Automatically Generating and Tuning GPU Code for Sparse Matrix-Vector Multiplication from a High-Level Representation

Dominik Grewe
Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh, UK
dominik.grewe@ed.ac.uk

Anton Lokhmotov
Media Processing Division
ARM
Cambridge, UK
anton.lokhmotov@arm.com

ABSTRACT

We propose a system-independent representation of sparse matrix formats that allows a compiler to generate efficient, system-specific code for sparse matrix operations. To show the viability of such a representation we have developed a compiler that generates and tunes code for sparse matrix-vector multiplication (SpMV) on GPUs. We evaluate our framework on six state-of-the-art matrix formats and show that the generated code performs similar to or better than hand-optimized code.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation,Compilers,Optimization*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures

General Terms

Experimentation, Languages, Performance

Keywords

Code generation, automatic tuning, GPGPU, OpenCL, CUDA, SpMV, sparse linear algebra

1. INTRODUCTION

Sparse matrices, i.e. matrices in which the majority of elements are zeros, occur in many applications. A plethora of storage formats have been proposed to efficiently perform matrix operations involving matrices having specific structure (e.g. diagonal) or on specific hardware (e.g. GPUs). A brief overview of common formats and formats recently proposed for GPUs is given in section 2.

It is, however, rarely obvious which format is optimal for a particular combination of the matrix operation, matrix structure, and target hardware. Developing and maintaining multiple code versions becomes infeasible in practice even for a modest number of such combination.

We solve this problem by presenting a framework consisting of three components:

1. a high-level representation for describing sparse matrix formats;

2. a compiler for generating low-level code from the high-level representation;

3. an automatic tuner for the generated low-level code.

The user describes data structures for holding the matrix values and auxiliary data and a way of accessing the matrix elements. Section 3 uses the Compressed Sparse Row (CSR) format to illustrate the representation. The compiler generates either CUDA[1] or OpenCL[2] code which is then compiled by the native compiler for execution on the GPU.

The compiler extracts information necessary for generating efficient code for basic linear algebra operations. In this paper we focus on Sparse Matrix-Vector Multiplication (SpMV), the most widely used sparse linear algebra operation [1]. Our compiler supports several compilation strategies to provide optimal memory accesses to the sparse matrix data structures. The compiler also supports generating code using vector types for full efficiency on vector architectures. A detailed description of the code generation and optimization for SpMV is given in section 4.

Whilst generated code often achieves similar performance to hand-written code out-of-the-box, automatic tuning allows us to find the optimal configuration for a given matrix and hardware, such as the number of work-items working on a matrix row in parallel, which often provides even better performance than hand-written code. Sections 5 and 6 provide experimental evidence on the effectiveness of our methodology on six state-of-the-art sparse matrix formats.

2. BACKGROUND

The coordinate (COO) format is a widely-used format for representing sparse matrices (see figure 1b). Each non-zero element and its row- and column indices are stored in arrays

[1]`http://www.nvidia.com/cuda`
[2]`http://www.khronos.org/opencl`

$$\begin{bmatrix} 0 & 5 & 1 & 0 \\ 2 & 0 & 8 & 3 \\ 0 & 0 & 5 & 0 \\ 0 & 9 & 4 & 0 \end{bmatrix}$$

(a) Example matrix

```
values = [5 1 2 8 3 5 9 4]
colIdx = [1 2 0 2 3 2 1 2]
rowIdx = [0 0 1 1 1 2 3 3]
```

(b) Coordinate (COO) format

```
values = [5 1 2 8 3 5 9 4]
colIdx = [1 2 0 2 3 2 1 2]
rowPtr = [0 2 5 6 8]
```

(c) Compressed Sparse Row (CSR) format

$$\texttt{values} = \begin{bmatrix} 5 & 1 & * \\ 2 & 8 & 3 \\ 5 & * & * \\ 9 & 4 & * \end{bmatrix} \texttt{colIdx} = \begin{bmatrix} 1 & 2 & * \\ 0 & 2 & 3 \\ 2 & * & * \\ 1 & 2 & * \end{bmatrix}$$

(d) ELLPACK/ITPACK (ELL) format

Figure 1: Examples of the COO, CSR and ELL formats.

`values`, `rowIdx` and `colIdx`, respectively. While this format is simple and provides the basis for many portable matrix formats it is rarely suitable for computation. Because of the lack of structure (elements can be stored randomly) operations such as SpMV require costly synchronization when performed in parallel.

