SpezialForschungsBereich F 32

Karl–Franzens Universität Graz
Technische Universität Graz
Medizinische Universität Graz

OpenACC and CUDA parallelization for the Elasticity Problem in CARP Techreport

S. Rosenberger G. Haase

SFB-Report No. 2017-004 April 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**

SFB Research Center
Mathematical Optimization and Applications in Biomedical Sciences

OpenACC and CUDA parallelization for the Elasticity Problem in CARP Techreport

Stefan Rosenberger and Gundolf Haase

April 4, 2017

CONTENTS

1 HARDWARE INFORMATION

For this description we used a *normal* computer. For comparison reasons, we note the Hardware properties:

```
pgaccelinfo -v

CUDA Driver Version:           8000
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  375.39  Tue Jan 31 20:47:00
    PST 2017

Device Number:                 0
Device Name:                   GeForce GTX 680
Device Revision Number:        3.0
Global Memory Size:            4232183808
Number of Multiprocessors:     8
Number of SP Cores:            1536
Number of DP Cores:            512
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    1058 MHz
Execution Timeout:             Yes
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   No
Memory Clock Rate:             3004 MHz
Memory Bus Width:              256 bits
L2 Cache Size:                 524288 bytes
Max Threads Per SMP:           2048
Async Engines:                 1
Unified Addressing:            Yes
Managed Memory:                Yes
PGI Compiler Option:           -ta=tesla:cc30

OpenCL Platform:               NVIDIA CUDA
OpenCL Vendor:                 NVIDIA Corporation

Device Number:                 0
Device Name:                   GeForce GTX 680
Available:                     Yes
Compiler Available:            Yes
Device Version:                OpenCL 1.2 CUDA
Global Memory Size:            4232183808
Maximum Object Size:           1058045952
Global Cache Size:             131072
Max Clock (MHz):               1058
Compute Units:                 8
Constant Memory Size:          65536
Local Memory Size:             49152
```

```
Workgroup Size:                 1024
Address Bits:                   64
ECC Support:                    No
```

and:

```
nvidia-smi
Wed Feb 22 15:27:57 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 375.39                 Driver Version: 375.39                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 680      Off | 0000:01:00.0     N/A |                  N/A |
| 30%   39C    P8    N/A /  N/A |    220MiB /  4036MiB |     N/A      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0                    Not Supported                                       |
+-----------------------------------------------------------------------------+
```

2 OpenACC parallelization

We consider the current implementation of the conjugate gradient method with an algebraic multigrid method as preconditioner (named inverse operator in the given code). To show the different methods we use the *linear_operator_acc* class[1].

```
template<class T, class S>
class linear_operator_acc : public generic_operator<T, S>
{
private:
   const toolbox_vector<T> &_acnt;
   const toolbox_vector<T> &_adsp;
   const toolbox_vector<T> &_acol;
   const toolbox_vector<S> &_aele;
   const T _max_length_local;

   void matrix_vector(const toolbox_vector<S> &_u, toolbox_vector<S> &_v) const
   {
      const T *__restrict acnt = _acnt.data(), *__restrict acol = _acol.data(),
      *__restrict adsp = _adsp.data();
      const S *__restrict aele = _aele.data();
      const S *__restrict u = _u.data();
      S *__restrict v = _v.data();
      int size = _acnt.size();

      const int acnt_size = _acnt.size();
      const int acol_size = _acol.size();
      const int adsp_size = _adsp.size();
      const int aele_size = _aele.size();
```

[1] Method to calculate the matrix vector product.

```
      const int v_size = _v.size();
      const int u_size = _u.size();
      // Begin matrix vector product
      T size_vector = _v.size();
      #pragma acc parallel loop vector_length(_block_size) pcopyin(acol[0:acol_size], aele[0:
          aele_size], u[0:u_size], v[0:v_size])
      for(int ii=0; ii<size_vector; ii+=_block_size){
         const T idx    = ii*_max_length_local;
         const int kk_max= _block_size+ii<size_vector ? _block_size+ii : size_vector;
         #pragma acc loop
         for(int kk=ii; kk<kk_max; ++kk){
            S s = 0.0;
            #pragma acc loop seq
            for(int jj=0; jj < _max_length_local; ++jj){
               const T q = acol[idx+kk-ii+jj*_block_size];
               s += aele[idx+kk-ii+jj*_block_size] * u[q];
            }
            v[kk] += s;
         }
      }
      // End matrix vector product
   }

public:
   linear_operator_acc(const toolbox_vector<T> & acnt,
                       const toolbox_vector<T> & adsp,
                       const toolbox_vector<T> & acol,
                       const toolbox_vector<S> & aele,
                       const T max_length_local):
                       _acnt(acnt),
                       _adsp(adsp),
                       _acol(acol),
                       _aele(aele),
                       _max_length_local(max_length_local)
   {
#ifdef _OPENACC
      todev();
#endif
   }

   void operator()(const toolbox_vector<S> & u, toolbox_vector<S> & v) const
   {
#ifdef _OPENACC
      v.zero_acc();
#else
      v.zero();
#endif
      matrix_vector(u, v);
   }

   void apply_add(const toolbox_vector<S> & u, toolbox_vector<S> & v) const
   {
      matrix_vector(u, v);
   }

   void todev(){
#ifdef _OPENACC
#pragma acc enter data pcopyin(this[0:1])
      _acnt.todev();
```

```
      _adsp.todev();
      _acol.todev();
      _aele.todev();
#endif
   }

   ~linear_operator_acc() {
      _acnt.fromdev();
      _adsp.fromdev();
      _acol.fromdev();
      _aele.fromdev();
   }
};
```

One can parallelize the *critical loop*[2] also with the pragma *kernels*.

```
#pragma acc kernels vector_length(_block_size) pcopyin(acol[0:acol_size], aele[0:aele_size], u
    [0:u_size], v[0:v_size])
   for(int ii=0; ii<size_vector; ii+=_block_size){
      const int idx    = ii*_max_length_local;
      const int kk_max= _block_size+ii<size_vector ? _block_size+ii : size_vector;
      for(int kk=ii; kk<kk_max; ++kk){
         double s = 0.0;
         for(int jj=0; jj < _max_length_local; ++jj){
            const int q = acol[idx+kk-ii+jj*_block_size];
            s += aele[idx+kk-ii+jj*_block_size] * u[q];
         }
         v[kk] += s;
      }
   }
```

2.1 Kernel vs. Parallel loop

If one compares these two different methods to parallelize the loops, one find in literature [3, page 5] and online information [7], that the use of *kernels* is to be preferred (gives more freedom for the compiler). But if one tries to execute the program, one find that the parallelization with kernels is **much** slower! We can determine the error pretty simple straight forward:
The difference is *easily* seen if one looks at the compiler output (with *-Minfo=ccff,loop*).
Paralellization with *parallel loop*:

```
linear_operator_parallel_loop(unsigned int, unsigned int, unsigned int, const int *, int,
    const double *, int, const double *, int, double *, int):
      1, include "toolbox_funcs.hpp"
           2, include "toolbox.h"
               69, include "sep_openacc_loops.cpp"
                    42, Generating copyin(acol[:acol_size],aele[:aele_size],u[:u_size],v[:
                        v_size])
                        Accelerator kernel generated
                        Generating Tesla code
                        44, #pragma acc loop gang /* blockIdx.x */
                        48, #pragma acc loop vector(_block_size) /* threadIdx.x */
                        51, #pragma acc loop seq
                    48, Loop is parallelizable
                    51, Loop is parallelizable
```

Parallelization with *kernels*:

[2] marked with *Begin* ... and *End* ...

```
linear_operator_parallel_loop(unsigned int, unsigned int, unsigned int, const int *, int,
    const double *, int, const double *, int, double *, int):
      1, include "toolbox_funcs.hpp"
           2, include "toolbox.h"
                69, include "sep_openacc_loops.cpp"
                     21, Generating copyin(acol[:acol_size],aele[:aele_size],u[:u_size],v[:
                         v_size])
                     23, Loop carried dependence of v-> prevents parallelization
                         Loop carried backward dependence of v-> prevents vectorization
                         Accelerator kernel generated
                         Generating Tesla code
                         23, #pragma acc loop seq
                         26, #pragma acc loop vector(128) /* threadIdx.x */
                         28, #pragma acc loop seq
                     26, Loop is parallelizable
                     28, Loop is parallelizable
```

The compiler (with the *#pragma* option kernels) does not parallelize the loop efficiently. *Kernels* recognizes that the most inner loop can not be parallelized, but as a consequence, it does not parallelize the most outer loop.
Therefore, the use of *#pragma acc kernels* is not the best choice for data dependent loops (or regions). Our experience is, that for simple loops, *kernel* works pretty well.

2.2 Use Integer Values in Parallel Regions

OpenACC updates single values (double or int) between host and device automatically. For example, if we look at the single values of the *matrix_vector* routine in the class *linear_operator_acc*, we see that we do not use any data update information for the values

```
    const T *__restrict acnt = _acnt.data(), *__restrict acol = _acol.data();
    const S *__restrict aele = _aele.data();
    const S *__restrict u = _u.data();
    S *__restrict v = _v.data();
    int size = _acnt.size();

    const int acnt_size = _acnt.size();
    const int acol_size = _acol.size();
    const int aele_size = _aele.size();
    const int v_size = _v.size();
    const int u_size = _u.size();

    T size_vector = _v.size();
```

since OpenACC handles those values automatically.

> ☞ Be very careful with automatic data handling in *large* programs. OpenACC does not support references of values. We found, a device data handling bug for the member value of *_max_length_local*, which occurs only after the construction (and of course) use of the method several times.

We fixed the bug by defining a local variable for this value:

```
  const int max_length_local = _max_length_local;
  T size_vector = _v.size();
```

```
#pragma acc parallel loop vector_length(_block_size) pcopyin(acol[0:acol_size], aele[0:
    aele_size], u[0:u_size], v[0:v_size])
for(int ii=0; ii<size_vector; ii+=_block_size){
   const int idx    = ii*max_length_local;
   const int kk_max= _block_size+ii<size_vector ? _block_size+ii : size_vector;
   #pragma acc loop independent
   for(int kk=ii; kk<kk_max; ++kk){
      double s = 0.0;
      #pragma acc loop seq
      for(int jj=0; jj < max_length_local; ++jj){
         const int q = acol[idx+kk-ii+jj*_block_size];
         s += aele[idx+kk-ii+jj*_block_size] * u[q];
      }
      v[kk] += s;
   }
}
```

which makes no sense for sequential programming on a host, but it is required for the OpenACC(PGI)-compiler.

2.3 Data Handling

We need a concept to handle the data on the device for the whole parallel solver. OpenACC provides the possibility to use data regions to handle the data in a clear way. e.g.:

```
#pragma acc data copyin(acol[0:acol_size], aele[0:aele_size], u[0:u_size], v[0:v_size])
{
   // Do some parallel stuff
}
```

This environments are easy to *overview*, but not useful for our problem. We use to handle the same data on the device for different classes (functions) on the device. e.g. the matrix entries are used, and if one uses this type of data regions, one risk a tremendous increase of copy operations from and to the device.
We prefer the use of

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

in the class *toolbox_vector* to handle the data of each toolbox vector[3]. Compared to the data regions, we have to define the *exit* point by our self. For this purpose, we define the function

```
void fromdev(){
   #pragma acc exit data delete( _data[0:_size], this[0:1])
}
```

in the class *toolbox_vector*.

2.3.1 No Overlapping Data Regions

OpenACC supports the use of overlapping data regions, but unfortunately this can lead to some tricky mistakes. For example, if one has two host vectors

[3] Note: The pragma options *pcopyin, pcopy, copyin, copy* does not necessarily copy the data to the device. It always checks for existing data on the device. If the data for the pointer *_data* is already on the device, OpenACC will not copy any value. For this purpose, we have the function *updatedev()* in our code, which is a deep copy operation for the array *_data*.

```
toolbox_vector<double> vec_1, vec_2;
```

Now we want to copy them for a loop on the device, therefore it is necessary to create a data pointer and one has to define the length of the data. e.g.:

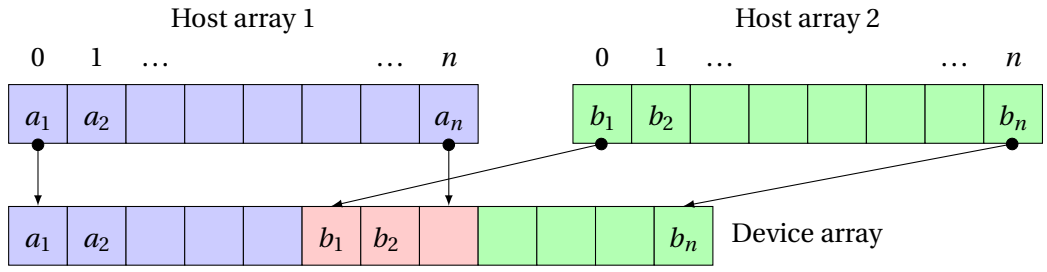
```
double* vec_1_data = vec_1.data();
double* vec_2_data = vec_2.data();

int vec_1_size = vec_1.size();
int vec_2_size = vec_2.size();
```

One can initialise the loop on the device with:

```
#pragma acc parallel loop copy(vec_1_data[0:vec_1_size], vec_2_data[0:vec_2_size])
for(int ii=0; ii<length_loop; ii++){
   // Do some parallel stuff
}
```

and execute the program. One will find a *working* device function (kernel).

