Parallel Hessian-Affine detector with SIFT descriptor

SPM Final Project 2016/2017

Luca Lovagnini

May 31, 2017

1 Introduction

1.1 Local Features Detection and Description Algorithms

The Scale Invariant Feature (SIFT) algorithm has been introduced in 1999 by Lowe et al. [Low99] and represented a revolution in Computer Vision to describe image feature, i.e. unique patches in the input image which can be compared and easily tracked.

Figure 1: A graphical representation of SIFT keypoints

Understanding the details of this algorithm is beyond the scope of this work, however some notions are necessary in order to understand this report. The result of these algorithm can be divided in two elements: 1. A list of *keypoints*, i.e. spatial locations that define something interesting in the image. They are usually represented as 2D coordinates along with other parameters (e.g. size, angle, response, etc.). Figure 1 is an example of the set of detected keypoints on an input image by SIFT. 2. For each keypoint, we have a *feature descriptor*, i.e. a D-dimension vector ($D = 128$ in SIFT) which describes the region identified by its keypoint.

How keypoints are generated is algorithm dependent, but a common approach is to use multiple scales in multiple octaves, each one of a different size. Since these two notions are crucial to understand this project, we can see in Figure 2 three octaves of different sizes, each one divided into three scales.

Depending on the considered application, we may need only keypoints, descriptors or both. For example, solutions for Content Based Image Retrieval (CBIR) applications such as VLAD [JDSP10] or Fisher Vectors [JPD+12] only descriptors are needed, since we don't care the actual positions of the local features.

Since SIFT, an huge number of local feature algorithms has been proposed, such as SURF [BTVG06], ORB [RRKB11] and many others. The main difference among these algorithms is the trade-off between feature accuracy and speed. In particular, in 2002 Mikolajczyk et al. proposed the Hessian-Affine (HA) keypoint detector [MS02], which is often considered the state of the art in local feature detection [MTS+05] in terms of feature quality. In 2009 the HA detector has been combined (and optimized in terms of precision) with the SIFT descriptor [PCM09] and a C++ implementation can be found on GitHub. From now on, we will refer to this approach as the Heassian-Affine algorithm.

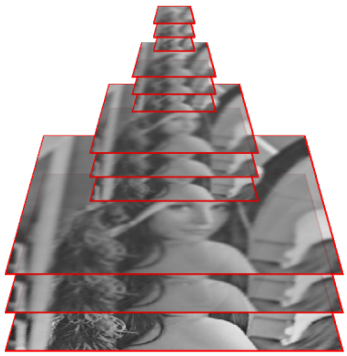

Figure 2: Flowchart of the Hessian-Affine algorithm

1.2 Project Requirements and Goals

The aim of this project is to propose a parallel version of the given implementation of the cited implementation, namely the Parallel Hessian-Affine (PHA) algorithm. These are some requirements, specifications and goals of the project:

1. Since this is the final project for a parallel systems exam, the proposed parallel solution will be tested on a 64-cores machine.

2. Image size is an important factor in local feature detection and descriptions algorithms. As it will be shown in section 4, the serial algorithm time execution is not linear w.r.t. the image size. In many applications where these kind of algorithms are used (e.g. CBIR systems), using high-resolution images dramatically improves the results. In real-time applications, smaller images are used, but usually different kinds of local features algorithms are used (e.g. BRIEF [CLSF10]). For these reasons, our approach will be tested on large size images, as explained in section 4.1.

3. Since this project will be proposed as an OpenCV contribution, OpenMP will be used as API. As it will be shown in section 2.2 this choice includes some performance limitations.

4. It's beyond the scope of this project to modify the original algorithm workflow to make it more suitable on parallel systems, as it usually happens in related works (e.g. [ZCZX08]).

5. For simplicity, we will consider only the descriptors matrix as the algorithm result, without considering the image keypoints.

6. The original algorithm has many parameters (e.g. the number of scales per octave) and their value may strongly influence the execution time of the algorithm. However, in this project we will consider only the default values of them.

2 Hessian-Affine Optimization

2.1 Serial Algorithm

As already said, it is behind the scope of this project to explain the details of the Hessian-Affine algorithm. However, it is important to illustrate the workflow of the original (sequential) algorithm to understand the author's choices to parallelize it. In Figure 3 we can see the two main loops of the Hessian-Affine algorithm and the most important functions used inside them. In particular, we can identify the three outer most functions: 1. `gaussianBlur` 2. `hessianResponse` 3. `findLevelKeypoints` According to Intel VTune Amplifier, more than 85% of the execution time is spent inside `findLevelKeypoints`, so we can consider it as the most time consuming function in our algorithm.

