Parallelization of the SPoCA2 algorithm

Stefan Hofmeister

March 27, 2017

1 Introduction

The SPoCA2 algorithm (Spatial Possibilistic Clustering Algorithm) is an algorithm for an unsupervised cluster analysis on images and part of the SPoCA suite. It was developed for the unsupervised segmentation of solar extreme ultraviolet (EUV) images, which comprises the identification, segmentation, and extraction of features in the solar atmosphere. It is a module of the Feature Finding Team, which is the main source on modules for the SDO Event Detection System, i.e., a system which segments unsupervised images taken by the satellite Solar Dynamics Observatory (SDO). The EUV images are typically taken at a cadence of 12 seconds at a resolution of $4096 \times 4096$ pixels in 10 wavelength. In this project, we parallelize the SPoCA2 algorithm with a shared-memory approach using OpenMP under the constraint of that the structure of the code shall not change significantly. The parallelization speeds up the algorithm significantly and allows to process large datasets in a reasonable time. The outline is as follows. First, we describe the test systems which were used for the benchmarks. Second, we introduce the SPoCA2 algorithm. Third, we describe issues and solutions with the parallelization of the SPoCA2 code. Fourth, we present the benchmark results. And fifth, we give an outlook on the potential of further optimization.

2 The Test Systems

For the benchmark following in Section 5, we used two test systems.

Test system 1 is the node „compute-0-5“ on the cluster Mephisto at the University of Graz. It contains two Intel Xeon E5-2650; each of them has eight cores clocking at 2 GHz (2.8 GHz at turbo boost), 32 kb L1-, 256 kB L2-, and 20 MB L3-Cache, a maximum memory bandwidth of 51.2 GB/s, and the AVX instruction set for vectorization.

Test system 2 is Intel(R) Xeon Phi 7210 (Knight’s Landing) at the University of Graz. It contains 64 cores clocking at 1.3 GHz (1.5 GHz at turbo boost), 32 kb L1-, 256 kB and 32 MB L2-Cache, a maximum memory bandwidth of 102 GB/s, the AVX-512 instruction set for vectorization, and four-times-hyper-threading.
3 The SPoCA2 algorithm

The SPoCA2 algorithm is an algorithm for cluster analysis on images. It is based on the Fuzzy C-Means (FCM) algorithm, the Possibilistic C-Means (PCM) algorithm, and the Spatial Possibilistic Clustering Algorithm (SPoCA), which are described shortly in the following. For more details, we refer to Barra et al. (2008) and Verbeek et al. (2014).

3.1 The FCM algorithm

Assume that an image taken in various channels contains a set of features, and that the image is appropriate for extracting the features. Then, the multi-dimensional histogram should fall into separated regions, each representing one of the features. A cluster analysis tries to find these regions unsupervised, and thus can be used for extracting the regions.

Let

- $S$ be the number of channels in which the image was taken,
- $N$ the number of pixels in the image to classify,
- $C$ the number of features to find, i.e., the class centers in the histogram,
- $x_j$ a vector with $S$ elements representing the pixel value in the different channels at the pixel $j$ in the image, i.e., the Feature Vector of the pixel $j$,
- $X$ the set of Feature Vectors in the image,
- $b_i$ a vector containing the value of an assumed class center,
- $B$ the set of class centers,
- $d(x_j, b_i)$ the euclidean distance of the Feature Vector $x_j$ to the class center $b_i$,
- $u_{ij}$ the membership value for the pixel $j$ belonging to the class center $i$,
- $U$ the set of membership values, i.e., the partition matrix.

