Concurrent Computation of the Max-tree on GPU

Nicolas Blin
(supervisor: Edwin Carlinet)

Technical Report n^ooutput, July 2021
revision

The max-tree is a data structure used to hierarchically represent connected-components efficiently. This morphological tree has a wide spectrum of application. Unfortunately, computing the max-tree of an image is a slow process, even with modern CPU concurrent algorithms. In this report, we present a new way of accelerating the max-tree creation by introducing the first-ever GPU implementation.

L'arbre-max est une structure de données efficace pour représenter de manière hiérarchique des composantes connexes. Cet arbre morphologique possède un large spectre d'application. Malheureusement, construire l'arbre-max d'une image est un processus lent, même avec les récents algorithmes concurrents sur CPU. Dans ce rapport, nous présentons une nouvelle manière d'accélérer la création d'un arbre-max en proposant la toute première implémentation sur GPU.

Keywords
Component tree, max-tree, connected operators, mathematical morphology, algorithms, GPU

Laboratoire de Recherche et Développement de l'EPITA
14-16, rue Voltaire – FR-94276 Le Kremlin-Bicêtre CEDEX – France
Tél. +33 1 53 14 59 22 – Fax. +33 1 53 14 59 13
nicolas.blin@epita.fr – http://www.lrde.epita.fr/

Copying this document

Acknowledgements

I would like to thank the LRDE for allowing me to keep on discovering the research world for a second year.

I would also like the especially thank Edwin Carlinet who allowed me to work on this fascinating subject. Thanks to him, I was able to truly step up my knowledge in C++, parallel programming, GPU programming, and micro-optimization. He helped me to improve on all the crucial skills I wanted and still want to master to perhaps pursue a PhD thesis.

Contents

Chapter 1

Introduction

1.1 Component Trees and Applications

In mathematical morphology, connected filters allow modifying an original image by only merging flat zones, thus without modifying original image contours. Multiple hierarchical representations (morphological trees) exists to represent these connected components : Max/Min-tree 1.1 and the Tree-of-shape. Numerous efficient algorithms exist to compute the min and max tree (Edwin Carlinet, Thierry Geraud (2014)). The Tree-of-shape can be built using the max-tree (Edwin Carlinet, Thierry Géraud, Sébastien Crozet (2018)).

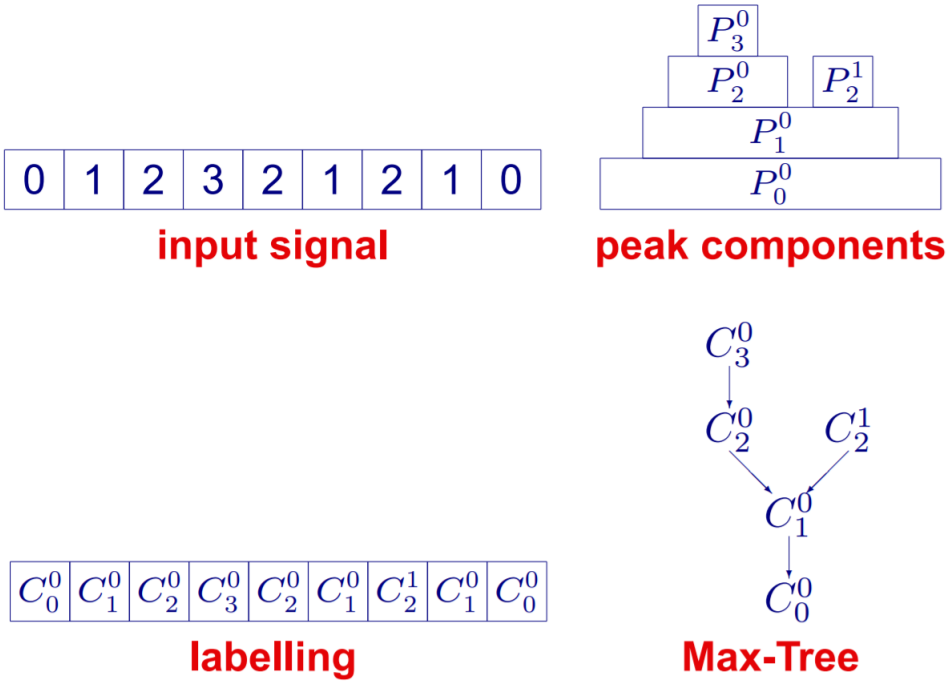


Figure 1.1: Max-tree representation

Thanks to these representations, new more advanced (compared to connected filters) forms of filtering were created (Michael Wilkinson (2004)): based on attributes 1.3, using multiscale approach strategies 1.2...

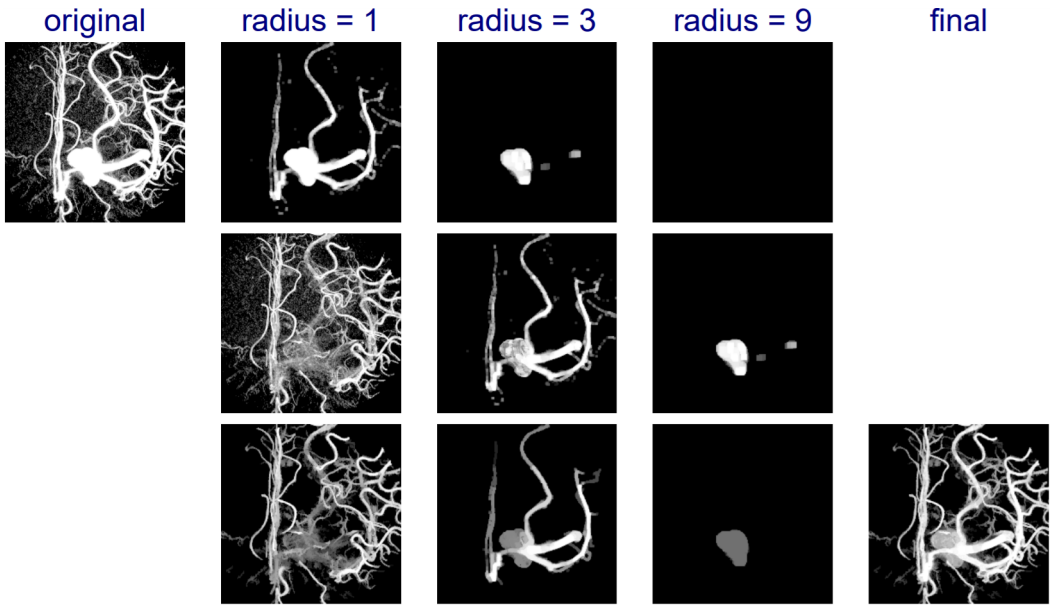


Figure 1.2: Multiscale approach used to identify an elongated structures (vessels) by using top-hat filters of different radius

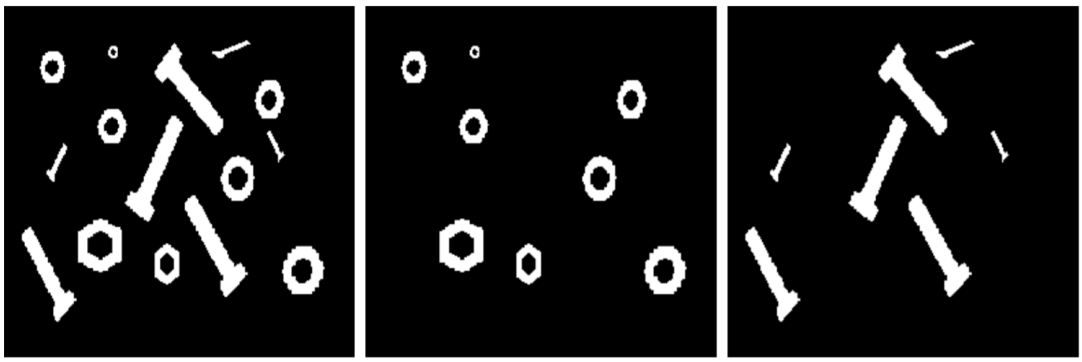


Figure 1.3: Selection of either screws or nuts based on their shape

Applications of the max-tree are wide: computer vision, features extraction, segmentation, 3D visualization...

1.2 Need for speed and models of high-performance computing (HPC) architectures

Even with the current state-of-the-art algorithms using multiple threads, computing the max-tree of an image is a slow process. It's currently not possible to compute max-trees in real-time (coming live from a video stream for example).

New approaches exist using distributed machines. The main focus of these distributed machine approaches is to compute max-trees on huge images (tens of gigapixels). A single machine could simply not hold that much data, hence the need to distribute the work. Therefore, these approaches were not aiming speed but rather scalabilty.

To compute a max-tree faster than the current state-of-the-art multi-threaded algorithms, we decided to take a look at GPUs. There is currently no implementation of the max-tree on GPU, this paper aims at correcting that and hopefully, get better performance than a CPU.

We will first start by reminding the GPU programming model. This will allow the reader to better understand the algorithmic choices. Then, we will present the main max-tree algorithms and how it is usually represented, in order to explained in detail our GPU algorithm to compute the max-tree. Also, we will precise the different optimization techniques we used to speed-up our implementation. Finally, we will take a look at the benchmarks.

Chapter 2

Reminder about GPU programming models

GPUs work by executing groups of threads (thread blocks).

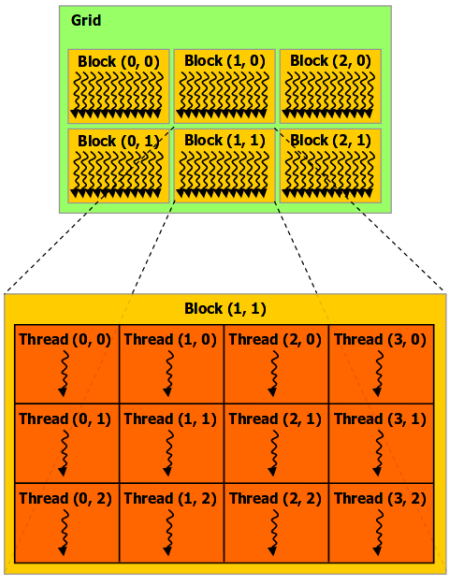


Figure 2.1: A grid of thread blocks

A GPU contains multiple SMs (streaming multiprocessors) and each SM can handle multiple thread blocks.

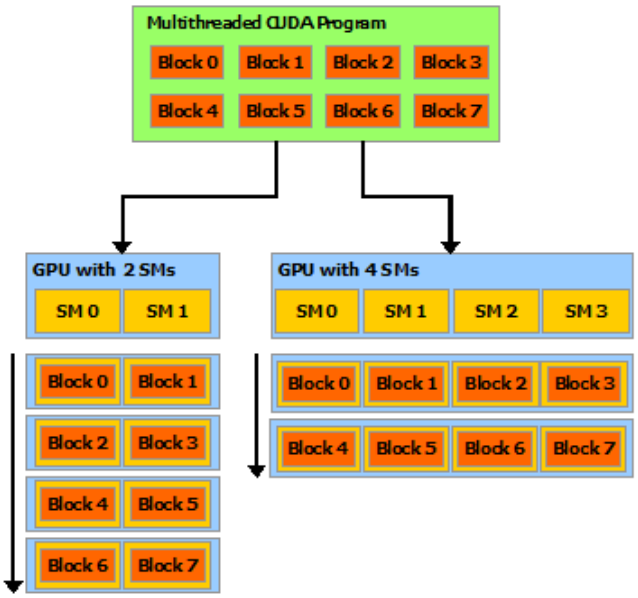


Figure 2.2: Scalabilty evolving with the number of SMs

In each thread block, threads are executed by groups of 32 called warps. These warps execute in a SIMD fashion called SIMT. Those threads need to execute the same instruction. When diverging (if else) the GPU will first, deactivate the threads going inside the else, only allowing the threads going through the if to execute and then do the opposite.

