PC - Integral Image computation

Luca Angioloni
luca.angioloni@stud.unifi.it

Francesco Pegoraro
francesco.pegoraro@stud.unifi.it

Abstract

We present the Integral Image computation algorithm in three different implementations: a serial implementation in C++, a parallel implementation using OpenMP on the CPU and a parallel implementation using CUDA on the GPU. The parallel approach separates the computational task into rows prefix sums and column prefix sums in order to make use of parallelization in the two phases of the task. We compare the performances of the parallel approaches running on the CPU and on the GPU with the sequential implementation across a range of image sizes and number of threads.

1. Introduction

An Integral Image (or summed-area table) is a data structure and algorithm for quickly and efficiently generating the sum of values in a rectangular subset of a grid.

The element $I_\Sigma(\mathbf{x})$ in position $\mathbf{x} = (x, y)^T$ of the Integral image represents the sum of all the pixels intensities in the input image I inside the rectangular region between the origin and $\mathbf{x}$.

$$I_\Sigma(\mathbf{x}) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i, j) \tag{1}$$

Once the Integral image is created, only three additions are required to compute the sum of the intensities on any rectangular region. (See Fig. 1)

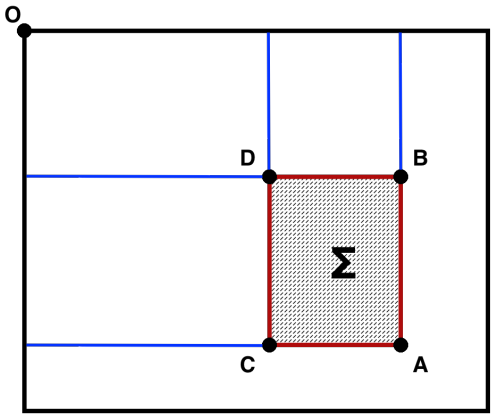

Figure 1. $\Sigma = A - B - C + D$ - Using *Integral Images*, only three additions and four memory accesses are required to compute the sum of the intensities on any rectangular region.

The time to compute this area is independent of the size of the area and is constant, which is very useful in many image processing tasks.

1.1. Integral Image Implementations

As described above, to compute the Integral image in a certain position, all the previous pixels from the input image are needed.

This naive approach would be computationally inefficient and so a recursive method is usually used (this is what the sequential algorithm does, see sub section 1.1.1).

Nonetheless this formulation isn't directly parallelizable and will require a further approach to make use of parallelization (see sub section 1.1.2).

1.1.1 Sequential Integral Image

For an image of size $w \times h$, we calculate the Integral Image using the following algorithm:

- I : input image of dimension $w \times h$.

- size : input size of I (h, w).
- I_Σ : Resulting Integral Image.

Algorithm 1 Sequential Integral Image

```
1:  procedure INTEGRAL-IMAGE(I, size)
2:      for i = 0 to size.h do
3:          for j = 0 to size.w do
4:              v ← I[i, j];
5:              if i >= 1 then
6:                  v ← v + I_Σ[i − 1, j];
7:                  if j >= 1 then
8:                      v ← v+I_Σ[i, j−1]+I_Σ[i−1, j−1];
9:              else
10:                 if j >= 1 then
11:                     v ← v + I_Σ[i, j − 1];
12:             I_Σ[i, j] ← v;
13:     return I_Σ
```

The sequential algorithm has been implemented in C++, with the original image in the form of a 1-D array (of *uint_8* type) and the resulting Integral Image again as a 1-D array (of *unsigned long int* type).

The sequential algorithm can be visualized in Fig. 2.

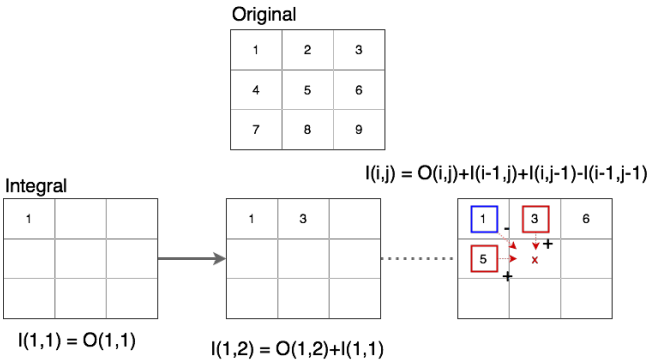

Figure 2. Figure to help visualize the sequential algorithm.

1.1.2 Parallel Integral Image

In order to parallelize the Integral Image procedure, it has been reformulated separating the task into two subsequent tasks, first computing the prefix sums along the rows of the image **independently** and then computing the prefix sums along the columns of the resulting image **independently**.

An example of this concept is shown in Fig. 3.

