PARALLEL COMPUTING TECHNOLOGIES IN VIDEO STABILIZATION FOR TEACHING PURPOSES

Márquez Rodríguez, César Gabriel
Systems Engineering and Automation, University of Vigo, Spain, cmrodriguez17@esei.uvigo.es

García Rivera, Matías
Systems Engineering and Automation, University of Vigo, Spain, mgrivera@uvigo.es

Díaz-Cacho Medina, Miguel
Systems Engineering and Automation, University of Vigo, Spain, mcacho@uvigo.es

Camaño Portela, José Luís
Systems Engineering and Automation, University of Vigo, Spain, cama@uvigo.es

Abstract

In this paper, the development of a video-stabilization program is described as part of the training in the subject Parallel Architectures in the Degree in Computer Engineering of the University of Vigo. The purpose is to take advantage of the parallelism methodologies in processors to teach students about computer vision and use it in applications like vibration sensors for maintenance in Industry 4.0 or computer vision for the autonomous vehicle. The main tool to implement this program is the C programming language and the OpenCV library.

Keywords: parallel computing, vector processing, video and image processing.

1 INTRODUCTION

The Department of Systems and Automation Engineering of the University of Vigo is responsible for teaching subjects such as Computer Architecture or Parallel Architectures in the Degree in Computer Engineering taught at High School of Computer Science at the Ourense campus. These provide theoretical and practical knowledge about computer hardware, its evolution and improvements.

In the subject 'Parallel Architectures' [1], students' knowledge in the area of Architecture and Computer Technology will be completed, studying the parallel execution of instructions in monoprocessor systems, the possibilities offered by multi-core processors, multiprocessor systems, vector processors, multicomputers and computer clusters.

The aim of this subject is to make students understand the advantages, risks and limitations of parallelism and concurrency techniques used by processors in order to reduce execution times. In addition, it seeks to enable the student to measure the performance of a processor when running a program.

With this project and according to some competences of this subject, students will:

- Know how to apply their knowledge to their work or vocation in a professional way. Particularly in this case, the image stabilization program could be useful to serve as vibration sensor for maintenance in Industry 4.0 or even to stabilize images in a vibrating environment like image sensors for autonomous driving.
- Have the ability to identify and analyze problems and design, develop, implement, verify and document software solutions based on appropriate knowledge of current theories, models and techniques.

In this frame, video and image processing are used as the leitmotiv to teach about parallelism technique in a practical and attractive way, and to show how this techniques represents significant improvement in performance.

2 DESCRIPTION AND APPROACH OF OF PROJECT AND ALGORITHM

The further presented program will allow you to remove unwanted movements from a video that

should display a fixed and stable image. The idea behind the algorithm is to set as a reference the central point of the first frame, look for it in the other frames and correct the difference between them and the first frame.

The user interface is based on the *comdlg32.dll* library to display dialog-windows, and basically it allows the user to select the video which will be processed. It must have AVI or MPEG format.

The video will be processed in blocks, taking as reference coordinates of the central block from the first frame (ctlf, rtlf) and looking for it in the following ones within pre-established limits.

$$rtlf := \frac{firstFrameHeight + blockHeight}{2} \quad (1)$$

$$ctlf := \frac{firstFrameWidth + blockWidth}{2} \quad (2)$$

where *rtlf* is the row to look for and *ctlf* the column to look for.

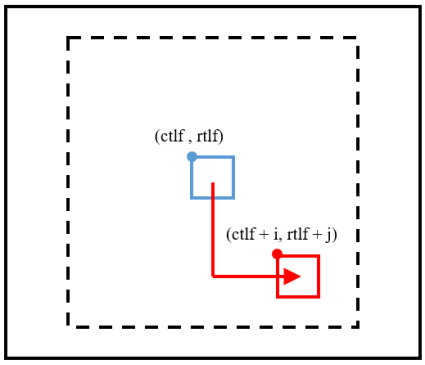

Figure 1. Representation of the displaced central block from the first frame in the following frames.

In order to find the displaced central block in the other frames, we must compare this initial block with the blocks within the pre-established limits in each frame, looking for the more similar one. Those limits allow us to be more efficient, as it is not likely that the image has experienced much change between consecutive frames.