The compressed storage row (CSR) format is a more structured format, because each row is stored in a contiguous chunk of memory (see figure 1c). The column indices of the non-zero elements need to be stored explicitly, but the row indices are implicit in the position of the data value. To map each row to its chunk of elements, the `rowPtr` array stores the index of the first element in each row.

The ELLPACK/ITPACK [3] (ELL) format (see figure 1d) is particularly suitable for vector architectures. Let n be the maximum number of non-zero elements in a row. Then n elements are stored for each row with rows with less than n elements being padded with zeros. While this may add storage overhead, it simplifies computation and eliminates the need to store row indices as these can be computed from the position in the data array. The `values` and `colIdx` arrays are stored in column-major order, hence adjacent values belong to adjacent rows.

2.1 Related Work

SpMV on GPUs.

Sparse linear algebra is well researched [7, 8]. With the rise of GPGPU, different formats for representing sparse matrices have been proposed that are specifically suited for GPUs.

Bell and Garland [1] provide an overview of some well-known formats and also propose a new "hybrid" format: The majority of the matrix elements are stored in the ELL format while the remaining elements are stored in the coordinate (COO) format. This can be beneficial when the number of non-zero elements per row varies significantly, because it reduces the number of explicitly stored zeros in the ELL format.

Monakov et al. [5] introduce a modified version of ELL where a matrix is divided into slices and each slice is separately stored in ELL format. Combined with row reordering to bring together rows of similar size, this can significantly reduce the number of explicitly stored zeros. The size of the slices, i.e. the number of rows per slice, is either fixed for all slices or a heuristic is used to find a good size for each slice individually. They use auto-tuning to find the optimal configuration, e.g. the number of rows in a slice.

Choi et al. [2] propose another modification of ELL. They divide the matrix into small, dense blocks and store each block that contains non-zero elements contiguously. For matrices with an inherent block structure this can reduce the amount of storage and allows for some optimizations. Similar to Monakov et al. [5] they also divide the matrix into slices that are stored in a blocked version of the ELL format and use row reordering to reduce storage. Choi et al. also use auto-tuning to find, for example, the optimal block size. Additionally they propose a model for analytically deriving the optimal configuration.

Sparse linear algebra code generation.

We have been inspired by the work of Mateev et al. [4] who proposed an approach for code generation of sparse linear algebra from a high-level representation. There are two main differences between their approach and ours: First, they used C++ classes and iterators to describe matrix formats. We have also considered this approach, but found it easier for both the compiler writer and the user to work with a domain-specific abstract language. Second, they generated *sequential* code rather than *parallel* code. Generating good parallel code is fundamentally more difficult, since it requires specific optimizations, especially when targeting GPUs.

3. EXAMPLE: REPRESENTATION OF COMPRESSED STORAGE ROW FORMAT

Figure 2 shows the specification of the CSR format (section 2) in our representation language. Due to lack of space we do not show the formal grammar of our language.

The `parameters` section describes basic parameters that are needed to define data structures and how to access them. These parameters are constant for each matrix instance. They usually include the number of rows and columns in the matrix as well as some format-specific parameters. In the case of CSR, for example, the number of non-zero elements must be known.

The `data` section defines arrays storing the matrix values and other, auxiliary data. The array `values`, of type `DATA_TYPE`, stores the actual data values of the non-zero elements of the matrix. In CSR only the non-zero elements of the matrix are stored, hence the size of `values` is `num_nonzeros`, specified in square brackets. The column indices are stored in the `colIdx` array of the same size. Unlike `values`, this array stores indices rather than matrix values, as specified by its type. Another index array is `rowPtr` that points to the beginning of each row within the `values` and `colIdx` arrays. The size of the array is `num_rows + 1`, with the last element pointing behind the last matrix element.

To fully specify a matrix format, the programmer must also describe a way of efficiently accessing the matrix el-

```
################################################################################
###                      Compressed Storage Row (CSR)                        ###
################################################################################

# parameter description
parameters:
  num_rows
  num_cols
  num_nonzeros

# data storage description
data:
  values [num_nonzeros] : DATA_TYPE
  colIdx [num_nonzeros] : INDEX_TYPE
  rowPtr [num_rows + 1] : INDEX_TYPE

# access description
access:
  for _row in [ 0 : num_rows-1 ]
  {
    s = rowPtr [_row]
    e = rowPtr [_row + 1]
    for i in [ s : e-1 ]
    {
      _col = colIdx[i]

      _val = values[i]
    }
  }
```

Figure 2: Specification of the Compressed Storage Row (CSR) format.

ements. For CSR, the only efficient way of accessing the matrix is to iterate over rows and in each row iterate over the non-zero elements. This information is provided in the final section of our matrix representation in figure 2. The outer loop enumerates all row indices from 0 to `num_rows-1` (inclusive) using the index variable `_row`. The underscore ("`_`") indicates that this variable is a *built-in* variable that must get assigned in any valid access description. The `_row` variable identifies the row of the matrix element being accessed. Within each row, we iterate over all matrix elements in the row identified by `rowPtr`. The column index `_col` and the element value `_val` are set using the `colIdx` and `values` arrays, respectively.