Then, the problem of the determination of the class centers can be described as an minimization problem of the intraclass variance $J$,

$$J_{FCM}(B, U, X) = \sum_{i=1}^{C} \sum_{j=1}^{N} (u_{ij})^m d^2(x_j, b_i),$$  \quad (1)$$

where $m$ is a parameter that controls the degree of fuzzification and usually is set to 2. This equation is subject to

$$\forall i \in \{1, ..., C\} : \sum_{j=1}^{N} u_{ij} < N, \quad (2)$$

$$\forall j \in \{1, ..., N\} : \sum_{i=1}^{C} u_{ij} = 1. \quad (3)$$
The minimization is reached when
\[ u_{ij} = \frac{1}{\sum_{k=1}^{C} \left( \frac{d^2(x_j, b_k)}{d^2(x_j, b_i)} \right)^{1/(m-1)}} , \] (4)
\[ b_i = \frac{\sum_{j=1}^{N} (u_{ij})^m x_j}{\sum_{j=1}^{N} (u_{ij})^m} . \] (5)

In practice in the code implementations, Formulas 4 and 5 are used to calculate and update the partition matrix and class centers at each iteration until the solution converges. The concept of this algorithm is presented in Figure 1.

### 3.2 The PCM algorithm

In the FCM algorithm, the membership degree of a feature vector to any class center depends on its distance to all other class centers, i.e., we have shared membership of all feature vectors to all class centers. To solve this shortcoming, the PCM algorithm minimizes
\[ J_{PCM}(B, U, X) = \sum_{i=1}^{C} \left( \sum_{j=1}^{N} (u_{ij})^m d^2(x_j, b_i) + \eta_i \sum_{j=1}^{N} (1 - u_{ij})^m \right) . \] (6)

The first term of J is the intraclass variance, whereas the second term enforces \( u_{ij} \) to only depend on \( b_i \). \( \eta_i \) is homogeneous to a squared distance. In the implementation, it can be fixed,
or updated at each iteration with

\[ \eta_i = \frac{\sum_{j=1}^{N} d^2(x_j, b_i)}{\sum_{j=1}^{N} (u_{ij})^m}. \]  

(7)

The minimization is reached when

\[ u_{ij} = \frac{1}{1 + (d^2(x_j, b_i)/\eta_i)^{1/(m-1)}}, \]  

(8)

\[ b_i = \frac{\sum_{j=1}^{N} (u_{ij})^m x_j}{\sum_{j=1}^{N} (u_{ij})^m}. \]  

(9)

Here, the membership value \( u_{ij} \) for a feature vector \( x_j \) belonging to a class center \( b_i \) only depends on its distance to the class center, i.e., it is the real membership to the class center (in contrast to the shared membership to all class centers, as in the FCM algorithm).

### 3.3 The SPoCA algorithm

The FCM and PCM algorithm are quite sensitive to outliers. Since images are often noisy, the classification results are expected to be better when not only the pixel values itself, but also the neighbourhood of the pixels are included into the analysis. Thus, the SPoCA algorithm minimizes

\[ J_{SPoCA}(B, U, X) = \sum_{i=1}^{C} \left( \sum_{j=1}^{N} (u_{ij})^m \sum_{k \in O_j} \beta_k d^2(x_k, b_i) + \eta_i \sum_{j=1}^{N} (1 - u_{ij})^m \right), \]  

(10)

where \( O_j \) is the neighbourhood of the pixel \( x_j \), and

\[ \beta_k = \begin{cases} 1 & \text{if } k = j, \\ \frac{1}{\text{Card}(O_j)-1} & \text{otherwise}. \end{cases} \]  

(11)

The minimization is reached when

\[ u_{ij} = \frac{1}{1 + (\sum_{k \in O_j} \beta_k d^2(x_k, b_i)/\eta_i)^{1/(m-1)}}, \]  

(12)

\[ b_i = \frac{\sum_{j=1}^{N} (u_{ij})^m \sum_{k \in O_j} \beta_k x_k}{2 \sum_{j=1}^{N} (u_{ij})^m}. \]  

(13)

### 3.4 The SPoCA2 algorithm

In practice, possibilistic clustering algorithms often suffer from convergence to a unique class center. To circumvent this problem, the term \( 1/(m - 1) \) in Equation of the SPoCA algorithm [12] is replaced by \( 2/(m - 1) \). This leads to a more compact membership function, i.e., the membership functions for the various class centers do not overlap so easily, and thus the algorithm does not converge so easily to a unique class center.
Table 1: Timings of the five steps at processing one image. The middle column gives the timings for processing one image with almost no preprocessing, the right column with preprocessing.

