UPPSALA
UNIVERSITET

# Project assignment
Accelerator-based programming

Niklas Wik

Niklas.Wik.9911@student.uu.se

Uppsala

October 24, 2021

## Introduction

Accelerating dense linear algebra with the use of GPUs, and achieving good performance in respect to the hardwares potential, is relatively easy to implement, as coalesced memory access comes naturally from the data layout and operations. However, in many high-performance computing applications, sparse matrix structures arises which presents additional challenges in utilizing the GPUs full potential. One of the most important operations for many applications, such as iterative methods for solving linear systems and eigenvalue problems, is the sparse matrix vector multiplication (SpMV).

In this project, two different sparse matrix storage formats are investigated and implemented, where access patterns and bandwidth throughput are of interest. The two storage formats in question is the popular and established general-purpose matrix representation *Compressed Row Storage* (CRS), and the *ELLPACK* variant *SELL-C-$\sigma$* introduced in [2], with $\sigma = 1$, which is also known as *sliced ELLPACK*. To evaluate the two different formats, the iterative conjugate gradient method will be used to solve a three dimensional finite element problem on both a CPU and a GPU. The iterative solver, as well as, a vector class, a sparse CRS matrix class, and a code with the finite element problem with hard-coded matrix entries were all provided. Missing in the provided code was CUDA kernels for the CRS matrix vector multiplication and a general vector addition, as well as transfer methods between the host and device for both the vector and the CRS matrix class. A modification to the matrix class was implemented where the SELL-C-$\sigma$ format was used.

## The CRS format

The CRS is a very straight forward format which stores all nonzero elements and its corresponding column indices in two one dimensional arrays `values` and `column_indices`. A third array `row_starts` stores the information of where each row in the matrix starts in the `values` and `column_indices` arrays. More particular, (`row_starts[i+1]-row_starts[i]`) will correspond to the number of nonzero elements on row `i`. The CRS format is efficient with respect to memory usage for irregular matrices with multiple nonzero values for each row.

$$\mathbf{A} = \begin{bmatrix} 0 & 3 & 1 & 0 \\ 4 & 0 & 0 & 7 \\ 0 & 0 & 6 & 0 \\ 9 & 0 & 5 & 3 \end{bmatrix}, \qquad \begin{aligned} \texttt{values} &= \begin{bmatrix} 3 & 1 & 4 & 7 & 6 & 9 & 5 & 3 \end{bmatrix} \\ \texttt{column\_indices} &= \begin{bmatrix} 1 & 2 & 0 & 3 & 2 & 0 & 2 & 3 \end{bmatrix} \\ \texttt{row\_starts} &= \begin{bmatrix} 0 & 2 & 4 & 5 & 8 \end{bmatrix} \end{aligned}$$

*CRS matrix of the example matrix A. Note that* `row_starts` *has length $M+1$ of a matrix of size $M \times N$, where* `row_starts[M+1]` *corresponds to the number of nonzeros in the matrix* $\mathbf{A}$.

The provided SpMV kernel for the CPU is given by

```
for (row = 0; row < n_rows; ++row) {
    sum = 0;
    for (idx = row_starts[row]; idx < row_starts[row + 1]; ++idx)
        sum += values[idx] * src(column_indices[idx]);
    dst(row) = sum;
}
```

The kernel is easily parallelized with openMP over the outer loop. When it comes to SIMD, the compiler (g++ (GCC) 8.3.0) is not able to automatically vectorize the inner loop since there is non consecutive access of the `src` vector. One can unroll the inner loop and vectorize by hand, however a secondary loop for the remaining values in each row has to be used which adds additional overhead. Furthermore, in order to gain any performance from the SIMD instructions, the number of non-zeros in each row needs to be substantially larger than the SIMD width for the additional overhead of the SIMD instruction to not be dominant. In the given finite element problem, the non-zeros in each row varies between 8, 12, 18, and 27, which is arguably not substantially larger than the 8 single precision, or 4 double precision values that fit inside 256-bit SIMD vectors of the used CPUs (two Intel Xeon E5-2660). The performance of the kernel using SIMD instructions was however never investigated.

