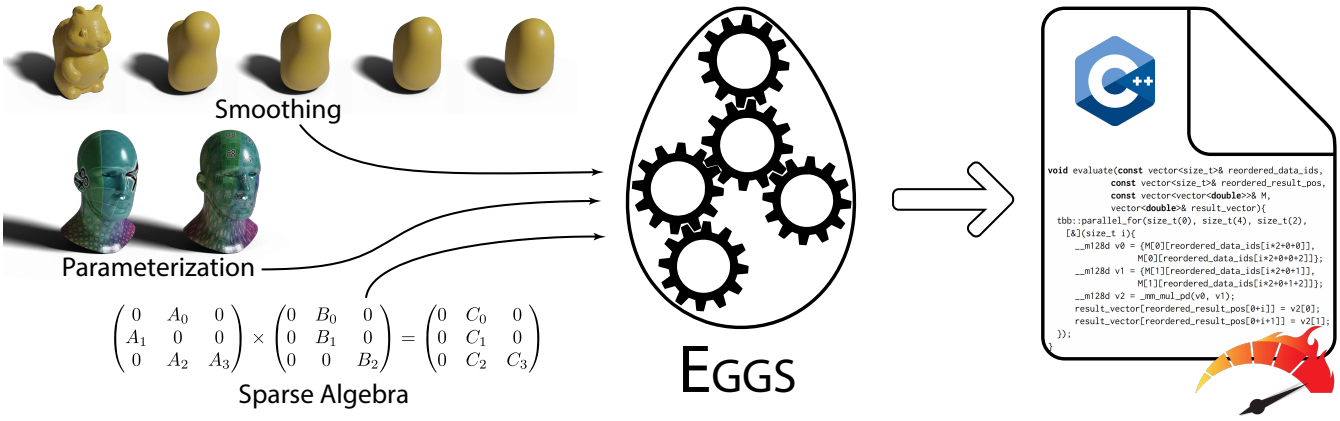
Eurographics Symposium on Geometry Processing 2020
Q. Huang and A. Jacobson
(Guest Editors)

Volume 39 (2020), Number 5

EGGS: Sparsity-Specific Code Generation

Xuan Tang[1], Teseo Schneider[1], Shoaib Kamil[2], Aurojit Panda[1], Jinyang Li[1], and Daniele Panozzo[1]
[1] New York University [2] Adobe Research



Abstract

Sparse matrix computations are among the most important computational patterns, commonly used in geometry processing, physical simulation, graph algorithms, and other situations where sparse data arises. In many cases, the structure of a sparse matrix is known a priori, but the values may change or depend on inputs to the algorithm. We propose a new methodology for compile-time specialization of algorithms relying on mixing sparse and dense linear algebra operations, using an extension to the widely-used open source Eigen package. In contrast to library approaches optimizing individual building blocks of a computation (such as sparse matrix product), we generate reusable sparsity-specific implementations for a given algorithm, utilizing vector intrinsics and reducing unnecessary scanning through matrix structures. We demonstrate the effectiveness of our technique on a benchmark of artificial expressions to quantitatively evaluate the benefit of our approach over the state-of-the-art library Intel MKL. To further demonstrate the practical applicability of our technique we show that our technique can improve performance, with minimal code changes, for mesh smoothing, mesh parametrization, volumetric deformation, optical flow, and computation of the Laplace operator.

1. Introduction

Linear algebra operations are at the foundation of most scientific disciplines: Due to their importance, countless approaches have been proposed to improve the performance of their implementation, both on traditional processors and on graphical processing units. In existing implementations, sparse linear algebra operations are handled similarly to their dense counterparts: Every elementary operation (such as matrix product, matrix sum, etc.) is implemented in an individually-optimized function/kernel. Unfortunately, while dense linear algebra kernels have many opportunities for vectorization and parallelization, efficiently executing their sparse linear algebra counterparts is challenging. This is because the data structures for sparse matrices require iterating through multiple index arrays in order to access the non-zeros elements. Doing so results in irregular memory access that depends on the sparsity pattern of the input/output matrices, that is, on the location of non-zero elements. Consequently, sparse kernels often lead to complex implementations that are hard to optimize.

In this paper, we propose a new computational paradigm for generating efficient kernels for linear algebra expressions or algorithms working on sparse inputs, including sparse linear algebra and sparse matrix assembly. Our proposed approach is motivated by two optimization opportunities that are missing from existing implementations.

First, it is common in applications to have the sparsity pattern of input/output matrices remain the same while the actual values of

non-zero elements change during the computation. We can generate an efficient implementation by specializing it to a specific sparsity pattern of inputs/outputs. Existing implementations do not perform such specialization. A few libraries (e.g., MKL's 2-stage routines) offer an option to dynamically evaluate the sparsity pattern, caching intermediate results to reduce the runtime. However, the sparsity pattern is only used in a limited fashion (e.g., for memory allocation) and there is no code generation and specialization.

Second, applications typically need to combine multiple operations together into linear algebra expressions (e.g., $A^T DA + C$). However, existing implementations use a separate kernel for each operation, incurring the overhead of writing intermediate matrices to main memory and reading them back later. As memory bandwidth is the bottleneck resource in sparse computation, we can achieve large performance gains by generating a kernel implementation that composes multiple operations together.

To generate sparsity-specific, composed implementations, we unroll arbitrary linear algebra expressions (or even algorithms) into expression trees, one for each non-zero element of the final output, generate and compile a kernel for that specific expression. The structures of the expression trees are determined by the sparsity pattern of inputs and we specialize the generated code according to these tree structures, resulting in unstructured, but fixed, set of operations performed on dense vectors. These operations can be optimized during the compilation, do not require memory allocation for temporaries, and can be trivially parallelized over multiple threads. With this approach, the unnecessary iteration through sparse matrix data structures is completely eliminated, and no intermediate matrices are created, reducing the problem to an unstructured, but fixed, set of operations performed on dense vectors. Such an approach is particularly beneficial for iterative computation, as the cost of code generation can be amortized across multiple iterations as the sparsity patterns of matrices remain unchanged across iterations.

We extensively compare our CPU implementation of this approach, which we call EGGS, against the state-of-the-art commercial Intel Math Kernel Library (MKL) [Int12], both on single expressions (Section 4) and on complete algorithms (Section 5). We evaluate its scaling with respect to the size of the matrices, their sparsity, and the length/complexity of the expressions. Overall, our prototype implementation is faster by a factor of $2\times$ – $16\times$ depending on the specific expression, both in single and multithreaded mode. The downside is that the setup cost to prepare the expression-specific kernel is longer than MKL, making it competitive in applications where the same operation is evaluated multiple times with the same sparsity structure. These applications are common in scientific computing, geometry processing, and computer vision: We showcase 4 such applications in Section 5, including geometric distortion minimization, optical flow, cotangent matrix assembly, and smoothing. The complete source code of our reference implementation, the data used in the experiments, and scripts to reproduce our results are available at `https://github.com/txstc55/EGGS`.

2. Related Work

Dense Linear Algebra Libraries & Compilers. A large number of libraries have been built for dense matrix computations for shared memory machines [ABB*99, WD98, GJ*10, VDWCV11, San10, Int12, WZZY13], both historically and in recent years due to the proliferation of deep neural networks and their use of dense matrix computations to perform convolutions. While many such libraries are hand-written, automated techniques have become more prevalent as the number of architectures and variants have increased. PHiPAC [BAwCD97] pioneered the use of auto-tuning [AKV*14] to automatically generate and search over candidate implementations of linear algebra functions to find the best-performing version, obtaining performance that matched or beat hand-tuned vendor implementations.

Automatic parallelization and optimization of dense nested loops, such as those found in linear algebra, motivate many techniques used within general (non domain-specific) compilers [Wol82, WL91, MCT96], including polyhedral transformation techniques [Fea91, Fea88, IT88]. Recently, new special-purpose compilers have arisen for dense linear and tensor algebra, due to their use in convolutional neural networks, including compilers for TensorFlow [ABC*16, PMB*19] and other dense tensor operations [VZT*18]. Build to Order BLAS [NBS*15, BJKS09] is a compiler for dense linear algebra that composes together user-specified dense linear algebra operations and creates implementations using a combination of analytic modeling and empirical execution.