We can define the difference between two pixels as the addition of the differences in absolute value of their RGB components:

$$dif(p_1, p_2) := |B_1 - B_2| + |G_1 - G_2| + |R_1 - R_2| \quad (3)$$

Therefore, the difference between two blocks of $n \times m$ dimensions will be the addition of the differences of their pixels:

$$dif(b_1, b_2) := \sum_{i=0,k=0}^{n \times m} dif(p_i, p_k) \quad (4)$$

Once the coordinates (foundCol, foundRow) of the displaced block have been found, we can draw some conclusions from the next equalities:

$$foundCol := ctlf + columnShift \quad (5)$$

$$foundRow := rtlf + rowShift \quad (6)$$

- If $rowShift = 0$, the image has not vertically moved.
- If $rowShift > 0$, the image has moved downwards. We will have to move the image up as much as $rowShift$ tells us to.
- If $rowShift < 0$, the image has moved upwards. We will have to move the image down as much as $rowShift$ tells us to.
- If $columnShift = 0$, the image has not horizontally moved.
- If $columnShift > 0$, the image has moved rightwards. We will have to move the image left as much as $columnShift$ tells us to.
- If $columnShift < 0$, the image has moved leftwards. We will have to move the image right as much as $columnShift$ tells us to.

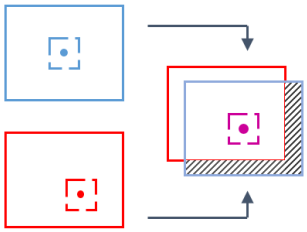

Figure 2. Correction a frame in relation previous one. The stripped space cannot be reconstructed.

As we can see in Figure 2, stabilize a frame in relation to the first frame, there are parts of the image that cannot be reconstructed, because the information of that newly shown area is not available. This areas will be painted black.

The previously explained process can be described by the following pseudocode:

```
WHILE thereAreFrames DO

  minDif ← ∞

  FROM r ← rtlf - LIMIT TO foundRow + LIMIT
  DO
    FROM c ← ctlf - LIMIT TO foundCol + LIMIT
    DO

      acDif ← compBloq(filBusq, colBusq, f,
      c)

      IF acDif<minDif THEN
```

```
      minDif ← acDif
      fMin ← f
      cMin ← c
    END_IF
  END_FROM
END_FROM
imageShift(rtlf - fMin, ctlf - cMin)
END_WHILE
```

It is possible to decompose any movement suffered by the image in the movements we have seen before:

- On the vertical axis: upwards or downwards.
- On the horizontal axis: rightwards or leftwards.

In order to stabilize the image, we must correct the displacements on the corresponding axes, moving the image in the opposite direction to that of the movement on that axis.

Moreover, we must not forget that the image cannot be completely reconstructed. In the solution proposed in this article, as we have already said, we have chosen to fill in these areas in black.

This process can be described in the following way with this pseudocode, where *rowShift* and *columnShift* represent the displacements suffered on the vertical and horizontal axes, respectively.

```
IF rowShift>0 THEN
// Correct downwards shift
END_IF
IF rowShift<0 THEN
// Correct upwards shift
END_IF
IF columnShift>0 THEN
// Correct rightwards
END_IF
IF columnShift<0 THEN
// Correct leftwards shift
END_IF
IF rowShift>0 THEN
// Paint the unknown area black
END_IF
IF rowShift<0 THEN
// Paint the unknown area black
END_IF
IF columnShift>0 THEN
// Paint the unknown area black
END_IF
IF columnShift<0 THEN
// Paint the unknown area black
END_IF
```

So as to correct the shift suffered on an axis in a concrete direction the process to follow is always similar: we must establish a correspondence between the position of the initial image and that of the displaced image. For example, in the case of a downward displacement:

```
IF rowShift>0 THEN
  FROM r ← height - rowShift - 1 TO 0 DO
    FROM c ← 0 TO width DO
      //Coordinates correspondence
      copyPixel(r,c,r+rowShift,c)
    END_FROM
  END_FROM
END_IF
//...
IF rowShift>0 THEN
  FROM r ← 0 TO rowShift DO
    FROM c ← 0 TO width DO
      blackPixel(r,c)
    END_FROM
  END_FROM
END_IF
```

On the other way, by determining the first derivative of *rowShift* and *columnShift*, the peaks of the vibration can be found and combined with the time difference between them, the energy of the vibration can be determined and therefore be compared with reference situations to know maintenance parameters. Nevertheless, this analysis is out of the scope of this paper and will be studied in future works.

