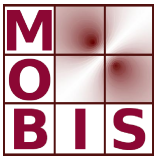
SpezialForschungsBereich F 32

Karl–Franzens Universität Graz
Technische Universität Graz
Medizinische Universität Graz



Effective OpenACC Parallelization for Sparse Matrix Problems

S. Rosenberger G. Haase

SFB-Report No. 2017-008 September 2017

A–8010 GRAZ, HEINRICHSTRASSE 36, AUSTRIA

Supported by the
Austrian Science Fund (FWF)

SFB sponsors:

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

Effective OpenACC Parallelization for Sparse Matrix Problems

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. Sparse matrices are a natural companion of finite element methods, therefore effective solvers for such problems are of wide interest. Several case studies for OpenACC show that pragma directive parallelization with OpenACC has advantages for code portability and programmer productivity compared to CUDA. But the same studies conclude that the performance with OpenACC is between 3 and 9 times lower than CUDA parallelized algorithms. In this paper we create an OpenACC parallelization for CG and GMRES algorithms that achieves the performance of a well implemented CUDA parallelized solver. We demonstrate the performance in an advanced simulation tool for cardiovascular simulations (CARP). Moreover we show the keystones a user follow to avoid performance leaks with OpenACC, and we show how coalescing strategies for sparse matrices as SELL-C-σ perform with OpenACC.

Keywords: OpenACC, C++, sparse matrix, multigrid, efficient parallelization

1 Introduction

Applications including partial differential equations (PDEs) use numerical schemes for discretization in time and in space. Assuming an implicit time discretization and a finite element discretization in space [12] we end up in each time step with a (linear) system of equations

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

wherein A is the quadratic stiffness matrix, u the discrete solution vector and f is the corresponding right hand side. The matrix A is sparse, unstructured and it is assumed to be positive definite, i.e., $\langle Au, u\rangle > 0 \quad \forall u \notin \ker(A)$.

We focus on the GPU-parallelization via OpenACC pragmas of iterative solvers for system (1) with symmetric or with non-symmetric matrix A. Therein we us a conjugate gradient (CG) or generalized minimal residual (GMRES) method with an algebraic multigrid (AMG) preconditioner or a weighted jacobi relaxion preconditioner as solving algorithm which are introduced in section 2.

There are plenty of solvers available for solving linear systems as (1) but we use the *Parallel Toolbox* [14] as reference code that provides Krylov iteration methods with Jacobi and multigrid preconditioning. This toolbox runs already with MPI on CPUs, and with CUDA on NVIDIA GPUs. We extended this toolbox with respect to pragma driven parallelization (CPU: OpenMP; GPU OpenACC) and we demonstrate in this paper how an OpenACC parallelization can achieve the same performance as CUDA. The detailed implementation issues and data designs for OpenACC are presented in section 3.

The numerical test examples in section 4 originate from the cardiovascular simulation package CARP(Cardiac Arrhythmia Research Package) that solves the bidomain equations in a physiologically accurate discretized heart for an anisotropic tissue by using algebraic multigrid (AMG) preconditioners for the elliptic PDE therein [16] and a Jacobi preconditioning for the parabolic part of the PDE system. Newer extensions of CARP incorporate also the elasticity equations which are also preconditioned via AMG [2].

Section 4.1 summarizes the impact of the various code improvements from the previous section on the GPU performance and the final section 5 presents some final remarks.

2 Linear Solver

The public available numerical package *Parallel Toolbox* [14] (written in C++) contains well performing CUDA and MPI implementation for linear solvers used in the CARP software [2]. To solve a symmetric positive definite system (1), a conjugate gradient method (CG) with *weighted Jacobi algorithm* or an *multigrid method* (MG) as preconditioner is used (c.f. 2.1 and 2.2). On the other hand, for a non symmetric system (1) a generalized minimal residual method (GMRES) with the same preconditioners as for the CG algorithm is used (which is an extension of [14] used in [4]). We will use the host version of this package as a starting point of our OpenACC parallelization.

Let us briefly present the implemented preconditioners because their OpenACC parallelization is explained in the next section. See [5] for more details also on the Krylov solvers.

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 $\omega = \frac{2}{3}$ [18, p. 431]. We use $\omega = 1$ for the parabolic

subproblems 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 subproblems.

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 [3]. 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 visualized 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).

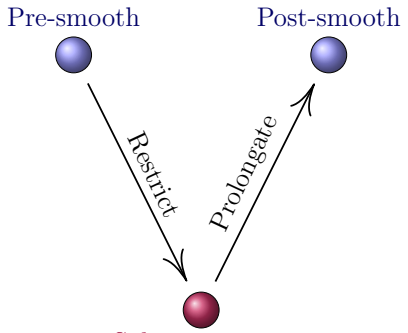


Fig. 1: V-Cycle

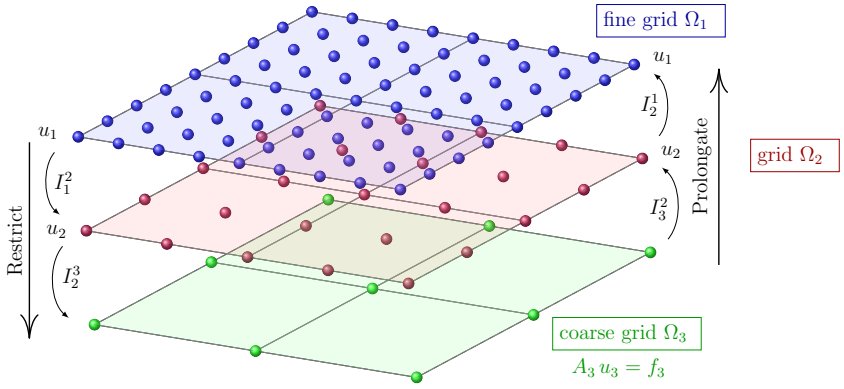


Fig. 2: Multigrid (3 levels)

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 $10^6 - 10^8$ 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 [3, chap. 8] and one parallelization approach of the AMG setup can be found in [7].

