SFB Research Center
Mathematical Optimization and Applications in Biomedical Sciences

Note 17-11
First Steps with OpenMP 4.5 on Ubuntu and Nvidia GPUs

Stefan Rosenberger

May 16, 2018

Contents

Preface: We want to emphasis that this document is a note on our OpenMP 4.5 parallelization. Therefore, we check only *sloppy* with respect to spellings and formulations.

1 Setup System

We use the following system to install and run OpenMP 4.5:

- OS:

```
No LSB modules are available.
Distributor ID: LinuxMint
Description:    Linux Mint 18.1 Serena
Release:        18.1
Codename:       serena
```

- Host:

```
/0/3d                          memory      32GiB System Memory
/0/3d/3                        memory      16GiB DIMM Synchronous 2133 MHz (0.5 ns)
/0/43                          memory      256KiB L1 cache
/0/44                          memory      1MiB L2 cache
/0/45                          memory      8MiB L3 cache
/0/46                          processor   Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
/0/1/0.0.0        /dev/sda     disk        250GB Samsung SSD 850
```

- GPU:

```
Mon Sep  4 14:59:44 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 375.82                 Driver Version: 375.82                    |
|-------------------------------+----------------------+----------------------+
| 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 106...  Off  | 0000:01:00.0     On |                  N/A |
| 40%   34C    P0    28W / 120W |    499MiB /  6072MiB |      3%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      1491     G    /usr/lib/xorg/Xorg                             286MiB |
|    0      1994     G    cinnamon                                       206MiB |
|    0      3605     G    /usr/lib/firefox/firefox                         1MiB |
+-----------------------------------------------------------------------------+
```

1.1 Install OpenMP for GPU's

First of all one needs to setup the local system for OpenMP 4.5. Change gcc version:

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo apt update
sudo apt install gcc-6
```

```

sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-6 60 --slave /usr/bin/g++
        g++ /usr/bin/g++-6
```

Furthermore we use the bash script (from [3])

```
#!/bin/sh

#
# Build GCC with support for offloading to NVIDIA GPUs.
#

work_dir=$HOME/offload/wrk
install_dir=$work_dir/install

# Location of the installed CUDA toolkit
cuda=/usr/local/cuda

# Build assembler and linking tools
mkdir -p $work_dir
cd $work_dir
git clone https://github.com/MentorEmbedded/nvptx-tools
cd nvptx-tools
./configure \
    --with-cuda-driver-include=$cuda/include \
    --with-cuda-driver-lib=$cuda/lib64 \
    --prefix=$install_dir
make
make install
cd ..

# Set up the GCC source tree
git clone https://github.com/MentorEmbedded/nvptx-newlib
svn co svn://gcc.gnu.org/svn/gcc/trunk gcc
cd gcc
contrib/download_prerequisites
ln -s ../nvptx-newlib/newlib newlib
cd ..
target=$(gcc/config.guess)

# Build nvptx GCC
mkdir build-nvptx-gcc
cd build-nvptx-gcc
../gcc/configure \
    --target=nvptx-none --with-build-time-tools=$install_dir/nvptx-none/bin \
    --enable-as-accelerator-for=$target \
    --disable-sjlj-exceptions \
    --enable-newlib-io-long-long \
    --enable-languages="c,c++,fortran,lto" \
    --prefix=$install_dir
make -j4
make install
cd ..

# Build host GCC
mkdir build-host-gcc
cd  build-host-gcc
../gcc/configure \
    --enable-offload-targets=nvptx-none \
```

```
    --with-cuda-driver-include=$cuda/include \
    --with-cuda-driver-lib=$cuda/lib64 \
    --disable-bootstrap \
    --disable-multilib \
    --enable-languages="c,c++,fortran,lto" \
    --prefix=$install_dir
make -j4
make install
cd ..
```

1.2 Run a test example

We test the following source code:

```
#include <set>
#include <iostream>
#include <assert.h>
#include <vector>

using namespace std;

void vec_mult(int N, const int n_print=10)
{
        double p[N], v1[N], v2[N];

        for(int ii=0; ii<N; ++ii){
                p[ii] = ii%5;
                v1[ii] = ii%6;
                v2[ii] = ii%7;
        }

        int i;
        #pragma omp target map(to: v1[0:N], v2[0:N], p[0:N])
        #pragma omp parallel for private(i)
        for (i=0; i<N; i++){
                        p[i] = v1[i] * v2[i];
        }

        int num_print = 0;
        if(n_print > N) num_print = N;
        else num_print = n_print;
        for(int ii=0; ii<num_print; ++ii) cout << p[ii] << "  ";
        cout << endl;
}


