Image Skeletonisation

A PERFORMANCE COMPARISON WITH THRUST

Marco Antognini

Fall 2014

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Introduction

This project aims to compare a theoretical analysis of the parallelisation of the image skeletonisation problem against an actual implementation in C++ using Thrust, a high level algorithm library with CUDA, TBB and OpenMP backends. The goal is to compare the speedup of the parallel implementation and highlight the effect of some hardware properties on the scalability of the performances.

Problem Description

The *Image Skeletonisation* algorithm consists in reducing a binary image to the shape made of the equidistant points to the boundary of the original object. To achieve such result, a category of algorithms erodes the pixels of the image until no more pixels are deleted according to local criteria such as 4/8-connectivity, end points or isolated pixels and so on.

The skeletonisation algorithm is especially useful to allow analyse of complex images by first reducing the information to its minimal representation. It is used, for example, for blood composition analysis, in the classification of fingerprints, or for the analysis of texts or geometrical shapes.

original image *skeletonised image*

Thinning Methodologies – A Comprehensive Survey[1] classifies the most common thinning schemes. For this project we will focus on a parallel algorithm introduced by Z. Guo and R. W. Hall[2], with 2 sub-iterations based on Hilditch's crossing number X_H. This algorithm is defined as follow:

> A pixel p – where its 8-neighbours are denoted x_1, x_2, …, x_8 and are ordered in a anticlockwise fashion starting from the east – is marked as deleted if and only if the following properties hold in the first sub-iteration:
>
> G0: p is black;
>
> G1: $X_H(p) = 1$
>
> G2: $\min(n_1(p), n_2(p)) \in \{ 2, 3 \}$

[1] Thinning Methodologies – A Comprehensive Survey, IEEE Transactions on pattern analysis and machine intelligence, vol. 14, no. 9, 1992, by L. Lam, S.-W. Lee and C. Y. Suen.

[2] Parallel thinning with two sub-iteration algorithms, Comm. ACM, vol. 32, no. 3, 1989.

G3: $(x_2 \vee x_3 \vee \neg x_8) \wedge x_1 = 0$

In the second sub-iteration the condition G3 is replaced by its 180° rotation, that is $(x_6 \vee x_7 \vee \neg x_4) \wedge x_5 = 0$. X_H, n_1 and n_2 formulae are defined as follow:

$$X_H(p) = \sum_{i=1}^{4} b_i, \quad b_i = \begin{cases} 1 & \text{if } x_{2i-1} = 0 \text{ and } (x_{2i} = 1 \text{ or } x_{2i+1} = 1) \\ 0 & \text{otherwise} \end{cases}$$

$$n_1(p) = \sum_{i=1}^{4} x_{2i-1} \vee x_{2i}$$

$$n_2(p) = \sum_{i=1}^{4} x_{2i} \vee x_{2i+1}$$

After each iteration the marked pixels are deleted from the source image.

This algorithm is interesting for several reasons. The main relevant aspect for this project is the data dependency. Every iteration (and sub-iteration) depends on the previous one. However, during one sub-cycle, there is no dependency between pixels so that every pixel can be marked for deletion in parallel.

Implementation Forewords

For the sequential implementation we use C++11 with a few external libraries in addition to STL such as boost[3] to parse the argument of the program and SFML[4] to load and save images and measure elapsed time in a 100% portable way.

The parallel implementation relies essentially on the same tools for the data acquisition, time measurement and argument parsing. For the parallelisation itself we use Thrust[5], a C++03 general purpose, high level, GPU and CPU accelerated library whose API is very similar to STL. Thrust targets maximal occupancy of the device system and will compare the resource usage of the kernel (e.g. the number of registers or the amount of shared memory) with the resources of the target GPU or CPU to determine a launch configuration with the highest occupancy. As a consequence we don't have to go through this cumbersome task ourself.

Resources

The source code, its resources and the raw measurement data are available online.

https://github.com/mantognini/Image-Skeletonisation

[3] http://www.boost.org/

[4] http://sfml-dev.org/

[5] https://thrust.github.io/

Hardware & Software Specifications

The computer used for the benchmarks is a MacBookPro 10,1 with the following hardware specifications and software versions:

Host CPU	Intel Core i7 3720QM @ 2600 MHz
Host Memory	16 GB DDR3 @ 1600 MHz
Device GPU	NVIDIA GeForce GT 650M @ 900 MHz
Device Memory	1GB GDDR5 @ 2508 MHz
Thrust	Version 1.7 with NVCC 6.5.12
CUDA	Driver 6.5, Runtime 6.5, Capability 3.0 (Kepler GK110)
TBB	Version 4.3 with Clang 3.5
OMP	Version 4.0 with GCC 4.9

It is important to distinguish which backend is used with Thrust since it implies significant behaviour shifts at the runtime. Beside having a higher parallelism potential with its 2 multiprocessors, which account for a total of 384 CUDA cores, the *GeForce GT 650M*[6] of the Kepler family[7] has a smaller, but faster, memory of 1 GB and more importantly is separated from the host system. A Kepler multiprocessor supports a total of 64 simultaneously living warps, but only 4 of them can be active concurrently. Hence, a total of 2048 threads residing on each chip and potentially up to $4 \cdot 32 = 128$ active threads par multiprocessor. It follows that we have $K = 256$ worker threads simultaneously executing 32-bit instructions.