3 Step by Step to Success

We started with an existing host implementation of CG and GMRES methods [4] and we will show step by step how to achieve an efficient GPU-parallelization via OpenACC pragmas.

3.1 Create Device Solver

First of all we introduced methods for the communication between host and device with OpenACC. We use for all data arrays a self written vector-class `toolbox_vector`, and wrote the member functions for the communication between host and device. In listing 1.1 one can find one of those functions, in particular it shows how we create a device array.

Listing 1.1: Initialize device data with deep copy in C++

```
1  void todev() {
2  #pragma acc enter data pcopyin(_data[0:_size])
3  }
```

For all classes of our solver we create the device data when we call their constructors. This is useful to use the classes as stand alone methods on the device.[1] For example the *members* and *constructor* of the `matrix_vector` class is shown in listing 1.2.

Listing 1.2: Initialize class with device elements

```
1  class matrix_vector
2  {
3  private:
4     const toolbox_vector<int>& _dsp,
5     const toolbox_vector<int>& _col;
6     const toolbox_vector<double>& _ele;
7
8  public:
9     matrix_vector(...): _dsp(dsp), _col(col), _ele(ele)
10    {
11       _dsp.todev();
12       _col.todev();
13       _ele.todev();
14    }
15 };
```

Further we've created OpenACC device kernels by using `#pragma acc kernels` directives. For example, we show the matrix vector multiplication for CRS matrices in listing 1.3.

Listing 1.3: CRS matrix-vector parallelization with *kernels*

```
1  #pragma acc kernels loop independent pcopyin(...)
2  for (int i=0; i<end; i++) {
3     int *p_col = col+dsp[i];
4     double *p_ele = ele+dsp[i];
5     double s = 0.0;
6     for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
7        s += p_ele[j]*u[p_col[j]];
8     }
9     v[i] = s;
10 }
```

For the first implementation, we copied before all device kernels the data to the device, and after the calculation we update the host. This is obviously not fast, but an easy way to check for correctness.
Step 1 was already a working device version for our solver.

[1] Note at this point, that we do not use the pragma directive *#pragma acc routine* at all.

3.2 Data Handling

Our fist improvement step is the reduction of the (user) communication between host and device to a working minimum version of the source code. At this point we want particularly note the practical usefulness to handle the data via member functions as one can see listing 1.1.

3.3 Improve Pragma Directives

To choose the `#pragma acc kernels` directive is seductive. One simply plug in some pragmas and has a device version for all routines in the following scope. Even if it works, it is not efficient to use the `kernels` directive for a *complex device kernel* like the matrix vector product (wherein we've used PGI 17.5 compiler). Therefore, we replaced all `kernels` directives with `parallel` directives and define the vectorization length via `vector_length(128)`. For example, the matrix-vector kernel (listing 1.3) is rewritten to listing 1.4.

Listing 1.4: CRS matrix-vector parallelization with *parallel*

```
1  #pragma acc parallel loop vector_length(v1) pcopyin(...)
2  for (int i=0; i<end; i++) {
3     int *p_col = col+dsp[i];
4     double *p_ele = ele+dsp[i];
5     double s = 0.0;
6  #pragma acc loop seq
7     for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
8        s += p_ele[j]*u[p_col[j]];
9     }
10    v[i] = s;
11 }
```

This improvement is really powerful. To understand that behaviour, we consider the *-Minfo* compiling output of the two kernels listing 1.3 and 1.4.

Listing 1.5: Minfo for CRS matrix-vector parallelization with *kernels*

```
1  CRSmatrixKernels<int, double>::matrix_vector(...) const:
2     20, Intensity = 0.0
3     Generating copyin(col[:ele_size],dsp[:dsp_size],
4        ele[:ele_size],v[:v_size],u[:v_size])
5     23, Intensity = ((stop-start)*(csize*2))/((stop-start)+
6        ((stop-start)+((stop-start)+(csize+csize))))
7     Loop is parallelizable
8     Accelerator kernel generated
9     Generating Tesla code
10    23, #pragma acc loop gang /* blockIdx.x */
11    29, #pragma acc loop vector(128) /* threadIdx.x */
12    32, Generating implicit reduction(+:s)
13    29, Intensity = 1.00
14    Loop is parallelizable
```

Listing 1.6: Minfo for CRS matrix-vector parallelization with *parallel*

```
1  CRSmatrixParallel<int, double>::matrix_vector(...) const:
2     20, Intensity = 0.0
3     Generating copyin(col[:ele_size],dsp[:dsp_size],
4        ele[:ele_size],v[:v_size],u[:v_size])
5     Accelerator kernel generated
```

```
6     Generating Tesla code
7     23, #pragma acc loop gang, vector(128) /* blockIdx.x
8        threadIdx.x */
9     30, #pragma acc loop seq
10    23, Intensity = ((stop-start)*(csize*2))/((stop-start)+
11       ((stop-start)+((stop-start)+(csize+csize))))
12    30, Intensity = 1.00
```

One can see, that the *auto parallelization* with `#pragma acc kernels` works for this routine (listing 1.5, lines 7 and 14). But the auto vectorization is *suboptimal* initialized. The compiler creates a vectorization for the inner loop (listing 1.3, line 6 and listing 1.5 line 11), but the efficiency is much higher for a vectorization on the outer loop (listing 1.4 line 1 and listing 1.6 line 7).
At this point we want to take a special attention to the lines 3-4 in listing 1.5 and 1.6. In both cases we use the same size of elements (*ele_size* and *v_size*), as often as the elements have the same length, to initialize the *device-kernels* (`#pragma acc ... pcopyin(...)`). This seams to be obvious, but note that we observe a performance loss if we use independent length for every element size.