Two CUDA SpMV kernels were implemented for the GPU. The first implementation parallelizes over the outer loop as the CPU kernel, that is, each thread computes the result over one row in the matrix. The kernel will be referred to as *Thread* and is given by

```
template <typename Number>
__global__ void compute_spmv_thread(const std::size_t n_rows,
                                    const std::size_t *row_starts,
                                    const unsigned int *column_indices,
                                    const Number *values,
                                    const Number *src,
                                    Number *dst) {

    const int row = threadIdx.x + blockIdx.x * blockDim.x;
    if (row < n_rows){
        Number sum = 0;
        for(int i = row_starts[row]; i < row_starts[row + 1]; i++)
            sum += values[i] * src[column_indices[i]];
        dst[row] = sum;
    }
}
```

The obvious drawback of the implementation is the non-coalesced memory access of the arrays values and column_indices. The other kernel instead lets a warp compute the result of one row, so that coalesced access is achieved. The kernel will be referred to as *warp* and is given by

```
template <typename Number>
__global__ void compute_spmv_warp(const std::size_t n_rows,
                                  const std::size_t *row_starts,
                                  const unsigned int *column_indices,
                                  const Number *values,
                                  const Number *src,
                                  Number *dst) {
    __shared__ volatile Number sum[32];
    const int row = blockIdx.x;
    const int tid = threadIdx.x;
    if (row < n_rows){ // should always be true
        const unsigned int row_strt = row_starts[row];
        const unsigned int row_end = row_starts[row+1];
        if (tid + row_strt < row_end){
            sum[tid] = values[row_strt + tid]
                        * src[column_indices[row_strt + tid]];
        }
        else {sum[tid] = 0;}

        if(tid < 16) sum[tid] += sum[tid + 16];
        if(tid < 8) sum[tid] += sum[tid + 8];
```

```
            if(tid < 4) sum[tid] += sum[tid + 4];
            if(tid < 2) sum[tid] += sum[tid + 2];
            if(tid == 0) dst[row] = sum[tid] + sum[tid + 1];

        }
    }
```

Note that it is assumed that the number of nonzero elements in each row is always less or equal to the warp size, 32. The obvious drawback from this implementation is instead that the threads are doing too little work, with some doing non at all, as well as the additional overhead of the reduction within the warp. The reduction overhead mimics the SIMD instruction overhead discussed for the CPU kernel. Again, if the number of non-zeros of each row was large enough, the overhead would not be as dominant.

### Kokkos

The Kokkos library provides kernels of sparse linear algebra operations, where the CRS matrix format is used and implemented with their Kokkos::View datatypes of rank-1 and rank-2. Their SpMV operation is of course more general than the simple implementations in this project. The kernel computes $y = \beta y + \alpha \mathrm{Op}(A)x$, where $A$ is a $m \times n$ matrix, $y$ and $x$ are vectors of length $m$ and $n$, and $\alpha$ and $\beta$ are scalars. The three different modes (Op) provided are the same as for the cuSPARSE kernel; non transpose, transpose, and conjugate transpose. In addition, multiple vector multiplications is provided where if $x$ is given as a rank-2 Kokkos::View, the multiplication is made for each $x(:, i)$ with $y(:, i)$ as result. The implementation uses single level parallelizm when run on a CPU, and three level parallelizm when run on a GPU to ensure good performance on different architectures. Furthermore, if available, the implementation uses the cuSPARSE backend. Moreover, if a structured grid is used, which for instance occurs when solving finite element or finite difference problems, a variant of the SpMV kernel is provided, which takes the stencil type as an input.

The use of the CRS format allows for the Kokkos kernels to interface with other libraries such as Trilinos and their distributed sparse linear algebra package Tpetra, as well as of course the cuSPARSE library. Because of this, and its good performance over a variety of platforms it might be an advantage to use the Kokkos kernels in application code.[1][3][4]

## *Sliced* ELLPACK and SELL-C-$\sigma$

The *sliced* ELLPACK and the SELL-C-$\sigma$ format introduced in [2] is built upon the ELLPACK format, which stores the values and column indices column-wise and uses zero-padding so that all rows has the same amount of values. The format allows coalesced access when parallelizing over the rows, but in the same time introduces an increase in memory usage, which is important when transferring data back and forth, as well as additional computations for each row. If for instance only one row in a matrix has double the amount of non-zeros elements, memory usage and the number of computations will double for all rows in the matrix.