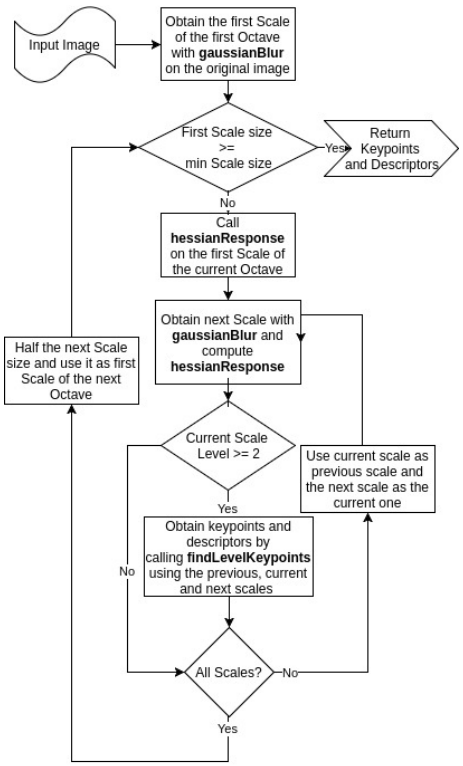

Figure 3: Flowchart of the Hessian-Affine algorithm

From this flowchart, it is very clear that there is a strong dependency between each loop iteration: for each scale level (i.e. the innermost loop), we have to use the scales of the previous and next levels in order to compute the keypoints and descriptors. Of course, this happens only starting from the second scale level (it doesn't exist a previous level at the first one).

In addition, the first scale of the considered octave is obtained by downsampling the last level of the previous octave (except for the first octave). In other words, the input of the i-th iteration of the external loop iteration (i.e. the first scale of the considered octave) is given by the result of the last `gaussianBlur` of the *(i-1)*-th external loop iteration (i.e. the last scale of the previous octave).

Because of these dependencies between different iterations, it is obvious that a straightforward parallelization is not an option for this algorithm. However, the number of iterations for both loops are deterministic (which is necessary for parallelization): the number of octaves (i.e. the number of iterations in the outer most loop) are logarithmic w.r.t. the smallest side of the input image, while the number of iterations of the innermost loop (i.e. the number of scales per octave) is an application parameter.

2.2 Parallel Algorithm

2.2.1 Ideal Algorithm

As we have seen in the previous section, we can identify three main functions in the Hessian Affine algorithm: for a given octave and scale, the keypoints and descriptors obtained by `findLevelKeypoints` depends both from `gaussianBlur` and `hessianResponse` (which depends from `gaussianBlur` itself). So, since it's a good practice to paralellize the outer most loops, the optimal parallel version of the Hessian-Affine algorithm is represented in Figure 4. Since the input image is halved for each octave and since each function complexity depends on the image size, it's clear that using a static scheduling leads to bad performance due to load unbalance. Compared to a default static solution, from experimental results `schedule(dynamic,1)` always improves performances for each parallel loop in our algorithm, even considering larger dynamic chunks. Notice that the final part of the algorithm is a critical section. However, it doesn't introduce any significant synchronization overhead because OpenCV matrix implementation is kinda of a smart pointer, so appending one matrix to another is a very cheap and fast operation. Finally, instead of creating a parallel region per parallel loop, we wrapped all the OpenMP parallel loops in one parallel region to reduce the library overhead introduced by each parallel region.

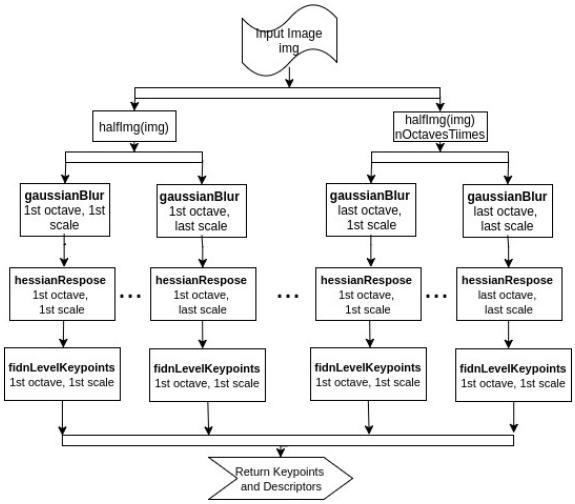

Figure 4: Flowchart of the Hessian-Affine algorithm

In the next sections we will see how it is not possible to implement this optimal solution or how it leads to bad performance.

2.2.2 Parallel Gaussian Blurs

As we have already seen in the previous section, each scale blur depends from the scale before, and each first scale depends on the last scale of the octave before. This is a so-called chain of gaussian blurs. However applying multiple, successive gaussian blurs to an image has the same effect as applying a single, larger gaussian blur, whose radius is the square root of the sum of the squares of the blur radii that were actually applied. This means that is possible to compute all the different gaussian blurs that we need independently from each other, and so it's possible to parallelize them. However, this approach introduce small cumulative errors for each gaussian blur and the number of detected keypoints is greater than 50% w.r.t. the original application. This difference is too much great to be considered acceptable, so the only solution left is to compute the chain of gaussian blurs in a serial way, without parallelizing this portion of the algorithm.

2.2.3 Splitting Keypoints Detection and Descriptors Computation

In the ideal algorithm reported in Figure 4 we assumed that each function call required the same amount of time. This not true at all: an huge spin time has been detected with this approach, caused by different execution times of `findLevelKeypoints` (the most time expensive function in our algorithm). This is because larger octaves generates more keypoints, moreover the number of keypoints created by different scales of the same octave is not the same. Therefore, the number of descriptors built by threads which process large scales is order of magnitudes larger compared to smaller scales, obtaining an high unbalanced workload between threads.

