Parallelizing Quadtree Image Decomposition with mixed approaches

Project for the Advanced Computing Architectures course at University of Trento

GABRIELE MASINA and ANDREA STEDILE, University of Trento, Italy

This report describes our work for the project required for the fulfillment of the Advanced Computing Architectures course at University of Trento, taught by professor R. Passerone in the 2021/2022 A.Y. We implemented and evaluated various approaches to the parallelization of the Quadtree image decomposition algorithm, such as CPU-based, CUDA-based and mixed approaches.

1 INTRODUCTION

A Quadtree (QT henceforth) is a tree data structure where the internal nodes have precisely four children. It is used by numerous image processing and image compression algorithms. These algorithms work by building a QT where the nodes identify unique regions of an image based on color uniformity [1]. A QT can be built using two top-down and bottom-up approaches, each with advantages and disadvantages. Our project's objective is to assess the performance differences between the two approaches. We specifically examine if running the parallelized algorithm on a GPU speeds it up compared to running it parallelly on a CPU.

The report is organized as follows. In Section 2 we provide a formulation of the recursive top-down and bottom-up approaches to the QT construction, hinting at their relative advantages and disadvantages. In Section 3 we describe some generalities of our implementations. In Section 4 we provide the implementation of the recursive QT construction approaches for execution on the CPU. In Section 5 we provide an *iterative* formulation of the bottom-up QT construction. This is an enabler for the implementation for execution on the GPU, which we provide in Section 6. In Section 7 we provide the implementation of the iterative algorithm for execution on the CPU, and in Section 8 one for mixed CPU+GPU execution. In Section 9 we benchmark the implementations and briefly discuss the results.

2 RECURSIVE FORMULATIONS

In this Section we describe the classical approaches to recursive QT construction.

2.1 Top-down approach

The top-down approach starts from the root, which is associated with the whole image. The recursive step is decided by whether or not to split a node in four children associated with four smaller regions of equal surface. The key observation is that a node associated with a non-uniformly colored region should be *split* in four nodes to be considered *differently*; if the region is uniformly colored, the node should not be split but considered *singularly*. The pseudocode of this procedure is shown in Algorithm 1. This approach is amenable to being parallelized, as the four recursive sub-problems induced by the split of a node are completely independent. Intuitively, this approach is advantageous when an image is uniformly colored, as the recursion stops earlier with respect to the bottom up approach. The disadvantage is that the recursive step requires to redundantly recompute the mean and the standard deviation of the pixels in the smaller region.

Authors' address: Gabriele Masina, gabriele.masina@studenti.unitn.it; Andrea Stedile, andrea.stedile@studenti.unitn.it, University of Trento, Trento, Italy.

Algorithm 1: top_down_build(Image *img*)

```
std ← standard deviation of the img pixels
if std ≤ threshold then
    mean ← mean color of the img pixels
    return Leaf(mean)
else
    tl ← top_down_build(top-left subquadrant of img)
    tr ← top_down_build(top-right subquadrant of img)
    br ← top_down_build(bottom-right subquadrant of img)
    bl ← top_down_build(bottom-left subquadrant of img)
    return Fork(tl, tr, br, bl)
```

2.2 Bottom-up approach

The bottom-up approach starts from the single leaves, which are associated with the single image pixels. The recursive step is decided by whether or not to *merge* four nodes in a parent node associated with the union of the four regions. The key observation is that four nodes associated with similarly colored regions should be merged in a node to be considered *singularly*; if the regions are differently colored, the nodes should not be split but considered *separately*. The pseudocode of this procedure is shown in Algorithm 2. The advantage of this approach is that the mean and the standard deviation of the pixels in the region associated with the parent node can be computed in constant time. This is because the mean and the standard deviation of the pixels in the regions associated with the four nodes are already available in the nodes themselves and do not need be recomputed.

Algorithm 2: bottom_up_build(Image *img*)

```
if the image contains a single pixel then
    mean ← color of the pixel
    return Leaf(mean)
else
    tl ← bottom_up_build(top-left subquadrant of img)
    tr ← bottom_up_build(top-right subquadrant of img)
    br ← bottom_up_build(bottom-right subquadrant of img)
    bl ← bottom_up_build(bottom-left subquadrant of img)
    std ← compute standard deviation from tl.std, tr.std, br.std, bl.std
    mean ← compute mean from tl.mean, tr.mean, br.mean, bl.mean
    if std ≤ threshold then
        return Leaf(mean)
    else
        return Fork(tl, tr, br, bl)
```

3 GENERALITIES

The QT construction algorithms accept square images only; therefore, we call regions *subquadrants*. For practical reasons that will be apparent in Section 6, images must be of size $M = N \times N$, where N is a power of two (and M a power of four). Rectangular images are padded into square images.

