Compiling High Performance Recursive Filters

Gaurav Chaurasia	MIT CSAIL
Jonathan Ragan-Kelley	Stanford University
Sylvain Paris	Adobe Research
George Drettakis	Inria
Fredo Durand	MIT CSAIL

How to make convolutions faster with recursive filters?

Why are most implementations of recursive filters not optimal?

How can a compiler solve this problem?

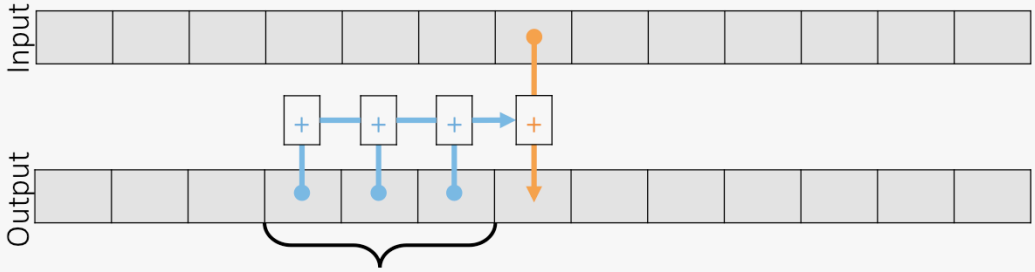
2

In this talk, I will tell you how to make convolutions faster with recursive filters. Then I will explain why most implementations of recursive filters are far from optimal and finally present a compiler that solves this problem.

Large images require convolution with large kernel sizes. A simple low pass filter can be up to 100 pixels wide. Therefore, we have always been looking for ways to make convolutions faster.

Recursive Filters

$$f(x) = I(x) + a_1 f(x-1) + a_2 f(x-2)...$$



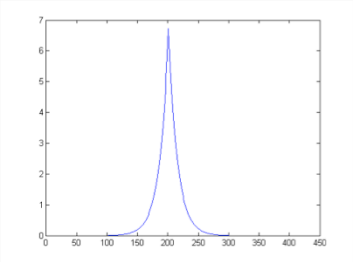
Filter order = number of previous pixels

4

Recursive filters are an interesting alternative to convolutions. A recursive filter compute the output at any pixel by looking at a fixed number of pixels ffrom the input image, and a fixed number of previous output pixels. This recursive nature gives them some interesting properties.

Why use recursive filters?

Large low pass filter using 4-5 operations per pixel

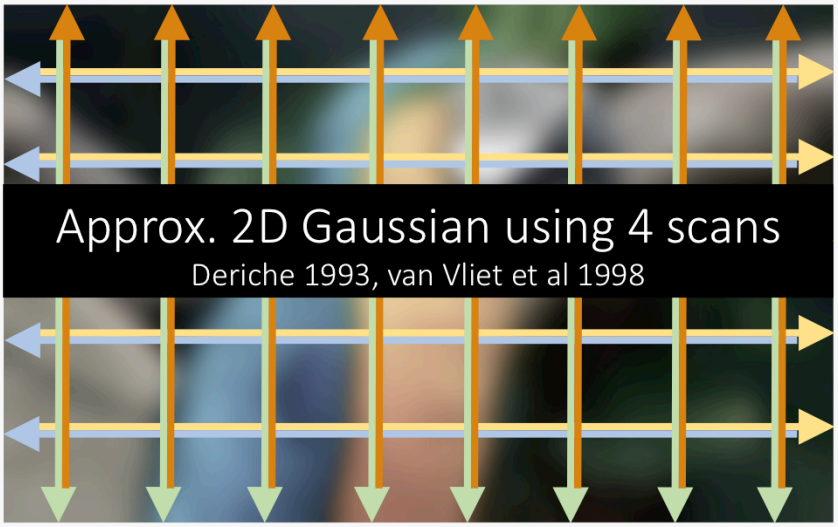


Impulse response

5

These filters look like large footprint low pass filters. So you can get a large footprint with a constant number of operations per pixel.