Alternatively, the TBB and OMP backends don't suffer from a potentially important transfer time when sending data from the host system to the GPU but they have a much lower scalability potential with only $K = 8$ concurrently executing threads for this specific hardware. Moreover, the CPU and GPU clock speeds are also very different and it has to be taken into account.

Using *bandwidthTest*, from the CUDA sample box, shows that CPU/GPU communication speed depends on the amount of data transferred. We use a very similar tool to analyse host memory speed and we measure the transfer time for the various data sizes that appear in the theoretical analysis.

Computation Model

Thrust handles everything for us by providing some computation constructs, such as *transform*, *reduce*, *copy*, *count*, *search*, or *partition* just to mention a few algorithms, and thus enables us to run a kernel function in parallel on all K worker threads. We therefore

[6] http://www.geforce.com/hardware/notebook-gpus/geforce-gt-650m/specifications

[7] http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

assume that is uses some pipelining technics to optimise the computation on the device and consider that all K threads actually run in parallel when there is no threads divergence. This assumption is supported by the fact that Kepler multiprocessor can activate another warp when one is waiting for data to be fetched from memory.

Compilers Optimisations

It can be shown that in some context – such as thinning *tree.png*, which is presented afterwards – we can get a 11x speedup on the serial implementation itself simply by relying on Clang capability to optimise the source code with the *-Ofast* option. We therefore heavily rely on such options with Clang, GCC and NVCC to make the serial and parallel softwares faster.

Complexity Analysis

We are interested here in the sequential complexity of the algorithm described previously and how it can be parallelised efficiently.

Sequential Algorithm

We focus here on the algorithm itself and not on how the data is acquired. During one iteration of the thinning process every pixel is considered for deleting twice: once in the first sub-cycle and once in the second sub-cycle. Roughly, 50 basic operations – such as *or*, *and*, *not*, *add*, *sub* and *compare* – are performed for each pixel in each sub-cycle of each iteration. Since the algorithm stops when the image has converged, in the worst case every pixel was deleted one at a time. To sum up, the time complexity of algorithm is $O(2 \cdot 50 \cdot N^2) = O(N^2)$ where N is the number of pixels. We can also express the complexity in terms of the required number of iterations, M. This complexity is $\theta(M \cdot N)$.

We will show later that the sequential implementation essentially has a space complexity of $\theta(N)$.

Parallel Algorithm

For every iteration of the algorithm, until the image has converged, each sub-cycle can be run in parallel. However, it is crucial to introduce some synchronisation mechanism between the two cycles to delete the pixels at an appropriate time because each sub-cycle relies on the previous one. For the parallel implementation, this means the introduction of a double buffering strategy. We will show later that the memory space for this version of the algorithm is $\theta(N)$ on the host system and $\theta(2 \cdot N)$ on the device system.

Even though the total number of operations is of the same order of magnitude, the program has now a time complexity of $O(N^2 / K)$ where K is the number of threads crunching the data simultaneously. Or, in terms of number of iterations, $\theta(M \cdot N / K)$.

Convergence

We let the reader convince himself that the convergence factor M of an image is bound by the thickness of the widest arm of the shape and not its size or perimeter. A step by step reduction of *tree.png* can be found within the resources of this project.

IMPLEMENTATION

For the serial version we rely on *std::vector<bool>* to store the image data in a very space-efficient manner in order to optimise cache access. Alternatively, we could have also tried *boost::dynamic_bitset* were the performances not satisfactory. However, the transition to Thrust would have been harder since it relies on containers similar to *std::vector*. The position of pixels marked for deletion in one of the sub-cycles are stored temporarily in a queue until the current sub-cycle is over at which point the pixel are actually deleted.

However, due to the fact that Thrust heavily rely on algorithms – such as *transform* or *reduce* – that run on the device system, we cannot easily use a queue to store the pixels that need to be deleted because we are not allowed to allocate memory from a kernel function. Therefore we use a double buffering mechanism approach: we copy the whole image into two buffers on the device and use one as the current state of the problem and the other one as its output. When a sub-cycle is done we copy (from device to device) the output into the buffer used as input for the next sub-iteration.

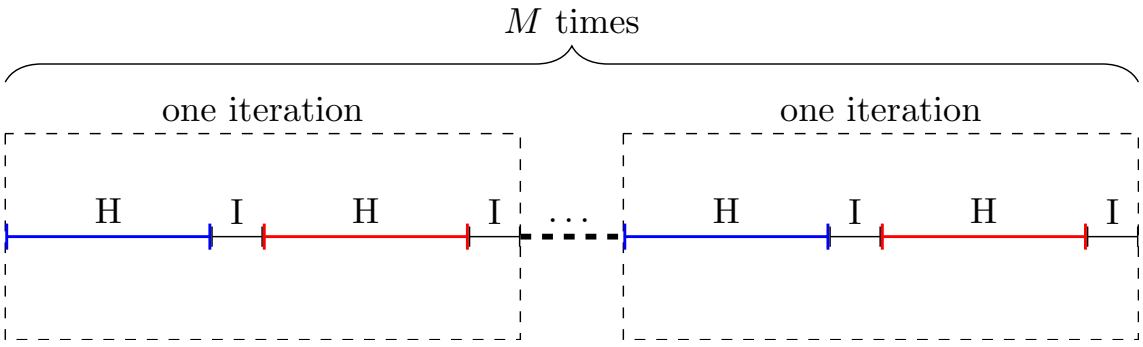
In order to avoid transferring the whole image after each iteration from the device to the host, we use *transform_reduce* construct to delete the pixels in the second buffer and at the same time count the number of deleted pixels. When both sub-cycle kernel functions returns 0 we know that the image has stabilised.