Sparse Linear Algebra Libraries. Compressed data structures, including Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) have existed for decades [TW67, McN71]. PETSc [BGMS97] is perhaps the most widely-used library for sparse computations in scientific computing. More recently, dense linear algebra libraries, including Eigen [GJ*10] and MKL [Int12] have also added sparse support, and are thus becoming increasingly used for sparse computation. A sparse version of the BLAS standard attempted to standardize the calling interfaces and functionality of these libraries [DHP02] while hiding implementation details of data structures. Like PHiPAC, the OSKI and pOSKI libraries [VDY05, BLYD12] pioneered the use of auto-tuning for sparse linear algebra, but support very few computational kernels and cannot compose them.

Sparse Linear Algebra Compilers. Early work in compiling sparse linear algebra includes a series of papers by Bik and Wijshoff [BW93, BW94] that transformed dense matrix code into sparse matrix code. SIPR [PS99] is an intermediate representation with a similar purpose. The key insight in Bernoulli [KPS97] was that using the language of relational algebra could enable compilers to generate efficient sparse matrix code. However, none of these systems utilize sparsity structure to generate code tailored to particular inputs.

Polyhedral techniques have been applied to sparse linear algebra [VHS15, MYC*19, SHO18], using *inspector-executor* transformations, which modify the code to first use an inspector to determine parts of the matrix structure, and an executor to use the information from the inspector to perform efficient computation. Most related to our work, Cheshmi et al. use *compile-time* inspection followed by sparsity specific code generation to create efficient implementations of sparse direct solvers [CDKS18, CKSD17]. Unlike

their work, we concentrate of generic sparse algorithms and compositions of linear algebra operations and maintain the interface used by Eigen. Rodríguez and Pouchet [RP18, ASPR19] use polyhedral tracing to transform sparse matrix-vector multiply (SpMV) into a series of dense affine loops and obtain nearly 30% performance improvement despite making the code size much larger. In contrast, we compose together operations that use sparse operands (rather than only SpMV) and use a more naïve code generation strategy that groups outputs by the structure of their computation trees.

The Tensor Algebra Compiler project (taco), aims to build a compiler for dense and sparse linear and tensor algebra that supports a large variety of sparse and dense data structures [KKC*17, KAKA19, CKA18]. Our approach could be integrated in taco, to allow it to generate sparsity-specific code. The current public version of taco does not correctly support sparse outputs, making a direct performance comparison impossible[†].

Domain-Specific Languages for Sparse Graphics Applications. Opt [DMZ*17] is a domain-specific language for sparse non-linear least-squares optimization problems, which can produce efficient GPU code from high-level problem descriptions. Opt enables users to produce *matrix-free* implementations, which avoid materializing full sparse matrices; like EGGS, such implementations do not need to iterate through sparse structures. EGGS deals with more general computations and can directly optimize existing code that uses Eigen. However, it targets only CPU computation, while Opt supports the generation of GPU kernels.

Simit [KKRK*16] and Ebb [BSL*16] are domain-specific languages that allow users to avoid complex indexing required to assemble matrices used in simulations. Programmers do not directly perform assembly, but rather use local stencil operations that together form the (implicit or explicit) sparse matrix. EGGS deals instead with general sparse computations, and maintains the idea of explicit assembly while generating code that avoids unnecessary computations.

3. Methodology

EGGS generates C++ code with vector intrinsics and parallelism for general algorithms including linear algebra operations by overloading the Eigen API. The code generation occurs in three major steps: first, the programmer specifies an algorithm and its input (more specifically, the sizes and sparsity structure of the input matrices) using the Eigen API; then, EGGS executes the operations *symbolically*; and finally, the results of symbolic execution are used to generate optimized code. To use this code, the programmer needs only fill the values arrays of the input matrices and vectors. The overall algorithm is shown in Algorithm 1.

3.1. Symbolic Execution

EGGS implements a new datatype, `SymbolicNum`, which represents a symbolic tree of computations: the leaves of the tree are either constants or entries of a matrix/vector. All other nodes of the tree represent operations such as addition and multiplication. This datatype is used instead of the usual value types (e.g., `double` and `float`) in Eigen.

Before performing symbolic execution, EGGS replaces the values of an input by the corresponding `SymbolicNum` leaf (lines 2–4 in Algorithm 1). Each `SymbolicNum` leaf is a tuple consisting of the variable name from which a value was read and the location of the value within the variable. The location is the index of the

Input: matrix operands M_i,
scalar operands c_j,
expression $G_{in} = E(M_0, .., c_0, ..)$
Result: generated code F

```
// Convert matrix/vector operands to abstract
   matrices
// L is the computed set of all SymbolicNum leaves.
```
1 $L \leftarrow \emptyset$
2 **foreach** *non-zero index* $i \in M_j$ **do**
3 $\quad L \leftarrow L \cup \{(i, j)\}$
4 **end**
```
// Perform overloaded operation using abstract
// inputs
// Entries of G are trees of operations
// with abstract input entries as leaves
```
5 $G \leftarrow E(M_0, ..., c_0, ..)$
```
// Find unique output trees
```
6 $T \leftarrow \emptyset$
7 $I_O \leftarrow \emptyset$
8 **foreach** $O_i \in G$ **do**
9 $\quad O \leftarrow O_i$
10 $\quad idxs \leftarrow []$
11 $\quad$ **foreach** $leaf(i,k) \in O$ **do**
```
        // Replace concrete entry with wildcard.
        // leaf is a reference to the entry
```
12 $\quad\quad leaf \leftarrow (i, ?)$
```
        // Append index to idxs list
```
13 $\quad\quad idxs \leftarrow idxs :: k$
14 $\quad$ **end**
```
    // Add wildcarded output tree to T
    // if it doesn't already exist in the set
```
15 $\quad T \leftarrow T \cup O$
```
    // Append idxs to list that corresponds
    // to the wildcarded tree
```
16 $\quad I_O \leftarrow I_O :: idxs$
17 **end**
```
// Generate code
```
18 **foreach** $T_i \in T$ **do**
```
    // k is the number of leaves in tree T_i
    // |I_i| is the number of instances of that tree
```
19 $\quad F \leftarrow F ::$ codegen_loop(T_i, I_O, $|I_i|$, k)
20 **end**

Algorithm 1: Overall algorithm for compiling a sparsity-specific kernel using EGGS.

[†] We communicated with Taco's authors, who confirmed this shortcoming in an issue on their public code repository https://github.com/tensor-compiler/taco/issues/297.

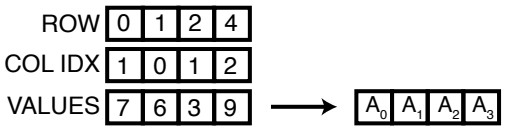


Figure 1: *Prior to symbolic execution, we transform the values array of the CSR matrix A to contain symbolic entries corresponding to the location in the array.*

$$\begin{pmatrix} 0 & A_0 & 0 \\ A_1 & 0 & 0 \\ 0 & A_2 & A_3 \end{pmatrix} \times \begin{pmatrix} 0 & B_0 & 0 \\ 0 & B_1 & 0 \\ 0 & 0 & B_2 \end{pmatrix} = \begin{pmatrix} 0 & C_0 & 0 \\ 0 & C_1 & 0 \\ 0 & C_2 & C_3 \end{pmatrix}$$

$C_0 = A_0 \times B_1$ $\quad C_1 = A_1 \times B_0$ $\quad C_2 = A_2 \times B_1$ $\quad C_3 = A_3 \times B_2$

Figure 2: *For $C = AB$ with all sparse operands, we show the computation trees after symbolic execution. In this example, only a single uniquely-structured computation tree covers all outputs in C.*

value: for dense vectors and matrices this is merely an offset from the beginning. For sparse matrices, which are stored in compressed sparse row (CSR) or column (CSC) forms, the location is an index into the *values* array. Figure 1 shows an example of this initial transformation for a CSR matrix.