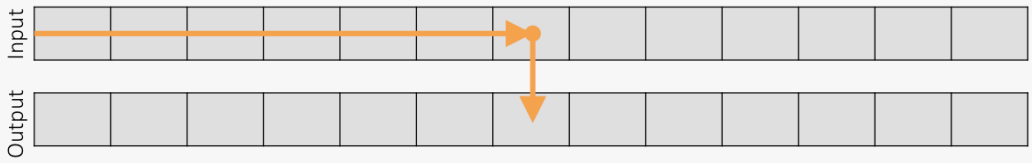
Forward/backward scans in all dimensions



In practice, you rarely use a single recursive filter. If you have a 2D image, you need scans in both dimensions. Sometimes, you need scans going from left to right as well as right to left for symmetry.

Computational challenges

Serial computation - no parallelism



How to exploit parallel hardware?

7

However, these filters are especially hard to optimize. They compute out pixels in a specific order. How can we exploit parallel hardware?

Previous work on parallel recursive filters

Parallel prefix sum [Sengupta et al 2007]

Parallelized special cases like summed table

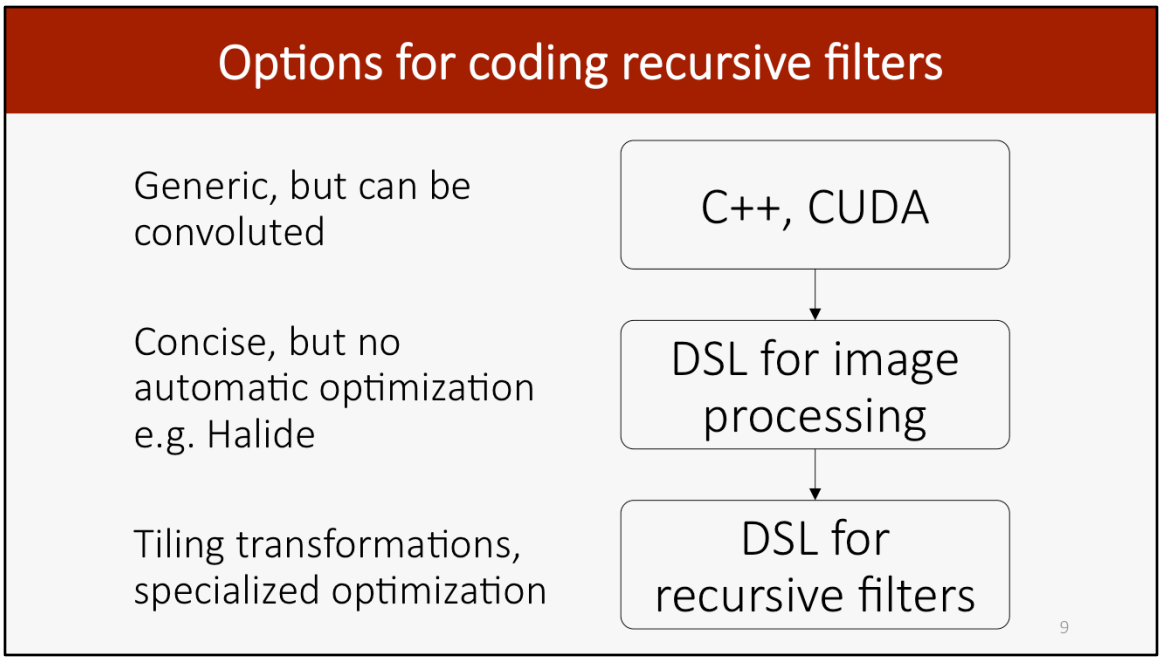
- Nehab et al 2011, Kasagi et al 2014

What about generic recursive filter pipelines?

- n^{th} order, multiple scans, multiple dimensions

8

Previous work has addressed these issues for specific filters. A lot of work has focused on parallel prefix sum – which is simplest 1D recursive filter.
These have been extended for other specific cases like summed area tables. But what about a generic pipeline of n filters of any order?



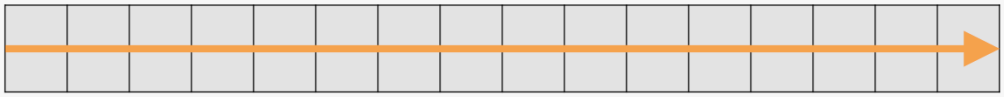
The most general approach for writing image processing code is in C++ or CUDA. Modern DSLs provide a concise way of writing image processing. But magically optimize your filters. The programmer is responsible for lots of details. We will go further down this road and present a more specific DSL that can perform specialized optimization for recursive filters.