The solution is to split the keypoints detection and the descriptor building: the complexity required to obtain a descriptor does not depend on the scale or octave of the correspondent keypoint, which means that most of the descriptors take the same amount of time to be computed. Descriptors in `findLevelKeypoints` are computed by calling `onHessianKeypointDetected`. In our parallel algorithm, each thread has a local vector where each element is the set of arguments to pass to `onHessianKeypointDetected`. This local vector is then appended to a shared vector between all threads in a critical section. The length of each local vector is highly unbalanced, which

proves the load imbalance to obtain the set of descriptors. A first version of this solution was based on an OpenMP reduction clause, but from experimental results we noticed that this approach is slower w.r.t. the critical section approach. This is because the reduction operation (in this case vector appending) is performed by a single thread once all the keypoints have been detected, while in the critical section approach described above each thread can append its local vector without waiting the other threads, so the last thread has to append only its own local vector (instead of appending the vectors all together). As final result, the synchronization overhead introduced by this critical section is negligible.

2.2.4 Keypoint Detection Load Imbalance

After the critical section previously described, each thread meets a barrier before starting to compute the image descriptors. This is necessary because otherwise each thread would start to generate descriptors without using the final version of the shared vector described in the previous paragraph, resulting in partially results. From a first analysis, where few (but powerful) cores and small images has been used, keypoints detection seemed to take less than 10% of the descriptors computation time, so a potential load imbalance in keypoints detection was not considered a problem. However, experiments on larger images and using many cores showed how this can lead to a significant load imbalance, so most of the threads would wait a long time because of the barrier previously described. The problem is caused by the same reason of the previous section: large scales require more time to detect keypoints than smaller ones. Since keypoints detection in `findLevelKeypoints` consists in two nested loops based on the image height and width, a possible solution could have been to collapse these two nested internal loops with the parallel for containing them (i.e. the parallel for calling `findLevelKeypoints`). With this method problably we would have obtained a perfect load balancing in keypoint detection. However, the two inner loop ranges are based on the width and height of the considered octave and scale, i.e. the number of inner loop iterations depend on the external loop element, which is not allowed in OpenMP. So apparently there is no solution to solve this load imbalance. However, the barrier previously described is necessary to obtain consistent results. A good solution for keypoint imbalance is to use the shared vector as a shared synchronized queue: the Intel Threading Building Block (TBB) is known to offer well-implemented data structures to be used in parallel applications and `concurrent_bounded_queue` is a perfect solution for this problem. Luckily, OpenCV can be built with TBB, so this choice is still compatible with the project requirements. Otherwise, similar results could have been obtained by using a combination of some standard queue and `std::condition_variable` (or similar). In section 4.3 we will see the overhead introduced by this data structure.

2.2.5 Descriptor Building Load Imbalance

Since we split keypoint detection and descriptor computation as described in 2.2.3, now descriptor computation is performed in `findAffineShape`. Depending on the considered image, `cv::GaussianBlur` called in `normalizeAffineShape` (one of the `findAffineShape`'s internal functions), can take between 35% and 75% of the overall time. In particular, less than 1% of the descriptors require such a long time to call `cv::GaussianBlur` that around 10% of the cores on the tested architecture (see section 4 for details) compute only one descriptor during the whole execution, while if we had a well balanced algorithm each core should compute hundreds or thousands of descriptors. From now on, we will refer to this small subset of time consuming descriptors as *big descriptors*.

It's true that some descriptors are more meaningful than others (and so they are important in applications where they are used), but according to experiment results, the big descriptors are not important. Actually, the Mean Average Precision of the application where the parallel algorithm correctness was tested is around 1% higher without considering these big descriptors. In conclusion, this means that we can obtain much better load balance in descriptors building by not computing the big descriptors. Luckily, it's very easy to identify them: we can notice that all the big descriptors are relative to the biggest input image patches for `cv::GaussianBlur` in `normalizeAffine`.

Let's consider p as the patch's size (a square image) for a given keypoint, w and h as the input image width and height respectively). If the following ratio r:

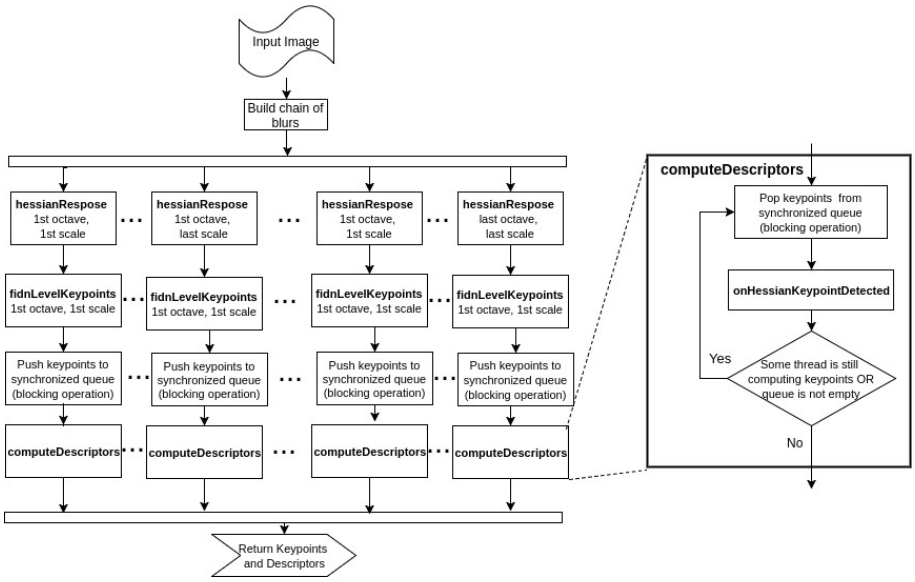

Figure 5: Parallel Hessian-Affine flowchart

$$r = \frac{p}{\max(w, h)}$$

is smaller than a given threshold t, then we don't compute the correspondent descriptor. Notice that t is the only parameter to be tuned in our parallel algorithm and it usually depends on the input image size andd the number of threads, but using $t = 0.23$ always removed the big descriptors, i.e. at most 1% of the original descriptors.

