

RealTime QuadTree Analysis using HistoPyramids

Gernot Ziegler*^a, Rouslan Dimitrov^b, Christian Theobalt^a, Hans-Peter Seidel^a

^aMax-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, D-66123 Saarbrücken, Germany.

^bInternational University Bremen GmbH, Postfach 750561, D-28725 Bremen, Germany.

ABSTRACT

Region quadtrees are convenient tools for hierarchical image analysis. Like the related Haar wavelets, they are simple to generate within a fixed calculation time. The clustering at each resolution level requires only local data, yet they deliver intuitive classification results. Although the region quadtree partitioning is very rigid, it can be rapidly computed from arbitrary imagery. This research article demonstrates how graphics hardware can be utilized to build region quadtrees at unprecedented speeds. To achieve this, a data-structure called *HistoPyramid* registers the number of desired image features in a pyramidal 2D array. Then, this HistoPyramid is used as an implicit indexing data structure through quadtree traversal, creating lists of the registered image features directly in GPU memory, and virtually eliminating bus transfers between CPU and GPU. With this novel concept, quadtrees can be applied in real-time video processing on standard PC hardware. A multitude of applications in image and video processing arises, since region quadtree analysis becomes a light-weight preprocessing step for feature clustering in vision tasks, motion vector analysis, PDE calculations, or data compression. In a sidenote, we outline how this algorithm can be applied to 3D volume data, effectively generating region octrees purely on graphics hardware.

Keywords: GPU, Image Processing, region quadtree, histopyramids, graphics hardware, real-time, hierarchical, octree

1. INTRODUCTION

As graphics hardware has become more programmable, new applications like general matrix calculation^{2,7}, image convolution¹² or physics processing⁴ have become feasible. But ever since the first of these applications had been implemented, it became obvious that the stream processing nature of graphics hardware, which gives it tremendous processing power, also requires considerable rethinking of data structures and algorithms. Many non-local calculations, virtually trivial on single-thread systems, like counting non-zero pixels in a 2D image, become hard to solve on the GPU, since its inherently parallel nature can only be utilized if the output of several parallel units is combined.

The thought approach of data pyramids solved the problem of global sums: A so-called *reduction operator*¹ summarizes the content of four cells at once, and writes the result into a 2D image of half the input size. This is repeated until only one cell prevails. This way, all parallel units in the GPU can contribute equally to the final calculation of the result.

We built on this concept¹⁵ and introduced the *Histogram Pyramid* (short: HistoPyramid), a special form of data pyramid which uses the reduction operator to count the number of active cells in a 2D image. “Active cells” are determined by applying a user-provided *discrimination function* to all cells in the image.

But instead of discarding the intermediate pyramid levels in the HistoPyramid, we retain it to solve a related algorithmic problem, the generation of a *list* of active cells in a 2D image (a *point list*) on stream processing hardware. On such architectures, it is *not allowed to forward data* from one output element to the next one. Therefore, the trivial CPU solution, which traverses a 2D image sequentially in order to count all occupied pixels, and writes down cell coordinates as they are encountered, can not be applied. The original HistoPyramid algorithm¹⁵ solved this problem.

In this article, we explain how a combination of the above point list generator and a *hierarchical feature combiner* is able to generate quadtrees and octrees on stream processors as the GPU.

2. RELATED WORK

Image pyramids have been used in Binary Tree Predictive Coding^{3,11}. For example, a quad tree leaf can signal if all of its descendants are identical, and hence skip the transmission of its descendants. Our algorithm uses this idea twice: First, it is used in QuadPyramid construction, to combine features as long as they are identical, creating an as-large quad

leaf as possible. Second, it is used to skip empty regions during the top-down QuadPyramid traversal, which builds up the quad list, and contributes thus to parallelization.

The build process for the mentioned QuadPyramid is adopting the well-known parallel "reduction operation". It is applied in custom mipmapping¹, and processes n^2 elements in $\log_2(n)$ passes. Our operator switches modes during the process; first, it acts as a feature combiner – then, in higher levels, when features can not be combined anymore, it starts to sum them, just like in the original point list algorithm¹⁵.

Pyramidal image processing has been used to generate new kinds of high-speed image filters and real-time texture completion¹³. While the used push-pull principle makes use of HistoPyramid-like data structures, it never explicitly generates a list, but instead conducts its operations by forwarding data between the pyramid levels. It can thus not be used for list generation.

Other GPU-based spatial data structures include indirection tables, N^3 trees and octrees¹⁴. Those data structures are more memory-efficient than our solution, but none of them can be generated directly by the graphics hardware and thus need preprocessing.

Recently, an approach to data compaction was introduced⁵, i.e. filtering of unwanted data elements from a given data stream. It does this by successively producing a running sum, which describes where to skip unwanted elements to obtain a packed result. The algorithm needs $\log(n)$ iterations to produce this running sum, and keeps the number of output elements constant during these iterations, which puts a considerable burden on memory transfers.