Recursive Filter Transformations

Now lets try to understand the transformations that can optimize recursive filters.

Tiling transformation

Extra parallelism by tiling



Kooge & Stone 1973, Blelloch 1989, Sengupta et al 2006

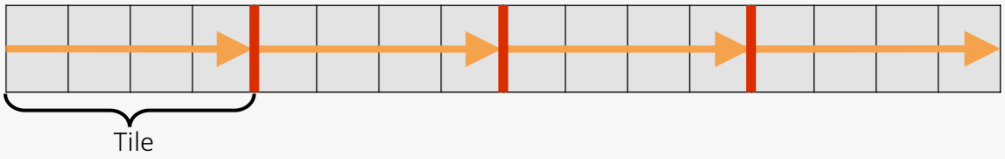
Parallel prefix sum in NVIDIA Thrust

11

Lets start with the simplest case: a 1D recursive filter. This filter scans the input array from left to right. Clearly there is no parallelism. We will try to extract some parallelism by tiling the input data. Variations of this transformation have appeared in previous work, as well as commercial solutions.

Intra-tile: Scan within tile

Compute all tiles in parallel



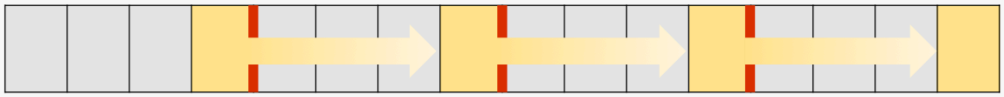
Computed result lacks contribution from other tiles

12

Instead of scanning the complete array from left to right, we split the array into tiles. The first operation is to compute the scan within tiles only, while running all tiles in parallel. This buys extra parallelism. The result is not complete because it does not include contribution from previous tiles.