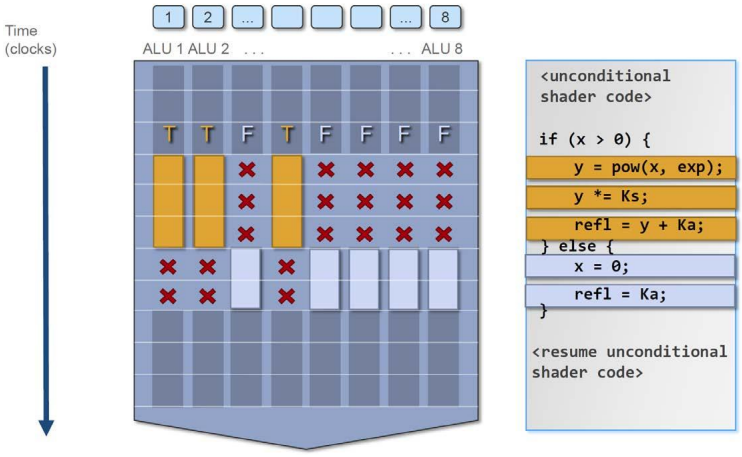


Figure 2.3: Handling thread divergence with SIMT

When a thread wants to access a memory location, a transaction is created. A memory transaction allows a thread to access a chunk of data (never a single byte). If a thread warp wants to access 32 contiguous bytes, since the usual size of the bus is 128-bits a single transaction is made.

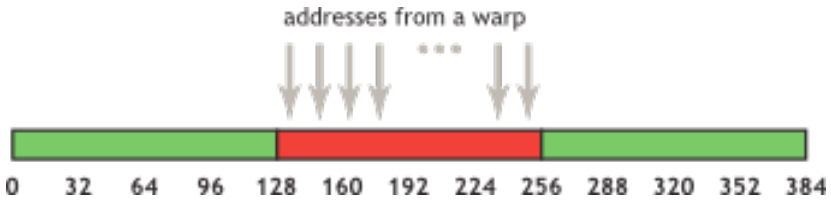


Figure 2.4: Aligned chunk access through warp

The time to access a memory location varies widely depending on its location. Accesses to registers or L1 cache take only a few cycles while accesses to global memory (DRAM) can take up to hundreds of cycles.

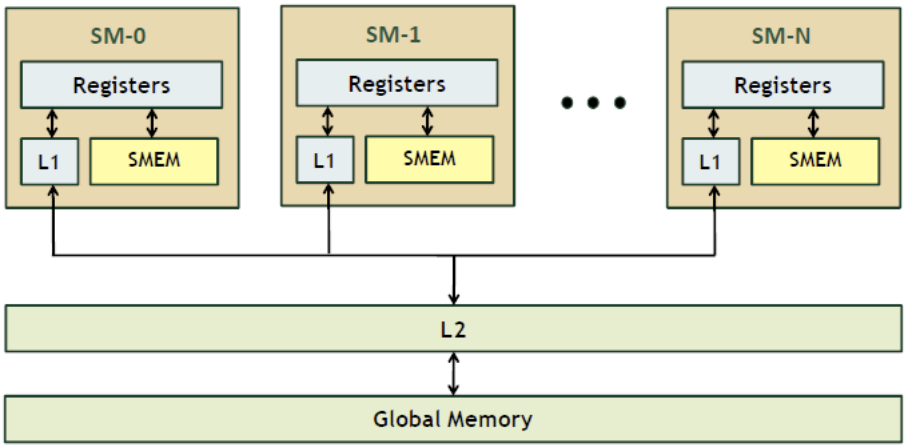


Figure 2.5: GPU's memory hierarchy

In consequence, if we want a fast GPU built max-tree, we need to respect a few conditions. The algorithm needs to be sufficiently split-up to benefit from the massive number of threads and thread blocks. We need to reduce as much as possible divergence between threads inside the same warps. We need to strive to make most of our memory accesses coalesced and the less random possible to benefit from the caches and not overload the GPU with memory transactions.

Chapter 3

State-of-the-art algorithms and Max-tree representation

3.1 State-of-the art algorithms

Mainly 3 kinds of algorithms exist to build a max tree :
Immersion algorithms, flooding algorithms and merge-based algorithms.

Immersion algorithms (Robert Tarjan (1975)) start by building disjoint singletons for each pixel of the image and then sort them according to their gray value (for grayscale image). Then, to merge these singletons and form the max-tree, we use the union-find algorithm. Union-find tracks disjoint connected components and merge them.
It's complexity is bound to the sorting algorithm and thus to quantization. On large images (>256 Mega-pixels), the union-by-rank technique described in L. Najman and M. Couprie (2006) (used to improved performance) leads to a huge memory cost and thus performance loss (swap memory). With no memory limitation, it is the most efficient algorithm (when working with high quantized images). However, memory cost can be reduced using level-compression (Edwin Carlinet, Thierry Geraud (2014)) which works well with low quantization.

Flooding (Michael Wilkinson (2011)) starts by scanning the image to retrieve the root which is the pixel at the lowest level in the image. Then, it performs a propagation by flooding first the neighbors at higher levels i.e. a depth-first propagation.
When memory is "unlimited" and quantization is low, the flooding works best (P. Salembier and A. Oliveras and L. Garrido (1998)) when using hierarchical queues and beats union-by-rank. With limited memory and high quantization, the non-recursive flooding algorithm (Michael Wilkinson (2011)) using heaps performs the best.

Merge-based algorithms (Michael Wilkinson, Hui Gao, Wim Hesselink, Jan-Eppo Jonker, A. Meijster (2008)), in contrary to the others, do not explain how to build a max-tree but rather how to parallelize it. Here we first start by dividing an image into blocks and compute the max-tree on each sub-image using another max-tree algorithm. Then, sub-max-trees are merged to form the tree of the whole image.
The efficiency of merged-base algorithm is tightly linked to the algorithm used to build the sub-tree. Because of post-processing, flooding algorithms scale poorly with a higher thread

count while union-find based algorithms (immersion) especially with level-compression scale the best.

More in-depth descriptions, comparisons and benchmarks can be found in Edwin Carlinet, Thierry Geraud (2014).

None of these algorithms are well-suited for GPUs. For immersion, sorting is not well suited for massively parallel workload and pixel merging induces a lot of non-coalesced accesses (during union-find). Flooding is worse, depth first search scales really poorly on GPUs and the algorithm is constantly making non-coalesced accesses. Merged-based algorithms guide us on how to merge our trees after building them but we need to find an efficient way to form them in a first place.

New distributed algorithms now exist. They allow to build the max-tree of multiple Terapixels images (Jan Kazemier, Georgios Ouzounis, Michael Wilkinson" (2017), Markus Gotz, Gabriele Cavallaro, Thierry Geraud, Matthias Book, Morris Riedel (2018)).

The last kind, less known but which will prove to be really useful in our case, a 1D max-tree algorithm for line/row images (David Menotti, Laurent Najman, Arnaldo Araujo (2007)).

3.2 Maxtree-representation

Max-trees can be represented using L. Najman and M. Couprie (2006) representation 3.1. The image in itself is a matrix but the tree is also represented as an image (i.e. a matrix). A parent relationship exists between two pixels of the image if A, a first component is directly included into B, a second component (i.e a covering relation). Inside a component, all the pixels point toward the same pixel that represents this component. It is called the canonical element (or level root). This image and parent representation allow us to directly access the gray level of any pixel as well as its parent from its index.

To represent the max-tree in our Cuda program we use two 2D matrices. One of bytes, the other one of 2D points.

A 2D point is a structure composed of 2 fields (x,y).

Each cell of the parent matrix represents a pixel of the image. The data contained in the cell is the localization of its parent. For example, if cell (0, 0) contains the value (1, 0), it means that the pixel of the image at position (0, 0) is the child of the pixel at position (1, 0) (and the pixel at position (1, 0) is the parent of the pixel at position (0, 0).

To represent the image we simply use a 2D matrix of bytes to hold the gray values.

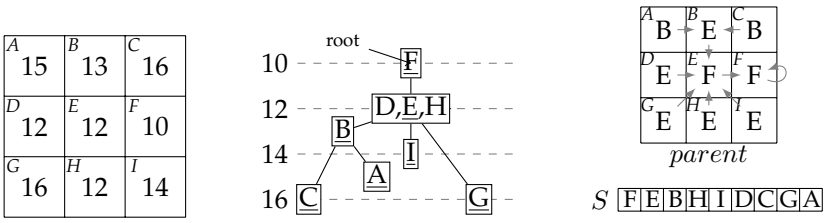


Figure 3.1: An image, its max-tree and parent image

Chapter 4

Max-Tree algorithm on GPU

4.1 Global Pipeline

In our approach, we first start by splitting our image into tiles (1).

Inside a tile, each thread starts by building a local max-tree for each column using the 1D algorithm (2a).

Those local max-trees are then merged concurrently. Each thread merges 2 column max-tree by calling along the common border an atomic union-find. After this step each tile contains one fully built max-tree (2b).

After this step, we use the same union-find to first merge the tiles horizontally and ultimately vertically (3)..

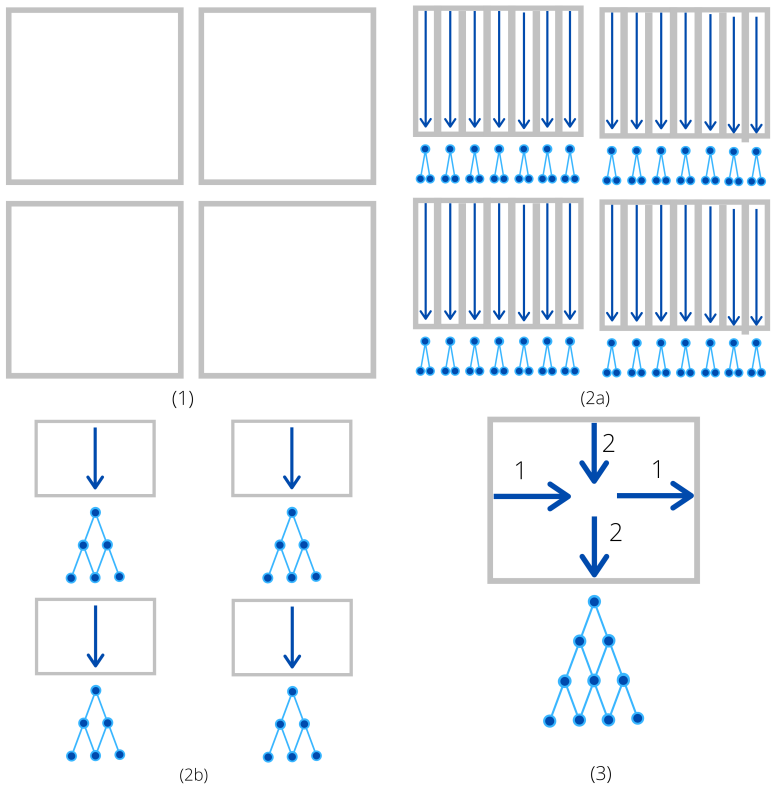


Figure 4.1: Global pipeline of the GPU algorithm

A first approach used a non concurrent union-find to merge the columns. Each thread would merge columns 2 by 2 using a reduce pattern. This first approach allowed for each thread to work on its own set of nodes avoid the use of atomic operations but suffered from 2 drawbacks. A $log2(n)$ steps complexity and and number of thread being halved at each step leading to poor performance. The same technique and issue would appear during the tile merging step.

Our new version algorithm addresses this issue by using a modified atomic version of the union-find.

4.2 Max-tree computation per-tiles

4.2.1 Column-wise processing

The first step of our algorithm starts by building local max-trees in each tile. One thread block is assigned to each tile. Inside a thread block, each thread is at first responsible for building one local max-tree for its column.

For example, with tiles of size 64x64, a thread block will contain 64 threads (2 thread warps). Each of those 64 threads will build a max-tree along its column, resulting here in 64 max-trees of height 64.