3.4 Implement Coalescing Strategy

At this stage we compared the existing CUDA implementation with our OpenACC parallelization. We didn't reach 50% of our CUDA performance, therefore we improved our data-layout for CRS matrices.
We've chosen SELL-C-σ (ELLPACK) coalescing strategy on the device [9]. This means, we reorder the matrix elements in the way that the storage format follows the sketch of figure 3 [2]. In this alignment, we expand the numbers of elements such that every row has the same number as the maximum number of non zero elements per row. The values of the *new* elements is set to 0, and the corresponding column index (which is required for the algorithm) is set to 1. Furthermore, we save the inverse diagonal elements (dark blue), which are required for the weighted Jacobi relaxation 2.1, additional to the standard CRS matrix elements.
The matrix-vector product is rewritten with this method to listing 1.7.

Listing 1.7: CRS matrix-vector for SELL-C-σ (ELLPACK) strategy

```
1  #pragma acc parallel loop vector_length(vl) pcopyin(...)
2  for (int i=0; i<sv; i+=vl) {
3     int idx = i*ml;
4     int k_max=vl+i<sv?vl+i:sv;
5  #pragma acc loop
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          s += ele[idx+k-i+j*vl]*u[col[idx+k-i+j*vl]];
11       }
12       v[k] = s;
13    }
14 }
```

[2] Figure 3 shows a sketch, naturally we choose the vectorization length as a power of 2.

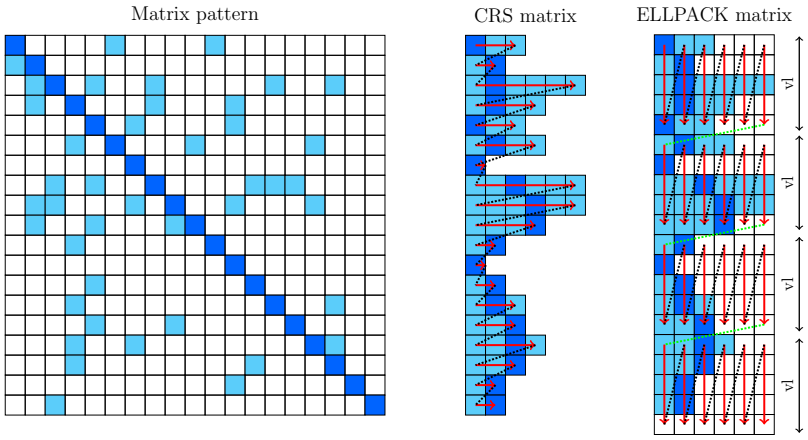


Fig. 3: ELLPACK data alignment

Wherein ml denotes the maximum number of entries per row, vl the vectorization length and sv the degrees of freedom. OpenACC starts for every i (line 1-2) a stride, and for every k (line 5-6) a new thread. Note, for an efficient implementation of the device kernel in listing 1.7 is it not advisable to use `std::max` to calculate the maximum (line 4, wherein we've used PGI 17.1 compiler).
As in step 3.3 we consider the compiling output with *-Minfo*.

Listing 1.8: Minfo for ELLPACK matrix-vector parallelization

```
1  ELLPACK<int, double>::matrix_vector(...) const:
2     60, Intensity = 0.0
3     Generating copyin(col[:ele_size],ele[:ele_size]
4        ,v[:v_size],u[:v_size])
5     Accelerator kernel generated
6     Generating Tesla code
7     63, #pragma acc loop gang /* blockIdx.x */
8     69, #pragma acc loop vector(128) /* threadIdx.x */
9     72, #pragma acc loop seq
10    63, Intensity = (((v_size+127)/128)*((k_max-i)
11       *(ml*2)))/(k_max-i)
12    69, Intensity = ((k_max-i)*(ml*2))/(k_max-i)
13    Loop is parallelizable
14    72, Intensity = 2.00
```

We can see, that OpenACC creates a *correct vectorization point* for the second loop in listing 1.7 (line 8 in listing 1.8).

With this improvement, we beat the 50% benchmark for the CUDA implementation.

3.5 Reduce the OpenACC Device Pointer Requests

To improve the performance further, we needed to redesign the whole implementation. The performance analysis showed some *curious* data behaviour, wherein we used Nvidia Visual Profiler (NVVP 8.0) for our analysis. Figure 4 shows a part of MG V-cycle, in particular we observed (marked line) sequences of copy operations between host and device, which should not occur (respectively of which we know, there is no need for those operations). To understand the problem we consider our implementation of the MG V-Cycle:

Listing 1.9: MG algorithm classic

```
1  void MG(int k) {
2     int l = k++;
3     if (l < _level) {
4        prolongation Pro(_rcnt[l], _rcol[l]);
5        restriction Rest(_rcnt[l], _rcol[l]);
6        jacobi Smooth(_acnt[l], _adsp[l], _acol[l], _aele[l],
7           _adia[l], _v[l], _omega, _com[l]);
8
9        Smooth(_f[l], _u[l]);
10       residual(_f[l], _u[l], _r[l]);
11       Rest(_r[l], _f[l + 1]);
12
13       MG(k);
14
15       Pro(_u[l + 1], _u[l]);
16       Smooth(_f[l], _u[l]);
17    } else {
18       SolveOnCoarsed->solve(_f[l], _u[l]);
19    }
20 }
```

This implementation is very flexible with respect to the CRS matrix elements. For every *grid-level l* we initialise the required Operators (lines 4-7). Note that every array (e.g. `_rcnt[l], _rcol[l]`) is created on the device during setup, therefore the call for the constructor do not copy data to the device (c.f. red constructors in listing 1.9 and constructor initialization in listing 1.2). Furthermore, the constructors assign only existing elements to the class (*set pointer to existing element*), there is no arithmetic routine in the constructors.