☞ **But** there is a tricky hidden error. The standard parallelization does not assume that there is no overlapping data, therefore the pointers aren't *safe*. That means, it might happen that the compiler copies the data of *vec_1* to the device **and** in the next step it starts to copy the data of *vec_2* to the device, but it initialises the device pointer in the *middle* of *vec_1* and **overwrites** the values of *vec_1* with the values of *vec_2*.
For a better understanding, here is a visualisation of the problem:



Fortunately we found two methods to protect us from such bug's:

1. The *fastest* way is it to set the compiler flag: *-Msafeptr=all*.
   With this flag, the compiler assumes that there can't be a overlapping data array at all and therefore it will never try to overwrite any data.

2. An alternative way is it to define the pointer with the key word *restrict*:

   ```
   double *__restrict vec_1_data = vec_1.data();
   double *__restrict vec_2_data = vec_2.data();

   int vec_1_size = vec_1.size();
   int vec_2_size = vec_2.size();
   ```

   With this additional information the compiler will not overwrite the existing data.[4]

[4] The first method seams to be better, but with the compiler flag one limit the possibilities of the compiler. With the second method, the compiler can do an better optimisation on the rest of the code.

3 Parallelization with OpenMP or OpenACC

Previous developments provided already an implemented OpenMP parallelization for the parallel toolbox. The idea between OpenMP and OpenACC is similar. We consider again the parallelization of the matrix-vector product.

```
#pragma omp parallel for schedule(guided)
for(int i = 0; i < size; i++)
{
   S s = v_data[i];
   const T *__restrict p_col = col + dsp[i];
   const S *__restrict p_ele = ele + dsp[i];
   const T csize = cnt[i];
   for(T j = 0; j < csize; j++)
   {
      T q = p_col[j];
      S a = p_ele[j];
      s += a * u_data[q];
   }
   v_data[i] += s;
}
```

One can compare the OpenMP parallelization with the OpenACC parallelization:

```
#pragma acc parallel loop independent pcopyin(cnt[0:cnt_size], col[0:col_size], dsp[0:dsp_size
    ], ele[0:ele_size], u_data[0:u_size], v_data[0:v_size])
for(int i = 0; i < size; i++)
{
   S s = v_data[i];
   const T *__restrict p_col = col + dsp[i];
   const S *__restrict p_ele = ele + dsp[i];
   const T csize = cnt[i];

   #pragma acc loop seq
   for(T j = 0; j < csize; j++)
   {
      T q = p_col[j];
      S a = p_ele[j];
      s += a * u_data[q];
   }
   v_data[i] += s;
}
```

OpenMP parallelization is very similar to OpenACC. A simple *#pragma omp* before the loop, and OpenMP parallelized the loop. But (of course) there is an elementary difference.
OpenMP splits **one thread** to *n* parallel threads for **one task**.

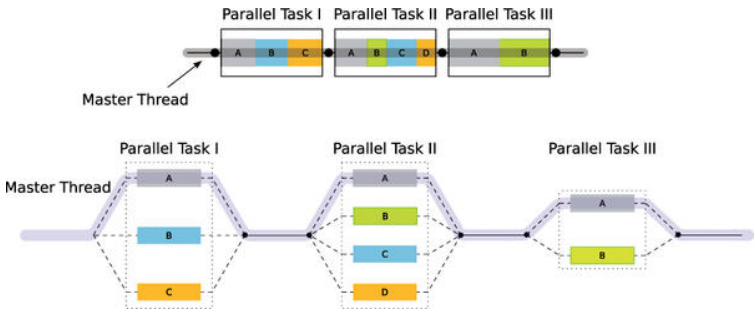


Figure 3.1: OpenMP Thread (Picture from [5])

On the other hand, OpenACC starts **several threads**, and execute each thread independently[5].

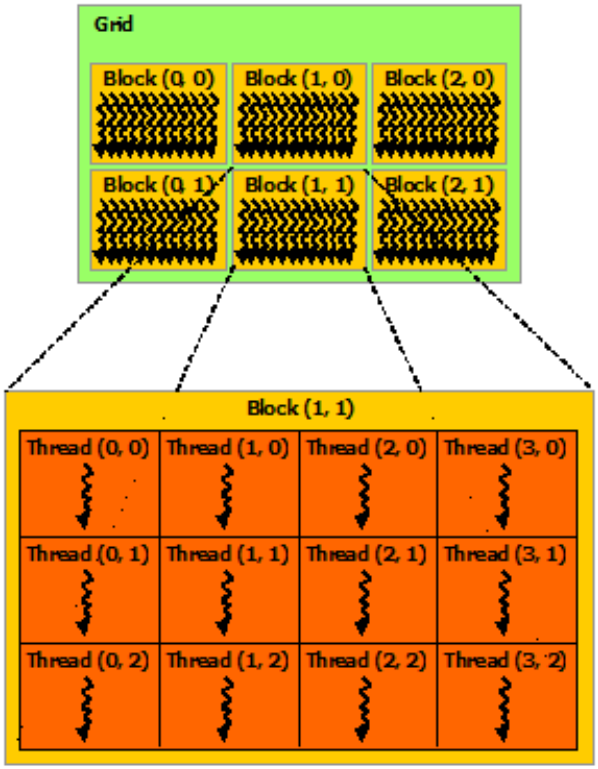


Figure 3.2: OpenACC Grid (Picture from [8])

The main difference for the use of those methods, is that one does not really need to take care on data availability for OpenMP. Since the data is on the host the parallelization is with *shared data*. This is of course different to any device method.

☞ Note, that in both cases (OpenMP and OpenACC) one must take care on *race conditions*.

3.1 Optimise Data Transfers

To create a real efficient program, one has to minimise the data transfer between host and device. Usually any *standard* introduction to OpenACC propose to use

```
#pragma acc kernels loop independent pcopyin(acnt[0:acnt_size], acol[0:acol_size], adsp[0:
    adsp_size], aele[0:aele_size], u[0:u_size]) pcopyin(v[0:v_size])
// Do some parallel stuff
```

or (since we have already seen, that *kernels* is in general not the best choice)

```
#pragma acc parallel loop vector_length(_block_size) pcopyin(acol[0:acol_size], aele[0:
    aele_size], u[0:u_size], v[0:v_size])
// Do some parallel stuff
```

In particular for a larger problems, it is not a good choice to use the (more ore less) automatically used *pcopy* options. Since we use the same operators for different operations, and at some points we

[5] Except one define dependencies explicit!

need to update the host data, we have to handle the data by our self's. For this purpose we use *data handling* functions for our toolbox vectors:

```
void todev(){ // move to device
   #pragma acc enter data pcopyin(this[0:1], _data[0:_size])
}
void todev() const{ // move to device
   #pragma acc enter data pcopyin(this[0:1], _data[0:_size])
}
void createdev(){ // move to device
   #pragma acc enter data pcreate(this[0:1], _data[0:_size])
}
void createdev() const{ // move to device
   #pragma acc enter data pcreate(this[0:1], _data[0:_size])
}
void fromdev(){ // remove from device
   #pragma acc exit data delete( _data[0:_size], this[0:1])
}
void fromdev() const{ // remove from device
   #pragma acc exit data delete( _data[0:_size], this[0:1])
}
void cfromdev(){ // copy and remove from device
   #pragma acc exit data copyout( _data[0:_size], this[0:1])
}
void cfromdev() const{ // copy and remove from device
   #pragma acc exit data copyout( _data[0:_size], this[0:1])
}
void updatehost(){ // update host copy of data
   #pragma acc update self( _data[0:_size] )
}
void updatehost() const { // update host copy of data
   #pragma acc update self( _data[0:_size] )
}
void updatedev(){ // update device copy of data
   #pragma acc update device( _data[0:_size] )
}
void updatedev() const{ // update device copy of data
   #pragma acc update device( _data[0:_size] )
}
```

With this functions we create the data for the device for all classes already in the constructor:

```
void todev(){
#ifdef _OPENACC
#pragma acc enter data pcopyin(this[0:1])
   _acnt.todev();
   _adsp.todev();
   _acol.todev();
   _aele.todev();
#endif
}
```

3.1.1 Run time Information for Device Operations

Now, with this functions one can start to reduce the required data transfers. For this purpose one can set the environment variable

```
export PGI_ACC_NOTIFY=3
```

We get the run time output (e.g.):

```
upload CUDA data  file=/home/rosenbs/src/carp-dcse-pt/branches/mechanics/PT_ACC_Local/
    toolbox_funcs.cpp function=_ZN14toolbox_vectorIdE5todevEv line=181 device=0 threadid=1
    variable=.pointer. bytes=8

launch CUDA kernel  file=/home/rosenbs/src/carp-dcse-pt/branches/mechanics/PT_ACC_Local/
    toolbox_funcs.cpp function=_Z13scale_add_accIdEvR14toolbox_vectorIT_ERKS2_S1_ line=517
    device=0 threadid=1 num_gangs=6739 num_workers=1 vector_length=128 grid=6739 block=128

download CUDA data  file=/home/rosenbs/src/carp-dcse-pt/branches/mechanics/PT_ACC_Local/
    toolbox_funcs.cpp function=_ZN14toolbox_vectorIdE10updatehostEv line=205 device=0 threadid
    =1 bytes=6900120
```

In this list, one can see the 3 characteristically outputs for device handling with OpenACC:

- *upload:* is shown if the program copies from host to device. For this specific example, it shows the latest *.cpp* file we used to compile the program, in our case *toolbox_funcs.cpp*, which is for the parallel toolbox useless since we initialize all parts with header files (*.h*).
  The second information is which specific function (and at which line in the source code) is used to upload the data (*todev*). In our case is this information again useless, since we use almost everywhere the upload function from the class *toolbox_vector*.
  The only interesting case would be, if the program uses another upload function. In this case, we have to double check our data management, since it is not planned to use an upload at this point.
  The next informations is the device number (one can use several devices with OpenACC) and the thread number which calls the upload function.
  The last information is about the data we upload. In this case the program upload a single pointer (we get the same information if OpenACC upload a single value).

- *launch:* is shown if the program calls a device kernel. Again we get some information about the called kernel. The first information is again the compiling source (which is again useless for us), the second information is the called device function (which is constructed by OpenACC). In the name of the device function, we see the name of the host function (which we wrote). In this case *scale_add_acc*. It shows us in addition how much worker are uses and how long the vectorization is (as we defined it). Grid an block is similar to CUDA.

- *download:* The download information is similar to the *upload* output. The only difference is that we copy data from the device to the host. And in this case, we copy an array with an length of 6900120 bytes.

3.1.2 Summarize the Data Handling

We wrote the program in that way, that it is not necessary that OpenACC handles data during the *loop-pragmas* e.g.:

```
#pragma acc parallel loop vector_length(_block_size) pcopyin(acol[0:acol_size], aele[0:
    aele_size], u[0:u_size], v[0:v_size])
// Do some parallel stuff
```

To check for an optimal data handling, one can use the compile option

```
-ta=nvidia:time
```

We get the output (after the run):

```
/home/rosenbs/src/carp-dcse-pt/branches/mechanics/PT_ACC_Local/toolbox_funcs.cpp
  _ZNK14toolbox_vectorIiE5todevEv  NVIDIA  devicenum=0
    time(us): 11,565,501
    184: data region reached 366912 times
        184: data copyin transfers: 375351
             device time(us): total=11,565,501 max=2,706 min=2 avg=30


/home/rosenbs/src/carp-dcse-pt/branches/mechanics/PT_ACC_Local/toolbox_funcs.cpp
  _Z18scalar_product_accIdEvRK14toolbox_vectorIT_ES4_RS1_  NVIDIA  devicenum=0
    time(us): 2,294,955
    58: compute region reached 15799 times
        58: data copyin transfers: 15799
             device time(us): total=130,152 max=33 min=1 avg=8
        58: kernel launched 15799 times
            grid: [6739]  block: [128]
             device time(us): total=1,720,038 max=123 min=106 avg=108
            elapsed time(us): total=2,000,477 max=1,212 min=123 avg=126
        58: reduction kernel launched 15799 times
            grid: [1]  block: [256]
             device time(us): total=190,085 max=19 min=11 avg=12
            elapsed time(us): total=439,345 max=1,186 min=26 avg=27
        58: data copyout transfers: 15799
             device time(us): total=254,680 max=70 min=11 avg=16


/home/rosenbs/src/carp-dcse-pt/branches/mechanics/PT_ACC_Local/toolbox_funcs.cpp
  _Z13add_scale_accIdEvR14toolbox_vectorIT_ERKS2_S1_  NVIDIA  devicenum=0
    time(us): 1,044,286
    403: compute region reached 7699 times
        403: kernel launched 7699 times
            grid: [6739]  block: [128]
             device time(us): total=1,044,286 max=162 min=133 avg=135
            elapsed time(us): total=1,172,822 max=296 min=148 avg=152
    403: data region reached 15398 times
```

Again, let us consider the output.

- The first output is about the function *_ZNK14toolbox_vectorIiE5todevEv*, which represent the *todev()* function from the toolbox vector.
  In tells us, how often and how long this function was used. In this case, it tells us how often the function was called and how long it took to run the function.

- The second output is about the function *_Z18scalar_product_accIdEvRK14toolbox_vectorIT_ES4_RS1_*, which represent the device kernel for the scalar product loop.
  We see 4 events which occur during this *region*.
  - *copyin:* For the scalar product, we need to *copyin* some values in every loop. We can't see which data is copied, but for sure we see that the required device time is marginal (compared to the kernel launched time).
  - *kernel launched*: This is the *main* part of the device function. It shows how often the kernel is executed.
  - *reduction kernel*: For the scalar product we have an reduction on the result value. It shows (again) how often we need this operation during the run.

– *copyout*: Shows how often (and how long) data is transferred from the device to the host. We can't see which data is copied (but if one look at the source code, one can see that it has to be the scalar values needed for the loop).