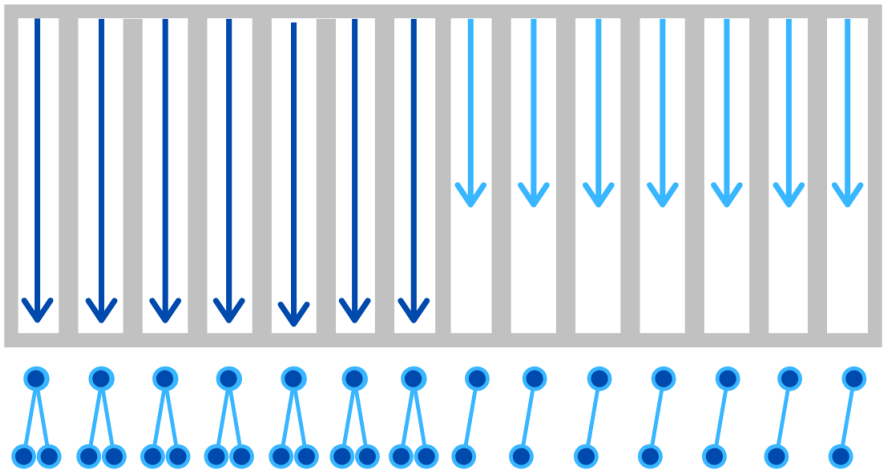


Figure 4.2: Two thread warps on one tile building the column max-trees

To build those local max-trees we use a slightly modified version of the 1D Menotti algorithm.

```
1  procedure 1D_max_tree(image, parent):
2      start_y := top_tile_index
3      r := (x,start_y)
4      for y := start_y + 1 to tile_height do
5          p := (x,y)
6          if image(r) < image(p) then
7              stack.push(r)
8              parent(r) := r
9              r := p
10         else if image(r) = image(p) then
11             parent(p) := r
12         else
13             while !stack.empty() and image(stack.top()) >= image(p)
                   do
14                 parent(r) := stack.top()
15                 r := stack.top()
16                 stack.pop()
17             if image(p) < image(r) then
18                 parent(r) := p
19                 r := p
20             else
21                 parent(p) := r
```

```
22
23      while !stack.empty() do
24          parent(r) := stack.top()
25          r := stack.top()
26          stack.pop()
```

Listing 4.1: Column max-tree algorithm

We iterate over the 1D image from top to bottom. *p* represents the current pixel (at index y in the current tile) and *r* the last pixel (at index y - 1). *Image* holds the gray values, *parent* the position of the parent of each pixel. They are both accessed through the *point* data-structure representing the pixel position.

The algorithm builds the max-tree based on the gray level of the current pixel compared to the ones of already seen pixels, especially the very last one (*r*).

Three main cases can arise based on the gray level. For the sake of simplicity we will use *ph* and *rh* to denote the gray levels of *p* and *r*.

If *ph* > *rh*, we need to create a new component. The last pixel (*r*) is thus placed inside a stack for later processing since it has at least one descendant. The parent of *r* is set to *r* to handle the root case (the parent of root is root).

If *ph* = *rh*, they are part of the same component, a parent relationship it thus set between the two. The last pixel (*r*) becomes the parent of *p*. In this case, we keep *r* as our last pixel (in the other cases *p* becomes the last pixel). This is done so that all the pixels of the same component are linked to the same pixel (level root).

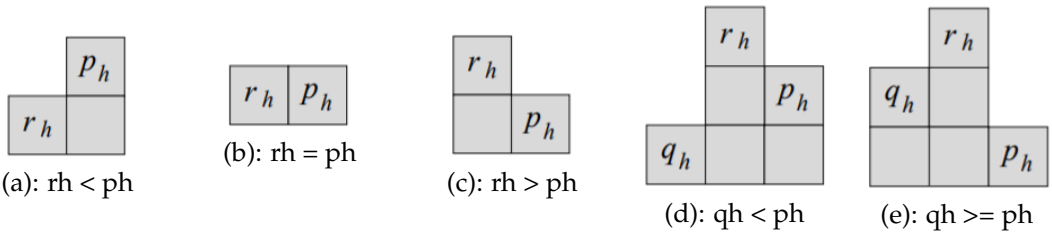
The third case (*ph* < *rh*) is itself divided into several scenarios:

If the stack is empty, we are assured that *ph* <= *rh*. If *ph* < *rh* then we are assured that p is the parent of *r*. Indeed, if there was any pixel with a gray level lower than *rh*, and that could thus be a parent, it would have been in the stack. Else, the levels are guaranteed to be equal, we thus act like in the second case.

If the stack is not empty, the parent of r might be in the stack. We pop nodes out of the stack while their levels are greater or equal to *ph*. If it is, then we found a parent for *r* and set *r* to the top of the stack to keep on linking all of the nodes together.

If the stack gets emptied or if the level of the top of the stack is smaller than the current pixel then the two only possibilities left are the one already described.

Figure 4.3 shows the different cases that can arise. (a) and (b) represent the first two cases. (c) represent the third case with an empty stack. For (d) and (e), *q* represents the pixel inside the stack. Image (d) shows the case *ph* > *qh*, making us break out of the while. Image (e) represents the case where we found a parent for *r* in the stack.



(a): rh < ph (b): rh = ph (c): rh > ph (d): qh < ph (e): qh >= ph

Figure 4.3: The 5 cases that can arise during the 1D algorithm

4.2.2 Merging columns using a concurrent union-find

After building the column local max-trees for each tile, we need to merge the columns using union-find.

To do so, threads are firstly synchronized to be sure that all the threads are done building their 1D max-tree. Then, each thread inside each block iterates along its column (from top to bottom) merging one by one its current pixel with the pixel directly to his right (leading to discarding the right-most thread). The columns will thus all be merged concurrently.

In our 64x64 example, the first 63 threads will start merging. Thread 0 will call union-find on pixels (0, 0) and (0, 1), thread 1 will call union-find on pixels (0, 1) and (0, 2)... After completing their first merge each thread then continues on to the next pixels, (1, 0) and (1, 1) for thread 0 and so on. Threads are not waiting for each other (no thread synchronization). To handle nodes being shared by multiple threads, writes to modify parent relationships are performed in an atomic way.

Listing 4.2 below shows the code used to merge two column max-trees inside each tile. The same code is used to merge tiles together.

```
function find_peak_root(a, level):
    level_root := a
    q := parent(a)

    // While peak root level or root is not reached, climb parent
    while !is_root(q) and image(q) >= level:
        level_root := q
        q = parent[q]

    return level_root, q

function find_level_root(a):
    // Call with level of a passed to reach level root of a
    return find_peak_root(a, image(a))

procedure union_find(a, b):
    while True:
        // Always climb right branch
        if image(a) > image(b):
            swap(a, b)

        // Find current level component roots & peak component roots
        a, a_parent = find_level_root(a)
        b, b_parent = find_peak_root(b, image(a))

        if a == b:
            return

        // Ensure an ordering between pixels based on indexes
        if image(a) = image(b) and b < a:
            swap(a, b)

        // Merge
        old = atomicCAS(parent(b), b_parent, a)

        if is_root(old):
            return
        if old == b_parent:
            b = old

procedure merge_tree(Di, Dj):
    for all (a, b) along Di and Dj do
        union_find(a, b)
```

Listing 4.2: Concurrent union-find algorithm

The goal of union-find is to take disjoint sets and merge them. Each set is represented by a branch (in our case, a branch of the max-tree).

Let *Di* and *Dj* be the two regions being currently merged. For two neighbors *a* and *b* in the junction of *Di* and *Dj*, we loop through *a* and *b* branches until a common ancestor is found.

We start by describing the union-find procedure (line 16). To avoid branching in the later code, we always climb the right branch. We start by retrieving the level-root of *a* and the peak-root *b*. The level-root is the node representing *a*'s connected component. The peak-root is the node in *b*'s branch that has the lowest gray level that is itself lower than the gray level of *a*. Once found, *parent(b)* can be set to a to create the new parent relationship.

If *a* is equal to *b*, a common ancestor was found and the procedure ends.

To ensure an ordering between pixels of same gray level we use indices. So if *b*'s index is lower than *a*'s, nodes are swapped.

At this point, we try to connect *parent(b)* to *a*. This is done atomically as nodes are shared among threads. *AtomicCAS* checks if *parent(b)* is equal to *b_parent*, if so, *a* is set as *parent(b)*, else nothing happens. If *parent(b)* is not equal to *b_parent* is means some other thread went first.

If *old* is root then we are done climbing the right branch and thus the procedure ends. If root is not reached we need to keep on climbing. *b* is set to *old* to continue the merging. If *b* is not equal to *b_parent*, another thread went first we thus try again using the new *b* (retrieved in *find_peak_root* at line 3).

Function *find_level_root* (line 12) is used to retrieved *a* level-root. This is performed by calling *find_peak_root* with *a*'s gray level as level. This effectively climbs *a*'s branch until the level-root is reached (as we climb while the level is the same).

Function *find_peak_root* (line 1) is used to get the peak-root element of *b*. It traverses the branch upward while the node level is greater or equal to *a*'s level and root is not reached.

Function *union_find* (line 16) traverses both branches up to find a common ancestor and connects the two pixels accordingly.

Our version differs from the classical version in two ways. Firstly, no path compression is performed as atomic writes are very expensive. Secondly, we use an atomicCAS to create the parent relationship instead of a regular write to allows this function to be called concurrently on every column.

4.3 Merging tiles

At this step, we have now P tiles, P max-trees that we need to merge. Before merging the tiles, we need to be sure that every thread block is done with its tile. Since CUDA only allows synchronization between threads inside a block and not between blocks, we need to wait for the whole kernel (CUDA function) to be done before calling the next one.

A first kernel is used to merge tiles horizontally, then a second to merge them vertically. The concurrent union-find described above is used to perform those merges.

A last kernel performs a flattening operation to ensure that all nodes of a connected component point toward the level-root.

4.4 Optimization and performance considerations

In this section, we explain why each of the algorithms chosen is relevant for GPUs. We will also discuss how the merging steps have been significantly sped up.

The 1D algorithm was selected because it perfectly fits our GPU constraints. Indeed, threads should avoid diverging and we should strive for coalesced accesses. When building the local max-trees inside each tile in such a way, both these conditions are respected. Every thread starts by loading the first pixel of its respective column and thus, all the threads together are computing a single line. Globally, threads inside a thread block are moving downward together, treating one line after another. This results in only a few transactions being made to the slow global memory. On the contrary, if we take the example of the flooding algorithm, each thread would be accessing different parts of the image. These accesses would be almost random and each thread, based on the gray value of its tile, would have very different execution paths.

The merging part did not fit at first what we needed. In our first approach, a non-concurrent union-find was used. We used to simply merge column max-tree 2 by 2 using a reduce pattern. This led to $log2(n)$ steps and the number of threads being halved at each step. This technique allowed each thread to work on separate sets of nodes (i.e. no data shared among threads) but yielded pool performances. Using a concurrent union-find meant we could merge all columns concurrently leading to a single merge step with a single thread being lost.

The same strategy was extended to tile merging. Since the previous version relied on the non-concurrent union-find, the tiles, like the columns, were merge 2 by 2 which inducing the same drawbacks. Using this new concurrent union-find led again to the same benefits, only 1 merge step and more active threads which again led to a great performance improvement.

Chapter 5

Benchmarks and performance analysis

All benchmarks were executed on a 6000x4000 image. The CPU used is an Intel Core i7 9750H with 6 cores / 12 threads. The GPU used is an Nvidia RTX 2060 with 1920 Cores spread across 30 SMs.

