DOI: 10.1111/cgf.13957

COMPUTER GRAPHICS *forum*
Volume 00 (2020), number 00 pp. 1–11

Analysis of Schedule and Layout Tuning for Sparse Matrices With Compound Entries on GPUs

J. S. Mueller-Roemer,[1] A. Stork[1] and D. Fellner[1,2]

[1]TU Darmstadt & Fraunhofer IGD, Darmstadt, Germany
{johannes.mueller-roemer, andre.stork, dieter.fellner}@igd.fraunhofer.de
[2]TU Graz, Graz, Steiermark, Austria,

Abstract

Large sparse matrices with compound entries, i.e. complex and quaternionic matrices as well as matrices with dense blocks, are a core component of many algorithms in geometry processing, physically based animation and other areas of computer graphics. We generalize several matrix layouts and apply joint schedule and layout autotuning to improve the performance of the sparse matrix-vector product on massively parallel graphics processing units. Compared to schedule tuning without layout tuning, we achieve speedups of up to 5.5×*. In comparison to cuSPARSE, we achieve speedups of up to* 4.7×*.*

Keywords: GPGPU, parallel computing, sparse matrix, SpMV

ACM CCS: Computing methodologies: Massively parallel algorithms; Parallel programming languages; Mathematics of computing: Computations on matrices

1. Introduction

While the use of sparse matrices with complex coefficients is common in computational physics, due to their ability to represent amplitude and phase in frequency-domain simulations, and therefore widely supported by linear algebra libraries, other extended number systems and compound entries are used in many areas of computer graphics. For example, quaternions have a long history of use in computer graphics due to their usefulness in representing and interpolating orientations in the special orthogonal group **SO**(3) [Sho85]. More recently, their dual extension, the dual quaternion algebra, has seen increasing use for interpolation and averaging of rigid transformations in the special Euclidean group **SE**(3) as well [KCŽO07].

As a result, sparse matrices with quaternionic entries have found uses in diverse fields used in computer graphics such as the simulation of rigid multi-body systems [Tas01] or geometry processing [CKPS18]. While workarounds using standard, real-valued matrices are available, these are inefficient in both performance and memory use (see Section 3.2).

In addition to extended number systems, the system matrices resulting from the finite-element method (FEM) and other discretizations used in physically based animation of deformables exhibit dense 3×3-blocks. These matrices can be viewed as sparse matrices with 3×3-matrices as entries that are used with vectors of three-dimensional vectors. We use the term *compound entries* as a generalization for both scenarios.

To make efficient use of manycore graphics processing units (GPUs), both memory layout and parallel schedule have to be chosen well. For example, interleaving the components of a compound entry, while natural due to how aggregate types are defined in most programming languages, leads to suboptimal performance on GPUs due to lack of coalescing (see Figure 5 and Section 3.1). Depending on the hardware as well as the domain- and discretization-dependent distribution of non-zero entries, different parallel schedules, e.g. dynamic or static scheduling, and different block sizes are necessary to achieve good performance.

This is an extended version of [MSF19]. The original paper examined how the concept of layout optimization used in dense array autotuners such as MATOG [WG17] can be applied to sparse matrices with compound entries on the GPU. Furthermore, it demonstrated tuning of and generalization over sparse matrix formats such as CSR, ELLPACK-R [VOFG10] and Sliced ELLPACK [MLA10]. In addition, our autotuner performs schedule optimization to deal with matrices with varying non-zero patterns and GPUs with a varying number of cores. In this work, we analyse and discuss the

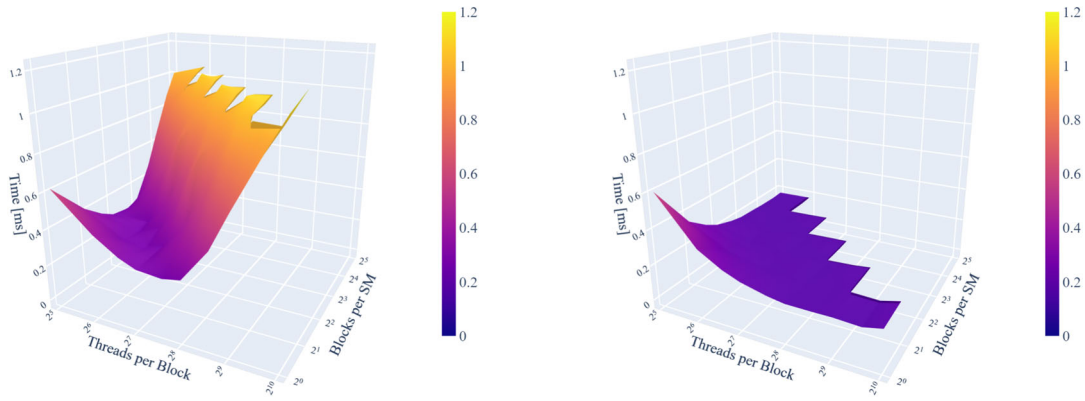

Figure 1: *Schedule tuning comparison for the default CSR–AoS–AoS layout (left) and the best performing ELL–SoA–AoS layout (right) for the double precision fem/NX_MotorH67k matrix on the GP100. The choice of layout significantly affects which schedule is optimal. In both cases, the base schedule type is static. The plot surface is coloured by time value.*

memory overheads of each format and the effects of schedule optimization in detail.

In the following sections, we provide an overview of related work including sparse matrix formats and code generation, just-in-time (JIT) compilation and autotuning for GPUs and use cases for sparse matrices with compound entries in Section 2. In Section 3, we provide a brief overview of multi- and manycore architectures and programming models, memory layouts and workarounds for the lack of quaternionic matrix support in current linear algebra libraries. We detail our approach in Section 4, followed by discussing the results of our evaluation in Section 5. Finally, we summarize the paper and suggest avenues for future research in Section 6.

2. Related Work

In this section, we list use cases and outline related work on formats for sparse matrices with compound entries. Furthermore, an overview over related JIT compilation, code generation and autotuning approaches is given.

2.1. Sparse matrices with compound entries

Sparse matrices with complex coefficients are common in frequency-domain simulations such as acoustic [Tho06] and electromagnetic simulations [Jin14]. In the latter case, system matrices have dense 3×3-blocks of complex entries. As complex matrices are a common use case, commercial sparse linear algebra libraries, e.g. NVIDIA cuSPARSE [NVI19b], provide well-tuned algorithms for such matrices.

In the field of geometry processing, Crane *et al.* use sparse quaternionic matrices to compute conformal transformations of triangle meshes in $\mathbb{R}^3$ [CPS11]. Later publications based on the quaternionic Dirac operator defined by Crane *et al.* result in quaternionic matrices as well [CPS13, LJC17, YDT*18]. Chern *et al.* use parallel transport of unit quaternions representing triangle orientations for the isometric immersion problem of orientable triangle meshes

AoS	r_0	i_0	r_1	i_1	r_2	i_2	r_3	i_3	…
SoA	r_0	r_1	r_2	…		i_0	i_1	i_2	…
AoSoA	r_0	r_1	i_0	i_1	r_2	r_3	i_2	i_3	…

Figure 2: *Array of structures (AoS), structure of arrays (SoA) and array of structures of arrays (AoSoA) layouts of an array of complex numbers $c_k = r_k + i_k\mathbf{i}$. For AoSoA, an inner array size of 2 is shown.*

[CKPS18]. Both Crane and Chern *et al.* suggest using the 4×4-matrix expansion of quaternions, which leads to a significant memory and compute overhead (see Section 3.2).