We see that the scalar product needs copyin and copyout operation during the kernel execution. But, the amount of time required for these data transfers, compared to the device time, is negligible.

- The last output shows a *perfect* data handling with OpenACC. The function *add_scale_acc* has no data transfer at all. This happens, since this function is only used in the algebraic multigrid, and therefore all scalar values needed, has been copied to the device before the kernel of *add_scale_acc* is called.

4 Use CUDA Kernels in OpenACC Programs

One can use (possibly existing) CUDA kernels and incorporate them into an OpenACC program. To report this, we use again the execution of the matrix vector multiplication.
We consider the following **CUDA** function:

```
__global__ void _device_generic_parallel_loop(const unsigned int size_vector, const unsigned
    int _block_size, const unsigned int _max_length_local, const int* acol, const int
    acol_size, const double* aele, const int aele_size, const double* u, const int u_size,
    double* v, const int v_size)
{
   const unsigned int kk = _block_size * blockIdx.x + threadIdx.x;
   const unsigned int ii = _block_size * blockIdx.x;
   if(kk < size_vector)
   {
      const unsigned int idx = ii*_max_length_local;
      double s = 0.0;
      for(unsigned int jj=0; jj < _max_length_local; ++jj){
         s += aele[idx+kk-ii+jj*_block_size] * u[acol[idx+kk-ii+jj*_block_size]];
      }
      v[kk] += s;
   }
}

extern "C" {
   void _device_linear_operator_SR(const int &size_vector, const int &_block_size, const int &
       _max_length_local, const int* d_acol, const int& acol_size, const double* d_aele,
         const int &aele_size, const double* d_u, const int &u_size, double* d_v, const int &
             v_size)
   {
         _device_generic_parallel_loop<<<(size_vector + _block_size - 1)/_block_size,
             _block_size>>>(size_vector, _block_size, _max_length_local, d_acol, acol_size,
            d_aele, aele_size, d_u, u_size, d_v, v_size);
   }
}
```

wherein the function *_device_linear_operator_SR* calls the CUDA kernel. Usually, one has to give precise data information for CUDA functions to execute them on the device. e.g.:

```
void assign(const T* s, const T* e)
{
   int n = (int)(e - s);
   if(_capacity < n)
   {
```

```
        if(_data != 0) cudaFree(_data);
        cudaMalloc((void**) &_data, n * sizeof(T));
        _capacity = n;
    }
    cudaMemcpy(_data, s, n * sizeof(T), cudaMemcpyHostToDevice);
    _size = n;
}
```

wherein *_size, _capacity* are integer and *_data* is a pointer to an array. Note, that a negative value of *n* would lead to an error.
With OpenACC one can handle device data much simpler:

```
void todev(){ // move to device
    #pragma acc enter data pcopyin(this[0:1], _data[0:_size])
}
```

With this simple *pargma* (wherein *_data* is a pointer to an array and *_size* is the length of the array) OpenACC creates **and** copies the data to the device (note, that the option *pcopyin* and *copyin* copies the data only if there is no already created array on the device. We consider the OpenACC data handling later.).
If one uses OpenACC to handle the data on the device, and want to use (in the same scope) a CUDA kernel, one needs to define the device pointers to the corresponding arrays. Fortunately, there is a *very simple* function to get the device pointers. e.g.:

```
double* __restrict d_aele = (double*) acc_deviceptr((double*) aele);
```

wherein *aele* must be a host pointer, and *d_aele* becomes the device pointer. Note, that the data for the pointer *aele* **must** be already on the device (otherwise one get an obvious segmentation fault).
With this we can now use the CUDA device kernel in out OpenACC parallelization like that:

```
extern "C" {
void _device_linear_operator_SR(const int &size_vector, const int &_block_size, const int &
    _max_length_local, const int* d_acol, const int& acol_size, const double* d_aele,
        const int &aele_size, const double* d_u, const int &u_size, double* d_v, const int &
            v_size);
}


.
.
.
void operator()(const toolbox_vector<S> &_u, toolbox_vector<S> &_v) const
{
    _v.zero_acc();

    const double* __restrict d_aele = (double*) acc_deviceptr((double*) _aele.data());
    const int* __restrict d_acol = (int*) acc_deviceptr((int*) _acol.data());

    const double* __restrict d_u = (double*) acc_deviceptr((double*) _u.data());
    double* __restrict d_v = (double*) acc_deviceptr((double*) _v.data());

    const int size_vector = _v.size();
    const int acol_size = _acol.size();
    const int aele_size = _aele.size();

    _device_linear_operator_SR(size_vector, _block_size, _max_length_local, d_acol, acol_size,
        d_aele, aele_size, d_u, _u.size(), d_v, size_vector);
}
```

4.1 Compiling and Linking

After a *complete* parallelization one has to compile the parts of the program seperately. We call our file with the CUDA functions *toolbox_sr.cu*. One hast to create an object file to link to program:

```
nvcc -c toolbox_sr.cu
```

In the next step we compile and link the program with the CUDA object file:

```
pgc++ -pgc++libs -L/opt/pgi/linux86-64/16.9/bin/ -L/usr/lib/x86_64-linux-gnu/ -ta=nvidia:cc2+,
    fastmath -Mlre=array -Msafeptr=all -Minfo=accel,loop -DUSECUDA_SR -Mcuda=8.0 /home/rosenbs
    /src/carp-dcse-pt/branches/mechanics/PT_C/toolbox/lib/toolbox_sr.o -O3 -D_OPENACC -
    DFAST_AMG -DNOSSE -DP2P_v1 -DMUMPS -c toolbox_funcs.cpp -I. -I/home/rosenbs/src/petsc
    -3.6.3/gnu-opt/include
```

where in one need to devine the *key* option *-Mcuda=8.0* which informs the PGI compiler that we have some CUDA kernels in our program.

> ☞ Note, that one can combine CUDA kernels and OpenACC kernels in one Program without any problems.

4.2 Test Comparision

We tried the parallelization with OpenACC and an corresponding parallelization with CUDA (and data handling with OpenACC) on a

- Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
- Nvidia GeForce GTX 680

system, and compared it for a *short* simulation of 10 μs:

CUDA & OpenACC 235,6s

OpenACC 239,0s

Therefore, we can conclude that the automatic OpenACC parallelization can keep up with self written CUDA kernels. But we are stil not close to an optimised CUDA program. A previous developed CUDA program get for the same simulation properties:

CUDA 106,9s

Note that the existing CUDA solver is a **very well** optimized program. For example, we use *prefeching* in all CUDA kernels. (This CUDA kernels are not the same as those for the CUDA & OpenACC combined program)

5 MPI Splitting with OpenACC

We can use an MPI splitting method from the parallel solver, to split the matrix system to several *dependent* subproblems.
At some *key* points one has to take care, that the different processes exchange the required data. For this purpose exist the *communicator*-class. We handle the data exchange as follows:

```
#ifdef _OPENACC
   int network_size;
   network::size(network_size);
   int nshared_acc = _com.shared_nodes().size();

   if(network_size > 1) _u.updatehost_shared_nodes(nshared_acc);
#endif
   _com.accumulate(_u);
#ifdef _OPENACC
   if(network_size > 1) _u.updatedev_shared_nodes(nshared_acc);
#endif
```

To understand the meaning of the update, we have to understand (at least a bit) the communicator. For example, we consider a problem with 100 nodes. Let us define the node vector as follows

$$(1,2,3,4,\ldots,99,100)$$

The communicator defines *master nodes* and reorder the vectors. Let us assume, the communicator has chosen 15,37 and 65 as the master nodes. Then we get an reordered node vector as:

$$(15,37,65,1,2,3,4,\ldots,99,100)$$

which means, we have only the first *nshared* elements to exchange between the threads. Since the data exchange between the host and device is very expensive, we reduce the *copy operations* to a minimum.
We compared the amount of time we would need with an *complete* update, to an reduced (to *nshared*) update version.

nshared_acc we get the following time table (with the option *-ta=nvidia:time*):

```
Thread 1:
Accelerator Kernel Timing data

/home/rosenbs/src/carp-dcse-pt/branches/mechanics/PT_ACC_Local/toolbox_funcs.cpp
  _ZN14toolbox_vectorIdE23updatehost_shared_nodesEi  NVIDIA  devicenum=0
    time(us): 1,142,760
    220: update directive reached 89230 times
        220: data copyout transfers: 89230
             device time(us): total=1363,385
/home/rosenbs/src/carp-dcse-pt/branches/mechanics/PT_ACC_Local/toolbox_funcs.cpp
  _ZN14toolbox_vectorIdE22updatedev_shared_nodesEi  NVIDIA max=2,893 min=2  devicenum=0
    time(us): 1,401,693
    228: update directive reached 89230 times
        228: data copyin transfers: 89230
             device time(us): total=1,401,693 avg=5
```

_u.size() we get the following time table, if we make a *complete* update of the vectors:

```
Thread 1:
Accelerator Kernel Timing data

/home/rosenbs/src/carp-dcse-pt/branches/mechanics/PT_ACC_Local/toolbox_funcs.cpp
  _ZN14toolbox_vectorIdE23updatehost_shared_nodesEi  NVIDIA  devicenum=0
    time(us): 15,509,566
    220: update directive reached 89230 times
        220: data copyout transfers: 89230
```

```
            device time(us): total=15,509,566 max=1,274 min=5 avg=173
/home/rosenbs/src/carp-dcse-pt/branches/mechanics/PT_ACC_Local/toolbox_funcs.cpp
  _ZN14toolbox_vectorIdE22updatedev_shared_nodesEi  NVIDIA  devicenum=0
    time(us): 15,231,985
    228: update directive reached 89230 times
        228: data copyin transfers: 89230
            device time(us): total=15,231,985 max=1,347 min=4 avg=170
```

We see immediately that the time for a *complete* update needs 10 times more time than an update restricted to the shared nodes.
In our program, with two threads, means this that we are 60 seconds faster in a program that need 430 seconds in total (per thread). Therefore, it is worth to reduce the updates between host and device in every step as much as possible.

6 Improvement of the Parallelization with OpenACC

In (nearby) every tutorial for OpenACC, one can find how to accelerate the parallelization. Since we want to argue why we improve our code, we want to follow a *usuall* tutorial.

6.1 Example

We consider the Jacobi iteration on a 2D square discretization:

$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}.$$

We tested the following simple solving algorithm:

```
while ( err > tol && iter < iter_max )
{
   err=0.0;
   for( int j = 1; j < n-1; j++)
   {
      for(int i = 1; i < m-1; i++)
      {
         Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j-1][i] + A[j+1][i]);
         err = max(err, abs(Anew[j][i] - A[j][i]));
      }
   }

   for( int j = 1; j < n-1; j++)
   {
      for( int i = 1; i < m-1; i++ )
      {
         A[j][i] = Anew[j][i];
      }
   }
   iter++;
}
```

One can apply now the OpenMP parallelization:

```
while ( err > tol && iter < iter_max )
{
   err=0.0;
   #pragma omp parallel for shared(m, n, Anew, A) reduction(max:err)
```

```
   for( int j = 1; j < n-1; j++)
   {
      for(int i = 1; i < m-1; i++)
      {
         Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j-1][i]...
             ... + A[j+1][i]);
         err = max(err, abs(Anew[j][i] - A[j][i]));
      }
   }

   #pragma omp parallel for shared(m, n, Anew, A)
   for( int j = 1; j < n-1; j++)
   {
      for( int i = 1; i < m-1; i++ )
      {
         A[j][i] = Anew[j][i];
      }
   }
   iter++;
}
```

And now with OpenACC pragmas:

```
while ( err > tol && iter < iter_max )
{
   err=0.0;
   #pragma acc parallel loop reduction(max:err)
   for( int j = 1; j < n-1; j++)
   {
      for(int i = 1; i < m-1; i++)
      {
         Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] + A[j-1][i]...
             ... + A[j+1][i]);
         err = max(err, abs(Anew[j][i] - A[j][i]));
      }
   }

   #pragma acc parallel loop
   for( int j = 1; j < n-1; j++)
   {
      for( int i = 1; i < m-1; i++ )
      {
         A[j][i] = Anew[j][i];
      }
   }
   iter++;
}
```

