SpezialForschungsBereich F 32

Karl–Franzens Universität Graz
Technische Universität Graz
Medizinische Universität Graz

GPU Parallelization for Unstructured Sparse Matrix Problems with OpenMP 4.5 and OpenACC

S. Rosenberger G. Haase

SFB-Report No. 2017-010 October 2017

A–8010 GRAZ, HEINRICHSTRASSE 36, AUSTRIA

Supported by the
Austrian Science Fund (FWF)

Der Wissenschaftsfonds.

SFB sponsors:

- **Austrian Science Fund** (FWF)
- **University of Graz**
- **Graz University of Technology**
- **Medical University of Graz**
- **Government of Styria**
- **City of Graz**

GPU Parallelization for Unstructured Sparse Matrix Problems with OpenMP 4.5 and OpenACC

Stefan Rosenberger and Gundolf Haase

University of Graz, 8010 Graz, Austria,
stefan.rosenberger@uni-graz.at,
WWW home page: https://mathematik.uni-graz.at/en/

Abstract. The effective use of parallelized hardware is an important goal of today's computer developments. Nvidia GPUs are an important footing in this context. While CUDA implemented algorithms focus on detailed optimized usage of GPU elements the pragma directive parallelization targets GPU computation for a broader community.
In this paper we focus on the implementation of OpenACC and OpenMP 4.5 parallelization for Nvidia GPUs for a sparse matrix solver on unstructured discretizations. We show similarities between these methods and current performance differences. We focus also on the possibilities to force pragma directive parallelized GPU code to a specific vectorization. Finally we demonstrate the performance of these methods in a *complex structured* C++ implementation of the CG and the GMRES method with an algebraic multigrid as preconditioner.

Keywords: OpenMP 4.5, OpenACC, C++, Nvidia GPU, Multigrid

1 Introduction

Pragma directive parallelization has the goal to be platform independent for shared memory systems. Two applications, OpenACC and OpenMP 4.5, are currently available to be used on an Nvidia GPU via pragma directives. OpenACC is provided by PGI and is well prepared for a broad community. On the other hand, to use OpenMP 4.5 for an Nvidia GPU the user can face some critical problems already at the installation step. We summarized our installation in [8], and therefore let us remark in advance that the performance analysis in this paper is for sure limited to a small community with insight to GPU parallelization via pragma directives or via CUDA.
In this paper we focus on a parallel solver [7] for unstructured sparse matrices extracted from the advanced simulation tool for cardiovascular simulation CARP (Cardiac Arrhythmia Research Package [6]). We introduce our solver elements and a coalescing strategy for the matrix vector product (SELL-C-σ) in section 2. In section 3 we compare the used pragma clauses of OpenACC and OpenMP 4.5. Further we show in the sections 4 and 5 the parallelization of the scalar

product and the matrix vector product for a device.
In section 6 we show the implementation of a self *designed* coalescing strategy as SELL-C-σ with OpenACC and OpenMP, and in section 7 we show how the directive parallelization performs for the conjugate gradient and the GMRES method.
Finally we will focus on current implementation problems of OpenMP 4.5 for Nvidia GPUs in section 8.

2 Solver Elements

We performed an OpenACC parallelized solver for sparse matrix systems that achieves the performance of an elaborate CUDA implementation in [9]. This parallelization serves as a starting point for our OpenMP parallelization.
We consider a system of equations

$$A\,u = f \tag{1}$$

wherein A is a quadratic stiffness matrix. The matrix A is sparse, unstructured and it is assumed to be positive definite, i.e., $\langle Au, u\rangle > 0 \quad \forall u \notin \ker(A)$.
In the none symmetric case we use the generalized minimal residual (GMRES) method to solve equation (1) and in the symmetric case we apply the conjugate gradient (CG) method. Regarding details to CG and GMRES methods we refer the interested reader to [10].
Our implementation provides two preconditioners for Krylov methods (GMRES and CG), a Jacobi relaxation and an algebraic multigrid (AMG). Since we consider the parallelization of our preconditioners in detail, we want to summarize briefly the applied operators.[1]

2.1 Weighted Jacobi Relaxation

One simple preconditioner for (1) is the Jacobi method. The weighted Jacobi method computes the iteration

$$u^{(k+1)} = u^{(k)} + \omega\, D^{-1}\left(f - A\,u^{(k)}\right) \qquad k = 0, 1, 2, \ldots, \nu - 1 \tag{2}$$

wherein $D = diag(A)$, $u^{(0)} = 0$. If iteration (2) is used as solver for eqn. (1) then $\omega = 1$ is the best choice. Using iteration (2) as smoother in Alg. 1 requires an under relaxation, i.e., $0 < \omega < 1$. In a 1D textbook example the optimal smoothing parameter is $\omega = \frac{2}{3}$ [10, p. 431]. We use $\omega = 1$ for the parabolic sub problems in CARP where only the Jacobi iteration is used as preconditioner and we use $\omega = 0.9$ whenever this iteration is used as multigrid smoother for our 3D unstructured discretizations on the elliptic sub problems.