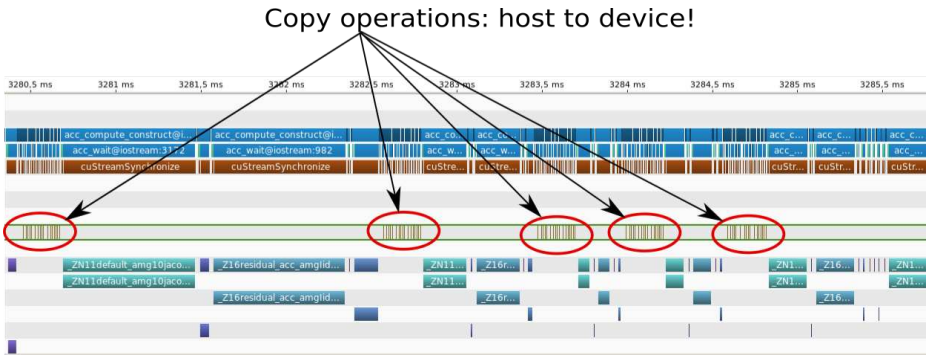


Fig. 4: NVVP 8.0 output (on Nvidia Pascal)

We followed the problem to the source, and found that OpenACC creates a pointer request for the device. The main problem which force those requests is, that OpenACC needs to check for existing data on the device.[3] To work around this problem, one can initialise as much as possible operators for the algorithm as **constant** members of the class. Note at this point, that additional to the members, we took special attention to an exemplary use of keywords like `const` and `__restrict`.

[3] This occurs also after the constructor call!

Listing 1.10: MG algorithm without initialization

```
1  void MG(int k)
2  {
3     int l = k++;
4     if (l<_level) {
5        _S[l]->PreSmooth(_f[l], _u[l]);
6        _A[l]->Residual( _f[l], _u[l], _r[l]);
7        _I[l]->Restrict(_r[l], _f[l + 1]);
8
9        MG(k);
10
11       _I[l]->Prolongate(_u[l + 1], _u[l]);
12       _S[l]->PostSmooth(_f[l], _u[l]);
13    }
14    else {
15       SolveOnCoarsed->solve(_f[l], _u[l]);
16    }
17 }
```

The redesigned code (listing 1.10) does not show pointer requests during the MG algorithm.

3.6 Improve Coalescing Strategy

The last step, to get an efficient implementation, is to improve the coalescing strategy. For this purpose, we implement the reduced SELL-C-σ (reduced ELLPACK) strategy. The difference to the ELLPACK format is, that we reduce the

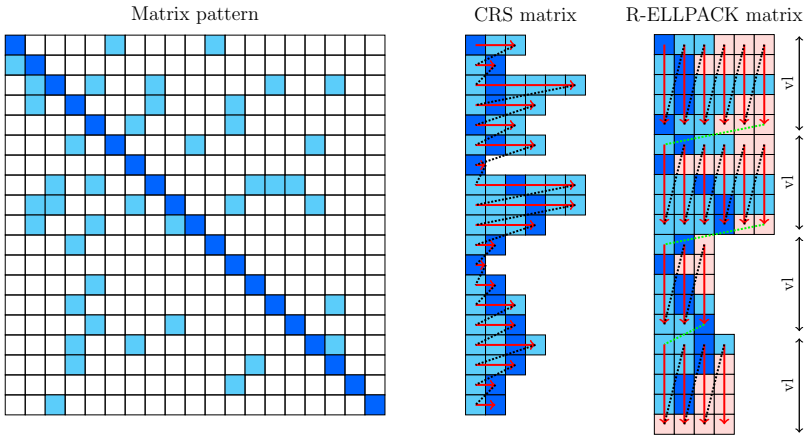


Fig. 5: Reduced ELLPACK data alignment

block length for every stride to the maximum number of non zero elements of each block, and we calculate only those elements which are really required (blue elements in figure 5) for the arithmetic. Furthermore, we assigned the *surplus* elements (red fields in figure 5) with the last *used* column index (last blue column entry per row). We also considered the parallelization information with *-Minfo* for this coalescing strategy. One get the same information as in listing 1.8, therefore we conclude that this improvement is independent to OpenACC, it is simply a better data-layout.

Finally, with this data alignment we can even beat the performance of our CUDA implementation.[4]

4 Performance Analysis

Needless to say, but we have been interested to compare our final implementation on different systems and different parallelization methods. For this purpose we distinguish between 5 different implementation strategies.

1. **Trainee:** We use for the device routines the pragma directives `kernels`, like listing 1.3.
2. **Standard:** Implement *simple* pragma `parallel` directives in the source code, like listing 1.4.
3. **ELLPACK:** Use the SELL-C-σ strategy, like listing 1.7.
4. **R-ELLPACK-A:** Use the reduced SELL-C-σ strategy, of step 3.6, **but** we use the padded entries (red fields, which are initialised with 0) in figure 5.
5. **R-ELLPACK-B:** Use the reduced SELL-C-σ strategy, of step 3.6.

4.1 Numerical Results

We mentioned in section 3 that we considered frequently our OpenACC performance during our code development. Since we have presented a grown process to improve our parallelization, we can only give a reconstruction (out of our notes) of the performance improvement to every step (except on the final implementation, step 3.6). We've tested our implementation during the development on an Intel(R) Core(TM) i7-7700 CPU 3.60GHz and a GeForce GTX 1060. Note, we do not use SELL-C-σ reordering for the reference calculation on the host. Local tests have shown, this would lead to a performance leak. Table 1 shows a time measurement for an elliptic problem with 862515 degrees of freedom, wherein we measure the solve time at runtime of the simulation tool CARP.
The very first device version of OpenACC was indeed slower than one CPU core. But this is not surprising, as mentioned in 3.1, we frequently copied data between host and device. Step 3.2 would correspond to the *Trainee* implementation with `#pragma acc kernels` directives. Step 3.3 would correspond to the *Standard matrix-vector* implementation, and we can see the importance to force the compiler to the correct vectorization (compared to step 3.2). The coalescing strategy SELL-C-σ (ELLPACK) reached the 50% performance of our CUDA solver, as described in step 3.4.
The most difficult part to achieve a good OpenACC performance is for sure step 3.5. We observed an acceleration of $\sim$ 20 - 30%, which depends on the execution time per kernel (c.f. parallelization steps 3.2, 3.3, 3.4 and 3.6). Moreover note that the *code-improvement* step 3.5 is also dependent on the hardware, it becomes more important on newer systems (which is independent of the chosen parallelization).