In physically based animation, the FEM approach for simulating deformable models results in dense 3×3-matrix blocks. Libraries such as cuSPARSE support the block compressed sparse row (BSR) format, a variant of the compressed sparse row (CSR) format [Saa03] for matrices with dense, fixed-size blocks that omits implicitly computable column indices. In academia, some researchers have used this fact to design GPU-optimized sparse matrix formats for FEM simulation. Examples include Weber *et al.*'s binned block compressed sparse row (Bin-BCSR) format [WBS*13], which only uses the block structure along one dimension and was recently improved to use it along both dimensions by Mueller-Roemer and Stork [MS18]. For the simulation of flexible cables in interactive and virtual reality applications, Lang *et al.* introduce a quaternionic discretization of the rotational degrees of freedom of Cosserat rods [LLA11]. A related approach was recently used for geometrically exact simulation of beams under large displacements and contact by Tasora *et al.* [TBMG20]. Furthermore, quaternionic matrices can be used to improve the performance of rigid multi-body system simulations, as shown by Tasora [Tas01]. Despite their use in computer graphics and physics, sparse quaternionic matrices are, to the best of our knowledge, not supported by any major (GPU-accelerated) linear algebra library.

2.2. Schedule and layout autotuning

In the domain of layout tuning of dense arrays for GPUs, Weber and Goesele have combined text template-based layout variations combined with model-based autotuning to great effect in MATOG [WG17]. Here, layout tuning refers to the selection of array of structures (AoS), structure of arrays (SoA) or array of structures of arrays (AoSoA) layouts (see Figure 2) and row-major or column-major orders for dense n-dimensional arrays with compound entries.

In the area of compiler technologies for sparse matrices, Bik introduced a compiler that automatically transforms dense codes into sparse codes as well as performing CPU vectorization and parallelization [Bik96]. He also introduces more advanced transforms that require the sparsity pattern at compile time. Cheshmi *et al.* also perform compile time analysis but combine it with a polyhedral loop optimizer to generate specialized direct solvers with improved vectorization on CPUs [CKSD17]. Venkat *et al.* use a runtime inspector and executer to analyse the sparsity pattern to perform reordering for GPU sparse matrix operations [VHS15]. Kjolstad *et al.* introduce the tensor algebra compiler TACO which allows the user to select various layouts for each tensor in a tensor expression [KKC*17]. For two-dimensional tensors, the possible layouts correspond to dense matrices in row- or column-major layout, sparse CSR and compressed sparse column (CSC) formats or their hypersparse (low-rank) extensions. However, the generated code is serial. In all cases, extended number systems are not supported.

Monakov *et al.* introduce the sliced ELLPACK format and perform tuning of slice and thread block size [MLA10]. They reorder matrix rows to achieve more compact storage. To efficiently use reordered matrices, permutation has to be performed rarely, e.g. before and after an iterative solver. Even though they do not consider compound entries and compare cards of the same generation, some cases are sped up by up to 10% when performing hardware-specific tuning.

3. Background

In this section, we provide a brief background on the performance properties of GPUs and describe the workarounds used with existing linear algebra libraries when using quaternionic matrices.

3.1. GPU performance and terminology

Manycore GPU performance characteristics differ in many aspects from those of multi-core CPUs. While this description focuses on NVIDIA GPUs and terminology, similar considerations apply to other manycore processors. CUDA follows a hierarchical single program multiple data (SPMD) programming model. Kernels are launched in a grid of blocks consisting of a number of threads. Blocks are scheduled on streaming multi-processors (SMs), analogous to CPU cores. Threads are grouped into warps of 32 threads that do not execute independently. Divergent code paths are implicitly serialized, as done explicitly when using vector instructions on a CPU. For good performance, memory accesses must follow specific patterns to achieve coalescing. Contiguous threads within a warp must access contiguous locations in memory, where alignment and size of the access must match and be a power of two between 1 and 16 bytes. As in the case of divergent control flow, this is done implicitly, but has similarities with vector loads which have alignment requirements too. Finally, multiple blocks are executed at once on an SM using hardware multi-threading to perform latency hiding. Therefore, it is important to consider occupancy, i.e. blocks should ideally not use too many resources such as registers so that more than one block can be scheduled at once. For additional details, refer to the NVIDIA CUDA programming guide [NVI19a].

3.2. Alternative quaternion representations

Quaternions, like complex numbers, have equivalent, non-unique, real matrix representations (see, e.g. [HL90]) such as:

$$q = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \equiv \begin{pmatrix} w & -x & -y & -z \\ x & w & -z & y \\ y & z & w & -x \\ z & -y & x & w \end{pmatrix}, \tag{1}$$

where $q \in \mathbb{H}$, $w, x, y, z \in \mathbb{R}$, and $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$ are the fundamental quaternion units. As a result, any quaternionic matrix $\mathbf{A} \in \mathbb{H}^{n\times m}$ can equivalently be represented as a real matrix $\mathbf{A}' \in \mathbb{R}^{4n\times 4m}$. However, this equivalence leads to a $4\times$ memory and compute overhead, but it allows the reuse of existing direct and iterative solvers.

Another approach is to decompose the matrices and vectors according to the Hamilton product:

$$\begin{aligned} \mathbf{A} &= \mathbf{W} + \mathbf{X}\mathbf{i} + \mathbf{Y}\mathbf{j} + \mathbf{Z}\mathbf{k} \\ \mathbf{q} &= \mathbf{w} + \mathbf{x}\mathbf{i} + \mathbf{y}\mathbf{j} + \mathbf{z}\mathbf{k} \\ \mathbf{A}\mathbf{q} &= (\mathbf{W}\mathbf{w} - \mathbf{X}\mathbf{x} - \mathbf{Y}\mathbf{y} - \mathbf{Z}\mathbf{z}) + (\mathbf{W}\mathbf{x} + \mathbf{X}\mathbf{w} + \mathbf{Y}\mathbf{z} - \mathbf{Z}\mathbf{y})\,\mathbf{i} \\ &\quad + (\mathbf{W}\mathbf{y} - \mathbf{X}\mathbf{z} + \mathbf{Y}\mathbf{w} + \mathbf{Z}\mathbf{x})\,\mathbf{j} + (\mathbf{W}\mathbf{z} + \mathbf{X}\mathbf{y} - \mathbf{Y}\mathbf{x} + \mathbf{Z}\mathbf{w})\,\mathbf{k}, \end{aligned} \tag{2}$$

where $\mathbf{A} \in \mathbb{H}^{n\times m}$, $\mathbf{q} \in \mathbb{H}^m$, $\mathbf{W}, \mathbf{X}, \mathbf{Y}, \mathbf{Z} \in \mathbb{R}^{n\times m}$ and $\mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}^m$. While this approach avoids the higher memory and compute overhead, the serial chaining of multiple matrix-vector products leads to increased latency, synchronization and kernel launch overheads. For small- to medium-sized matrices typically used in computer graphics, kernel launch overheads can make up a significant portion of execution time. Furthermore, most existing solvers cannot be used with this decomposition.

4. Approach

In this section, we describe how we apply layout variations to sparse matrices with compound entries, the resulting code generator and the autotuning approach.