Inter-tile: Transfer residual across tiles

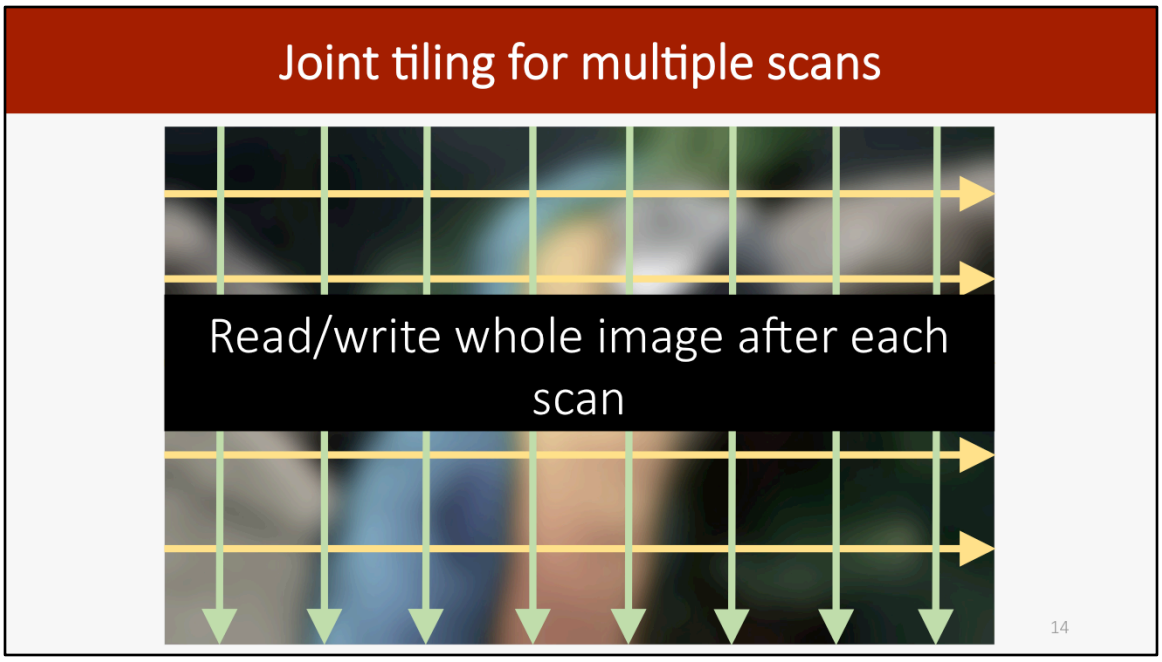
Residual: last element(s) of each tile



Add it to next tile

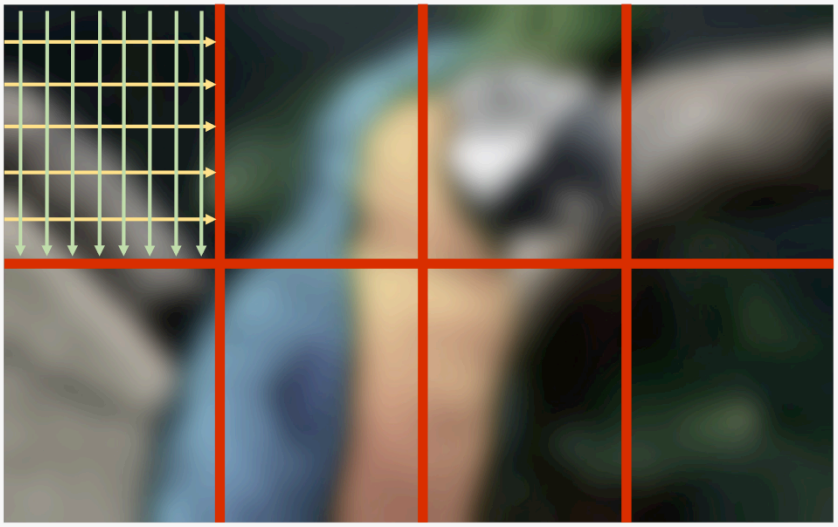
13

In order to complete the result, we extract a residual from each tile – which is the last element or last few elements, depending upon filter order and incorporate into the next tile. This operation loops across tiles and we call it the inter-tile operation. I am skipping lots of details here, but the gist is that the original full array scan is transformed into intra- and inter tile scans. These stages have to be performed one by one to compute the final result.



But what if you needed multiple scans on your input. Consider a 2D image and you want to perform 4 scans – going left to right, right to left, top to bottom and so on. We can use the previous transformation to accelerate each individual scan and cascade all these operations. The problem is that the whole image is read and written after each scan. This leads to high input IO. Can we tile all these scans jointly.

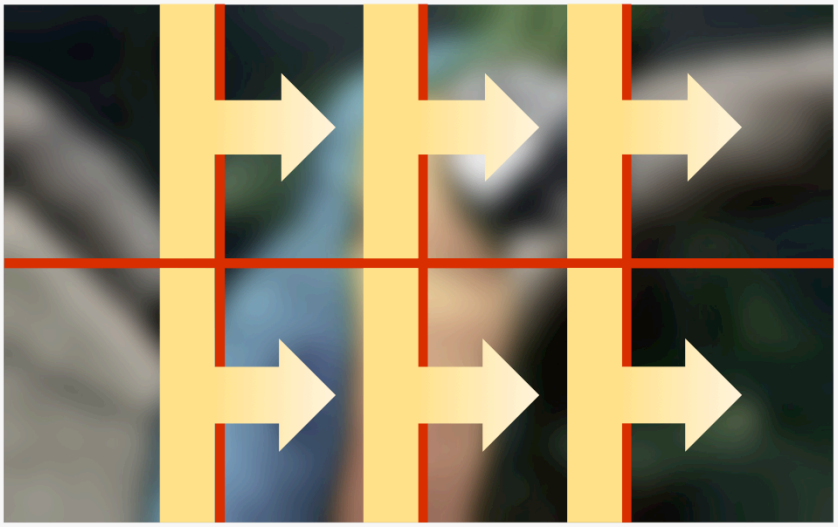
Intra-tile: Scans within tiles



15

We can address this problem computing all the scans inside each tile simultaneously. We compute all scans within the tile. This new intra-tile operation is a simple extension of the corresponding operation in the 1D case.