This can be improved by utilizing all intermediate data levels in the point list reconstruction, a reduction from $\log_2(n) \cdot n$ to $2 \cdot n$ data elements in the intermediate data output¹⁴.

Our algorithm uses a similar reduction approach, but in *2D space (cell coverage: 2x2:1)*, in contrast to the 1D nature (2:1) of both previously introduced algorithms. Besides mapping closer to the GPU's 2D image storage and texture caching mechanisms, it can smoothly integrate with the feature combiner (which is 2D in nature, in the case of quadtrees). Our approach further omits the assumption of a certain number of processors in order to keep the algorithm general for future GPU generations.

3. OVERVIEW

Figure 1 illustrates the workflow between the different computation steps. For ease of understanding, we demonstrate a simple quadtree system, one that is only able to extract a color quad list, i.e. a list of quads of a certain color (which the discriminator identifies, here: yellow). All data is processed on the GPU – the CPU is only handling data if the quad list shall be downloaded, in case a non-GPU application requires so.

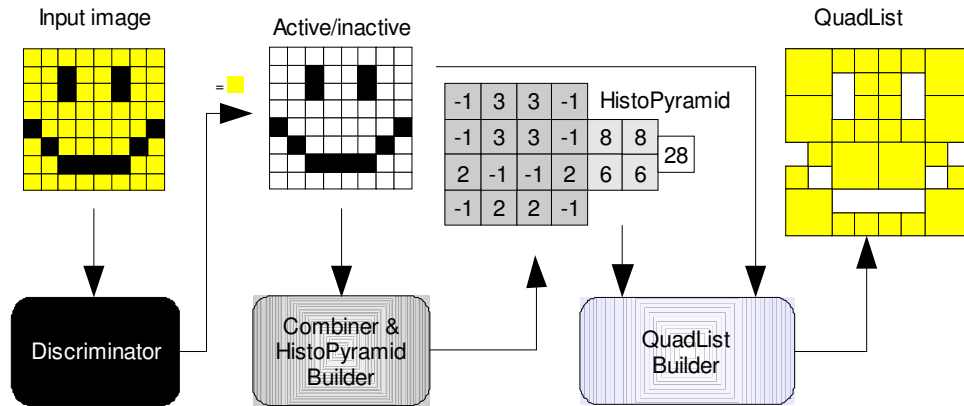


Figure 1. Data flow for a simple quadtree analysis.

The **Discriminator** decides if a cell's content is regarded as active (-1) or not (0). The image cells may be of arbitrary type (single/RGBA, byte/float), as long as the Discriminator is able to handle cells of such type. Section 4 explains more details.

The **QuadPyramid Builder** receives the discriminator output, and first combines identical features as long as it can. When it cannot combine similar cells anymore, it switches to count mode and builds the pyramid of partial histograms.

Its reduction operator repeatedly processes four input cells into one, starting at the resolution level of the original input image. It finishes when only one output cell remains. We describe its GPU implementation in Section 5.

The **QuadList Builder** takes the prepared HistoPyramid, creates a 2D array, also known as quad list, and fills every list entry by using a hierarchical top-down traversal of the HistoPyramid. Section 6 describes details of the traversal in diagrams, and presents a log of the traversal decisions. Further, it is discussed which GPU restrictions hamper performance for the 3D case, and how their removal might improve future implementations.

We have also devised algorithmic variants, including one which uses the GPU's native vector capabilities, and a version which utilizes bilinear texture interpolation. A discussion of these variants can be found in section 7.

The very basic nature of our proposed concepts can make it hard to understand the full range of new applications that a GPU implementation opens. Therefore, section 8 outlines several real-time applications that become feasible with a GPU implementation of this algorithm.

Section 9 summarizes the current performance results that we obtained by running the algorithm's variants on state-of-the-art graphics hardware. It describes the surrounding test setup, and analyzes their runtime behaviours.

4. DISCRIMINATOR

Before a quadtree can be constructed from input data, we first need to determine what we regard as active cells at the base level (or: as the smallest quad tree leaves at the base level). The meaning of the input data can vary greatly: in a sparse matrix, all non-zero entries might be of interest, while in a vision application, it might be important to extract all occurrences a certain color range (see Figure 2 for an example). In PDE applications, it could be important to re-calculate all areas which exceed a certain maximum error threshold.

The discriminator is responsible for concealing all these data modalities from the later stages. It transforms the input into a simple decision: quadtree leaf (-1) or not (0). Of course, even if the output is *binary*, the parameters can still be modified by the user. In our test application `ffkvadrat`, for example, we can modify the gradient threshold on the fly, allowing us to be more or less rigid in merging areas of similar colour.

The discriminator function does not have to be restricted to its associated input – it can also take the local neighborhood into consideration. We use this in Section 7.2 to perform gradient-based edge detection in the discriminator. Further, more advanced examples can be found in our previous report on GPU point list generation¹⁶.