First of all, we followed the tutorial from [6], and found that code isn't working at all. Therefore, we wrote our own (simple) solver to compare OpenMP with OpenACC. We have:

```
#include <iostream>
#include <cmath>

using namespace std;
#include <cstdlib>
#include <sys/timeb.h>
#include <omp.h>
#include <algorithm>
```

```
int getMilliCount(){
   timeb tb;
   ftime(&tb);
   int nCount = tb.millitm + (tb.time & 0xfffff) * 1000;
   return nCount;
}

int getMilliSpan(int nTimeStart){
   int nSpan = getMilliCount() - nTimeStart;
   if(nSpan < 0)
      nSpan += 0x100000 * 1000;
   return nSpan;
}

void SetInitialFunktion(double *A, double *Anew, double *b, int nn)
{
    for(int j=0; j<nn; j++)
   {
      for(int i=0; i<nn; i++)
      {
         A[j +i*nn] = std::pow((double)-1,i) + std::pow((double)-1,j);
         Anew[j + i*nn] = 0;
      }
      b[j] = std::sqrt((double)j);
   }
}

int main()
{
   double err(1.0), tol(0.0001);
   int iter(0), iter_max(100);
   int nn = 5000;
   int MatrixSize = nn*nn;

   double* A = new double[MatrixSize];
   double* Anew = new double[MatrixSize];
   double* b = new double[nn];
   double* Acompare = new double[MatrixSize];

/*****************************************************************************/
    // First Calculation

    SetInitialFunktion(A, Anew, b, nn);
    int start = getMilliCount();
   while ( err > tol && iter < iter_max )
   {
      err=0.0;
      for( int j = 1; j < nn-1; j++)
      {
         for(int i = 1; i < nn-1; i++)
         {
            Anew[j + i*nn] = 0.25 * (A[j + (i+1)*nn]
                   + A[j + (i-1)*nn] + A[(j-1) + i*nn]
                      + A[(j+1) + i*nn] + b[j]);
            err = fmax(err, fabs(Anew[j + i*nn] - A[j + i*nn]));
         }
      }

      for( int j = 1; j < nn-1; j++)
```

```
    {
       for( int i = 1; i < nn-1; i++ )
       {
          A[j + i*nn] = Anew[j + i*nn];
       }
    }
    iter++;
}
 int milliSecondsElapsed = getMilliSpan(start);
 // Compare the result
 for( int j = 0; j < nn; j++)
 {
     for( int i = 0; i < nn; i++ )
     {
         Acompare[j + i*nn] = A[j + i*nn];
     }
 }

 /*************************************************************************/
 // Second Calculation
 // Reset initial conditions
 for(int num_of_mythread=2; num_of_mythread < 9; num_of_mythread+=2)
 {
     SetInitialFunktion(A, Anew, b, nn);
     err = 1.0;
     iter = 0;
     start = getMilliCount();

     omp_set_num_threads(num_of_mythread);
     while ( err > tol && iter < iter_max )
     {
         err=0.0;
         #pragma omp parallel for shared(nn, Anew, A) reduction(max:err)
         for( int j = 1; j < nn-1; j++)
         {
             for(int i = 1; i < nn-1; i++)
             {
                 Anew[j + i*nn] = 0.25 * (A[j + (i+1)*nn]
                         + A[j + (i-1)*nn] + A[(j-1) + i*nn]
                             + A[(j+1) + i*nn] + b[j]);
                 err = fmax(err, fabs(Anew[j + i*nn] - A[j + i*nn]));
             }
         }

         #pragma omp parallel for shared(nn, Anew, A)
         for( int j = 1; j < nn-1; j++)
         {
             for( int i = 1; i < nn-1; i++ )
             {
                 A[j + i*nn] = Anew[j + i*nn];
             }
         }
         iter++;
     }
     milliSecondsElapsed = getMilliSpan(start);

     err = 0.0;
     for( int j = 0; j < nn; j++)
     {
```

```
            for(int i = 0; i < nn; i++)
            {
                err = max(err, fabs(Acompare[j + i*nn] - A[j + i*nn]));
            }
        }
    }


    /*************************************************************************/
    // Forth Calculation
    // Reset initial conditions
    SetInitialFunktion(A, Anew, b, nn);
    err = 1.0;
    iter = 0;
    start = getMilliCount();

    #pragma acc data copy(A[0:MatrixSize]), copy(b[0:nn]), create(Anew[0:MatrixSize])
   while ( err > tol && iter < iter_max )
   {
      err=0.0;
        #pragma acc parallel loop reduction(max:err)
      for( int j = 1; j < nn-1; j++)
      {
         for(int i = 1; i < nn-1; i++)
         {
            Anew[j + i*nn] = 0.25 * (A[j + (i+1)*nn]
                   + A[j + (i-1)*nn] + A[(j-1) + i*nn]
                      + A[(j+1) + i*nn] + b[j]);
            err = fmax(err, fabs(Anew[j + i*nn] - A[j + i*nn]));
         }
      }


        #pragma acc parallel loop
      for( int j = 1; j < nn-1; j++)
      {
         for( int i = 1; i < nn-1; i++ )
         {
            A[j + i*nn] = Anew[j + i*nn];
         }
      }
      iter++;
   }
    milliSecondsElapsed = getMilliSpan(start);

    err = 0.0;
    for( int j = 0; j < nn; j++)
    {
        for(int i = 0; i < nn; i++)
        {
            err = max(err, fabs(Acompare[j + i*nn] - A[j + i*nn]));
        }
    }

   delete[] Anew;
   delete[] A;
   delete[] b;

   return 0;
```

```
}
```

One find, that we can improve the performance of the sequential code **up to a factor of 20** compared with the sequential calculation. Therefore, it makes sence to aply this *type* of parallelization.

6.2 First OpenACC parallelization for PT

Motivated from this introduction example, we aplied the parallelization to our parallel solver. We used for the whole structure a parallelization like the matrix-vector product:

```
#pragma acc kernels pcopyin(cnt[0:cnt_size], col[0:col_size], dsp[0:dsp_size], ele[0:ele_size
    ], u_data[0:u_size]) pcopy(v_data[0:v_size])
for(int i = 0; i < size; i++)
{
   S s = v_data[i];
   const T *__restrict p_col = col + dsp[i];
   const S *__restrict p_ele = ele + dsp[i];
   const T csize = cnt[i];

   for(T j = 0; j < csize; j++)
   {
      T q = p_col[j];
      S a = p_ele[j];
      s += a * u_data[q];
   }
   v_data[i] += s;
}
```

And we get: *We efficiently slowed down our parallel solver*!

- A first improvement of the code is that we used our methods for the data handling 3.1. For this purpose, we had to consider the runtime information (c.f. 3.1.1) for every data handling point. That is a lot of work (but indispensable). With this improvement, we get that the device code is (only) ~ 2 times slower than the sequential code on the host. For sure, we need more!

- In a second step, we considered the parallelization itself. We found that it is not *that easy* to parallelize nested loops. The *#pragma kernels* is for sure not the best choice. We considered that in 2.1. Therefore, we changed all loops similar to:

```
#pragma acc parallel loop independent pcopyin(cnt[0:cnt_size], col[0:col_size], dsp[0:
    dsp_size], ele[0:ele_size], u_data[0:u_size]) pcopy(v_data[0:v_size])
for(int i = 0; i < size; i++)
{
   S s = v_data[i];
   const T *__restrict p_col = col + dsp[i];
   const S *__restrict p_ele = ele + dsp[i];
   const T csize = cnt[i];

   #pragma acc loop seq
   for(T j = 0; j < csize; j++)
   {
      T q = p_col[j];
      S a = p_ele[j];
      s += a * u_data[q];
   }
   v_data[i] += s;
}
```

With this step we get a code on the device which is similar fast than a sequential code on the host. For sure, this can't be the final result.
To improve the code we need to consider the specific order of calculation on the device.

6.3 Interleaved Ellpack

In a first step we use a simplified ELLPACK-format to store the system matrix. That means, we reorder the matrix as follows:

$$A = \begin{pmatrix} 1 & 3 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 8 & 2 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 9 & 0 & 0 & 0 & 7 \end{pmatrix} \quad \rightsquigarrow \quad A = \begin{pmatrix} 1 & 3 & 0 \\ 2 & 0 & 0 \\ 1 & 0 & 0 \\ 3 & 8 & 2 \\ 4 & 0 & 0 \\ 9 & 7 & 0 \end{pmatrix}; \quad \text{col: } = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 1 & 1 \\ 3 & 1 & 1 \\ 2 & 4 & 2 \\ 2 & 1 & 1 \\ 2 & 6 & 1 \end{pmatrix};$$

So we set up a *kind of CRS-format*, wherein we fill up all rows with zeros such that all rows have the same length.

$$\begin{matrix} blocksize\left\{ \right. \\ blocksize\left\{ \right. \\ blocksize\left\{ \right. \end{matrix} \begin{pmatrix} 1 & 3 & 0 \\ 2 & 0 & 0 \\ 1 & 0 & 0 \\ 3 & 8 & 2 \\ 4 & 0 & 0 \\ 9 & 7 & 0 \end{pmatrix} \quad \begin{matrix} blocksize\left\{ \right. \\ blocksize\left\{ \right. \\ blocksize\left\{ \right. \end{matrix} \begin{pmatrix} 1 & 2 & 1 \\ 2 & 1 & 1 \\ 3 & 1 & 1 \\ 2 & 4 & 2 \\ 2 & 1 & 1 \\ 2 & 6 & 1 \end{pmatrix}; \quad \rightsquigarrow \quad A = \begin{pmatrix} 1 & 2 \\ 3 & 0 \\ 0 & 0 \\ 1 & 3 \\ 0 & 8 \\ 0 & 2 \\ 4 & 9 \\ 0 & 7 \\ 0 & 0 \end{pmatrix}; \quad \text{col: } = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ 1 & 1 \\ 3 & 2 \\ 1 & 4 \\ 1 & 2 \\ 2 & 2 \\ 1 & 6 \\ 1 & 1 \end{pmatrix}$$

6.3.1 Implementation of Ellpack

```
void convert_matrix_simple(toolbox_vector<int>& cnt, toolbox_vector<int>& dsp, toolbox_vector<
    int>& col, toolbox_vector<double>& ele, int& max_length_local){
  // Construct the rest
  max_length_local = 0;
  int new_num_lines = (((int) cnt.size()/_block_size)+1) * _block_size;

  for(int ii=0; ii < cnt.size(); ii++){
    if(max_length_local < cnt[ii]){
      max_length_local = cnt[ii];
    }
  }


  toolbox_vector<double> ele_new(max_length_local * new_num_lines, 0.0);
  toolbox_vector<int> col_new(max_length_local * new_num_lines, 0);
  toolbox_vector<int> dsp_new(new_num_lines, 0);
  toolbox_vector<int> cnt_new(new_num_lines, max_length_local);

  for(int ii=0; ii<dsp_new.size(); ii++){
    dsp_new[ii] = ii * max_length_local;
  }
```

```
    for(int ii=0, kk=0; ii<cnt.size(); ii++){
       for(int jj=0; jj<max_length_local; jj++, kk++){
          if(jj < cnt[ii]){
             ele_new[kk] = ele[dsp[ii] + jj];
             col_new[kk] = col[dsp[ii] + jj];
          }
          else{
             if(ii==0) col_new[kk] = 1;
          }
       }
    }


    int num_block = new_num_lines / _block_size;
    int num_line_block = num_block * max_length_local;

    toolbox_vector<double> ele_new_2(ele_new.size(), 0.0);
    toolbox_vector<int> col_new_2(col_new.size(), 0);
    toolbox_vector<int> dsp_new_2(num_line_block, 0);
    toolbox_vector<int> cnt_new_2(num_line_block, _block_size);

    for(int ii=0; ii<num_block; ii++){
       dsp_new_2[ii] = ii * _block_size*max_length_local;
    }

    for(int ii=0, ll=0; ii<new_num_lines; ii+=_block_size){
       for(int kk=0; kk<max_length_local; kk++){
          for(int jj=ii; jj<ii+_block_size; jj++,ll++){
             ele_new_2[ll] = ele_new[dsp_new[jj] + kk];
             col_new_2[ll] = col_new[dsp_new[jj] + kk];
          }
       }
    }

    dsp = dsp_new_2;
    col = col_new_2;
    ele = ele_new_2;
    cnt = cnt_new_2;
}
```

☞ One can improve this implementation.

The reordering of the matrix, requires (or has the goal) a change of the execution of the loop for the matrix-vector calculation.

- The standard way to calculate the matrix vector product is *simply equivalent* to the basic mathematical definition of the product $A \cdot x = v$. For the i-th component of the result vector:

$$v_i = \sum_{j=1}^{m} a_{ij} \cdot x_j$$

  and therefore the implementation (for a matrix in CRS-format) as:

```
for(int i = 0; i < size; i++)
{
   S s = v_data[i];
```

```
    const T *__restrict p_col = col + dsp[i];
    const S *__restrict p_ele = ele + dsp[i];
    const T csize = cnt[i];

    for(T j = 0; j < csize; j++)
    {
       T q = p_col[j];
       S a = p_ele[j];
       s += a * u_data[q];
    }
    v_data[i] += s;
}
```

- Now we change the order of the matrix entries

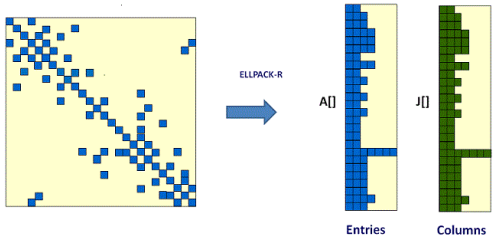


Figure 6.1: Matrix reodering (Picture from [4])

The reason for this is not obvious. We want to have, that every thread has (as much as possible) requierd elements at it's onw memory.

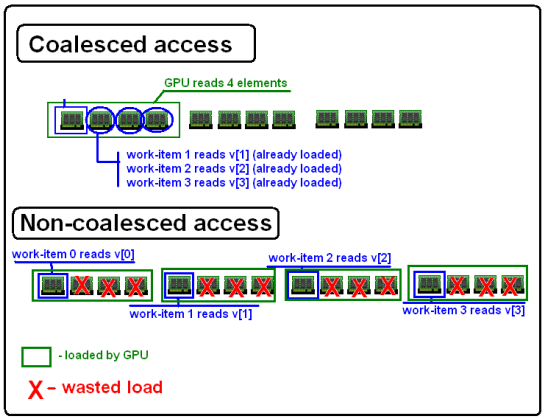


Figure 6.2: Coalesced access (Picture from [2])

With this reordering we have to change the way how we execute the matrix-vector product to:

```
const int max_length_local = _max_length_local;
T size_vector = _v.size();

#pragma acc parallel loop vector_length(_block_size) pcopyin(acol[0:acol_size], aele[0:
    aele_size], u[0:u_size], v[0:v_size])
for(int ii=0; ii<size_vector; ii+=_block_size){
   const int idx    = ii*max_length_local;
```

```
    const int kk_max= _block_size+ii<size_vector ? _block_size+ii : size_vector;
    #pragma acc loop independent
    for(int kk=ii; kk<kk_max; ++kk){
      double s = 0.0;
      #pragma acc loop seq
      for(int jj=0; jj < max_length_local; ++jj){
        const int q = acol[idx+kk-ii+jj*_block_size];
        s += aele[idx+kk-ii+jj*_block_size] * u[q];
      }
      v[kk] += s;
    }
}
```

With this reordering of the matrix, we get a speedup up of 10.