3.1 Programming language and libraries

Our project is implemented in the C++17 programming language and CUDA. We use the notorious stb[1] open-source library to load images in memory. We use the notorious Eigen[2] open-source linear algebra library to efficiently store and reduce arrays in memory. We use the CUDA toolkit 11 in the implementation for GPU execution.

3.2 Input and output files

The stb libary allows to read a variety of image formats, such as JPEG, PNG and bitmap formats, in an RGB array. We allow the possibility to export the intermediate steps of the QT construction to visualize the decomposition process. Some examples are shown in Figures 5 and 6.

3.3 Pre-processing and post-processing

The fundamental pre-processing step the algorithms perform is to rearrange the pixel array in such a way that pixels in a unique subquadrant of the image become contiguous inside the array, exhibiting spatial locality. This not only makes implementations simpler, but also speeds up the computation of the color uniformity of a subquadrant due to the spatial locality of the pixels it contains. This process is shown in Figure 1.

The pixel array is also reorganized in a SoA (Structure of Arrays) layout, such that the structure comprises three arrays: one for the red component of the pixels, one for the green component and one for the blue component. This is particularly helpful for the top-down version of the algorithm, in which the mean and the standard deviation of the pixels in a subquadrant can be computed independently for each of the three color components.

3.4 Computing color uniformity

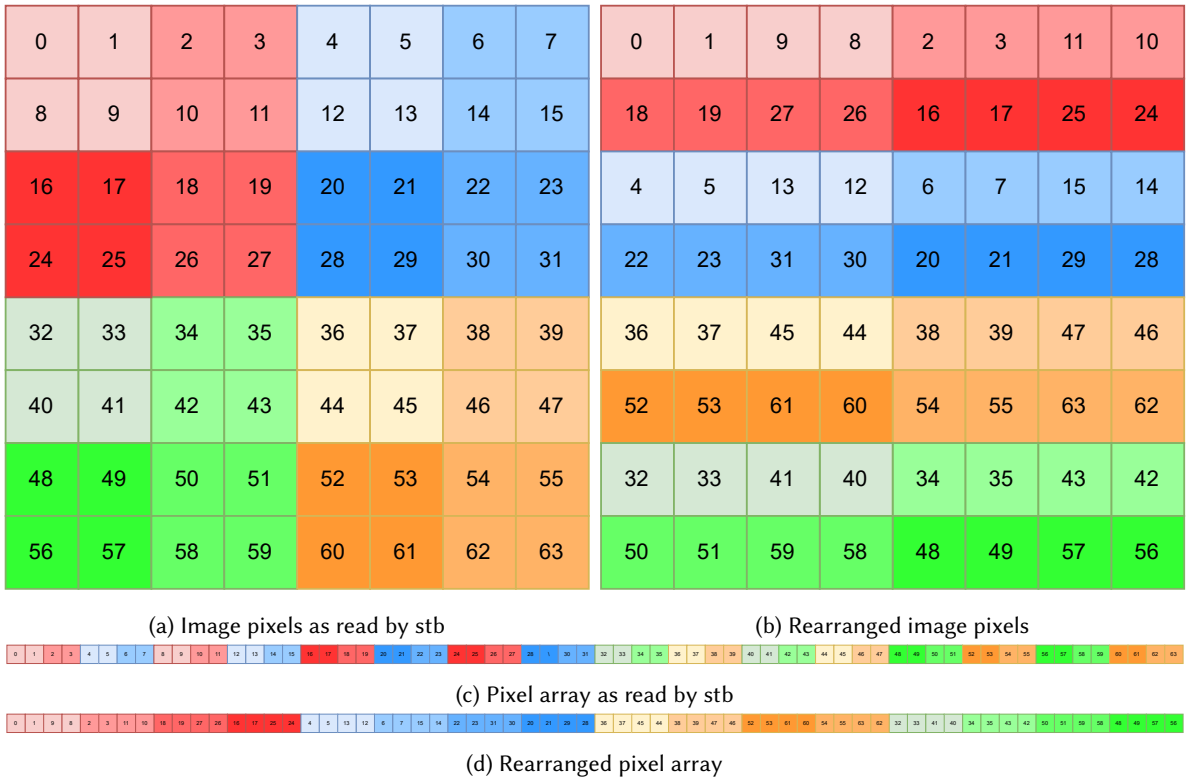
We provide two implementations for the computation of the color uniformity of the pixels in a subquadrant. Algorithm 3 and 4 summarize the first one: it is applied to an SoA data layout described in Section 3.3, and exploits Eigen's capability of performing reductions[3].