5. QUADPYRAMID BUILDER

Usually, a reduction operator is used for a simple task, namely to *apply a global operation* to all elements while still making use of stream parallelization. Our reduction operator is slightly more advanced: It changes its mode of operation while it builds the reduction pyramid, and thus combines feature mat with the data structure necessary for GPU list construction.

Initially, the reduction operator acts as a Feature Combiner. For every new cell it outputs, it checks the input cells' features for a match in order to create a new, larger quad leaf. In simple versions, this is a check if all incoming cells are leaf cells themselves.

If the cells do match, then the leaf marker value is written to designate the single, combined quad tree leaf. In our case, the leaf marker value is a -1. Figure 2 shows a simple example where only this combiner mode is used. Cells are combined into larger quads, and the leaf marker thus propagates all the way to the top of the pyramid. In the following build process (to be described later), the resulting quad list would only contain one quad, one that covers the whole input image. In more advanced versions, the feature combiner acts as a recursive feature filter. Section 7 explains this in more detail.

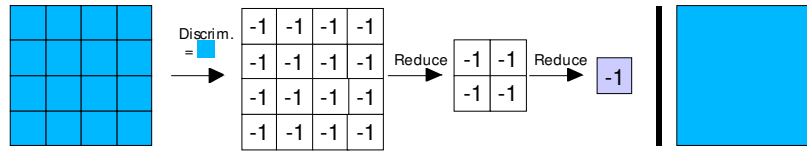


Figure 2. A simple example to demonstrate combiner mode. Right: Resulting quad.

If the cells do *not* match, then the reduction operator becomes a sum operator, and effectively starts to count incoming elements. All present quad leafs are now being treated as single elements in the forthcoming reduction. These leaf markers need of course special treatment in the counting operation. But since we have marked quad leafs with a -1, an `abs()` function call before the sum-up operation suffices to count a quad leaf as one single element in the reduction process. Not even the base layer needs special treatment, since active cells have already been marked with a -1 there. Figure 3 shows an example where this happens for the first time in the upper right corner at L1 of the pyramid. Here, the three leaf markers cannot be combined into a new leaf marker, and thus must be treated as separate elements.

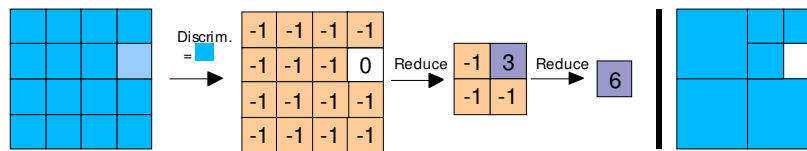


Figure 3. A switch to count mode in L1, some leaf markers cannot be combined. Right: Result.

We have effectively extended the original concept of the HistoPyramid with “a quad tree leaf marker” to register cells with alike features. We call this new variant a quadtree HistoPyramid, or short: *QuadPyramid*.

For N pixels in a quadratic arrangement, the GPU needs 2N read-write operations to create the QuadPyramid, just like for mip-mapping.

6. QUADLIST BUILDER

Given the QuadPyramid as input, it is now possible to determine the *number of list entries*. QuadList Builder accesses the top level of the pyramid to retrieve this value, and allocates the *quad list*, a 2D array sized large enough to hold the number of entries (see Figure 4 for an example 2D quad list). The reason for choosing a 2D layout is that the GPU currently only can handle 4096 entries at maximum in a 1D image. The GPU treats this array as 2D image.

Now, actual quad list reconstruction commences. In our example in Figure 4, the QuadList Builder shader will now be called for all six possible list entries in the 2D image.

The shader first determines its own index (the *key index*) from its 2D coordinate in the point list. Since it has also been given the total number of entries (the quad count, here: 6), it immediately terminates if the key index exceeds the list count (this did not happen in this example, but is possible if the 2D image is larger than the number of entries).

The algorithm descends if the key index lies within the *index range* of a QuadPyramid cell. Intuitively, the index range of a QuadPyramid cell describes the *covered* range of key indices that quad leaves can receive in this part of the 2D input image. The top level's single cell is a good example: its range covers all quad leaves' indices.

There is one exception to this rule: if the traversal encounters a leaf marker (-1), and the key index matches the index of this marker, then no descend occurs. Instead, traversal is *terminated early* to immediately write the output (see e.g. key index 4 in the example). The current position is scaled up to correspond to base level coordinates, and the size of the quad leaf is determined from the pyramid level at which the traversal terminated. In the example case of key index 4, the traversal terminates at L1, the coordinate is scaled from L1's (0,1) to L2's (0,2), and the output size is 2x2, since the traversal terminated one level above the base level. Figure 5 shows all leaf markers which cause traversal to stop in our example.