4 Code Optimization

In this section, we present problems we are dealing with and some solutions. First, we run a benchmark to identify the parts of the code with the highest workload, and then present the parallelization concept. Last, we identify issues with the parallelization in the original code and solutions.

4.1 Structure of the code and a first benchmark

The code consists of five parts. First, the images are read from disk and preprocessed; note that preprocessing often reduces the number of pixels to classify considerably. Second, the set of feature vectors is build as an input for the classifier. Third, the class centers of the SPoCA2 classifier are initialized by running the FCM classifier. Fourth, the cluster analysis is performed using the SPoCA2 algorithm. And fifth, the pixels are attributed to the final class centers, and the result is written to disk.

To find the routines with the highest workloads, the time of each routine was measured by running one image at full resolution (4096 × 4096 pixels) two times, i.e., with almost no preprocessing and with preprocessing, on test system 1. Due to the large number of routines, here, we will not present the timing of each individual routine, but only of blocks of routines; the timings are given in Table 1.

From Table 1, it is clearly visible that most of the computational time is spent during the initialisation and classification. Though, when we assume a speedup of a factor of 10 by parallelization, it becomes clear that we also have to parallelize the preprocessing, the building of the set of feature vectors, and the attribution.

4.2 Concept for parallelization

For parallelization, we decided to use OpenMP 4.5, since it is the common version on the target servers.

4.3 Issues

Although the code was well written for linear processing, it was not thought to be parallelized. Thus, we had to face the following issues:

- inappropriate alignment of variables in memory for vectorization,
• STL vectors are used throughout the program, and especially its method push_back(),
• methods of classes are used to access the class variables,
• indirect indexing of variables,
• small workloads in the loop cycles, therefore no atomic memory access is possible,
• the STL vector elements are partially calculated in a non-linear access pattern, i.e., writing collisions of different threads are probable at parallel execution.

Inappropriate alignment of variables:

```c++
if(fuzzifier == 2)
{
    #pragma omp parallel for default(shared) schedule(guided) num_threads(NumThreads)
    for (unsigned long j = 0; j < X.size(); ++j)
    {
        MembershipSet::iterator uij = U.begin() + j*numberOfClasses;
        #pragma omp simd
        #pragma loop count min(2) avg[4]
        for (unsigned i = 0; i < numberOfClasses; ++i, ++uij)
        {
            *uij /= ota[i];
            *uij = l / (l + *uij) * *uij;
        }
    }
}
```

In the code sample above, X is a vector with \( \approx 16 \cdot 10^6 \) elements, numberOfClasses is \( \approx 4 \), and U \( \approx 4 \times 16 \cdot 10^6 \). To use a reasonable vectorization, the loops should be interchanged, i.e., the vectorization should work on the \( 16 \cdot 10^6 \) elements. Though, this would result in that the vector U has to be processed numberOfClasses times, which leads to a memory bandwidth bottleneck. Further, the vector U would have to be processed at a stride of numberOfClasses, but numberOfClasses is not fixed at compilation time. Leaving the order of the loops, we can only vectorize at the vector length of numberOfClasses. In order to increase the vector length, we thought about collapsing the loops in the omp statement; though, since eta depends only on the inner loop count, also this technique did not give a reasonable result.