While CSR is a relatively simple format, more complex formats have been specifically proposed for GPUs as described in section 2.1. Our simple language can express all these formats and more.[3]

4. GENERATING SPMV CODE FOR GPUS

We have developed a compiler that takes a matrix format description as input and generates code for sparse matrix-vector multiplication (SpMV) on GPUs. The compiler uses the `flex` and `bison` tools to parse the description and generate an abstract syntax tree (AST). Some transformations are applied to the AST as described in this section before it is passed to either the CUDA or the OpenCL backend for code generation. While CUDA programs can only be executed on NVIDIA GPUs, the OpenCL backend allows us to run the code on devices from different vendors, e.g. NVIDIA and AMD. But since most hand-written implementations are in CUDA, we have also developed a CUDA backend to allow for a fair comparison of our generated code to the hand-written code. Even though our compiler has not been optimized for performance, it only takes a few milliseconds to generate the SpMV code.

In SpMV, a sparse matrix M is multiplied by a dense vector x to compute the resulting dense vector y, thus $y = Mx$. This is done by computing the dot-product of each row in M with x, which requires a reduction across each row of M and hence some form of synchronization between computations in a row. Because both OpenCL and CUDA only support synchronization within a work-group,[4] there are basically two ways of efficiently implementing SpMV: we either assign *one* work-item or *multiple* work-items from the same work-group to process each row. If a single work-item works on each row ("1IPR"), no synchronization is required. If multiple work-items are assigned to the same row ("nIPR"), a reduction needs to be performed across these work-items (see section 4.4.1).

If a format does not allow for efficient random access to rows (e.g. the coordinate format), we provide two alternatives. The first only creates a single work-item and accesses matrix elements sequentially ("SEQ"). The second uses atomic functions to synchronize work-items assigned to the same row ("ATOMIC"). Both options generally perform poorly on GPUs, but may be useful for composite formats where a matrix is divided into sub-matrices that are stored differently, e.g. the hybrid format [1]. If only a small subset of the matrix is processed inefficiently this may not hurt performance significantly.

4.1 Optimizing memory bandwidth

Almost all sparse matrix formats proposed for SpMV on GPUs allow accessing entire matrix rows individually. The only exception is the hybrid format by Bell and Garland [1] where parts of the matrix are stored in the coordinate format. Whether to use one (1IPR) or multiple work-items per row (nIPR) depends on the memory layout of the format's data structures. On most GPUs it is important to "coalesce" memory accesses, i.e. successive work-items should access successive (and prefereably aligned) memory locations [6]. Choosing between 1IPR and nIPR follows naturally from the data layout: For example, in CSR (see section 2.1), the non-zero elements of a row are stored in a contiguous chunk of memory, requiring nIPR to achieve coalescing; on the other hand, in ELLPACK [1], the non-zero elements are stored in column-major order, requiring 1IPR to achieve coalescing. Because the best compilation option follows from the format's data layout, we leave it to the user to make this decision.

[3]All format specifications used in this paper can be found at `http://homepages.inf.ed.ac.uk/s0898672/spmv`.

[4]An OpenCL work-group corresponds to a thread-block in CUDA. We will use OpenCL terminology throughout this paper.

Blocked formats, e.g. blocked ELLPACK [2], divide the matrix in fixed-sized blocks. They also allow for accessing rows individually, but it is more efficient to process rows in batches of size equal to the block-height because of improved memory accesses. Our compiler thus detects when rows are accessed in fixed-sized blocks and generates code where a single work-item (for 1IPR) or a set of work-items (for nIPR) processes *multiple rows*.

4.2 Automatic vectorization

Because some GPUs, e.g. AMD's Evergreen series, are vector-based, vectorized code using OpenCL's built-in vector data types generally improves performance on these architectures. Depending on the number of work-items per row, automatic vectorization works as follows:

For nIPR the compiler vectorizes the inner loop of the computation for a single row. For the CSR format, for example, the inner loop with iterator `i` will be vectorized (see figure 2). Instead of accessing the `colIdx` and `values` arrays one at a time, data is loaded in batches of n values where n is the vector-width. Consequently, the loop count is reduced by a factor of n. Our compiler refuses to vectorize code if it cannot load the required values as a single block.