The reader is invited to get the source code and compare the serial implementation (*Skeleton.cpp*) to its parallelised counterpart (*Skeleton.cu*) for more details.

Theoretical Prediction

Timing Diagram

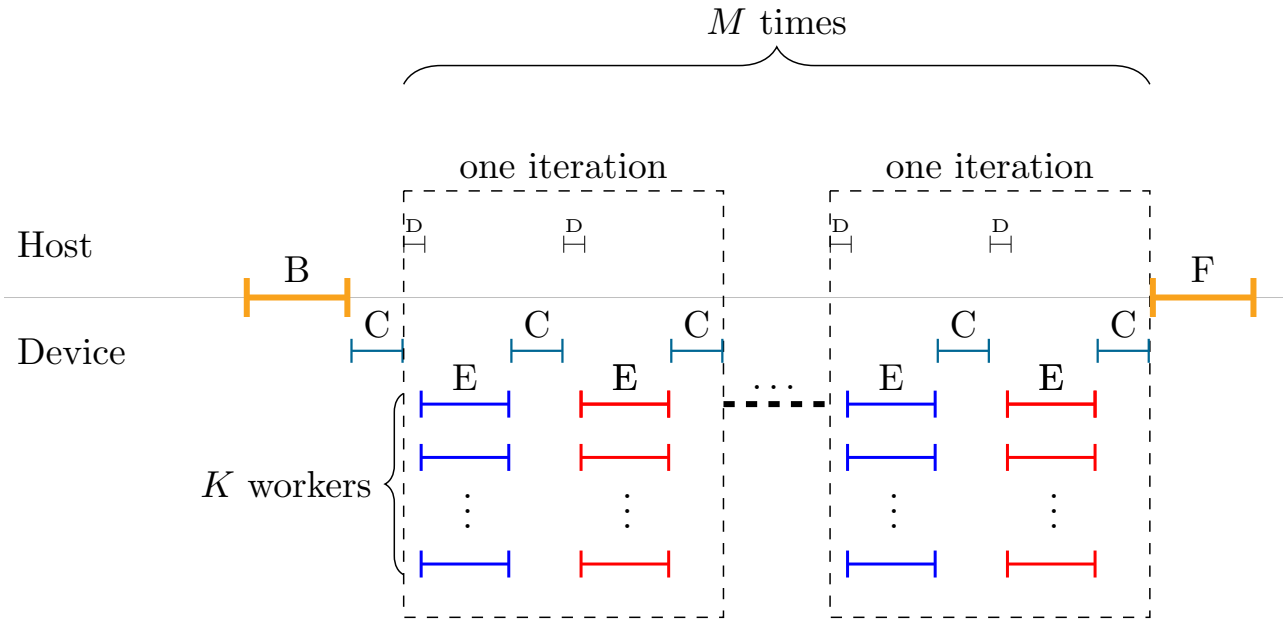
The schema below depicts how the sequential algorithm works:



The sequence [H I H I] is repeated until there is no more pixel to delete. The events are as follow:

- H: a sub-iteration of the algorithm, as previously defined, is executed:
  - the blue ones denote the first sub-cycles and
  - the red ones the second sub-cycles;
- I: the marked pixels are deleted from the image.

The next diagram illustrates how the parallel implementation behaves:



The host system dictates what should be computed on the device system with its K workers. To do so, the host sends data to the device memory first and then launches kernel functions to transform the data. Here we have the following events:

- B: the image is copied a first time to a first buffer, denoted ❶, in the device memory;

- C: this event acts as a synchronisation of the data on the device: the buffer ❶ is copied to a second buffer, denoted ❷, also living in the device memory;
- D: the host launch a kernel function to perform a sub-cycle of the general algorithm;
- E: the kernel function is executed on K worker threads on the device and similarly to H the blue ones represent the first sub-cycles while the red ones are the second sub-iterations. Pixels are directly removed in ❶;
- F: once the algorithm has converged the data in ❶ is sent back to the host system.

We make the hypothesis that D is very small – only a few μs – and can therefore be neglected in further analysis.

Performance Predictions

The serial and parallel times can be represented by the following formulae:

$$t_{ser} = 2M\left(t_{sub,ser} + t_{update}\right)$$

$$t_{par} = t_{cp,H\to D} + t_{cp,D\to D} + 2M\left(t_{sub,par} + t_{cp,D\to D}\right) + t_{cp,D\to H}$$

We then rewrite those equations as functions of the number of pixels in the image, N; the number of iterations before the image converges, M; the number of worker threads, K; the transfer speed between host and device memory, $H{\to}D_s$ and $D{\to}H_s$, as well as from device memory to device memory, $D{\to}D_s$; and, finally, the host and device clock speed ratio, λ. Later we will refer to $t_{sub,ser}$ as t_{sub} and express $t_{sub,par}$ in terms of t_{sub}, K and λ.