[1] One can find a very similar description in [9], in order to create a contiguously structured paper, we repeat parts of the description at this point.

2.2 Multigrid

Each Jacobi iteration (2) updates the solution vector u only locally and therefore the number of iterations for solving (1) increases with the number of unknowns. A constant number of iterations can be achieved by combining the Jacobi iterations on the original discretization with Jacobi iterations on various coarser discretizations (or operators) which is the basic idea of multigrid (MG).

We start to describe the components of the MG method on a two grid method, see also [2]. In order to distinguish between fine and coarse grid (discretization), we denote operators and elements on the coarse grid with index c and on the fine grid with index f, e.g., Ω_f contains all indices belonging to the fine grid and similarly Ω_c stores only those indices related to the coarse grid. As a consequence we re-write eqn. (1) as the *fine* equation

$$A_f\, u_f = f_f \qquad \text{or} \qquad \sum_{j \in \Omega_f} a_{ij,f}\, u_{j,f} = f_{i,f} \qquad \forall i \in \Omega_f \tag{3}$$

and corresponding for the two *grid method* the coarse equation as

$$A_c\, u_c = f_c \qquad \text{or} \qquad \sum_{j \in \Omega_c} a_{ij,\,c}\, u_{j,\,c} = f_{i,\,c} \qquad \forall i \in \Omega_c. \tag{4}$$

In order to derive a *correct* coarse set $\Omega_c := \Omega_f \backslash \Sigma_f$ we define the complementary indices $\Sigma_f \subsetneq \Omega_f$, $\Sigma_f \neq \emptyset$, i.e., those indices that belong exclusively to the fine grid. The intergrid transfer is performed by the *interpolation (prolongation)* operator I_c^f and the *restriction* operator I_f^c. Finally, the coarse matrix A_c is constructed via the *Galerkin approach*

$$A_c := I_f^c\, A_f\, I_c^f. \tag{5}$$

Using several iterations of (2) as smoother on the fine grid and a direct solver for the coarse system with operator (5) describes already the components for the two-grid method which is visualised in Fig. 1.

A recursive application of the two grid method, i.e., interpreting Ω_c as indices of the (intermediate) fine grid, leads to the *multigrid* method in Fig. 2. Furthermore, we present the basic multigrid V-cycle as algorithm 1. Applying multigrid directly to equation (1) is done via the call $\mathrm{MG}(1, f_1, A_1, u_1)$ on the finest discretization level with index 1 with some initial guess for u_1 (might be 0).

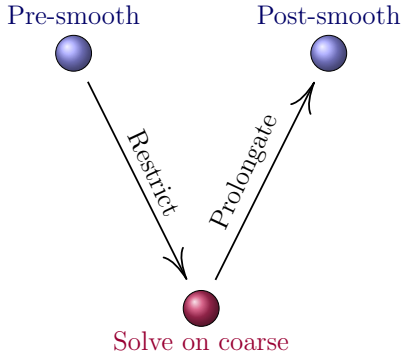

Fig. 1: V-Cycle

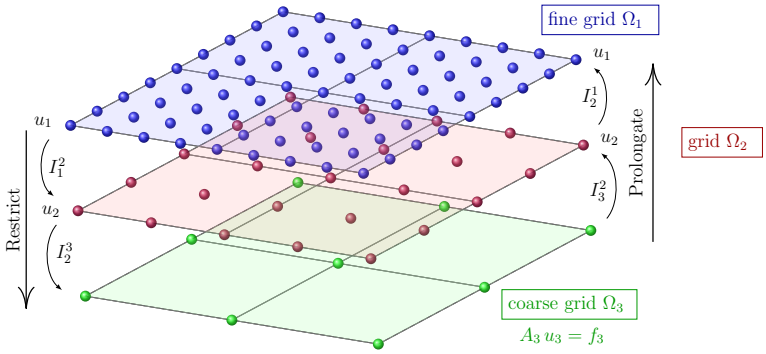

Fig. 2: Multigrid (3 levels)

function MG($\ell, f_\ell, A_\ell, u_\ell$)
if $l < max\_level$ **then**
  $u_\ell := u_\ell + \omega D_\ell^{-1}\,(f_\ell - A_\ell u_\ell)$ — ν pre-smoothing iterations
  $f_{\ell+1} := I_\ell^{\ell+1}(f_\ell - A_\ell\, u_\ell)$ — restrict defect
  $u_{\ell+1} := 0$ — initialize coarse grid correction
  MG($\ell + 1, f_{\ell+1}, A_{\ell+1}, u_{\ell+1}$) — apply MG to coarse grid correction system
  $u_\ell := u_\ell + I_{\ell+1}^\ell u_{\ell+1}$ — add interpolated correction
  $u_\ell := u_\ell + \omega D_\ell^{-1}\,(f_\ell - A_\ell u_\ell)$ — ν post-smoothing iterations
else
  $A_\ell\, u_\ell = f_\ell$ — solve the coarse system directly
end

Algorithm 1: Multigrid function for the V-cycle.