For 1IPR there are two alternatives: Vectorizing *across* multiple rows or *within* a single row. In the former case, instead of processing a single row, each work-item processes one row per vector lane. However this often leads to slowdowns over the scalar version, due to the reduced number of work-items. We have therefore implemented the latter case, vectorizing within a single row. Because most formats that are compiled using 1IPR do not allow batch-access to successive elements in a row, e.g. ELL, the memory layout needs to be changed slightly. In the case of ELL, for example, instead of storing one element of each row at a time, n elements are stored together, where n is the vector-width.

4.3 Exploiting data reuse

GPUs are designed to hide memory latency by interleaving execution of work-items. However, they also provide caches that can be exploited to improve performance.

4.3.1 Storing the input vector as an image

A common optimization for SpMV on GPUs is to store the input vector x as an image in texture memory, which is cached on-chip and significantly reduces memory latency and bandwidth when data is reused. Because elements of x are potentially reused but the accesses are data-dependent, using images improves overall performance. Our compiler provides an option to specify whether the input vector should be stored as an image or not. However, in our experiments using images was always better.

4.3.2 Eliminating loads from the input vector

For blocked formats, the number of loads from the input vector can be reduced, because elements in the same column of a block have to be multiplied by the same value. Our compiler exploits this characteristic by only loading the value once and then reusing it for all elements in a column.

Loads from the input vector can also be eliminated by checking if the matrix value is zero. Some formats, e.g. ELL, store some zero-elements explicitly. In these cases it may be beneficial to check if a value is non-zero before loading the corresponding element from the input vector. This transformation constitutes a trade-off of additional control flow and reduced fetches from memory. Depending on the matrix and the format, it may or may not be a beneficial optimization. We therefore include this compiler option in our search space.

Card	SDK
NVIDIA Tesla S1070	CUDA SDK 3.0/3.2
AMD Radeon HD 5970	Stream SDK 2.1

Table 1: Experimental setup. CUDA SDK 3.2 was only used for SELL which did not compile with version 3.0. Due to time-constraints we were not able to evaluate the other formats with version 3.2.

4.3.3 Caching matrix data

In many cases elements in data structures, e.g. matrix values or column indices, are only accessed once. Caching those values is thus not beneficial. However, there are some exceptions. For example in the diagonal format, the "offset" array stores the location of each diagonal in the matrix [1]. Because all work-items read the same value from the array, caching can significantly improve memory performance. We therefore introduced the `__reuse` keyword that allows the user to specify arrays whose values will be reused. The compiler will then generate code that stores the array as an image rather than a standard buffer. Another possibility would be to store these arrays in constant memory.

4.4 Further optimizations

4.4.1 Optimizing the reduction phase

The reduction needed for the nIPR option is implemented as a parallel reduction, i.e. values are combined in a tree-like fashion, taking care to avoid bank conflicts [6]. In general, this requires a synchronization after each reduction step; however, on some GPUs we can exploit the fact that a batch of work-items perform in lock-step,[5] i.e. they execute the same instruction at the same time. In this case, no synchronization is required between such batches of work-items.

4.4.2 Loop unrolling

For small loops with a fixed iteration space loop unrolling can improve performance. This is the case in blocked formats, for example, where small loops are used to iterate over a block. Currently our compiler always unrolls a loop if the iteration space is statically known. However, loop unrolling could be easily made optional to account for cases where it is not beneficial.

5. EVALUATION METHODOLOGY

In this section we describe how we compare the generated code to hand-written code. Furthermore, we compare vector code to scalar code on a vector-unit based GPU, showing the necessity for high-level descriptions to provide portability. Table 1 shows the setup of our experimental evaluation.

5.1 Formats

We evaluated six different sparse matrix formats and compared the performance of the generated code to hand-written

[5]NVIDIA call such a batch a "warp", AMD a "wavefront".

code: CSR, DIA, ELL and HYB from Bell and Garland [1], sliced ELLPACK (SELL) from Monakov et al. [5] and blocked ELLPACK (BELL) from Choi et al. [2]. These represent state-of-the-art formats for hand-optimized SpMV on GPUs. We re-evaluated the authors' original code on our hardware to enable a fair comparison to our generated code.