EGGS uses C++ operator overloading to intercept arithmetic operations and perform symbolic execution. In order to minimize the amount of memory required, EGGS stores `SymbolicNum` objects in a memory pool and allows parent nodes to reference other subtrees within the pool, instead of duplicating the subtrees.

EGGS use of C++ operator overloading allows it to piggyback onto existing Eigen code for execution, allowing EGGS to implement symbolic execution in very few lines of code, and allowing it to be used to accelerate existing C++ code that already uses Eigen by simply changing the matrix template type. When the usual Eigen functions finish execution, the result is a matrix, vector, or scalar of `SymbolicNum`, which we process further to generate efficient code. Figure 2 shows example computation trees for a sparse matrix multiplication.

3.2. Generating Efficient Code from Symbolic Results

EGGS generates a compute loop for each *uniquely-structured* computation tree in the output structure. In this context, two trees are equivalently-structured if they contain the same non-leaf nodes, the number of leaves are equal, and the leaves load from the same input arrays. If two result entries are equivalently-structured, we can use the same compute loop in both cases, by simply changing which entries of the input operands we use.

Identifying Uniquely-Structured Computation Trees. The process for identifying unique computation trees is shown in lines 6–17 of Algorithm 1. Iterating through the result array, EGGS first replaces each leaf of the tree with a wildcard, representing any possible input location. After wildcard replacement, EGGS checks whether the wildcarded tree already exists in the collection of unique trees, and adds it if necessary. For the example in Figure 2, only one unique computation tree is generated.

During this process, EGGS also builds an index list for each unique tree (I_O in line 16 of Algorithm 1). Since each tree has a unique number of inputs, no additional information is required during code generation; this index list can be used directly by consuming the correct number of inputs during each call.

```
void evaluate(const vector<size_t>& reordered_data_ids,
              const vector<size_t>& reordered_result_pos,
              const vector<vector<double>>& M,
              vector<double>& result_vector){
  tbb::parallel_for(size_t(0), size_t(4), size_t(2),
    [&](size_t i){
      __m128d v0 = {M[0][reordered_data_ids[i*2+0+0]],
                    M[0][reordered_data_ids[i*2+0+0+2]]};
      __m128d v1 = {M[1][reordered_data_ids[i*2+0+1]],
                    M[1][reordered_data_ids[i*2+0+1+2]]};
      __m128d v2 = _mm_mul_pd(v0, v1);
      result_vector[reordered_result_pos[0+i]] = v2[0];
      result_vector[reordered_result_pos[0+i+1]] = v2[1];
  });
}
```

Figure 3: *Generated parallel vectorized code for computing the output sparse matrix C from Figure 2.*

Generating Code. The final step outputs a single function that computes the output sparse array. For each uniquely-structured tree, EGGS generates a loop nest (lines 18–20) that computes the output entries corresponding to that tree.

We rely on the compiler to pack inputs into vectors and to generate efficient code for storing vector lanes into output memory locations (see Section 3.3 for discussion). Producing the actual computation is straightforward: the code generator walks the wildcarded symbolic computation tree, generating vectorized intrinsics per operation. Figure 3 shows an example generated loop for the computation tree from Figure 2.

As shown in lines 18–20 of Algorithm 1, when EGGS generates code, it groups outputs by their unique computation tree; that is, first all outputs with the first unique computation tree are computed, followed by outputs using the second unique tree, and so on. Within each loop nest for a specific computation tree, the outputs are grouped into vector-width-sized outputs per loop iteration, in order to utilize vectorized computation code. Parallelism across cores is introduced by using the Intel TBB [Phe08] library to parallelize the per-tree compute loops. Thus, the generated code utilizes both parallel execution and vectorization, without requiring any additional effort from the user of our system.

Avoiding Redundant Computation. In some cases, multiple outputs may store the exact same value: two outputs may share not just the structure of the computation tree, but also include the same

leaf nodes. To avoid this redundant work, we pre-filter the set of outputs to store the same computed value in multiple locations. A more aggressive version could avoid even redundantly computing sub-trees, but in our current version we only avoid redundant computation if two trees are entirely equivalent.

Compressed Index Arrays. In addition to the output sparse values array, to generate a usable sparse CSR matrix EGGS must also produce the row start and column index arrays (or the column start and row index arrays if producing CSC output). Since these are already computed by Eigen when generating the code, EGGS embeds them into the generated code directly.

3.3. Limitations

Fundamentally, EGGS does not support data-dependent operations, such as those that arise in direct solvers for pivoting and other operations. As a result, EGGS does not currently support Eigen's sparse solvers, which generally rely on values in the matrices for making decisions such as pivoting. Instead, we expect most users will use EGGS to generate sparsity-specific code for constructing inputs to solvers, and to build code that operates on the solution returned. Sparsity-specific approaches to solvers, such as those in ParSy [CKSD17, CDKS18], are complementary to EGGS and can be applied where they lead to speedup. As we show in Section 4, EGGS maintains interoperability with existing Eigen solvers.

In the current implementation, we have not aggressively optimized the memory usage: a single node in the tree requires approximately 56 bytes, and each entry in the output is a tree made up of multiple nodes in a global list. Furthermore, increasing the number of operands in the original expression makes it more likely that each output tree is larger, resulting in large memory consumption at compile time. While this limitation only affects the precomputation phase (the generated code is oblivious to the memory required during its generation), it makes our system unable to generate code for long expressions.

Relying on the compiler to produce efficient vector packing/unpacking code may result in lower performance than if we generated code with more efficient loads when the input locations are contiguous. An alternative implementation would generate packed stores directly; in order to do so, the computation trees for each of the output locations in a vector must match as well as be contiguous. We leave such an implementation for future work.

4. Results

We implemented our system as a C++ code generator, which only requires using `SymbolicNum` as the basic numeric type of Eigen. With this change, our system generates, compiles, and executes an efficient kernel on-the-fly using the Clang compiler. Most arithmetic operations are supported and, in particular, we support sparse Eigen expressions as input, allowing us to generate highly-optimized kernels for algorithms working on sparse matrices.

We use two sets of tests in order to evaluate the performance of our approach. In Section 4.1, we compare our method on core sparse matrix expressions supported by both Eigen and MKL, showing that our method compares favorably to state of the art libraries. For all our core routines we use MKL two-stage computations; that is, the sparsity of the result of the operation is precomputed in a first stage (which we do not count in our measurements), while the result is computed in a second stage. We remark that this strategy, while similar to ours, can only be done for simple expressions in MKL and is done at runtime. We then experiment with compound expressions (Section 4.2) which are naturally supported by our approach, while requiring combinations of basic operations when using other libraries. Finally, we discuss integrating our algorithm into practical applications in Section 5.

We run our experiments on a workstation with two 10-core Intel(R) Xeon(R) E5-2660 v3 CPUs @ 2.60GHz with 128GB of memory and 50GB of swap space, running Linux kernel version 4.12.0 and Clang version 7.0.1-8. We compare both serial and parallel performance (limiting to 8 threads, which is the point of saturation, after which adding more cores no longer increases performance). To foster applicability of our approach and support the replicability of our results, we include our reference implementation and a script to reproduce all results in the paper as part of the supplemental material.

4.1. Core Routines

We perform a series of experiments using basic operations involving sparse matrix operands with sparse matrix outputs:

$$AB, \quad A^T DA, \quad \text{and} \quad A^T A, \tag{1}$$

where A, B are sparse matrices and D is a diagonal matrix.

For every test we generate a set of random input matrices, with 5 and 15 non-zero entries per row on average. In Figure 4 we compare the speedup of our method with respect to both Eigen and MKL as we change the size of the matrix and its density, while keeping the number of non-zero entries per row constant. Both the number and the positions of the non-zero entries are synthetic and targeted for benchmarking purposes; we study EGGS' performance on real matrices in Section 5. The advantage of our method grows with matrix size but is reduced as density increases. Speedup over Eigen is massive even with EGGS in single-threaded mode (between $10\times$ to $40\times$), and the speedup over the heavily-optimized MKL library (ignoring precomputation time for both EGGS and MKL) is between $3.5\times$ (AB with 15 non-zeros per row for each input, 1M rows) and $15\times$ (5 non-zero per row per input, 1M rows, for $A^T A$) for sizes and densities common in geometry processing applications. Parallelization further reduces running times (as shown in Figure 5), providing an even larger advantage for our approach.

