Parallel Computing Fundamentals: 2D Pattern Recognition, CUDA and OpenMP implementations

Alberto Baldrati
alberto.baldrati@phd.unipi.it

Lorenzo Agnolucci
lorenzo.agnolucci@unifi.it

Abstract

In this work, we present a comprehensive study on the performance of the 2D pattern recognition algorithm. We compare a sequential version with two parallel ones: the first using OpenMP on CPU and the second exploiting GPU with CUDA. The results show that, due to the embarrassingly parallel structure of the 2D pattern recognition algorithm, the parallel implementations obtain noteworthy speedups.

1. Introduction

The pattern recognition algorithm consists in locating the closest portion to a given query in a data target. In our experiments, we focused on 2D pattern recognition, so both the queries and the targets are matrices (*e.g.* images). To assess the similarity between the query and the target, we need to define a distance metric. The metric we relied on is the Sum of Absolute Differences (SAD). Let $T \in \mathcal{N}^{m\times n}$ and $Q \in \mathcal{N}^{r\times c}$ be the target and query matrices, the algorithm computes the similarity value for each possible shift of Q w.r.t. T. The output is a matrix $SAD \in \mathcal{N}^{(m-r+1)\times(n-c+1)}$ in which each element (i, j) is given by:

$$SAD(T,Q,i,j) = \sum_{k=0}^{r}\sum_{l=0}^{c}|T_{i+k,j+l} - Q_{k,l}| \qquad (1)$$

The minimum value in the SAD matrix indicates the portion of the target matrix closest to the query matrix.

Figure 1 shows a schematic example of the algorithm. In Figure 2 is displayed an application of the algorithm to a real-world image using a random query matrix.

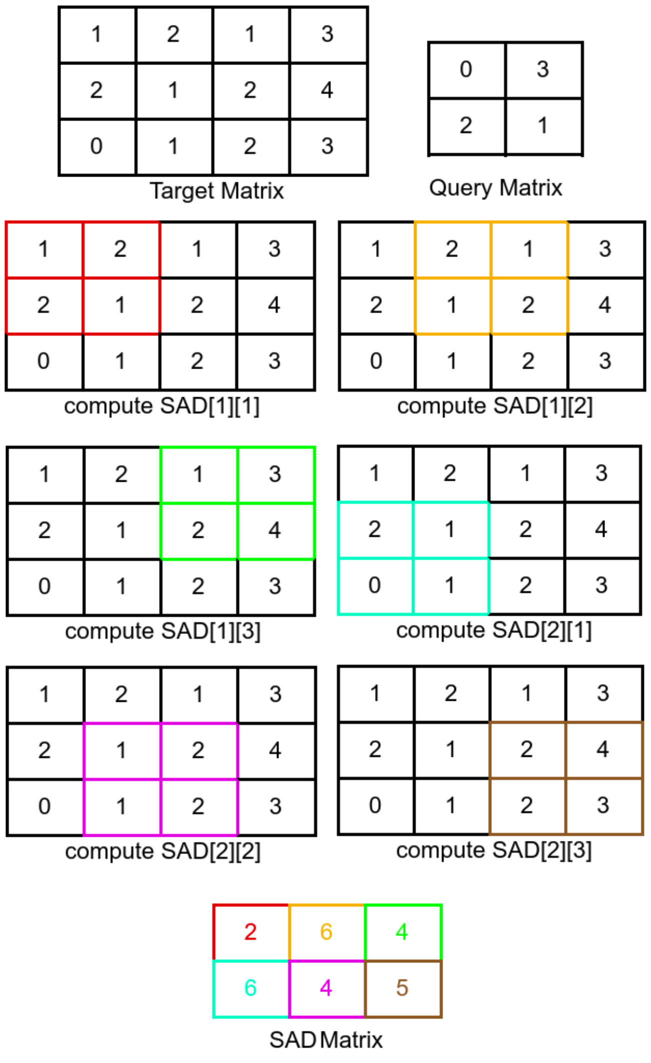

Figure 1. Schematic example of 2D pattern recognition algorithm

2. Algorithm

2.1. Sequential version

We formalize the 2D pattern recognition algorithm with pseudocode in Algorithm 1 and Algorithm 2:

1. Algorithm 1 is fundamentally a one-to-one translation of Equation (1): this subroutine computes and returns a single value of the SAD matrix.