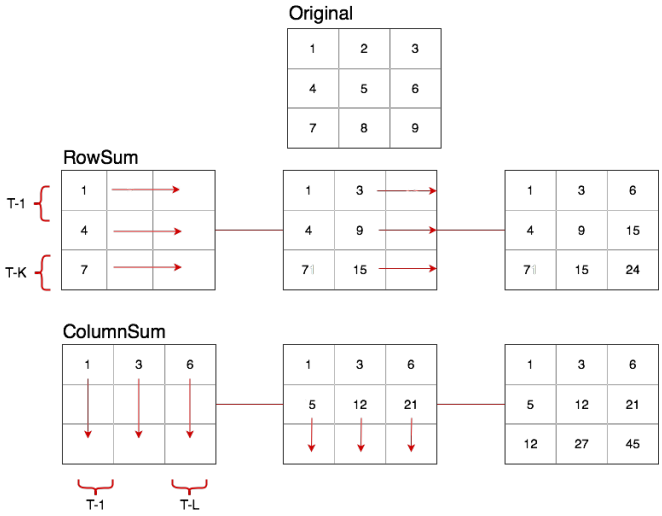

Figure 3. Figure to help visualize the parallel algorithm.

Being the image, in practice, represented as a 1-D array, in order to maximize the cache usage, between the two tasks, the resulting image of the first phase is transposed and the prefix sums along the columns are instead done on the rows of the transposed image; finally the resulting image is one last time transposed in order to obtain the expected result.

So the Integral image of an image I of size $w \times h$ is computed using the following algorithm:

- I : input image of dimension $w \times h$.
- size : input size of I (h, w).
- I_Σ : Resulting Integral Image.

Algorithm 2 Parallel implementation of the Integral Image algorithm

1: **procedure** INTEGRAL-IMAGE-PARALLEL(I, size)
2: $rows \leftarrow []$; *// empty array, same size as I*
3: *// this for is done in parallel*
4: **for** $i = 0$ to $size.h$ **do**
5: $rows[i, 0] \leftarrow I[i, 0]$;
6: **for** $j = 1$ to $size.w$ **do**
7: $rows[i, j] \leftarrow I[i, j] + rows[i, j - 1]$;
8: $rows^T \leftarrow Transpose(rows)$;
9: *// this for is done in parallel*
10: **for** $i = 0$ to $size.w$ **do**
11: $I^T_\Sigma[i, 0] \leftarrow rows^T[i, 0]$;
12: **for** $j = 1$ to $size.h$ **do**
13: $I^T_\Sigma[i, j] \leftarrow rows^T[i, j] + I^T_\Sigma[i, j - 1]$;
14: $I_\Sigma \leftarrow Transpose(I^T_\Sigma)$;
15: *return* I_Σ

1.1.3 OpenMP

The first technology used to implement the parallel algorithm was OpenMP. *Open Multi-Processing* is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

This implementation has been written and developed using the C++ programming language.

Usually, one of the advantages of this technology is that no big changes to the sequential code are required. In this case, we reformulated the algorithm as described in section 1.1.2 and then introduced the usual OpenMP compiler directives. The two for cycles are computed in parallel by OpenMP with the directive: *#pragma omp parallel for*.

Particularly with this technology, the introduction of the transpose operations in the routine, gave a significant boost in the overall performances allowing the multiple thread in the column sums phase to access different lines of cache, not interfering with each other and speeding up the process.

1.1.4 CUDA

The second technology used to implement the parallel algorithm was CUDA.
CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. The CUDA platform is designed to work with programming languages such as C, C++, and Fortran.

This implementation has been written and developed using the C++ programming language and follows the pseudo code presented in the Algorithm 2.

2. Results

All the results, in terms of optimal number of threads and execution time, are obtained on a machine with *Dual Socket Intel(R) Xeon(R) E5-2620 v3 @ 2.40GHz* CPU (total 24 cores) and a *Nvidia Titan X* GPU.

All the tests have been performed using the same set of images of increasing size, multiple times and in a controlled environment. The image sizes (heights) used: $[100, 200, 500, 1000, 2000, 5000, 10000]$.

2.1. Number of thread

First, for each parallel technology, the implementation has been tested with different numbers of threads (from 1 to 4096 - and even 10000 with CUDA), using the same image (10000×16000), in order to evaluate the performances in each configuration and the number of threads that yields the best results.

With the OpenMP technology, the best results are obtained using 64 threads in a machine that has 24 cores in the CPU, meaning 2.6 threads per core. (See figure 4)

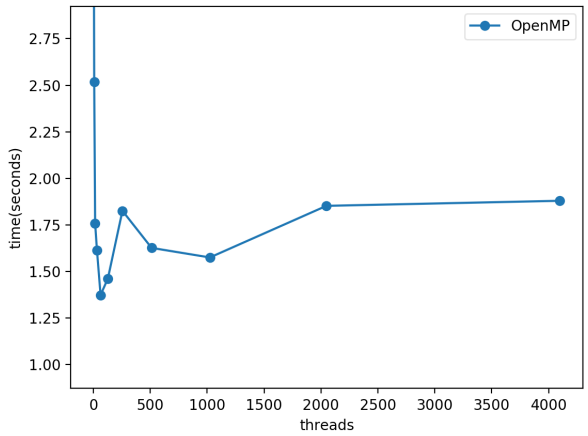

Figure 4. OpenMP - In this plot, along the x axis, the different numbers of threads tested; on the y axis the mean time to compute the integral image on the 10000×16000 image.