7 Result of the Parallel Improvement

We used an example problem for the elasticity problem to test the solver method. In a first step, we calculate OpenMP reference values. For this purpose, we use a *consumer* machine:

```
Intel Core i7-7700 CPU @ 3.60GHz x 4
Nvidia GeForce GTX 1060
```

For a problem with 862515 Nodes we get the following result:

First parallelization (time)						
OpenMP	1 Thr.	2 Thr.	4 Thr.	8 Thr.	OpenACC	+ELLPACK
Jacobi	9.60s	6.22s	5.70s	24.5s	2.80s	2.60s
AMG	2.47s	1.40s	1.15s	4.60s	1.05s	0.46s

To see the scalability, we print the same table as factor representation, wherein we use the solution time of 1 thread as reference.

First parallelization (factor)						
OpenMP	1 Thr.	2 Thr.	4 Thr.	8 Thr.	OpenACC	+ELLPACK
Jacobi	1×	1.6×	1.7×	0.4×	3.39×	3.43×
AMG	1×	1.8×	2.1×	0.5×	2.4×	5.4×

Note that we use a system with **4 cores**, which leads to bandwidth problems for 8 threads.
Previous work on CARP provide a CUDA parallelization for out problem, and we can compare our OpenACC result with a (pure) CUDA code. For the Solution method with Jacobian solver we get a speed up of 8.6×, and with AMG a speed up of 9.2×.
Therefore we miss a factor of 2.5× for Jacobi and 1.7× for AMG. Unfortunately, we can not improve our code further, since this problem is a result of the deep copy operations of OpenACC. (c.f. next sub-section)

7.1 Pointer Handling of OpenACC for Deep Copy's

We have tested the parallel solver for an example solution on the Jouron cluster in Jülich [1] with a Nvidia Pascal card and considered the nvvp[6] output to find the missing *speed up* of our code.

[6] Nvidia GPU visual profiler.

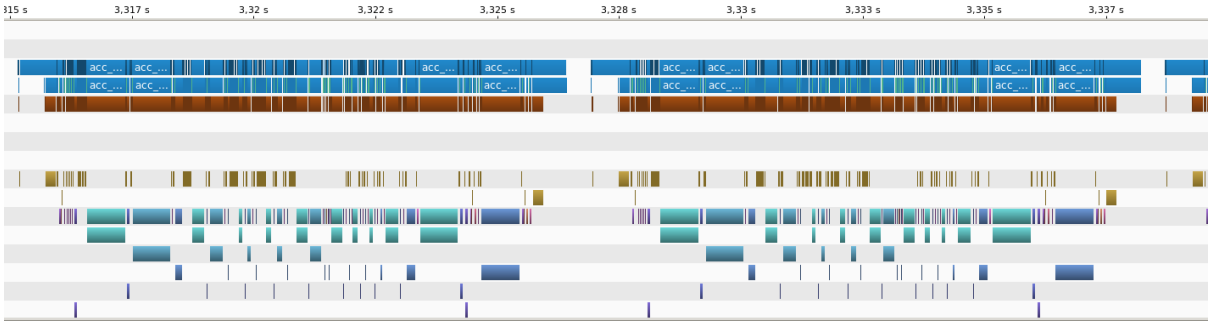


Figure 7.1: Nvvp output of PCG with AMG

We can see in figure 7.1 that nearby every calculation is on the device (first line blue), but we have *big or a lot of* gaps between the device kernels, and (corresponding to the gaps) a lot of data copy operations during the kernel executions (ocher).
We zoomed to a smaller part of the visualisation, and see:

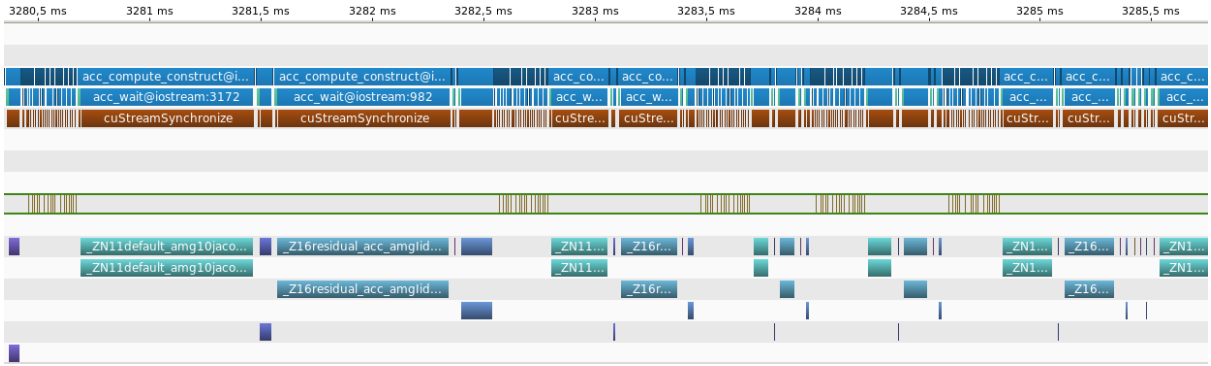


Figure 7.2: Nvvp output of PCG with AMG

We can see a lot of small data transfers. After a discussion with a PGI-OpenACC development member, we've found that this copy operations happen, if OpenACC has to check for existing data on the device. Every transfer has a size of 8 bytes (a pointer). Unfortunately, many of this *checks* tokes longer than the kernels it self.
This behaviour is reported to PGI as *bug report under reference FS24002.*[7]

8 2x2 Model Implementation

The idea for a 2x2 implementation starts with the bidomain equation.
We formulate the problem with the intra and extracellular potentials $u_i(x,t)$ and $u_e(x,t)$. and the transmembrane potential $v(x,t) = u_i(x,t) - u_e(x,t)$. We consider the following problem (c.f. [9]):
Given $I_{\text{app}} : \Omega \times]0,T[\to \mathbb{R}$ and $u_e^0, u_i^0 : \Omega \to \mathbb{R}$, find $u_i, u_e : \Omega \times]0,T[\to \mathbb{R}$ and $v = u_i - u_e$ such that

$$\begin{cases} c_m \partial_t v - \nabla \cdot M_i \nabla u_i = I_{\text{app}} - I_{\text{ion}} & \text{in } \Omega \times]0,T[\\ c_m \partial_t v + \nabla \cdot M_e \nabla u_e = I_{\text{app}} - I_{\text{ion}} & \text{in } \Omega \times]0,T[\\ n^T M_{i,e} \nabla u_{i,e} = 0 & \text{on } \Gamma \times]0,T[\\ v(x,0) = 0 & \text{in } \Omega \end{cases}$$

[7] Unsolved up to now (04.04.2017)

wherein $\Omega \subset \mathbb{R}^2$ models the heart tissue, $\Gamma = \partial\Omega$, n denotes the outward unit normal to the boundary Γ and I_{app} is an applied current used to initiate the process.
The time discretization is performed by a semi-implicit scheme used for the diffusion term the implicit Euler method, while the nonlinear reaction term I_{ion} is treated explicitly. Then the following general algebraic system can be obtained:

$$A\xi^{k+1} = b \qquad \text{with} \qquad A = \begin{pmatrix} C_t + A_i & -C_t \\ -C_t & C_t + A_e \end{pmatrix}$$

$$b = \begin{pmatrix} C_t\, v^k - I^h_{\text{ion}}(v^k) + I^h_{\text{app}} \\ -C_t\, v^k + I^h_{\text{ion}}(v^k) - I^h_{\text{app}} \end{pmatrix}$$

$C_t = \dfrac{c_m}{\tau} C$ diagonal with positive diagonal entries, τ the time step, $v^k = u_i^k - u_e^k$ and $\xi^{k+1} = [u_i^{k+1}; u_e^{k+1}]$.
A is positive semidefinite[8].

8.1 Preparing the Implementation

We consider (in a first step) the discretization of the bidomain equation for one time step Δt. From the Carp-manual we get the equation system:

$$K_{i+e}\,\phi_e = -K_i\, v_m - M\, I_e$$
$$K_i\, v_m = -K_i\, \phi_e + \beta\, I_m$$

wherein M is the mass matrix and K is the stiffness matrix.
Therefore we solve the coupled system

$$A\,u := \begin{pmatrix} K_{i+e} & K_i \\ K_i & K_i - \kappa M_i \end{pmatrix} \begin{pmatrix} \phi_e^{t+1} \\ v_m^{t+1} \end{pmatrix} = \begin{pmatrix} -M_e\, I_e \\ \beta\, I^t_{ion} - \kappa\, M_i\, v_m^t \end{pmatrix}$$

Therefore, we have to solve a 2×2 block matrix. We include the matrix in the following way:

```
for(int ii=0; ii<Ki_cnt.size(); ii++) {
   SRcnt[2*ii] = Kie_cnt[ii] + Ki_cnt[ii];
   SRcnt[2*ii+1] = Ki_cnt[ii] + M_cnt[ii];

   int csize_1 = Kie_cnt[ii];
   int csize_2 = Ki_cnt[ii];
   int csize_3 = M_cnt[ii];

   for(int jj=0; jj<csize_1; jj++, ll++, mm++) {
      SRcol[ll] = Kie_col[mm] * 2;
      SRele[ll] = Kie_ele[mm];
   }
   for(int jj=0; jj<csize_2; jj++, ll++, nn++) {
      SRcol[ll] = Ki_col[nn] * 2 + 1;
      SRele[ll] = Ki_ele[nn];
   }
   for(int jj=0; jj<csize_2; jj++, ll++, oo++) {
      SRcol[ll] = Ki_col[oo] * 2;
      SRele[ll] = Ki_ele[oo];
   }
```

[8] Note, there is a hidden minus!

```
    for(int jj=0; jj<csize_3; jj++, ll++, pp++) {
        SRcol[ll] = Ki_col[pp] * 2 + 1;
        SRele[ll] = Ki_ele[pp] + kappa * M_ele[pp];
    }
}
```

We see for the implementation, that we get a matrix, with block entries

$$\begin{bmatrix} k_{i+e,j} & k_{i,j+1} \\ k_{i,j} & k_{i,j+1} + \kappa m_{i,j+1} \end{bmatrix}$$

wherein j is always an even number. Therefore the solution vector is (has to be) ordered that in every even entry the element correspond to ϕ and en every odd entry to v.
For this matrix we use the already implemented version of the linear operator for the matrix - vector multiplication.

8.2 Jacobi Algorithm

In the first step we implement a *special* form of the CG-algorithm with an jacobian solver.

$$x^{m+1} = D^{-1}\left(b - (L+U)\,x^m\right)$$

Usually, the precondition matrix D is a diagonal matrix. In our case, we consider a special form of D. We define D as follows (with the **source matrix** A) as

$$\left[\begin{array}{cccccccc} a_{11} & a_{12} & 0 & 0 & \dots & & & 0 \\ a_{21} & a_{22} & 0 & 0 & \dots & & & 0 \\ 0 & 0 & a_{33} & a_{34} & 0 & \dots & & 0 \\ 0 & 0 & a_{43} & a_{44} & 0 & \dots & & 0 \\ \vdots & \vdots & 0 & 0 & a_{55} & a_{56} & \dots & 0 \\ & & 0 & 0 & a_{65} & a_{66} & \dots & 0 \\ & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{array}\right]$$

Obvious, it is similar *easy* to invert this matrix as a Diagonal matrix. We construct this matrix with the following function:

```
void setup_block_diagonal(const toolbox_vector<T>& cnt,
      const toolbox_vector<T>& col, const toolbox_vector<S>& ele,
      toolbox_vector<S>& dia, toolbox_vector<S>& dia_det) {
   int size_matrix = cnt.size();
   for (int ii = 0, kk = 0, ll = 0; ii < size_matrix; ii++) {
      int csize = cnt[ii];
      if (ii % 2 == 0) {
         for (int jj = 0; jj < csize; jj++, ll++) {
            if (ii == col[ll])   dia[kk]   = ele[ll];
            if (ii+1 == col[ll]) dia[kk+1] = ele[ll];
         }
      }
      if (ii % 2 == 1) {
         for (int jj = 0; jj < csize; jj++, ll++) {
            if (ii-1 == col[ll]) dia[kk]   = ele[ll];
            if (ii == col[ll])   dia[kk+1] = ele[ll];
         }
      }
```

```
        kk += 2;
    }

    for(int ii=0, jj=0; ii<dia.size(); ii+=4, jj++){
        dia_det[jj] = 1/(dia[ii + 0] * dia[ii + 3] - dia[ii + 1] * dia[ii + 2]);
    }
}
```

With this we implement the Jacobi method as follows:

- The Jacobi solver *operator()*

```
void operator()(const toolbox_vector<S> &_f, toolbox_vector<S> &_u) {
        default_amg::jacobi_2x2<T, S> &J = *_J;

        J.zero_iteration(_f, _u);

        for (int i = 1; i < _nsmooth; i++) {
            J(_f, _u);
        }
    }
```

- The Jacobi 2x2 *operator()*:

```
void operator()(const toolbox_vector<S>& _f, toolbox_vector<S>& _u) {
    const T* __restrict cnt = _cnt.data();
    const T* __restrict col = _col.data();
    const S* __restrict ele = _ele.data();

    const S* dia = _dia.data();
    const S* dia_det = _dia_det.data();

    const S* __restrict f = _f.data();
    S* u = _u.data();
    S* v = _v.data();

    for (int ii = 0; ii < _nsize; ii++) {
        s = f[ii];
        int csize = *cnt++;
        for (int jj = 0; jj < csize; jj++) {
            #ifndef NOSSE
                _mm_prefetch((char*) &pcol[64], _MM_HINT_NTA);
                _mm_prefetch((char*) &pele[64], _MM_HINT_NTA);
            #endif
            T c = *pcol++;
            S t = *pele++;
            s -= t * u[c];
        }
    v[ii] = s;
    }
    _com.accumulate(_v);

    int dsize = _dia_det.size();
    for(int ii=0; ii< dsize; ii++) {
        S t0 = *dia++, t1 = *dia++, t2 = *dia++, t3 = *dia++;
        S td = *dia_det++;
        u[2*ii] += _omega * ( v[2*ii] * t3 - v[2*ii + 1] * t1) * td;
        u[2*ii + 1] += _omega * (-v[2*ii] * t2 + v[2*ii + 1] * t0) * td;
    }
}
```

This source code works for the PT implementation.