We also tried manual vectorization by using the vectorclass library of Agner Fog
(http://www.agner.org/optimize/vectorclass.pdf), and got a speed-up of about 6 due to vectorization. However, since this increased the intricacy of the code, and since the processing time of this code snippet could be disregarded in comparison to the total processing time of the program, we finally decided to not use the manual vectorization. In the end, we decided to keep the order of the loops and perform the vectorization using the simd pragma with the vector length of numberOfClasses, whereby we added the pragma loop_count. This pragma gives the compiler an estimated loop count and thus prevents peel loops at short loop lengths.
STL vectors:

```cpp
#pragma omp parallel for schedule(guided) firstprivate(pixelValue, pixelRadius2, sunCenterX, sunCenterY, sunRadius, xAxes, yAxes)
for (int y_pos = 0; y_pos < yAxes; ++y_pos)
{
    Real y = -sunCenterY + y_pos;
    pixelValue = $6$ pixels[y_pos * xAxes];
    vector<Real> annulusMean_tmp(unsigned)[maxLimRadius-minLimRadius]/deltaR]+3,0];
    vector<unsigned> annulusNbrPixels_tmp(unsigned)[maxLimRadius-minLimRadius]/deltaR]+3,0];
    vector<EUFPixelType> onDiskList_tmp;
    onDiskList_tmp.reserve(xAxes);
    for (Real x = -sunCenterX; x < xmax; ++x)
    {
        if ({*pixelValue} != null\pixelValue)
        {
            pixelRadius2 = $x$ * $x$ + $y$ * $y$;
            if (pixelRadius2 <= minLimRadius2)
            {
                onDiskList_tmp.push_back({*pixelValue});
            }
            else if (pixelRadius2 <= maxLimRadius2)
            {
                indice = unsigned(sqrt(pixelRadius2)-minLimRadius2)/deltaR);                        
                annulusMean_tmp[indice] += {*pixelValue};
                annulusNbrPixels_tmp[indice] += 1;
            }
        }
        ++pixelValue;
    }
    #pragma omp critical
    {
        #pragma omp simd
        for (int i = 0; i < unsigned(maxLimRadius-minLimRadius)/deltaR)+2; ++i)
        {
            annulusMean[i] = annulusMean_tmp[i];
            annulusNbrPixels[i] = annulusNbrPixels_tmp[i];
        }
        onDiskList.insert_or_assign(onDiskList.end(), onDiskList_tmp.begin(), onDiskList_tmp.end());
    }
}
```

STL vectors were used throughout the code, and especially its method push_back. When we parallelize the code snippet above simply by adding the omp pragma, using push_back for onDiskList directly would lead to writing collisions. To avoid this, the parallelization is done only on the outer loop. Each thread processes the total inner loop and saves its result temporarily in onDiskList_tmp, and then copies onDiskList_tmp to the common variable onDiskList in the critical section. Note that the order of onDiskList can be kept if the ordered keyword would be added to the omp statement.

**Methods of classes to access class variables:** In the original algorithm, methods of classes are used access class variables. Although this is good practice in serial codes, it is problematic in parallel codes. The class variables are saved in the L3-cache of a specific core. When using methods to access these variables, at each call, the variable has to be transferred from the L3-cache where the class is saved to the L3-cache of the requesting thread, leading to a memory bandwidth bottleneck. This was circumvented by replacing the method to access the class variable by the class variable itself, and putting the variable into the private clause.
Indirect indexing

In the code snippet above, U and beta are accessed in the inner loop by indirect indexing \((Nj \rightarrow data()[k])\). Prefetching data does not work well at indirect indexing, and thus, a large proportion of the processing time of the whole program is needed in this one line. This code snippet is parallelized in an outer loop (not presented in the code snippet), which gives a speed-up of the number of threads used. To solve the problem of indirect indexing, the structure of the variables would have to be changed, which would induce re-writing larger parts of the code. Thus, we decided not to change the structure of the variables and keep the problem of indirect indexing.

But we remark that this issue should be taken care of when the whole code is revised.

Small workloads in the loop cycles:

```c
long maxChunkSize = ceil((double) smoothedX.size()) / NUMTHREADS / 8;
#pragma omp parallel for default(shared) schedule(guided)
for (long j = 0; j < smoothedX.size(); j = maxChunkSize)
{
    ClassCenterSet B_tmp;
    B_tmp.resize(numberClasses, 0);
    Vector<Real> sum_tmp(numberClasses, 0.);

    long vsize = min(maxChunkSize, (long) smoothedX.size() - j);
    for (long dj = 0; dj < vsize; dj++)
    {
        #pragma omp simd
        #pragma loop count min(2) avg(4)
        for (unsigned i = 0; i < numberClasses; ++i)
        {
            Real uij_n = U[[j+dj]*numberClasses+i] * U[[j+dj]*numberClasses+i];
            B_tmp[i] = smoothedX[j+dj] * uij_n;
            sum_tmp[i] += uij_n;
        }
    }
    #pragma omp critical
    for (unsigned i = 0; i < numberClasses; ++i)
    {
        B[i] += B_tmp[i];
        sum[i] += sum_tmp[i];
    }
```