Our GPU algorithm yields the following results when compared to the fastest multi-threaded CPU algorithm

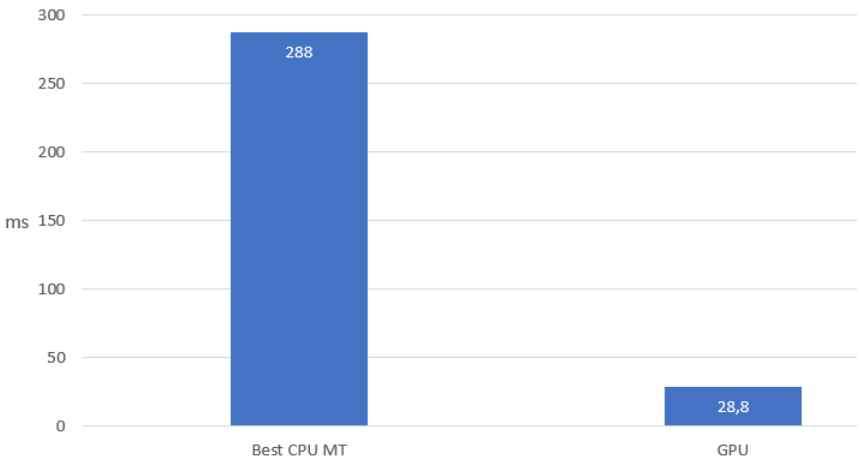


Figure 5.1: Benchmark comparing a full max-tree building CPU vs GPU

Our GPU algorithm leads to a 10x speed-up. In practice, this translate into 793 MB/s. For HD images (1920x1080), this result meant handling 382 frames per second. Our initial real-time goal is thus met.

We can now analyse the time spent during each step :

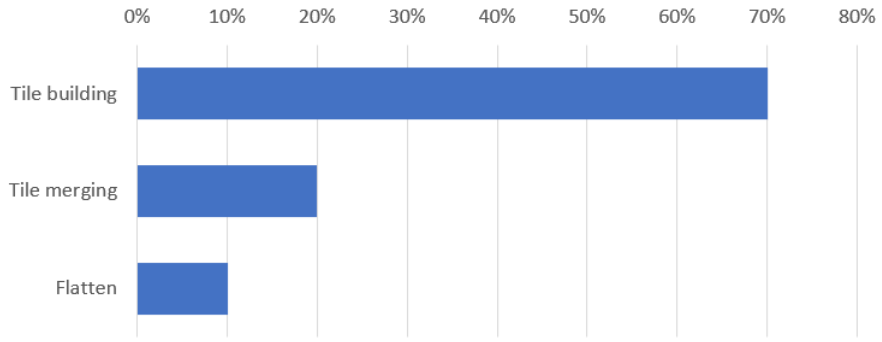


Figure 5.2: Percentage of time spent during each step of the GPU algorithm

The tile construction is already highly optimized: fast 1D algorithm, shared maximal utilization, concurrent column merging...

Our goal is to reduce the time spent on tile merging. The issue with those merges is global sparse atomic writes. Indeed, during the execution of our modified union-find, both the image and parent image are accessed in a not-coalesced way (one node and its parent are most of the time far from each other in memory) in global memory. Changing parent relationship means making an atomic write in global memory.

Our approach tries to solve those issues by increasing data locality, lowering memory use, and thus increase the chance of having cached reads/writes.

Chapter 6

Border max-tree : new approach

6.1 General idea

As stated before, the issue during the tiles merging is data locality. Our approach tries to address this issue by using two techniques.

- Instead of merging directly both tiles, the merge is performed on the tiles border max-trees. A border max-tree contains only the nodes concerned by the merge which allows us to significantly reduce the size of the parent image and thus, increase the chance of it fitting in cache.
- To speed-up the horizontal merge step, border max-trees in each column are placed next to each other in memory thus increasing data locality.

These two techniques result in an optimal horizontal merge: data is reduced to the strict minimum and is placed in memory as close as possible. This reduces long jumps in memory when accessing a parent node significantly.

6.2 Border max-tree

The border max-tree was first introduced in Jan Kazemier, Georgios Ouzounis, Michael Wilkinson" (2017). The idea is as follows:

During a merge of two tiles, an union-find algorithm is called along the shared border. Union-find algorithm works by climbing both branches and updating the parent relationship along the way. We can conclude that during a merge, only nodes on the border and parents higher in the hierarchy will be concerned by the merge. Thus, to reduce memory consumption, one can extract the border max-trees of both tiles, call union-find over the border max-trees and then write back the changes in the global parent image.

6.3 New algorithm pipeline

In this new algorithm pipeline, two steps are added. As before, the image is split into tiles. Inside each tile, a local max-tree is built using the 1D algorithm and an union-find.

The first new steps appears here. The goal is to extract the border max-trees of the local max-tree (from the top and bottom borders). This operation is processed in the same kernel as the tile border max-tree construction. It only works with one thread by column. Once the border max-tree is extracted for each tile, the borders are compressed next to each other column wise.

The next step is to merge the tile horizontally by using the union-find algorithm. This time, the merge is processed within the border max-trees (instead of the global parent image). The algorithm merge each couple of top-border and bottom-border.

This merge is processed within the border. Thus, a new step is required. The merged border must now be committed into global parent image. Every parent in the border is translated to the global parent images using a node-map (built during the border extraction and compression).

After this step, the global parent image is merged column wise. The column must be merged between them. This step also uses the union-find algorithm but this time to merge vertically. Finally, the last step is to flatten the global image parent (these last two steps are identical to the former GPU algorithm).

Figure 6.1 below shows how two pixels far in memory are now going to end up close to each other. This is thanks to the column-wise memory ordering and size reduction induced by the border max-tree.

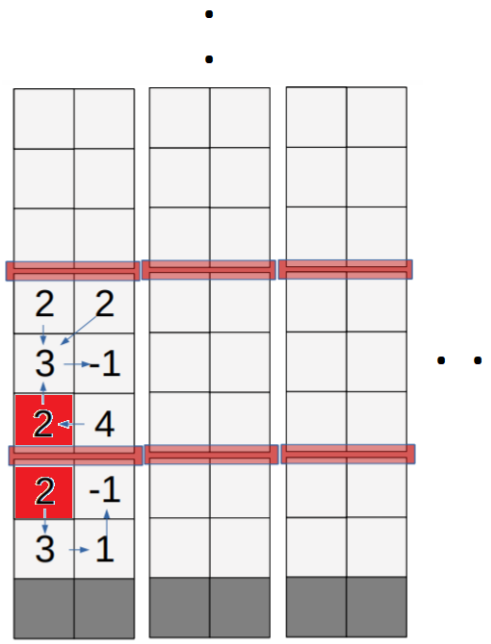


Figure 6.1: Two pixels, far in memory

A complete example of our entire algorithm execution can be found in appendix (8).

6.4 Border extraction

In this section we will detail the functioning of our parallel border extraction algorithm.

The general idea is described in the pseudo-code below :

```
1  + Each thread starts on one node x of the border
2  + mark[x] is set for every node on the border
3  + Count = border size and tacks size of the border
4
5  mark[x] = True
6  Count = border_size
7
8  while True:
9     q = parent[x]
10
11    // Node already visited
12    if Atomic_Test(&mark[q]) == True:
13       break
14
15    x = q
16
17     // Increment counter and mark node
18    Atomic_Add(&Count, 1)
19    mark[x] = True
```

6.4.1 Parallel border extraction algorithm

Needed buffers

To perform the border extraction we need new buffers :

- gborder: border max-tree of tiles stored by column (size = image size)
- gtranslate: associate each node in the border to its location in the global image block wise. (size = image size)
- gtranslate_compress: associate to each node in the border to its location in the global image column wise. It uses the same indices as gborder. (size = image size)
- border_input: associate to each node in the border to its grayscale level in the global image column wise. It qlso uses the sqme indices as gborder
- lut_border_pos: index for each tile of its position in the gborder (size = number of tiles)
- gcolumn_sizes: Store the accumulated size of every border for each column (size = number of columns)

State of data when starting the extraction

Border extraction is the last function called during the intra-tile step (after 1D, column merges, fattening and global commit). Border extraction is performed in shared memory to speed up reads and writes. Data left in shared from previous steps will be partially erased and partially reused.

Shared memory is organised as a 2D array representing both the image and parent image of the tile. Each cell is 4 bytes long. 1 byte stores the node gray level, 1 byte stores the parent gray level, 1 byte stores the parent y position, 1 byte stores the parent x position. Root is signified using INT8_MAX (127) in the parent y position byte and UINT8_MAX (256) in the parent x position byte. This restricts our tile sizes to 256 in width and 128 in height. Choosing to use INT8_MAX for the y and not UINT8_MAX is because we need the sign bit to mark visited nodes during border extraction algorithm. We need to ensure we do not miss-treat root for an already visited node.

State of data during the extraction

During the border extraction, shared memory is reused. The first two bytes (big endianness is considered here) were used to store the gray levels of the node and the parent's node. This data is now useless and can be overwritten without any problem. The two last bytes are parent position inside the shared memory. This information can be stored locally by the thread handling the node by overwriting as no other thread will ever need this information again: only the node currently handling the node will jump to its parent.

The memory is thus reused as follows: The first two bytes are used to store the extracted border max-tree in a 1D way. In the first tile width cells, we store the location of their parent inside this same 1D considered shared memory. Meaning: if cell 0 has value 7 in its first two bytes, the parent of the first node of the top border (we place ourselves in the general case were both border needs extraction) is located at the position 7 inside shared memory. The last two bytes are used during border extraction. Here, we store the position of this node inside the border. Meaning : if cell (3,3) of shared memory is accessed and 7 is found in the last two bytes, we already know that is node will be placed at position 7 inside the border. To be mode specific, the value found will not be 7 but -8. The - is used to mark visited nodes. When a thread handles a node, it compute its position in the border and encode it in the last two bytes. This allows future threads handling a node which parent is this already visited node to directly write in the shared border the corresponding index. The 8 instead of 7 is used to handle the 0 case. If we want to mark the node of index 0 with our technique, it would result in writing -0 inside the cell resulting in not marking the node. To cope with this issue, all parent indices in the last two bytes are offset by 1.

This data layout allows to directly commit the extracted border in global memory: we just loop through shared memory and read the first two bytes. It also allows during border extraction to mark nodes as visited and store their border position, allowing threads that revisits this node to directly know their parent position inside the border.

Extract border algorithm setup

The algorithm starts by initializing the shared border size variable. This variable serves 2 purposes. Keep track of this tile border size and know at each step where should be append the next border node. This variable is either set to tile width or tile width * 2 whether we need to extract both or only one border (details about tile width potentially being smaller for the last tiles of each row is omitted here).

The next step is to initialize the border information we already have. At this step, we can already mark the border nodes as visited since we already know what their border position will be. As examples: node n°0 of top border is marked with value -1 (because his index in the border will be 0), node n°0 of the bottom border is marked with value -(tile width + 1) (because again, we already know his position inside the border). We choose by convention to place the bottom border nodes right next to the top border nodes to be able to easily find them later during merge. If one wants to merge the bottom border of a tile, he just need to go at border index + tile width. The second information we already have is the global position of those nodes for our gtranslate buffer.

After this set-up step, border extraction can now start.