2. Algorithm 2 is the outer cycle that iterates over the rows and columns of the target matrix to compute the

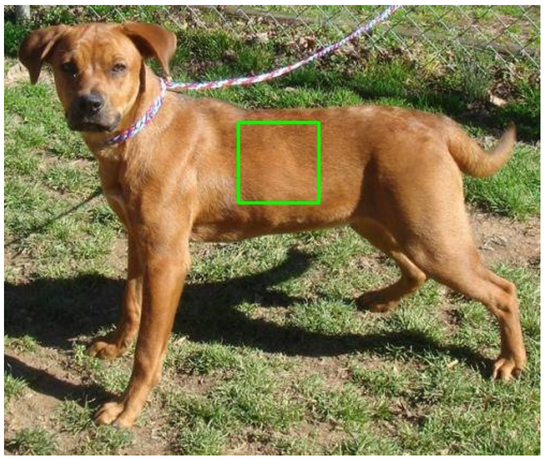
Figure 2. 2D pattern recognition algorithm applied to an image. The green box represents the best match using a random query matrix.

SAD matrix. This is the embarrassingly parallel section of the algorithm, and thus the one we will focus on in the parallel implementations.

Algorithm 1: computeSAD

```
Data: queryMatrix Q, targetMatrix T,
      startRowIndex i, startColIndex j
Result: localSadValue
1 localSadValue = 0
2 for from $k = 0$ to $Q.rows$ do
3     for from $l = 0$ to $Q.cols$ do
4         targetV = T[i+k][j+l]
5         queryV = Q[k][l]
6         localSadValue += | targetV - queryV |
7 return localSadValue
```

Algorithm 2: PatternRecognition

```
Data: queryMatrix Q, targetMatrix T
Result: SADMatrix S
1 Define SADMatrix S
2 S.rows = T.rows - Q.rows + 1
3 S.cols = T.cols - Q.cols + 1
4 for from $i = 0$ to $S.rows$ do
5     for from $j = 0$ to $S.cols$ do
6         S[i][j] = computeSAD(P,Q,i,j)
7 cx, cy = argmin(S)
```

2.2. Time Complexity Analysis

The algorithm has four nested loops: two in the outer cycle that iterates over rows and columns and two in the computation of each SAD matrix value. To analyze the time complexity of the algorithm we define m and n as target matrix rows and columns and r and c as query matrix rows and columns.

2.2.1 Sequential example

The complexity of Algorithm 1 is

$$SADcomplexity = r * c \tag{2}$$

The complexity of Algorithm 2 (*i.e.* the complexity of the entire 2D pattern recognition algorithm) is:

$$(m - r + 1) * (n - c + 1) * SADcomplexity \tag{3}$$

2.2.2 Parallel example

As we said before Algorithm 2 is embarrassingly parallel, so it is suitable for an effective parallelization. Assuming that we use a number of threads equal to N_t, the complexity becomes:

$$\frac{(m - r + 1) * (n - c + 1)}{N_t} * SADcomplexity \tag{4}$$

From Equation (4) we can infer that in an ideal case, i.e. without overhead and with a number of cores equal to N_t, we should achieve a linear speedup up to N_t due to the structure of our algorithm.

3. Parallel implementations

We have developed two parallel versions of the sequential algorithm, the first one written in C++ using the OpenMP [3] library, the second one written in CUDA [5] for leveraging the parallel power of GPUs.

3.1. OpenMP

OpenMP [3] is a library that can easily transform a sequential program into a parallel one, with the only addition of #pragma directives.

This library suffers from some limitations, particularly when synchronization operations between threads or asynchronous operations are to be performed. For our purposes these limitations are not a problem, in fact the section of code we want to parallelize is embarrassingly parallel.

Algorithm 3 presents the pseudocode of the OpenMP implementation.