2.3 Algebraic Multigrid (AMG)

The classical geometric multigrid in the previous section assumes that the whole grid hierarchy is given a priori. This is not realistic in applications when 10^6 to 10^8 tetrahedrons are necessary to describe the computational domain. In these cases the coarser index sets $\Omega_{\ell+1}$, the intergrid transfer operators $I_{\ell+1}^\ell$ and $I_\ell^{\ell+1}$ and the coarse operators $A_{\ell+1}$ have to be generated before Alg. 1 can be applied. This setup step for AMG will be performed on the CPU and its parallelization is not in the scope of this paper. Regarding details of the AMG setup we forward the interested reader to [2, chap. 8] and one parallelization approach of the AMG setup can be found in [4].

2.4 Coalescing Strategies

Coalescing strategies are essential to use a GPU effectively. We have chosen the SELL-C-σ (ELLPACK) and reduced SELL-C-σ (reduced ELLPACK) strategies for our application.
Figure 3 shows the difference between CRS matrix data alignment and SELL-C-σ data alignment.

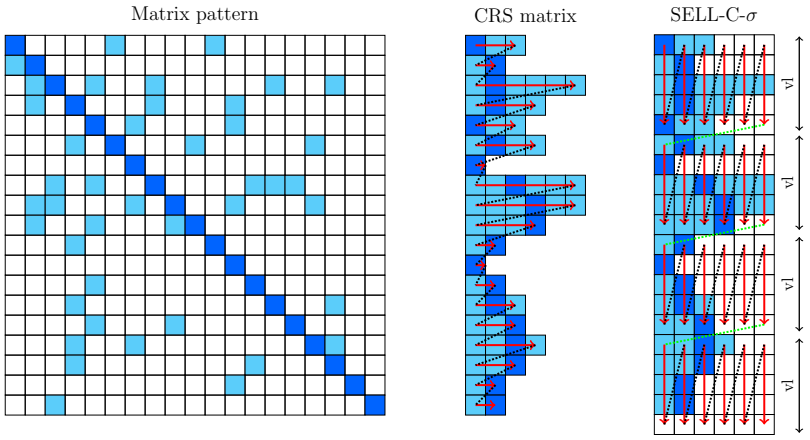

Fig. 3: SELL-C-σ data alignment

The CRS matrix consist 3 vectors for the none zero elements of the matrix:

- the column (`col` $= \{0, 5, 10, 0, 1, 1, 2, 4, 7, 13, 16, \dots\}$))
- the elements (`ele` consist the value of the matrix entry, corresponding to the column index)
- the displacement (`dsp` $= \{0, 3, 5, 11, \dots\}$) points to the index in `col` and `ele` of the first element of each row.

information.[2]

To get from a CRS matrix to a SELL-C-σ matrix structure, we expand the column and elements data such that the matrix has the same number of elements in every row, and we initialize the additional elements with a value of 0. Furthermore, we *reorder* the elements in the arrays such that the value-caching follows the marked arrows. Wherein `vl` denotes the vectorization length on the GPU.
The reduced SELL-C-σ strategy is very similar to the *normal* SELL-C-σ method. The only difference is that we bound the length of each matrix row to the maximal number of non zero elements per vectorized block of the matrix[3]. We refer the interested reader to [5] for more on *wide simd units* for the matrix vector product.

3 Parallelization via Pragma

Pragma directive parallelization has the huge advantage that code can be easily transferred to several different platforms. We consider the parallelization with OpenMP 4.5 and OpenACC (PGI 17.5). The conceptual idea is the same for both methods, even if the syntax has several differences (c.f. [1]). We want to summarize shortly some directives used in our further analysis.

[2] We show the corresponding values for our example matrix, and we use 0 as first index (C++ style).