[4] Note, our CUDA implementation uses a similar coalescing strategy!

Method	Solve-time	stdev (10 runs)	multiplicity
1 CPU	198.72s		1.00×
CUDA	21.90s	±0,0099s	9.07×
OpenACC			
Step 3.1:	∼ 330.00s	-	∼ 0.60×
Step 3.2:	∼ 130.00s	-	∼ 1.53×
Step 3.3:	∼ 56.00s	-	∼ 3.55×
Step 3.4:	∼ 42.00s	-	∼ 4.73×
Step 3.5:	∼ 31.00s	-	∼ 6.4×
Step 3.6:	21.19s	±0,0107s	9.38×

Table 1: OpenACC performance during developing; CG with AMG preconditioner for an elliptic problem.

The final parallelization (step 3.6) is the *R-ELLPACK-B* format, and we can achieve the CUDA performance.
Note that we use for the rest of the analysis section 4 our well prepared source code which turns out of development step 3.6.

4.2 Kernel Performance for Jacobi Preconditioner

We have used a Tesla P100-PCIE-16GB device from the ARA-Cluster [8] to test the 5 different parallelization strategies introduced at the beginning of this section. We measured for our well performing solver of step 3.6 the calculation-

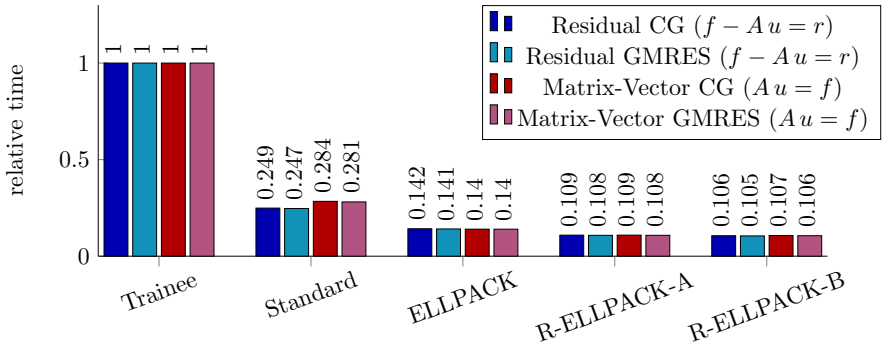


Fig. 6: Kernel performance for Jacobi preconditioner

time for the most important kernels of the jacobi preconditioner (c.f. section 2.1) in the GMRES and CG methods. Note at this point that all error bars are below 0.06% (stdev for 10 runs) of the measuring result and therefore we do give further notes on the occurring errors.

We use relative time scheme to visualise our performance (wherein we use as a reference time CG = 5.4287ms, GMRES = 5.4711ms for the residual and CG = 5.4725ms, GMRES = 5.5134ms for the matrix-vector product). We find that the different solve strategies (CG or GMRES) do not influence the kernel

performance (as it is expected). Moreover, we find that it is worth to implement *self designed* coalescing strategies.

Take special attention on the *Standard* parallelization (which uses `#pragma acc parallel` directives). We observe that OpenACC can create more efficient kernels for the residual than for the matrix vector product if one gives the compiler some space to optimize the device algorithm (which means it is a device kernel wherein the compiler can see some optimizable structure). We do not observe this behaviour if we use the compiler option *-O0* (usually we use *-O3*).

4.3 Kernel Performance for AMG Preconditioner

The elliptic PDE of the bidomain equation is the more challenging part for our parallel solver. Therefore we want to take special attention on the GPU-parallelization for the AMG preconditioner. Similar as in 4.1 we observe an

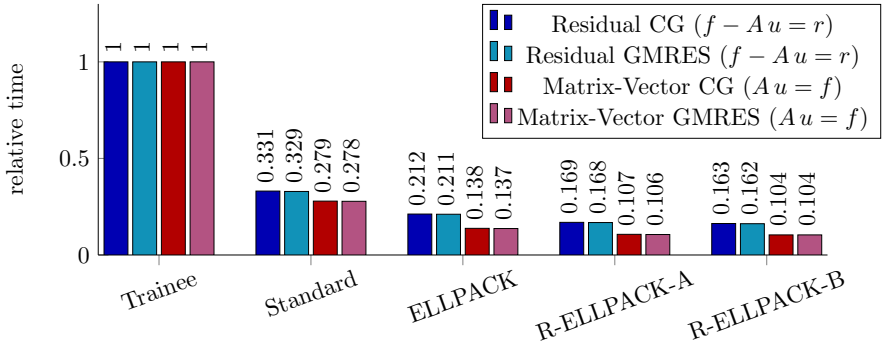


Fig. 7: Kernel performance in for AMG preconditioner

improvement for every type of parallelization strategy. Figure 7 shows the runtime for the most important kernels of the inverse method (CG or GMRES), wherein we use for each kernel the *trainee* parallelization as a reference time to show the relative time development (the reference time for *Trainee* is for Residual 1335.8[ms] and for the Matrix-Vector product 5575.6[ms], wherein we use again the Tesla P100-PCIE-16GB device from the ARA-Cluster [8]). We did the same analysis for GMRES as for the CG algorithm, and we find a very similar result (c.f. figure 7).

We use the calculation of the residual during the solve algorithm of AMG. Therefore we need this kernel for all different grid levels Ω_c (c.f. section 2.2) and as a consequence the kernels varies in data-length during the calculation. On the other hand, the matrix vector product is only used for the fine equation (1). Therefore we observe (as to be expected) that the ratio between kernel-call and data-length improves for larger systems.