The second implementation is to be applied to an AoS data layout (each structure containing the red, green and blue components of a pixel). Trivially, the implementation strides through the array to sum the individual pixel components.

Algorithm 3: compute_mean(SUBQUADRANT *subq*)

1 $left \leftarrow$ Index in the pixel array where *subq* starts
2 $length \leftarrow$ Number of pixels in *subq*
3 $r \leftarrow soa.r[left \dots left + length]$.mean()
4 $g \leftarrow soa.g[left \dots left + length]$.mean()
5 $b \leftarrow soa.b[left \dots left + length]$.mean()
6 **return** $\langle r, g, b \rangle$

[1] https://github.com/nothings/stb
[2] https://eigen.tuxfamily.org/
[3] https://eigen.tuxfamily.org/dox/group__TutorialReductionsVisitorsBroadcasting.html



(a) Image pixels as read by stb

(b) Rearranged image pixels

(c) Pixel array as read by stb

(d) Rearranged pixel array

Fig. 1. Rearrangement of the pixel array. 1a and 1c depict the pixel array as read by stb. 1b and 1d depict the pixel array after the rearrangement. After the rearrangement, pixels belonging to the same subquadrant exhibit spatial locality.

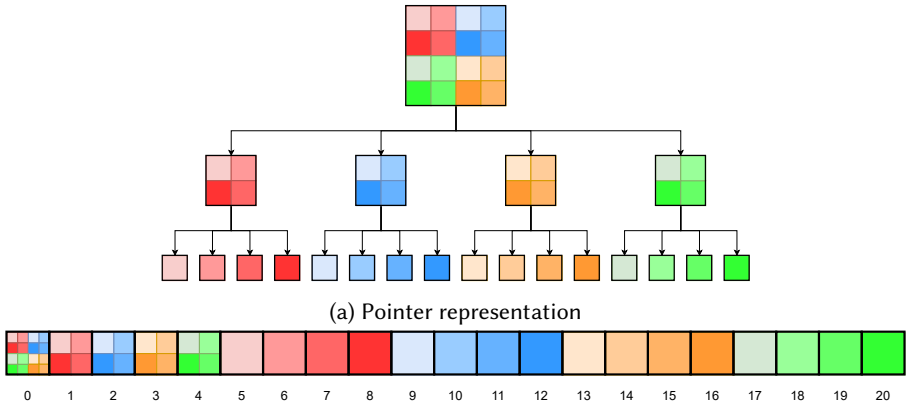
Algorithm 4: compute_std(SUBQUADRANT *subq*)

1 $\langle r_m, g_m, b_m \rangle \leftarrow$ compute_mean(*subq*)
2 *left* $\leftarrow$ Index in the pixel array where *subq* starts
3 *length* $\leftarrow$ Number of pixels in *subq*
4 $r_s \leftarrow soa.r[left \dots left + length]$.sq().mean()
5 $g_s \leftarrow soa.g[left \dots left + length]$.sq().mean()
6 $b_s \leftarrow soa.b[left \dots left + length]$.sq().mean()
7 **return** $\langle \sqrt{r_s - r_m^2}, \sqrt{g_s - g_m^2}, \sqrt{b_s - b_m^2} \rangle$

4 RECURSIVE CPU IMPLEMENTATIONS

Implementing the two recursive approaches to the QT construction algorithm is straightforward. The `Quadtree` class represents a QT node. It contains the extents of the unique subquadrant of the image it is associated with, the mean and standard deviation components of the pixels in the subquadrant and pointers to children (if any).

The top-down approach is summarized by Algorithm 1. As we described in Section 1, this approach computes the color uniformity of a subquadrant at each split. Consequently, to make this operation efficient, the computation makes use of the SoA data layout to store the image pixels.



(a) Pointer representation

(b) Array representation

Fig. 2. Different QT representations. Figure 2a shows the typical tree representation. Figure 2b show the array representation.

If a node is split, the algorithm is applied recursively to the four children. Each child node identifies an independent sub-problem and can be solved independently (and thus, parallelly). We identify two approaches to parallelization:

(1) Using a thread pool: at each node split, assign a recursive sub-problem to a new thread of execution if the thread pool is not exhausted (when no more concurrency can be effectively exploited).
(2) At each node split, assign a recursive sub-problem to a new thread if the depth reached by the recursion is less than a certain threshold.