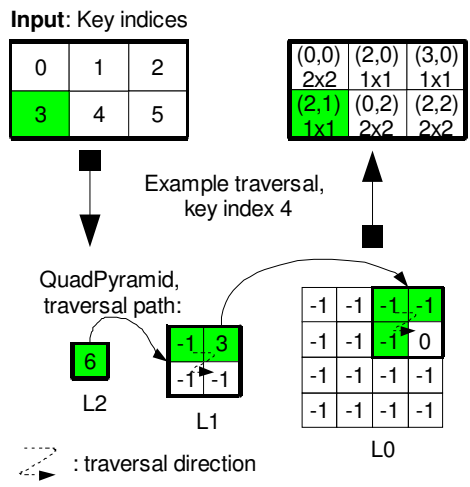


Figure 4. Example traversal for key index 3 (marked in green).

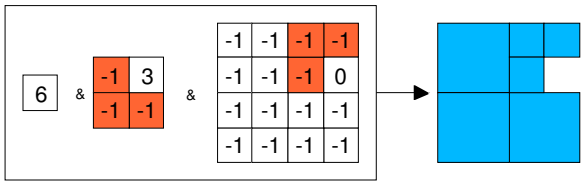


Figure 5. Left: Red marks where traversal ends in the quadlist construction. Right: Result.

During traversal of the QuadPyramid, the current index range **[start, end]** is updated as follows (initially, start is initialized to zero):

- The cell's content is looked up from the QuadPyramid. If it is a leaf marker and the index matches, then the algorithm terminates with the current position as quad leaf origin, and deducts the quad leaf size from the current QuadPyramid descend level.
- **end** is assigned the sum of the cell content and **start**.
- A descend happens if the key index falls into $[start;end[$. On descend, we retain the **start** value of the pre-descend range check.
- Before a new cell is examined on the same level, **start** becomes the former index range **end**.

Note that the traversal order is irrelevant as no sorting is enforced. It only needs to be the *same* order for all quad list entries to avoid doublettes. That is certainly fulfilled as the QuadList Builder shader is the same for all pixels. QuadList Builder assigns a *unique* active cell to each index, but the indexing order is somewhat unintuitive (based on a fractal traversal pattern). The final result is thus a 2D array containing combined origin/size entries of all quad leafs in the image, the *quad list*. For a square image of N pixels, the GPU thus needs at most $m \times \log(\sqrt{N})$ texture accesses to generate all m quad leaf entries in the list, depending on where the quads' leaf markers were encountered.

7. ALGORITHMIC VARIANTS

The simple version of the algorithm depends on the Discriminator to identify the active cells of an image. Since this is a binary decision, it is not able to group different features at once. Instead, it has to run multiple passes, every time with the Discriminator set to identify a new feature. Therefore, we have put some thought into how the feature combiner can be extended to become an image analyzer, and thus to process more multi-featured input data in one pass, especially for cases where the exact features are not known beforehand.

7.1. Common Feature Combiner variant

In this variant, the Discriminator becomes unnecessary. Instead, we let the feature combiner calculate the feature average over the 4 input cells, and then calculate the deviation of the input features to the average feature. A user threshold on the deviation defines if the input cells are considered for potential merging.

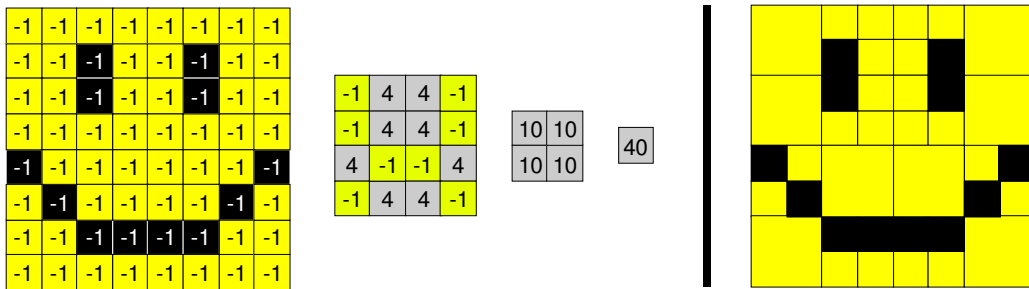


Figure 6. The common feature combiner creates an average of the contributing cell colors. Right: Result.

Since the deviation thresholding is baked into the reduction operator and thus recursive, we store the calculated averages in the QuadPyramid. When no merger is possible, the average becomes uninteresting (grey cell background in Figure 6).

The advantage of this algorithm lies in its simplicity: No pre-processing is required to acquire multi-feature clustering. The features do not need to be known in advance. It works accurately down to base level.

However, there are two disadvantages: First, the QuadPyramid needs to accommodate the averaged feature vectors, which increases memory consumption considerably. Secondly, the maximum deviation will *differ* for cells in one quad, depending on which level they have been combined. The reason for this is that at each level only receives the *averages* from one level below, not the actual cell values. Still, the algorithm delivers fast results if the deviation value is chosen conservatively, and especially if only *feature-identical* cells shall be grouped (and deviation can thus be set to zero).