3 IMPLEMENTATION

3.1 PROGRAMMING LANGUAGE, DEVELOPMENT TOOLS AND LIBRARIES.

For the development of this program the programming language C was chosen, which is the one used in the practical part of the subject. Since C is a structured language [2], it makes it easier to think of programs in terms of function modules or blocks, for example, with the approach described before for the resolution of the proposed problem. In addition, there are a lot of libraries supported by the C library, making the task of programming easier.

NetBeans integrated development environment has been used as a development tool in its version 8.2. It is used in this subject as in many others in the degree. This IDE [3] is a useful tool for large scale projects and makes it easy to bring in new developers, since its structure is very visible.

To compile our programs, we used the GCC free compiler [4], GNU Compiler Collection. However, due to the fact that most PCs use Windows as an operating system, Cygwin will be used [5]. Cygwin is a large collection of GNU and open source tools that provides functionality similar to a Linux distribution on Windows.

The OpenCV [6] open source library, released under the BSD 3 clause license, has been used for the video processing part. It is a library focused on real time applications, which is also multiplatform, because it supports different languages and operating systems. OpenCV is free for commercial use.

3.2 PERFORMANCE IMPROVEMENT. USE OF PARALLELISM TECHNIQUES.

Image and video processing involves operations on many occasions with high computational load. When an instruction is executed, it is likely that it will be run on a large set of data. As a result, SIMD (*Single instruction, Multiple Data*) techniques emerge, with the aim of achieving data level parallelism.

Consequently, the process explained in the section 2 of this article can be accelerated using SIMD. As a first approach, we have chosen the Intel® SSE2 set of instructions to make these improvements. The data type __m128i, which is a 128-bit record, has been used. Among the functions provided by SSE2 [7], the following were used:

- *__m128i mm_loadu_si128 (__m128i const* mem_addr)*, loads 128-bits of integer data into *mem_addr*.
- *void mm_storeu_si128 (__m128i* mem_addr, __m128i b)*, stores 128-bits of integer data from *b* into memory.
- *__m128i _mm_set1_epi32 (int c)*, broadcasts 32-bit integer *c* to *dst*.
- *__m128i _mm_add_epi32 (__m128i b, __m128i c)*, adds packed 32-bit integers in *b* and *c*, and stores results in *dst*.
- *int _mm_cvtsi128_si32 (__m128i b)*, copies the lower 32-bit integer in *b* to dst.
- *__m128i _mm_srli_si128 (__m128i b, int imm8)*, shifts *b* right by *imm8* bytes while shifting in zeros, and stores results in *dst*.

The *_m128i _mm_sad_epu8 (__m128i a, __m128i b)* function deserves a deeper explanation. It computes the absolute differences of packed unsigned 8-bit integers in *a* and *b*, then it horizontally sums each consecutive 8 differences to produce two unsigned 16-bit integers, and pack these unsigned 16-bit integers in the low 16 bits of 64-bit elements in *dst*. This instruction is perfect to use in our block comparison function because it allows us to calculate the differences between two pixel blocks in parallel, which means an enormous saving in computational effort.

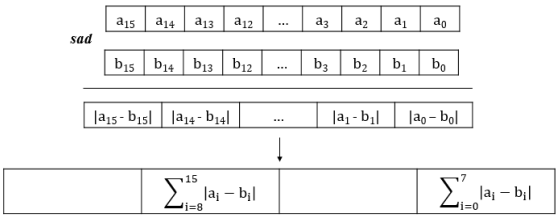

Figure 3. Representation of SSE2 _mm_sad_epu8.

```
unsigned int compareBlocks(IplImage* img1, int r1, int c1,
                           IplImage* img2, int r2, int c2, int height,
                           int width) {

    unsigned int dif = 0;

    int row, column;

    for (row = 0; row < height; row++) {

        uchar* pImg1 = (uchar*) (img1->imageData +
        (r1 + row) * img1->widthStep + c1 * img1->nChannels);
        uchar* pImg2 = (uchar*) (img2->imageData +
        (r2 + row) * img2->widthStep + c2 * img2->nChannels);

        for (column = 0; column < width; column++) {
            dif += abs(*pImg1++ - *pImg2++);
            dif += abs(*pImg1++ - *pImg2++);
            dif += abs(*pImg1++ - *pImg2++);


        }
    }
    return dif;
}
```