[3] e.g. this means in figure 3 that the first and second `vl`-block would have a length of 6, the third a length of 3 and the forth a length of 4

During our code development we observed a huge dependency on the chosen directive clauses for both methods. The easiest way to execute a device method via pragma is

OpenACC	OpenMP
`#pragma acc kernels`	`#pragma omp target`

One can simply plugin those pragmas in front of the a kernel to create a device parallelization. We observe that OpenACC creates device kernels, which can be considered as an acceleration (compared to a CPU). We have shown in [9] how to improve an OpenACC parallelization. On the other hand, OpenMP is not prepared to parallize an arbitrary kernel effectively on a GPU with a straight forward approach.[4]
It is advisable for both methods to give *specific* directives to the compiler, how one want to parallize (respectively vectorize) the scope.

OpenACC	OpenMP
`#pragma acc parallel loop`	`#pragma omp target parallel for simd`
`#pragma acc loop seq`	`#pragma omp ordered`

Note that OpenACC allows nested parallelization via `#pragma acc parallel loop`, but OpenMP is not prepared to create a *complex* parallelization in a `teams` construct. The compiler output (`-Minfo`) of OpenACC shows that PGI tests the source code on *race-conditions*, respectively *loop-dependencies*[5]. For OpenMP it is required that the user understands the shared memory parallelization.
For both concepts is it advisable to define vectorization informations via directives to control the vectorization and threads of the kernels.

OpenACC	OpenMP
`vector_length(···)`	`thread_limit(···)`
`num_gangs(···)`	`num_teams(···)`

Note that we do not observe an advantage by using `num_threads(···)` directives for OpenMP (we analyse this behaviour in section 8).
One of the most important operations to use a device is the implemented data handling method. Both methods (OpenMP and OpenACC) provide keywords for the pragma directives to create the required kernel data for every device array.

[4] We observe a performance loss up to a factor of 100 compared to a CPU. c.f. [8]
[5] Note: One can create a race condition with OpenACC, if the user forces the compiler to a specific parallelization.

OpenACC	OpenMP
`copyin(···)`	`map(to: ···)`
`copyout(···)`	`map(from: ···)`
`copy(···)`	`map(tofrom: ···)`

Note, that this directives can be used to communicate between host and device, but since we use a *complex* C++ structure for our solver, we have to create deep copies on the device.

3.1 Deep Copy with Pragma Directives

We use a self written vector class `toolbox_vector` to handle our data arrays.

Listing 1.1: `toolbox_vector` class declaration

```
1 template <class T> class toolbox_vector {
2 public: // Inline functions
3 private:
4    T* _data;
5    size_t _size;
6 };
```

The data type `T` for a device array is in our application set as `int` or `double`. With pragma directive GPU parallelization one can communicate very easy between host and device. We show how we create a device array with pragma directives in listings 1.2 and 1.3..

Listing 1.2: Create Device Array with OpenACC

```
1 template<class T>
2 void toolbox_vector<T>::void todev() {
3 #pragma acc enter data pcopyin(_data[0:_size])
4 }
```

Listing 1.3: Create Device Array with OpenMP 4.5 for GPUs

```
1 template<class T>
2 void toolbox_vector<T>::void todev() {
3    const T *data = _data;
4 #pragma omp target enter data map(to:data[0:_size])
5 }
```

We want to take special attention on the private member `_data` of the `toolbox_vector` class. For OpenMP we have to define the pointer `data` in the same scope as the `#pragma omp ... data` directive (line 2 in listing 1.3). OpenACC on the contrary can resolve the address of the private member `_data`, but note that the dereference operation with OpenACC in C++ structures can lead to run-time errors in *complex* structured codes[6].

[6] Wherein we use PGI 17.5 compiler.

4 Scalar Product on GPUs with Pragma Directives

An elementary kernel for mathematical applications is the scalar product. We show how we parallize this arithmetics on the GPU.

Listing 1.4: Scalar Product with OpenACC

```
1  void scalar_product(const toolbox_vector<double> &_x,
2                      const toolbox_vector<double> &_y,
3                                                   double &_s) {
4      double s = 0.0;
5      const double *const __restrict x = _x.data();
6      const double *const __restrict y = _y.data();
7      const int xs = _x.size();
8
9  #pragma acc parallel loop vector_length(v1) pcopyin(x[0:xs],y[0:xs])
10     for(int i=0; i<xs; i++) {
11         s += x[i]*y[i];
12     }
13     _s = s;
14 }
```

Note, that OpenACC uses unified memory for scalar values. OpenMP on the contrary needs the map clause `map(tofrom: s)` to copy the value s between host and device.