2.2.6 Parallel Hessian-Affine Algorithm

In the previous sections, we have seen how the ideal parallel solution for the Hessian-Affine algorithm introduced in Figure 4 can not be implemented or it would lead to bad performance.

Figure 5 shows the entire workflow of the parallel algorithm that we described in the previous sections.

2.3 Vectorization

Vectorization is an important aspect to achieve scalability in a parallel algorithm. As we have described in section 1.2, it is not part of this project to rewrite portions of the original code so it would be more suitable for parallel applications, but to reorganize the already-existing code as best that we can to achieve the best performances possible. Vectorization analysis has been conducted by using Intel Advisor. In the next sections we will see how the most expensive functions have been vectorized to reduce their cost and why we could not achieve better results in many cases. These functions will be presented from the most expensive to the less expensive.

2.3.1 cv::GaussianBlur

As we have previously seen, this is a crucial function in the Hessian-Affine algorithm. It is used to build the chain of blurs and even more in `normalizeAffine` as described in section 2.2.5. Unfortunately, this function is offered by OpenCV and its implementation is based on Integrated Performance Primitives (IPP), the Intel library which implements many image processing functions, such as gaussian filters. These functions are usually considered the state-of-the-art in terms of efficiency, so it would be probably useless to improve them. On a medium-size image, more than 56% of the overall time is spent in `ippFilterGaussianBorder`, the IPP function to compute gaussian blurs and it's already been vectorized.

2.3.2 interpolate

This is the most expensive function implemented in the original Hessian-Affine code. By disabling vectorization, it takes more than 36% of the overall time. Interpolation and warp-affine transofrmations are very common in image processing and of course such a function has been implemented in OpenCV. However, using the OpenCV equivalent introduced cumulative errors, increasing the number of detected keypoints and making the code slower in general.

In the original version, vectorization was not possible because of false dependency assumed by compiler. In addition, `*out++ = ...` prevents vectorization. Vectorization was enabled by simply incrementing the pointer value in the for loop declaration and by forcing vectorization using OpenMP.

The gain obtain by vectorizing this function is about of 39%, which is already a good result. Unfortunately, the number of loops iteration depends on the image to be interpolated, which size greatly change during the algorithm execution and since they are generally not a multiple of the vector length (i.e. the number of elements that can be processed in the same operation), we can not improve the vectorization performance of this function.

2.3.3 samplePatch

By disabling vectorization, this function used to compute descriptors takes almost 16% of the overall time. Straight vectorization is not possible, since for each iteration 8 different cells are modified and values in different iterations could refer to the same locations. By generating 8 different partial vectors (which will be reduced into the final vector later) this is a simple cumulative sum which can be auto vectorized by the Intel Compiler.

However, introducing 8 different vectors per iteration results in high register spilling, i.e. the register allocator runs out of registers with high frequency and so it must save and restore values from memory many times. However, by introducing four `distribute_point`, which splits the original loop is 4 smaller equivalent loops, the register usage is significantly reduced and vectorization is improved to a 35% gain.

More than 44% of memory instructions have an irregular access pattern and this results in poor cache locality, which make any further optimization impossible for this function. Unfortunately, nothing can be done here since it is part of the algorithm that the position of the written elements of the resulting vector greatly change from iteration to iteration.

3 User Manual

3.1 Compilers and Libraries

3.1.1 Intel Tools

The code of this project has been developed by using the C++ Intel Compiler 2017.3.191. Analysis tools used for this project are Intel VTune Amplifier 2017.2.0, Intel Advisor 2017.1.2 and Intel Inspector 2017.1.2.

3.1.2 OpenCV

This project uses OpenCV 3.2, cloned from the GitHub repository, commit number 9053839.

In the project directory, a bash script `cmake.sh` is provided to build OpenCV with the best options for this project, where most of the modules have been disabled since they are not necessary. Since the most efficient gaussian-blur implementation is offered by the Intel Integrated Performance Primitives (IPP) library, using this library it is necessary to obtain the best possible performance. OpenCV provides its own version of IPP, so in case you didn't install IPP on your system, you have to delete `-DIPPROOT="$IPPROOT"` from the bash script.

Unfortunately, OpenCV doesn't support AVX-512 instructions, but only AVX2. This means that OpenCV matrices are at most 256-bit aligned/optimized. This is resource underutilization for the KNL machine, since it is AVX-512 compatible. This means that we can not compile this project with AVX-512 flags, since the OpenCV allocated data are not 512-bits aligned. In addition, notice that all the CMake options to manage optimizations (such as enabling/disabling AVX) are available only when using `gcc` as compiler and not for `icpc`.