Furthermore, we considered the impact of our parallelization on the intergrid transfer operators $I_{\ell+1}^{\ell}$ and $I_{\ell}^{\ell+1}$ (c.f. section 2.3). We use the *injection* as restriction operator and $I_{\ell+1}^{\ell} := (I_{\ell}^{\ell+1})^T$ as prolongation operator (c.f. [3, p. 35]). We choose again the relative time scheme to visualise our performance

(reference time for *Trainee* is for restriction 244.62[μs], wherein we show only the CG method since the result for GMRES leads to the same conclusion). We observe an improvement for the restriction as expected, since the restriction operation is similar to the calculation of the matrix-vector product. On the other hand, the prolongation is independent to the data structure. This happens because every thread in the prolongation gets only one value of the fine grid and therefore we cannot benefit from a better data layout. Note that the calculation of the intergrid operations needs only integer arrays (the cast and calculation operations between integer and doubles can be done local per node), therefore we see less impact for the SELL-C-σ strategies on the restriction operation than we see it for the matrix-vector product (c.f. figure 8). Moreover, please note that the improvement between ELLPACK and R-ELLPACK-A was less than 1% of the overall calculation time, and therefore we did not implement the R-ELLPACK-B strategy for the intergrid operators.

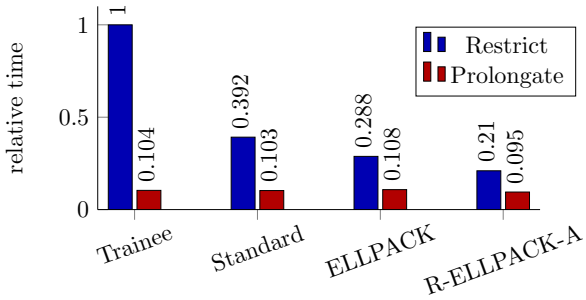


Fig. 8: Kernel performance intergrid transfer in AMG

4.4 Performance for Vector-length restrictions

One can use the pragma directive `vector_length(vl)` to force the compiler to set the vectorization length to a fixed number. We have tested our solver with different vectorization lengths. For this purpose we have calculated in the first

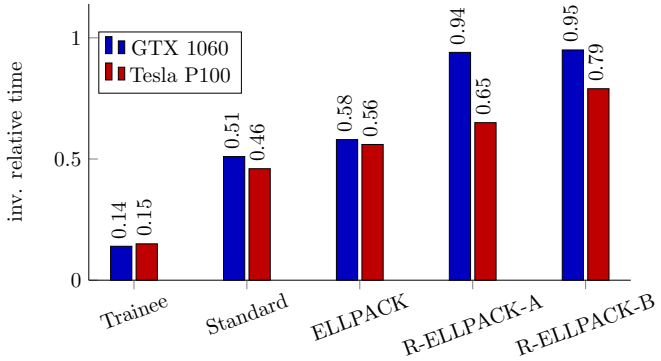


Fig. 9: Residual **without** `vector_length` directives.

place our code without `vector_length()` directives (e.g. in listing 1.4 line 1 becomes to `#pragma acc parallel loop pcopyin(...)`). Note that we use the

minimal measured time as reference for a better visualization of our results (i.e. inv. relative time means $t_{\text{min}}/t_{\text{measured}}$, and as a consequence larger values have a better performance). Furthermore, we have tested several different vector lengths vl wherein we considered $vl \in \{16, 32, 64, 128, 256, 512\}$. We do not present every detail of our search at this point (the interested reader can find the summarised results of this search online at http://imsc.uni-graz.at/rosenberger/thesis.php in section *Scientific Results*). Figure 10 shows the best performance for the residual

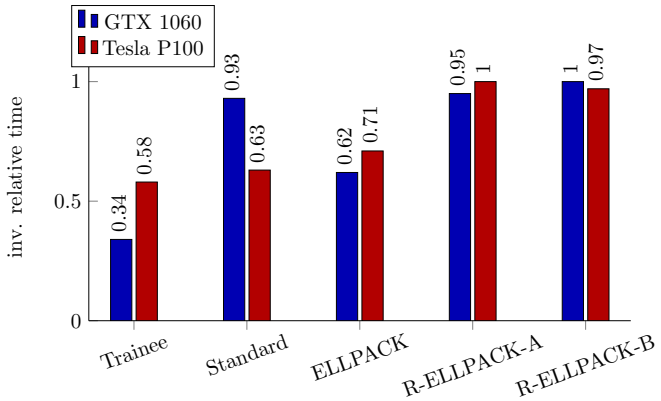


Fig. 10: Residual **with** `vector_length` directives.

kernel we have reached for the different implemented strategies. We found that we have the best performance if we define the vectorization length. But note that we are also interested to create a platform independent source code. Therefore the implementation without vector length directives is of special interest. Here we see that a self implemented coalescing strategy is very useful (c.f. figure 9). We achieved the best performance on GTX 1060 for the R-ELLPACK-B coalescing strategy wherein we defined the vector length $vl = 512$ (reference time 466.15μs). On the other hand on Tesla P100 we found the fastest setting for the R-ELLPACK-A coalescing strategy wherein we defined the vector length $vl = 16$ (reference time 206.87μs).

Furthermore, we considered the overall solve time for CG method with AMG-

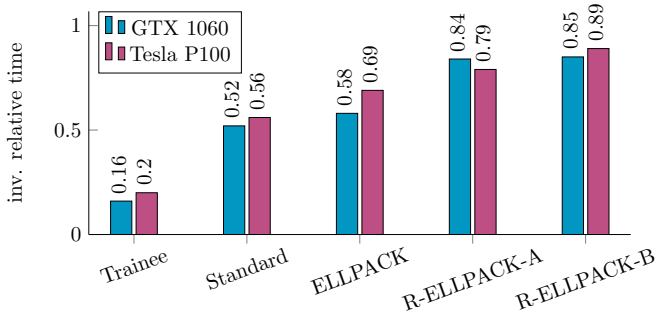


Fig. 11: Solve time **without** `vector_length` directives.

preconditioner. We performe exactly one solution step for an elliptic problem

with 862515 degrees of freedom and a CG-stopping criteria $\varepsilon < 10^{-10}$. We get reference times for GTX 1060 as 396ms and for Tesla P100 as 213ms (best performance for the solver setup).