Figure 6 shows the time EGGS requires for precomputation, including symbolic execution, code generation, and compilation. The detailed preprocessing timings for all the experiments in the paper are provided in Appendix A. This overhead demonstrates that EGGS is suitable for operations that will be executed numerous times, as the overhead is non-trivial. In Table 1 we collect all the raw timings for $A^T A$.

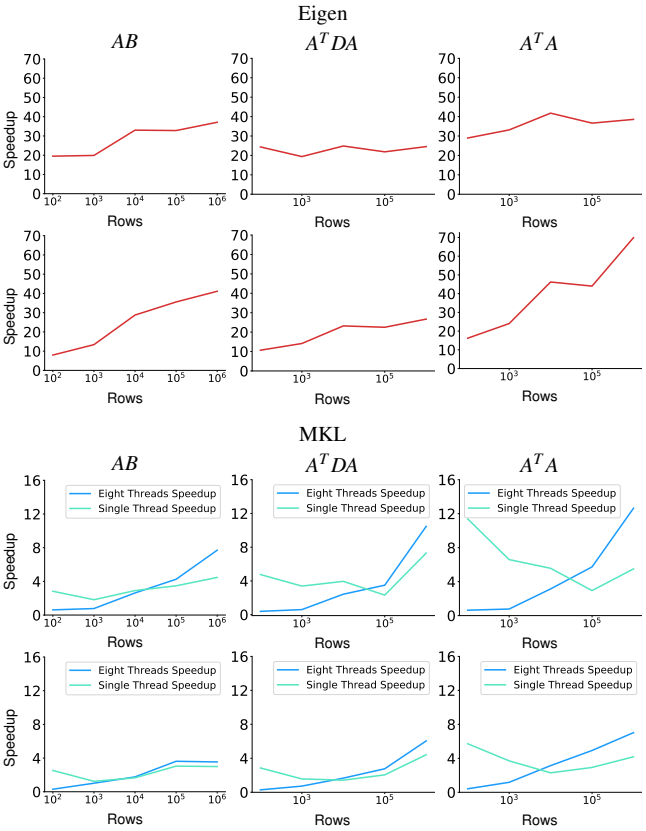


Figure 4: *Speedup of our method for increasingly large matrices compared with Eigen and MKL for the expressions in (1) for 5 (top) and 15 (bottom) non-zero entries per row. We show single and 8 threads speedups.*

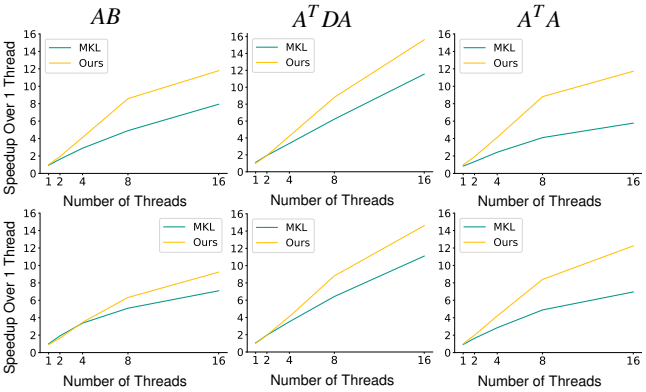


Figure 5: *Scaling results of our method compared with MKL for a matrix $10^6 \times 10^6$ and 5 (top) and 15 (bottom) non-zeros per row.*

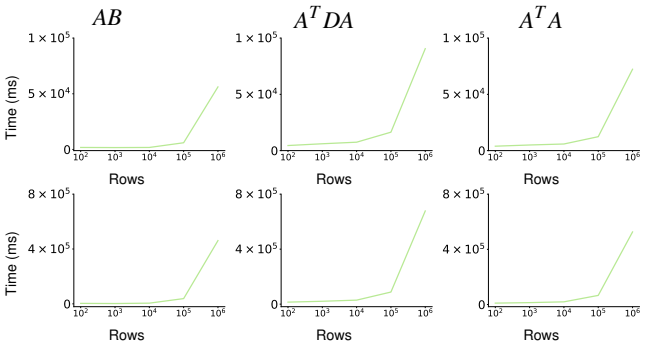


Figure 6: *Time needed to perform symbolic execution, code generation, and compilation of the kernel generated by our method, for 5 (top) and 15 (bottom) non-zeros per row.*

ROWS	10^2	10^3	10^4	10^5	10^6
EIGEN ST	6.37e-2	1.42e0	3.11e1	5.94e2	1.29e4
MKL ST	2.50e-2	2.83e-1	4.15e0	4.78e1	1.84e3
MKL MT	3.02e-2	8.51e-2	8.93e-1	1.30e1	4.84e2
OURS PC	3.92e3	4.99e3	5.87e3	1.24e4	7.23e4
OURS ST	2.20e-3	4.29e-2	7.44e-1	1.62e1	3.35e2
OURS MT	4.77e-2	1.10e-1	2.84e-1	2.26e0	3.83e1

ROWS	10^2	10^3	10^4	10^5	10^6
EIGEN ST	3.38e-1	1.06e1	3.87e2	5.26e3	1.35e5
MKL ST	1.19e-1	1.63e0	1.92e1	3.50e2	8.10e3
MKL MT	4.74e-2	3.09e-1	4.23e0	7.66e1	1.67e3
OURS PC	1.01e4	1.37e4	1.94e4	6.60e4	5.26e5
OURS ST	2.09e-2	4.41e-1	8.38e0	1.20e2	1.94e3
OURS MT	1.15e-1	2.63e-1	1.35e0	1.56e1	2.38e2

Table 1: *Timings in ms for $A^T A$ for 5 (top) and 15 (bottom) non-zeros per row. By ST and MT we denote multi- and single-thread performance and PC stands for precompute.*

4.2. Composite Expressions

We study the relative performance of our method, MKL, and Eigen on three composite expressions:

$$(\alpha A + B)^T(\beta B^T + C), \quad ABC, \quad \text{and} \quad (A+B)(A+B+C), \quad (2)$$

where A, B, and C are sparse matrices; α, β are scalars; and all outputs are sparse matrices. Figure 7 shows the speedup of our method. Our current subtree elimination is unable to reuse common subexpressions, thus leading to more operations than what other libraries perform, since those libraries can reuse intermediate results. Despite this, our method is still faster. We expect that the performance for these expressions could be further improved by adding a more aggressive policy for common subexpression elimination, which is an interesting direction for future work (see Section 6).

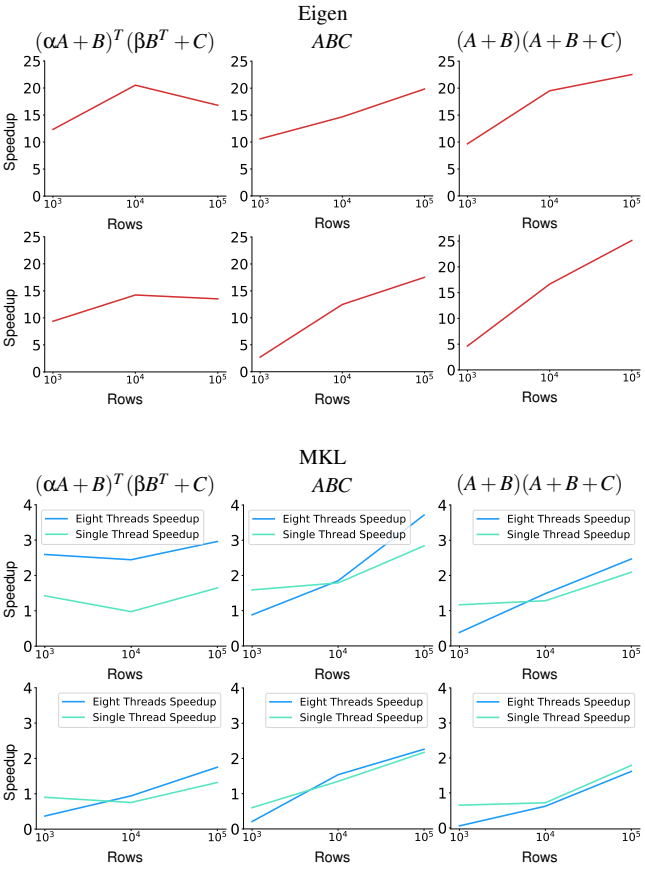


Figure 7: *Speedup of our method for increasingly large matrices compared with Eigen and MKL for the expressions in (2) for 5 (top) and 15 (bottom) non-zero entries per row. We show single and 8 threads speedups.*