In the #pragma directive there are some points of interest:

Algorithm 3: PatternRecognition *OpenMP version*

```
Data: queryMatrix Q, targetMatrix T, numThread $N_t$
Result: SADMatrix S
1 Define SADMatrix S
2 S.rows = T.rows - Q.rows + 1
3 S.cols = T.cols - Q.cols + 1
4 #pragma omp parallel for num_threads($N_t$)
    collapse(2) schedule(static)
5 for from $i = 0$ to S.rows do
6     for from $j = 0$ to S.cols do
7         S[i][j] = computeSAD(P,Q,i,j)
8 cx, cy = argmin(S)
```

- For our purposes we do not need dynamic scheduling because the workload is well balanced since the number of tasks is fixed a priori. To avoid useless overhead we use static scheduling.
- Given that we want to parallelize over rows and columns wee need nested parallelism.

3.2. CUDA

With the CUDA [5] parallel implementation of the 2D pattern recognition algorithm we want to exploit the massive computing power of GPUs.

One of the disadvantages of the CPUs over the GPUs is that they suffer from the context switching cost when they are exposed to a number of active threads higher than the cores count. Conversely, the GPUs, having a huge number CUDA cores, are able to manage an high amount of threads. Therefore, to fully leverage the high cores count of a GPU we start a thread for each element of the SAD matrix.

From now on, we will use the CUDA terminology [1]. In NVIDIA GPUs, threads are organized in *blocks*. Blocks can have different dimensions in order to adapt to the shape of data. Since our data are matrices we use 2-dimensional blocks. The threads composing each block are executed in parallel (by the SIMT paradigm) in group of 32 named warps. It is good practice keep the block size multiple of this number.

The CUDA implementation is split in two sections: the kernel launch and the kernel function, reported in Algorithm 4 and Algorithm 5, respectively.

In Algorithm 4 we can see that our 2-dimensional blocks size is TILE_WIDTH× TILE_WIDTH. In our experiments in Section 4 we will use TILE_WIDTH dimension equal to 8, 16, 24 and 32 for maintaining a number of threads in a block multiple of 32.

In Algorithm 5 we can notice that the access to the target matrix is coalesced. Indeed T[i+row][j+col] is in the

Algorithm 4: Kernel Launch

```
Data: queryMatrix Q, targetMatrix T,
      SADMatrix S, TILE_WIDTH
1 DimGrid(ceil(S.rows / TILE_WIDTH),
    ceil(s.cols / TILE_WIDTH));
2 dimBlock(TILE_WIDTH, TILE_WIDTH)
3 PatternRecognitionKernel
    <<<dimGrid, dimBlock>>>(Q,T,S)
```

Algorithm 5: PatternRecognitionKernel

```
Data: queryMatrix Q, targetMatrix T, SADMatrix S
1 bx = blockIdx.x
2 by = blockIdx.y
3 tx = threadIdx.x
4 ty = threadIdx.y
5 col = bx * blockDim.x + tx
6 row = by * blockDim.y + ty
7 if row<S.rows and col<S.cols then
8     for from $i = 0$ to Q.rows do
9         for from $j = 0$ to Q.cols do
10            targetV = T[i+row][j+col]
11            queryV = Q[i][j]
12            localSadValue += | targetV - queryV |
13    S[row][col] = localSadValue
```

form of T[(expression with terms independent of tx) + tx] that implies that with a single access more requests to the VRAM will be satisfied at the same time [1]. It is also important to notice that in the kernel function the target and query matrices are not modified, so we can mark them as *const* and *restrict* to perform pointer aliasing optimization. In this way the compiler is sure that both matrices are read-only data and can use a cache designed for this type of data, available for devices with compute capability greater than 3.5 [2].

3.2.1 Tiled implementation

For an extra performance boost we tried to implement a tiled version of the 2D pattern recognition algorithm, which does not only use the global memory of the GPU but the shared memory as well [4].

The shared memory can be seen as a programmable cache: it is a memory inside the GPU, characterized by a little dimension and very low latency. The values of this memory are logically shared between the element of a block, so it is useful when there is a massive reuse of the same values in the same block.

Given that the size of the query matrix can be very large, the algorithm does not suit well for the shared memory usage and our tiled implementation performs worse than the the global memory only version.

4. Experimental Result

Several experiments have been carried out to compare execution times and speedups of the sequential and parallel versions.
The speedup S_P is computed as

$$S_P = \frac{t_s}{t_p} \tag{5}$$

where t_s and t_p are the execution times of the sequential and parallel version, respectively.