7.2. Edge Thresholding variant

In this variant, we make use of the observation that feature grouping is based on local neighborhood information. Since gradient-based edge detection calculates the local variance of features, it is subsequently possible to group areas of low feature variance, having ensured that the maximum deviation is below a certain threshold. It is of course possible to have a certain drift in larger areas as there is no global enforcement, but this is after all more natural for humans, as their impression of continuity is based on local similarity decisions. In Figure 7, we exemplify how this general concept can be applied to color as a feature. Here, the discriminator makes its initial decisions based on thresholding after a classic gradient-based edge detection: pixels with high local variance are not considered possible quad leaves (are 0), and will thus break up any grouping attempts.

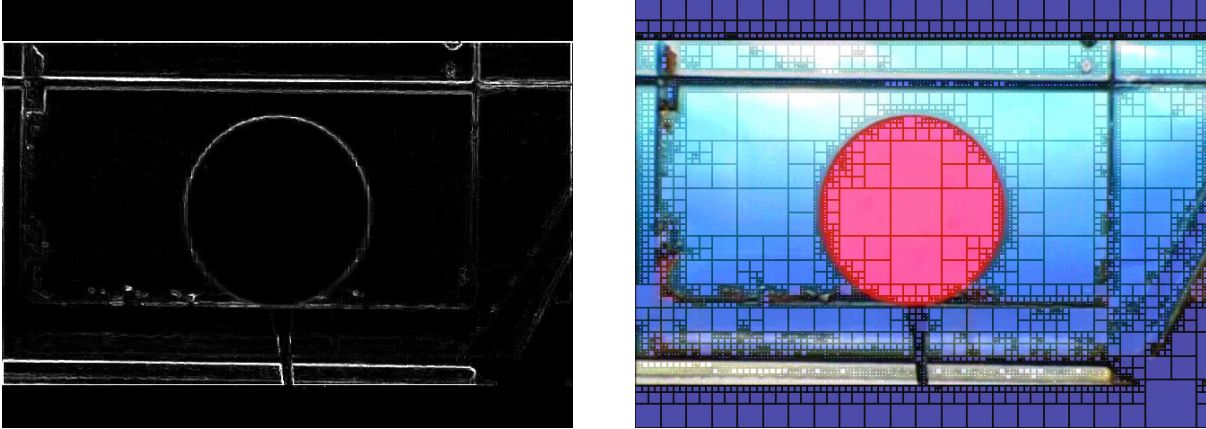


Figure 7. Left: Edge detection indicates non-mergeable areas. Right: Resulting QuadList.

The advantage of this method is that its memory consumption remains low, and that the combiner decisions are very intuitive for humans (low local variance leads to grouping). One disadvantage is that grouping is based on edge information and thus not pixel-exact (pixels with high variance in their neighborhood will not be grouped at all, see the red disc's outline in Figure 7, or use Figure 6 for a thought experiment with this algorithm).

8. APPLICATIONS

8.1. Feature clustering in Computer vision

One of the basic problems of computer vision is local feature clustering, as e.g. clustering of color, motion vectors or depth. While numerous algorithms exist for this issue, many of them fail in achieving real-time speed for video, and are thus only of limited use to interactive vision applications. Region Quadtree extraction can of course not provide a complete connected component analysis, but it does retrieve a quick, light-weight clustering (as shown in e.g. Figure 7), and can thus be used to detect objects of a certain minimum size, or allow to focus calculations on regions of interest, e.g. ones with high variance (by inverting the discriminator behaviour of the Edge thresholding variant in Section 7.2)

8.2. QuadTrees in Data Compression

Since data compression can take advantage of quadtrees³ and higher-dimensional versions, like octrees¹⁷, it is probable that this algorithm can be used in compression applications for stream processors, especially for GPU-preprocessed volume and image data. To achieve even higher compression for the quad positions and size, it might be necessary to store the actual traversal path for each quad list entry. Actually, this poses no larger problem as each shader pass knows about the full traversal path from the pyramid top down to the quad leaf marker, and can store this sequence of 2-bit cell descend decisions in one or more extra output variables.

8.3. Light-weight preclustering of linearly behaving regions

It is typical for PDE-based applications to operate on a specific spatial discretization and to require calculations of varying intensity for different locations^{2,18}. A simple solution to stay below a certain error measure is to increase spatial resolution (that is, tighten the grid of discretization). Unfortunately, this is usually wasteful in many areas except for the hotspots. By using the algorithm in this article and a specially adapted discriminator (specific for each PDE problem, and thus not described in this article), it is possible to identify regions where more fine-grained calculation is necessary.