8.3 OpenMP Implementation

Now we want to start to improve the implementation. In the first step, we use OpenMP to get a faster *program*.
We use OpenMP to improve the for-loops, therefore, we get the following *modified* loops:

```
#pragma omp parallel for shared(cnt, col, ele, dsp, u, f) private(ii, jj, s, pcol, pele)
    schedule(guided, 2)
for (ii = 0; ii < _nsize; ii++) {

   pcol = col + dsp[ii];
   pele = ele + dsp[ii];

   s = f[ii];
   int csize = cnt[ii]; //*cnt++;
   for (jj = 0; jj < csize; jj++) {
      #ifndef NOSSE
         _mm_prefetch((char*) &pcol[64], _MM_HINT_NTA);
         _mm_prefetch((char*) &pele[64], _MM_HINT_NTA);
      #endif
         T c = *pcol++;
         S t = *pele++;
         s -= t * u[c];
      }
      v[ii] = s;
   }
   _com.accumulate(_v);

   int dsize = _dia_det.size();      // must be 2x2 diagonal block matrix

#pragma omp parallel for shared(u, v, dsize) private(ii) schedule(guided, 2)
for(ii=0; ii< dsize*4; ii += 4) {
   S t0 = dia[ii], t1 = dia[ii+1], t2 = dia[ii+1], t3 = dia[ii+1];
   S td = dia_det[ii/4];
   u[2*ii]   += _omega * ( v[2*ii] * t3 - v[2*ii + 1] * t1) * td;
   u[2*ii + 1] += _omega * (-v[2*ii] * t2 + v[2*ii + 1] * t0) * td;
}
```

8.4 Acceleration Tests

Now, we want to consider different implementations of the parallel sections.

1. First of all, we initialise the loop variables in the loop. That means:

   ```
   #pragma omp parallel for shared(cnt, col, ele, dsp, u, f)
         private(s, pcol, pele) schedule(guided, 2)
   for (int ii = 0; ii < _nsize; ii++) {
   .
   #pragma omp parallel for shared(u, v, dsize) schedule(guided, 2)
   for(int ii=0; ii< dsize*4; ii += 4) {
   .
   }
   ```

2. For the second Test, we consider loops with less private variables:

```
#pragma omp parallel for shared(cnt, col, ele, dsp, u, f)
      private(s) schedule(guided, 2)
for (int ii = 0; ii < _nsize; ii++) {
.
#pragma omp parallel for shared(u, v, dsize) schedule(guided, 2)
for(int ii=0; ii< dsize*4; ii += 4) {
.
}
```

3. For the third test, we exclude the command lines:

```
#ifndef NOSSE
   _mm_prefetch((char*) &pcol[64], _MM_HINT_NTA);
   _mm_prefetch((char*) &pele[64], _MM_HINT_NTA);
#endif
```

4. For the last Test, we consider the *full* parallel loops:

```
#pragma omp parallel for shared(cnt, col, ele, dsp, u, f)
           private(ii, jj, s, pcol, pele) schedule(guided, 2)
for (ii = 0; ii < _nsize; ii++) {
.
#pragma omp parallel for shared(u, v, dsize) private(ii) schedule(guided, 2)
for(ii=0; ii< dsize*4; ii += 4) {
.
}
```

We consider 100 runs, and get the following calculation times:

1. 10.26 s
2. 11.00 s
3. 10.09 s
4. 10.08 s

The second test becomes slower, since we didn't declare the pointers to *col* and *ele* as private members.[9]
Moreover, we can see (test 3) that *prefetching* is for this code useless.

9 AMG 2x2 Functions

We summarize the core functions for the AMG 2x2 method!

9.1 AMG Setup

For the solution of the bidomain equation (c.f. Note 15-11) we use the CG-method and define the (required) inverse operator as the AMG - calculation. The class *amg_solver_2x2* describes this inverse operator.
We want to focus on the *heart - part* of the code:

[9] Note, that the compiler produced a correct program, but slower.

```
while ((l < _max_level - 1) && (glb > _min_nodes) && (rel < _rel)) {
   // construct the next amg level
   amg_restriction_2x2(_epsilon, _anod[l], _acnt[l], _acol[l],
      _aele[l], _adia[l], _com[l], _rcnt[l], _rcol[l], _rele[l],
      _anod[l + 1], _acnt[l + 1], _acol[l + 1], _aele[l + 1],
      PT_COMM_WORLD, _com[l + 1]);
   l++;  // Now we just calculate in the last level

   size = (int) _acnt[l].size();
   _u[l].resize(size, 0.0);
   _f[l].resize(size, 0.0);
   _s[l].resize(size, 0.0);
   _r[l].resize(size, 0.0);
   _v[l].resize(size, 0.0);

   old = glb;
   get_level_sizes(l, buff, min, max, glb, avg_nnz, _show);

   rel = (float) glb / (float) old;

   amg_cplx += avg_nnz;
   last_nnz = avg_nnz;
}

_level = l;

// Calculate the inverse operator for the last grid
parallel_setup_2x2<T, S> setup;

setup.extract_2x2_diagonal(_acnt[l], _acol[l], _aele[l], _block_size, _adia[l]);
_inv = new block_diagonal_2x2_solver<T, S>(_adia[l], _com[l]);
```

This loop create the AMG-coarsening nodes for every level l. In the last level, it build the inverse block-diagonal for the nodes which remain in the last level (*block_diagonal_2x2_solver*). This inverse operator is used as the precondition matrix for the Jacobi method.

9.2 AMG Restriction

The AMG restriction produce the coarse nodes and the interpolation matrix between the AMG levels. For this purpose, the function creates the parallel setup and the triple product $P^T AP$.

```
parallel_setup_2x2<T, S> setup;  // construct AMG-setup
setup(_anod, _acnt, _acol, _aele, _adia, _acom, _mcnt, _mcol, _mele, _cnod,
      _mpi_com, _ccom, _epsilon);
new_triple_product<T, S> triple_product;
triple_product(_acnt, _acol, _aele, _mcnt, _mcol, _mele, _cnod, _ccnt, _ccol, _cele);
```

9.3 AMG Parallel Setup

The parallel Setup creates the coarse nodes. We consider the **main** steps in the algorithm:

```
.
.
.
   extract_norm_matrix(_anod, _acnt, adsp, _acol, _aele, _block_size, _acom, enod,
      srt_enod, ecnt, ecol, eele);
.
```

```
.
.
   for (int ii = 0; ii < enodes; ii++) {
      if (esel[ii] == 0) {
         esel[ii] = 1;   //coarse node
         const T dsp = edsp[ii], end = dsp + ecnt[ii];
         for (int jj = dsp; jj < end; jj++) {
            const T col = ecol[jj];
            if (eele[jj] * eele[jj] >= edia[ii] * edia[col] * epsilon) {
               ecox[jj] = 1;
               if (esel[col] == 0) esel[col] = 2; // fine node
               epcnt[col]++;
            }
         }
      }
   }
.
.
.
   for (int ii = 0; ii < enodes; ii++) {
      if (esel[ii] == 1) {
         const T dsp = edsp[ii], end = dsp + ecnt[ii];
         const T nod = enod[ii];
         for (int jj = dsp; jj < end; jj++) {
            if (ecox[jj] != 0) {
               epcol[epdsp[ecol[jj]]++] = nod;
            }
         }
         epnod[epnodes] = nod;
         epnodes++;
      }
   }
.
.
.
   {
      toolbox_vector<T> perm(epsize);
      for (int ii = 0; ii < epsize; ii++) perm[ii] = ii;
         const toolbox_vector<T> p = _ccom.reorder_map();
         binary_sort_copy(epcol, perm);
         int lastidx = epnod.size() - 1;
         for (int ii = 0, idx = 0; ii < epsize; ii++) {
            while (epnod[idx] < epcol[ii] && idx < lastidx) idx++;
            epcol[ii] = p[idx];
         }
      binary_sort_copy(perm, epcol);
   }
.
.
.
   for (int ii = 0, end = epnod.size(); ii < end; ii++) {
      for (int jj = 0; jj < _block_size; jj++) {
         _pnod[_block_size * ii + jj] = epnod[ii] * _block_size + jj;
      }
   }

   for (int ii = 0, end = epcnt.size(); ii < end; ii++) {
      for (int jj = 0; jj < _block_size; jj++) {
         _pcnt[_block_size * ii + jj] = epcnt[ii];
```

```
    }
  }

  for (int ii = 0, end = epcnt.size(), index_P = 0, index_E = 0; ii < end; ii++) {
    const S epval = 1.0 / S(epcnt[ii]);

    for (int kk = 0; kk < _block_size; kk++) {
      for (int jj = 0; jj < epcnt[ii]; jj++) {
        _pcol[index_P] = epcol[index_E] * _block_size + kk;
        _pele[index_P] = epval;
        index_P++;
        index_E++;
      }
      index_E -= epcnt[ii];
    }
    index_E += epcnt[ii];
  }
```

9.3.1 The Norm matrix

We create in a first step the *norm matrix* with the class *extract_norm_matrix*. We consider for this an initial condition matrix a *sparse* block matrix, like:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & a_{15} & a_{16} & \dots & 0 \\ a_{21} & a_{22} & 0 & 0 & a_{25} & a_{26} & \dots & 0 \\ 0 & 0 & a_{33} & a_{34} & 0 & \dots & & 0 \\ 0 & 0 & a_{43} & a_{44} & 0 & \dots & & 0 \\ a_{51} & a_{52} & 0 & 0 & a_{55} & a_{56} & \dots & 0 \\ a_{61} & a_{62} & 0 & 0 & a_{65} & a_{66} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

For this matrix (should) hold $A = A^T$. The entries in this matrix have always 2×2 *character*. For the norm, we consider the Frobenius norm, restricted to a 2×2 block:

$$\begin{bmatrix} a_{k,l} & a_{k,l+1} \\ a_{k+1,l} & a_{k+1,l+1} \end{bmatrix} \quad \longrightarrow \quad e_{\frac{k+1}{2},\frac{l+1}{2}} = \sqrt{a_{k,l}^2 + a_{k,l+1}^2 + a_{k+1,l}^2 + a_{k+1,l+1}}$$

With this we get the norm matrix as

$$E = \begin{bmatrix} e_{11} & 0 & 0 & e_{13} & \dots & 0 \\ 0 & e_{22} & 0 & 0 & & 0 \\ e_{31} & 0 & 0 & e_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

We use this matrix E to find the coarse nodes for the AMG algorithm.

9.3.2 Coarsening with Mandel-Brezina

In the next step, we calculate the *strong connection* between the nodes.
For this purpose, we give **all** entries in E a *bool* variable if the node is strong or not. We use the following criteria:

$$e_{k,l}^2 \geq e_{k,k} \cdot e_{l,l} \cdot \varepsilon \tag{9.1}$$

for a constant $\varepsilon \in [0,1]$.
For the selection, we consider all nodes in E. The first node is **always** a coarse node.
For every line (node) we compare the entries with the Mandel-Brezina criteria (9.1), and set the node **coarse - connected** if the criteria is true.

9.3.3 Create the Temporary Interpolation matrix

In the next step, we construct a temporary interpolation matrix with the *coarsening information*.

> ☞ In several parts of the code, we find for the displacement information (usually dsp) that this value changes during the code. Why?
> **Answer:** It happens that this variable is used as a *run-index*, which counts *up* in the loop. Usually, just double check if the function returns the variable or not!

The counting value (*cnt*) is created during the node selection. The next loop, iterates over all coarse nodes and set the *corresponding* nod number.

9.3.4 Recalculation of the Column Entries

In the next *closed* section, we recalculate the node numbers, since if we use the *old* values, we will get empty rows (and columns) and therefore a matrix with no inverse matrix.

9.3.5 Creation of the Interpolation Matrix

In the last three loops we calculate the interpolation Matrix. For this purpose, we have to *rescale* the information of the coarsening.

> ☞ Why do we use for P the **exact** same number of counts as in the temporary interpolation matrix EP. I would expect we have to rescale the value with a factor of *_block_size*!?! (*_pcnt[_block_size * ii + jj] = epcnt[ii];*)
> **Answer:** Because it **must** be the same number of counts. The implementation has to be corrected. (c.f. following Sections).

9.4 Triple Product

We remain the triple product as it was from the original code.