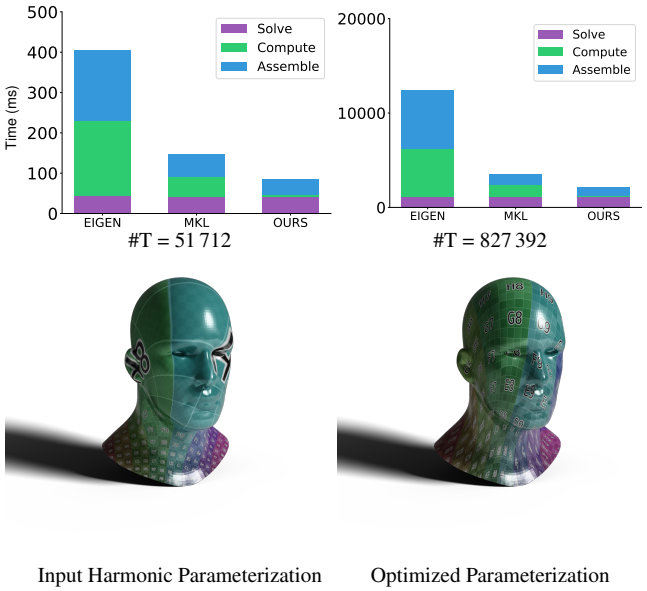


Figure 8: *Cutoff of the run time of computing a parameterization of a mesh with SLIM.*

5. Applications

Integrating our technique in existing applications that already use Eigen only requires minimal code changes. To demonstrate achievable speedups in real applications, we selected a few open-source applications and used our system to replace existing Eigen code with optimized kernels. For all these applications, the only required change was to switch the numerical type from double to `SymbolicNum`, in addition to the software engineering required to interface our system with the codes. The source for all these applications is available at `https://github.com/txstc55/EGGS`. Note that, for fairness, we use the optimized Pardiso solver [DCDBK*16, VCKS17, KFS18] wrapper in Eigen for all linear solves. In this section, we report the speedup with respect to the end-to-end algorithm, including the parts that are not optimized by our method, to provide a fair evaluation of the benefits that users adopting our system can expect. In all these applications, the ratio between matrix preparation (which we accelerate) and the linear solve heavily depends on the problem size: the larger the problem, the more dominant the solve time will be since it scales superlinearly, while the matrix preparation scales linearly. In our experiments, we show that the overall speedup is significant for problem sizes that are common in the respective applications. The detailed preprocessing timings are provided in Appendix A.

5.1. Geometric Distortion Minimization

Many geometry processing algorithm are based on the minimization of energies measuring the geometric distortion of a 2D or 3D simplicial mesh. We integrated our algorithm in the SLIM framework [RPPSH17] available in the libigl library [JP*18]. SLIM is a recent approach to minimize geometric distortion energies which relies on a proxy function to approximate the Hessian of the distortion energies. We anecdotally compare the difference of performance on two of the standard examples included in libigl (2D parametrization and 3D mesh deformation), by using our technique to generate an optimized kernel for the expression $A^T DA + B$ in the inner loop of the optimization.

2D Parametrization. Parameterization is a common task in computer graphics; the goal is to bijectively "flatten" a disk-like mesh to the plane while reducing distortion. By optimizing the inner loop of the optimization with our technique, we gain a 3× speedup *on the runtime of the whole algorithm*, as shown in Figure 8. We note that the algorithm also requires a linear solve, which, in the original Eigen/MKL implementation, is not the bottleneck. After our technique the time for assembly and non-solve computations are drastically reduced, making the actual solve become the slowest part of the algorithm.

3D Mesh Deformation. Animations and posing of characters often relies on handle-based mesh deformation: as the user moves around a set of anchors, an algorithm deforms the mesh by minimizing a physicaly-based distortion energy. By using our system to optimize the inner optimization loop, the end-to-end speedup is

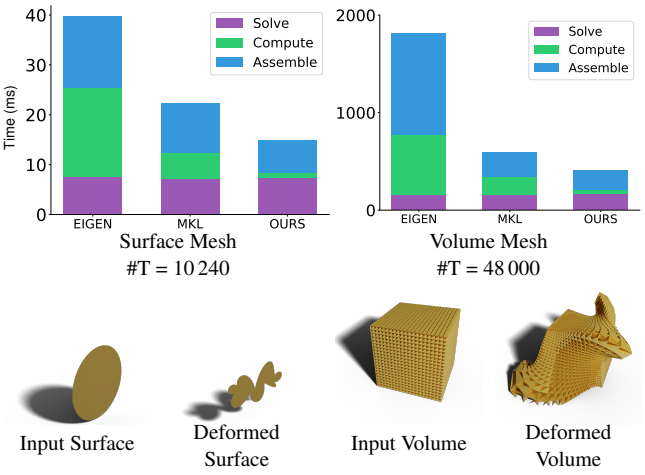
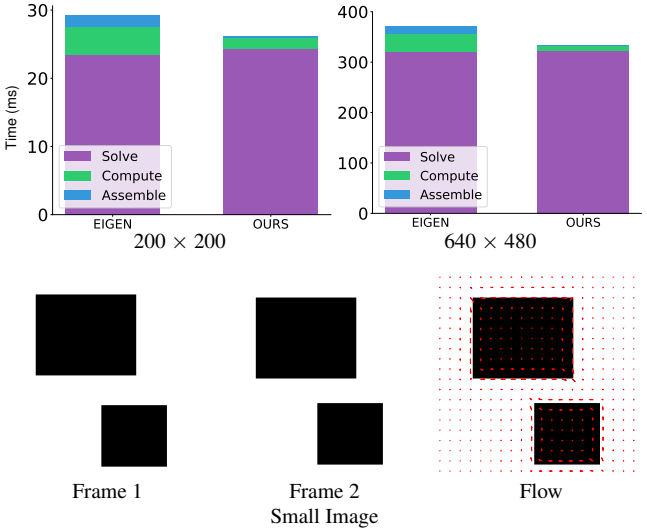


Figure 9: *Cutoff of the run time of computing mesh deformation.*



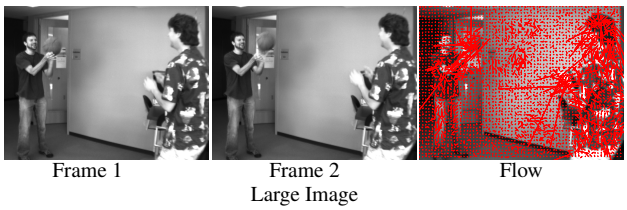


Figure 10: *Runtime of optical flow for our algorithm on a small and large image.*

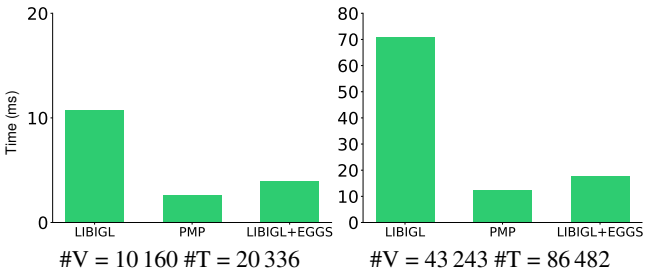


Figure 11: *Time for assembling the cotangent matrix with libigl, PMP, and using our approach to accelerate the libigl version.*

2.67× and 5× when applied to meshes with 10 240 triangles and 48 000 triangles respectively, compared to the original Eigen-based implementation (Figure 9). Note that this applications is usually interactive, and our speedup allows a single iteration to complete in 0.36 seconds instead of 1.8 seconds, reducing the time the user has to wait to see a preview of the deformation.