The code snippet above could have been parallelized by using the reduction statement in the omp clause. However, the reduction statement on arrays was not available in openMP 4.5. Thus, we had to find a work-around. Here, the easiest way would have been to use atomic memory access in the parallelized loop region. But, since the workloads in the inner loops are too small, the locking time of the memory regions during atomic memory access overweights the processing time, leading to a bottleneck. To solve this problem, we process the outer loop in chunks, where the results of the inner loop are saved in a temporary variable (B_tmp, sum_tmp) and added to the common variable at the end of the processed
chunks in the critical section. This leads to a shift of the processing time as compared to the memory locked time, allowing a reasonable parallelized processing.

Non-linear access pattern in STL vectors:

```cpp
// for Parallelization: now start the threads at offsets of 3x (maximum distance of pixels in neighborhood)
// i.e., if the neighborhood consists of n lines (x n pixels), start the threads (=chunks) every 3rd line
// no overlap at writing. (Each chunk processed has a length of maxneiglen, and is indexed by chunkN [chunk
for (long chunkOffset = 0; chunkOffset < i; chunkOffset++)
{
    #pragma omp parallel for default(shared) firstprivate(chunkOffset, maxneiglen) schedule(guided)
    for (int chunkN = chunkOffset; chunkN < (long) ceil(X.size() / float(maxneiglen)); chunkN += i)
    {
        //if the last chunk has less elements left than maxneiglen, set chunklen to the number
        //of elements remaining
        const long chunklen = min(maxneiglen, (long) X.size() - (chunkN * maxneiglen));
        // calculate starting position j of chunk (corresponding to i)
        const long j_chunkStart = chunkN * maxneiglen;

        //process chunk
        for (unsigned long dj = 0; dj < chunklen; ++dj)
        {
            Real d2BLXj[numberClasses];
            #pragma omp simd
            #pragma loop_count min(2) avg(4)
            for (unsigned i = 0; i < numberClasses; ++i)
            {
                d2BLXj[i] = distance_squared(X[j_chunkStart + dj], O[i]);
                U[i][j_chunkStart + dj]*numberClasses + i -= d2BLXj[i];
            }
        }
    }
}
```

In the code snippet above, the elements of U, i.e., U at a given pixel position within an image, are calculated by including the neighbourhood of the pixel Nj. Here, the inclusion of the neighbourhood is done by writing the value of U of a pixel to the elements of U of the neighbouring pixels. Thus, processing the code in parallel can lead to writing collisions when two threads are writing to the same neighbouring pixel. In the code snippet above, we solved this problem by colouring lines of the images, i.e., first, we calculated the maximum distance of neighbouring pixels as distance in array index. Then, we coloured the image into three colours, in the order 1, 2, 3, 1, 2, 3, 1, 2, 3, ..., whereby each colour 1, 2, 3 consists of the maximum distance of neighbouring pixels consecutive pixels. Finally, we process the image in three steps, i.e., one step for each colour (≡ chunkOffset). At each step, one line of the colour is only allowed to be processed by one thread. This order of processing avoids writing collisions.

An alternative way to solve this problem is given by the following code snippet:
Here, we changed the code that the neighbourhood is included not by writing to the neighbourhood, but by reading from neighbouring pixels. However, to get the correct result, the loop has to be processed in two steps. First, U has to be calculated for each pixel without including the neighbourhood, and saved to U_tmp. Second, the neighborhood is included and U is recalculated from U_tmp. This processing seems to be easier; though, the vector U (respectively, U_tmp) has to be processed two times, i.e., the memory bandwidth used is doubled as compared to the first version as described above. The second version, as described here, is faster at a low number of threads, as for the calculation of U of a pixel, only one indirect indexing has to be used (beta is indexed directly), but slower at a higher number of threads due to the increased memory bandwidth usage.