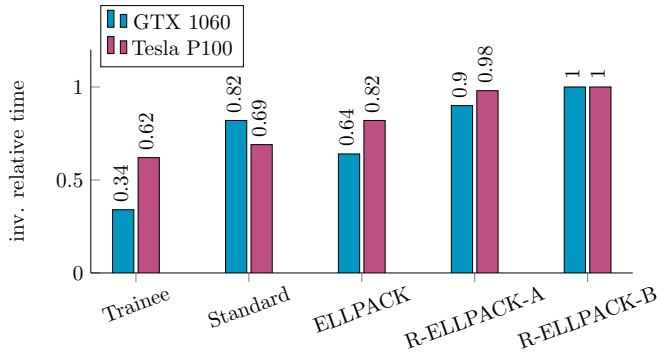


Fig. 12: Solve time **with** `vector_length` directives.

The overall solution time behaves similar as the performance of the most important kernel of our solver in figures 9 and 10 ($\sim$ 70% of the calculation is the residual calculation). We can see that a self implemented coalescing strategy is very useful to get well performed independent source code. Nevertheless is it particularly noteworthy that one can reach up to 80% of the performance of a well performed CUDA code (c.f. table 1) with a source code which is simply prepared for shared memory parallelization (e.g. OpenMP for a host) by plug in `#pragma acc` directives (c.f. figure 12 Standard).

4.5 Kernel Performance for Vector-length and Gang Restrictions

As mentioned in several introductions to OpenACC (e.g. [11] or [6]), one can improve the performance of OpenACC kernels via `num_gang` and `vector_length` clauses. We have tested the performance of our most important kernel in the multigrid preconditioner, the residual calculation. For this purpose, we included those clauses for our `#pragma acc` directives (e.g. in

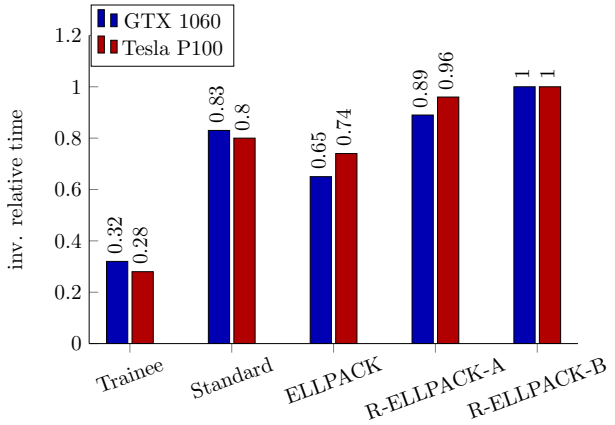


Fig. 13: Residual with `num_gang` and `vector_length` directives.

listing 1.4 line 1 becomes to `#pragma acc parallel loop num_gangs(ng) vector_length(vl) pcopyin(...)`). Note, we also paid attention on the compiler information (*-Minfo*) for correct vectorization/gangs initialization points (c.f. listing 1.8). To analyse the influence of those directives we did a *grid-search* for the number of gangs ng and the vector length vl, wherein we considered $vl, ng \in \{16, 32, 64, 128, 256, 512\}$. We found the best performance for R-ELLPACK-B strategy, with a reference time of 501.79μs on GTX 1060 and 226.26μs on Tesla P100. We do not present every detail of our search at this point (the interested reader can find the summarized results of this search online at http://imsc.uni-graz.at/rosenberger/thesis.php in section *Scientific Results*).

1. **Trainee:** We found the best performance for
   - **GTX:** $vl = 16$, $ng = 512$
   - **P100:** $vl = 16$, $ng = 512$.

   Overall we can conclude that it advisable to bound the vector length for the `#pragma acc kernels` directives, but one should not limit the number of gangs.

2. **Standard:** We found the best performance for
   - **GTX:** $vl = 256$, $ng = 16$
   - **P100:** $vl = 32$, $ng = 512$.

   Overall we can conclude that it is advisable to bound exact one of the values of `vector_length` and `num_gang`.

3. **ELLPACK:** We found the best performance for
   - **GTX:** $vl = 512$, $ng = 512$
   - **P100:** $vl = 256$, $ng = 256$.

   Overall we can conclude that for self implemented coalescing strategies with ELLPACK it is better set large values for the gang and vector clauses.

4. **R-ELLPACK-A:** We found the best performance for
   - **GTX:** $vl = 256$, $ng = 512$
   - **P100:** $vl = 64$, $ng = 512$.

   Overall we can conclude that for self implemented coalescing strategies with R-ELLPACK-A it is better set large values for the gang and vector clauses.

5. **R-ELLPACK-B:** We found the best performance for
   - **GTX:** $vl = 512$, $ng = 256$
   - **P100:** $vl = 128$, $ng = 256$.

   Overall we can conclude that for self implemented coalescing strategies with R-ELLPACK-B it is better set large values for the gang and vector clauses.

The influence of `num_gang` and `vector_length` is very similar on the different SELL-C-σ strategies, which shows again that OpenACC has no special influence on the self designed coalescing method.

4.6 Memory usage with OpenACC

The final point for our analysis is that we consider the required memory on the device for our parallelization. We want to summarize the memory usage for a test example with 862515 degrees of freedom on a GeForce GTX 1060. Figure 14 shows the used device memory during run-time. *Trainee* and *Standard* are simply different `#pragma` directives and we would not expect a different memory usage. The ELLPACK format uses 92% more memory on the device than the regular CRS format, nevertheless we observe a significant acceleration for our solver (c.f. figure 3 and table 1). Additional, we observe for our problem (1) that the reduced ELLPACK data layout has huge influence on the used memory. We need 32% less memory for the reduced ELLPACK format (figure 5) than for the normal ELLPACK format (figure 3), this is obvious problem dependent. Similar as in our improvement step 3.6, we cannot conclude to any *special* connection between OpenACC and the data layout. It is simply a better data-strategy.