We implemented approach 1 with the C++ concurrency library[4]; in particular, by using a combination of the `async` and `deferred` policies (this is nicely discussed at [5]). We implemented approach 2 by passing the recursion threshold as a control parameter to the `top_down_build` function. We set the threshold to 1, so that a maximum of 16 threads execute concurrently.

Note that the QT image decomposition entails a load balancing problem: highly non-uniformly colored regions require many recursive node splits and long computational times, whereas uniformly colored regions are resolved more quickly. Initially, approach 1 seemed better to us: once a thread solves a sub-problem (for example, one of a uniformly colored subquadrant), it can be assigned to solve another (possibly of a non-uniformly colored subquadrant). However, our empirical evaluations show that approach 1 is consistently slower in practice, therefore we use approach 2 in the implementations .

The bottom-up approach is summarized by Algorithm 2. As we described in Section 2, when four nodes are merged, the color uniformity of the resulting node can be computed in constant time by using the mean and standard deviation values that are available in the four nodes themselves. The unfolding of the recursions is such that the actual construction starts from the leaves, which are associated with the single pixels. It follows that, to compute the values efficiently at the leaf nodes, it is best to store the pixels using an AoS. The approaches to parallelization are analogous to the top-down case.

5 ITERATIVE FORMULATION

In this Section we provide an iterative formulation of the bottom-up QT construction, which is an enabler for the implementation for execution on the GPU, which we provide in Section 6. We first describe the required data structure, then an iterative procedure which constructs it.

5.1 Array representation of a QT

Given a square image of $N = M \times M$ pixels, we want to produce a *complete* QT. A complete QT is a QT in which each level i of the tree contains contains 4^i nodes. The height of a complete QT is $h = \log_4 N$; thus, a complete QT contains a total of $\sum_{i=0}^{h} 4^i = (4^{h+1} - 1)/3$ nodes. A QT *array* is an array representation of a complete QT with the following properties:

(1) Each element of the array represents a node of the complete QT. As such, it contains the type (either Fork or Leaf) and the mean and standard deviation values of the pixels in the subquadrant uniquely associated with it.
(2) The root of the complete QT is stored in position 0.
(3) The children of a node stored in position i are stored in positions $4i + 1$, $4i + 2$, $4i + 3$ and $4i + 4$.

As consequences:

(4) The parent of a node stored in position i is stored in position $\lfloor \frac{i-1}{4} \rfloor$.
(5) The 4^i nodes of level i in the complete QT are stored in positions $[(4^{i-1} - 1)/3, \ldots, (4^{i-1} - 1)/3 + 4^i - 1]$.

Figure 2 depicts the array representation of an image with $N = 4 \times 4$ pixels; it is easy to see that it verifies all the properties above.

5.2 QT array reduction

The reduction of a QT array first needs to initialize the leaf nodes. The initialization involves setting the mean value and leaf type for each node. Since a leaf is associated with a unique pixel, its mean value is the color of the pixel itself. It is easy to see that the initialization is completely parallelizable.

The reduction itself produces the intermediate nodes in the QT array and mimics the bottom-up construction of the complete QT. The intuition of a reduction step is as follows. At level i, a complete QT contains 4^i nodes, which are stored in the array starting from position $(4^h - 1)/3$ (due to property 5). A reduction step produces the 4^{i-1} nodes of level $i - 1$, to be stored in the array starting from position $(4^{i-1} - i)/3$. The reduction starts at the leaf level, thus producing the second-to-last level nodes, and is repeated for $h - 1$ times, producing the nodes of all the remaining levels.

Figure 3 depicts the QT array reduction of an image with $N = 4 \times 4$ pixels.

We reiterate that the QT array corresponds to a complete QT; therefore, it occupies a significant amount of memory. Note that the time to reduce a QT array solely depends on the number of pixels in the image and not on the color uniformity.

6 ITERATIVE GPU IMPLEMENTATION

Inspired by the bottom-up CPU implementation, we provide an implementation for a general-purpose GPU. The first challenge is that the QT representation must not use pointers: pointers in the device memory cannot be copied back to the host memory. The second challenge is that the algorithm itself cannot be recursive: CUDA threads have a limited stack size. The QT array and the QT array reduction schema we described in Section 5 is appropriate as it meets these requirements.