Differently put: If a function is locally linearly interpolatable, as in Equation 1,

$$f(t.a + (1-t).b) = t.f(a) + (1-t).f(b) \tag{1}$$

then a quadtree or octree analysis can be used to isolate regions where such function result interpolation could replace sample-by-sample calculations (for complex $f()$), and it can isolate regions where an increased step width in PDE solvers would not do any harm in terms of simulation stability and thus benefit general purpose GPU computation¹¹.

8.4. Octree generation for computer graphics and visualization

Volume analysis, raytracing applications and three dimensional PDE solvers can make good use of octrees, a linked data structure which describes 3D blocks that share a common feature or are empty.

Even this can be realized on the GPU, if the presented algorithm is modified to handle volume data in the following way: First of all, it is necessary to lay out the slices of 3D volume data as tiles in a 2D texture^{4,15}, as there is no way to render to 3D textures on current GPU architectures (this has changed on Nvidia's G80 hardware). Next, we need to adapt the reduction operator: instead of 2x2 cells in 2 dimensions, it will now fetch *2x2x2 cells in 3 dimensions* to build an *OcPyramid*. The concept of a leaf marker remains the same, a name suggestion would be "octree leaf marker" or "ocleaf marker". A similarly adapted *Oclist Builder* will then have to decide a descend among 8 cells instead of 4, but remains conceptually the same as QuadList Builder.

Performance characteristics of such a 3D version should remain similar, only the GPU texture cache will have to handle two seemingly "unrelated" 2D texture accesses for each run of the reduction operator, if a 2D tiling is used.

If a list is not needed, then the OcPyramid can be used directly in the application by adapting the concepts of the Oclist Builder according to needs.

9. RESULTS

Finally, we present the results of some performance tests. The tests were conducted on a Dell Precision M70 laptop with Nvidia Quadro FX Go 1400 and 256 MB video memory, connected over PCI Express. It contained an Intel Pentium M (2.13 Ghz) and 2 GB of main memory. We could not test the algorithm on ATI graphics hardware without intense redesign due to the ATI driver's OpenGL API restrictions.

Our first tests were run with *kvadrat*, a Linux application which is able to decompose an image based on the feature combiner variant, as described in Section 7.1. It decides about a merger based on the sum of the squared deviation from four input cells' common RGB color average. Figure 8 shows an example from freely downloadable video footage²⁰.

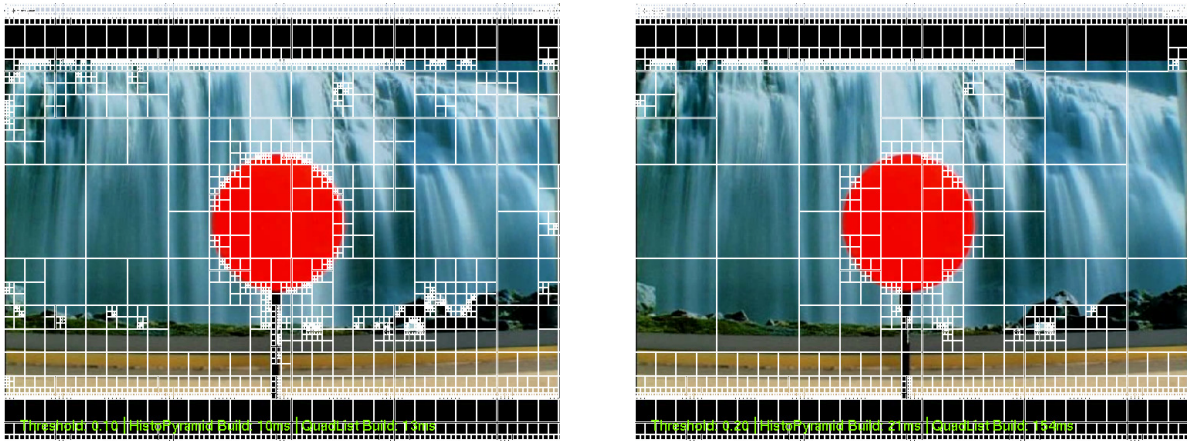


Figure 8. Two quadlist visualizations for different thresholds. Left: 0.20, Right: 0.10.

At certain thresholds, the result seems to be intuitive. But as the threshold changes, it quickly becomes obvious that this is not what a human what segment: Parts of the disc are still considered separate from the background, while others have already merged with it. See Table 1 for timing measurements. Instead of taking the time consumed by the OpenGL call delays on the CPU side, we measured actual GPU timings with the `GL_EXT_timer_query` extension¹⁰.

Threshold value	0.1	0.2
QuadPyramid Builder	10 ms	10 ms
QuadList Builder	12.4 ms	5.7 ms
Resulting quads	13261	7981

Table 1. Timings for different thresholds, feature combiner algorithm.