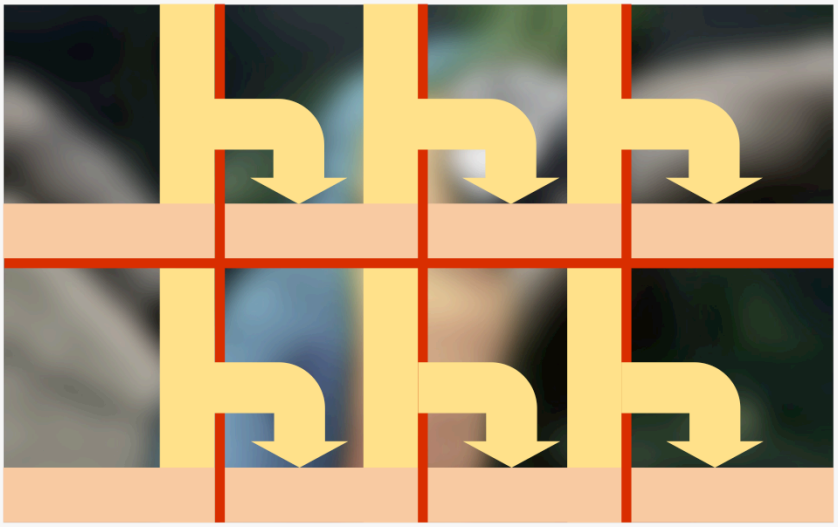
Inter-tile: Residuals from first scan



16

The real problem is resolving the residuals. Remember that previously we had a residual from each tile which we could add to the next tile. Now, since we have multiple scans, we have residuals because of each of the these scans. So we have scans from horizontal scans in both directions and same for the vertical scans. I am again skipping details, but the key point is that we have lots of residuals and these residuals have cross interactions. The residuals for vertical scans need contribution from residuals of horizontal scans.

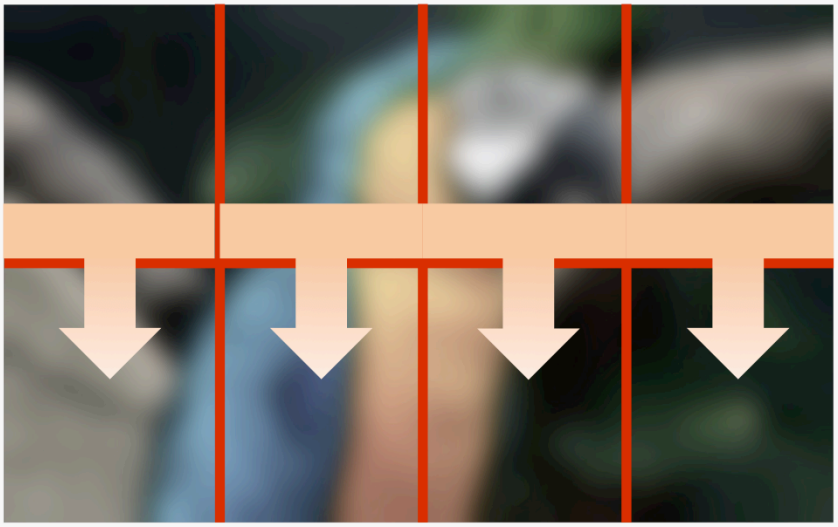
Inter-tile: Cross-interaction between residuals



17

The real problem is resolving the residuals. Remember that previously we had a residual from each tile which we could add to the next tile. Now, since we have multiple scans, we have residuals because of each of the these scans. So we have scans from horizontal scans in both directions and same for the vertical scans. I am again skipping details, but the key point is that we have lots of residuals and these residuals have cross interactions. The residuals for vertical scans need contribution from residuals of horizontal scans.

Inter-tile: Residuals from second scan



The real problem is resolving the residuals. Remember that previously we had a residual from each tile which we could add to the next tile. Now, since we have multiple scans, we have residuals because of each of the these scans. So we have scans from horizontal scans in both directions and same for the vertical scans. I am again skipping details, but the key point is that we have lots of residuals and these residuals have cross interactions. The residuals for vertical scans need contribution from residuals of horizontal scans.