[4] https://en.cppreference.com/w/cpp/thread/async
[5] https://stackoverflow.com/questions/9359981/stdasync-stdlaunchasync-stdlaunchdeferred

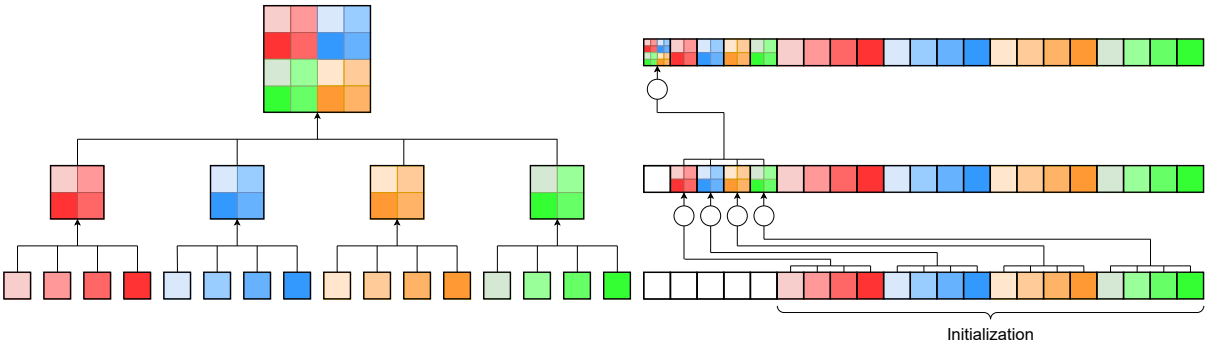


Fig. 3. Analogy between the unfolding of bottom-up QT construction and the QT array reduction.

Remember that the recursive approaches to QT construction enjoy the property that distinct QT branches can be constructed independently and parallelly. We now discuss the behavior of the CUDA kernel implementing the QT array reduction. A block performs a sequence of reductions corresponding to the unfolding of the bottom-up recursion of the construction of a specific QT branch. Each thread of a block produces a node by reducing four nodes. (More precisely, in the first iterations of the reduction, a thread can produce/reduce an amount of nodes being a power of four). Because in subsequent iterations the amount of nodes to reduce is divided by a factor of four, some threads eventually become idle. The reduction stops when the number of nodes to reduce amounts to the number of blocks, and must be completed with some other process. We describe this process in Section 8.

What remains to be discussed is the choice of the grid size and block size (i.e., the number of blocks and number of threads per block) in a kernel launch. Remember that in Section 3 we assumed the input be of $N = M \times M$ pixels, M being a power of two (and N a power of four). The assignment schema constraints the number of blocks n_b to be a power of four and sets the number of threads to be $n_t = (\frac{N}{n_b})/4^i$. (We allow i to be either 1, 4, or 16).

7 ITERATIVE CPU IMPLEMENTATION

In this Section we provide an implementation of the QT array reduction of Section 5 that fully executes on the CPU.

Given a square image of $N = M \times M$ pixels, the QT array contains $(4^{h+1} - 1)/3$ nodes. The parallel portions of the code are implemented with OpenMP threads (we set the number of threads to 8). The initialization of the leaf nodes is trivial: each thread initializes a subset of the 4^h leaves in parallel. The reduction phase is implemented by a loop of $h - 1$ iterations where the threads reduce equal shares of the nodes.

This CPU implementation intuitively corresponds to an execution of the GPU implementation of Section 6 with one block.

8 ITERATIVE MIXED CPU+GPU IMPLEMENTATION

The GPU implementation of the QT array reduction of Section 6 stops at the level where the number of nodes to reduce amounts to the number of blocks. The obvious way to complete the reduction is to copy the QT array back to the host and apply the implementation of Section 7 starting from the level at which the reduction stopped.

9 EVALUATIONS OF THE IMPLEMENTATIONS

In this Section we describe the testing methodology and discuss the results. Our project uses the CMake build system. We instruct CMake to use the GCC 9.1 compiler and the CUDA 11.1 Toolkit. We also instruct CMake to

Table 1. Evaluation of the mixed GPU+CPU implementation of the QT array reduction algorithm

Image	Resolution (px)	# Blocks	# Threads	# leaves per thread	Mean time (ms)
nature	1024x1024	256	256	16	147.8
		4096	16	16	151.2
		4096	256	1	158.0
		16384	16	4	160.0
		16384	4	16	160.0
		16384	64	1	165.4
		4096	64	4	166.6
		1024	64	16	171.4
		1024	256	4	173.8
earth	4096x4096	4096	256	16	453.4
		16384	256	4	543.8
		16384	64	16	568.0
pino	8192x8192	16384	256	16	1448.4

compile in Release mode (this emits the '-O3' optimization flag) and to target the Volta GPU microarchitecture. We run the implementations on a host of the University of Trento's HPC cluster, with an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, 330 GB of RAM and an NVIDIA®V100. We test the implementations against three images with different resolutions.