5 Benchmark

In order to evaluate the speed-up of the parallelization of the SPoCA2 algorithm, we ran to the SPoCA2 algorithm for two configurations on the two test systems. At configuration 1, we processed 1 image at full resolution (4096 × 4096 pixel) with almost no preprocessing, i.e., all pixels were classified. At configuration 2, we processed 100 images at full resolution (4096 × 4096 pixel) with the configuration for identifying coronal holes on solar EUV images; the preprocessing contains the annulus limb correction, the division of the pixel values by the median, the nullifying of all pixels which are not on the Sun’s disk, and setting all pixels which are above of the mode of the histogram of the image to the mode. This results in that only about 45% of all pixels have to be classified, that distribution of the pixels gets flatter, that thus the intraclass-variance decreases, and that the process converges faster. Thus, at configuration 2, the processing time per image decreases as compared to configuration 1. The progressing time was measured by running each configuration on each test system 5 times, and taking the minimal processing time.
5.1 Test System 1

Figures 2, 3, and Tables 2, 3 give the processing times, speed-ups, and efficiencies for configuration 1 and 2 for 1-16 threads on test system 1. The processing time of configuration 1, i.e., processing 1 image at full resolution, for the original algorithm was 478 s, and for the parallelized version at 1 thread 260 s, giving a serial speed-up of 1.84. Increasing the number of threads to 16, the processing time decreases to 30 s; we achieve a further speed-up of 8.6, and the efficiency decreases to 38%. For configuration 2, the processing time of the original algorithm was 3782 s. Processing the images with the parallelized version at 1 thread took 2671 s, giving a serial speed-up of 1.42, and increasing the number of threads to 16 decreased the processing time to 443 s, increased the speed-up by a further factor of 6.03, and decreased the efficiency to 54%.

5.2 Test System 2

Figures 4, 5, and Tables 4, 5 give the processing times, speed-ups, and efficiencies for configuration 1 and 2 for 1-256 threads on test system 2. The processing time of
Figure 3: Processing time, speed-up, and efficiency for processing 100 images at full resolution (4096 × 4096 pixel) and preprocessing with the SPoCA2 algorithm on an Intel E5-2650. The cross gives the processing time of the original algorithm.
<table>
<thead>
<tr>
<th>Threads</th>
<th>Time [s]</th>
<th>Speed-up</th>
<th>Efficiency [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>260</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>148</td>
<td>1.8</td>
<td>98</td>
</tr>
<tr>
<td>3</td>
<td>99</td>
<td>2.7</td>
<td>89</td>
</tr>
<tr>
<td>4</td>
<td>76</td>
<td>3.4</td>
<td>82</td>
</tr>
<tr>
<td>5</td>
<td>88</td>
<td>2.9</td>
<td>78</td>
</tr>
<tr>
<td>6</td>
<td>65</td>
<td>4.6</td>
<td>74</td>
</tr>
<tr>
<td>7</td>
<td>52</td>
<td>5.0</td>
<td>65</td>
</tr>
<tr>
<td>8</td>
<td>62</td>
<td>4.2</td>
<td>57</td>
</tr>
<tr>
<td>9</td>
<td>42</td>
<td>6.2</td>
<td>57</td>
</tr>
<tr>
<td>10</td>
<td>42</td>
<td>6.2</td>
<td>49</td>
</tr>
<tr>
<td>11</td>
<td>48</td>
<td>5.4</td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td>35</td>
<td>7.4</td>
<td>45</td>
</tr>
<tr>
<td>13</td>
<td>34</td>
<td>7.5</td>
<td>40</td>
</tr>
<tr>
<td>14</td>
<td>49</td>
<td>5.4</td>
<td>39</td>
</tr>
<tr>
<td>15</td>
<td>31</td>
<td>8.4</td>
<td>37</td>
</tr>
<tr>
<td>16</td>
<td>30</td>
<td>8.6</td>
<td>38</td>
</tr>
</tbody>
</table>