In the next formula, the speedup is defined in terms of the image size, the number of iterations and the number of workers:

$$Speedup(N, M, K) = \frac{t_{ser}}{t_{par}} = \frac{t_{ser}}{\frac{N}{H\to D_s} + \frac{N}{D\to D_s} + 2M\left(\frac{\lambda}{K}t_{sub} + \frac{N}{D\to D_s}\right) + \frac{N}{D\to H_s}}$$

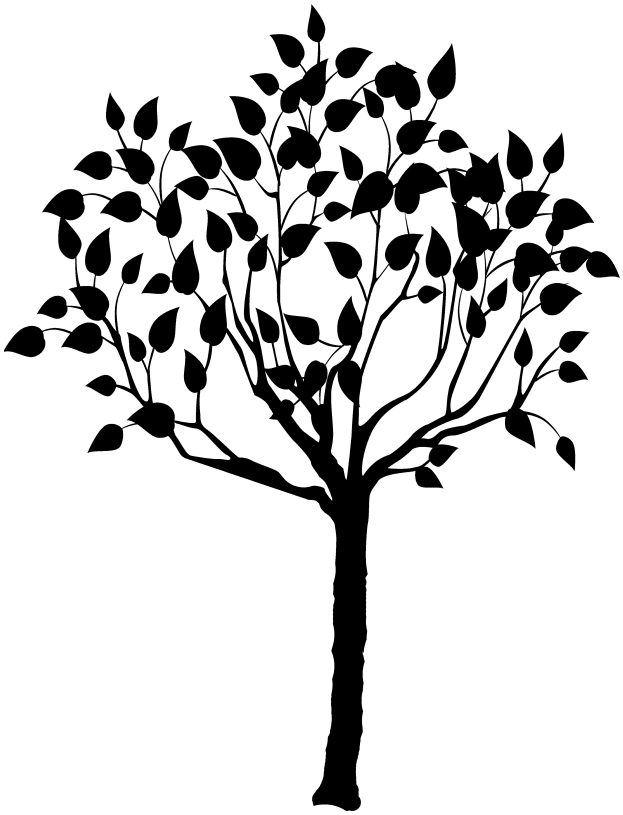
As explained in the introduction, it is crucial to distinguish which backend is used. We focus first on the numerical values for the CUDA backend of our system. We use the tool presented in the introduction to measure the bandwidths of the system. As a concrete case, we analyse the performance of a specific image: *tree.png*.

$$H \to D_s \leq \begin{cases} 1.6 \text{ GB/s} & \text{for 2 MB} \\ 1.6 \text{ GB/s} & \text{for 15 MB} \end{cases}$$

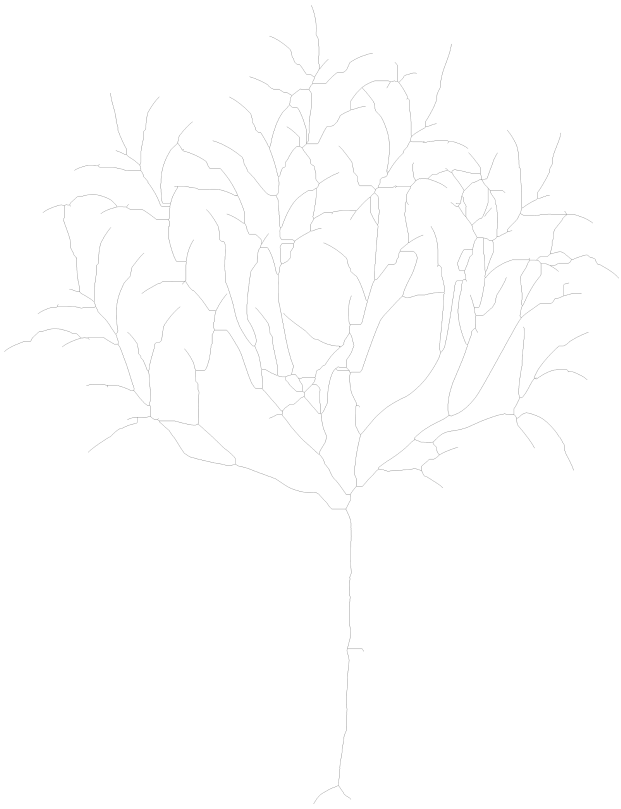
$$D \to H_s \leq \begin{cases} 6.2 \text{ GB/s} & \text{for 2 MB} \\ 6.2 \text{ GB/s} & \text{for 15 MB} \end{cases}$$

$$D \to D_s \leq \begin{cases} 31.5 \text{ GB/s} & \text{for 2 MB} \\ 37.9 \text{ GB/s} & \text{for 15 MB} \end{cases}$$

$$\lambda = \frac{2600 \text{ MHz}}{900 \text{ MHz}} = 2.88$$



original image



skeletonised image

To skeletonise this image, the algorithm needs $M = 126$ passes. The number of processed pixels is $N = 16{,}069{,}074$, which are just bits since it is a boolean image, hence a size of roughly 1,961 KB. Measuring the runtime of the serial implementation gives us $t_{ser} = 2M(t_{sub} + t_{update}) = 21{,}595$ ms. It was also observed that t_{update} was on average 50 μs and therefore can be discarded for the sake of simplicity. We will refer to $t_{sub} = 85.69$ ms as the average time of an iteration.