5.2. Optical Flow

Optical flow [HS81] is a common algorithm used in many computer vision tasks. It computes a displacement field that maps the pixels of one frame to the next. While real-time implementations are possible using a GPU kernel, in this section we analyze performance running on a CPU, since we leave the extension of our system to GPUs as a future work. The algorithm involves computing three differential operators E_x, E_y, and E_t which depend on the brightness $E_{i,j,k}$ of a pixel i, j for frame k. The operators are approximations of the brightness derivative with respect to x, y (pixel position) and t (frame in the video). Then, using these operators and a user-parameter smoothness control α, it solves a non-linear system of the form

$$(\alpha^2 + E_x^2)u^{i+1}E_xE_yv = (\alpha^2 u^i - E_xE_t)$$
$$(\alpha^2 + E_y^2)v^{i+1}E_xE_yu = (\alpha^2 v^i - E_yE_t),$$

which can be rewritten as $Ex^{i+1} = \alpha^2 x^i - b$. We use our method to compute a kernel that takes a pair of images and computes the sparse matrix E and right-hand side directly (Figure 10). For this application, since the operators act on a regular grid (i.e., the pixels of an image), the linear solve dominates the timings, leading to only 1.1× speedup for the full application. Note that an improvement of 10% is still relevant for such an application, especially if the algorithm is used to process long video sequences. If we ignore the solve time, and consider only the computations, our method is 5× faster than the Eigen implementation.

We note that, even if in theory for every pair of images one would require one assembly and several solves (i.e., until the iterative process x^{i+1} converges), in practice the iterations are initialized with the solution from the previous pair, thus requiring only one iteration (and one solve) per step.

5.3. Cotangent Matrix Assembly

Our method provides benefits even for algorithms generating sparse matrices, such as the assembly of the classical cotangent Laplacian matrix (Figure 11). Our method automatically converts the libigl [JP*18] "cotmatrix" function, which takes as an input the vertices and connectivity of a mesh and returns its cotangent Laplacian matrix, with a fully optimized kernel. Our method provides a 4× speedup on a large mesh and 2× on a smaller mesh. Note that the runtime of *automatically* using our method on the naïve libigl implementation is slightly higher than the hand-optimized assembly

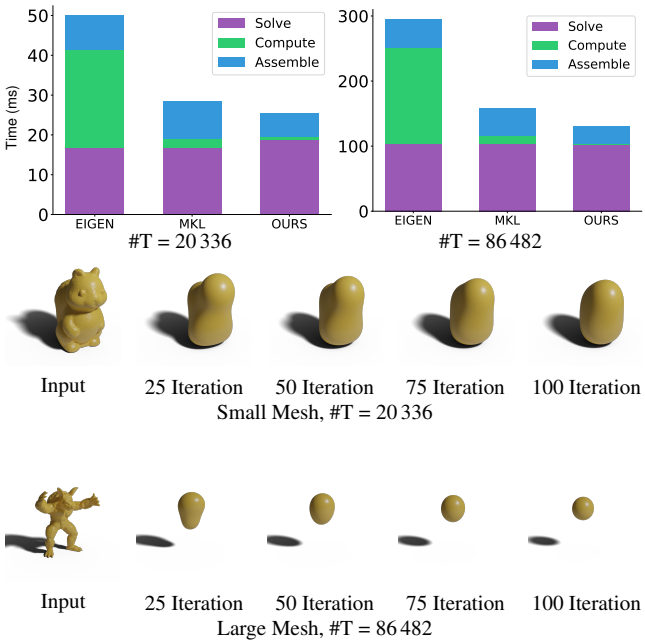


Figure 12: *Cutoff of the runtime of implicit Laplacian smoothing.*

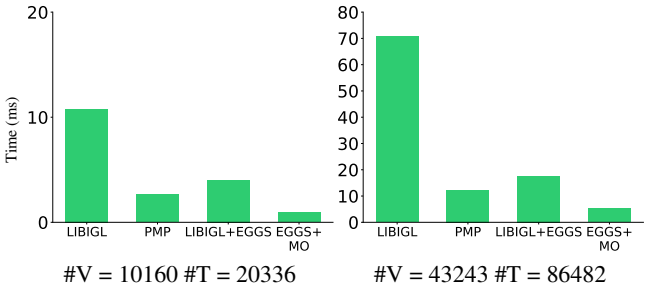


Figure 13: *Additional manual optimizations (EGGS + MO) can further reduce the running time.*

in PMP [SB20] (5ms vs 2.65ms for the small mesh and 20ms vs 13ms for the large one).

5.4. Smoothing

Implicit bi-Laplacian smoothing [KCVS98, DMSB99] is ubiquitous in geometry processing to remove high-frequency noise from surface meshes. The algorithm removes noise iteratively by solving the following linear system:

$$(L^T ML + wM)p' = wMp,$$

where p are the current vertices, p' are the unknown smoothed vertices, M is the lumped mass matrix, L is the area-weighted cotangent Laplacian ($L = M^{-1}L_w$ with L_w the cotangent matrix), and w is a user-controlled parameter deciding the strength of the filter. We use our method to replace the computation of $L^T ML + wM$, which changes at every iteration. With a classical Eigen implementation, the assembly of the linear system matrix has a comparable cost as the linear system solve, while with our method (and MKL) the bottle neck is the linear solve. EGGS obtains a 2× end-to-end speedup (1.1× compared to MKL) for a small mesh and 2.2× (1.2× versus MKL) for a large mesh. If we count only the optimized computation (i.e., the actual computation of the operator without solve) our method is 68× and 6.3× faster on average than Eigen and MKL respectively on the large mesh (Figure 12).

6. Concluding Remarks

We introduced a new paradigm and algorithm to automatically generate parallel vectorized kernels for algorithms involving sparse matrix and vector operations, and demonstrated that it can surpass the performance of commercial libraries on sparse linear algebra operations and that it provides practical speedups on geometry processing algorithms.

There are three major opportunities to further improve the benefits of this approach: (1) the code generation step could be extended to target the generation of parallel GPU kernels, thus providing an automated way to convert existing geometry processing algorithms to exploit the high parallelism of GPUs; (2) tree construction could be improved by finding common subexpressions and adding support for intermediate value computations; and (3) the memory and runtime could be further optimized to enable processing expressions with many more operands or even dense matrices. Preliminary experiments for (2) show (Figure 13) that, by manually finding common subexpressions for the case of the cotangent matrix assembly, it would be possible to further speed up the code by a factor of 4×, becoming faster than the hand-optimized library PMP.

To foster replicability of our results and adoption of our algorithm by the community, we have released the reference implementation of our algorithm and code for all the showcased applications as an open-source project at `https://github.com/txstc55/EGGS`. We hope that the community will integrate this solution into existing libraries based on Eigen, such as spectra [Qiu20], PolyFEM [SDG*19], and libigl [JP*18], or to other programming languages targeting sparse computation [KKRK*16, KKC*17].

Acknowledgements

This work was supported in part through the NYU IT High Performance Computing resources, services, and staff expertise. This work was partially supported by the NSF CAREER award 1652515, the NSF grants IIS-1320635, OAC-1835712, OIA-1937043, CHS-1908767, CHS-1901091, CNS-1816717, a gift from Adobe Research, a gift from nTopology, a gift from VMware, a gift from NVIDIA, and a gift from Advanced Micro Devices, Inc.