Table 2: Processing time, speed-up, and efficiency for processing 1 image at full resolution (4096 × 4096 pixel) with the SPoCA2 algorithm on an Intel E5-2650.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time [s]</th>
<th>Speed-up</th>
<th>Efficiency [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2671</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1362</td>
<td>1.9</td>
<td>87</td>
</tr>
<tr>
<td>3</td>
<td>996</td>
<td>2.7</td>
<td>86</td>
</tr>
<tr>
<td>4</td>
<td>807</td>
<td>3.4</td>
<td>86</td>
</tr>
<tr>
<td>5</td>
<td>680</td>
<td>3.9</td>
<td>58</td>
</tr>
<tr>
<td>6</td>
<td>598</td>
<td>4.6</td>
<td>77</td>
</tr>
<tr>
<td>7</td>
<td>584</td>
<td>4.6</td>
<td>73</td>
</tr>
<tr>
<td>8</td>
<td>587</td>
<td>4.5</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>520</td>
<td>5.1</td>
<td>71</td>
</tr>
<tr>
<td>10</td>
<td>538</td>
<td>5.0</td>
<td>62</td>
</tr>
<tr>
<td>11</td>
<td>546</td>
<td>4.9</td>
<td>51</td>
</tr>
<tr>
<td>12</td>
<td>490</td>
<td>5.4</td>
<td>61</td>
</tr>
<tr>
<td>13</td>
<td>509</td>
<td>5.2</td>
<td>57</td>
</tr>
<tr>
<td>14</td>
<td>489</td>
<td>5.5</td>
<td>38</td>
</tr>
<tr>
<td>15</td>
<td>479</td>
<td>5.6</td>
<td>56</td>
</tr>
<tr>
<td>16</td>
<td>443</td>
<td>6.0</td>
<td>54</td>
</tr>
</tbody>
</table>

Table 3: Processing time, speed-up, and efficiency for processing 100 images at full resolution (4096 × 4096 pixel) and preprocessing with the SPoCA2 algorithm on an Intel E5-2650.
Figure 4: Processing time, speed-up, and efficiency for processing one image at full resolution (4096 × 4096 pixel) with the SPoCA2 algorithm on an Intel Xeon Phi 7210. The cross gives the processing time of the original algorithm.

configuration 1, i.e., processing 1 image at full resolution, for the original algorithm was 1620 s on the Knight’s Landing, and for the parallelized version at 1 thread 1480 s, giving a serial speed-up of 1.09. Increasing the number of threads to 256, the processing time decreases to 33 s, and we achieve a further speed-up of 44. The calculation of the efficiency at this test-system seems unreasonable since all all programs which are processed with only 1 thread are not distributed on the available threads, but are all processed of thread 1 of core 1. For configuration 2, the processing time of the original algorithm was 29521 s. Processing the images with the parallelized version at 1 thread took 25362 s, giving a serial speed-up of 1.16, and increasing the number of threads to 256 decreased the processing time to 774 s and increased the speed-up by a further factor of 33. Note that at the Knight’s Landing, the usage of hyper-threading, i.e., using more than 64 threads, has almost no effect on the processing time. This means that hyper-threading is almost useless for this algorithms, probably because most of the parallelized loops have almost no workload. Thus, the registers in the cores are already filled most of the time, and hyper-threading gets useless. Further, it is remarkable that the speed-up by using a higher number of threads does not saturate as on test system 1, but increases with an almost constant factor for up to 64 threads. Since the Knight’s Landing has a much higher memory bandwidth, this can be interpreted as that the bottleneck for the algorithm is still the memory bandwidth, and that it is worth porting the code to a GPU. Comparing the total calculation times for test system 1 and test system 2, test system 2 is about as fast as test system 1. By looking into the details of the processing times (not given here), we find that the parallelized parts of the algorithm run at test system 2 faster by a factor of 1.7 as compared to test system 1, but that the serial parts of the program are much slower on the Knight’s Landing even as expected. To get insight in this unexpected behaviour, further studies are needed.
Figure 5: Processing time, speed-up, and efficiency for processing 100 images at full resolution (4096 × 4096 pixel) and preprocessing with the SPoCA2 algorithm on an Intel Xeon Phi 7210. The cross gives the processing time of the original algorithm.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Time [s]</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1480</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>749</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>418</td>
<td>3.5</td>
</tr>
<tr>
<td>8</td>
<td>208</td>
<td>7.1</td>
</tr>
<tr>
<td>16</td>
<td>114</td>
<td>12.9</td>
</tr>
<tr>
<td>32</td>
<td>61</td>
<td>24.2</td>
</tr>
<tr>
<td>64</td>
<td>39</td>
<td>37.6</td>
</tr>
<tr>
<td>128</td>
<td>33</td>
<td>44.8</td>
</tr>
<tr>
<td>256</td>
<td>33</td>
<td>44.0</td>
</tr>
</tbody>
</table>