The *sliced* ELLPACK format improves on this by dividing the data into chunks of equal size C, where the rows are zero-padded to the length of the row with the most non-zero elements within the chunks. The values are stored column-wise within the chunks, and the chunks are stored consecutively. Moreover, the number of rows needs to be padded to be a multiple of the chunk size. The format can improve on the drawbacks of the ELLPACK format, but can in the same time have about equal memory usage and performance if every chunk includes a row with a lot more non-zero values than the rest of the rows within the chunks. Here is where the SELL-C-$\sigma$ format comes in, which introduces initial sorting of the rows such that equally sized rows are stored together. The parameter $\sigma$ indicates the number of rows in which the sorting should occur. However, in this project the sorting scope $\sigma$ was set to 1, which obtains the *sliced* ELLPACK format.

The *sliced* ELLPACK, or SELL-C-1 format uses two arrays `values` and `column_indices` to store the non-zero values (with the zero padding) and the column indices to each value. Furthermore, the arrays `chunk_starts` and `chunk_widths` stores the information about where in `values` and `column_indices` each chunk starts, and how wide each chunk is. Note that width of a chunk can be computed as `chunk_widths[i]` = (`column_indices[i+1]-column_indices[i]`)/C, where C is the chunk size. For a $M \times N$ matrix, the number of chunks is the integer division $n\_chunks = (M + C - 1)/C$, and the length of `chunk_widths` and (`column_indices` are $n\_chunks$ and $n\_chunks + 1$ respectively.

$$\mathbf{A} = \begin{bmatrix} 0 & 3 & 1 & 0 \\ 4 & 0 & 0 & 7 \\ 0 & 0 & 6 & 0 \\ 9 & 0 & 5 & 3 \end{bmatrix}, \qquad \begin{aligned} \texttt{values} &= \begin{bmatrix} 3 & 4 & 1 & 7 & 6 & 9 & 0 & 5 & 0 & 3 \end{bmatrix} \\ \texttt{column\_indices} &= \begin{bmatrix} 1 & 0 & 2 & 3 & 2 & 0 & 2 & 2 & 2 & 3 \end{bmatrix} \\ \texttt{chunk\_starts} &= \begin{bmatrix} 0 & 4 & 10 \end{bmatrix} \\ \texttt{chunk\_widths} &= \begin{bmatrix} 2 & 3 \end{bmatrix} \end{aligned}$$

*SELL-2-1 sparse matrix of the example matrix A. Note that* `chunk_starts[n_chunks+1]` *corresponds to the number of entries in* `values` *and* `column_indices`. *Here is also the column indices of the padded zeros set to the first index of the row.*

When using the *sliced* ELLPACK or SELL-C-1 format on a Nvidia GPU, parallelizing over the chunks and rows within the chunks is the obvious choice since we seek coalesced memory access of the sparse matrix. The choice of chunk size C should therefore be a multiple of the warp size 32, where a good choice is just C=32 as it might reduce the zero-padding for some chunks. The implemented kernel will be referred to as *scs* and is given by

```
template <typename Number>
__global__ void compute_spmv(const std::size_t n_rows,
                             const std::size_t *chunk_starts,
                             const std::size_t *chunk_widths,
                             const unsigned int *column_indices,
                             const Number *values,
                             const Number *src,
                             Number *dst) {
    const unsigned int chunk = blockIdx.x;
    const unsigned int row = threadIdx.x;
    const unsigned int real_row = row + chunk * blockDim.x;
    const unsigned int cs = chunk_starts[chunk];

    Number sum = 0;
    for (unsigned int i = 0; i < chunk_widths[chunk]; i++) {
        sum += values[cs + row + i * blockDim.x] *
                  src[column_indices[cs + row + i * blockDim.x]];
    }
    if (real_row < n_rows) dst[real_row] = sum;
}
```

Note that the number of rows of the original matrix $n\_rows$ is used, since the dense vectors `src` and `dst` are not padded to the new height of the sparse matrix which is a multiple of the chunk size C. Same as for the *warp* kernel, coalesced access of the arrays `values` and `column_indices` is achieved, however all threads in the *scs* kernel are active and executes more work, which of course is profitable since we always want every thread to do enough work for the parallelization to be worthwhile. In the *scs* kernel there is also no reduction overhead, although some extra unnecessary operations are done following the zero-padding. The additional operations are not expected to significantly reduce the performance more than the increase in performance from the better parallelization, together with coalesced memory access, and the lack of the warp reduction. If sorting of the rows was introduced with $\sigma > C$, the extra unnecessary work would be lowered as less zero padding would be used, but might result in a cost of extra overhead when accessing the dense vectors in an unordered way.