Listing 1.5: Scalar Product with OpenMP 4.5 for GPUs

```
1  void scalar_product(const toolbox_vector<double> &_x,
2                      const toolbox_vector<double> &_y,
3                                                   double &_s) {
4      double s = 0.0;
5      const double *const __restrict x = _x.data();
6      const double *const __restrict y = _y.data();
7      const int xs = _x.size();
8
9  #pragma omp target teams distribute parallel for simd map(tofrom: s)
10                 reduction(+:s)
11     for(int i=0; i<xs; i++) {
12         s += x[i]*y[i];
13     }
14     _s = s;
15 }
```

4.1 Kernel Performance for the Scalar Product

Let us compare the two different parallelization strategies. For this purpose we calculate the result of a scalar product of two vectors with $862,515$ elements on a GeForce GTX 1060. We used NVIDIA Visual Profiler 8.0 to analyse our performance. Out of this, we found immediately that OpenACC creates 2 device kernels.[7] On the other hand, OpenMP 4.5 *only* one kernel. Out of 1000 function calls, we get the following table (note that we use the standard deviation for our calculation):

[7] A reader which is familiar with CUDA would realise that one kernel is required for the vectorized calculation and the second for the reduction step.

	OpenACC	OpenMP
Calculation time	129μs $\pm$ 31μs	701μs $\pm$ 115μs
# Copy data Host - Device	2002	6002
Registers (first/second kernel)	13 / 12	70
Grid Size (first/second kernel)	[6739,1,1] / [1,1,1]	[30,8,1]
Block Size (first/second kernel)	[128,1,1] / [256,1,1]	[30,1,1]

We observe for both methods a large runtime variation. This turns out of GPU intern operations (e.g. cooling, alignment). The error reduction of the measured time is not the aim of this paper, but we want to note that one can reduce the error by calling the function with a time delay.
A closer look to the performance table of NVIDIA Visual Profiler 8.0 shows that not only the better efficiency of an independent reduction step leads to an advantage of OpenACC, also *overhead operation* such as copy operations and kernel launch calls need much less time with OpenACC. We tried to create a vectorization by *ourselves* for OpenMP. Since we have only limited possibilities to control threads on the device (c.f. section 8) it is currently not possible to create an effective scalar product which can keep pace with CUDA implementations.
Furthermore we want to take special attention to the number of copy operations between host and device. We need to copy 2 vectors to the device for the scalar product, which are the only user defined data operations. OpenACC uses unified memory for scalar values, and copies automatically the result `s` to and from the device (2 times for each function call). This operations sums up to 2002 calls.
OpenMP creates 4 more copy operations (each one copies 8 byte to the device) for each function call[8]. We want to emphasize that the copy operations are an important overhead, but they do not dominate the overall calculation time in this comparison.

5 CRS Matrix Vector multiplication on GPUs with Pragma Directives

The matrix vector multiplication is one of the most important kernels for our solver. Therefore we want to show how we implement this kernel on the host and on the GPU with OpenACC and OpenMP 4.5. We use the CRS format for our solver, and therefore we get the matrix-vector multiplication on the host as

[8] If one want to find the reason for this, one can consider the assembler code of OpenMP. But this is not the aim of this paper.

Listing 1.6: CRS matrix-vector multiplicaion host

```
1  int i, s;
2  #pragma omp parallel for private(i, s) schedule(guided, 2)
3  for (i=0; i<sv; ++i) {
4     const int *const __restrict p_col = col+dsp[i];
5     const double *const __restrict p_ele = ele+dsp[i];
6     double v[i] = 0.0;
7     for (int j=0; j<dsp[i+1]-dsp[i]; ++j) {
8        v[i] += p_ele[j]*u[p_col[j]];
9     }
10 }
```

wherein `ele, dsp` and `col` are pointers to vectors of the CRS matrix and `sv` denotes the degrees of freedom.[9]
To calculate the same kernel on the GPU with OpenMP or OpenACC one can use pragma directives as in listings 1.7 and 1.8.

Listing 1.7: CRS matrix-vector multiplicaion with OpenACC

```
1  #pragma acc parallel loop independent vector_length(vl) pcopyin(...)
2  for (int i=0; i<sv; ++i) {
3     const int *const __restrict p_col = col+dsp[i];
4     const double *const __restrict p_ele = ele+dsp[i];
5     const int rsize = dsp[i + 1]-dsp[i];
6     double s = 0.0;
7  #pragma acc loop seq
8     for (int j=0; j<rsize; ++j) {
9        s += p_ele[j]*u[p_col[j]];
10    }
11    v[i] = s;
12 }
```

Listing 1.8: CRS matrix-vector multiplicaion with OpenMP 4.5 for GPUs

```
1  #pragma omp target teams distribute parallel for simd default(none)
2                  num_teams(nt)
3  for (int i=0; i<sv; ++i) {
4     const int *const __restrict p_col = col+dsp[i];
5     const double *const __restrict p_ele = ele+dsp[i];
6     const int rsize = dsp[i + 1]-dsp[i];
7     double s = 0.0;
8  #pragma omp reduction(+:s)
9     for (int j=0; j<rsize; ++j) {
10       s += p_ele[j]*u[p_col[j]];
11    }
12    v[i] = s;
13 }
```

The GPU parallelization is similar for OpenACC and OpenMP 4.5, one can simply plug in parallelization pragmas. Note that we use for our whole application keywords like `const` and `__restrict` in an exemplary extent. Furthermore, we took special attention on the choice of `num_teams(nt)`. The interested reader can find our development note online at [8] as Note 17.11 wherein we show the results of or implementation tests.