/*
 *  ========== MAIN =============
 *  Test OpenMP 4.5
 */
int main(int argc, char **argv)
{
        cout << "##########################################" << endl;
        cout << "#######        Test Start        #########" << endl;
        cout << "##########################################" << endl << endl;


    vec_mult(100000);
```

```

    cout << "Test complete!\n";
    return 0;
}
```

We compile and run the code with the makefile:

```
1  default:
2          rm -f a.out
3          /home/rosenbs/offload/wrk/install/bin/g++ -std=c++11 -O3 -fopenmp -DOPENMP
4                          -foffload=nvptx-none -Wall test.cpp
5          ./a.out
6
7  run:
8          ./a.out
9
10 clean:
11         rm -f a.out
```

Note, in line 3 one **has to use the g++ compiler** which is created from the bash file.
We get the runtime output:

```
1  ./a.out
2  ##########################################
3  #######        Test Start       #########
4  ##########################################
5
6  0  1  2  3  4  0  1  2  3  4
7  Test complete!
```

To check that we are really on the GPU we test the code wich nvprov:

```
1  rosenbs@math068 ~/src/test/testomp $ nvprof ./a.out
2  ##########################################
3  #######        Test Start       #########
4  ##########################################
5
6  ==14296== NVPROF is profiling process 14296, command: ./a.out
7  0  1  2  3  4  0  1  2  3  4
8  Test complete!
9  ==14296== Profiling application: ./a.out
10 ==14296== Profiling result:
11 Time(%)      Time     Calls       Avg       Min       Max  Name
12  95.25%  3.8015ms         1  3.8015ms  3.8015ms  3.8015ms  _Z8vec_multii$_omp_fn$0
13   4.75%  189.63us         7  27.090us     544ns  62.655us  [CUDA memcpy HtoD]
14
15 ==14296== API calls:
16 Time(%)      Time     Calls       Avg       Min       Max  Name
17  59.93%  104.82ms         1  104.82ms  104.82ms  104.82ms  cuCtxCreate
18  34.01%  59.484ms         1  59.484ms  59.484ms  59.484ms  cuCtxDestroy
19   2.20%  3.8485ms         1  3.8485ms  3.8485ms  3.8485ms  cuCtxSynchronize
20   1.20%  2.1007ms        22  95.485us  15.600us  886.99us  cuLinkAddData
21   0.92%  1.6123ms         1  1.6123ms  1.6123ms  1.6123ms  cuModuleLoadData
22   0.42%  729.89us         1  729.89us  729.89us  729.89us  cuLinkComplete
23   0.26%  457.66us         1  457.66us  457.66us  457.66us  cuMemAllocHost
24   0.25%  430.49us         1  430.49us  430.49us  430.49us  cuLaunchKernel
25   0.25%  428.78us         7  61.254us  4.1860us  133.51us  cuMemcpyHtoD
26   0.24%  427.22us         1  427.22us  427.22us  427.22us  cuLinkCreate
27   0.15%  256.86us         1  256.86us  256.86us  256.86us  cuMemFreeHost
28   0.09%  149.88us         2  74.940us  5.0250us  144.85us  cuMemAlloc
```

```
29    0.04%   76.635us    15  5.1090us     101ns  70.902us  cuDeviceGetAttribute
30    0.04%   64.396us     2  32.198us  6.4940us  57.902us  cuMemFree
31    0.00%   5.5170us     4  1.3790us     440ns  3.7350us  cuDeviceGetCount
32    0.00%   4.1140us     3  1.3710us     676ns  2.4450us  cuDeviceGet
33    0.00%   2.7490us    12     229ns     106ns     892ns  cuCtxGetDevice
34    0.00%   2.4150us     8     301ns     195ns     542ns  cuMemGetAddressRange
35    0.00%   2.2930us     1  2.2930us  2.2930us  2.2930us  cuLinkDestroy
36    0.00%   1.4190us     1  1.4190us  1.4190us  1.4190us  cuInit
37    0.00%   1.3550us     1  1.3550us  1.3550us  1.3550us  cuModuleGetFunction
38    0.00%      961ns     1     961ns     961ns     961ns  cuCtxGetCurrent
39    0.00%      837ns     1     837ns     837ns     837ns  cuMemHostGetDevicePointer
40    0.00%      557ns     2     278ns     233ns     324ns  cuFuncGetAttribute
41    0.00%      386ns     1     386ns     386ns     386ns  cuModuleGetGlobal
```

Additional settings (motivited by: Adam Dziekonski

☞ adding the following was crucial to make it work with gcc 9.0 (experimental)
- add LIBDIR to the 'LD_LIBRARY_PATH' environment variable during execution
- add LIBDIR to the 'LD_RUN_PATH' environment variable during linking

But:

☞ This is NOT working for MPI applications!

2 Test OpenMP for the sandbox example

To let OpenMP run with MPI I need to change the default g++ compiler.
OLD:

```
g++ -> /etc/alternatives/g++*
```

NEW:

```
g++ -> /home/rosenbs/offload/wrk/install/bin/g++
```

Furthermore, I've created softlinks to all *wrapper* we have created in the first section, and put them into */usr/bin/*:

```
c++ -> /home/rosenbs/offload/wrk/install/bin/c++*
cpp -> /home/rosenbs/offload/wrk/install/bin/cpp*

g++ -> /home/rosenbs/offload/wrk/install/bin/g++
gcc -> /home/rosenbs/offload/wrk/install/bin/gcc

gcc-ar -> /home/rosenbs/offload/wrk/install/bin/gcc-ar*
gcc-nm -> /home/rosenbs/offload/wrk/install/bin/gcc-nm*
gcc-ranlib -> /home/rosenbs/offload/wrk/install/bin/gcc-ranlib*

gcov -> /home/rosenbs/offload/wrk/install/bin/gcov*
gcov-dump -> /home/rosenbs/offload/wrk/install/bin/gcov-dump*
gcov-tool -> /home/rosenbs/offload/wrk/install/bin/gcov-tool*

gfortran -> /home/rosenbs/offload/wrk/install/bin/gfortran

nvptx-none-ar -> /home/rosenbs/offload/wrk/install/bin/nvptx-none-ar*
nvptx-none-as -> /home/rosenbs/offload/wrk/install/bin/nvptx-none-as*
```

```
nvptx-none-ld -> /home/rosenbs/offload/wrk/install/bin/nvptx-none-ld*
nvptx-none-ranlib -> /home/rosenbs/offload/wrk/install/bin/nvptx-none-ranlib*
nvptx-none-run -> /home/rosenbs/offload/wrk/install/bin/nvptx-none-run*
nvptx-none-run-single -> /home/rosenbs/offload/wrk/install/bin/nvptx-none-run-single*


x86_64-pc-linux-gnu-accel-nvptx-none-gcc -> /home/rosenbs/offload/wrk/install/bin/x86_64
        -pc-linux-gnu-accel-nvptx-none-gcc*
x86_64-pc-linux-gnu-c++ -> /home/rosenbs/offload/wrk/install/bin/x86_64-pc-linux-gnu-c++*
x86_64-pc-linux-gnu-g++ -> /home/rosenbs/offload/wrk/install/bin/x86_64-pc-linux-gnu-g++*
x86_64-pc-linux-gnu-gcc -> /home/rosenbs/offload/wrk/install/bin/x86_64-pc-linux-gnu-gcc*
x86_64-pc-linux-gnu-gcc-8.0.0 -> /home/rosenbs/offload/wrk/install/bin/x86_64-pc-linux-gnu-
        gcc-8.0.0*
x86_64-pc-linux-gnu-gcc-ar -> /home/rosenbs/offload/wrk/install/bin/x86_64-pc-linux-gnu-gcc-ar*
x86_64-pc-linux-gnu-gcc-nm -> /home/rosenbs/offload/wrk/install/bin/x86_64-pc-linux-gnu-gcc-nm*
x86_64-pc-linux-gnu-gcc-ranlib -> /home/rosenbs/offload/wrk/install/bin/x86_64-pc-linux-gnu-gcc-ranlib*
x86_64-pc-linux-gnu-gfortran -> /home/rosenbs/offload/wrk/install/bin/x86_64-pc-linux-gnu-gfortran*
```

☞ This is still NOT working for the sandbox example we have used for the paper *Effective OpenACC Parallelizaion for Sparce Matrix Problems!*
Durning Link-step we get the error message:

```
x86_64-pc-linux-gnu-accel-nvptx-none-gcc: error: unrecognized command line option
        '-nvptx-none'
```

2.1 Working Test CODE

We create a *standanlone* test example, which works for our MPI parallized solver.
Makefile:

```
default:
        rm -f a.out
        mpicxx -std=c++11 -O3 -fopenmp -DOPENMP -DOPENMP_OFFLOAD -foffload=nvptx-none test.cpp
        ./a.out /home/rosenbs/src/test/Sandbox_Maxim/genBi_ellmat.bin /home/rosenbs/src/
                test/Sandbox_Maxim/genBi_rhs_Ie.bin

run:
        ./a.out /home/rosenbs/src/test/Sandbox_Maxim/genBi_ellmat.bin /home/rosenbs/src/
                test/Sandbox_Maxim/genBi_rhs_Ie.bin

clean:
        rm -f a.out
```

And in the file test.cpp we include our libraries:

```
include <set>
#include <iostream>
#include <assert.h>
#include <vector>


#include<stdlib.h>
#include<errno.h>
#include </home/rosenbs/src/sandbox_hackatron/PT_C/toolbox/toolbox_sandbox.h>
#include </home/rosenbs/src/sandbox_hackatron/PT_C/toolbox_funcs.hpp>
```

```

using namespace sr_solver;

#define LOAD_RHS
....
```

The rest is copy-past of the my sanbox_example.cpp file.

☞ This works!

3 CRS Matrix-Vector mutliplication

We tried several different types of the Matrix-Vector parallelization with OpenMP 4.5

- First of all, we try the simplest:

```
#pragma omp target teams distribute parallel for
for (int i=0; i<sv; i++) {
   const int *const __restrict p_col = col+dsp[i];
   const double *const __restrict p_ele = ele+dsp[i];
   double s = 0.0;
   for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
      s += p_ele[j]*u[p_col[j]];
   }
   v[i] = s;
}
```

☞ – Solve time: 775ms

- With simd instruction

```
#pragma omp target teams distribute parallel for simd
for (int i=0; i<sv; i++) {
   const int *const __restrict p_col = col+dsp[i];
   const double *const __restrict p_ele = ele+dsp[i];
   double s = 0.0;
   for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
      s += p_ele[j]*u[p_col[j]];
   }
   v[i] = s;
}
```

☞ – Solve time: 9.3ms

- With simd instruction and reduction

```
#pragma omp target teams distribute parallel for simd
for (int i=0; i<sv; i++) {
   const int *const __restrict p_col = col+dsp[i];
   const double *const __restrict p_ele = ele+dsp[i];
   double s = 0.0;
#pragma omp reduction(+:s)
   for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
      s += p_ele[j]*u[p_col[j]];
   }
   v[i] = s;
}
```

☞ – Solve time: 9.01ms

- With ordered instruction

```
#pragma omp target teams distribute parallel for simd
for (int i=0; i<sv; i++) {
   const int *const __restrict p_col = col+dsp[i];
   const double *const __restrict p_ele = ele+dsp[i];
   double s = 0.0;
#pragma omp ordered simd
   for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
      s += p_ele[j]*u[p_col[j]];
   }
   v[i] = s;
}
```

☞ – Solve time: 35.7ms

- With simd in inner loop!

```
#pragma omp target teams distribute parallel for
for (int i=0; i<sv; i++) {
   const int *const __restrict p_col = col+dsp[i];
   const double *const __restrict p_ele = ele+dsp[i];
   double s = 0.0
#pragma omp simd
   for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
      s += p_ele[j]*u[p_col[j]];
   }
   v[i] = s;
}
```

☞ – Solve time: 8.84ms; WRONG RESULT!!! Race condition!

- With critical!

```
#pragma omp target teams distribute parallel for
for (int i=0; i<sv; i++) {
   const int *const __restrict p_col = col+dsp[i];
   const double *const __restrict p_ele = ele+dsp[i];
   double s = 0.0
#pragma omp critical
   for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
      s += p_ele[j]*u[p_col[j]];
   }
   v[i] = s;
}
```

☞ – *Endless loop:* break after 1 min of calculation! ⟶ a closer look at the loop — obvious!

- With reduction

```
#pragma omp target parallel for
for (int i=0; i<sv; i++) {
   const int *const __restrict p_col = col+dsp[i];
   const double *const __restrict p_ele = ele+dsp[i];
   double s = 0.0
#pragma omp reduction(+:s)
```

```
    for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
        s += p_ele[j]*u[p_col[j]];
    }
    v[i] = s;
}
```

☞ – Solve time: 114ms

- With simd instruction and reduction and num_teams definition

```
#pragma omp target teams distribute parallel for simd num_teams(kk)
for (int i=0; i<sv; i++) {
    const int *const __restrict p_col = col+dsp[i];
    const double *const __restrict p_ele = ele+dsp[i];
    double s = 0.0
#pragma omp reduction(+:s)
    for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
        s += p_ele[j]*u[p_col[j]];
    }
    v[i] = s;
}
```

☞
- Solve time kk=5: 10.4ms
- Solve time kk=10: 7.7ms
- Solve time kk=25: 8.3ms
- Solve time kk=50: 8.8ms
- Solve time kk=100: 8.8ms
- Solve time kk=200: 8.9ms
- Solve time kk=500: 8.9ms
- Solve time kk=1000: 8.9ms

- With simd instruction and reduction and num_teams, mun_threads definition