4.1. Sparse matrix formats and layouts

Much like layouts such as AoS, SoA and AoSoA as well as row- or column-major orderings can be applied to dense n-dimensional arrays without changing the semantics of the array (see Figure 2 and Section 2.2), we differentiate between semantically different sparse matrix data structures and those that only differ in their

in-memory layout. For example, the CSR, ELLPACK(-R) and Sliced ELLPACK formats are all semantically arrays of length n of variable length arrays of tuples of column index and entry value for a matrix with n rows. Semantically different data structures such as hierarchical [DZSS17] or bitmap-based [ZG18] sparse matrix data structures are not considered layout variants.

CSR stores the column index and value tuples in a pair of contiguous arrays. Essentially, the tuples are stored in a 1D array in SoA layout. To mark the starting and ending positions of the per-row variable length arrays, CSR includes an array of $n + 1$ offsets into the contiguous arrays.

In the original ELLPACK format [RB85], the per-row arrays are padded with explicit zeros such that they all have the same length. The resulting dense 2D arrays are stored in column-major order. With respect to CSR, which is in row-major order by definition, the data are therefore padded and transposed. As the resulting arrays are dense, the offset array can be omitted. The ELLPACK-R format [VOFG10] replaces it with a non-zero count array of length n instead, which makes it possible to avoid performing any computations on added padding. If the stride between rows is additionally padded to a multiple of the warp size, the column-major ordering of these layouts leads to good coalescing on GPUs.

As ELLPACK(-R) can lead to a very large memory overhead when a small number of rows has a much larger number of non-zero entries than the others, Sliced ELLPACK [MLA10] first partitions the matrix into slices of k rows before padding and transposing the individual slices. As offsets within slices can be computed implicitly, only $\lceil n/k \rceil + 1$ offsets are required. For coalescing, k should be 16 (half-warp-sized) or 32 (warp-sized). The number of stored rows is padded to a multiple of k in the same way as the AoSoA layout requires padding the length of a dense array to a multiple of the inner array size.

We call the choice between row-major (CSR), padded column-major (ELLPACK-R) and sliced padded column-major (Sliced ELLPACK) orderings the *outer layout* of the sparse matrix. When compound entries are used, the dense entry array and the vector can be stored in AoS or SoA layouts. These choices define the *inner* and *vector layouts*.

While the CSC format of matrix $\mathbf{A}$ is identical to storing the transpose $\mathbf{A}^T$ in CSR format, we did not implement such transposed input layouts. As summation of each row and therefore entry of the output vector cannot be performed independently, supporting these requires different parallel algorithms. While Steinberger *et al.* have shown that the naïve approach of using atomic summation only leads to limited slowdown [SDZS16], doing so leads to the loss of determinism.

4.2. Code generator

To generate the code for the layout variants, we use a text templating approach based on Jinja2. The generated code is compiled with the system C++ and CUDA compilers and linker. We focus on the sparse matrix-vector product (SpMV) as it is the most costly component of iterative solvers such as the conjugate gradient algorithm.

To generate the code for a particular layout and schedule, the following inputs are required:

- A compound entry definition (a list of identifiers with associated types). A separate definition can be given for vector entries.
- Multiplicative and additive operator definitions, as well as zero initializer(s).
- The outer (including slice size if a sliced layout is chosen) and inner layouts of the matrix, as well as the layout of the vector.
- The schedule type (static or dynamic), as well as the numbers of SMs n_s, blocks per SM n_b and threads per block n_t.

The generated kernels use constant size blocks and grids, independent of the matrix size. Depending on schedule type, each block processes either chunks of n_t rows with a static stride of $n_s \cdot n_b \cdot n_t$ or selects chunks dynamically using an atomic counter. The block size n_t and the number of blocks per SM n_b are passed to the CUDA compiler using the `__launch_bounds__` annotation. This allows the compiler to generate code with the appropriate number of registers per thread.

As an additional performance optimization, the pointers to the vector array(s) for the right-hand side are annotated with the `__restrict__` keyword. This allows the optimizer to use non-coherent loads which typically perform better for random access. Furthermore, AoS entries are annotated with the largest power of two alignment between 1 and 16 of which their size is a multiple. This enables the use of vectorized loads where possible.

Besides the SpMV kernel, the code generator outputs header files for the generated matrix and vector classes. In addition to an interface callable from standard C++ code, the classes provide constructors to convert from CSR matrices in default AoS layout on the host to the chosen layout on the GPU. The source code of the generator is available for non-commercial use under https://github.com/fh-igd-iet/FhSparseGen.

4.3. Autotuner

Given a set of matrices as well as the entry and operator definitions, the autotuner jointly optimizes layout and schedule for the given matrices. To do so, it first determines the compute capability (CC) and the number of SMs n_s of the GPU. The CC determines the warp size w (32 for all currently available NVIDIA GPUs), maximum numbers of blocks per SM and threads per block supported by the GPU, as well as other indirectly relevant factors such as supported instruction set and number of registers per SM.

This information determines the bounds for the tuning parameters n_b and n_t. To limit the size of the resulting Cartesian product of variants, n_b is chosen from all 2^i and $3 \cdot 2^i$ with $i \geq 0$ that are within the bounds. Similarly, n_t is chosen from all $w \cdot 2^i$ and $w \cdot 3 \cdot 2^i$ that are within the bounds. The slice size k is limited to half-warp and warp sizes. The scheduling parameters (static/dynamic, n_b and n_t) can either be tuned separately, or jointly with the layout parameters (outer, inner and vector layouts).

The generated variants are built with the CUDA compiler, passing the CC as a parameter to generate code for the specific architecture.

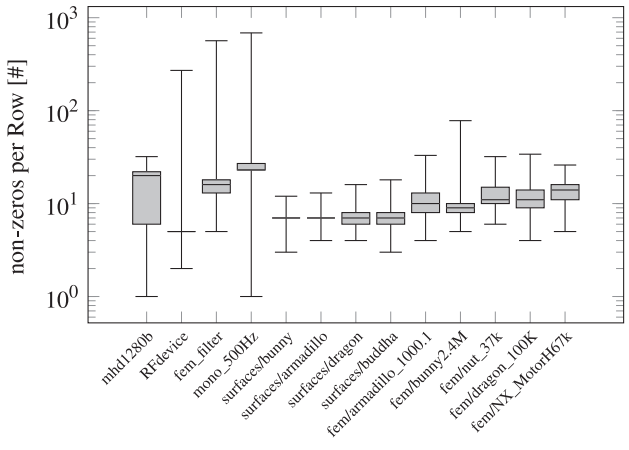

Figure 3: *Distribution of non-zero entries per row for each matrix used in the evaluation. For matrices beginning with 'fem/', this is the number of non-zero* 3×3*-blocks per group of three rows.*

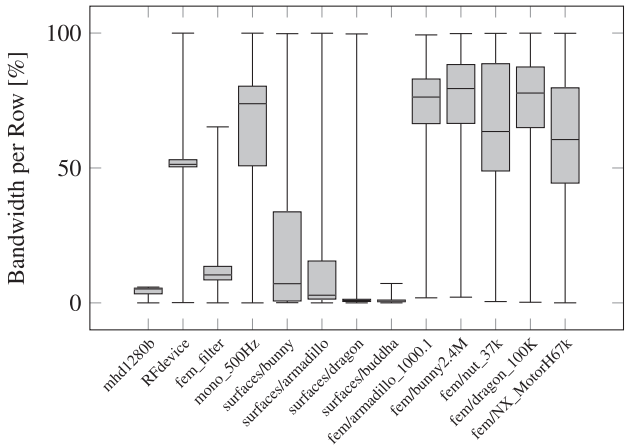

Figure 4: *Distribution of bandwidths per row for each matrix used in the evaluation. Given in percent normalized by number of rows/columns in each matrix. For matrices beginning with 'fem/', block-row and -column indices are used.*

These are linked to a benchmarking fixture that calls and measures the runtime of the SpMV a given number of times for each matrix.