The tests have been conducted on an Ubuntu 20.04 LTS machine equipped with:

- Intel Core i7-4790 3.6GHz with Turbo Boost up to 4Ghz, 4 cores/8 threads processor
- RAM 16 GB DDR4
- NVIDIA GeForce 750 2GB compute-capability 5.0 (running on CUDA 10.1)

The high precision C++11 library *chrono* has been used for measuring the execution time.

Each time has been measured running each test 5 times and taking the average as a result. In this section we provide mostly graphs for a better understanding of the results, in Section 6 are available the complete numerical results. We have conducted experiments on three different combinations of query and target matrices sizes:

- **Test1**: target matrix with size 1500×1500 and query matrix with size 150×150
- **Test2**: target matrix with size 2000×2000 and query matrix with size 200×200
- **Test3**: target matrix with size 2500×2500 and query matrix with size 250×250

4.1. OpenMP Result

Figure 3, Figure 4 and Figure 5 show the results of our OpenMP implementation. In each test we used a number of threads ranging from 1 (sequential version) to 10.

In the graphs we can identify three sections, which correspond to the three different colors:

- **Blue** section where each thread can be executed on a different **physical core**
- **Yellow** section where each thread can be executed on a different **logical core**

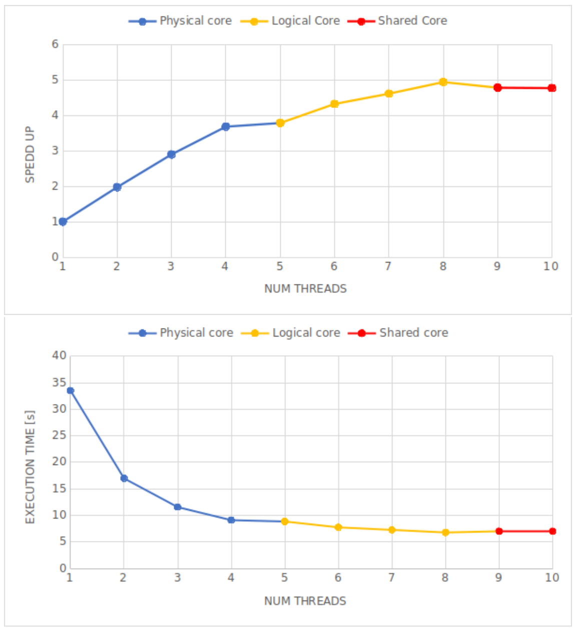

Figure 3. Test1 speedup and execution time varying the number of threads using OpenMP

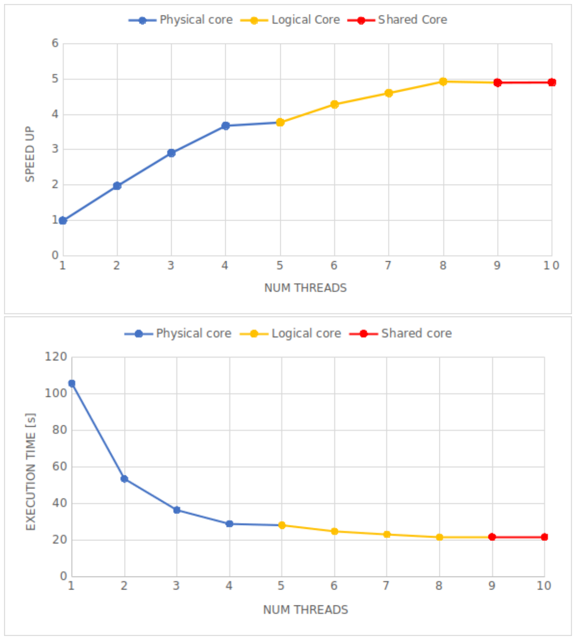

Figure 4. Test2 speedup and execution time varying the number of threads using OpenMP

- **Red** section where threads have to **share** both physical and logical core

All three combinations of target and query matrix show very similar (almost identical) behaviors, therefore we can infer that the performance of our implementation is independent from the size of the matrices.