9.5 Inverse Function

The inverse operator is represented with the function *block_diagonal_2x2_solver*.
The constructor produce a 2×2 inverse block diagonal matrix, wherein all 4 entries of the of a block are saved in a sequence.

```
block_diagonal_2x2_solver(const toolbox_vector<S> &adia, accumulator<T, S> &com) {
   _adia.resize(adia.size());
   for (int ii = 0; ii < adia.size(); ii += 4) {
      const S inv_det = 1 / (adia[ii + 0] * adia[ii + 3] - adia[ii + 1] * adia[ii + 2]);
      _adia[ii + 0] = adia[ii + 3] * inv_det;
      _adia[ii + 1] = -adia[ii + 1] * inv_det;
      _adia[ii + 2] = -adia[ii + 2] * inv_det;
      _adia[ii + 3] = adia[ii + 0] * inv_det;
```

```
    }
}
```

Note that in the original call of the *restriction* method, R denotes the interpolation matrix for the corresponding level.

10 Application of the Multigrid Method

During the CG-Method, we call the inverse function (the 2×2 AMG method). We use the following sequence:

```
void operator()(const toolbox_vector<S> &_f0, toolbox_vector<S> &_u0) {
   int dim = _f0.size();

   _f[0].assign(_f0.data(), _f0.data() + dim);
   multigrid(0);  // Start multigrid with the finest grid
   _u0.assign(_u[0].data(), _u[0].data() + dim);
}
```

This function returns the solution $u0$. We use the *multigrid* function to calculate the vector.

```
void multigrid(int _k) {
   int l = _k++;
   if (l < _level) {
      // Constructor, needs column and row definition of the Matrix
      simple_prolongation<T, S> _P(_rcnt[l], _rcol[l]);

      // Constructor, needs column and row definition of the Matrix
      simple_restriction<T, S> _R(_rcnt[l], _rcol[l]);

      // Construct jacobi methode
      jacobi_2x2<T, S> _J(_acnt[l], _acol[l], _aele[l], _adia[l], _com[l], _v[l],
            _omega);

      // Calculate the component of Jacobi iteration with the right
      // hand side for the first Jacobi preconditioning step
      _J.zero_iteration(_f[l], _u[l]);

      // Calculate _u with the Jacobi methode
      for (int ii = 1; ii < _nsmooth; ii++) _J(_f[l], _u[l]);

      // Return the residual r = f - A*u
      _J.compute_residuum(_f[l], _u[l], _r[l]);

      // Calculate the sum of the error (on all neighbourhood nodes) for the next level
      // AND write it into _f
      _R(_r[l], _f[l + 1]);

      // Call the function until reach the highest level of multigrid, and calculate
      // the corresponding residual
      multigrid(_k);

      // Calculate the average of the error-solution and distribute it to _s
      _P(_u[l + 1], _s[l]);

      // Add _s to _u
      add(_s[l], _u[l]);
```

```
        // Calculate (smooth) _u with the Jacobi Methode
        for (int ii = 0; ii < _nsmooth; ii++) _J(_f[l], _u[l]);

    } else {
        // In the last level of grids, (A^{-1}f = u
        (*_inv)(_f[l], _u[l]);
    }
}
```

The *simple_prolongation* and *simple_restriction* methods calculate the values between the different levels.
Furthermore, we need the Jacobian method to solve the inverse problem.
For our problem we have the following sequence:

- Calculate the residuum r on the current level l.
- Restrict the residuum to the next level $l+1$.
- Go to the next level of the AMG method.

Until we get to the most coarse grid level.
In this level we use the inverse of the block diagonal (constructed in the parallel setup) and calculate the inverse solution of $A_{\text{diag}}^{-1} f = u$.
After this, we distribute the values to the finer grids with the following sequence:

- Prolong the solution at level $l+1$ to the current level.
- Smooth the solution u at the current level with the Jacobi method.

With this we can calculate the solution u.

10.1 Prolongation Operator

Note that we define our *prolongation operator* with blocks as

$$P_{i,j} = \begin{cases} \begin{pmatrix} \frac{1}{n_i} & 0 \\ 0 & \frac{1}{n_i} \end{pmatrix} & \text{if Node is coarse} \\ \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \text{else} \end{cases}$$

wherein n_i is the number of strong connected notes to node i.

11 The 2x2 AMG Code with OpenACC and OpenMP Parallelization

We want to summarise the important steps for the OpenACC parallelization:

11.1 Construction of the Solver

```
#if defined(OPENMP) || defined(_OPENACC)
    amg_solver_2x2<int, double> amg(nod, cnt, col, ele, settings);
#else
```

One can call the same constructor for the OpenMP parallelization and for the OpenACC parallelization.

☞ It is **NOT** possible to use OpenMP and OpenACC simultaneously!
But there is a OpenACC flag *-ta=multicore*, which is basically a combination of OpenACC and OpenMP (c.f. [3, page 51]).

The constructor requires a matrix in CRS-format (*cnt, col, ele*), the node information *nod* and general settings *settings*.
The constructor itself is (more or less) similar to the normal constructor. The only thing we have to take care of is, when we copy our data to the device for the OpenACC part. For this purpose we define copying functions for toolbox_vectors:

```
void todev(){ // move to device
   #pragma acc enter data pcopyin(this[0:1], _data[0:_size])
}
void fromdev(){ // remove from device
   #pragma acc exit data delete( _data[0:_size], this[0:1])
}
void updatehost(){ // update host copy of data
   #pragma acc update self( _data[0:_size] )
}
void updatedev(){ // update device copy of data
   #pragma acc update device( _data[0:_size] )
}
```

In the setup of the *amg_solver_2x2* we have to implement for every level *l*:

```
// Matrix data
   _acnt[l].todev();
   _acol[l].todev();
   _aele[l].todev();
   _adia[l].todev();

// Calculation data
   _u[l].todev();
   _f[l].todev();
   _s[l].todev();
   _r[l].todev();
   _v[l].todev();

   _rdsp[l].todev();
   _scnt[l].todev();
   _sdsp[l].todev();
   _scol[l].todev();
   _sele[l].todev();
```

We calculate the setup it self on the host.
Moreover, we copy the inverse operator *block_diagonal_2x2_solver* (for the last level *l*) also to the device.

11.2 Solving Step

After the construction, we call the inverse operator (AMG) during the CG algorithm. The CG method it self has no parallelization. The solve operator (AMG-method) has the structure:

```
void operator()(const toolbox_vector<S> &_f0, toolbox_vector<S> &_u0) {
      int dim = _f0.size();

      _f[0].assign(_f0.data(), _f0.data() + dim);
```

```
#ifdef _OPENACC
      _f[0].updatedev();   // Copy initial data to the device
#endif
      multigrid(0);  // Start multigrid with the finest grid

#ifdef _OPENACC
      _u[0].updatehost();  // Copy solution data to the host
#endif
      _u0.assign(_u[0].data(), _u[0].data() + dim);
   }
```

In this step it is important to note, that for an efficient calculation, we have to copy the data in *_f0* to the device at the beginning, and we have to copy the solution *_u0* back to the host at the end of the calculation.

11.2.1 MULTIGRID

Now, we consider the multigrid step. We wrote the solve functions such that we can use OpenMP or OpenACC for the same function. It is important to note, that it is not possible to use both. In particular, for OpenACC we copy the data already at the construction step to the device.

```
void multigrid(int _k) {
   int l = _k++;
   if (l < _level) {
//********************************** Constructors ***********************************

#if defined(OPENMP) || defined(_OPENACC)
      simple_prolongation<T, S> _P(_rcnt[l], _rdsp[l], _rcol[l]);
      linear_operator<T, S> _R(_scnt[l], _sdsp[l], _scol[l], _sele[l]);
#else
      simple_prolongation<T, S> _P(_rcnt[l], _rcol[l]);
      simple_restriction<T, S> _R(_rcnt[l], _rcol[l]);
#endif
//********************************* Calculate solutions *****************************
// apply jacobi methode
#if defined(OPENMP) || defined(_OPENACC)
      jacobi_2x2<T, S> _J(_acnt[l], _acol[l], _aele[l], _adsp[l],
            _adia[l], _com[l], _v[l], _omega);
#else
      jacobi_2x2<T, S> _J(_acnt[l], _acol[l], _aele[l],
            _adia[l], _com[l], _v[l], _omega);
#endif
      int ii;
#ifdef _OPENACC
      const T*__restrict pt_col = _acol[l].data();
      const S*__restrict pt_ele = _aele[l].data();
      const T*__restrict pt_cnt = _acnt[l].data();
      const S*__restrict pt_f = _f[l].data();
      const S*__restrict pt_u = _u[l].data();

// Copy Matrix data to the device
#pragma acc data pcopyin(pt_col[0:_acol[l].size()], pt_ele[0:_aele[l].size()],\
          pt_cnt[0:_acnt[l].size()], pt_f[0:_f[l].size()], pt_u[0:_u[l].size()])
      {
#endif
// Calculate the component of Jacobi iteration with the right hand side for the
// first Jacobi preconditioning step
         _J.zero_iteration(_f[l], _u[l]);
```

```
// Calculate _u with the Jacobi methode
        for (ii = 1; ii < _nsmooth; ii++) _J(_f[l], _u[l]);

// Return the residual r = f - A*u
        _J.compute_residuum(_f[l], _u[l], _r[l]);

// Calculate the sum of the error (on all neighbourhood nodes) for the next level
// AND write it into _f
        _R(_r[l], _f[l + 1]);

// Call the function until reach the highest level of multigrid
        multigrid(_k);

// Calculate the average of the error-solution and distribute it to _s

        _P(_u[l + 1], _s[l]);

// Add _s to _u
        add(_s[l], _u[l]);

// Calculate (smooth) _u with the Jacobi Methode
        for (int ii = 0; ii < _nsmooth; ii++) _J(_f[l], _u[l]);
     }

  } else {
// In the last level of grids, (A^{-1}f = u
     (*_inv)(_f[l], _u[l]);
  }
}
```

- We do not apply any parallelization information for the constructors of the operators *_P* and *_R*.
- Note that the parallelization is done in the functions.

11.2.2 Parallelization of the Jacobi Functions

zero_iteration:

```
void zero_iteration(const toolbox_vector<S> &_f, toolbox_vector<S> &_u) {

   const S * dia = _dia.data();
   const S * f = _f.data();
   S * u = _u.data();
   const int dsize = _dia.size() / 4;     // must be 2x2 diagonal block matrix

#ifdef OPENMP
#pragma omp parallel for default(none) shared(u, f, dia) schedule(guided, 2)
   for (int ii = 0; ii < dsize; ii++) {
      const S t0 = dia[4*ii];
      const S t1 = dia[4*ii+1];
      const S t2 = dia[4*ii+2];
      const S t3 = dia[4*ii+3];

      u[2 * ii] = _omega * (t3 * f[2 * ii] - t1 * f[2 * ii + 1]);
      u[2 * ii + 1] = _omega * (-t2 * f[2 * ii] + t0 * f[2 * ii + 1]);
   }
```

```
#elif _OPENACC
#pragma acc data pcopyin(f[0:_f.size()], dia[0:_dia.size()]) pcopy(u[0:_u.size()])
   {
      int ii;
#pragma acc loop private(ii)
      for (ii = 0; ii < dsize; ii++) {
         u[2*ii + 0] = _omega*(  f[2*ii] * dia[4*ii + 3] - f[2*ii + 1] * dia[4*ii + 1]);
         u[2*ii + 1] = _omega*(- f[2*ii] * dia[4*ii + 2] + f[2*ii + 1] * dia[4*ii + 0]);
      }
   }
#else
   for (int ii = 0; ii < dsize; ii++) {
      S t0 = *dia++, t1 = *dia++, t2 = *dia++, t3 = *dia++;

      u[2 * ii] = _omega * (t3 * f[2 * ii] - t1 * f[2 * ii + 1]);
      u[2 * ii + 1] = _omega * (-t2 * f[2 * ii] + t0 * f[2 * ii + 1]);
   }
#endif
   _com.accumulate(_u);
}
```

- The only thing to note for this part of the code is, that we have to be careful with pointers. In the sequential code, it is possible to call the values of *dia* with pointer arithmetic. Obviously, this is very dangerous for parallel codes.