Each run has been repeated 10 times to make sure results be consistent over time. Given the kernel launch constraints we described in Section 6, 512 does not constitute a valid block size. Moreover, when selecting 1024 threads per block, the error message "Got error too many resources requested for launch" is reported by CUDA. Given that the maximum block size for this GPU microarchitecture is 1024, it follows we can only specify a block size of 256 or less. Our tests are aimed at identifying the best grid size and block size for kernel launches. Table 1 summarizes the obtained results. The execution time comprises the time required to allocate memory on the device, copy the SoA on the device, initialize the QT array and reduce it, copy the QT array back to the host and complete the reduction. Results suggest that the best performance is obtained when the number of blocks is minimized. This can be intuitively justified by the fact that the less blocks are used, the more reduction steps the algorithm performs on the device.

Image 4 shows the result of the multiple CPU implementations of the QT construction algorithms and a comparison with the GPU implementation, for images of sizes 1024×1024, 4096×4096, 8192×8192. Unsurprisingly, the data we gathered suggest that the use of a GPU is beneficial when the image size is very large. In fact, for small images, the transfer time of the AoS on the GPU is non-negligible. However, we observe for the 8192×8192 image that the iterative CPU parallel implementation's execution time is overall faster than the GPU implementation's. Table 2 compares the time taken by the single operations in the implementations. We can observe that a non-negligible shares of the time are taken by the communication between the host and the device and the memory allocations, but that the actual time spent on constructing the QT is better for the GPU version.

As a note side, we are surprised by the fact that the recursive bottom-up approach is drastically outperformed by both the the top-down recursive and the bottom-up iterative algorithms. We are not sure why this is the case; this issue can be discussed orally with the professor.

Step	GPU time(ms)	CPU time (ms)
allocate device	107.2	
copy to device	38.0	
construct on device	82.3	
allocate on host	837.0	839.5
copy to host	383.9	
construct on host	0.0	314.2
total	**1448.4**	**1153.7**

Table 2. Comparison of CPU (iter) and GPU performance on a 8192×8192 image, divided by phases.