[9] We use for OpenACC the value `vl` to define the number of threads. We focus in section 8.1 on the thread definition for OpenMP 4.5.

5.1 Kernel Performance for the CRS Matrix Vector Product

In this section we compare the GPU kernels for the matrix vector multiplication. For this purpose we use a numerical test examples originating from the cardiovascular simulation package CARP [3]. In particular we use a stiffness matrix from an elliptic problem with $862,515$ degrees of freedom. We consider again 1000 function calls to compare the two different strategies.

	OpenACC	OpenMP
Calculation time	3.03 ms $\pm$ 0.05 ms	7.77 ms $\pm$ 0.12 ms
# Copy data Host - Device	11	6011
Registers / Thread	32	70
Grid Size	[6739,1,1]	[10,1,1]
Block Size	[128,1,1]	[32,28,1]

Similar as for the scalar product (c.f. section 4) we find that OpenMP requires much more copy operations, but this overhead is not the dominating factor ($\sim 7\%$ of the calculation time). We find also that OpenACC needs much less registers, and uses grids and blocks more efficient than OpenMP on the device.

6 SELL-C-σ Matrix Vector multiplication on GPUs with Pragma Directives

Similar as for the CRS matrix vector multiplication (c.f. 5) we show the parallelization with SELL-C-σ strategy.
The idea to parallelize the matrix-vector product with SELL-C-σ and CUDA is, that the compiler starts for every block a stride (in figure 3 marked with `vl`), and every thread gets one row of the matrix (`ml` is the maximal number of non zero elements per row). One can create kernels with the same properties with OpenACC by defining `parallel` scopes with `#pragma acc`, wherein we use the `seq` clause for the innermost loop, to prevent the compiler from creating a reduction at this stage.

Listing 1.9: SELL-C-σ matrix-vector multiplicaion with OpenACC

```
1  #pragma acc parallel loop pcopyin(...)
2  for (int i=0; i<sv; i+=vl) {
3     const int stride = i*ml;
4     const int k_max = (ml+i<sv)?(ml+i):sv;
5  #pragma acc loop independent
6     for (int k=i; k<k_max; ++k) {
7        double s = 0.0;
8  #pragma acc loop seq
9        for (int j=0; j<ml; ++j) {
10          const int index = stride+k-i+j*vl;
11          s += ele[index] * u[col[index]];
12       }
13       v[kk] = s;
```

```
14     }
15  }
```

We also used ELLPACK strategy with OpenMP 4.5, see listing 1.10.

Listing 1.10: SELL-C-σ matrix-vector multiplicaion with OpenMP 4.5 for GPUs

```
1  #pragma omp target teams distribute default(none) num_teams(10000)
2  for (int i=0; i<sv; i+=vl) {
3     const int stride = i*ml;
4     const int k_max = (ml+i<sv)?(ml+i):sv;
5  #pragma omp parallel for simd
6     for (int k=i; k<k_max; ++k) {
7        double s = 0.0;
8  #pragma omp reduction(+:s)
9        for (int j=0; j<ml; ++j) {
10          const int index = stride+k-i+j*vl;
11          s += ele[index] * u[col[index]];
12       }
13       v[kk] = s;
14    }
15 }
```

We also initialized the kernel such that we can take advantage of streams and threads with OpenMP 4.5. We will present the most effective parallelization we found[10].

6.1 Kernel Performance for the SELL-C-σ Matrix Vector Product

We use the same matrix as in section 5 with 862,515 degrees of freedom, and similar as in the last sections we calculate 1000 function calls resulting in the following table.

	OpenACC	OpenMP
Calculation time	2.23ms ± 0.03ms	26.77ms ± 0.4ms
# Copy data Host - Device	9	6009
Registers / Thread	32	70
Grid Size	[6739,1,1]	[30,1,1]
Block Size	[128,1,1]	[32,28,1]

We observe a huge difference between OpenACC and OpenMP. A self written coalescing strategy (as SELL-C-σ) is useful for OpenACC. The compiler can create a more efficient device kernel. On the other hand, for OpenMP we find a decreasing performance (compared to the CRS matrix vector product) on the device. OpenMP does not realize the self written vectorization, and creates a two dimensional device array (Block Size), which is definitely a disadvantage

[10] We want to refer the interested reader again to [8] wherein we show our test results with different parallelization strategies.

on the GPU. We observe (similar as for the scalar product in section 4) that we can not force the compiler to use a specific number of threads that corresponds to our vectorization (c.f. section 8).