Because the hand-written code was implemented in CUDA, we also generate CUDA code. We compare the code versions in two ways. First, we show how the performance of the generated code compares to the hand-written code when using the same configuration (e.g. the work-group size or the block size for blocked formats). This shows to some extent the "quality" of the generated code. Second, we use auto-tuning to find the best configuration for our code, showing the maximum available performance. While Bell and Garland [1] use either fixed values or heuristics to find a good configuration, both Monakov et al. [5] and Choi et al. [2] also rely on auto-tuning to find the best configuration.

The time spent automatically tuning a format heavily depends on the format itself. In some cases, e.g. ELL, only very few parameters such as work-group sizes need to explored. However, for other formats, e.g. blocked formats, the search space is significantly bigger and thus auto-tuning is more expensive (up to one hour).

For most formats, the generated code is similar to the hand-written code. There are two noteworthy exceptions, however, which are described below.

The hybrid format proposed in [1] stores the majority of the matrix in the ELL format and the remainder in the COO format. Generally the COO format is unsuitable for GPUs, but Bell and Garland use "segmented reduction" to speed up the computation under the condition that the elements stored in COO are ordered by row. We were unable to generalize this idea and thus did not use it in our compiler. Because using both the SEQ and the ATOMIC compilation option (section 4) did not provide good performance, we decided to slightly change the hybrid format by using a modified version of CSR in conjunction with ELL: Rather than storing each row, we only store those rows that contain nonzero elements. Hence, when comparing the auto-generated code of HYB to the hand-written code, note that the two versions differ slightly.

There are two variants of the SELL format. The slices either have a fixed height or the height varies depending on the structure of the input matrix. Both versions are evaluated during auto-tuning and the best one is picked for each matrix. Each slice is stored in the ELL format, thus 1IPR generally leads to memory coalescing. With variable-height slices, however, slices may contain a small number of rows. In that case it is beneficial to have more work-items than there are rows working on a slice in column-major order to achieve memory coalescing. We therefore use a slight modification of 1IPR for SELL with variable-height slices. We assign one work-group to each slice and limit the height of a slice to the size of a work-group. If the number of rows in a slice equals the number of work-items in a work-group, exactly one work-item operates on a single row (just like 1IPR). If the number of rows in a slice is smaller than the size of the work-group, however, multiple work-items operate per row in column-major order.

5.2 Matrices

We evaluated the code on 14 different matrices used in [1].

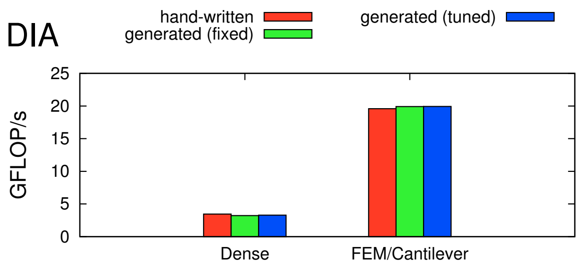

Figure 3: Performance of the generated code compared to the hand-written code for the DIA format. Only two of the 14 matrices are suitable for this format. The performance of the auto-generated code is almost identical to the hand-written code.

All previous work on SpMV on GPUs used this set of matrices, because it represents a wide spectrum of sparse matrices typically found in applications. For some matrices the code by Bell and Garland [1] would not produce results for the DIA or the ELL format, because these formats were deemed unsuitable for these matrices. In these cases we simply skipped the evaluation of the matrix for this format.

5.3 Vectorization

We also evaluated the performance of vectorized code over scalar code on a vector-unit based AMD GPU as described in table 1. Here we used the OpenCL backend, because the target platform does not support CUDA. Currently our compiler only supports vectorization for the nIPR (multiple work-items per row) compilation strategy. We therefore only show results for the CSR format. We have not evaluated the vector code on a scalar GPU yet.

6. EVALUATION RESULTS

This section presents the performance results of the code generated by our compiler. We compare the generated versions of various format to hand-written versions and show the benefits of vectorized code on vector-based architectures.

6.1 Evaluating the Generated Scalar Code

Figures 3 to 8 compare the performance of generated code to hand-written code for each of the six formats described in section 5.1. The left columns show the performance of the hand-written code versions using either fixed values (CSR, DIA and ELL), a heuristic (HYB) or auto-tuning (SELL and BELL) to find a good configuration. The performance of the auto-generated code using the same configuration is presented in the central columns. While this allows for a direct comparison between the two code versions, note that the versions may differ slightly (especially for the HYB format). The right columns show the performance of the generated code using auto-tuning.