Joint tiling: Tedious implementation

Combinatorial explosion of inter-tile operations

- 2 scans: 3 operations
- 4 scans: 10 operations

500-1000 lines of CUDA for 2-4 scans

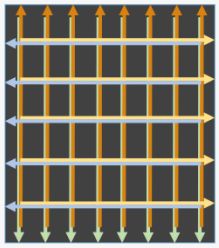
- Nehab et al 2011

Different implementations for different pipelines

19

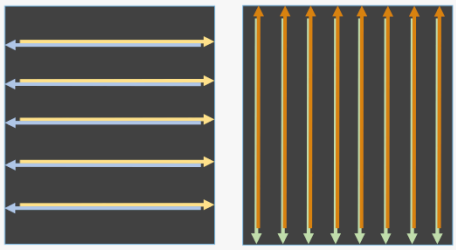
The overall implementation gets more and more complex as you try adding more scans. CUDA implementation of a 2-4 scan pipeline can be 500-1000 lines of code. You will have to implement the whole thing from scratch if you change the number of scans, or change the filter order. This transformation has been described for specific cases like summed area tables. Our contribution is that we develop the algebra to handle a generic set of scans.

Overlap transformation



Tile all 4 scans jointly

Low memory I/O
Complex computational pattern



2 jointly tiled scans in 2 filters

High memory I/O
Simpler computation pattern

20

Now we have transformations for tiling any number of scans. Then for a pipeline with 4 scans, we can choose to tile all of them jointly as in the first example. Or we can tile 2 of them jointly, and the other 2 jointly. These will give different implementations: tiling more scans jointly leads to more complex computation but low memory IO. This tradeoff has a big impact on performance.

Overlap transformation

Lots of possibilities for joint tiling
- Each requires a new implementation

Our solution: code generator to mechanize transformations

21

This transformation gives an insight into the real complexity of the task. If you have n scans, you have multiple combinations of joint tiling, and each requires a different implementation. From a practical point of view, a programmer will never go through all implementations to find the optimal. We present a complier that makes all of this automatic. You just have to specify a filter, pick a tiling option and the complier can generate a made to order implementation.

The Language

Now lets look at how our DSL makes these transformations easy to implement.

Domain-specific language

Automatic tiling transformation

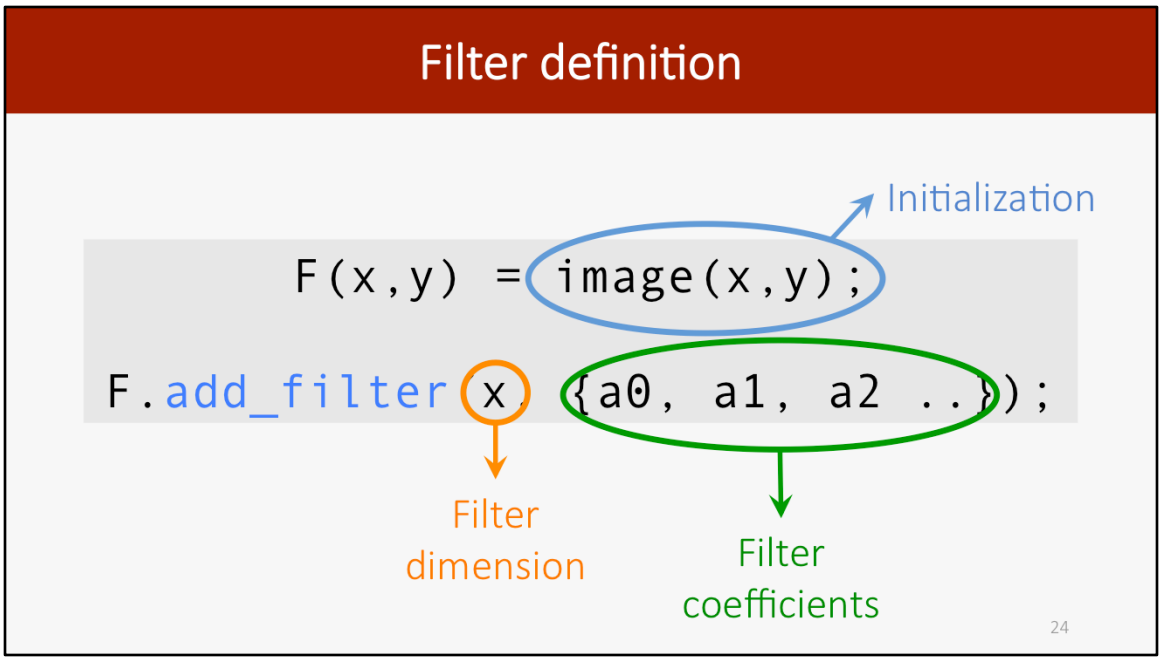
Written on top of Halide [Ragan-Kelley et al 2012]