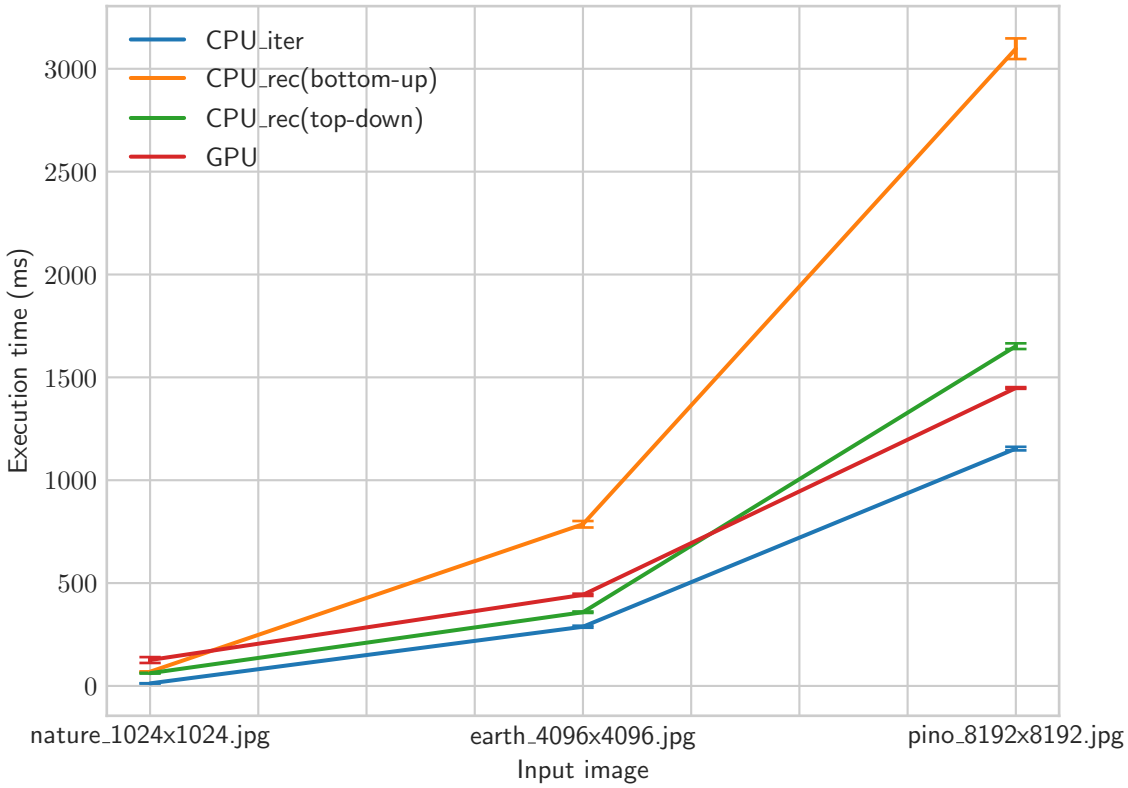


Fig. 4. Comparison of CPU and GPU parallelization approaches

10 FIGURES

REFERENCES

[1] Eli Shusterman and Meir Feder. 1994. Image compression via improved quadtree decomposition algorithms. *IEEE Transactions on Image Processing* 3, 2 (1994), 207–215.

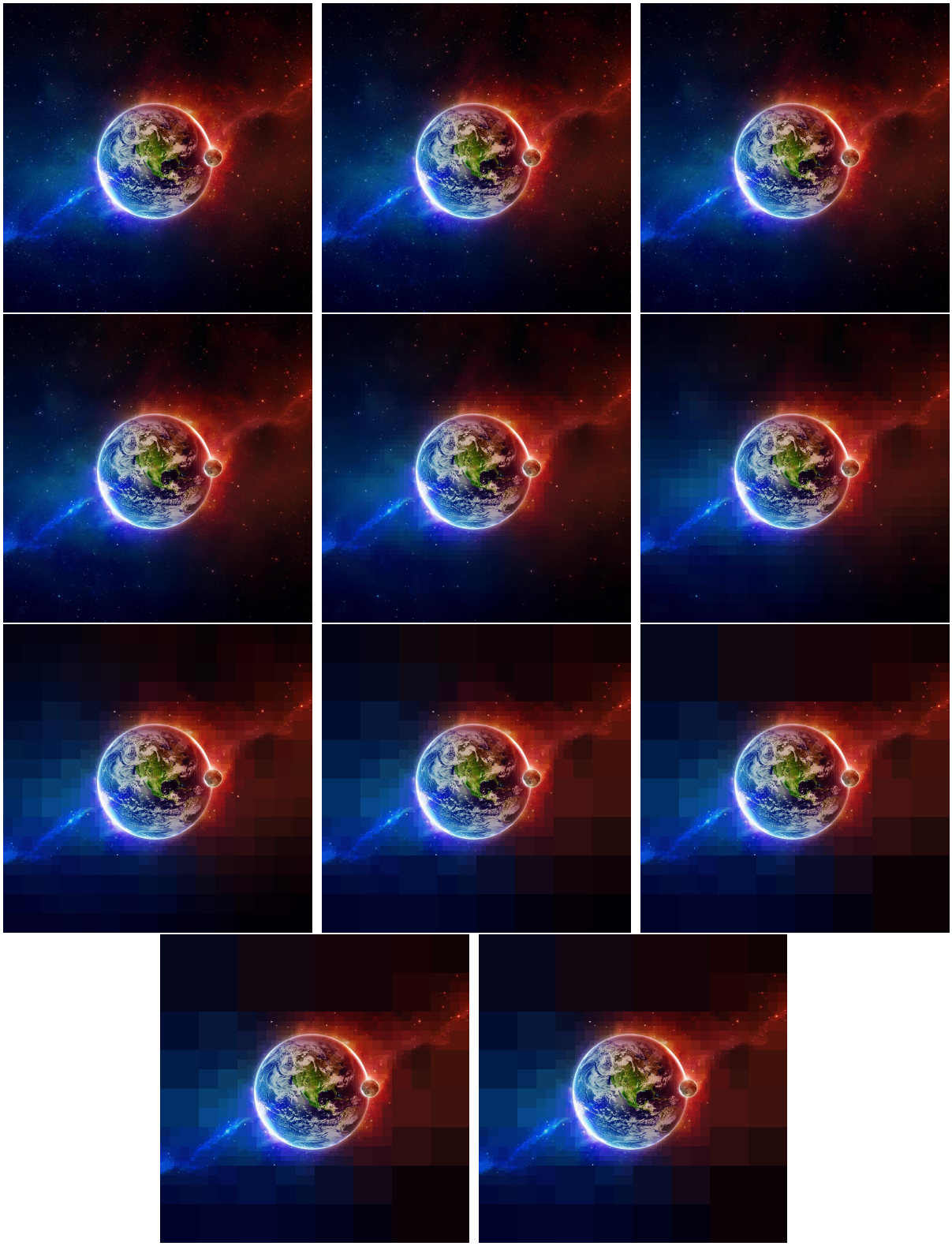


Fig. 5. Bottom-up quadtree decomposition.