5. Results

In this section, we describe the setup of the benchmarks performed and evaluate their results. Figures 3 and 4 show statistical information about numbers of non-zero entries and bandwidths of the rows of the matrices used in the evaluation. The extent of the boxes ranges from the lower to the upper quartile, with a line at the median. The whiskers extend from the minimum to the maximum. The bandwidth of a row is defined as

$$b_i = \max_{\{j|\mathbf{A}_{ij}\neq 0\}} j - \min_{\{j|\mathbf{A}_{ij}\neq 0\}} j \tag{3}$$

Table 1: *Sizes of the matrices used in rows/columns and number of compound non-zeros.*

Matrix	Rows/Columns	Non-zeros
mhd1280b	1280	22 778
RFdevice	74 104	365 580
fem_filter	74 062	1 731 206
mono_500Hz	169 410	5 033 796
surfaces/bunny	34 834	243 410
surfaces/armadillo	172 974	1 210 806
surfaces/dragon	435 545	3 049 463
surfaces/buddha	543 524	3 805 882
fem/armadillo_1000.1	1016	11 180
fem/bunny2.4M	27 112	284 846
fem/nut_37k	9095	112 685
fem/dragon_100K	26 436	315 006
fem/NX_MotorH67k	66 840	899 478

and provides information about the locality of accesses. In addition, the sizes of the matrices are given in Table 1.

The matrices 'mhd1280b', 'RFdevice', 'fem_filter' and 'mono_500Hz' are complex matrices from the SuiteSparse Matrix Collection [DH11] chosen to cover a large range of sizes and non-zero entry distribution patterns. The matrices beginning with 'surface/' are quaternionic matrices that were generated from meshes available in the Stanford 3D Scanning Repository [Sta14] using Crane *et al.*'s algorithm [CPS11]. The matrices beginning with 'fem/' are matrices with 3×3-block entries resulting from a linear FEM discretization on tetrahedral meshes. The tetrahedral meshes were generated with Gmsh [GR09] (armadillo_1000.1), TetGen [Si15] (bunny2.4M, dragon_100K), CGAL [CGA18] (nut_37k) and Siemens NX [Sie18] (NX_MotorH67k). All matrices used in the evaluation are square.

The evaluations were performed on three machines with GPUs from various generations or CCs, including both professional and consumer (restricted double precision performance) GPUs, with the following hardware:

1. NVIDIA Quadro K2000 (CC 3.0, 2 SMs, 2 GiB GDDR5), Intel i5-4670 (4 cores, 3.4 GHz), 16 GiB DDR3-1600
2. NVIDIA GeForce GTX 980 (CC 5.2, 16 SMs, 4 GiB GDDR5), Intel i7-4790K (4 cores, 4.0 GHz), 16 GiB DDR3-1600
3. NVIDIA Quadro GP100 (CC 6.0, 56 SMs, 16 GiB HBM2), Intel i7-6700K (4 cores, 4.0 GHz), 32 GiB DDR4-2133

All systems were running Windows 10 and benchmarks were compiled with Visual Studio 2015 and CUDA 9.2.

To determine the best layout-schedule combination, the generated SpMV was called 1000 times per matrix for each combination. CUDA kernels were timed using CUDA events, to avoid primarily measuring the CPU–GPU synchronization overhead on small matrices. The full set of measurements (minimum, lower quartile, median, upper quartile, maximum and average times per matrix, layout and schedule) is provided as supplemental material.

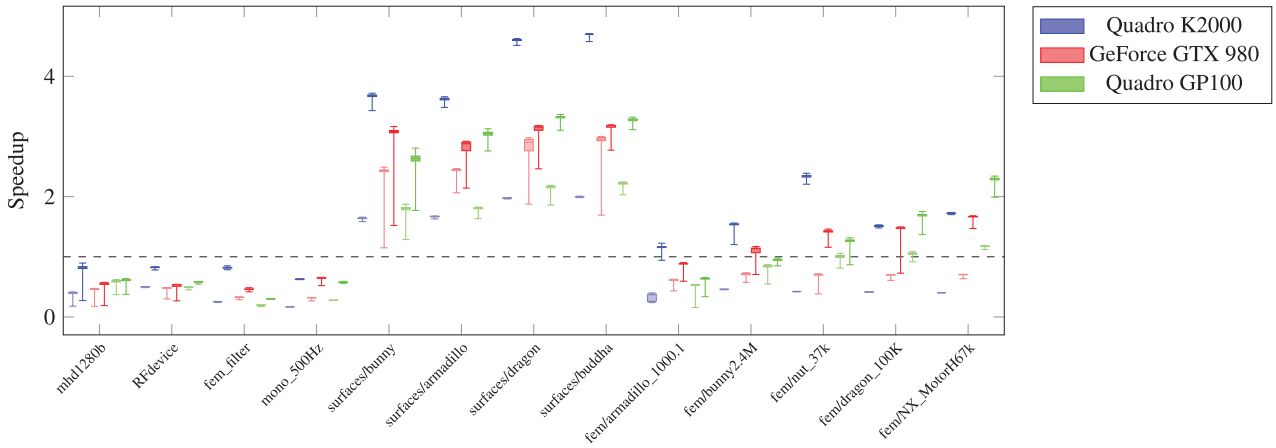

Figure 5: *Speedup relative to cuSPARSE in single precision with (dark) and without (light) layout optimization. For large compound entries ('fem/*') and extended number systems ('surface/*'), speedups of up to 4.7× are achieved.*

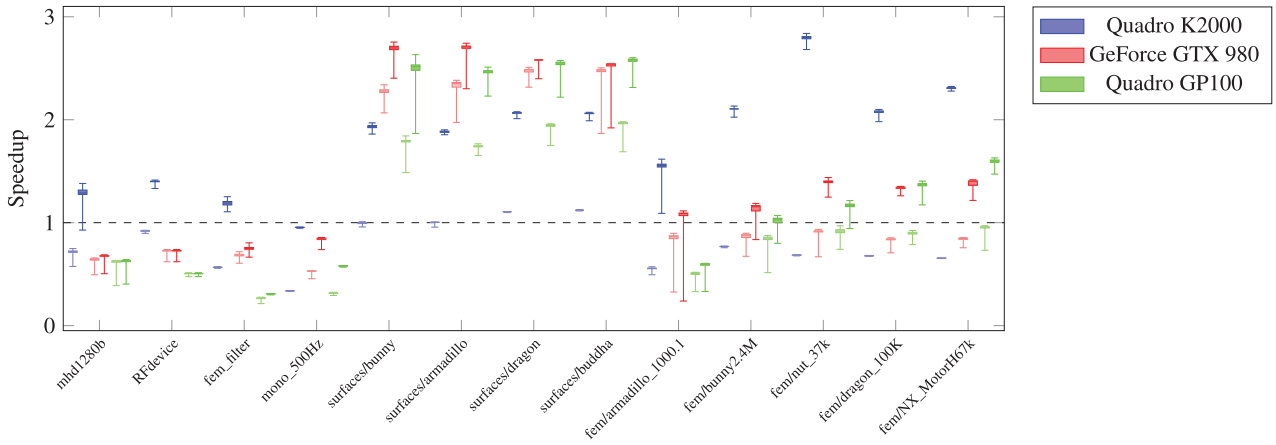

Figure 6: *Speedup relative to cuSPARSE in double precision with (dark) and without (light) layout optimization. For large compound entries ('fem/*') and extended number systems ('surface/*'), speedups of up to 2.8× are achieved.*

All matrix-vector multiplications were also performed using cuSPARSE, NVIDIA's own highly tuned sparse linear algebra library. For the complex matrices, the built-in support for complex linear algebra was used. For the block-sparse matrices, the built-in support for the BSR format was used. For the quaternionic matrices, we used the matrix expansion (see Section 3.2) on the matrix only. $\mathbf{x} \in \mathbb{H}^n$ was represented as $\mathbf{x}' \in \mathbb{R}^{4n}$. As the resulting matrices are block-sparse too, BSR was used in this case as well. Therefore, only the number of values, not the numbers of offsets and column indices, of the matrix are quadrupled. The measured speedups are given in Figures 5 and 6 for single and double precision, respectively.