With those values and the one for the CUDA backend we get a theoretical speedup of approximately 83x, thanks to its $K = 256$ worker threads.

$$
\begin{aligned}
Speedup_{\text{CUDA,tree.png}} &= \frac{21595 \text{ ms}}{\frac{1961 \text{ KB}}{1.6 \text{ GB/s}} + \frac{1961 \text{ KB}}{31.5 \text{ GB/s}} + 2 \times 126 \times \left(\frac{2.88}{256} \times 85.69 \text{ ms} + \frac{1961 \text{ KB}}{31.5 \text{ GB/s}}\right) + \frac{1961 \text{ KB}}{6.2 \text{ GB/s}}} \\
&= \frac{21595 \text{ ms}}{1.20 \text{ ms} + 0.06 \text{ ms} + 252 \times (0.97 \text{ ms} + 0.06 \text{ ms}) + 0.30 \text{ ms}} \\
&= \frac{21595 \text{ ms}}{260 \text{ ms}} \approx 83
\end{aligned}
$$

Even though we used the average transfer speed for a 2 MB transfer here we have to be extremely careful with this result. We make a second prediction where a pixel is stored on 8 bits instead of only one as it is the case with the implementation of *std::vector<bool>*: it is highly likely that the storage system used by Thrust – *thrust::device_vector<bool>* and *thrust::host_vector<bool>* – is not highly dense for this specific type of data. With that in

mind, we have a total of roughly 16 MB to transfer between hosts and devices and the theoretical speedup is now of approximatively 61x. It is therefore obvious that the compactness of the data representation can have a significant impact on the performance.

We also make two predictions for the CPUs backends. TBB and OpenMP fundamentally work the same way because the host and device system are the same identity, which provide $K = 8$ worker threads. This last property implies that we have only one memory transfer speed to consider and therefore we can use the following numerical values:

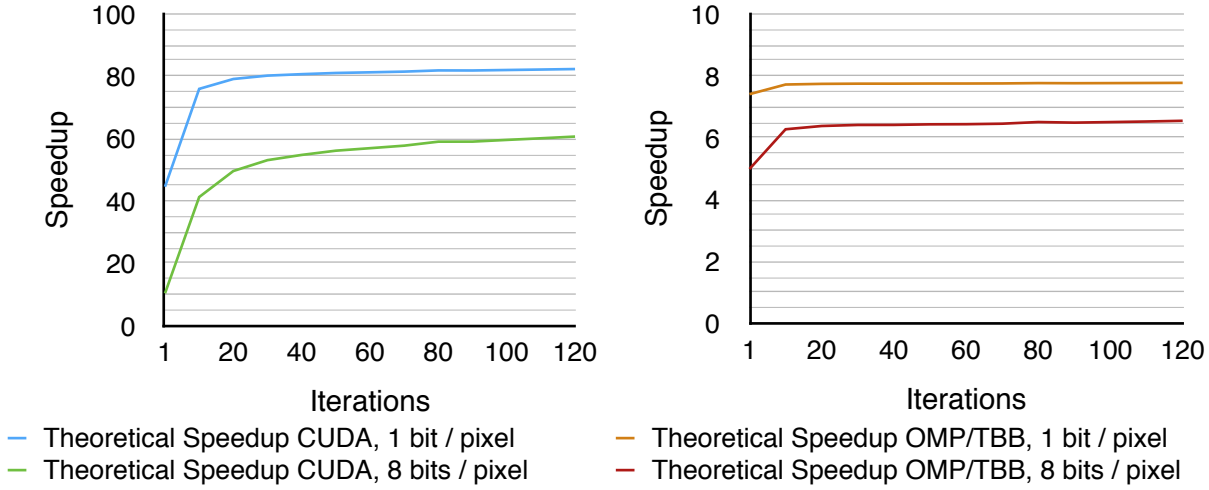
$$H \to H_s \leq \begin{cases} 6.6 \text{ GB/s} & \text{for 2 MB} \\ 6.4 \text{ GB/s} & \text{for 15 MB} \end{cases}$$

$$\lambda = \frac{2600 \text{ MHz}}{2600 \text{ MHz}} = 1$$

It follows we have a theoretical speedup of 7.8x for the 1 bit per pixel storage version and a speedup of 6.6x with the 8 bits per pixel storage version. Compared to the CUDA speedups, those are much closer to one another but the storage density still has a significant impact on the results.