Table 4: Processing time, speed-up, and efficiency for processing one image at full resolution (4096 × 4096 pixel) with the SPoCA2 algorithm on an Intel Xeon Phi 7210.
<table>
<thead>
<tr>
<th>Threads</th>
<th>Time [s]</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25362</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>12491</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>4722</td>
<td>5.4</td>
</tr>
<tr>
<td>8</td>
<td>1854</td>
<td>13.7</td>
</tr>
<tr>
<td>16</td>
<td>1251</td>
<td>20.3</td>
</tr>
<tr>
<td>32</td>
<td>908</td>
<td>27.9</td>
</tr>
<tr>
<td>64</td>
<td>729</td>
<td>34.8</td>
</tr>
<tr>
<td>128</td>
<td>707</td>
<td>35.8</td>
</tr>
<tr>
<td>256</td>
<td>774</td>
<td>32.7</td>
</tr>
</tbody>
</table>

Table 5: Processing time, speed-up, and efficiency for processing 100 images at full resolution (4096 × 4096 pixel) and preprocessing with the SPoCA2 algorithm on an Intel Xeon Phi 7210.

6 Outlook

On the E5-2650, in linear processing of one images with almost no preprocessing, we achieved a speed-up of about 1.7, and a further speed-up of ≈ 8 by processing the code in parallel at 16 threads. On the Xeon Phi 7210, we achieved a serial speed-up of 1.09, and a further speed-up of ≈ 44 by processing the code in parallel by 256 threads.

The processing time of the SPoCA2 algorithm can be further reduced significantly by rearranging the structure of the variables for a reasonable vectorization, and by avoiding indirect indexing. The benchmarks suggested that porting the code to a GPU could also significantly reduce the processing time due to the much lower memory access time and the higher amount of threads. We tried to port the code to GPUs using openACC, however, it was mainly not possible due to the usage of STL vectors. Thrust vectors were not appropriate due to the structure of the code, and the replacement by simple arrays was not reasonable due to the large usage of the push_back method of STL vectors. Managed memory handling was not possible due to the not supported error handling in the recent openMP and openACC versions. We also tried to port the code to heterogeneous system architectures, in specific an AMD A10-7700k using gcc7 ([https://github.com/HSAFoundation/HSA-OpenMP-GCC-AMD](https://github.com/HSAFoundation/HSA-OpenMP-GCC-AMD)), but also this was not possible due to the not supported error handling.

For a general revision of the code, we suggest that great importance is attached to the structure of the variables for vectorization and indexing (in particular avoiding indirect and non-linear indexing), to the replacement of STL vectors by simple arrays, and to a revision of the error handling so that the code can also be ported to GPUs.

References


Barra, V. and Delouille, V. and Hochedez, J.-F. *Segmentation of extreme ultraviolet solar images via multichannel fuzzy clustering*. Advances in Space Research (2008), Volume 42,
917-925.