7 Solver Performance with Pragma Directives

As mentioned in our solver description (section 2) we use the conjugate gradient method for symmetric problems of (1), the GMRES method is used in the non-symmetric case. We performed a well implemented OpenACC solver (as shown in [9]) which is competitive with a well performing CUDA implementation of the same solver. Now we extend our solver for OpenMP 4.5 on GPUs, and we present the performance differences between OpenACC and OpenMP for one solving step.

7.1 CG and GMRES with Jacobi Preconditioner

The Jacobi method is the simplest preconditioner available for both parallelized solve strategies (CG and GMRES). We use the same problem matrix as in the previous sections with 862515 degrees of freedom, unstructured, positive definite and symmetric and call the solve function 10 times.
Furthermore, the stopping criterion $\varepsilon < 10^{-5}$ is applied with respect to the relative error. We measured the following timings:

Solve time device CG with Jacobi preconditioner		
	OpenACC	OpenMP
CRS fromat	1.58s ± 0.01s	4.82s ± 0.08s
SELL-C-σ	1.33s ± 0.01s	11.27s ± 0.12s
reduced SELL-C-σ	0.94s ± 0.008s	9.38s ± 0.07s

Solve time device GMRES with Jacobi preconditioner		
	OpenACC	OpenMP
CRS fromat	8.73s ± 0.07s	44.62s ± 0.58s
SELL-C-σ	8.55s ± 0.08s	60.62s ± 3.27s
reduced SELL-C-σ	8.46s ± 0.08s	56.94s ± 1.53s

We observe that OpenACC performs well on the GPU for every solve strategy[11].

[11] The influence of SELL-C-σ is much lower for GMRES compared to the individual matrix kernels (c.f. sections 5 and 6). This happens because we need 198 GMRES iterations to solve the problem, and the matrix vector product becomes less important for more iterations.

On the other hand, we observe for OpenMP the same behaviour as for the matrix vector product (c.f. section 6). OpenMP does not realize the coalescing strategy, and therefore we observe a performance loss on the GPU for ELLPACK matrices. Moreover the kernel overhead (c.f. copy operations in sections 4, 5 and 6) becomes more important for a high iteration number.

7.2 CG and GMRES with AMG Preconditioner

Now we compare the performance for our solver with a *more complex* preconditioner, the algebraic multigrid. We keep the same settings for the solver as we did for the Jacobi preconditioner in 7.1. We find the following performances:

Solve time device CG with AMG preconditioner		
	OpenACC	OpenMP
CRS fromat	0.396s ± 0.011s	1.61s ± 0.17s
SELL-C-σ	0.322s ± 0.008s	2.93s ± 0.03s
reduced SELL-C-σ	0.221s ± 0.01s	2.29s ± 0.02s

Solve time device GMRES with AMG preconditioner		
	OpenACC	OpenMP
CRS fromat	0.380s ± 0.011s	1.19s ± 0.04s
SELL-C-σ	0.318s ± 0.009s	2.57s ± 0.03s
reduced SELL-C-σ	0.230s ± 0.011s	2.16s ± 0.02s

We find again that OpenACC is well prepared to be used on a Nvidia GPU, and the bounded possibilities to use OpenMP 4.5 lead to a performance loss.

7.3 Performance Analysis of one solve step

Since we have also an OpenMP parallelization on the host for our solver, we also measured our performance there. We run the code on an Intel(R) Core(TM)

i7-7700 CPU 3.60GHz (#4 Cores, #8 Threads) for our program.

Solve Time Host with AMG preconditioner		
	CG	GMRES
1 Thread	1.89s ± 0.20s	1.87s ± 0.18s
2 Thread	1.29s ± 0.02s	1.14s ± 0.05s
4 Thread	1.09s ± 0.02s	0.95s ± 0.01s
6 Thread	0.95s ± 0.01s	0.98s ± 0.01s
8 Thread	0.95s ± 0.02s	0.99s ± 0.02s

Unfortunately, we observe that our OpenMP 4.5 parallelization for a NVIDIA device is slower than a host parallelization. Nevertheless, it is possible to use the GPU with OpenMP pragmas.
We observe for OpenMP 4.5 on a Nvidia GPU a similar behaviour for the whole solver, as we found for the individual kernels (matrix vector product, scalar product). The performance with OpenMP is currently only in the range of a CPU core.