```
#pragma omp target teams distribute parallel for simd num_teams(10) mun_threads(kk)
for (int i=0; i<sv; i++) {
    const int *const __restrict p_col = col+dsp[i];
    const double *const __restrict p_ele = ele+dsp[i];
    double s = 0.0
#pragma omp reduction(+:s)
    for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
        s += p_ele[j]*u[p_col[j]];
    }
    v[i] = s;
}
```

☞
- Solve time kk=16: 7.6ms
- Solve time kk=32: 7.7ms
- Solve time kk=64: 7.6ms
- Solve time kk=128: 7.6ms
- Solve time kk=256: 7.6ms
- Solve time kk=512: 7.7ms

- Solve time kk=1024: 7.6ms
- Solve time kk=2048: 7.8ms

No influence on the parallelization!

- With simd instruction and reduction and num_teams, thread_limit definition

```
#pragma omp target teams distribute parallel for simd num_teams(10) thread_limit(kk)
for (int i=0; i<sv; i++) {
   const int *const __restrict p_col = col+dsp[i];
   const double *const __restrict p_ele = ele+dsp[i];
   double s = 0.0
#pragma omp reduction(+:s)5
   for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
      s += p_ele[j]*u[p_col[j]];
   }
   v[i] = s;
}
```

☞
- Solve time kk=16: 7.7ms
- Solve time kk=32: 9.0ms
- Solve time kk=64: 9.0ms
- Solve time kk=128: 8.9ms
- Solve time kk=256: 9.0ms
- Solve time kk=512: 9.0ms
- Solve time kk=1024: 9.0ms
- Solve time kk=2048: 9.1ms

Only negative influence on the parallelization!

3.1 Compiler options

We consider the following kernel:

```
#pragma omp target teams distribute parallel for simd num_teams(10)
for (int i=0; i<sv; i++) {
   const int *const __restrict p_col = col+dsp[i];
   const double *const __restrict p_ele = ele+dsp[i];
   double s = 0.0
#pragma omp reduction(+:s)
   for (int j=0; j<dsp[i+1]-dsp[i]; j++) {
      s += p_ele[j]*u[p_col[j]];
   }
   v[i] = s;
}
```

- Compile line:

  mpicxx -std=c++11 -O3 -DNDEBUG -fopenmp -DOPENMP -foffload=nvptx-none test.cpp

☞
- Solve time: 7.7ms

- Add to compile line:

-fopenmp-simd

☞ – Solve time: 7.8ms

- Add to compile line:

-march=native

☞ – Solve time: 7.8ms

- Add to compile line (and replace O3):

-fopenmp-simd -O2 -march=native

☞ – Solve time: 7.7ms

- Add to compile line:

-mlong-double-64

☞ – Solve time: 7.65ms

- Add to compile line:

-m64

☞ – Solve time: 7.7ms

- Add to compile line:

-m64 -mlong-double-64

☞ – Solve time:7.6ms

- Add to pragma `num_threads(128)` and add to compile line:

-mprefer-avx128

☞ – Solve time: 7.56ms

- Add to compile line:

-ftree-vectorize

☞ – Solve time: 7.61ms

- Add to compile line:

-ftree-vectorizer-verbose=1

☞ – Solve time: 7.61ms

- Add to compile line:

-m64 -mlong-double-64 -mprefer-avx128 -ftree-vectorize

☞ – Solve time: 7.61ms

4 Diagonal-Matrix Vector mutliplication

We tried several different types of the Diagonal-Matrix Vector parallelization with OpenMP 4.5. Wherein we calculate

$$v = \omega D u \tag{4.1}$$

wherein ω is a scalar.

- First of all, we try the simplest:

```
#pragma omp target parallel for
for (int ii = 0; ii < vs; ii++) {
        v[ii] = omega * D[ii] * u[ii];
}
```

☞ – Solve time: 29.88ms

- With teams distribute

```
#pragma omp target teams distribute parallel for
for (int ii = 0; ii < vs; ii++) {
        v[ii] = omega * D[ii] * u[ii];
}
```

☞ – Solve time: 1.97ms

- With simd

```
#pragma omp target teams distribute parallel for simd
for (int ii = 0; ii < vs; ii++) {
        v[ii] = omega * D[ii] * u[ii];
}
```

☞ – Solve time: 0.86ms

- With num_teams

```
#pragma omp target teams distribute parallel for simd num_teams(kk)
for (int ii = 0; ii < vs; ii++) {
        v[ii] = omega * D[ii] * u[ii];
}
```

☞
- Solve time kk=5: 0.84ms
- Solve time kk=10: 0.72ms
- Solve time kk=25: 0.75ms
- Solve time kk=50: 0.77ms
- Solve time kk=100: 0.76ms
- Solve time kk=200: 0.77ms

- With num_threads