- Domain-specific language for image processing

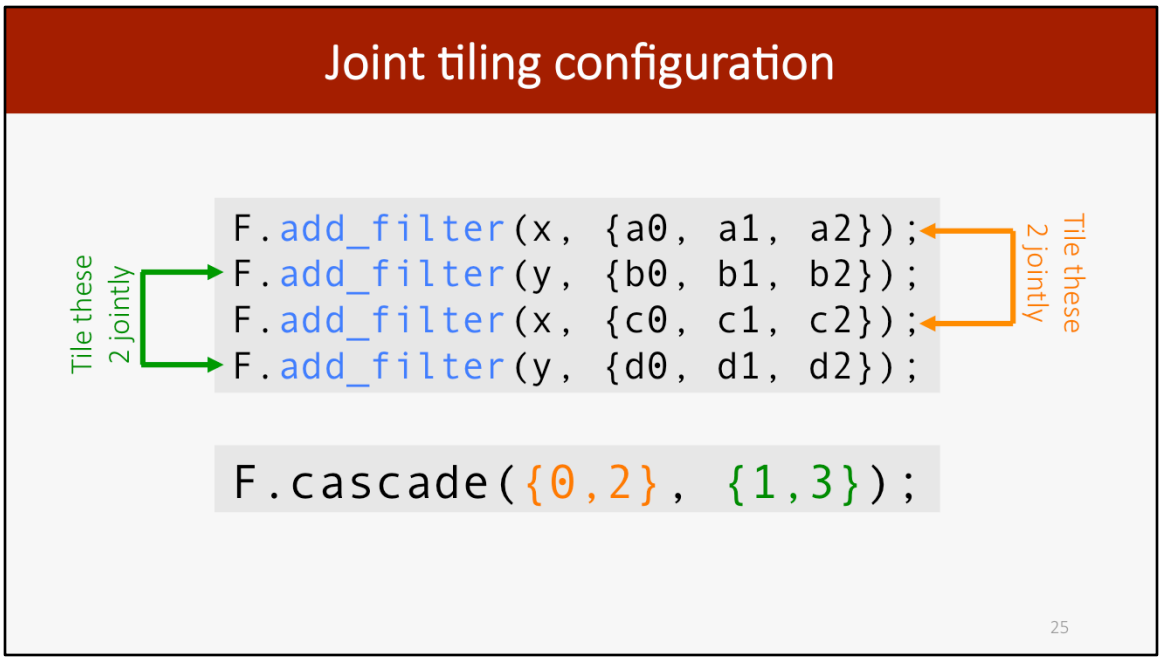
Allows interleaving C++ and our DSL

23

The main purpose of our DSL is to allow convenient implementation of any recursive filter. The compiler is responsible for performing the tiling transformations, the programmer can invoke the transformations with a couple of lines of code. We chose to implement our DSL on top on Halide. Halide itself is a DSL for image processing and any data arranged as a regular grid. The main reason we chose Halide is because it gives a clear separation between algorithm and hardware specific parameters. So far, we have presented tiling transformations. These describe the algorithm only, implementation details such as storage layout, number of threads, loop nest order etc have to be decided separately. Halide offers a clear distinction. Our DSL is embedded in C++, so you can write your recursive filters in our DSL and write everything else in Halide or C++ seamlessly.



Our DSL allows a one line definition of the filter. Here each pixel of the filter is initialized by corresponding pixel of an input image. The second line add one recursive filter in the x dimension. This means that each row of the input image has to be scanned from left to right, applying the filter as we go on. These two lines define a simple recursive filter. If the weights a0 and a1 are 1, then we get a prefix sum. This is a single scan filter.



We can call the same routine multiple times to add multiple scans. In this case, we have added one scan along all rows, then one along columns, then another along rows and another along columns. Each scan is supposed to use the result of the previous scan as input and overwrite it. This gives a filter with multiple scans.