5.1. Complex matrices

The best layouts per GPU for each complex precision matrix as well as the speedups compared to only performing schedule tuning, i.e. using the 'natural' CSR layout with entries in AoS layout, are given in Tables 2 and 3.

While most cases show and absolute speedup of less than 1× compared to cuSPARSE, speedups of approximately 1–1.5× are achieved for double precision matrices on the Quadro K2000 (see Figures 5 and 6). Furthermore, the largest layout tuning gains are achieved on the K2000 as well. The speedups on the two newer GPUs are similar, despite the significantly lower double precision performance on consumer GPUs. While AoS is preferred for both inner and vector entry layout in most cases, no clear preference in outer layout can be observed. As both single and double precision complex entries can be aligned to 8 and 16 bytes, respectively, the preference of AoS layout is expected.

5.2. Quaternionic matrices

As for complex matrices in the previous section, we list the best layouts and speedups relative to not performing layout tuning for all quaternionic sparse matrices in Tables 4 and 5.

Table 2: *Best layouts and layout tuning speedups for all complex single precision matrices. Layouts are given as* outer-inner-vector*, where ELL is ELLPACK-R and Sl-k is Sliced ELLPACK with a slice size of k.*

Matrix	GPU	Layout	Speedup
mhd1280b	K2000	ELL-AoS-AoS	2.09×
	GTX 980	ELL-AoS-AoS	1.22×
	GP100	CSR-SoA-AoS	1.05×
RFdevice	K2000	ELL-AoS-AoS	1.64×
	GTX 980	Sl-32-SoA-AoS	1.11×
	GP100	CSR-SoA-AoS	1.18×
fem_filter	K2000	ELL-AoS-AoS	3.20×
	GTX 980	ELL-AoS-AoS	1.40×
	GP100	ELL-AoS-AoS	1.52×
mono_500Hz	K2000	Sl-32-SoA-AoS	3.74×
	GTX 980	Sl-32-SoA-AoS	2.02×
	GP100	ELL-AoS-AoS	2.09×

Table 3: *Best layouts and layout tuning speedups for all complex double precision matrices. Layouts are given as in Table 2.*

Matrix	GPU	Layout	Speedup
mhd1280b	K2000	Sl-32-AoS-AoS	1.79×
	GTX 980	Sl-16-AoS-AoS	1.05×
	GP100	CSR-SoA-SoA	1.00×
RFdevice	K2000	ELL-AoS-AoS	1.52×
	GTX 980	CSR-AoS-AoS	1.00×
	GP100	CSR-AoS-AoS	1.00×
fem_filter	K2000	ELL-AoS-AoS	2.11×
	GTX 980	ELL-AoS-AoS	1.10×
	GP100	ELL-AoS-AoS	1.16×
mono_500Hz	K2000	Sl-32-AoS-AoS	2.81×
	GTX 980	Sl-16-AoS-AoS	1.59×
	GP100	ELL-AoS-AoS	1.84×

Table 4: *Best layouts and layout tuning speedups for all quaternionic single precision matrices. Layouts are given as in Table 2.*

Matrix	GPU	Layout	Speedup
bunny	K2000	ELL-AoS-AoS	2.24×
	GTX 980	ELL-AoS-AoS	1.27×
	GP100	ELL-AoS-AoS	1.46×
armadillo	K2000	ELL-AoS-AoS	2.18×
	GTX 980	ELL-AoS-AoS	1.18×
	GP100	ELL-AoS-AoS	1.68×
dragon	K2000	ELL-AoS-AoS	2.33×
	GTX 980	ELL-AoS-AoS	1.09×
	GP100	ELL-AoS-AoS	1.54×
buddha	K2000	ELL-AoS-AoS	2.35×
	GTX 980	ELL-AoS-AoS	1.08×
	GP100	ELL-AoS-AoS	1.48×

Table 5: *Best layouts and layout tuning speedups for all quaternionic double precision matrices. Layouts are given as in Table 2.*

Matrix	GPU	Layout	Speedup
bunny	K2000	ELL-SoA-AoS	1.93×
	GTX 980	ELL-SoA-AoS	1.18×
	GP100	ELL-AoS-AoS	1.40×
armadillo	K2000	ELL-SoA-AoS	1.88×
	GTX 980	ELL-AoS-AoS	1.15×
	GP100	ELL-SoA-AoS	1.42×
dragon	K2000	ELL-SoA-SoA	1.87×
	GTX 980	ELL-SoA-AoS	1.04×
	GP100	ELL-AoS-AoS	1.31×
buddha	K2000	ELL-SoA-SoA	1.84×
	GTX 980	ELL-AoS-AoS	1.02×
	GP100	ELL-AoS-AoS	1.31×

While there was no clear outer layout preference in Section 5.1, the padded transpose (ELLPACK-R) is preferred in all cases. As can be seen in Figure 3, the difference between the longest and shortest rows is much smaller for these matrices, therefore these matrices incur a significantly smaller amount of padding. As expected for the 16-byte aligned single precision quaternion entries, AoS layout is preferred in Table 4. Double precision quaternions are 32 bytes in size. Therefore, AoS layout requires two consecutive 16 byte loads and cannot achieve full coalescing. However, in all but two cases AoS layout continues to be preferred for the vector entries, due to the mostly random access patterns. For the matrix entries, SoA is preferred in many but not all cases for the double precision matrices. As before, the largest speedups due to layout tuning are achieved on the K2000. Unlike in the last section, the speedups on the GP100 are slightly larger than on the GTX 980, potentially due to the higher flops-per-byte ratio.

5.3. 3 × 3-block matrices

As in the previous sections, we list the best layouts and speedups relative to not performing layout tuning for all sparse matrices with 3 × 3-block entries (and vectors of 3D vectors) in Tables 6 and 7.

Both 3 × 3 blocks and 3D vectors cannot be aligned to power-of-two addresses without introducing padding, independent of scalar precision. Combined with the large entry size, the preference of SoA inner layout is expected. For the vector layout, AoS is preferred in most cases except for the fem/armadillo_1000.1 and fem/nut_37k matrices. As in the previous sections, the greatest gains are achieved on the K2000. This is followed by the GTX 980 and the GP100 benefits the least. Except for the smallest matrix, the tuned matrix layouts and schedules are faster than cuSPARSE using BSR as seen in Figures 5 and 6.