Figure 4. Block comparison function in SISD *(Single Instruction, Single Data)*.

```
int compareBlocks(IplImage* img1, int i, int j,
IplImage* img2, int k, int l, int height, int width) {

    int dif;
    __m128i difReg = _mm_setl_epi32(0);
    __m128i aux;
    __m128i rec1, rec2;
    int r1, r2, cc;

    for (r1 = i, r1 = k; r1 < i + height; r1++, r2++) {

        __m128i *pImg1 = (__m128i *) (img1->imageData
        + r1 * img1->widthStep + j * img1->nChannels);

        __m128i *pImg2 = (__m128i *) (img2->imageData
        + r2 * img2->widthStep + l * img2->nChannels);

        for (cc = 0; cc < img1->nChannels * width; cc += 16)
            rec1 = _mm_loadu_si128(pImg1++);
            rec2 = _mm_loadu_si128(pImg2++);
            aux = _mm_sad_epu8(rec1, rec2);
            difReg = _mm_add_epi32(aux, difReg);
        }
    }
    dif = _mm_cvtsi128_si32(difReg);
    difReg = _mm_srli_si128(difReg, 8);
    dif += _mm_cvtsi128_si32(difReg);

    return dif;
}
```

Figure 5. Block comparison function in SIMD *(Single Instruction, Multiple Data)* with SSE2.

These improvements can also be included in the image shifting function, since the task of moving each frame also involves a significant workload.

```
if (rowShift > 0) {
    for (row = image->height - rowShift - 1; row >= 0; row--) {
        unsigned char* pImgOrigen = (unsigned char*)
        (image->imageData + row * image->widthStep);
        unsigned char* pImgDestino = (unsigned char*)
        (image->imageData + (row + rowShift) * image->widthStep
        for (column = 0; column < image->width; column++) {
            *pImgDestino++ = *pImgOrigen++;
            *pImgDestino++ = *pImgOrigen++;
            *pImgDestino++ = *pImgOrigen++;
        }
    }
}
```

Figure 6. Correction in the image shifting function of downward shift, programmed only with SISD.

```
if (rowShift > 0) {
    for (row = image->height - rowShift - 1; row >= 0; row--) {
        __m128i* pImgOrigen = (__m128i*) (image->imageData
        + row * image->widthStep);
        __m128i* pImgDestino = (__m128i*) (image->imageData
        + (row + rowShift) * image->widthStep);
        for (cc = 0; cc < image->widthStep; cc += 16) {
            aux = _mm_loadu_si128(pImgOrigen++);
            _mm_storeu_si128(pImgDestino++, aux);
        }
    }
}
```

Figure 7. Correction in the image shifting function of downward shift, programmed with SIMD instructions.

```
if (rowShift > 0) {
    for (row = 0; row < rowShift; row++) {
        unsigned char* pImgOrigen = (unsigned char*)
        (image->imageData + row * image->widthStep);
        for (column = 0; column < image->width; column++) {
            *pImgOrigen++ = 0;
            *pImgOrigen++ = 0;
            *pImgOrigen++ = 0;
        }
    }
}
```

Figure 8. Correction in the image shifting function of the unknown area when downwards shift, SISD.

```
if (rowShift > 0) {
    for (row = 0; row < rowShift; row++) {
        __m128i* pImgOrigen = (__m128i*)
        (image->imageData + row * image->widthStep);
        for (cc = 0; cc < image->widthStep; cc += 16) {
            _mm_storeu_si128(pImgOrigen++, _mm_setl_epi32(0));
        }
    }
}
```

Figure 9. Correction in the image shifting function of the unknown area when downwards shift, SIMD.

To further improve the performance of the program, a version of it has been developed using instructions from Intel®AVX2 [8], which doubles the size of the SSE2 records we were working with: AVX2 has 32 records of 256 bits.