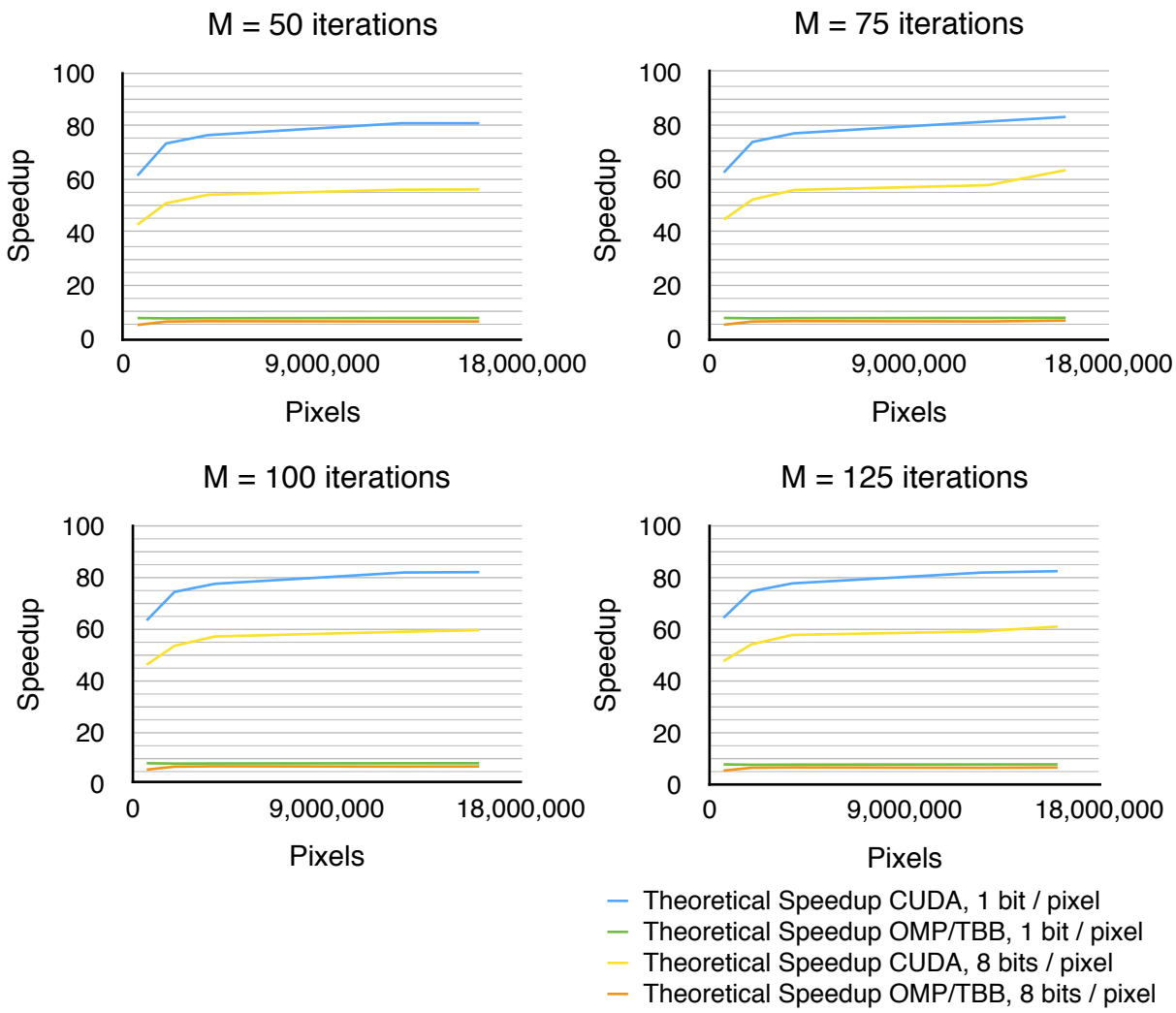
Generalisation: Iterations and Image Size

The following graphs generalise the speedup estimations for a wide range of required iterations for a fixed image – *tree.png* – but at various stage of the thinning process.



The data shows that the CPU backends have a relatively constant speedup unlike the GPU backend which sees its speedup follow a logarithmic slop. Analysing more images leads to the same observations.

Next, we analyse how the theoretical speedups behave in terms of the image size. Doing so for a few different values of the required iteration metric reveals that for all studied iteration count, on the one hand, the CPU backends have relatively stable speedups, for both 1 bit per pixel and 8 bits per pixel, and, on the other hand, the CUDA backend sees its speedup stabilise when the image size is relatively big.

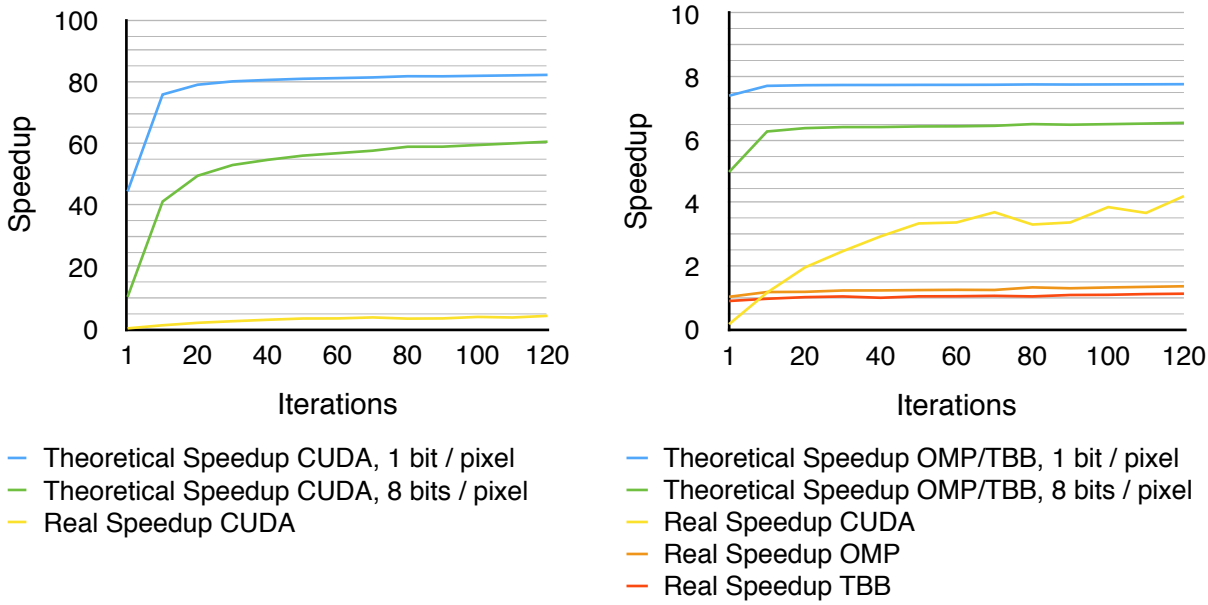


Before we continue with the discussion on whether these results are close to the reality or not, we can already find an important flaw in the model used above: at no point is memory caches and concurrent access to memory banks taken into account. This is typically where the prediction can fail to match the behaviour on the hardware. Another aspect that is not covered here is the active/inactive threads ratio on GPU, even though this algorithm should not suffer from this problem because it doesn't need any branching in the kernel function[8].