We further compared this algorithm with a CPU implementation. Aggressive compiler optimization accelerates the CPU based analysis. No SIMD techniques were used. CPU timings were taken as virtual process time by the `getitimer()` function of Linux systems. The CPU performs one pass at around *50 ms*, which seems fast, considering that no SIMD techniques were involved. It has to be considered, though, that this performance was due to the fact that the data *already resided in CPU memory*. Most of the time, the GPU has been used in preprocessing or generation of data, which would require a *bus transfer* to download the data to CPU memory before it can be analyzed. If this is the case, then it ends all possible performance competition, as the bus transfers take 3-10 times the time of the actual computation¹⁶.

In order to test the real-time behaviour of the edge thresholding algorithm from Section 7.2, we implemented *ffkvadrat*, a Linux player which analyzes video while it is being played back from GPU texture memory. After the video has been decoded as YUV 4:1:1 subsampled data planes, it is uploaded into GPU memory as three separate textures. Before running a simple fragment shader which converts the Y, U and V textures into RGB color values, we pre-process the video frame with edge analysis in the four major directions. Now, we use this edge texture as the input for a region quadtree analysis as described in Section 7.2, where a quad leaf marker is a squared gradient whose sum of components stays below a certain threshold (that is, it doesn't contain any edge information to speak of). Again, you find timings in Table 2.

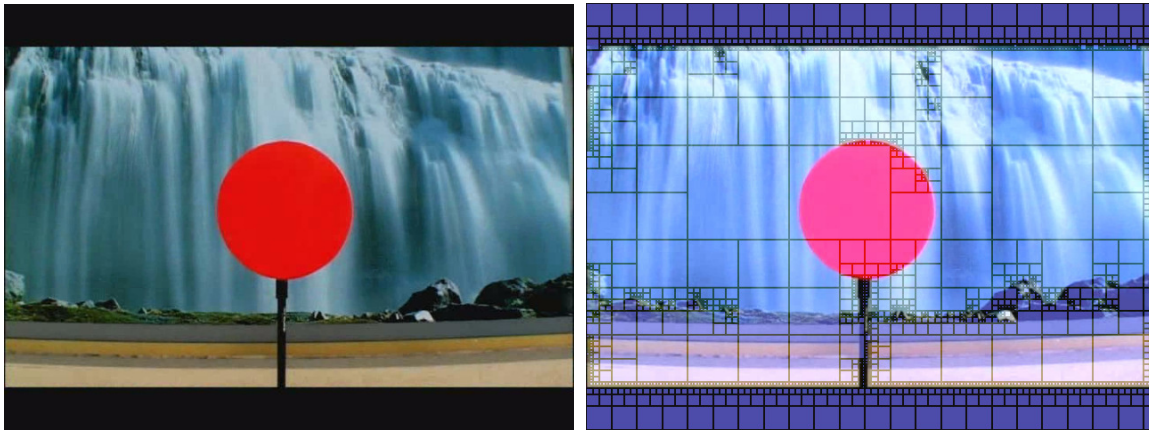


Figure 9. Left: Input frame. Right: Threshold value 0.92.

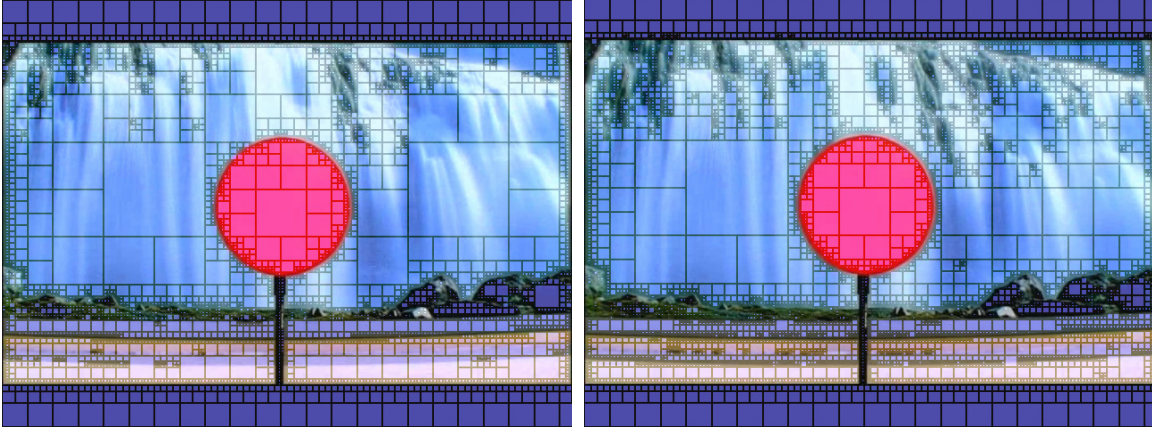


Figure 10. Left: Threshold value 0.22. Right: Threshold value 0.10.

Threshold value	0.10	0.22	0.92
QuadPyramid Builder	4 ms	4 ms	4 ms
QuadList Builder	13 ms	12 ms	11 ms
Resulting quads	22809	15375	4474

Table 2. Timings for different thresholds, edge thresholding algorithm.