When it comes to using *sliced* ELLPACK or SELL-C-1 format for a CPU implementation, the choice of chunk size C may allow for vectorization over the rows in each chunk. E.g using a CPU with 256-bit vectors (as the Intel Xeon E5-2660) C=4 should be used. The implemented SpMV kernel for the CPU is given by

```
for (chunk = 0; chunk < n_chunks; ++chunk) {
    cs = chunk_starts[chunk];
    sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0;
    for (i = 0; i < chunk_widths[chunk]; ++i){
            sum0 += values[cs + 0 + i * C]
                    * src(column_indices[cs + 0 + i * C]);
            sum1 += values[cs + 1 + i * C]
                    * src(column_indices[cs + 1 + i * C]);
            sum2 += values[cs + 2 + i * C]
                    * src(column_indices[cs + 2 + i * C]);
            sum3 += values[cs + 3 + i * C]
                    * src(column_indices[cs + 3 + i * C]);
    }
    dst(chunk * C + 0) = sum0;
    dst(chunk * C + 1) = sum1;
    dst(chunk * C + 2) = sum2;
    dst(chunk * C + 3) = sum3;
}
```

which uses 4-way loop unrolling of the chunks of size C=4. Note that the outer loop is easily parallelized with OpenMP in the same way as for the CRS kernel, where each thread uses cached memory access. Moreover, it is required that the `dst` and `src` vectors are of length `C*n_chunks`. The compiler is again not able to vectorize the inner loop since there is non consecutive access to the `src` vector. One would instead have to vectorize by hand, however without the need of a secondary remainder loop compared to the CRS kernel. This vectorization corresponds to unrolling the outer loop of the CRS kernel but with cashed access. Since none of the CPU kernels were vectorized in the implementation, one could expect better performance of the CRS implementation since both kernels has cached access, and since the extra zero padding only introduces extra operations to be made for each row of the SELL-C-$\sigma$ implementation.

## Kokkos

Similar to the existing Kokkos kernels mentioned above which uses the CRS format, the SELL-C-$\sigma$ is of course also possible to implement with Kokkos. The easiest way to implement the format is to again only use rank-1 or rank-2 Kokkos::View datatypes where for instance the `values` and `column_indices` arrays can be stored together. Since the chunks can have different widths, the use of rank-3 Views is not feasible.

To ensure performance on both CPUs and GPUs we need cached and coalesced memory access respectively as the implementations shown above have. However, since the two different kernels use different parallelization patterns we need hierarchical parallelism, as the CPU kernel only parallelizes over the chunks (outer level), and the GPU kernel parallelizes both over the chunks and over each row within the chunks (inner level). The use of *thread teams* will enable this adaptation, where the *team size* and *number of teams* are set depending on the execution space. The different thread teams execute the outer level parallelism and the threads within a team execute the inner parallelism. It follows easily that the team size should be one for the CPU, and the chunk size C for a Nvidia GPU (where C is a multiple of 32).

## Result

The different kernels for the two sparse matrix formats were evaluated by first measuring the time it took to do 200 SpMV operations, where also the memory throughput in GB/s was observed. Correctness of all kernels was confirmed by observing a $4^{th}$ order convergence of the $L^2$-norm of the error with respect to the number of discrete points $N$ in $x$, $y$, and $z$ direction of the finite element problem. Moreover, the transfer time of the sparse matrices and the two vectors from the host to the device was measured and compared to the SpMV kernel timings. Lastly, the iterative Conjugate gradient solver was used and timed. Since the number of iterations until convergence of the solver would vary slightly for the different kernels, matrix sizes, and precision, comparison of the number of million updates per second and number of iterations was made. Measurements was made for both single and double precision. All tests where done on the UPPMAX Snowy cluster and the CPU kernels were run using 16 openMP threads on 2 Intel Xeon E5-2660.

| | |
|---|---|
| GPU | Nvidia Tesla T4 |
| CPU | 2x Intel Xeon E5-2660 |
| Compiler CPU | g++ (GCC) 8.3.0 |
| Compiler GPU | nvcc V10.1.243 |

Table 1: Test specifications.