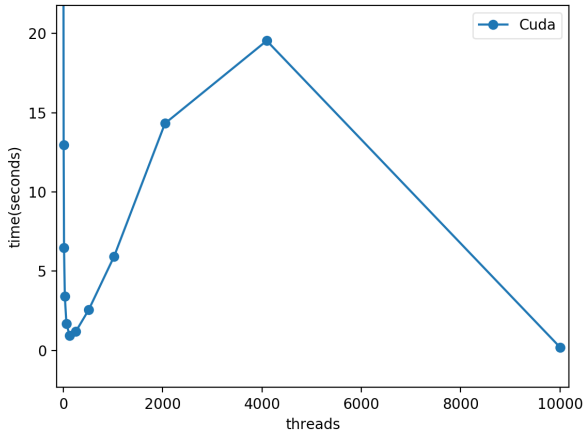

Figure 5. CUDA - In this plot, along the x axis, the different numbers of threads tested; on the y axis the mean time to compute the integral image on the 10000×16000 image.

With the CUDA technology, the best results are obtained using at least a thread per row (or column) of the image in each parallel task. Infact, using the 10000×16000 image, the best performances are achieved using 10000 threads in the row sums and 16000 threads in the column sums. Decent results, but far from the best, are obtained with 128 threads. (See figure 5)

2.2. Serial vs Parallel

Once evaluated the optimal number of threads for each implementation, the different approaches have been compared across the set of images of varying and increasing size.

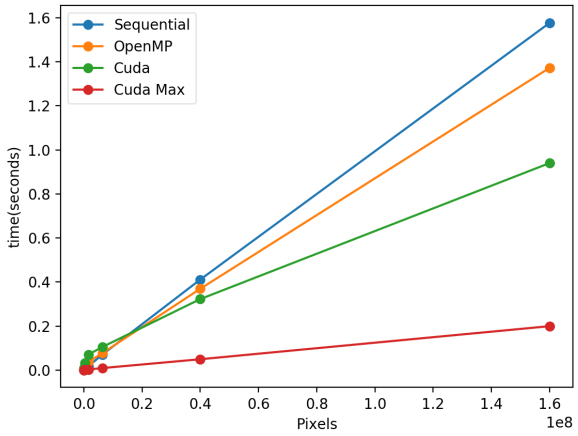

Figure 6. In this plot, along the x axis, the number of pixels elaborated; on the y axis the mean time to compute the integral image on this pixels. (Here Cuda - green is the CUDA implementation with 128 threads, while Cuda Max - red is the CUDA implementation with as many thread as rows or columns)

As shown in figure 6 the serial approach is the slowest compared to all the other parallel approaches. In general with really small images the serial implementation has comparable performances with the other approaches, but from a certain size up it is outperformed.

The fastest implementation in every circumstance is CUDA with as many threads as rows and columns, confirming what the modern trend for scientific computation is highlighting: GPUs are suited for these tasks.

2.2.1 Speed Up

In figures 7 and 8 is shown the speed up obtained with the different parallel approaches with relation to the serial algorithm.

While OpenMP achieves a small speed up factor, CUDA speeds the task up until more than 8 times the serial implementation.

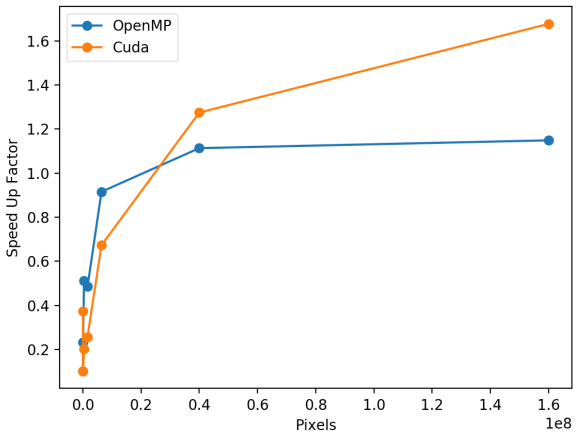

Figure 7. The speed up obtained with OpenMP and CUDA 128 threads

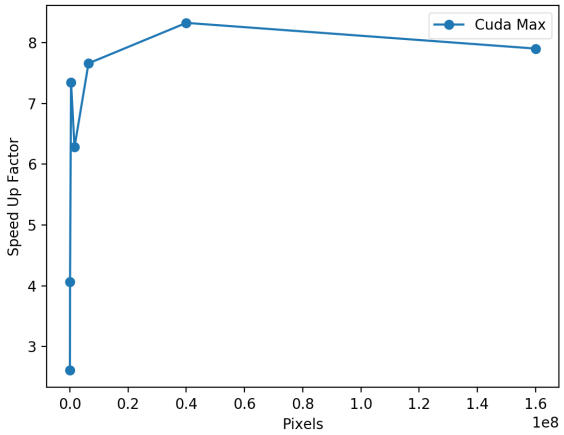

Figure 8. The speed up obtained with CUDA with as many threads as rows or columns