Extract border algorithm setup pseudocode

```
1  function extract_border_max_tree()
2      w = tile_width;
3      h = tile_height;
4
5      __shared__ int size;
6      if (extract top border is required)
7          if (threadIdx.x == 0)
8              if (extract_only_one_border)
9                  size = w
10             else
11                 size = w * 2
12     __syncthreads()
13
14     top_pos = threadIdx.x
15     sbottom_pos = top_pos + (tile_width * (tile_height - 1))
16
17     gtranslate += global_offset
18
19     # Same data but different representation
20     aux_border = (info_border*)aux
21
22     if (extract top border is required)
23         top_parent = aux[top_pos]
24         aux_border[top_pos].id = -(top_pos + 1)
25         gtranslate[top_pos] = compute_global_index(threadIdx.x, 0)
26
27         if (!extract_only_one_border)
28             bbottom_pos = threadIdx.x + w
29             bottom_parent = aux[sbottom_pos]
30             aux_border[sbottom_pos].id = -(bbottom_pos + 1)
31             gtranslate[bbottom_pos] = compute_global_index(threadIdx.
                   x, h - 1)
32         handle_border(top_parent, top_pos, size)
33
34     if (extract bottom border is required)
35         bbottom_pos = threadIdx.x
36         if (extract_only_one_border)
37             bottom_parent = aux[sbottom_pos]
38             aux_border[sbottom_pos].id = -(bbottom_pos + 1)
39             gtranslate[bbottom_pos] = compute_global_index(threadIdx.
                   x, h - 1)
40         else
41             bbottom_pos += w
42         handle_border(bottom_parent, bbottom_pos, size)
43
44     __syncthreads();
45
46     compress_border(size)
```

Extract border algorithm

The algorithm starts with a syncthreads to ensure that all threads of the thread block have visibility over the marked node. Then the main loop works as follows:

While root is not reached, keep on extracting. Start by checking if the parent of our current node (in our example, current node is 0) is not already marked. A syncwarp is placed before to ensure visibility over data at least at warp level, a syncthreads would be slower as it would wait for all thread of the block. Exiting here is not mandatory, it just avoids the next atomic operation that will, this time, ensure breaking from the loop if the node is already marked. If the parent node is not already marked, we try to mark it with an atomicMin(-1). Using atomicMin with -1 allows to effectively mark not visited nodes by setting the bit sign. It allows, on another hand, to avoid overwriting already marked node that would already store their position as -(position + 1) which can only be smaller than -1.

Then check the old value returned by the atomicMin. If it is negative, then the parent node was either, already visited and the initial syncwarp did not give us enough visibility, or another thread is also trying to mark this node and his atomicMin went first. In both case, the thread breaks out of the loop.
If the threads did not break, it managed to go first and marked the parent node. The border index of this node can be now computed. It atomically increments the border_size shared variable. For example, with a tile width of 4, we know that the first 8 indices (0 to 7) are reserved for the top & bottom border and the border_size shared variable is thus equal to 8. Atomically adding 1 signals to all threads inside the block that the thread is adding a new node to the border. The old value return (here 8) gives the index of the parent node. Now that the threads knows the border index of the parent node, it can safely write in the border that my current node father is 8. It can also write in gtranslate the global index of this node now that it knows its location (global index is computed using tile information and block id). Then, it writes in shared memory the -(position + 1) so that threads can later know what is this node position in the border. Finally, it updates indices to set current node to parent node and parent node to parent of the parent node (which can be found in the old value returned by the atomicMin).

Eventually all threads break from the loop, either because root was reached or because their parent was already marked. A syncthread ensure that, before the final writes, all threads are done climbing their respective branches and give visibility over data written in shared memory. If a thread broke from the loop because root was reached, it writes -1 in its current position. In the other case, the thread can safely write its parent in the border (shared memory first 2 bytes) and the parent position has been filled by another thread at some point and can be find in the last 2 bytes.

Extract border algorithm pseudocode

```
1  function concurrent_branch_extract(spos, bpos, size):
2      __syncthreads()
3
4      if (threadIdx.x >= w)
5          return
6
7      while (!spos.is_root())
8
9          __syncwarp()
10         if (aux[spos].id < 0)
11             break
12
13         old = atomicMin(&(aux[spos].id), -1)
14
15         if (old < 0)
16             break
17         else
18             border_pos = atomicAdd_block(size, 1);
19
20             aux[bpos].parent_border = border_pos;
21             gtranslate[border_pos] = compute_global_index(spos.x,
                   spos.y)
22
23             aux[spos].id = -(border_pos + 1)
24             spos = old
25             bpos = border_pos
26
27     __syncthreads()
28
29     if (spos.is_root())
30         aux[bpos].parent_border = -1
31     else
32         aux[bpos].parent_border = (-aux[spos].id) - 1
```

6.4.2 Parallel border compression in global

Arriving this step, the border max-tree is stored in the shared memory. This border max-tree must be stored column wise in global memory to be used by another kernel (border max-trees merge) later. A look up table is also needed to know where the border max-tree is stored in the column border max-trees.

This functions handle the border of each tile (CUDA block). The input of this function is the size of the tile border. As a reminder the border max-trees are stored column wise. However, the blocks should not wait each other to commit their border max-trees. The rule is first finished first stored in the column border (according to the SM blocks scheduling). Thus, the ordering of the column compression border is non-deterministic. That is why a global variable (named gcolumn_sizes) is required. This global variable stores the current accumulated column borders size.

In this function, each thread needs to know the position of their tile border in the column border. This is the purpose of the shared variable output_pos. This offset is computed as the current size of its corresponding column current size. This size is increased atomically. Thus, the next block will get the increased offset. Then, the lookup table of the block is updated with the computed offset in the column border max-trees. Finally, the synchtreads is required for all threads in the block to be aware of the new value of output_pos.

The buffers need to be offset to the column position. For instance, gborder must be offset to the start of the column border max-trees. Then, a block loop design pattern is used to commit the result into the shared memory. This pattern is efficient because it uses all the available threads in the block. The loop does the following work: for all elements in the border (stored in shared so far), store their parent in the global border max-trees parent buffer and fill the compressed node-map buffer with the uncompressed node-map buffer (offset with the blockIds). And the same happens with the border_input buffer.

Parallel border compression in global pseudocode

```
1  function compress_border(border_size):
2      __shared__ output_pos;
3      if (threadIdx.x == 0)
4          output_pos = atomicAdd(&(gcolumn_sizes[blockIdx.x]),
               border_size)
5          lut[blockId] = output_pos;
6      __syncthreads()
7
8      offset = height * tile_width * blockIdx.x;
9      gborder += offset; border_input += offset; gtranslate_compress +=
           offset;
10
11     for (i = threadIdx.x; i < border_size; i += blockDim.x)
12         parent = aux[i].parent_border
13         if (parent == -1)
14             gborder[output_pos + i] = -1
15         else
16             gborder[output_pos + i] = parent + output_pos
17         gtranslate_compress[output_pos + i] = gtranslate[i]
18         border_input[output_pos + i] = input[gtranslate[i]];
```

6.5 Commit merged border max-trees after the horizontal merge

Once the border max-trees are merged horizontally, the updated parents in the border max-trees must be updated in the global parent image. This kernel is straight forward and use a stride loop design pattern (for the same reason as using a block loop design pattern). First of all, the buffers are offset to their current column position. For each element in the border max-trees, update the global parent image with the parents in the border max-tree translated in the global parent image. If the the node is a root, no translation is required.

Commit merged border max-trees after the horizontal merge pseudocode

```
1  function write_border():
2      offset_column = height * tile_width * blockIdx.y;
3      gborder += offset_column;
4      gtranslate_compress += offset_column;
5
6      for (i = threadIdx.x + blockIdx.x * blockDim.x;
7           i < gcolumn_sizes[blockIdx.y];
8           i += blockDim.x * gridDim.x)
9
10         parent_border = gborder[i]
11         if (parent_border == -1)
12           parent[gtranslate_compress[i]] = -1
13         else
14           parent[gtranslate_compress[i]] = gtranslate_compress[
                 parent_border]
```

6.6 Benchmark and optimization consideration of the new GPU algorithm

6.6.1 Border extraction benchmark

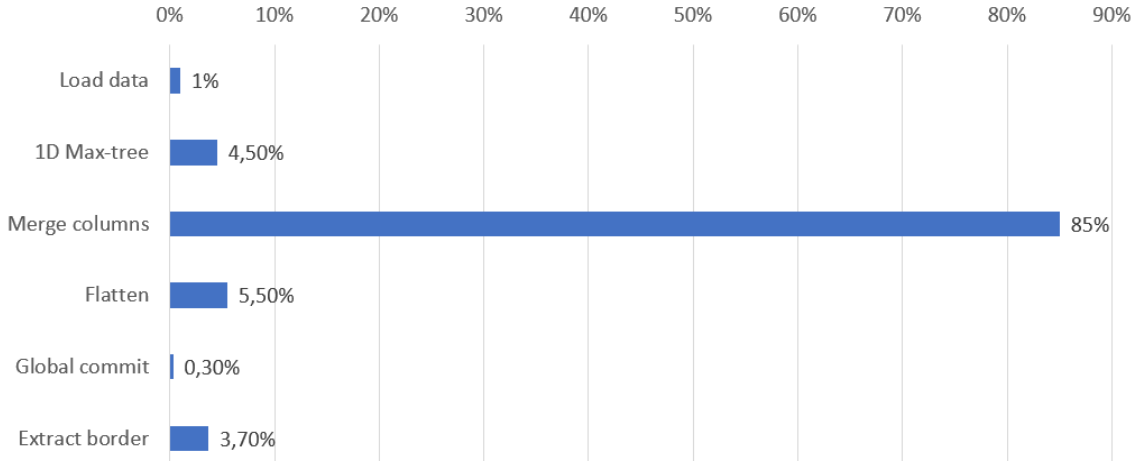


Figure 6.2: Benchmark former algorithm VS new approach

Over a 6000x4000=24 000 000 pixels images with 64x16=1024 tiles size, the mean border max-tree size for each tile is 193.5 which is around 18% of the tile. The size of a tile is reduced by 82%.

About the speed, the extraction border max-tree step followed by the compression of the buffers takes up 3% of the tile construction time. The extraction time plus the compression time is very low compared to the other steps.

The low extraction and compression times can be explained with two reasons:

- extensive use of shared memory. During the whole extraction, the tile border max-trees are stored and processed in shared memory. This shared memory is used by the previous steps and smartly reused. There is a least use of global memory. The result is only committed in the global memory at the last step of compression (mostly data transfer).
- low number of nodes to extract. The threads jumps in average to only 18% of the nodes in the tile.

The worst case is that only one thread walks over the whole tile performing 256 operations. The complexity in this case is $\mathcal{O}(256)$. And the best case is that the border max-tree is contained inside itself. In this case, the threads do not jump to other nodes. The complexity is $\mathcal{O}(0)$.

6.6.2 Max-tree construction benchmark

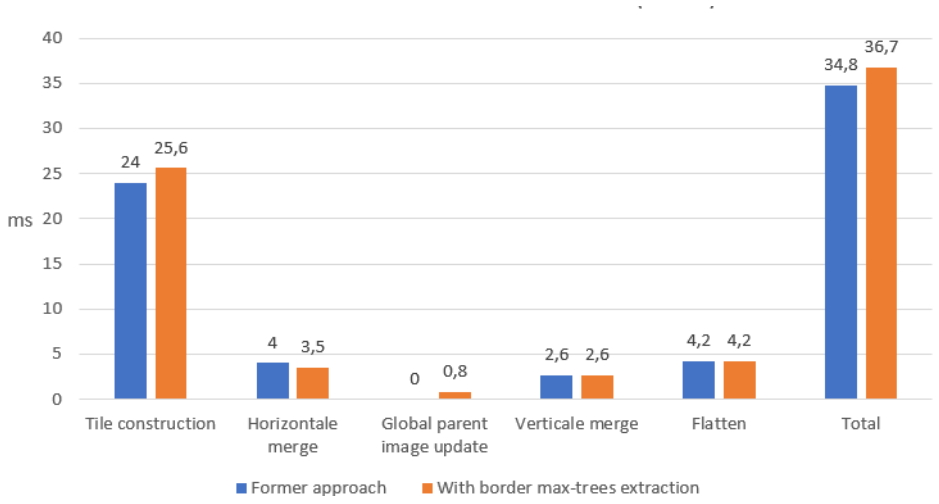


Figure 6.3: Benchmark former algorithm VS new approach

The goal of extracting the border max-tree was to speed up the horizontal merge of the tiles. On the benchmark below it can be seen the horizontal merge of the border max-trees is 0.5ms faster. The speed up is low. Moreover, with the new approach, an additional kernel must be called to commit the merged borders in the global parent image. Thus, there is not real speed up for any of the steps but there is one more step which slows down the whole max-tree construction.