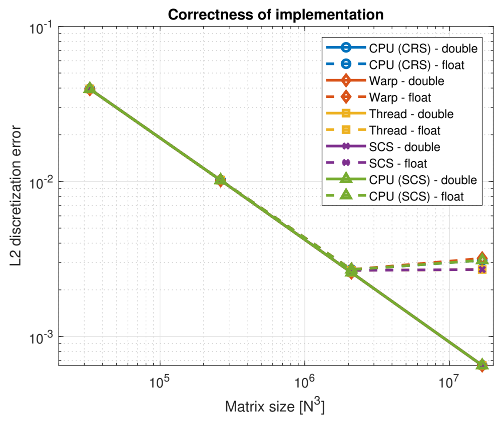

Figure 1: Convergence plot for all kernels to confirm correct implementations.

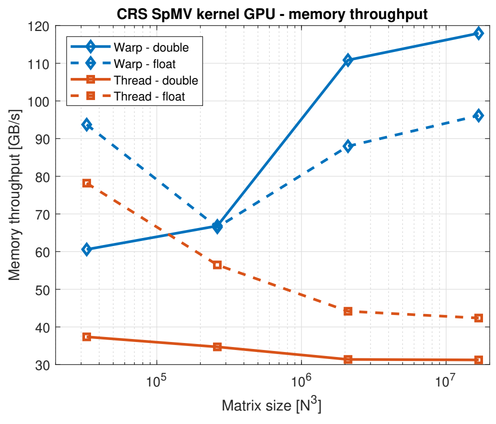

Figure 2: Memory throughput in GB/s for the two different CUDA SpMV kernels using the CRS format.

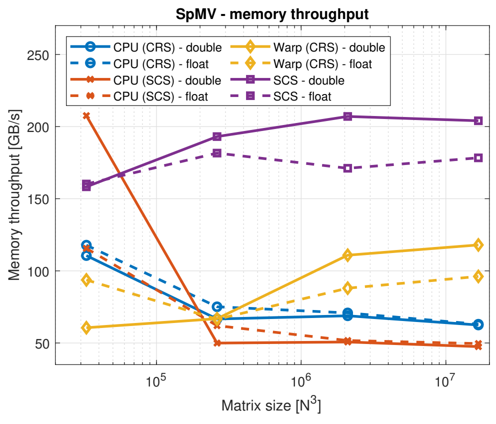

Figure 3: Memory throughput in GB/s for CPU and CUDA SpMV kernels for both the CRS and SELL-C-$\sigma$ (SCS) matrix format.

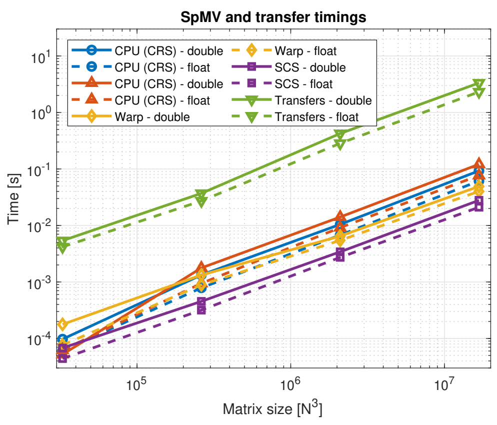

Figure 4: Timings in seconds for CPU and CUDA SpMV kernels for both the CRS and SELL-C-$\sigma$ (SCS) matrix format, as well as the combined transfer time for the matrices and vectors from the host to the device.

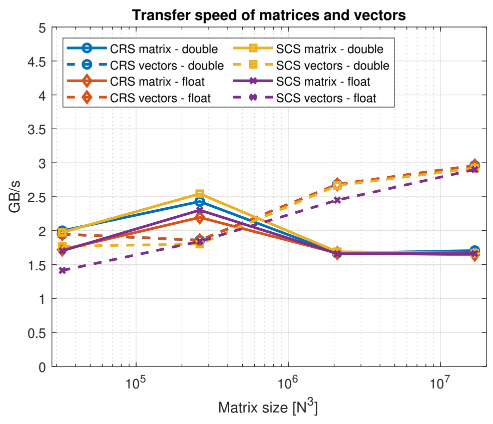

Figure 5: Transfer speed in GB/s of the matrices and 2 vectors from the host to the device.