Figure 9 and Figure 10 show the more intuitive results of the edge thresholding approach. While a high threshold does not consistently isolate any features, it starts to separate out the disc from the background at value 0.22, and even separates the street's different elements at value 0.10. We can therefore recommend this algorithmical variant for color segmentation, but more investigations will of course be needed for other feature data types.

We eagerly await Shader Model 4.0-capable graphics hardware in order to test how integer handling can improve performance. We would also like to test render-to-texture for 3D textures, as OcPyramid traversal would cache more efficiently in a real 3D layout.

10. CONCLUSION

In this submission, we presented how quadtree based image partitioning can be done at unprecedented speed on stream processors. We have extended our previously published GPU point list generator to create region quadtrees from video within a dozen of milliseconds. The proposed method is novel, fast and easy-to-use, and has a multitude of applications in image and video processing, as it makes region quadtree analysis a light-weight preprocessing step.

Through experiments we have shown that our purely GPU-based implementation is significantly faster than a hybrid GPU/CPU implementation. The algorithm is highly cache-friendly, and scales innately with increasing parallelization. It will be highly interesting to see how it maps into image analysis, data compression and general purpose computation. In general, there should now only be few computational tasks left that can *not* be done on GPUs.

REFERENCES

1. Buck, I., and T. Purcell: *A Toolkit for Computation on GPUs*. GPU Gems, pp.621-636,2004.
2. Bolz, J., Farmer, I., Grinspun, E., and Schröder, P.: *Sparse matrix solvers on the GPU: Conjugate Gradients and Multigrid*. ACM Transactions on Graphics 22, pp. 917-924, 2003.
3. Hanan, S.: *Data structures for quadtree approximation and compression*. Communications of the ACM, Volume 28, Issue 9, pp. 973-993, 1985.
4. Harris, M.: *Fast Fluid Dynamics Simulation on the GPU*. GPU Gems, pp.637-665,2004.
5. Horn, D.: *Stream Reduction Operations for GPGPU applications*. GPU Gems 2, pp. 621-636, Addison-Wesley.
6. S. Sengupta, A. E. Lefohn, J.D. Owens: *A Work-Efficient Step-Efficient Prefix Sum Algorithm*. Proc. 2006 Workshop on Edge Computing Using New Commodity Architectures, pp. D26-27.
7. Krüger, J. and Westermann, R.: *Linear algebra operators for GPU implementation of numerical algorithms*. Proc. ACM SIGGRAPH 2003, pp. 908-916, 2003.
8. Simon Green, NVidia Corp.: *OpenGL Image Processing Tricks*. GDC 2005 Presentations, 2005.
<http://tinyurl.com/f59r4>
9. NVidia Corp.: *Image Histogram*. SDK Code Samples - Video and Image Processing, 2004.
<http://tinyurl.com/kmlr2>
10. Simon Green, NVidia Corp.: *NVidia OpenGL Update*. GDC 2005 Presentations, 2005, pp. 40-42.
<http://tinyurl.com/pngld>
11. J A Robinson, *Efficient General-Purpose Image Compression with Binary Tree Predictive Coding*. IEEE Transactions on Image Processing, Vol 6, No 4, April 1997, pp 601-607.
12. Fung J., and Mann S.: *OpenVIDIA: parallel GPU computer vision*. Proc. 13th annual ACM international conference on Multimedia, 2005, pp. 849 - 852.
<http://openvidia.sf.net>
13. Magnus Strengert, Martin Kraus, and Thomas Ertl. *Pyramid Methods in GPU-Based Image Processing*. Proceedings VMV 2006, pp. 169 - 176
14. A. Lefohn, J. Kniss, R. Strzodka, S. Sengupta and J. Owens. *Glift: Generic, efficient, random-access GPU data structures*. ACM Transactions on Graphics 25, 2006.
15. Gernot Ziegler, Art Tevs, Christian Theobalt, and Hans-Peter Seidel, [*GPU PointList Generation using HistoPyramids*](#), In Proceedings VMV2006, Germany, 2006, pp. 137-144
16. G. Ziegler, A. Tevs, C. Theobalt and H-P. Seidel. *GPU Point List Generation through Histogram Pyramids*. Technical Reports of the MPI for Informatics, June 2006, MPI-I-2006-4-002.
17. J. Zhang, C. Owen, *Octree-based Animated Geometry Compression*, Proceedings of the Conference on Data Compression, 2004, Page 508
18. Losasso, F., Gibou, F., and Fedkiw, R. 2004. *Simulating water and smoke with an octree data structure*. ACM Transactions on Graphics 23, 3 (Aug.), 457-462.
19. M. Houston, *GPUBench Report, GeForce 8800 GTX/PCI/SSE2*, Nov 2007,
<http://graphics.stanford.edu/projects/gpubench/results/8800GTX-0003/>
20. GusGus, *Desire*. Full quality music video footage, <http://www.gusgus.com/>