6.6.3 Optimization considerations

There might be ways to optimize the approach described in this report:

- Store border max-tree of each blocks next to each other in the column borders in a deterministic order using a compact pattern. This will improve the data locality for the horizontal merge but requires another kernel.
- Use streams for each column for the horizontal merge. After a column has finished merging, do not wait other blocks to write the merged border max-trees in the global parent image.
- Hierarchical merge in order to store the border max-trees in shared memory. This approach disables concurrent merge. It has $\log(n)$ steps. Moreover for each step, the border max-tree must be extracted, stored in shared, performed the merge and write back in global memory. This might result to an overuse of global memory.

Also, if the height of the image is small enough, the border max-trees could be programmatically stored in shared memory. This ensures more memory locality and less global memory usage.

Chapter 7

Conclusion

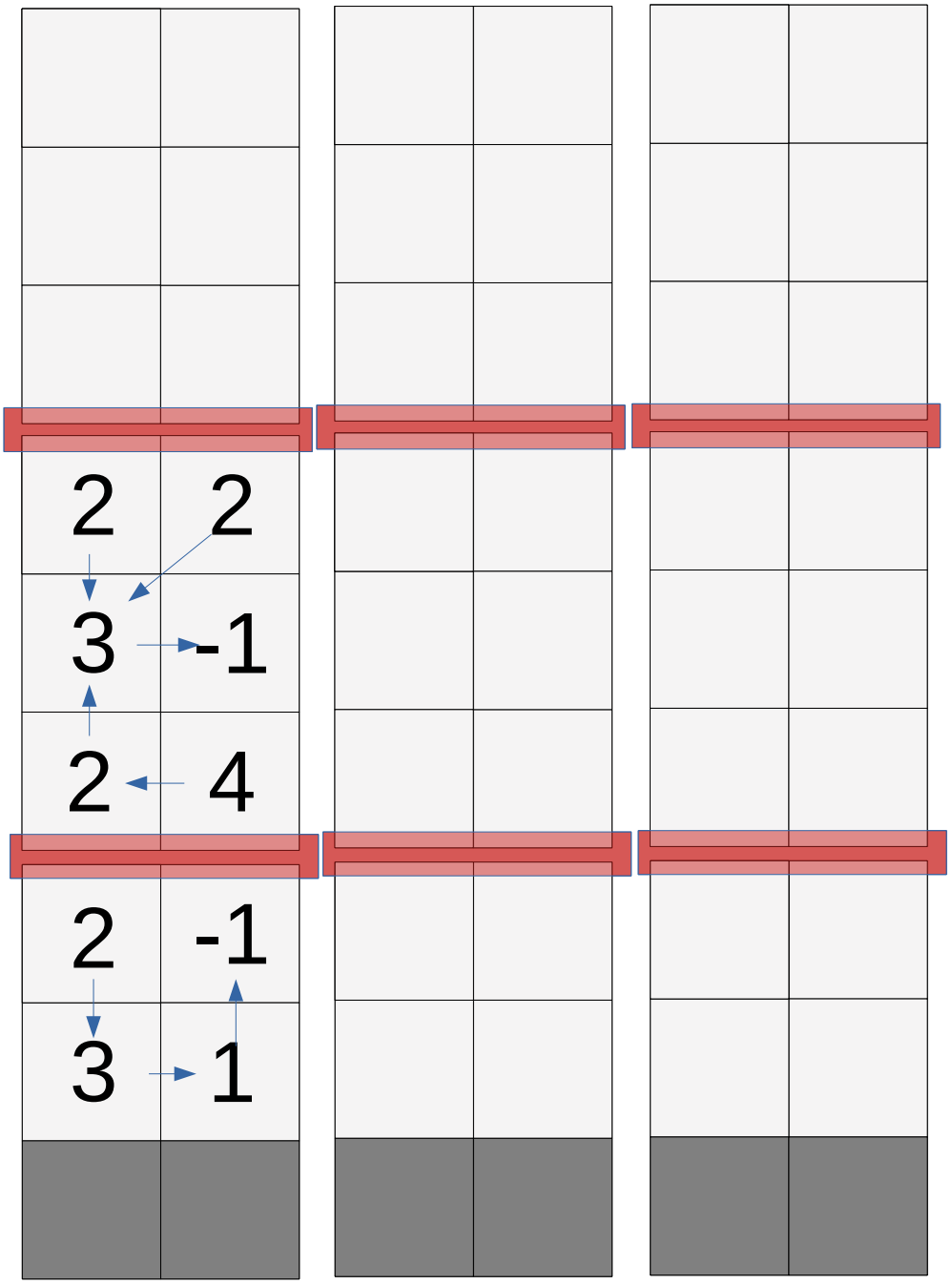
In this paper, we proposed an efficient GPU implementation of the max-tree. We highlighted the fact that writing an efficient algorithm on GPU is not trivial. One must know about the hardware constraints and, in our case, CUDA options to write an efficient program. As we saw, the main issue lies in using the memory right. Making coalesced accesses and being able to benefit from the caches is crucial.

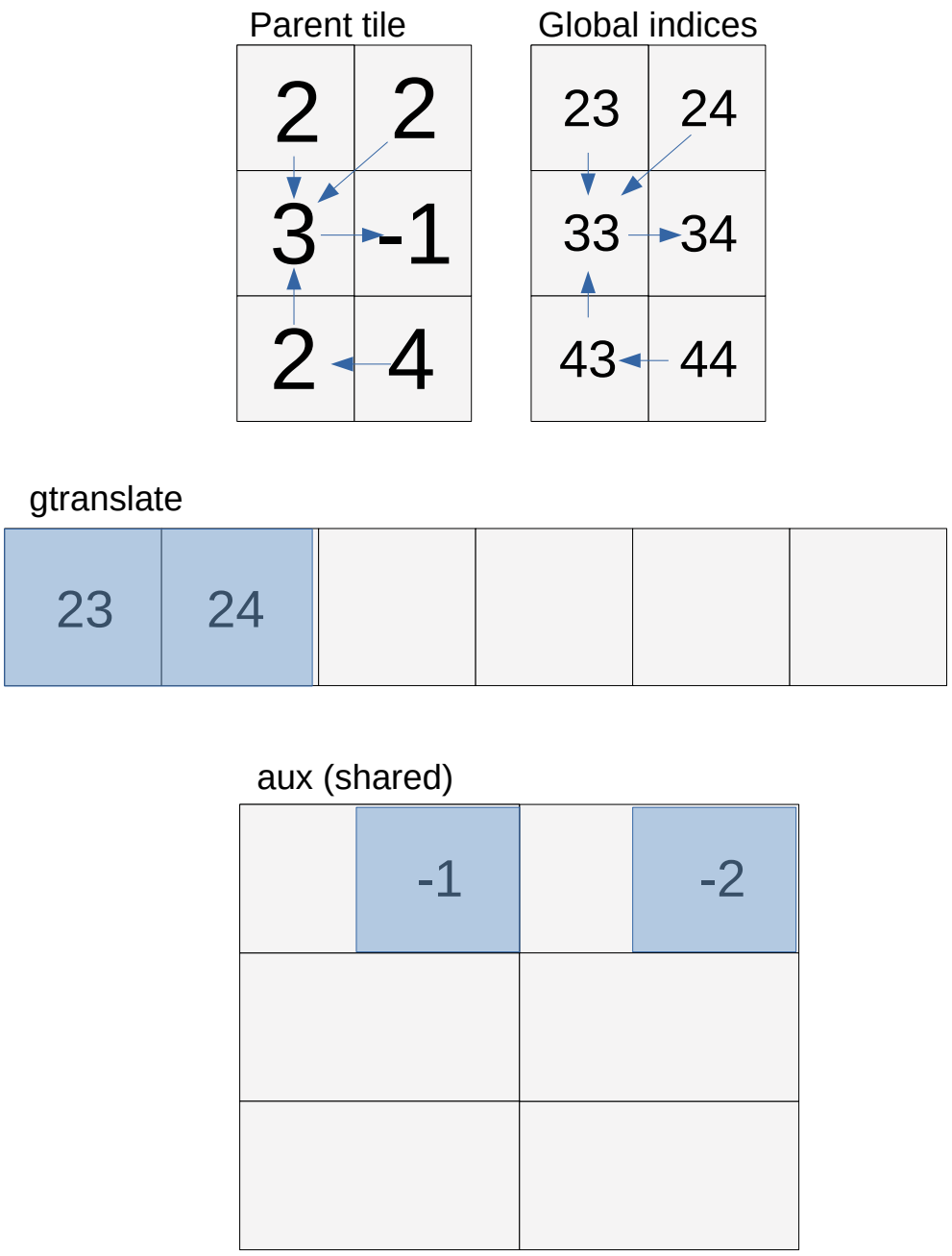
The proposed algorithm offers an x10 speed-up over the best multi-threaded CPU algorithm. This means that our real-time goal has been reached.

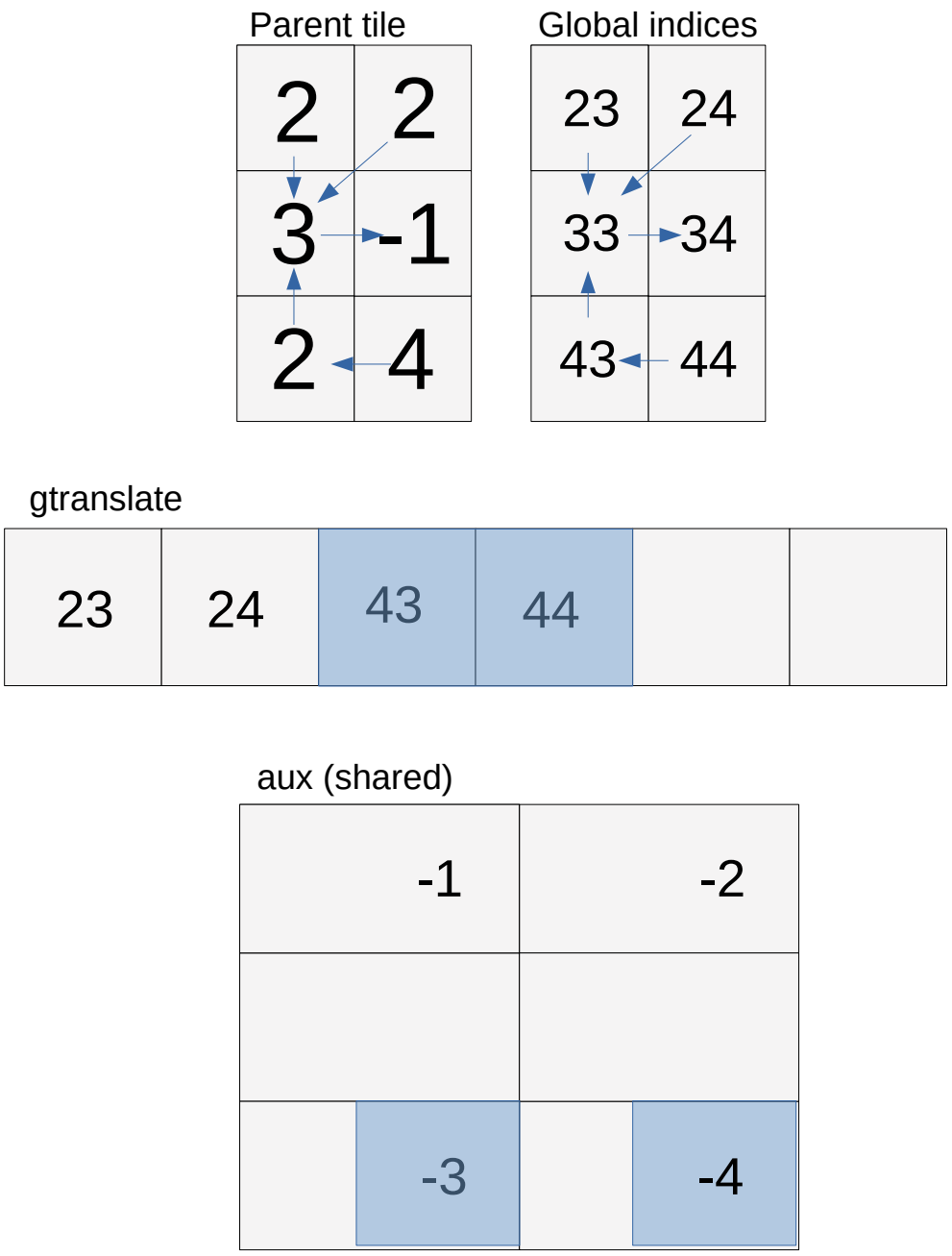
The border max-tree algorithm is a really interesting idea. It paves the path to interesting new algorithms. Our parallel implementation can rapidly extract the border max-trees. It allowed us to realize that data can, on average, be reduced to 18% of its original size. Sadly, even with column-wise border max-trees, data is still poorly cached because of its still significant size. Ultimately, it didn't speed up the max-tree global construction.