5.4. Schedule tuning

In this section, we quantify the effects of schedule tuning and analyse if it is necessary to perform schedule and layout tuning jointly or not, whereas the speedups listed in Sections 5.1 to 5.3 compare the best schedule for each layout with the best schedule for the reference layout.

As described in Sections 4.2 and 4.3, schedule tuning is performed over schedule type, i.e. static or dynamic, the number of threads per block n_t and the number of blocks per SM n_b. Figure 1 compares the median time per SpMV for all n_t and n_b using the

Table 6: *Best layouts and layout tuning speedups for all single precision matrices with* 3×3 *blocks. Layouts are given as in Table 2.*

Matrix	GPU	Layout	Speedup
armadillo_1000.1	K2000	Sl-32-SoA-SoA	3.15×
	GTX 980	Sl-32-SoA-SoA	1.44×
	GP100	Sl-16-SoA-SoA	1.20×
bunny2.4M	K2000	ELL-SoA-AoS	3.34×
	GTX 980	ELL-SoA-AoS	1.58×
	GP100	ELL-AoS-AoS	1.12×
nut_37k	K2000	ELL-SoA-SoA	5.54×
	GTX 980	Sl-32-SoA-SoA	2.03×
	GP100	Sl-32-SoA-SoA	1.27×
dragon_100K	K2000	ELL-SoA-AoS	3.67×
	GTX 980	ELL-SoA-AoS	2.12×
	GP100	ELL-AoS-AoS	1.61×
NX_MotorH67k	K2000	ELL-SoA-AoS	4.28×
	GTX 980	ELL-SoA-AoS	2.37×
	GP100	ELL-SoA-AoS	1.95×

Table 7: *Best layouts and layout tuning speedups for all double precision matrices with* 3×3 *blocks. Layouts are given as in Table 2.*

Matrix	GPU	Layout	Speedup
armadillo_1000.1	K2000	ELL-SoA-SoA	2.80×
	GTX 980	Sl-32-SoA-SoA	1.26×
	GP100	Sl-32-SoA-SoA	1.18×
bunny2.4M	K2000	ELL-SoA-AoS	2.74×
	GTX 980	ELL-SoA-AoS	1.31×
	GP100	ELL-AoS-SoA	1.21×
nut_37k	K2000	ELL-SoA-SoA	4.08×
	GTX 980	ELL-SoA-SoA	1.53×
	GP100	ELL-AoS-AoS	1.27×
dragon_100K	K2000	ELL-SoA-AoS	3.05×
	GTX 980	ELL-SoA-AoS	1.59×
	GP100	ELL-AoS-AoS	1.52×
NX_MotorH67k	K2000	ELL-SoA-AoS	3.52×
	GTX 980	ELL-SoA-AoS	1.66×
	GP100	ELL-SoA-AoS	1.67×

double precision fem/NX_MotorH67k matrix and static schedules on machine 3. No values are available for large numbers of threads per SM $n_t \cdot n_b$, as these would exceed the capabilities of the GPU. In both cases, the number of registers required varies between 32 and 40 registers, due to the `__launch_bounds__` annotation causing the CUDA compiler and PTX assembler to choose different optimizations. No register spilling to local memory occurs in either case.

Despite the similarities in register use and therefore occupancy, peak performance is achieved at a much lower number of threads per SM for the CSR–AoS–AoS layout than for the ELL–SoA–AoS layout. Therefore, schedule and layout must be tuned jointly for best performance. However, it may be possible to reduce the search space by one dimension with little loss of performance by tuning $n_t \cdot n_b$ instead of n_t and n_b individually, as the schedule tuning graphs are close to constant along diagonals of constant thread number per SM.

Table 8: *Cases in which dynamic schedules are preferred for either single precision (SP), double precision (DP), both (SP/DP) or neither (—) when using the best performing layout.*

Matrix	K2000	GTX980	GP100
mhd1280b	—	—	—
RFdevice	—	—	—
fem_filter	SP/DP	SP/DP	—
mono_500Hz	SP/DP	—	—
surfaces/bunny	—	—	—
surfaces/armadillo	SP	SP	—
surfaces/dragon	SP	SP	DP
surfaces/buddha	SP	SP	DP
fem/armadillo_1000.1	—	—	—
fem/bunny2.4M	SP/DP	SP/DP	DP
fem/nut_37k	—	—	—
fem/dragon_100K	SP/DP	SP	DP
fem/NX_MotorH67k	DP	—	—

Figure 7 compares the static and dynamic schedule types for all n_t and n_b using the double precision fem/NX_MotorH67k matrix and the ELL-AoS-SoA layout on machine 3. While similar in shape, we observed that the dynamic schedules are more sensitive to the effects of schedule tuning than the static schedules, i.e. the execution time ratios between the worst and best schedules for a given layout are larger. Dynamic schedules perform better in roughly 30% of all evaluated cases, as shown in Table 8.

5.5. Memory overhead

The padding required for the padded column-major (ELLPACK/ELL) and sliced padded column-major (Sliced ELLPACK/Sl-k) outer layouts leads to a potentially significant memory overhead, as does the matrix expansion for quaternionic matrices when

Table 9: *Memory use of each memory layout for each input matrix. The baseline is CSR/BSR with the appropriate entry type and given in bytes (computed using 4-byte indices and 8-byte values). All others are given as a multiple of the baseline. The value for cuSparse (cuSp.) is based on quaternion to matrix expansion and 4×4 BSR for quaternionic matrices and otherwise identical to the baseline.*

Matrix	CSR/BSR	ELL	Sl-16	Sl-32	cuSp.
mhd1280b	449.9 KiB	1.8×	1.5×	1.7×	1.0×
RFdevice	7.3 MiB	52.8×	1.0×	1.0×	1.0×
fem_filter	33.3 MiB	24.0×	1.8×	2.1×	1.0×
mono_500Hz	96.7 MiB	22.9×	1.2×	1.3×	1.0×
surfaces/bunny	8.5 MiB	1.7×	1.1×	1.2×	3.6×
surfaces/armadillo	42.2 MiB	1.8×	1.2×	1.2×	3.6×
surfaces/dragon	106.4 MiB	2.3×	1.3×	1.4×	3.6×
surfaces/buddha	132.7 MiB	2.5×	1.3×	1.4×	3.6×
fem/armadillo_1000.1	833.7 KiB	3.0×	1.5×	1.6×	1.0×
fem/bunny2.4M	20.7 MiB	7.4×	2.0×	2.3×	1.0×
fem/nut_37k	8.2 MiB	2.6×	1.3×	1.4×	1.0×
fem/dragon_100K	22.9 MiB	2.8×	1.5×	1.6×	1.0×
fem/NX_MotorH67k	65.4 MiB	1.9×	1.2×	1.3×	1.0×