To build OpenCV for this project:

1. Open `cmake.sh`
2. Edit `ROOT_DIR` to the path of the directory containing the cloned OpenCV directory.
3. Edit `INSTALL` to the path where you want to install OpenCV.
4. Change `IPPROOT` to the root directory of the Intel Integrated Performance Primitives library. If you didn't install IPP on your system, you can use the built-in version provided by OpenCV. In that case, you have to delete `-DIPPROOT="$IPPROOT"` from the script.
5. Execute `./cmake.sh gcc|icpc` depending on which compiler you want to use.

3.2 Environment Variables and Thread Affinity

First of all, the system variable `LD_LIBRARY_PATH` should be edited by adding the absolute path of `PARALLELOPENCV` described in section 3.3.

Thread affinity is a popular tuning phase to improve performance in parallel applications. We used environment variable offered by Intel OpenMP compilers to manage thread binding. Since we don't consider hyperthreading in performance evaluation, supposing that we are using n physical cores, we should set `KMP_HW_SUBSET=`n`c,1t` which binds one thread per core. As suggested by Intel documentation, setting this variable along with `OMP_NUM_THREADS` is discouraged. In order to manage how threads are bound to resources allocated by `KMP_HW_SUBSET` , the `KMP_AFFINITY` variable is used: using `compact` always returned better results than the default value `scatter`.

Finally, the following variables always returned better results than the default values:

- `KMP_BLOCKTIME=0`
- `KMP_STACKSIZE=16m`
- `KMP_LIBRARY=turnaround`

3.3 Building Parallel Hessian-Affine

Before executing `make` to compile the project, you should open the `makefile` so:

1. `PARALLELOPENCV` is the path where OpenCV has been installed. In case we are using the OpenCV built in section 3.1.2, then use the same directory as `INSTALL` followed by `_icpc|gcc` depending on the compiler that you used.
2. The descriptor threshold described in section `sect:descriptorImbalance` is set at compile time with `-DRATIOTHRESHOLD`. From experimental results good values are between 0.23 and 0.3.

Notice TBB is necessary to use the shared queue as described in section 2.2.4.

3.4 Running Parallel Hessian-Affine

This is the syntax to get descriptors for an input image using the PHA:

```
./hesaff image [maxThreads][nRuns][scability]
```

Where:

- `image` is the input image. The image extension must be supported by OpenCV.
- `maxThreads` (default: `omp_get_max_threads()`) is the upper bound of the number of threads used in the program execution. Look at `scalability` for more details.
- `nRuns` (defualt: `1`) how many times each parallel configuration is executed. This is useful for evaluation purpose.
- `scalability` (default: `0`) if `1` perform a scalability test, otherwise get the descriptors in `image` using `maxThreads` threads for `nRuns` times. In a scalability test, `maxThreads` different parallel configurations are executed: the first one uses `omp_set_num_threads(1)`, the `i-th` configuration uses `omp_set_num_threads(i)` and the last one `omp_set_num_threads(maxThreads)`. Each configuration is executed `nRuns` times.

A typical execution output is:

```
[spm1428@ninja hesaffCC]$ ./hesaff darkKnight.jpg
------------------------------------------------
File: darkKnight.jpg
Using 64 threads
descriptor time 2.13762 seconds
parallel time=1.00586
descriptors.rows=41131
avg descriptor time=2.13762
avg parallel time=1.00586
```

Where 41131 SIFT descriptors from `darkKnight.jpg` have been computed in 2.13 desconds. The `paralle time` is the time spent in the OpenMP region only, while `descriptor time` includes the chain of gaussian blurs too. `avg descriptor/parallel time` are for the same parallel configuration (and they make sense only when `nRuns` is set).

If you want to see more detailed times during execution, then uncomment `#define VERBOSE` in `pyramid.cpp`.

If you want to test the scalability on `thor.png` from 1 to 32 cores and each parallel configurations executed 5 times, then use:

```
[spm1428@ninja hesaffCC]$ ./hesaff darkKnight.jpg 32 5 1
```

4 Performance Evaluation

In this section we will examine different benchmarks on the PHA algorithm, giving motivations whenever the application doesn't perform well as expected. Unless specified otherwise, all the results in this section are obtained by the average of five different runs per parallel configuration.

This project has been deployed and tested on an Intel®Xeon Phi™ 7210, aka Knights Landing (KNL), with 64 hardware-threaded cores.

It is well-known that performance gain using logical cores via hyper-threading is much worse than using physical cores [Hypa] [Hypb]. Since PHA will not scale as expected even by using physical cores only, we limited the number of threads on the KNL machine during our analysis to the number of physical cores (i.e. 64 cores).

Figure 6: `all_souls_000002.jpg`

Figure 7: `darkKnight.jpg`

4.1 Image Dataset

The idea of this project came up for an augmented reality project, where movie posters, paintings and building can be recognized.

Four movie posters of different sizes and one building picture have been used to test PHA (sorted in ascending order according to the image size):

- `darkKnight.jpg` 4050x6000pxs showed in Figure 7.
- `doctorStrange.png` 4000x5663pxs.
- `darkKnight50.jpg` 2025x3000pxs.
- `starWars.jpg` 1369x2125pxs.
- `piratesOfTheCarribean.jpg` 1382x2048pxs.
- `darkKnight25.jpg` 1013x1500pxs.
- `all_souls_000002.jpg` 677x1024pxs, from the Oxford Building Dataset [PCI+07], showed in 6.
- `darkKnight10.jpg` 405x600pxs.