[8] The minimum function can be written without any if-statement as shown by S. E. Anderson https://graphics.stanford.edu/~seander/bithacks.html

Measurements & Appreciations

Analysing *tree.png* performance with regard to the number of iterations reveals that the real speedups are much smaller than the one expected: the CUDA implementation gives a speedup of less than 5 but the logarithm-like curve is still present and the CPU backends are roughly equivalent to the serial implementation with relatively constant speedups between 1 and 1.5.

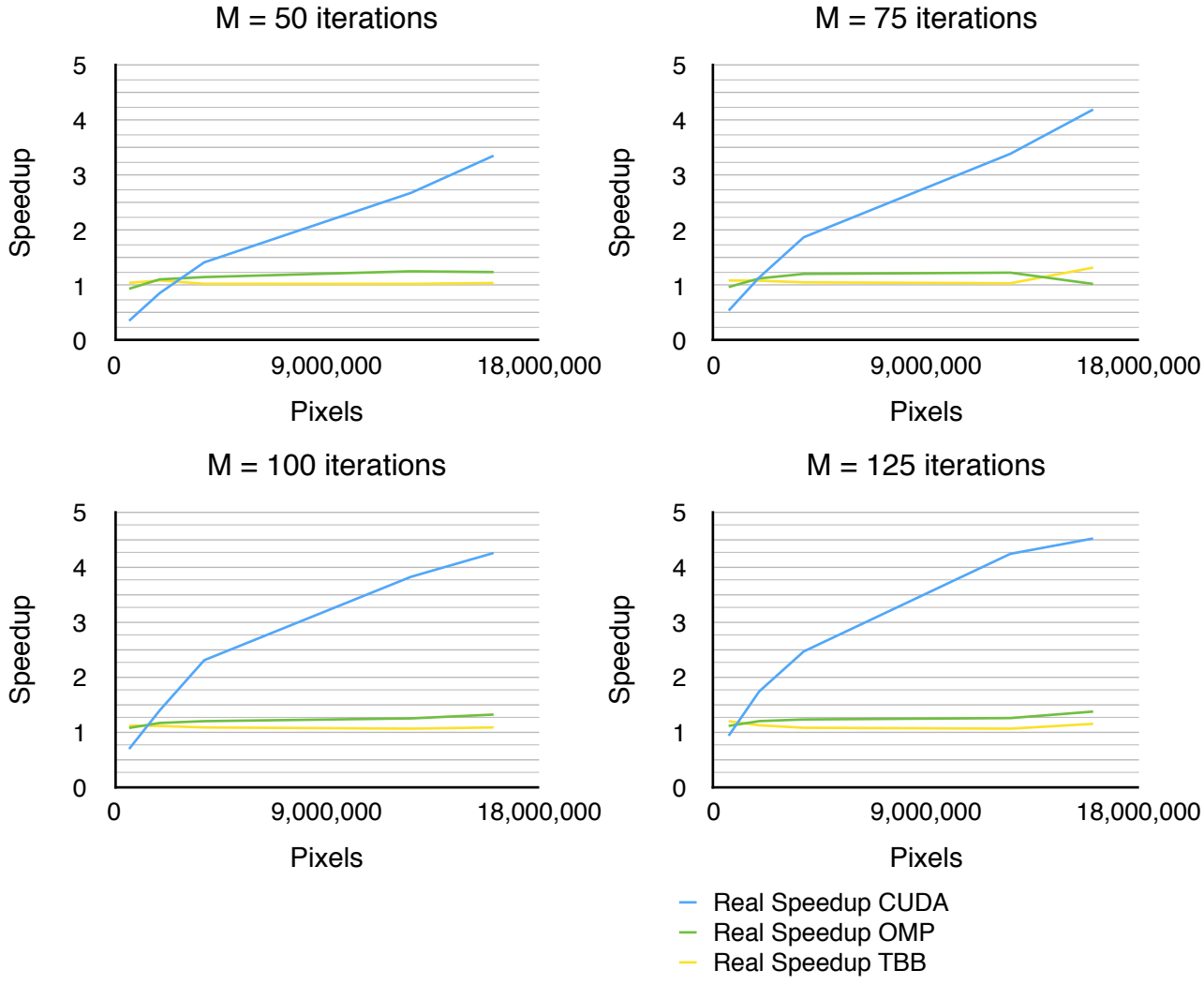


For *fractalA.png*, which is a 10,000 by 10,000 image that requires 2028 iterations to converge, the speedup is 2x with OpenMP, 1.56x with TBB and 7.91x with the CUDA backend while the theoretical speedup are 6.91x, 6.91x and 53x, respectively. The next table presents different theoretical and measured speedups for a variety of pictures and the 8 bits per pixel version.