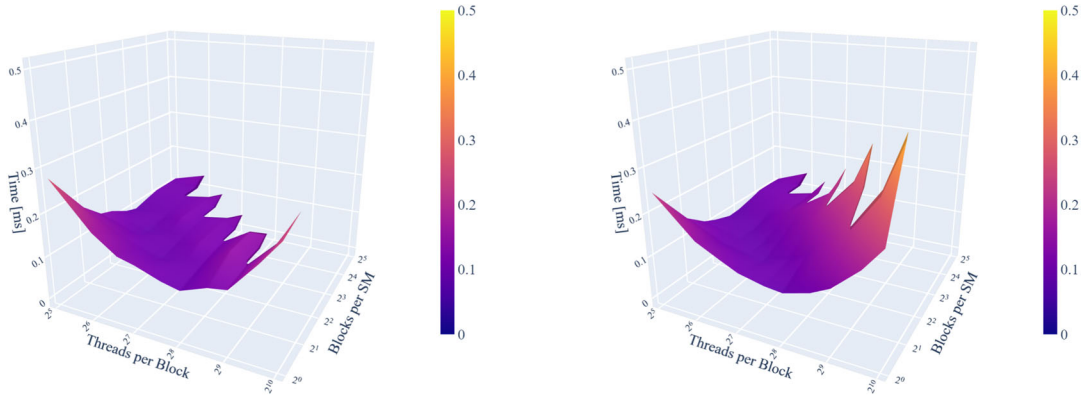

Figure 7: *Schedule tuning comparison between the static (left) and dynamic (right) base schedule types. In both cases, the double precision fem/NX_MotorH67k matrix with ELL-AoS-SoA layout is shown and optimized for the GP100. The plot surface is coloured by time value.*

quaternions are not directly supported. The amount of memory required is affected by the outer layout and sparsity pattern of each matrix, but not by the inner layout. The memory requirements for all matrices and outer layouts evaluated in this paper are given in Table 9 for 4-byte indices (offsets, counts and column indices) and 8-byte double precision values. While the CSR/BSR layouts are the reference, they are not guaranteed to be optimal. For example, the factor 1.0× for RFdevice using Sl-16 and Sl-32 are rounded from 0.98× and 0.99×, respectively. While the 4×4-block expansion of the quaternionic matrices results in a 4× expansion of the values, the index and offset arrays are not expanded due to the use of BSR, resulting in a 3.6× expansion for cuSparse.

The overhead of the ELL layout is particularly significant for the RFdevice, fem_filter and mono_500Hz matrices (and to a lesser extent fem/bunny2.4M). These matrices have a single row or small number of rows with a significantly larger number of non-zero entries than the average. The row with the largest number of non-zeros directly determines the amount of memory required for the ELL layout. However, even in the case of RFdevice a dense matrix would require more than 200× more memory than the ELL layout.

While more compact layouts are preferred with respect to performance for RFdevice on the GTX 980 and GP100 GPUs and mono_500Hz for the K2000 and GTX 980 (see Tables 2 and 3), this is not the case for the fem_filter or fem/bunny2.4M matrices or on all GPUs (see also Tables 6 and 7). At the same time, more compact layouts are preferred for mhd1280b, fem/armadillo_1000.1 and fem/nut_37k even though these matrices do not suffer from excessive memory overheads with the ELL layout. Despite potentially higher cache pressure, even large memory overheads therefore only slightly influence execution time.

6. Conclusion

We have shown that significant speedups can be achieved by performing joint schedule and layout autotuning for sparse matrices with compound entries. Compared to only performing schedule tuning, speedups of up to 5.5× are achieved (see Table 6). Compared to the highly tuned vendor library cuSPARSE, we achieve speedups of up to 4.7× for the SpMV (see Figure 5). Even for matrices with dense blocks, which are supported directly in cuSPARSE, we achieve speedups of up to 2.8× using our approach (see Figure 6). While the speedups are smaller than what can be achieved with sparsity pattern specific compilation approaches [CKSD17], similar matrices typically require similar layouts (see Tables 4 and 5). Therefore, our approach can be applied to domain-specific tuning of SpMVs, which can be performed beforehand for each new GPU using a domain-specific set of input matrices, resulting in shorter computation times, especially in computer graphics applications.

6.1. Limitations

For complex matrices, performance does not match the well-tuned operations provided by cuSPARSE, except on the older Quadro K2000 GPU. However, there is no reason not to use the well-supported vendor library in such cases.

The padded transposed layout without slicing can incur very large memory overheads. As even large memory overheads only slightly influence execution time, tuning for reduced memory while maintaining performance would require weighting performance and memory overhead for scoring. Alternatively, the compact CSR outer layout could always be generated as a fallback.

6.2. Future work

Extension to a more complete set of linear algebra operations would be beneficial. The `axpby` ($\mathbf{y} \leftarrow a\mathbf{x} + b\mathbf{y}$) and `dot` procedures are a good choice, as they would allow for implementation of several iterative solvers such as the conjugate gradient solver. Aside from the reduction within the dot product, however, they are trivially parallelizable and perform significantly fewer operations than the matrix-vector product, reducing the need for tuning.

While the Cartesian product approach to autotuning guarantees that the best variant is found, it is very expensive. By collecting a larger set of input matrices of varied structure, potentially by reusing the sparsity structure but not the entries of matrices in

existing matrix collections, a predictive tuning model could be built. This could potentially be achieved using machine learning (see, e.g. [AKC*19]). In addition, the search space could be reduced by tuning the number of threads per SM and factoring it into the number of threads per block and blocks per SM, instead of tuning both individually (see Section 5.4).

An extension to sparse tensor algebras as TACO offers for CPU codes [KKC*17] involves many interesting challenges. In particular, can the padded transposed and sliced layouts be generalized to tensors? How would they interact with hypersparse (low-rank) matrices or tensors? Slicing could potentially be represented as an index transform, i.e. $y_i = A_{ij}x_j \rightarrow y_i = A'_{\lfloor i/s \rfloor (i \bmod s)j}x_j$, where s is the slice size.

Another extension would be the support of encoded entries. For example, Mueller-Roemer *et al.* [MAS17] use sparse matrices to describe meshes. They encode the sign of a ternary matrix in the column index of CSR matrices. Compound entries can also benefit from compact encodings. For example, unit quaternion matrices used in various methods can be encoded using only three values and a sign bit, reducing memory and bandwidth requirements by nearly 25%. However, the advantage of alignment for single precision quaternions would be lost.

Acknowledgements

The authors are grateful to the best paper committee of VMV 2019 for recommending the original paper [MSF19] to Computer Graphics Forum, and providing us with an opportunity for presenting this extended work. This work was supported by the European Union project CloudiFacturing which is co-funded by the Horizon 2020 Framework of the European Union under Grant No. 768892.

References

[AKC*19] Ashouri A. H., Killian W., Cavazos J., Palermo G., Silvano C.: A survey on compiler autotuning using machine learning. *ACM Computing Surveys 51*, 5 (2019), 13:1–13:42. https://doi.org/10.1145/3197978.

[Bik96] Bik A. J. C.: *Compiler Support for Sparse Matrix Computations*. PhD thesis, Rijksuniversiteit Leiden, 1996.

[CGA18] CGAL: Computational Geometry Algorithms Library, 2018. https://www.cgal.org.

[CKPS18] Chern A., Knöppel F., Pinkall U., Schröder P.: Shape from metric. *ACM Transactions on Graphics 37*, 4 (2018), 63:1–63:17. https://doi.org/10.1145/3197517.3201276.

[CKSD17] Cheshmi K., Kamil S., Strout M. M., Dehnavi M. M.: Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis. In *SC '17: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA, 2017), Association for Computing Machinery (ACM). https://doi.org/10.1145/3126908.3126936.

[CPS11] Crane K., Pinkall U., Schröder P.: Spin transformations of discrete surfaces. *ACM Transactions on Graphics 30*, 4 (2011), 104:1–104:10. https://doi.org/10.1145/2010324.1964999.

[CPS13] Crane K., Pinkall U., Schröder P.: Robust fairing via conformal curvature flow. *ACM Transactions on Graphics 32*, 4 (2013), 61:1–61:10. https://doi.org/10.1145/2461912.2461986.