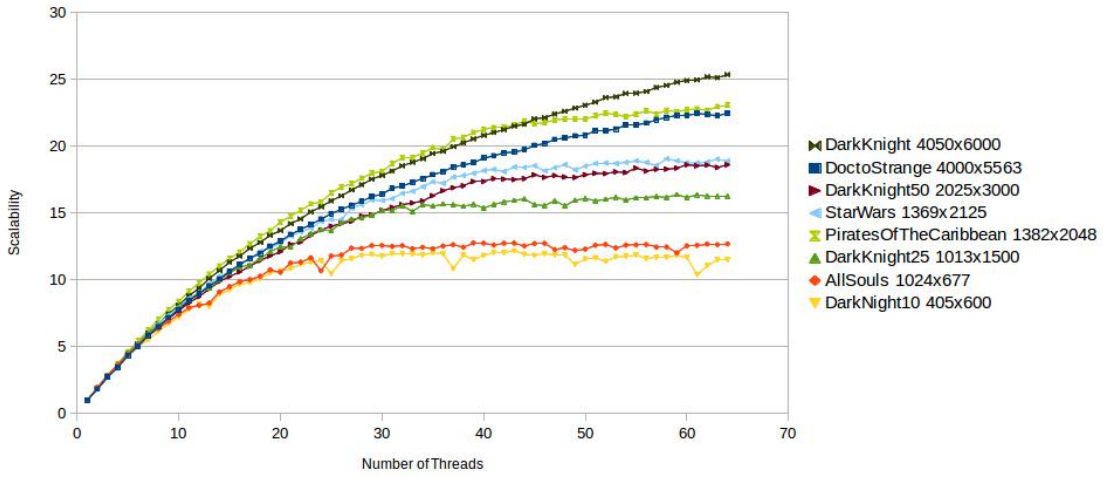

Figure 8: Scalability of the whole PHA algorithm.

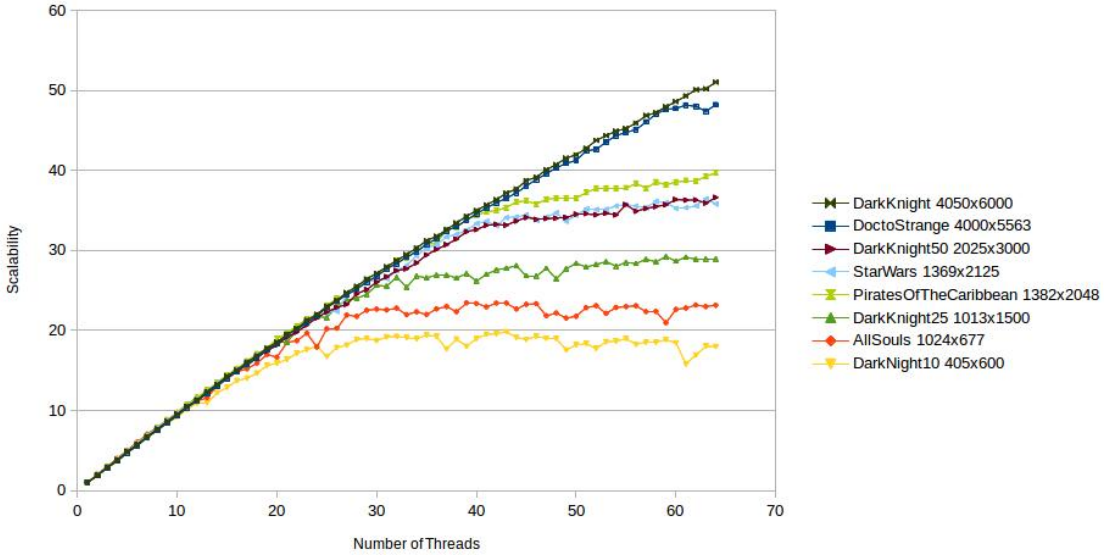

Figure 9: Scalability of the OpenMP section.

Notice that `darkKnight10/25/50.jpg` are the resized versions of `darkKnight.jpg`. Testing resized images is very important since it is a common pre-processing operation in Computer Vision applications. However, in many applications it is important to use full-sized images, especially when real-time detection is not a requirement (e.g. during training).

4.2 Scalability and Efficiency

As first experiment, scalability of the PHA algorithm has been tested on the KNL machine, as shown in Figure 8. As we can see, the application does not scale well. However, as we have seen in section 2.2.2, there is a consistent serial part in our algorithm. Then, we can see that for image of similar size, we obtain similar performance, which means that the the algorithm performance doesn't depend on the image content but mainly on the image size. Finally, we can notice that if we increase the workload for each thread (i.e. the image size) the algorithm scales better.