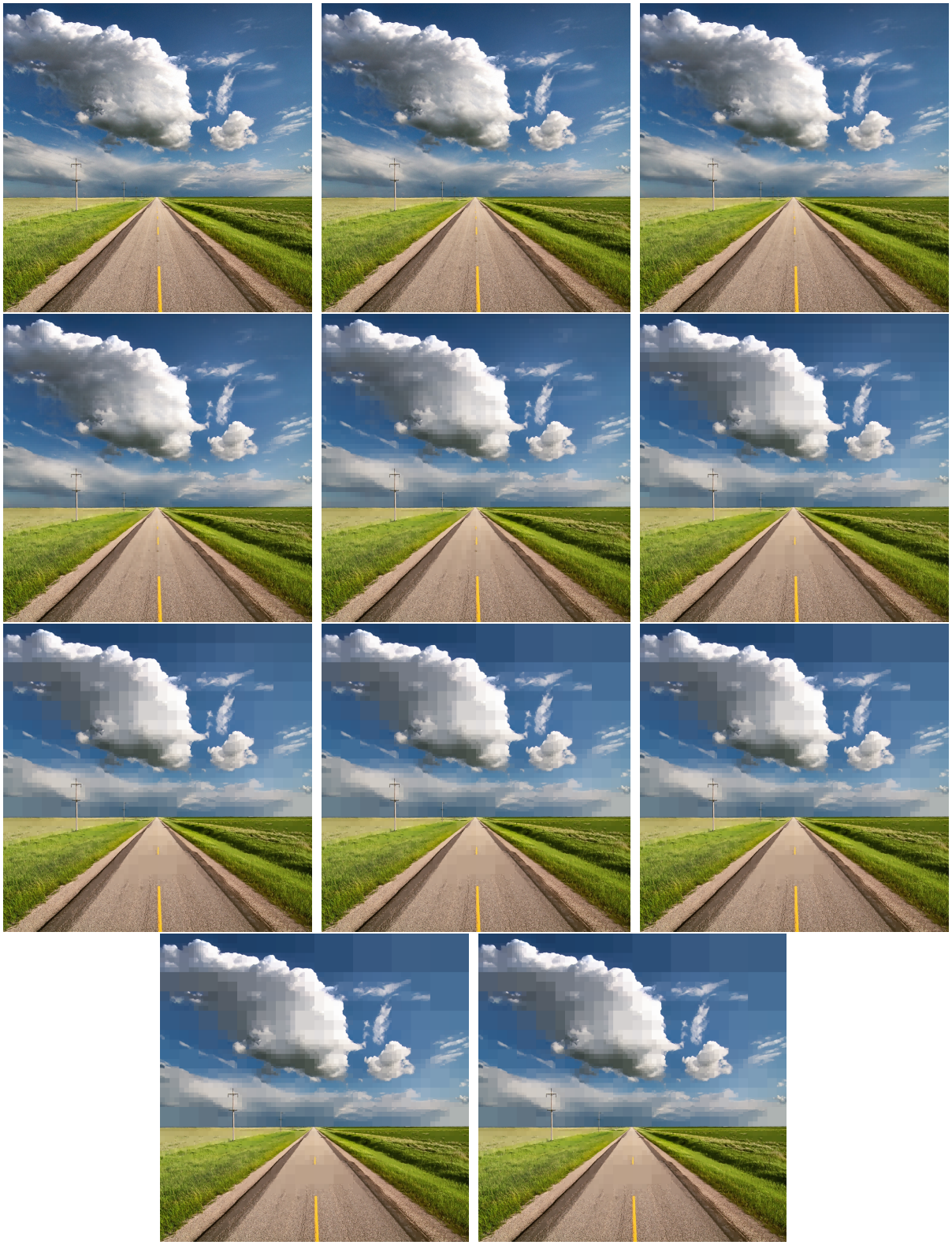


Fig. 6. Bottom-up quadtree decomposition.