Input			Measured Speedup			Theoretical Speedup	
File	Pixels	Iterations	CUDA	OMP	TBB	CUDA	OMP/TBB
bw_02.png	151,600	11	0.03	0.88	0.91	24.94	4.73
hiboux.png	626,400	151	1.14	1.15	1.32	50.66	5.61
EPFL2.png	1,920,000	254	2.90	1.28	1.19	58.60	6.67
fingerprint.png	2,250,000	22	0.64	1.01	1.18	51.94	6.66
bw_07.png	3,807,864	289	4.56	1.58	1.41	63.89	6.88
bw_06.png	9,980,928	89	4.08	1.49	1.32	62.57	6.73
bw_04.png	12,564,000	520	6.21	1.46	1.22	64.68	6.70
tree.png	16,069,074	126	4.47	1.36	1.15	60.86	6.55
bicycle.png	26,526,851	156	5.66	1.48	1.21	63.95	6.71

Input			Measured Speedup			Theoretical Speedup	
File	Pixels	Iterations	CUDA	OMP	TBB	CUDA	OMP/TBB
bw_03.png	64,050,006	1172	7.51	1.87	1.46	51.83	6.89
fractalA.png	100,000,000	2028	7.91	2.00	1.56	53.03	6.91

Next we turn our attention to the impact of size on speedup. On the one hand, as we were expecting, the CPU backends have a stable speedup across image size for the different number of required iterations. On the other hand, the GPU backend offers a speedup increasing with a logarithm-like slope that is more visible with higher number of iterations. Furthermore, it is obvious that in all cases the theoretical speedups are much higher than the measured ones.



Model Limitation

As we have briefly discussed before, there are a few factors that are not taken into account by the computation model used to make the theoretical prediction. We will mention a few of them here.

First of all, the memory model does not consider any limitation on concurrent access to memory banks. Then, it should be noted that cache misses and its additional latency are ignored. And probably more importantly, the access pattern to memory, which can have a highly significant impact on performance, is not integrated in the theoretical model. For example, a stride-two access implies a 50% bandwidth penalty. More generally, the impact of the delay of memory access, even though the CUDA pipeline attempt to hide it with its parallelism and warps schedulers, can have a noticeable effect on the running time.

Beside memory limitations, GPUs have a limited number of ALUs and some threads need to queue when all computational circuits are processing data until the pipeline can be further filled. Although this algorithm doesn't involve any, 64-bit operations can reduce the throughput down to 1/3 of its maximal value.

Finally, the model doesn't include the impact of thread divergence but, similarly to operation on wide types, this algorithm should be exempt of such limitation. However, in some cases using *if*-statements in the kernel function can in fact increase the performances. For example, if we avoid examining the neighbours of white pixels then we get the following speedups:

tree.png	Measured Speedup		
	CUDA	**OMP**	**TBB**
Without branching	4.47	1.36	1.15
With branching	9.93	7.66	3.70

On the CPU backend we get better speedups as expected since there is no concept of thread divergence on that hardware. The same phenomenon occurs on GPU with this specific image. This result highly depends on the topology of the input data: in this case, white pixels are grouped together in a favourable fashion for the warps and their schedulers, the thread divergence is low and is compensated by the mere fact that less operations are carried out.

While all those elements can explain the difference between the theoretical and real speedups, having a model that take them all into consideration would be unfortunately too complex to use.

That being said, the model is not the only reason that can explain such difference: the serial implementation was significantly optimised – such as relying on highly dense data structure, taking short circuit paths in boolean expressions or using a single buffer in combination with a queue for marked pixels – but such optimisations could not be applied to the CUDA implementation mostly for hardware reasons, hence more work for the parallel version has to be carried out which result in a longer processing time.

Conclusion

We have seen that the thinning algorithm by Z. Guo and R. W. Hall can be easily parallelised with a functional design and the help of Thrust. We can get a speedup close to 8x on GPU for some input image, such as *fractalA.png*, but only a speedup of 2x on CPU, which is not really convenient on this hardware where the optimal speedup would be 8x.

A few shortcomings of the model were mentioned in the previous section explaining why there is a big gap between the theoretical performance predictions and the measurements with CUDA. It could be informative to perform an in-depth analyse of how the TBB and OpenMP versions could be improved to take further advantage of the hardware. The design could be revised to avoid maintaining two buffers. This would therefore imply devising a new speedup formula and recompute theoretical speedups.

Without severely changing the parallel implementation, one could implement a denser data structure to store each pixel on only 1 bit of memory instead of 8. Comparing the running time of this new version and the current one could highlight some interesting metrics involved in memory and cache transactions and their limits.

It was also noticed that the OpenMP backend was more efficient than the TBB backend of Intel for relatively big images. However, it might well be that TBB is more suited for other kind of application.

Additionally, it was highlighted that using branching with CUDA can, in some contexts, improve the overall performance of the algorithm even if this implies introducing divergence among threads in a warps.

Finally, when I/O are included in the speedup measurements, we see the theoretical acceleration significantly drop. For example, with *tree.png*, we have a read time of about 0.5 second and 2.5 more seconds are required to save the image to disk on our hardware. It follows that the theoretical speedup is now 7.33x with CUDA, respectively 3.91x with OpenMP, with the 8 bits per pixel storage. Measuring the running time gives us speedups of 3.14x and 1.31x, respectively, which substantially reduce the gap between the theory and the practice. Furthermore, this shows that, for concrete scenarios, the bottleneck is not necessarily the main algorithm itself but often what is involved to gather and store the data.