When using auto-tuning the generated code for the CSR format (figure 4) clearly outperforms the hand-written version. Because the best configuration for the CSR format strongly depends on the input matrix, a fixed configuration rarely leads to good performance. The results not only show the benefits of auto-tuning but also the quality of the code generated by our compiler.

The DIA format is only suitable for very specific matrices, thus figure 3 only shows results for these matrices. The auto-

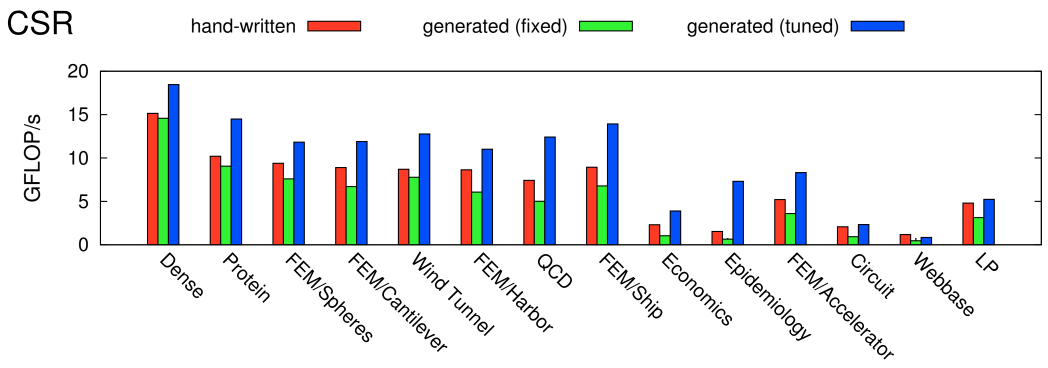

Figure 4: Performance of the generated code compared to the hand-written code for the CSR format. The central bar shows the performance of the generated code using the same configuration as the hand-written code. Using auto-tuning, the generated code outperforms the hand-written version for all but one matrix.

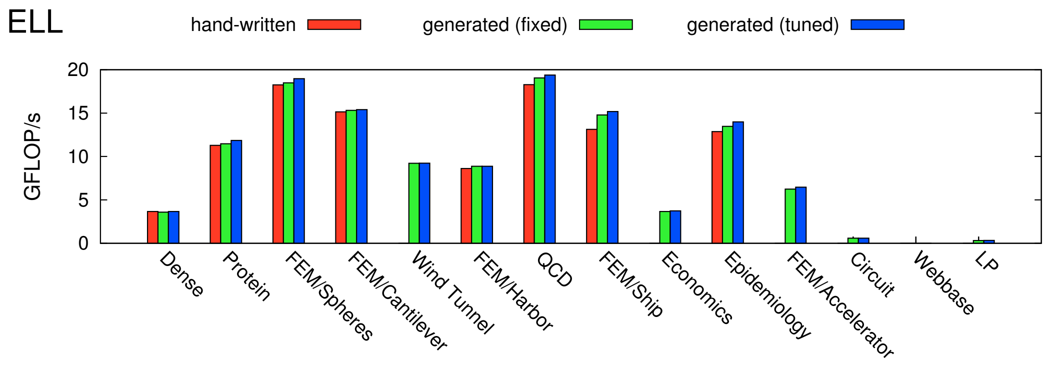

Figure 5: Performance of the generated code compared to the hand-written code for the ELL format. The code by Bell and Garland [1] refuses to benchmark the ELL format on some matrices it deems unsuitable for this format. The generated code almost always outperforms the hand-written version.

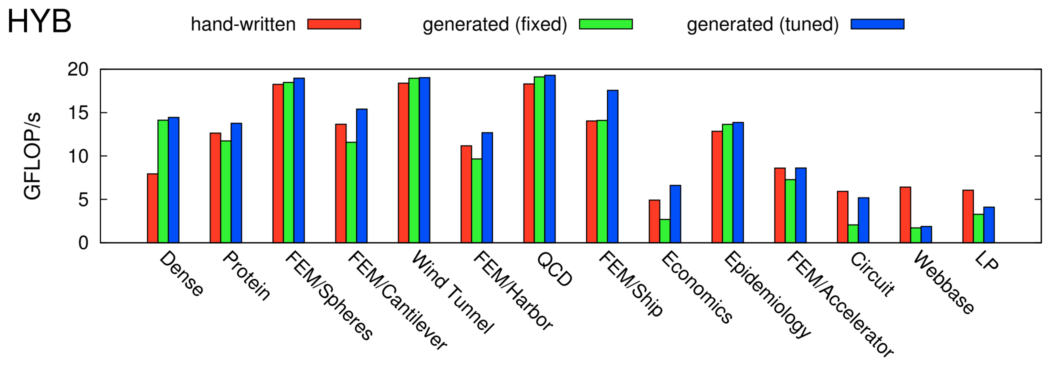

Figure 6: Performance of the generated code compared to the hand-written code for the HYB format. The central bar shows the performance of the generated code using the heuristic in [1]. For most matrices, the generated code for our version of HYB (see section 5.1) outperforms the hand-written one.