In every speedup graph we can notice that in each area

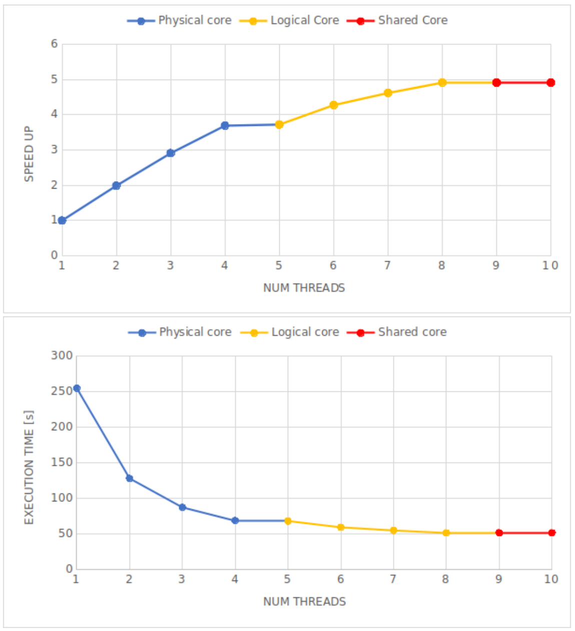

Figure 5. Test3 speedup and execution time varying the number of threads using OpenMP

we have a different speedup evolution: in the blue area it is almost **linear**, in the yellow area is **sub-linear** and in the red area we have no more speedup.

4.2. CUDA results

Figure 6, Figure 7 and Figure 8 show the results of our CUDA implementation. In each test we have used four TILE_WIDTH dimensions and three different implementations:

- The implementation which uses global memory and the optimization for pointer aliasing (blue line in the graph with the name **Global1**)
- The implementation which uses global memory but does not use the optimization for pointer aliasing (yellow line in the graph with the name **Global2**)
- The implementation which uses shared memory (orange line in the graph with the name of **Shared**)

The graphs confirm that is the shared memory implementation does not perform well and that the pointer aliasing optimization gives a good performance boost. In all three tests the best version is the one which uses such optimization and with TILE_WIDTH equal to 16.

4.3. OpenMP and CUDA comparison

In both CUDA and OpenMP experiments we have used the same matrices and we have measured the time in the same way, so we can compare the execution time between

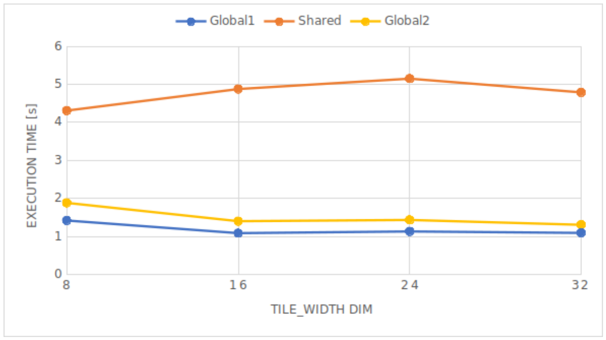

Figure 6. Test1 execution time using different implementation and TILE_WIDTH dimension

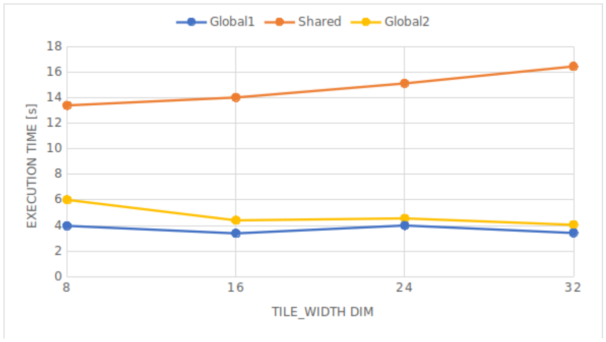

Figure 7. Test2 execution time using different implementation and TILE_WIDTH dimension

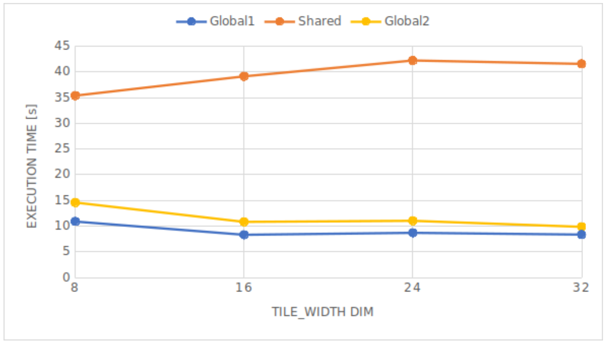

Figure 8. Test3 execution time using different implementation and TILE_WIDTH dimension

the different implementations. For each test we take only the best time achieved with each different implementation. In Table 1 can be seen that the CUDA implementation abundantly outperforms both the sequential and the OpenMP ones, at the expense of an higher development cost. However OpenMP lets to achieve a noticeable speedup with just a single directive.