8 Runtime Problems with OpenMP on NVIDIA GPUs

8.1 No Explicit Definition of the Number of Threads on the Device

In order to optimize our algorithms, we considered the influence of different pragma clauses on the parallelization. In particular, since we observe a huge performance loss for SELL-C-σ strategies, we have been looking for optimisation potential. For this purpose we considered a very simple device kernel in listing 1.11 to read out the device properties of the parallelization.

Listing 1.11: Threads test with OpenMP 4.5 for GPUs

```
1  int size = nthreads;
2  double val[size];
3  memset(val, 0, sizeof(val));
4
5  int nt_device=0, nt_host=0;
6  #pragma omp target parallel for map(tofrom: nt_device, val[0:size])
7  for(int i=0; i<size; ++i){
8     nt_device = omp_get_num_threads();
9     val[i] = omp_get_thread_num();
10 }
11 #pragma omp parallel
12 {
13    nt_host = omp_get_num_threads();
14 }
```

The specific analysis of the thread behaviour gave some unexpected results. We observe that the kernel in listing 1.11 uses only 8 threads on the device. To raise the number of threads on the host, a user can use the clause

`num_threads(nthreads)`, the function `omp_set_num_threads(nthreads)` or the environment variable `export OMP_NUM_THREADS=nthreads`. All three methods influence **only** the number of threads on the host, one can not define the number of threads on the device with this commands.
One can influence the number of device threads with the clauses `num_teams(nteams)` and `thread_limit(nthreads)`, but unfortunately we can not define a fix number of threads on the device. We found, that OpenMP creates maximal 28 threads on the device (in case that we set `num_teams(nteams)` low). But unfortunately, this depends therefore very strong on the number of required *teams*. For a different amount of teams, we observe a different amount of used threads. In particular if we apply the AMG algorithm, with different kernel lengths then we observe a huge loss of performance (c.f. section 7.2)).

8.2 Overhead for Every Device Kernel

We saw for all our device routines that OpenMP creates a kernel overhead in form of copy operations (c.f. tables in sections 4.1, 5.1, 6.1). This overhead is not the *lion's share* of the runtime, nevertheless, as we have shown in [9], with OpenACC the users can get rid of this overhead by applying an exemplary use of `const` declarations, which is not possible for OpenMP 4.5.

9 Conclusion and Future Work

We have shown that OpenMP 4.5 is able to handle a *complex* C++ structure on an Nvidia device. But the liaison between host and device is yet not ready for an inexperienced user. During our development steps we saw also that a wrong application of OpenMP clauses can lead easily to a performance loss of 10000%[12].
The major problem for our application is for sure the limited possibilities to control the number of threads on the device, which has the clear consequence that a user of OpenMP can not create self implemented coalescing strategies for an effective usage of the device.
Nevertheless, we want to emphasize that OpenMP provides the possibility to be used for a GPU only since version 4.0. Therefore, we are looking hopefully into the future, and hope (in some sense we expect) for great progress in the development of OpenMP.
In a future work, we will compare the performance of OpenACC and OpenMP with respect to code portability.

References

1. Beyer, J.: OpenACC 2.0 versus OpenMP 4.0 device constructs (2017), http://on-demand.gputechconf.com/gtc/2013/webinar/gtc-express-webinar-openacc-vs-openmp.pdf, online; accessed 07-09-2017

[12] One can find corresponding measuring results in our Note 17.11 online at [8]

2. Briggs, W.L., Henson, V.E., McCormick, S.F.: A Multigrid Tutorial. SIAM, Philadelphia (2000)
3. CARP: Cardiac Arrhytmia Research Package (2017), https://carp.medunigraz.at/, online; accessed 05-09-2017
4. Haase, G., Kuhn, M., Reitzinger, S.: Parallel AMG on Distributed Memory Computers. SIAM SISC 24(2), 410–427 (2002), http://dx.doi.org/10.1137/S1064827501386237
5. Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Bishop, A.R.: A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. SIAM SISC 36(5), C401–C423 (2014)
6. Kunisch, K., Stollberger, R.: SFB Research Center, Mathematical Optimization and Applications in Biomedical Sciences (2017), http://imsc.uni-graz.at/mobis/, online; accessed 31-07-2017
7. Liebmann, M.: Parallel toolbox web page (2017), http://paralleltoolbox.sourceforge.net/, online; accessed 09-08-2017
8. Rosenberger, S.: Note 17-11; First Steps with OpenMP 4.5 on Ubuntu and Nvidia GPU's (2017), http://imsc.uni-graz.at/rosenberger/Arbeiten/Note17-11.pdf, online; accessed 15-09-2017
9. Rosenberger, S., Haase, G.: Effective OpenACC Parallelization for Sparse Matrix Problems. (2017), https://imsc.uni-graz.at/mobis/publications/SFB-Report-2017-008.pdf, SFB-Report No. 2017-008
10. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia (2003)