[DH11] Davis T. A., Hu Y.: The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software 38*, 1 (2011), 1:1–1:25. https://doi.org/10.1145/2049662.2049663.

[DZSS17] Derler A., Zayer R., Seidel H.-P., Steinberger M.: Dynamic scheduling for efficient hierarchical sparse matrix operations on the GPU. In *ICS '17: Proceedings of the International Conference on Supercomputing* (Chicago, IL, USA, 2017), Association for Computing Machinery (ACM). https://doi.org/10.1145/3079079.3079085.

[GR09] Geuzaine C., Remacle J.-F.: Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering 79*, 11 (2009), 1309–1331. https://doi.org/10.1002/nme.2579.

[HL90] Hile G. N., Lounesto P.: Matrix representations of Clifford algebras. *Linear Algebra and Its Applications 128* (1990), 51–63. https://doi.org/10.1016/0024-3795(90)90282-H.

[Jin14] Jin J.: *The Finite Element Method in Electromagnetics* (3rd edition). Wiley-IEEE Press, Hoboken, NJ, 2014.

[KCŽO07] Kavan L., Collins S., Žára J., O'Sullivan C.: Skinning with dual quaternions. In *I3D '07: Proceedings of the Symposium on Interactive 3D Graphics and Games* (Seattle, WA, USA, 2007), Association for Computing Machinery (ACM), pp. 39–46. https://doi.org/10.1145/1230100.1230107.

[KKC*17] Kjolstad F., Kamil S., Chou S., Lugato D., Amarasinghe S.: The tensor algebra compiler. *Proceedings of the ACM on Programming Languages 1*, *OOPSLA* (2017), 77:1–77:29. https://doi.org/10.1145/3133901.

[LJC17] Liu H.-T. D., Jacobson A., Crane K.: A Dirac operator for extrinsic shape analysis. *Computer Graphics Forum 36*, 5 (2017), 139–149. https://doi.org/10.1111/cgf.13252.

[LLA11] Lang H., Linn J., Arnold M.: Multi-body dynamics simulation of geometrically exact Cosserat rods. *Multibody System Dynamics 25*, 3 (2011), 285–312. https://doi.org/10.1007/s11044-010-9223-x.

[MAS17] Mueller-Roemer J. S., Altenhofen C., Stork A.: Ternary sparse matrix representation for volumetric mesh subdivision and processing on GPUs. *Computer Graphics Forum 36*, 5 (2017), 59–69. https://doi.org/10.1111/cgf.13245.

[MLA10] Monakov A., Lokhmotov A., Avetisyan A.: Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *Proceedings of 5th International Conference on High Performance Embedded Architectures and Compilers*

(Pisa, Italy, 2010), Springer Berlin Heidelberg, pp. 111–125. https://doi.org/10.1007/978-3-642-11515-8_10.

[MS18] Mueller-Roemer J. S., Stork A.: GPU-based polynomial finite element matrix assembly for simplex meshes. *Computer Graphics Forum 37*, 7 (2018), 443–454. https://doi.org/10.1111/cgf.13581.

[MSF19] Mueller-Roemer J. S., Stork A., Fellner D. W.: Joint schedule and layout autotuning for sparse matrices with compound entries on GPUs. In *VMV '19: Proceedings of Vision, Modeling and Visualization* (Rostock, Germany, 2019), The Eurographics Association, pp. 109–116. https://doi.org/10.2312/vmv.20191324.

[NVI19a] NVIDIA: CUDA C++ Programming Guide, Version 10.2. 2019. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Accessed 8 January 2020.

[NVI19b] NVIDIA: cuSPARSE Library, Nov. 2019. https://docs.nvidia.com/pdf/CUSPARSE_Library.pdf. Accessed 8 January 2020.

[RB85] Rice J. R., Boisvert R. F.: *Solving Elliptic Problems Using ELLPACK*. Springer, New York, NY, 1985.

[Saa03] Saad Y.: *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, 2003.

[SDZS16] Steinberger M., Derlery A., Zayer R., Seidel H.-P.: How naive is naive SpMV on the GPU? In *HPEC '16: Proceedings of IEEE High Performance Extreme Computing Conference* (Waltham, MA, USA, 2016), IEEE. https://doi.org/10.1109/hpec.2016.7761634.

[Sho85] Shoemake K.: Animating rotation with quaternion curves. *SIGGRAPH Computer Graphics 19*, 3 (1985), 245–254. https://doi.org/10.1145/325165.325242.

[Si15] Si H.: TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Software 41*, 2 (2015), 11:1–11:36. https://doi.org/10.1145/2629697.

[Sie18] Siemens: NX, 2018. https://www.plm.automation.siemens.com/global/en/products/nx/. Accessed 1 November 2018.

[Sta14] Stanford University: The Stanford 3D scanning repository, 2014. http://graphics.stanford.edu/data/3Dscanrep. Accessed 31 October 2019.

[Tas01] Tasora A.: An optimized Lagrangian-multiplier approach for interactive multibody simulation in kinematic and dynamical digital prototyping. In *Proceedings of International Symposium on Computer Simulation in Biomechanics* (Calgary, Canada, 2001), VIII ISCSB, International Society of Biomechanics.

[TBMG20] Tasora A., Benatti S., Mangoni D., Garziera R.: A geometrically exact isogeometric beam for large displacements and contacts. *Computer Methods in Applied Mechanics and Engineering 358* (2020), 112635:1–21. https://doi.org/10.1016/j.cma.2019.112635.

[Tho06] Thompson L. L.: A review of finite-element methods for time-harmonic acoustics. *Journal of the Acoustical Society of America 119*, 3 (2006), 1315–1330. https://doi.org/10.1121/1.2164987.

[VHS15] Venkat A., Hall M., Strout M.: Loop and data transformations for sparse matrix code. *ACM SIGPLAN Notices 50*, 6 (2015), 521–532. https://doi.org/10.1145/2813885.2738003.

[VOFG10] Vázquez F., Ortega G., Fernández J. J., Garzón E. M.: Improving the performance of the sparse matrix vector product with GPUs. In *CIT '10: Proceedings of the 10th IEEE International Conference on Computer and Information Technology* (Bradford, United Kingdom, 2010), IEEE, pp. 1146–1151. https://doi.org/10.1109/CIT.2010.208.

[WBS*13] Weber D., Bender J., Schnoes M., Stork A., Fellner D.: Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications. *Computer Graphics Forum 32*, 1 (2013), 16–26. https://doi.org/10.1111/j.1467-8659.2012.03227.x.

[WG17] Weber N., Goesele M.: MATOG: Array layout autotuning for CUDA. *ACM Transactions on Architecture and Code Optimization 14*, 3 (2017), 28:1–28:26. https://doi.org/10.1145/3106341.

[YDT*18] Ye Z., Diamanti O., Tang C., Guibas L., Hoffmann T.: A unified discrete framework for intrinsic and extrinsic Dirac operators for geometry processing. *Computer Graphics Forum 37*, 5 (2018), 93–106. https://doi.org/10.1111/cgf.13494.

[ZG18] Zhang J., Gruenwald L.: Regularizing irregularity: Bitmap-based and portable sparse matrix multiplication for graph data on GPUs. In *GRADES-NDA '18: Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Houston, TX, USA, 2018), Association for Computing Machinery (ACM). https://doi.org/10.1145/3210259.3210263.

Supporting Information

Additional supporting information may be found online in the Supporting Information section at the end of the article.