5. Conclusions

In this paper we show how the computationally expensive 2D pattern recognition algorithm can be parallelized. The OpenMP implementation achieves a good speedup (of

	Sequential time	*OpenMP time*	*OpenMP speedUp*	*CUDA time*	*CUDA SpeedUP*
Test1	33,52s	6,79s	**4,94x**	1,09s	**30,75x**
Test2	105,68s	21,47s	**4,92x**	3,39s	**31,17x**
Test3	254,50s	51,80s	**4,91x**	8,31s	**30,62x**

Table 1. Comparison and speedup between CUDA and OpenMP version

about *5x*) directly on CPU with minimal code changes. However the use of GPUs brings further improvement reaching a speedup of about *31x*. From our experiments we can state that for the 2D pattern recognition algorithm the GPU processing is heavily faster than the CPU one.

References

[1] NVIDIA, CUDA documentation, `https://docs.nvidia.com/cuda/`, 2019.

[2] J. Appleyard. Cuda pro tip: Optimize for pointer aliasing, `https://devblogs.nvidia.com/cuda-pro-tip-optimize-pointer-aliasing/`, 2014.

[3] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[4] M. Harris. Using shared memory in cuda c/c++, `https://devblogs.nvidia.com/using-shared-memory-cuda-cc/`, 2013.

[5] NVIDIA, P. Vingelmann, and F. H. Fitzek. Cuda, release: 10.1.89, 2020.

6. Appendix

In this section we provide all numerical results for the tests.

OpenMP Test1		
Num threads	Execution time	speedup
1	33,52s	1,00x
2	16,98s	1,97x
3	11,57s	2,90x
4	9,11s	3,68x
5	8,86s	3,78x
6	7,76s	4,32x
7	7,27s	4,61x
8	**6,79s**	**4,94x**
9	7,01s	4,78x
10	7,03s	4,77x

Table 2. Test1 OpenMP speedup and execution time numerical results, graph displayed in Figure 3

OpenMP Test2		
Num threads	Execution time	speedup
1	105,68s	1,00x
2	53,47s	1,98x
3	36,34s	2,91x
4	28,76s	3,67x
5	28,04s	3,77x
6	24,70s	4,28x
7	22,99s	4,60x
8	**21,47s**	**4,92x**
9	21,61s	4,89x
10	21,59s	4,90x

Table 3. Test2 OpenMP speedup and execution time numerical results, graph displayed in Figure 4

OpenMP Test3		
Num threads	Execution time	speedup
1	254,50s	1,00x
2	128,09s	1,99x
3	87,53s	2,91x
4	69,01s	3,69x
5	68,40s	3,72x
6	59,60s	4,27x
7	55,16s	4,61x
8	**51,80s**	**4,91x**
9	51,85s	4,91x
10	51,88s	4,91x

Table 4. Test3 OpenMP speedup and execution time numerical results, graph displayed in Figure 5

CUDA Test 1			
TILE_WIDTH	Global1	Shared	Global2
8	1,42s	4,31s	1,88s
16	**1,08s**	4,88s	1,40s
24	1,13s	5,15s	1,43s
32	1,10s	4,79s	1,31s

Table 5. Test1 CUDA execution time numerical results, graph displayed in Figure 6

CUDA Test 2			
TILE_WIDTH	Global1	Shared	Global2
8	3,97s	13,40s	6,01s
16	**3,39s**	14,02s	4,41s
24	4,00s	15,12s	4,56s
32	3,42s	16,45s	4,05s

Table 6. Test2 CUDA execution time numerical results, graph displayed in Figure 7

CUDA Test 3			
TILE_WIDTH	Global1	Shared	Global2
8	10,90s	35,40s	14,59s
16	**8,31s**	39,15s	10,82s
24	8,69s	42,23s	11,03s
32	8,34s	41,56s	9,86s

Table 7. Test3 CUDA execution time numerical results, graph displayed in Figure 8