The data type __m256i, which is a 256-bit record, has been used. Among the functions provided by AVX2 [7], the following were used:

- __m256i _mm256_loadu_si256 (__m256i const * mem_addr)
- void _mm256_storeu_si256 (__m256i * mem_addr, __m256i a)
- __m256i _mm256_set1_epi32 (int a)
- __m256i _mm256_sad_epu8 (__m256i a, __m256i b)
- __m256i _mm256_add_epi32 (__m256i a, __m256i b)
- int _mm256_cvtsi256_si32 (__m256i a)
- __m256i _mm256_srli_si256 (__m256i a, const int imm8)

All these instructions have the same functionality as their corresponding in SSE2. The difference is that AVX2 works with 256-bit records instead of 128-bit records.

As shown in this paper, the stabilization program has been implemented using parallelism tools and its performance has been measured in order to compare the level of improvement, both by applying these techniques and by not doing so. Such reference implementations have been made available to the public in the repository accessible from [11]

3.3 OTHER WAYS TO IMPROVE PERFORMANCE.

Another way of increasing the performance of the program could be by using threads which share the computational load during execution.

In addition, techniques such as loop unwinding, which allows the microprocessor to better organize the instructions to be executed, could be used. On the other hand, we can obtain a more detailed analysis by using profilers [9]. These are tools that allow us to obtain:

- Information about performance.
- Identification of bottlenecks.
- Mechanisms of optimization.
- Tracking of running threads.
- Time lost in subroutines.

The use of this program is not only limited to the individual sphere. We propose a possible commercial and industrial utility that it could have: the program could be used for video stabilization in surveillance cameras, allowing their images to remain fixed and sharper against movements of almost all kinds.

4 CONCLUSIONS

SIMD instruction sets represents a huge improvement in the performance in a certain types of applications.

As explained in section 2.2, this improvement is very noticeable in multimedia applications since large amounts of data must be handled in the same way. This is the case of the program described in this article.

A video stabilization program has been improved using parallelism techniques with Intel® SSE2 and Intel® AVX2.

5 ACKNOWLEDGMENT

This work has been partially supported by Spanish Science and Technology Ministry in the Project DPI2016-79278-C2-2-R, co-financed by the European Regional Development Fund (ERDF) and MINECO, and by the European Project 'Algerian National Laboratory for Maintenance Education" Project No 586035-EPP-1-2017-1-DZ-EPPKA2-CBHE-JP.

$$SpeedupwithSSE2 \cong 9.19$$

$$SpeedupwithAVX2 \cong 14.65$$

	Time
SISD	184.56 s
SIMD with SSE2	20.07 s
SIMD with AVX2	12.56 d

Figure 8. Speedups with SSE2 and AVX2 obtained with a 7th generation Intel® Core i7 processor.

References:

[1] "Guia docente de Arquitecturas Paralelas. ESEI. Uvigo.", *secretaria.uvigo.gal*, 2019. [Online]. Available: https://secretaria.uvigo.gal/docnet-nuevo/guia_docent/index.php?centre=106&ensenyament=O06G150V01&assignatura=O06G150V01401.

[2] H. Soffar, "C programming language features, advantages and disadvantages", Science online, 2019. [Online]. Available: https://www.online-sciences.com/programming/c-programming-language-features-advantages-and-disadvantages/.

[3] "Netbeans vs. Eclipse: Comparing Two Java IDEs", 2019. [Online]. Available: https://www.upwork.com/hiring/development/netbeans-vs-eclipse/.

[4] "GCC, the GNU Compiler Collection- GNU Project - Free Software Foundation (FSF)", gcc.gnu.org, 2019. [Online]. Available: https://gcc.gnu.org/.

[5] "Cygwin", Cygwin.com, 2019. [Online]. Available: https://www.cygwin.com/.

[6] "OpenCV", Opencv.org, 2019. [Online]. Available: https://opencv.org/.

[7] "Intel® Intrinsics Guide", Software.intel.com, 2019. [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SSE2.

[8] "Taxonomía de Flynn", Parallel Arquitectures class material, Department of Systems and Automation Engineering, University of Vigo, Spain.

[9] "Ley de Amdahl", Parallel Arquitectures class material, Department of Systems and Automation Engineering, University of Vigo, Spain.

[10] A. González, M. García, M. Díaz-Cacho, "Desarrollo de un efecto de mosaico para docencia en la materia de Arquitecturas Paralelas", paper for 'Jornadas de Automática 2018'.

[11] https:///github.com/cesargmr2107/EstabilizacionDeVideo/