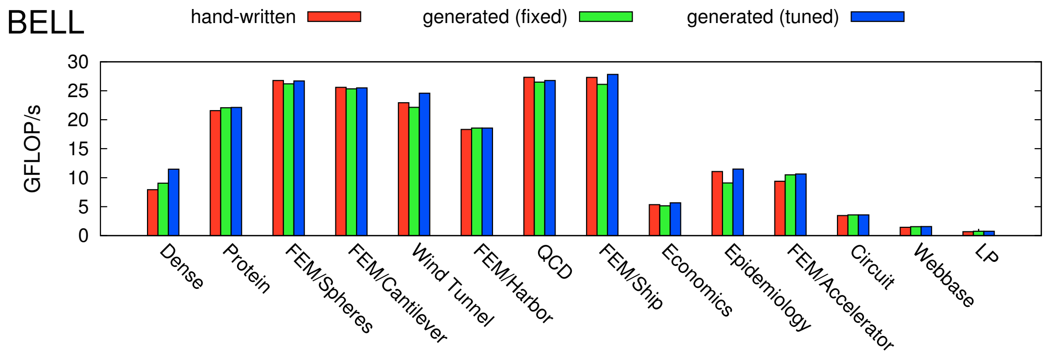

Figure 7: Performance of the generated code compared to the hand-written code for the BELL format. The central bar shows the performance of the generated code using the configuration found by auto-tuning in [2]. Both versions achieve comparable performance.

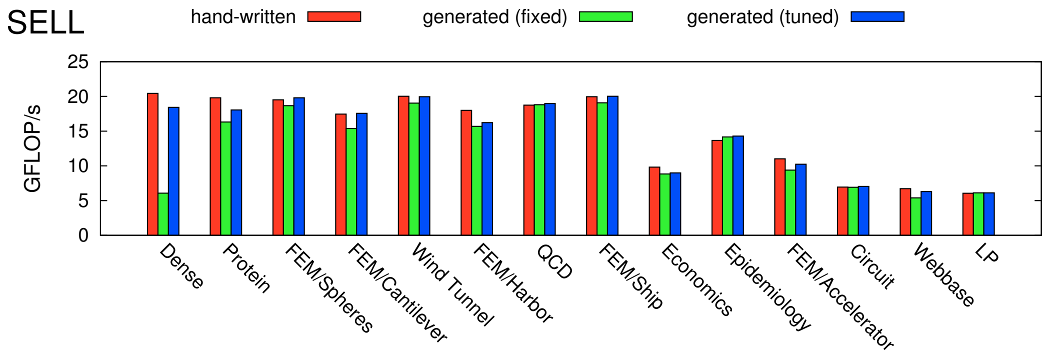

Figure 8: Performance of the generated code compared to the hand-written code for the SELL format. The central bar shows the performance of the generated code using the configuration found by auto-tuning in [5]. Both versions achieve comparable performance. The only exception is the dense matrix, where using a fixed configuration for the generated code leads to poor performance as explained in section 6.1.

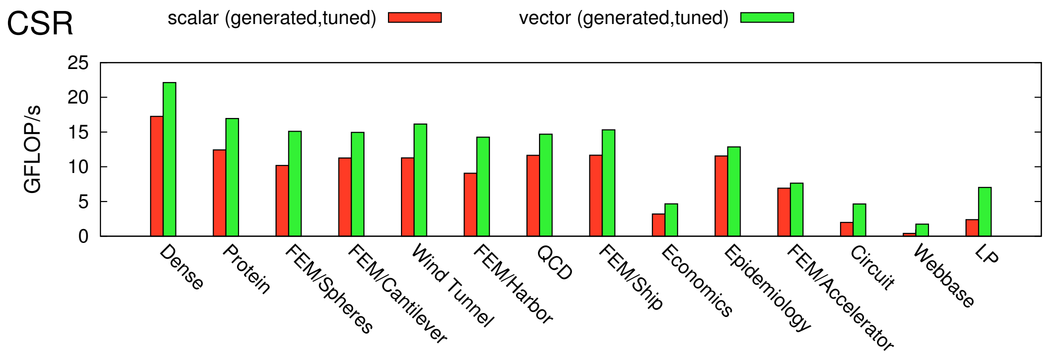

Figure 9: Performance comparison of the generated OpenCL code for the CSR format using a scalar and a vector version. On a vector-based GPU (AMD Radeon HD 5970) the vectorized version outperforms the scalar version by up to a factor of 4.

generated code shows comparable performance to the hand-written version and automatic tuning is hardly beneficial.