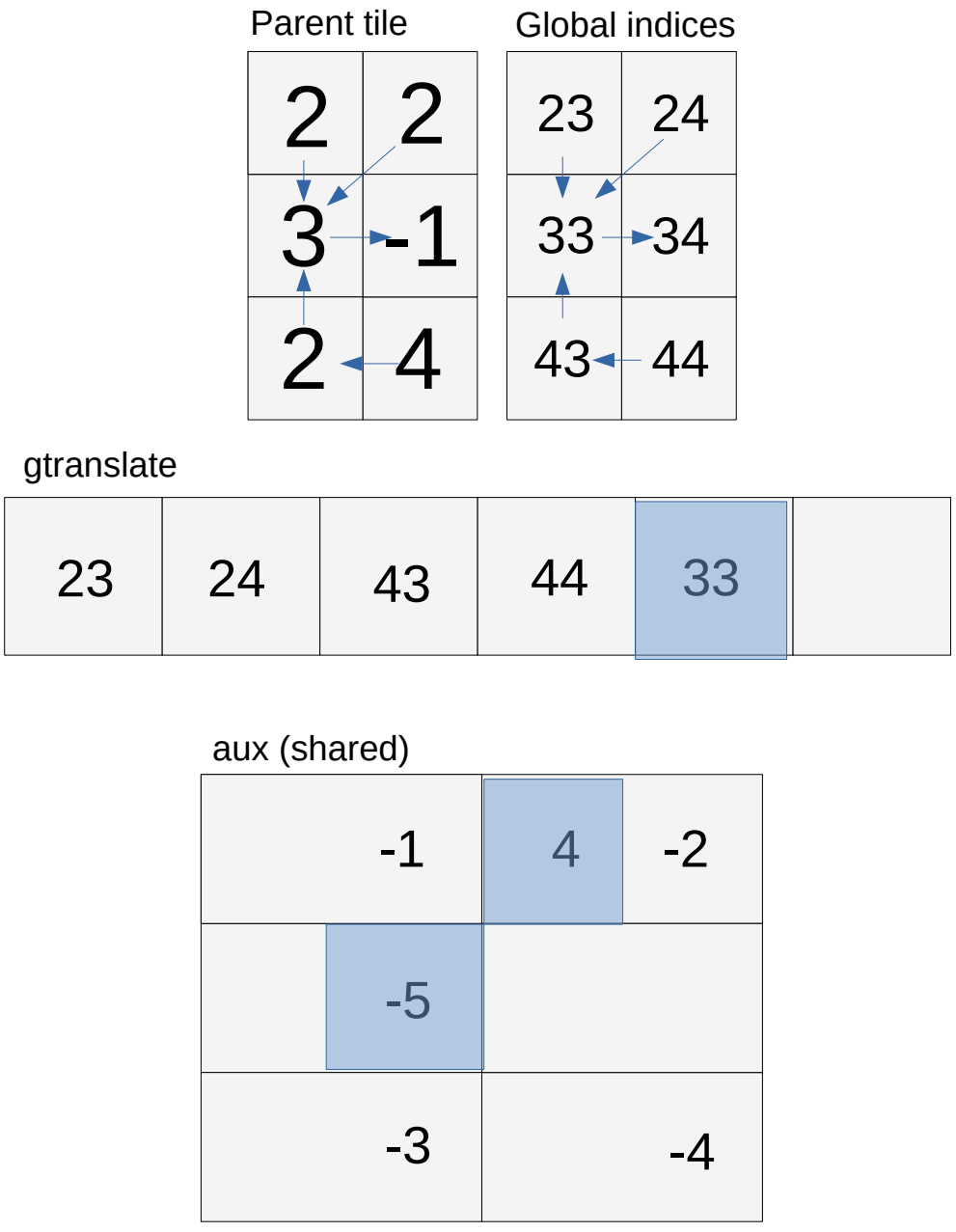
Chapter 8

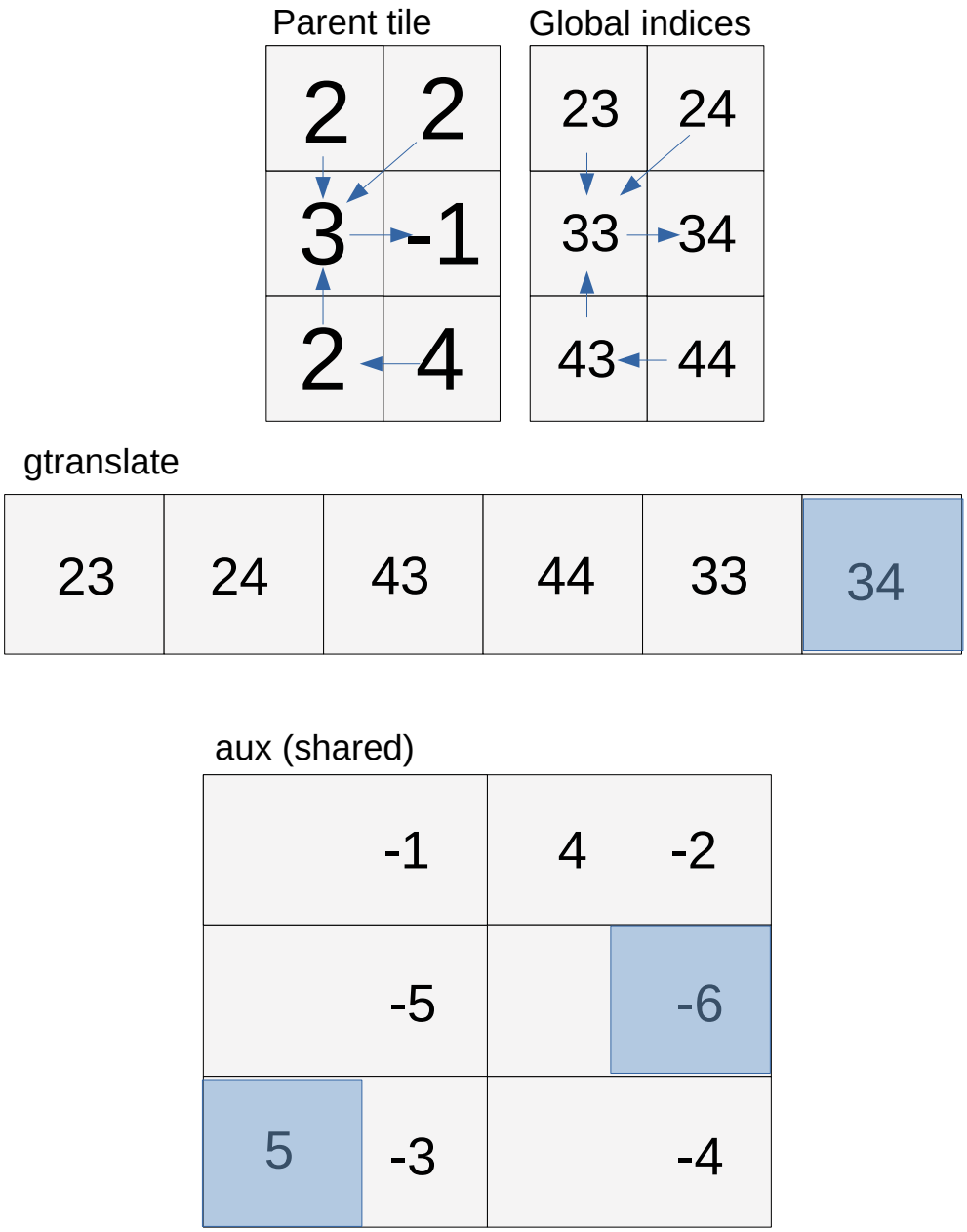
Appendix

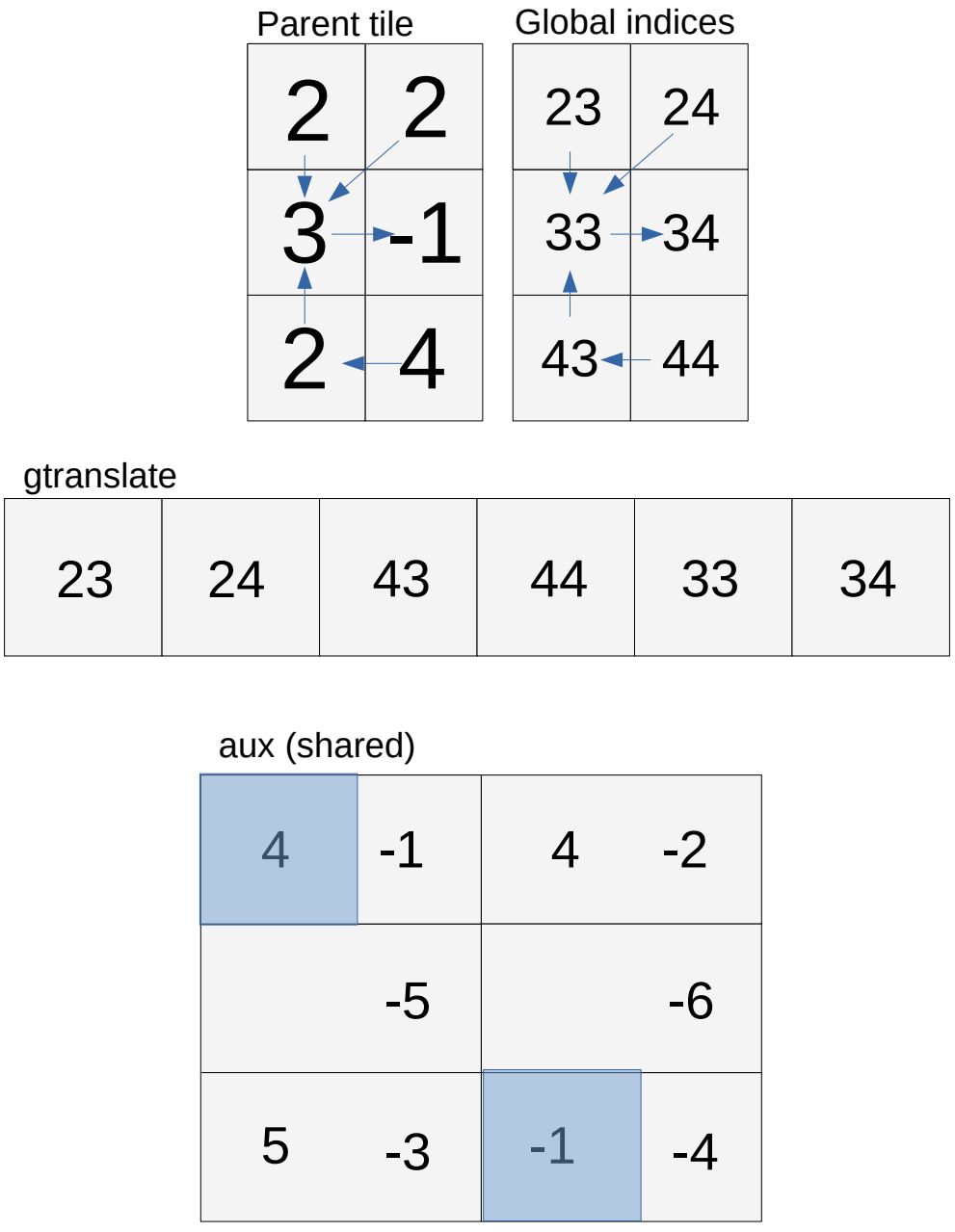

2
2
3
-1
2
4
2
-1
3
1

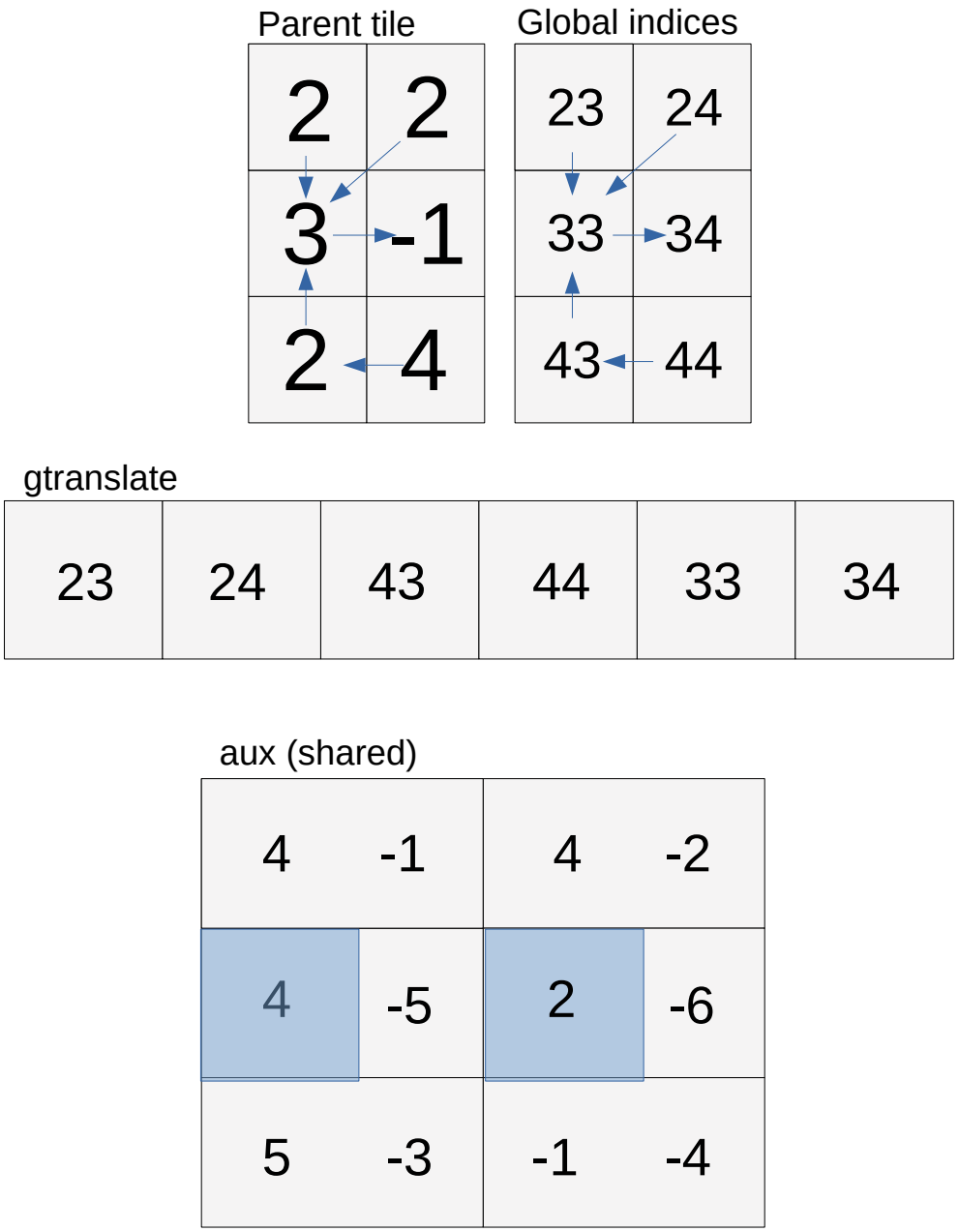

Parent tile
2
2
3
-1
2
4
Global indices
23
24
33
34
43
44
gtranslate
23
24
aux (shared)
-1
-2

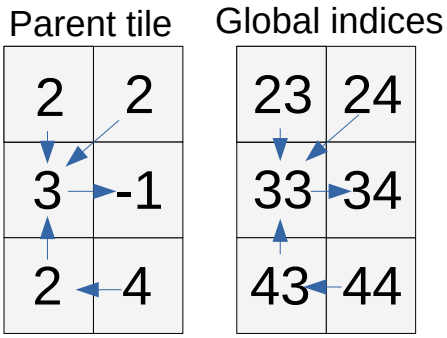

Parent tile
2
2
3
-1
2
4
Global indices
23
24
33
34
43
44
gtranslate
23
24
43
44
aux (shared)
-1
-2
-3
-4


Parent tile
Global indices
2
2
3
-1
2
4
23
24
33
34
43
44
gtranslate
23
24
43
44
33
aux (shared)
-1
4
-2
-5
-3
-4


Parent tile
Global indices
2
2
3
-1
2
4
23
24
33
34
43
44
gtranslate
23
24
43
44
33
34
aux (shared)
-1
4
-2
-5
-6
5
-3
-4


Parent tile
2
2
3
-1
2
4
Global indices
23
24
33
34
43
44
gtranslate
23
24
43
44
33
34
aux (shared)
4
-1
4
-2
-5
-6
5
-3
-1
-4


Parent tile
2
2
3
-1
2
4
Global indices
23
24
33
34
43
44
gtranslate
23
24
43
44
33
34
aux (shared)
4
-1
4
-2
4
-5
2
-6
5
-3
-1
-4



aux (shared)

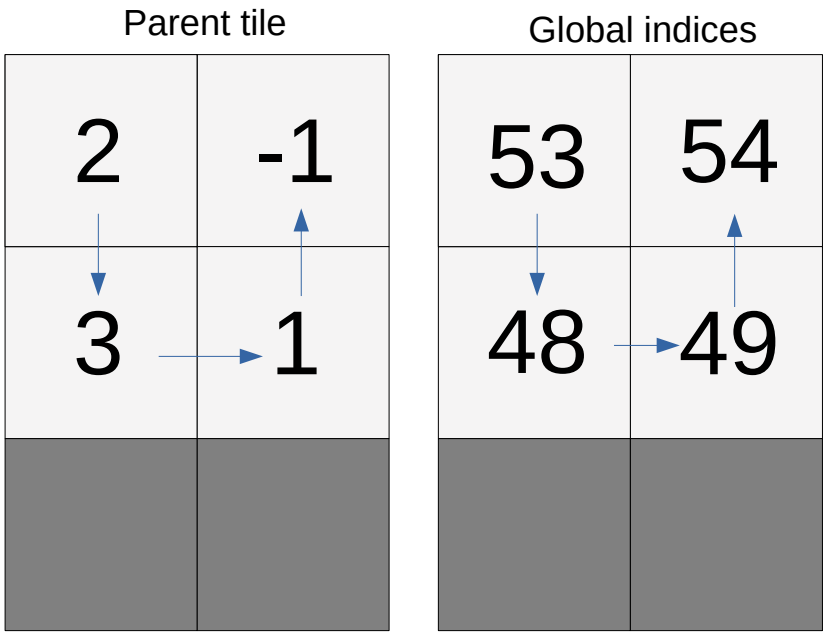
4 -1	4 -2
4 -5	2 -6
5 -3	-1 -4

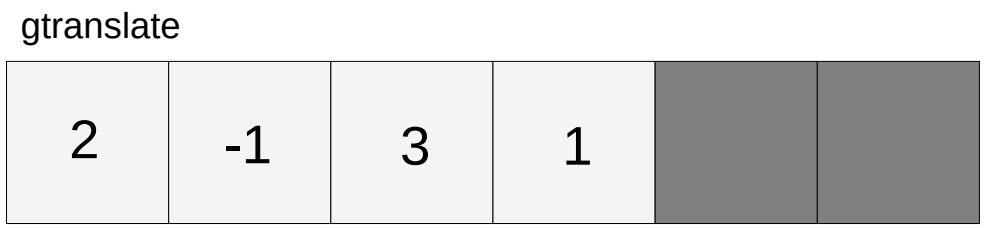
gtranslate

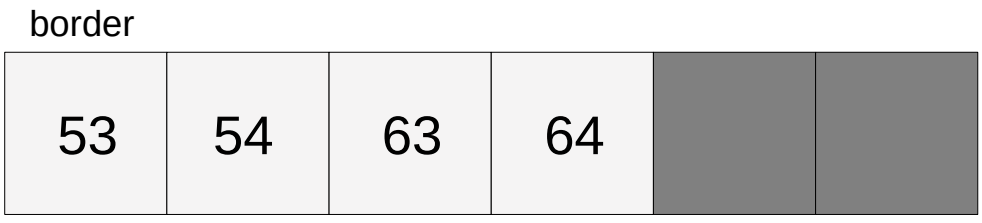
23	24	43	44	33	34

border

4	4	4	2	5	-1


Parent tile
Global indices
2
-1
3
1
53
54
48
49


gtranslate
2
-1
3
1


border
53
54
63
64

Compressed border

2	-1	3	1						
53	54	63	64						

LUT

	0

Compressed border

2	-1	3	1	8	8	8	6	9	-1
53	54	63	64	23	24	43	44	33	34

LUT

4	0

Compressed border

2	-1	3	1	8	8	8	6	9	-1
53	54	63	64	23	24	43	44	33	34

LUT

4	0

Chapter 9

Bibliography

Michael Wilkinson (2011). A fast component-tree algorithm for high dynamic-range images and second generation connectivity. *18th IEEE International Conference on Image Processing*. (page 11)