operator()

```
void operator()(const toolbox_vector<S>& _f, toolbox_vector<S>& _u) {

   const T* __restrict col = _col.data();
   const S* __restrict ele = _ele.data();
   const S* __restrict dia = _dia.data();

   const S* __restrict f = _f.data();
   const S* uu = _u.data();

   S* u = _u.data();
   S* v = _v.data();

// ********************************************************************************
#if defined(OPENMP) || defined(_OPENACC)
   const T* __restrict cnt = _cnt.data();
   const T *__restrict dsp = _dsp.data();
   const T *__restrict pcol;
   const S *__restrict pele;
   int ii, jj;    // Define run variables outside of the pragma's

   T c;
   S t, s;

#ifdef OPENMP
#pragma omp parallel for default(none) shared(cnt, col, ele, dsp, uu, v, f)\
    private(ii, jj, s, c, t, pcol, pele) schedule(guided, 2)
   for (ii = 0; ii < _nsize; ii++) {
      s = f[ii];
      // We have to be careful with shared pointers col, since we calculate parallel
      pcol = col + dsp[ii];
      pele = ele + dsp[ii];
```

```
      // First part of the matrix
      const T csize = cnt[ii];
      for (jj = 0; jj < csize; jj++) {
#if !defined(NOSSE) && !defined(_OPENACC)
         _mm_prefetch((char*) &pcol[64], _MM_HINT_NTA);
         _mm_prefetch((char*) &pele[64], _MM_HINT_NTA);
#endif
         c = pcol[jj];
         t = pele[jj];

         s -= t * uu[c];
      }
      v[ii] = s;
   }
#elif OPENACC
#pragma acc loop private(ii, jj)
   for (ii = 0; ii < _nsize; ii++) {
      s = f[ii];
      // We have to be careful with shared pointers col, since we calculate parallel
      pcol = col + dsp[ii];
      pele = ele + dsp[ii];

      // First part of the matrix
      const T csize = cnt[ii];

#pragma acc loop // reduction(-:s)
      for (jj = 0; jj < csize; jj++) {
         c = pcol[jj];
         t = pele[jj];

         s -= t * uu[c];
      }
      v[ii] = s;
   }
#else
   for (int ii = 0; ii < _nsize; ii++) {
      S s = f[ii];

      // First part of the matrix
      const int csize = _cnt[ii];
      for (int jj = 0; jj < csize; jj++) {
#ifndef NOSSE
         _mm_prefetch((char*) &col[64], _MM_HINT_NTA);
         _mm_prefetch((char*) &ele[64], _MM_HINT_NTA);
#endif
         T c = *col++;
         S t = *ele++;

         s -= t * uu[c];
      }
      v[ii] = s;
   }
#endif
#endif

//*****************************************************************************
   _com.accumulate(_v);
   const int dsize = _dia.size() / 4;     // must be 2x2 diagonal block matrix
```

```
#ifdef OPENMP
#pragma omp parallel for default(none) shared(u, v, dia) schedule(guided, 2)
   for (ii = 0; ii < dsize; ii++) {
      const S t0 = dia[4*ii];
      const S t1 = dia[4*ii+1];
      const S t2 = dia[4*ii+2];
      const S t3 = dia[4*ii+3];

      u[2 * ii] += _omega * (v[2 * ii] * t3 - v[2 * ii + 1] * t1);
      u[2 * ii + 1] += _omega * (-v[2 * ii] * t2 + v[2 * ii + 1] * t0);
   }
#elif _OPENACC
#pragma acc data pcopyin(dia[0:_dia.size()], v[0:_v.size()]) pcopy(u[0:_u.size()])
#pragma acc loop private(ii)
   for(ii = 0; ii < dsize; ii++) {
      u[2*ii + 0] += _omega * ( v[2*ii] * dia[4*ii + 3] - v[2*ii + 1] * dia[4*ii + 1]);
      u[2*ii + 1] += _omega * (-v[2*ii] * dia[4*ii + 2] + v[2*ii + 1] * dia[4*ii + 0]);
   }

#else
   for (int ii = 0; ii < dsize; ii++) {
      S t0 = *dia++, t1 = *dia++, t2 = *dia++, t3 = *dia++;
      u[2 * ii] += _omega * (v[2 * ii] * t3 - v[2 * ii + 1] * t1);
      u[2 * ii + 1] += _omega * (-v[2 * ii] * t2 + v[2 * ii + 1] * t0);
   }
#endif
//*****************************************************************************
}
```

- The operator solves explicit by the Jacobi method $x^{m+1} = D^{-1}(b-(L+U)x^m)$, wherein we use D as a 2×2 block diagonal matrix, after the zero iteration $(D^{-1}b)$ c.f. Note 15-11.
- Note that we need pointers for the data copy *pragmas* in OpenACC.
- The loop parallelization is a main topic of OpenACC, therefore we focus on this parts of the code.
- The main difference between OpenMP and OpenACC.
- For our *simple* loops, OpenACC detects automatically the reduction operator. Therefore, we did not need this clauses.

Calculate the residuum:

```
void compute_residuum(const toolbox_vector<S> &_f, const toolbox_vector<S> &_u,
      toolbox_vector<S> &_r) {

#if defined(OPENMP) || defined(_OPENACC)
   const T *__restrict cnt = _cnt.data();
   const T *__restrict col = _col.data();
   const S *__restrict ele = _ele.data();
   const S *__restrict f = _f.data();
   const S *__restrict u = _u.data();
   S *__restrict r = _r.data();

   int ii, jj;
   S s;
   const T *__restrict pcol;
   const S *__restrict pele;
```

```
   const T *__restrict dsp = _dsp.data();
#ifdef OPENMP
#pragma omp parallel for shared(cnt, col, ele, dsp, u, r, f) private(ii, jj, s,\
      pcol, pele) schedule(guided, 2)
#elif _OPENACC
#pragma acc data pcopyin(col[0:_col.size()], dsp[0:_dsp.size()], ele[0:_ele.size()],\
       f[0:_f.size()], u[0:_u.size()]) pcopy(r[0:_r.size()])
#pragma acc loop private(ii, jj, s, pcol, pele)
#endif
   for (ii = 0; ii < _nsize; ii++)
   {
      s = f[ii];
      pcol = col + dsp[ii];
      pele = ele + dsp[ii];

      int csize = cnt[ii];
      for(jj = 0; jj < csize; jj++) {
#if !defined(NOSSE) && !defined(_OPENACC)
         _mm_prefetch((char*) &pcol[64], _MM_HINT_NTA);
         _mm_prefetch((char*) &pele[64], _MM_HINT_NTA);
#endif
         T c = pcol[jj];
         S t = pele[jj];
         s -= t * u[c];
      }
      r[ii] = s;
   }
#else
      // use the residual routine from algorithm.h
      residual(_cnt, _col, _ele, _f, _u, _r);
#endif
}
```

- The function works only with references and pointers (necessary for a efficient code).
- Similar to the operator() we can use OpenMP or OpenACC. But we have to take care of the place of the data.

11.2.3 Other applied Functions

First of all to note: **I did not choose the variables**. In particular I copied the variables *_r...* for the **P**rolongation operator and *_s...* for the **R**estriction Operator.

Restriction Operator *_R*:

```
void operator()(const toolbox_vector<S> &_u, toolbox_vector<S> &_v) const
{
   // acol is the column index
   const T *__restrict acnt = _acnt.data();
   const T *__restrict acol = _acol.data();
   const S *__restrict aele = _aele.data();
   const S *__restrict u = _u.data();
   S *__restrict v = _v.data();

   int size = (int)_acnt.size();

#if defined(OPENMP) || defined(_OPENACC)
   int ii, jj, csize;
```

```
   const T *__restrict p_col;
   const S *__restrict p_ele;
   S s;
   const T *__restrict adsp = _adsp.data();
#ifdef OPENMP
#pragma omp parallel for shared(acnt, aele, adsp, u, v, size) private(ii, jj, csize, s,\
       p_col, p_ele) schedule(guided)
#endif
#ifdef _OPENACC
#pragma acc data pcopyin(acnt[0:_acnt.size()], acol[0:_acol.size()], aele[0:_aele.size()
    ],\
       u[0:_u.size()]) pcopy(v[0:_v.size()])
#pragma acc loop private(ii, jj, csize, s, p_col, p_ele)
#endif
   for(ii = 0; ii < size; ii++)
   {
      s = S(0.0);                                  // Set the result value zero
      p_col = acol + adsp[ii];
      p_ele = aele + adsp[ii];
      csize = acnt[ii];
      for(jj = 0; jj < csize; jj++)
      {
#if !defined(NOSSE) && !defined(_OPENACC)
         _mm_prefetch((char*)&p_col[64],_MM_HINT_NTA);
         _mm_prefetch((char*)&p_ele[64],_MM_HINT_NTA);
#endif
         T q = p_col[jj];
         S a = p_ele[jj];
         s += a * u[q];
      }
      v[ii] = s;
   }
#else
.
.
.
#endif
}
```

- The restriction operator is in this case implemented as a linear operator. We transpose and construct the elements already during the construction of the AMG-method.

Prolongation Operator _*P*:

```
void operator()(const toolbox_vector<S> &_u, toolbox_vector<S> &_s) const
{
   const S *__restrict u = _u.data();
   S *__restrict s = _s.data();
   int size = _rcnt.size();
   const T *__restrict rcnt = _rcnt.data();
   const T *__restrict rcol = _rcol.data();
   S t;

#if defined(OPENMP) || defined(_OPENACC)
   const T *__restrict rdsp = _rdsp.data();
   const T *__restrict pcol;
   int ii, jj;
#ifdef OPENMP
#pragma omp parallel for shared(rcol, rcnt, rdsp, u, s, size) private(ii, jj, t, pcol)\
```

```
 schedule(guided)
   for(ii = 0; ii < size; ii++)
   {
      pcol = rcol + rdsp[ii];

#if !defined(NOSSE) && !defined(_OPENACC)
      _mm_prefetch((char*)&pcol[64],_MM_HINT_NTA);
#endif

      const int c = rcnt[ii];

      switch(c)
      {
      case 0:
         //cout << "*";
         break;
      case 1:
         s[ii] = u[pcol[0]];
         break;
      case 2:
         t = u[pcol[0]];
         t += u[pcol[1]];
         s[ii] = t / S(2.0);
         break;
      case 3:
         t = u[pcol[0]];
         t += u[pcol[1]];
         t += u[pcol[2]];
         s[ii] = t / S(3.0);
         break;
      case 4:
         t = u[pcol[0]];
         t += u[pcol[1]];
         t += u[pcol[2]];
         t += u[pcol[3]];
         s[ii] = t / S(4.0);
         break;
      default:
         t = S(0.0);
         for(jj = 0; jj < c; jj++)
         {
            t += u[pcol[jj]];
         }
         s[ii] = t / S(c);
         break;
      }
   }
#endif
#ifdef _OPENACC
#pragma acc data pcopyin(rcol[0:_rcol.size()], rcnt[0:_rcnt.size()], rdsp[0:_rdsp.size()
    ],\
 u[0:_u.size()]) pcopy(s[0:_s.size()])
#pragma acc loop private(ii, jj, pcol, t)
   for(ii = 0; ii < size; ii++)
   {
      pcol = rcol + rdsp[ii];
      const int c = rcnt[ii];

      t = S(0.0);
```

```
#pragma acc loop
      for(jj = 0; jj < c; jj++)
      {
         t += u[pcol[jj]];
      }
      s[ii] = t / S(c);
   }
#endif
#else
.
.
.
#endif
}
```

- For the prolongation operator we have to distinguish between OpenMP and OpenACC. It is not efficient for OpenACC to split the algorithm with *cases*. The default parallelization of the compiler is much better.

Add function:

```
void add(const toolbox_vector<S> &_a, toolbox_vector<S> &_b) {
#ifdef OPENMP
   const S *__restrict a = _a.data();
   S *__restrict b = _b.data();
   int i, size = _a.size();

#pragma omp parallel for shared(a, b) private(i) schedule(guided, 2)
   for(i = 0; i < size; i++)
   {
      b[i] += a[i];
   }
#elif _OPENACC
   const S *__restrict a = _a.data();
   S *__restrict b = _b.data();
   int ii, size = _a.size();

#pragma acc data pcopyin(a[0:_a.size()], b[0:_b.size()])
#pragma acc loop private(ii)
   for(ii = 0; ii < size; ii++)
   {
      b[ii] += a[ii];
   }
#else
   const S *__restrict a = _a.data(), *__restrict end = _b.end();
   S *__restrict b = _b.data();

   while (b != end)
   *b++ += *a++;
#endif
   }
```

- Standard parallelization.

Inverse Function:

```
void operator()(const toolbox_vector<S> &_u, toolbox_vector<S> &_v) {
   const S *__restrict u = _u.data();
```

```
   const S *__restrict adia = _adia.data();
   S *__restrict v = _v.data();

#ifdef _OPENACC
   const int nsize = _u.size() / 2;
   int ii;
#pragma acc data pcopyin(adia[0:_adia.size()], u[0:_u.size()], v[0:_v.size()])
   {
#pragma acc loop private(ii)
      for (ii = 0; ii < nsize; ii++) {
         v[2*ii]     = adia[4*ii + 0] * u[2*ii] + adia[4*ii + 1] * u[2*ii + 1];
         v[2*ii + 1]    = adia[4*ii + 2] * u[2*ii] + adia[4*ii + 3] * u[2*ii + 1];
      }
   }
#else
   const int nsize = _u.size() / 2;
   for (int ii = 0; ii < nsize; ii++) {
      const S t0 = *adia++, t1 = *adia++, t2 = *adia++, t3 = *adia++;
      const S u0 = *u++, u1 = *u++;
      v[2*ii]     = t0 * u0 + t1 * u1;
      v[2*ii + 1]    = t2 * u0 + t3 * u1;
   }
#endif
}
```

- Note that we copied the element *_adia* in the constructor of the class. Therefore we have to be careful with update functions, we have to update this vector too.

REFERENCES

[1] Julia und juron: Pilotsysteme fÃijr einen interaktiven superrechner, 2017. [Online; accessed 04-04-2017].

[2] CMSoft. Optimization strategies, 2017. [Online; accessed 23-02-2017].

[3] Rob Farber. *Parallel Programming with OpenACC*. Elsevier / Morgan kaufmann, 2017.

[4] Sparse Matrix Computation. FastSparse. Fastsparse based on ellpack-r, 2017. [Online; accessed 23-02-2017].

[5] Admin New Day HPC. Openmp, 2017. [Online; accessed 22-02-2017].

[6] NVIDIA Jeff Larkin. Getting started with openacc, 2015. [Online; accessed 05-June-2015].

[7] Jeff Larkin. Introduction to openacc, 2017. [Online; accessed 27-02-2017].

[8] Microway. Accelerating code with openacc and the nvidia visual profiler, 2017. [Online; accessed 22-02-2017].

[9] B. J. Roth. Bidomain model. *Scholarpedia*, 3(4):6221, 2008.

[10] F. Vazquez, E. M. Garzon, J.A. Martinez, and J. J. Fernandez. The sparse matrix vector product on gpus. *Dpt Computer Architecture and Electronics*, 2009.