References

[ABB*99] ANDERSON E., BAI Z., BISCHOF C., BLACKFORD S., DEMMEL J., DONGARRA J., DU CROZ J., GREENBAUM A., HAMMARLING S., MCKENNEY A., SORENSEN D.: *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999. 2

[ABC*16] ABADI M., BARHAM P., CHEN J., CHEN Z., DAVIS A.,

DEAN J., DEVIN M., GHEMAWAT S., IRVING G., ISARD M., KUDLUR M., LEVENBERG J., MONGA R., MOORE S., MURRAY D. G., STEINER B., TUCKER P., VASUDEVAN V., WARDEN P., WICKE M., YU Y., ZHENG X.: Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 265–283. 2

[AKV*14] ANSEL J., KAMIL S., VEERAMACHANENI K., RAGAN-KELLEY J., BOSBOOM J., O'REILLY U.-M., AMARASINGHE S.: Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (New York, NY, USA, 2014), PACT '14, ACM, pp. 303–316. 2

[ASPR19] AUGUSTINE T., SARMA J., POUCHET L.-N., RODRÍGUEZ G.: Generating piecewise-regular code from irregular structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2019), PLDI 2019, Association for Computing Machinery, p. 625âĂŞ639. 3

[BAwCD97] BILMES J., ASANOVIĆ K., WHYE CHIN C., DEMMEL J.: Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing* (Vienna, Austria, Jul 1997). 2

[BGMS97] BALAY S., GROPP W. D., MCINNES L. C., SMITH B. F.: Efficient management of parallelism in object-oriented numerical software libraries. In *Modern software tools for scientific computing*. Springer, Birkhäuser Boston, 1997, pp. 163–202. 2

[BJKS09] BELTER G., JESSUP E. R., KARLIN I., SIEK J. G.: Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 59:1–59:12. 2

[BLYD12] BYUN J.-H., LIN R., YELICK K. A., DEMMEL J.: Autotuning sparse matrix-vector multiplication for multicore. *EECS, UC Berkeley, Tech. Rep* (2012). 2

[BSL*16] BERNSTEIN G. L., SHAH C., LEMIRE C., DEVITO Z., FISHER M., LEVIS P., HANRAHAN P.: Ebb: A dsl for physical simulation on cpus and gpus. *ACM Trans. Graph. 35*, 2 (May 2016). 3

[BW93] BIK A. J., WIJSHOFF H. A.: Compilation techniques for sparse matrix computations. In *Proceedings of the 7th international conference on Supercomputing* (1993), ACM, pp. 416–424. 2

[BW94] BIK A. J., WIJSHOFF H. A.: On automatic data structure selection and code generation for sparse computations. In *Languages and Compilers for Parallel Computing*. Springer, 1994, pp. 57–75. 2

[CDKS18] CHESHMI K., DEHNAVI M. M., KAMIL S., STROUT M. M.: Parsy: Inspection and transformation of sparse matrix computations for parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2018), SC '18, ACM. 2, 5

[CKA18] CHOU S., KJOLSTAD F., AMARASINGHE S.: Format abstraction for sparse tensor algebra compilers. *Proc. ACM Program. Lang. 2*, OOPSLA (Oct. 2018), 123:1–123:30. 3

[CKSD17] CHESHMI K., KAMIL S., STROUT M. M., DEHNAVI M. M.: Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2017), SC '17, ACM, pp. 13:1–13:13. 2, 5

[DCDBK*16] DE CONINCK A., DE BAETS B., KOUROUNIS D., VERBOSIO F., SCHENK O., MAENHOUT S., FOSTIER J.: Needles: Toward large-scale genomic prediction with marker-by-environment interaction. 543–555. 7

[DHP02] DUFF I. S., HEROUX M. A., POZO R.: An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Transaction on Mathematical Software 28*, 2 (June 2002), 239–267. 2

[DMSB99] DESBRUN M., MEYER M., SCHRÖDER P., BARR A. H.: Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques* (1999), SIGGRAPH âĂŹ99, p. 317âĂŞ324. 9

[DMZ*17] DEVITO Z., MARA M., ZOLLÖFER M., BERNSTEIN G., THEOBALT C., HANRAHAN P., FISHER M., NIESSNER M.: Opt: A domain specific language for non-linear least squares optimization in graphics and imaging. *ACM Transactions on Graphics 2017 (TOG)* (2017). 3

[Fea88] FEAUTRIER P.: Array expansion. In *2nd International Conference on Supercomputing (ICS'88)* (1988), ACM, pp. 429–441. 2

[Fea91] FEAUTRIER P.: Dataflow analysis of array and scalar references. *International Journal of Parallel Programming 20*, 1 (1991), 23–53. 2

[GJ*10] GUENNEBAUD G., JACOB B., ET AL.: Eigen v3. http://eigen.tuxfamily.org, 2010. 2

[HS81] HORN B. K., SCHUNCK B. G.: Determining optical flow. *Artificial Intelligence 17*, 1 (1981), 185 – 203. 8

[Int12] INTEL: *Intel math kernel library reference manual*. Tech. rep., 630813-051US, 2012. http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf, 2012. 2

[IT88] IRIGOIN F., TRIOLET R.: Supernode partitioning. In *Symposium on Principles of Programming Languages (POPL'88)* (San Diego, CA, January 1988), pp. 319–328. 2

[JP*18] JACOBSON A., PANOZZO D., ET AL.: libigl: A simple C++ geometry processing library, 2018. https://libigl.github.io/. 7, 8, 9

[KAKA19] KJOLSTAD F., AHRENS P., KAMIL S., AMARASINGHE S.: Tensor algebra compilation with workspaces. 180–192. 3

[KCVS98] KOBBELT L., CAMPAGNA S., VORSATZ J., SEIDEL H.-P.: Interactive multi-resolution modeling on arbitrary meshes. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (1998), SIGGRAPH âĂŹ98, p. 105âĂŞ114. 9

[KFS18] KOUROUNIS D., FUCHS A., SCHENK O.: Towards the next generation of multiperiod optimal power flow solvers. *IEEE Transactions on Power Systems PP*, 99 (2018), 1–10. 7

[KKC*17] KJOLSTAD F., KAMIL S., CHOU S., LUGATO D., AMARASINGHE S.: The tensor algebra compiler. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017), 77:1–77:29. 3, 9

[KKRK*16] KJOLSTAD F., KAMIL S., RAGAN-KELLEY J., LEVIN D., SUEDA S., CHEN D., VOUGA E., KAUFMAN D., KANWAR G., MATUSIK W., AMARASINGHE S.: Simit: A language for physical simulation. *ACM Trans. Graphics* (2016). 3, 9

[KPS97] KOTLYAR V., PINGALI K., STODGHILL P.: A relational approach to the compilation of sparse matrix programs. In *Euro-Par'97 Parallel Processing*. Springer, 1997, pp. 318–327. 2

[McN71] MCNAMEE J. M.: Algorithm 408: a sparse matrix package (part i)[f4]. *Communications of the ACM 14*, 4 (1971), 265–273. 2

[MCT96] MCKINLEY K. S., CARR S., TSENG C.-W.: Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS) 18*, 4 (1996), 424–453. 2

[MYC*19] MOHAMMADI M. S., YUKI T., CHESHMI K., DAVIS E. C., HALL M., DEHNAVI M. M., NANDY P., OLSCHANOWSKY C., VENKAT A., STROUT M. M.: Sparse computation data dependences simplification for efficient compiler-generated inspectors. In *Programming Languages Design and Implementation (PLDI)* (2019). 2

[NBS*15] NELSON T., BELTER G., SIEK J. G., JESSUP E., NORRIS B.: Reliable generation of high-performance matrix algebra. *ACM Trans. Math. Softw. 41*, 3 (June 2015), 18:1–18:27. 2

[Phe08] PHEATT C.: IntelÂő threading building blocks. *J. Comput. Sci. Coll. 23*, 4 (Apr. 2008), 298. 4

[PMB*19] PRADELLE B., MEISTER B., BASKARAN M., SPRINGER J., LETHIN R.: Polyhedral optimization of tensorflow computation graphs. In *Programming and Performance Visualization Tools* (Cham, 2019),

Bhatele A., Boehme D., Levine J. A., Malony A. D., Schulz M., (Eds.), Springer International Publishing, pp. 74–89. 2