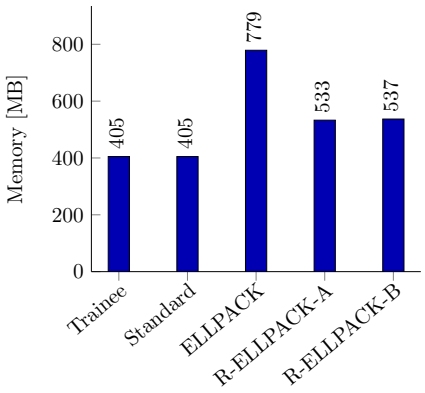


Fig. 14: GPU memory usage

On the other hand, for *R-ELLPACK-B* format we need additionally to the column, element and displacement information the number of non zero elements per row on the device. We need more memory on the device, but we can conclude that OpenACC creates efficient device kernels and uses free resources if possible, since we observe an improvement for our solver.
At this point we want to note, that our improvement step 3.5 had also huge influence on the memory usage. Since we used a different solver structure we could reduce the used device memory to 60% of the original space.

5 Conclusion and Future Work

As we have shown, one can create a very effective GPU implementation with OpenACC, as long as the user avoids any kind of sloppiness. One has to work as we all learn it in good programming courses (and as most of us forget about it in practice). This means, if you know that a parameter will not change than define it constant. If you know that a pointer is *restricted* than use the keyword *__restrict*. If you know a method cannot *change the class members* than define this method constant. If you can construct an operator before the scope then do not wait to call the constructor, do it outside the scope.

In several papers and thesis ([13], [19], [15] and many others we do not list here) the authors concluded that the performance with OpenACC cannot keep up with CUDA implementations, except on some algorithmic test examples. For sure, we spend a lot of time to find a way to improve a *complex* C++ source

code such that OpenACC can achieve a good performance. But nevertheless we can conclude that OpenACC is competitive to well performed CUDA implementations also for advanced simulation tools like CARP.

In a future work, we will present how we expand our source code to an OpenMP 4.5 parallelization, wherein we will use our well performed OpenACC solver as a starting point. Furthermore, we will show the usage of auto-vectorization on linear problems, wherein the matrix has an *inner structure* such that every entry of the matrix can be considered as a quadratic $k \times k$ matrix (k usually small, 2 or 3) as it is used in [2].

Acknowledgements

The authors gratefully acknowledge support from FWF project F32-N18, special research group on "Mathematical Optimization and Applications in Biomedical Sciences". Further we thankfully acknowledge access to the ARA-Cluster in Jena [8].

References

1. Amorim, R., Haase, G., Liebmann, M., Weber dos Santos, R.: Comparing CUDA and OpenGL implementations for a Jacobi iteration. In: Smari, W.W. (ed.) Proceedings of the 2009 High Performance Computing & Simulation (HPCS'09) Conference. pp. 22–32. IEEE, New Jersey, Logos Verlag, Berlin (2009)
2. Augustin, C., Neic, A., Liebmann, M., Prassl, A., Niederer, S., Haase, G., Plank, G.: Anatomically accurate high resolution modeling of human whole heart electromechanics: A strongly scalable algebraic multigrid solver method for nonlinear deformation. Journal of Computational Physics 305 (2016)
3. Briggs, W.L., Henson, V.E., McCormick, S.F.: A Multigrid Tutorial. SIAM, Philadelphia (2000)
4. Costa, C., Haase, G., Liebmann, M., Neic, A., Plank, G.: Stepping into fully GPU accelerated biomedical applications. In: Lirkov, I., Margenov, S., Wasniewski, J. (eds.) Large Scale Scientific Computing LSSC'13. Lecture Notes in Computer Science, vol. 8353, pp. 4–15. Springer (2014)
5. Douglas, C., Haase, G., Langer, U.: A Tutorial on Elliptic PDE Solvers and Their Parallelization. Software, Environments, and Tools, SIAM, Philadelphia (2003)
6. Farber, R.: Parallel programming with OpenACC. Morgan Kaufmann, Amsterdam, 1st edn. (2017)
7. 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
8. Friedrich-Schiller-Universität Jena: Ara - das neue HPC- Cluster für Forschung und Lehre (2017), https://www.uni-jena.de/Universit%C3%A4t/Einrichtungen/URZ/Dienste/Computeserver/Ara_+Cluster.html, online; accessed 07-08-2017
9. 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)

10. 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
11. Larkin, J.: Advanced OpenACC Programming (2015), http://on-demand.gputechconf.com/gtc/2015/presentation/S5195-Jeff-Larkin.pdf, online; accessed 27-07-2017
12. Larson, M.G., Bengzon, F.: The Finite Element Method: Theory, Implementations and Applications, Texts in Computational Science and Engineering, vol. 10. Springer, Berlin, Heidelberg, 1st edn. (2013)
13. Li, X., Shih, P.C., Overby, J., Seals, C., Lim, A.: Comparing Programmer Productivity In Openacc and CUDA: An Empirical Investigation. International Journal of Computer Science, Engineering and Applications (IJCSEA) 6(5) (2016)
14. Liebmann, M.: Parallel toolbox web page (2017), http://paralleltoolbox.sourceforge.net/, online; accessed 09-08-2017
15. Mikalsen, M.A.: OpenACC-based Snow Simulation. Master's thesis, NTNU-Trondheim, Norwegian University of Science and Technology, NTNU, NO-7491 Trondheim, Norway (2013)
16. Plank, G., Liebmann, M., Weber dos Santos, R., Vigmond, E., Haase, G.: Algebraic multigrid preconditioner for the cardiac bidomain model. IEEE Transactions on Biomedical Engineering 54(4), 585–596 (2007), http://dx.doi.org/10.1109/TBME.2006.889181
17. Roth, B.J.: Bidomain model. Scholarpedia 3(4), 6221 (2008)
18. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia (2003)
19. Springer, P.: Openacc a step towards heterogeneous computing (2017), http://hpac.rwth-aachen.de/people/springer/openacc_seminar.pdf, online; accessed 01-08-2017