The portion of serial time (i.e. building the chain of gaussian blurs) depends on the image size. To prove this, we used the four versions of `darkKnight.jpg` described in the previous section: in Figure 12 we can see that the serial part of the algorithm increases according to the image

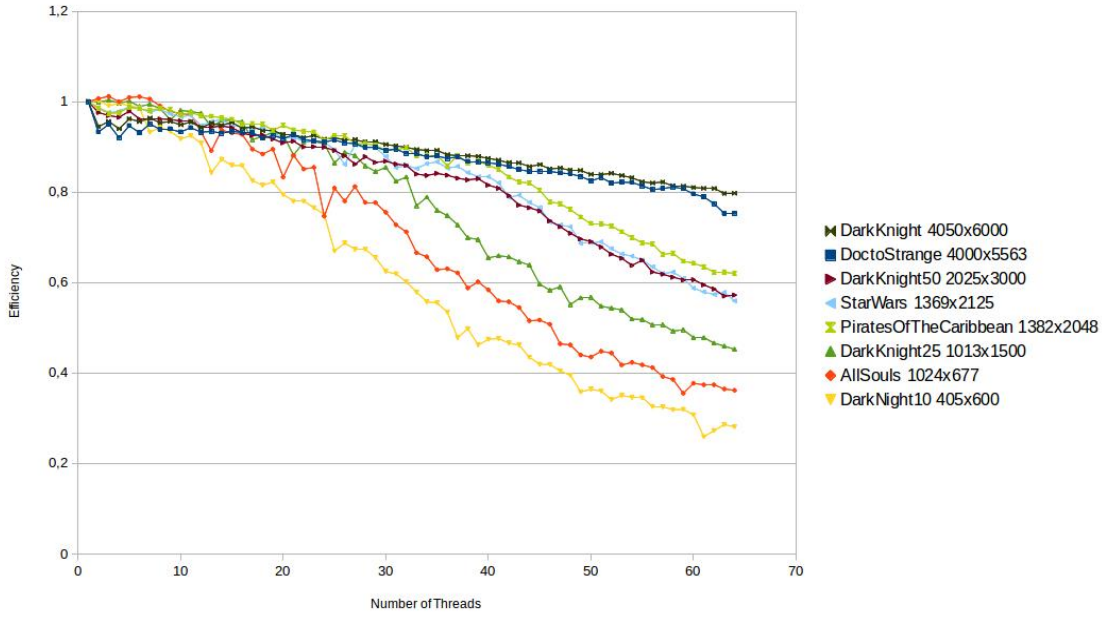

Figure 10: Efficiency of the OpenMP section.

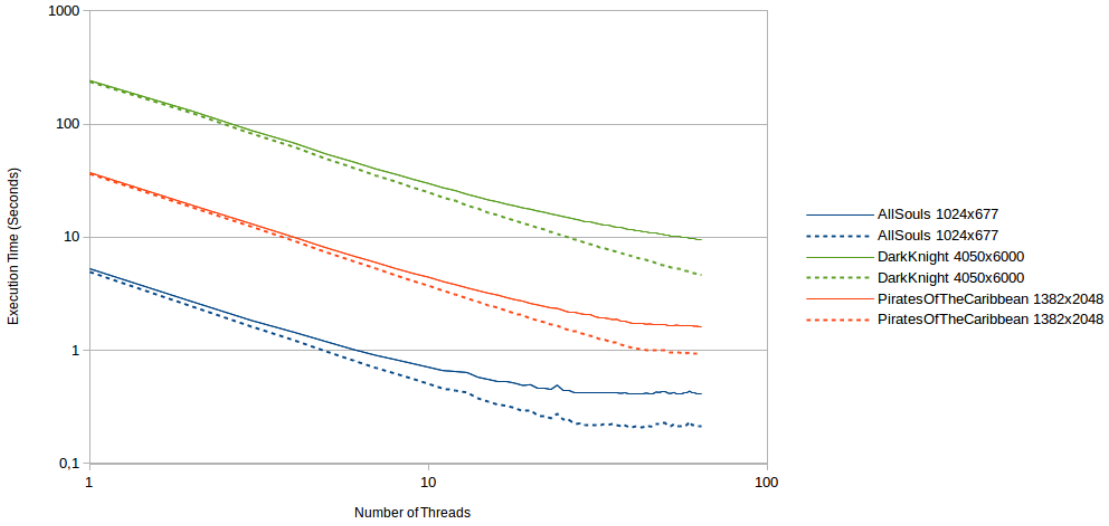

Figure 11: Execution times for different size images.

size. If we consider only the parallel region, we can see that the scalability improves, as shown in Figure 9, reaching almost the expected scalability for large images. In Figure 10 we can see how efficiency in the parallel region quickly decreases for smaller images and how it is more unstable than larger inputs. The good scalability for large images is proven by the CPU usage histogram in the OpenMP computed with VTune Amplifier in Figure 13.

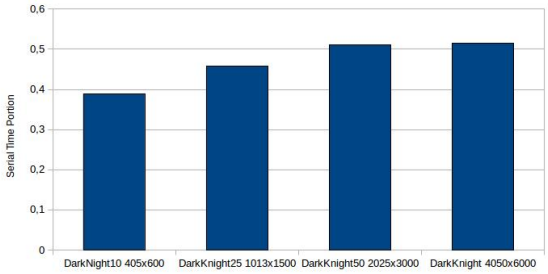

Figure 12: Serial portion according to image size.

Finally, in Figure 11 we can see execution times depending on the number of threads in a log-log scale: continuous lines represent the execution time considering the serial and parallel sections, while dashed lines are the execution time for the OpenMP region only. For sake of clarity, only three images of small, medium and large size are shown. Notice that the gap between the correspondent continuous and dashed lines is almost the same for all images and, considering that we are in a log-log scale, this proves how large images scales better than small images.