[PS99] PUGH W., SHPEISMAN T.: Sipr: A new framework for generating efficient code for sparse matrix computations. In *Languages and Compilers for Parallel Computing*. Springer, 1999, pp. 213–229. 2

[Qiu20] QIU Y.: Spectra, 2020. URL: https://spectralib.org. 9

[RP18] RODRÍGUEZ G., POUCHET L.-N.: Polyhedral modeling of immutable sparse matrices. In *Proceedings of the Eigth International Workshop on Polyhedral Compilation Techniques* (2018), ACM. 3

[RPPSH17] RABINOVICH M., PORANNE R., PANOZZO D., SORKINE-HORNUNG O.: Scalable locally injective mappings. *ACM Trans. Graph. 36*, 2 (Apr. 2017). 7

[San10] SANDERSON C.: *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*. Tech. rep., NICTA, Sept. 2010. 2

[SB20] SIEGER D., BOTSCH M.: The polygon mesh processing library, 2020. http://www.pmp-library.org. 9

[SDG*19] SCHNEIDER T., DUMAS J., GAO X., ZORIN D., PANOZZO D.: Polyfem. https://polyfem.github.io/, 2019. 9

[SHO18] STROUT M. M., HALL M., OLSCHANOWSKY C.: The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of IEEE 106*, 11 (Nov 2018), 1921–1934. 2

[TW67] TINNEY W. F., WALKER J. W.: Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE 55*, 11 (1967), 1801–1809. 2

[VCKS17] VERBOSIO F., CONINCK A. D., KOUROUNIS D., SCHENK O.: Enhancing the scalability of selected inversion factorization algorithms in genomic prediction. *Journal of Computational Science 22*, Supplement C (2017), 99 – 108. 7

[VDWCV11] VAN DER WALT S., COLBERT S. C., VAROQUAUX G.: The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering 13*, 2 (2011), 22–30. 2

[VDY05] VUDUC R., DEMMEL J. W., YELICK K. A.: OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series 16*, 1 (2005), 521+. 2

[VHS15] VENKAT A., HALL M., STROUT M.: Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), PLDI 2015, pp. 521–532. 2

[VZT*18] VASILACHE N., ZINENKO O., THEODORIDIS T., GOYAL P., DEVITO Z., MOSES W., VERDOOLAEGE S., ADAMS A., COHEN A.: Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. 2

[WD98] WHALEY R. C., DONGARRA J.: Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing* (1998). 2

[WL91] WOLF M. E., LAM M. S.: A data locality optimizing algorithm. *SIGPLAN Not. 26*, 6 (May 1991), 30–44. 2

[Wol82] WOLFE M. J.: *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1982. AAI8303027. 2

[WZZY13] WANG Q., ZHANG X., ZHANG Y., YI Q.: Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 25:1–25:12. 2

Appendix A: Additional Statistics

We provide in Table 2 detailed statistics on EGGS preprocessing time and size of the generated binaries for all the experiments in the paper. We also report the corresponding preparation time for MKL. The table is divided into 3 parts: simple expressions (Section 4.1), composite expressions (Section 4.2), and applications (Section 5).

Name	NNZ	Rows	SE (ms)	CG (ms)	CC (ms)	Size	MKL (ms)
AB	5	10^2	1.269	2.029	2317.67	313K	24.803
AB	5	10^3	17.409	11.881	2223.02	308K	4.944
AB	5	10^4	186.002	112.479	1935.62	307K	5.152
AB	5	10^5	2839.94	1347.51	1950.49	307K	34.835
AB	5	10^6	34622.4	14472.4	1957.61	307K	271.076
ATDA	5	10^2	3.828	8.017	4585.26	352K	31.555
ATDA	5	10^3	36.904	24.464	6138.41	365K	4.687
ATDA	5	10^4	464.985	150.737	6829.11	370K	8.584
ATDA	5	10^5	5308.46	1939.52	8845.97	395K	43.899
ATDA	5	10^6	59186.3	20757.5	9957.09	400K	713.861
ATA	5	10^2	2.941	5.066	4000.36	348K	23.53
ATA	5	10^3	31.404	15.538	5045.37	361K	4.248
ATA	5	10^4	355.607	99.213	5505.74	366K	6.377
ATA	5	10^5	4404.83	1160.75	6744.52	383K	35.301
ATA	5	10^6	49235	14634.7	7368.92	388K	413.234
AB	15	10^2	6.897	15.154	4023.42	348k	20.658
AB	15	10^3	112.489	119.418	2577.31	315K	4.409
AB	15	10^4	1704.82	1201.37	2398.06	313K	8.512
AB	15	10^5	19605.3	12002.8	2261.46	308K	66.684
AB	15	10^6	237432	136062	3324.36	308K	1432.21
ATDA	15	10^2	22.841	81.828	14547.7	436K	24.22
ATDA	15	10^3	280.794	227.213	21200.9	470K	4.831
ATDA	15	10^4	3535.24	1767.95	24607.4	485K	12.227
ATDA	15	10^5	38018.8	18244.4	32191.2	527K	132.73
ATDA	15	10^6	432281	197296	39394.1	560K	2661.51
ATA	15	10^2	21.237	42.591	10856.7	416K	21.62
ATA	15	10^3	234.147	127.418	14032	442K	4.692
ATA	15	10^4	3100.96	1005.24	16008	457K	9.564
ATA	15	10^5	34018.2	12873.5	20364.6	491K	113.682
ATA	15	10^6	363015	134313	23440.8	512K	1532.9
Name	**NNZ**	**Rows**	**SE (ms)**	**CG (ms)**	**CC (ms)**	**Size**	**MKL (ms)**
COMP1	5	10^3	132.213	215.542	29867.5	766K	0.812
COMP1	5	10^4	1492.24	1235.22	11230.6	463K	3.915
COMP1	5	10^5	16618.4	13216.1	7226.39	398K	50.539
COMP2	5	10^3	139.362	103.94	5074.85	373K	24.012
COMP2	5	10^4	1959.66	1061.75	3159.91	328K	15.275
COMP2	5	10^5	21366.6	11054.4	2828.13	321K	47.928
COMP3	5	10^3	199.822	1314.45	89446.5	1.8M	20.944
COMP3	5	10^4	2466.83	2009.59	27896.1	736K	5.42
COMP3	5	10^5	26054.3	21829	13611.9	321K	47.992
COMP1	15	10^3	712.933	97926	613082	11M	2.023
COMP1	15	10^4	10319.9	49890.3	74354.1	1.7M	11.956
COMP1	15	10^5	124195	246783	32619.3	820K	210.089
COMP2	15	10^3	2928.32	767785	1208650	15M	28.635
COMP2	15	10^4	48309	66660.7	17318.4	587K	50.592
COMP2	15	10^5	698976	615200	8086.12	380K	478.521
COMP3	15	10^3	1243.17	3128920	10186400	59M	24.389
COMP3	15	10^4	18157.8	295109	347103	6.0M	11.842
COMP3	15	10^5	211191	854873	103518	2.0M	153.984
Name	**NNZ**	**Rows**	**SE (ms)**	**CG (ms)**	**CC (ms)**	**Size**	**MKL (ms)**
SLIM1 M1	13	51810	9667.16	4898.7	25293.6	449K	53.139
SLIM1 M2	13	827778	155741	74344	25304.7	449K	1076.34
SLIM2	13	10402	1725.43	706.084	7436.72	354K	11.287
SLIM3	41	27783	63873	41503.9	44648.2	560K	194.368
SMOOTH1	20	10160	718.669	271.362	6893.55	381K	19.341
SMOOTH2	20	43243	3369.61	1208.18	6309.26	376K	35.101
FLOW1	1.5	80000	NA	1017.848	7688.23	667K	NA
FLOW2	1.5	614400	NA	8122.62	6844.6	667K	NA
COTMAT1	7	10160	NA	54826.6	309960	1.9M	NA
COTMAT2	6	43243	NA	96318.8	240015	1.5M	NA

Table 2: *Additional statistics. From left to right: name of the experiment, average number of non zeros per row (NNZ), number of rows, symbolic execution time (SE), code generation time (CG), compilation time (CC), size of the generated binary, MKL preparation time.*