David Menotti, Laurent Najman, Arnaldo Araujo (2007). 1d component tree in linear time and space and its application to gray-level image multithresholding. *8th International Symposium on Mathematical Morphology*. (page 12)

Edwin Carlinet, Thierry Geraud (2014). A comparative review of component tree computation algorithms. *IEEE Transactions on Image Processing*. (pages 5, 11, and 12)

Edwin Carlinet, Thierry Géraud, Sébastien Crozet (2018). The tree of shapes turned into a max-tree: A simple and efficient linear algorithm. *24th IEEE International Conference on Image Processing*. (page 5)

Jan Kazemier, Georgios Ouzounis, Michael Wilkinson" (2017). Connected morphological attribute filters on distributed memory parallel machines. *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*. (pages 12 and 23)

L. Najman and M. Couprie (2006). Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*. (pages 11 and 12)

Markus Gotz, Gabriele Cavallaro, Thierry Geraud, Matthias Book, Morris Riedel (2018). Parallel computation of component trees on distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*. (page 12)

Michael Wilkinson (2004). One-hundred-and-one uses of a max-tree. (page 6)

Michael Wilkinson, Hui Gao, Wim Hesselink, Jan-Eppo Jonker, A. Meijster (2008). Concurrent computation of attribute filters on shared memory parallel machines. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. (page 11)

P. Salembier and A. Oliveras and L. Garrido (1998). Antiextensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing*. (page 11)

Robert Tarjan (1975). Efficiency of a good but not linear set union algorithm. *ACM*. (page 11)