```
#pragma omp target teams distribute parallel for simd num_teams(10) num_threads(kk)
for (int ii = 0; ii < vs; ii++) {
        v[ii] = omega * D[ii] * u[ii];
}
```

☞
- Solve time kk=16: 0.72ms
- Solve time kk=32: 0.72ms
- Solve time kk=64: 0.72ms
- Solve time kk=128: 0.69ms
- Solve time kk=256: 0.71ms
- Solve time kk=512: 0.70ms
- Solve time kk=512: 0.70ms

We find the same setting as for the matrix vector product!

5 ELLPACK Matrix Vector mutliplication

We calculate the matrix vector product for ELLPACK type matrices.

- First of all, we try the simplest:

```
#pragma omp target teams distribute parallel for
for (int ii = 0; ii < v_size; ii += _block_size) {

        const int stride = ii * max_length;
        const int kk_max = (_block_size + ii < v_size) ? (_block_size + ii) : v_size;

        for (int kk = ii; kk < kk_max; ++kk) {
                S s = 0.0;

                for (int jj = 0; jj < max_length; ++jj) {
                        const int index = stride + kk - ii + jj * _block_size;
                        s += ele[index] * u[col[index]];
                }
                v[kk] = s;
        }
}
```

☞
- Solve time: 87.67ms

- With the directives from the previous sections:

```
#pragma omp target teams distribute parallel for simd default(none) num_teams(10)
                thread_limit(128)
for (int ii = 0; ii < v_size; ii += _block_size) {

        const int stride = ii * max_length;
        const int kk_max = (_block_size+ii < v_size) ? (_block_size+ii):v_size;

        for (int kk = ii; kk < kk_max; ++kk) {
                S s = 0.0;

                for (int jj = 0; jj < max_length; ++jj) {
                        const int index = stride + kk - ii + jj * _block_size;
                        s += ele[index] * u[col[index]];
                }
                v[kk] = s;
        }
}
```

☞ – Solve time: 47.65ms

- With reduction:

```
#pragma omp target teams distribute parallel for simd default(none) num_teams(10)
                thread_limit(128)
for (int ii = 0; ii < v_size; ii += _block_size) {

        const int stride = ii * max_length;
        const int kk_max = (_block_size+ii < v_size)?(_block_size+ii):v_size;

        for (int kk = ii; kk < kk_max; ++kk) {
                S s = 0.0;

#pragma omp reduction(+:s)
                for (int jj = 0; jj < max_length; ++jj) {
                        const int index = stride + kk - ii + jj * _block_size;
                        s += ele[index] * u[col[index]];
                }
                v[kk] = s;
        }
}
```

☞ – Solve time: 46.16ms

- Second initialization:

```
#pragma omp target teams distribute default(none) num_teams(100) thread_limit(128)
for (int ii = 0; ii < v_size; ii += _block_size) {

        const int stride = ii * max_length;
        const int kk_max = (_block_size+ii < v_size)?(_block_size+ii):v_size;

#pragma omp parallal for simd
        for (int kk = ii; kk < kk_max; ++kk) {
                S s = 0.0;

#pragma omp reduction(+:s)
                for (int jj = 0; jj < max_length; ++jj) {
                        const int index = stride + kk - ii + jj * _block_size;
                        s += ele[index] * u[col[index]];
                }
                v[kk] = s;
        }
}
```

☞ – Solve time: 962.57ms

- Second initialization and define threads interior:

```
#pragma omp target teams distribute default(none) num_teams(10000)
for (int ii = 0; ii < v_size; ii += _block_size) {

        const int stride = ii * max_length;
        const int kk_max = (_block_size+ii < v_size)?(_block_size+ii):v_size;

#pragma omp parallel for num_threads(128)
        for (int kk = ii; kk < kk_max; ++kk) {
                S s = 0.0;

#pragma omp reduction(+:s)
                for (int jj = 0; jj < max_length; ++jj) {
                        const int index = stride + kk - ii + jj * _block_size;
                        s += ele[index] * u[col[index]];
                }
                v[kk] = s;
        }
}
```

☞ – Solve time: 87.47ms

- Second initialization with inner simd:

```
#pragma omp target teams distribute default(none) num_teams(10000)
for (int ii = 0; ii < v_size; ii += _block_size) {

        const int stride = ii * max_length;
        const int kk_max = (_block_size+ii < v_size)?(_block_size+ii):v_size;

#pragma omp parallel for simd num_threads(128)
        for (int kk = ii; kk < kk_max; ++kk) {
                S s = 0.0;

#pragma omp reduction(+:s)
                for (int jj = 0; jj < max_length; ++jj) {
                        const int index = stride + kk - ii + jj * _block_size;
                        s += ele[index] * u[col[index]];
                }
                v[kk] = s;
        }
}
```

☞ – Solve time: 27.15ms

- Change inner loops