Results for the ELL format displayed in figure 5 show that the generated code slightly outperforms the hand-written version. Some results for the hand-written version are missing because Bell and Garland [1] deploy a heuristic for determining whether the ELL format is suitable for a given matrix.

Figure 6 shows the results for the HYB format. It has to be noted that rather than using the coordinate format to store parts of the matrix, the generated code uses a variation of CSR (as explained in section 5.1). For most matrices the performance of the two versions is comparable, with the generated code outperforming the hand-written version for 9 out of 14 matrices. Similarly, the performance of both versions for the BELL format is roughly equal as presented in figure 7.

The performance of the generated code for the SELL format often matches the performance of the hand-tuned version, and sometimes even exceeds it. For the dense matrix the code generated with a fixed configuration is significantly worse than the hand-written version. This is due to a difference in the code for SELL with fixed-height slices. The hand-written version uses potentially multiple work-items per row, whereas the generated version always uses a single work-item per row. Hence, the optimal configuration for the hand-written version is a bad choice for the generated code in this case. Auto-tuning can make up for this by finding a configuration whose performance almost matches the performance of the hand-written version.

6.2 Evaluating the Generated Vector Code

As explained in section 5.3, we compared a scalar version of CSR to a vector version on a vector-based GPU. The performance results on all 14 matrices are shown in figure 9. For every single matrix the vector version clearly outperforms the scalar version, with speedups of up to a factor of 4. The average performance gains of vectorizing the code are 1.6x (geometric mean). This shows that even though SpMV is a memory-bound operation, specializing the code for a particular architecture can have significant benefits.

7. CONCLUSION

In this paper we have introduced an abstract representation language for sparse matrix formats. Given a description of the format a compiler can generate code for various matrix operations. This was demonstrated by developing a compiler that takes a format description and automatically generates code for sparse matrix vector multiplication (SpMV) on GPUs. The generated code provides similar and sometimes even better performance compared to hand-written code. Our framework also allows for automatically tuning the SpMV code, which often improves performance even further. Additionally, the format description can be used to automatically generate vectorized code to fully exploit the capabilities of vector-architectures.

The results given in this paper demonstrate the benefits of our approach: providing comparable performance to hand-written code while eliminating the need for machine-specific tuning. This ensures that format specifications are portable across systems, because rewriting the code is not necessary. Currently we have only implemented a compiler generating SpMV code, but we believe that our format representation language is general enough to allow for other sparse matrix operations such as matrix-matrix multiplication.

7.1 Future Work

Our framework provides an easy way for exploring new sparse matrix formats. To evaluate a new format one simply writes an abstract representation of the format and the compiler is responsible for the tuning that is required for an appropriate comparison of the new format to existing ones. This enables efficient experimentation with new formats, e.g. new hybrid formats. To simplify such experimentation, it would also be helpful to automatically generate code for converting matrices to a specified format.

The automatic tuning of code by exhaustively evaluating the configuration space can be prohibitively expensive. For some formats, e.g. BELL, it can take up to an hour to find the best configuration for a given matrix. As this process is highly matrix-dependent it needs to be repeated for every single matrix. In practice, this is often infeasible. We are planning to investigate the use of machine learning techniques to significantly shorten this process. Machine learning could be used to both predict the most appropriate format for a matrix and then predict the optimal configuration for this format without the need of exhaustive experiments.

Acknowledgements

The authors thank the European Network of Excellence on High Performance and Embedded Architectures and Compilation (HiPEAC) for funding Dominik Grewe's internship at ARM.

8. REFERENCES

[1] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC*, 2009.

[2] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *PPoPP*, 2010.

[3] David R. Kincaid, John R. Respess, and David M. Young. ITPACK 2.0 user's guide. Technical Report CNA-150, Center for Numerical Analysis, University of Texas, Austin, Texas, 1979.

[4] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *ICS*, 2000.

[5] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *HiPEAC*, 2010.

[6] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, 2008.

[7] Richard W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, CA, USA, 2004.

[8] Samuel Williams, Leonid Oliker, Richard W. Vuduc, John Shalf, Katherine A. Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 2009.