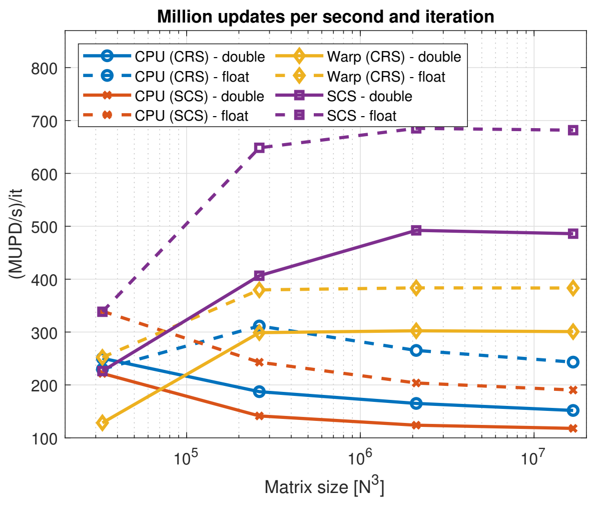

Figure 6: Millions updates per second and number of iterations of the Conjugate gradient solver while using the CPU and CUDA SpMV kernels for both the CRS and SELL-C-$\sigma$ (SCS) matrix format.

## Discussion

First of all, as shown in Figure 2, it is clear that the *thread* kernel, where each thread computes one row each, performs much worse than the *warp* kernel, where each warp computes one row. It speaks for the importance of coalesced memory access, even though not all threads are active in the *warp* kernel, as well as its reduction overhead.

Moreover, in both Figure 3 and 4 it is shown that the *sliced* ELLPACK or SELL-C-1 matrix format is more fitting for the GPU than the CRS format, as more work is done by each thread, in the same time as coalesced memory access is achieved. It shows in the same time that the SELL-C-1 format performs worse on the scalar implementations on the CPU, which was expected, neglecting the good performance for $N = 32$ with double precision (which probably is some caching effect). Furthermore, it is of course important to solve problems big enough such that the GPU is fully saturated, which is not the case when $N = 32$ and $N = 64$.

In Figure 4 we can observe that transferring the matrix and two vectors from the host to the device takes about 100 times longer than computing the SpMV CUDA kernel with the SELL-C-1 format, and still about 30 times longer than CPU kernel using the CRS format. So in order to have any performance benefit of using the GPU, the number of SpMV computations to be done needs to be substantial enough to overcome the data transfer bottleneck. In Figure 5 we can observe the poor transfer speed achieved. One way to overcome this bottleneck might be to assemble the matrices and vectors directly on the device. If this would be faster is however unclear, but an efficient implementation is most likely important.

Lastly, in Figure 6 we see again the importance of running problems large enough in order to use the GPUs full potential and gain any performance at all. It is of course most likely possible to implement a kernel which saturates the GPU better for smaller problem sizes, using three level parallelism. Although, for $N > 64$ we can again observe the benefit of using a GPU and that the *sliced* ELLPACK or SELL-C-1 sparse matrix format is a better choice than the CRS format for the GPU because of the improved thread work with coalesced memory access.

## References

[1] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* 74.12 (July 2014). ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003. URL: https://www.osti.gov/biblio/1106586.

[2] Moritz Kreutzer et al. "A unified sparse matrix data format for modern processors with wide SIMD units". In: *CoRR* abs/1307.6209 (2013). arXiv: 1307.6209. URL: http://arxiv.org/abs/1307.6209.

[3] NVIDIA. *cuSPARSE, the CUDA sparse matrix library.* Last Updated October 20, 2021. 2021. URL: https://docs.nvidia.com/cuda/cusparse/index.html.

[4] Sivasankaran Rajamanickam et al. "Kokkos Kernels: Performance Portable Sparse/Dense Linear Algebra and Graph Kernels". In: *CoRR* abs/2103.11991 (2021). arXiv: 2103.11991. URL: https://arxiv.org/abs/2103.11991.