```
_v.zero_acc();

#pragma omp target teams distribute default(none) num_teams(10000)
for (int ii = 0; ii < v_size; ii += _block_size) {
        const int stride = ii * max_length;
        const int kk_max = (_block_size+ii<v_size) ? (_block_size+ii):v_size;
        for (int jj = 0; jj < max_length; ++jj) {
#pragma omp simd
                for (int kk = ii; kk < kk_max; ++kk) {
                        const int index = stride + kk - ii + jj * _block_size;
                        v[kk] += ele[index] * u[col[index]];
                }
        }
}
```

☞ – Solve time: 32.63ms

- Define all 3 parallelizations

```
_v.zero_acc();

const int num_teams_local = v_size/_block_size + 1;
#pragma omp target teams distribute default(none) num_teams(60000)
for (int ii = 0; ii < v_size; ii += _block_size) {
        const int stride = ii * max_length;
        const int kk_max = (_block_size < v_size - ii) ?
                (_block_size) : v_size - ii;
#pragma omp parallel for simd
        for (int jj = 0; jj < max_length; ++jj) {
                const int index = stride + jj * _block_size;
                for (int kk = 0; kk < kk_max; ++kk) {
                        v[kk + ii] += ele[index + kk] * u[col[index + kk]];
                }
        }
}
```

☞ – Solve time: 41.26ms
Race condition!!!!

6 Scalar Product

We try to improve our parallelization. The working version is

```
template<class S>
void scalar_product(const toolbox_vector<S> &_x, const toolbox_vector<S> &_y, S &_s)
{
   S s = 0.0;
   const S *const __restrict x = _x.data(), *const __restrict y = _y.data();
   const int x_size = _x.size();

#pragma omp target teams distribute parallel for simd map(tofrom: s)
         reduction(+:s) schedule(simd:static, _block_size)
   for(int i = 0; i < x_size; i++)
   {
      s += x[i] * y[i];
   }
   _s = s;
}
```

Since we observe with OpenACC 2 kernels for the execution, we try to create a *copy version* of a CUDA scalar product. Which means, we create a parallel executable region, and than a reduction step:

```
template<class S>
void scalar_product(const toolbox_vector<S> &_x, const toolbox_vector<S> &_y, S &_s)
{
    S s = 0.0;
    const S *const __restrict x = _x.data(), *const __restrict y = _y.data();
    const int x_size = _x.size();

    const double* x_ptr = &x[0];
    const double* y_ptr = &y[0];

    const int loc_size = 1;

    double loc_val[loc_size];
    for(int ii=0; ii<loc_size; ++ii) loc_val[ii] = 0.0;
    double* loc_ptr = &loc_val[0];
```

```

#pragma omp target enter data map(to:loc_ptr[0:loc_size])

#pragma omp target parallel for simd safelen(loc_size)
        for(int ii=0; ii<x_size; ++ii){
                const int jj = ii%loc_size;
                        loc_ptr[jj] += x_ptr[ii] * y_ptr[ii];
        }
#pragma omp target parallel for reduction(+:s) map(tofrom:s)
        for(int ii=0; ii<loc_size; ++ii) s += loc_ptr[ii];

    _s = s;
}
```

☞ This version has a race condition. Note every try to use an atomic operation to fix the bug fails → compiling error.

An alternative can be a use of *number of threads*, therefore we try to call on the target in a parallel region.

6.1 Linear Clause

We tried:

```
omp_get_num_threads();
```

This leads to the bug:

☞ error: there are no arguments to 'omp_get_num_threads' that depend on a template parameter, so a declaration of 'omp_get_num_threads' must be available [-fpermissive] nthreads = omp_get_num_threads();

As a consequence we try to use the *linear* clause. We found online the description:

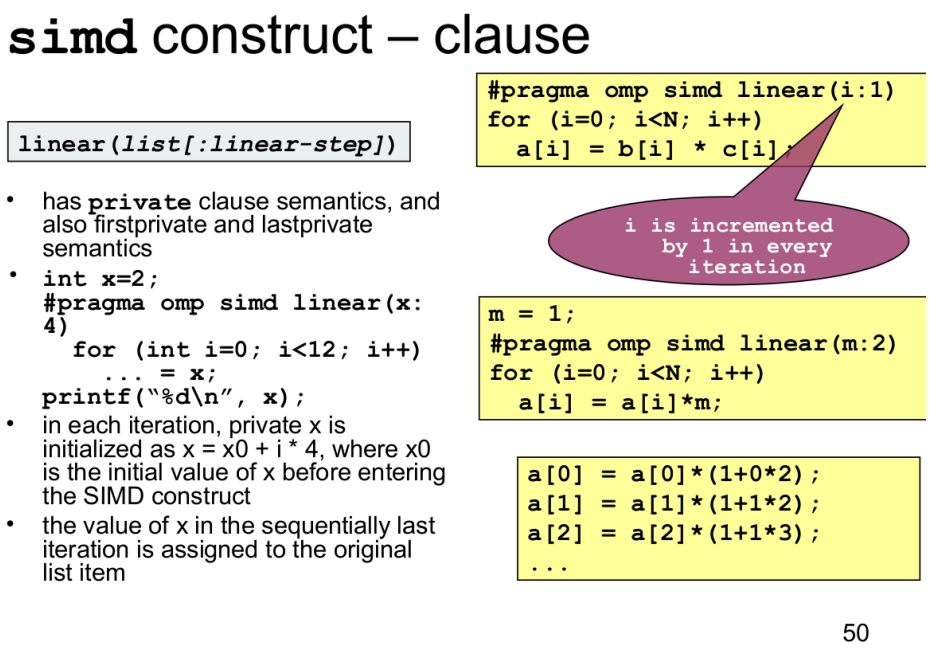

Figure 6.1: Linear clause for OpenMP4.0 (Picture from [1])

Therefore we try:

```
const int loc_size = 12;
double loc_val[loc_size];
int x = 1;

#pragma omp simd linear(x:4)
for(int i=0; i<loc_size; i++) loc_val[i] = static_cast<double>(x);

for(int ii=0; ii<loc_size; ++ii) cout << loc_val[ii] << "␣␣";
```

and get the output:

☞ 1 1 1 1 1 1 1 1 1 1 1 1

Therefore the description of 6.1 is wrong. Lets try something else:

```
const S *const __restrict x = _x.data(), *const __restrict y = _y.data();
const int x_size = _x.size();

const int loc_size = 12;
double loc_val[loc_size];
for(int ii=0; ii<loc_size; ++ii) loc_val[ii] = 0.0;
double* loc_ptr = &loc_val[0];

int m = 1;

#pragma omp target parallel for simd linear(m:2) map(tofrom:loc_ptr[0:loc_size])
for(int i=0; i<loc_size; i++){
        loc_ptr[i] = x[i] + m;
}

for(int ii=0; ii<loc_size; ++ii) cout << loc_ptr[ii] << "␣␣";
```

wherein we initialize x with 1.0.

☞ 2 2 6 6 10 10 14 14 18 20 22 24

This is also not what I would expect from 6.1 .....
We try (wherein we consider 20 elements of the array):

```
#pragma omp target parallel for simd linear(m:5) safelen(10) map(tofrom:loc_ptr[0:loc_size])
```

and get

☞ 2 2 2 17 17 17 32 32 32 47 47 47 62 62 72 72 82 82 92 92

By guessing I would conclude that the clause `linear` add in every iteration 5 to the value `m`, but not individual for every thread. It sums up the values for every thread until OpenMP starts a new stride.

☞ Sh**: this means we do not have access to the thread ID of the parallelization ....

Possibly we can use it for the stride steps in ELLPACK .....
BUG fixing:

☞ Out of history, `#include<omp.h>` was included as last in the headers ....

We tried:

```
#pragma omp target enter data map(to:loc_ptr[0:loc_size])
#pragma omp target parallel for simd map(tofrom: loc_size) num_threads(100)
for(int ii=0; ii<loc_size+10; ++ii){
   loc_size = omp_get_num_threads();
   const int nthread = omp_get_thread_num();
   loc_ptr[nthread] = nthread;
}
cout << "val:␣␣" << loc_size << endl;

#pragma omp target update from(loc_ptr[0:loc_size])
for(int ii=0; ii<loc_size; ++ii) cout <<"␣␣" << loc_ptr[ii];
```

and get the output:

☞ val: 8
0 0 0 0 0 0 0

This means we have a bug!

References

[1] Chapman, B., Eachempati, D., Li, K.: OpenMP 4.0 features (2017), http://extremecomputingtraining.anl.gov/files/2014/01/OpenMP-40-features-ATPESC-final-v2.pdf, [Online; accessed 12-09-2017]

[2] heise Developer: Accelerator offloading mit gcc (2017), https://www.heise.de/developer/artikel/Accelerator-Offloading-mit-GCC-3317330.html, [Online; accessed 29-08-2017]

[3] Walfridsson, K.: Building gcc with support for nvidia ptx offloading (2017), https://kristerw.blogspot.co.at/2017/04/building-gcc-with-support-for-nvidia.html, [Online; accessed 28-08-2017]