4.3 Synchronization overhead

As explained in section in 2.2.4, a synchronized queue is used to collect detected keypoints, before descriptors are generated from each one of them. This mechanism is introduced to ensure load balancing during keypoint detection, but the `concurrent_bounded_queue` could introduce a consistent overhead time. Surprisingly, according to Intel VTune amplifier, for `darkKnight.jpg` the most expensive waiting object is the barrier at the end of the parallel section where `hessianResponse` is called.

However, the total waiting time for this barrier is 4.48 seconds with 63 counts (all the threads expect the last one), which means less than 7 milliseconds per thread on average. Considering that the whole parallel region takes more than 4.5 seconds, this waiting time is negligible. Instead, the total synchronized queue's overhead is 0.118 seconds only, with 40 counts, so for big images the queue mechanism doesn't introduce a relevant overhead. Instead, if we look at smaller images, this overhead is larger: for `all_souls_000002.jpg` the waiting time introduced by the queue is 1.171 seconds with 36 counts, which means 0.032 seconds per count on average; considering that the parallel OpenMP region takes around 0.213 seconds, we can see that this overhead is more relevant. This is a first possible explanation about why the PHA algorithm doesn't scale well for small images.

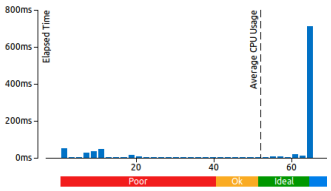

Figure 13: CPU usage in the OpenMP region for `darkKnight.jpg`.

4.4 Memory Bandwidth Usage

Function / Call Stack	CPU Time ▼	Memory Bound		
		L2 Hit Rate	L2 Hit Bound	L2 Miss Bound
n0_ownFilterColumnPipeline_32f_C1R_X_G9E9	17.154s	78.6%	100.0%	100.0%
n0_ownFilterRowBorderPipeline_32f_C1R_X	8.853s	100.0%	0.1%	0.0%
AffineShape::normalizeAffine	5.486s	72.9%	5.0%	25.2%
n0_ownFilterRowBorderPipeline_32f_C1R_X_AVX	5.090s	75.0%	0.7%	3.2%
SIFTDescriptor::samplePatch	2.983s	100.0%	1.3%	0.0%
AffineShape::findAffineShape	2.641s	38.9%	1.1%	22.5%

Figure 14: Memory bandwidth analysis of `darkKnight.jpg`.

Another possible reason for bad scaling is an high memory bandwidth utilization or an high rate of cache misses. Memory Access analysis in VTune Amplifier was used to inspect this factor. For both large and small images, the memory bandwidth is very low. However, for large images, the gaussian blur implementation is very cache inefficient: as it can be seen in Figure 14, there is a huge number of cache misses during the filter computation. This could be a reason for non-scalability in medium and large images, while we didn't find this problem in the OpenCV function for smaller images.

4.5 Why it doesn't scale?

In the previous sections we have seen different reasons why the parallel algorithm may not scale as expected, especially for small images. In particular:

- Queue synchronization: as we have seen in section 4.3, since the workload per thread is low for small images, there is an higher probability of contention for the shared queue, i.e. synchronization overhead.
- Cache misses: in the last section we have seen how the OpenCV implementation of the Gaussian Blur filter is not cache efficient, make the code slower for large images and less scalable.
- Vectorization: in section 2.3 we have seen how vectorization improved the application performance. However, we also have seen how vectorization can not be fully exploited for the given algorithm implementation.

As we described in the project goals described in section 1.2, the target input of this object are medium and large images, where the latter ones scale almost as expected.

References

[BTVG06] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. *Computer vision–ECCV 2006*, pages 404–417, 2006.

[CLSF10] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. *Computer Vision–ECCV 2010*, pages 778–792, 2010.

[Hypa] How to determine the effectiveness of hyper-threading technology with an application. https://software.intel.com/en-us/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application.

[Hypb] Hyper-threading may be killing your parallel performance. https://www.pugetsystems.com/labs/hpc/Hyper-Threading-may-be-Killing-your-Parallel-Performance-578/.

[JDSP10] Hervé Jégou, Matthijs Douze, Cordelia Schmid, and Patrick Pérez. Aggregating local descriptors into a compact image representation. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3304–3311. IEEE, 2010.

[JPD+12] Herve Jegou, Florent Perronnin, Matthijs Douze, Jorge Sánchez, Patrick Perez, and Cordelia Schmid. Aggregating local image descriptors into compact codes. *IEEE transactions on pattern analysis and machine intelligence*, 34(9):1704–1716, 2012.

[Low99] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[MS02] Krystian Mikolajczyk and Cordelia Schmid. An affine invariant interest point detector. *Computer Vision—ECCV 2002*, pages 128–142, 2002.

[MTS+05] Krystian Mikolajczyk, Tinne Tuytelaars, Cordelia Schmid, Andrew Zisserman, Jiri Matas, Frederik Schaffalitzky, Timor Kadir, and Luc Van Gool. A comparison of affine region detectors. *International journal of computer vision*, 65(1-2):43–72, 2005.

[PCI+07] James Philbin, Ondrej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007.

[PCM09] Michal Perd'och, Ondrej Chum, and Jiri Matas. Efficient representation of local geometry for large scale object retrieval. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 9–16. IEEE, 2009.

[RRKB11] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2564–2571. IEEE, 2011.

[ZCZX08] Qi Zhang, Yurong Chen, Yimin Zhang, and Yinlong Xu. Sift implementation and optimization for multi-core systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.