Now before we actually tile the filter, we can choose which scans to tile jointly. Here we chose to tile scans 0 and 2 jointly, and scans 1 and 3 jointly. This means that internally, scans 0 will happen on the input image, followed by scan 2 which will overwrite the result. Then scan 1 will use this as input followed by scan 3. This only line syntax allows us to choose all combinations easily.

Tiling transformation

Tile dimension

```
F.split(y, tile_width);
```

or

```
F.split_all_dimensions(tile_width);
```

26

Finally, we invoke the tiling transformation. We can choose to tile only a single dimension of all dimensions. This generates a tiled implementation where subsets of scans are tiled jointly as specified in the previous slide. Our DSL has three steps – define the filter by adding some scans, then choose which ones to tile jointly, and then tile. These three operations can be used to used to explore all tiling alternatives without sweating over 1000 lines of code.

Two implementation related choices

Where to compute each operation?

- Shared or global memory

How to compute each loop nest?

- Parallelize/unroll/vectorize

27

So far we have discussed transformations describe how to mathematically compute the recursive filter on parallel hardware. We need to answer two final questions to convert the tiled algorithm into an implementation. The first is regarding the locality of operations – which operations to compute once and store in global memory, and which operations to recompute on demand in shared memory. The second important decision is map loop nests of each operation onto parallel threads, or unroll them or vectorize.

Where to compute?

Intra-tile operations: shared memory

- Involve repeated scans on a tile
- Avoids global memory IO

Inter-tile operations: global memory

- Required by multiple downstream operations

28

How to map loop nests on to GPUs?

Pixel indices to CUDA threads

- Transpose storage to coalesce memory

Tile indices to CUDA warps

Unroll recursive scan loops

29

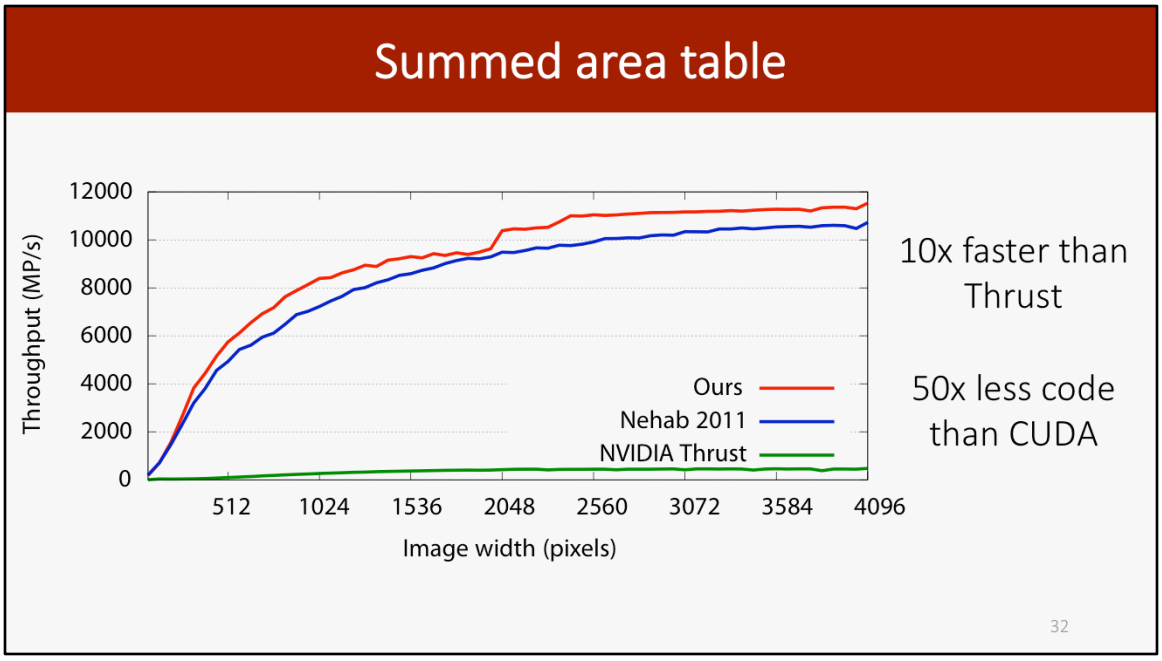
How to map loop nests on to CPUs?

Parallelize dimension with maximum stride

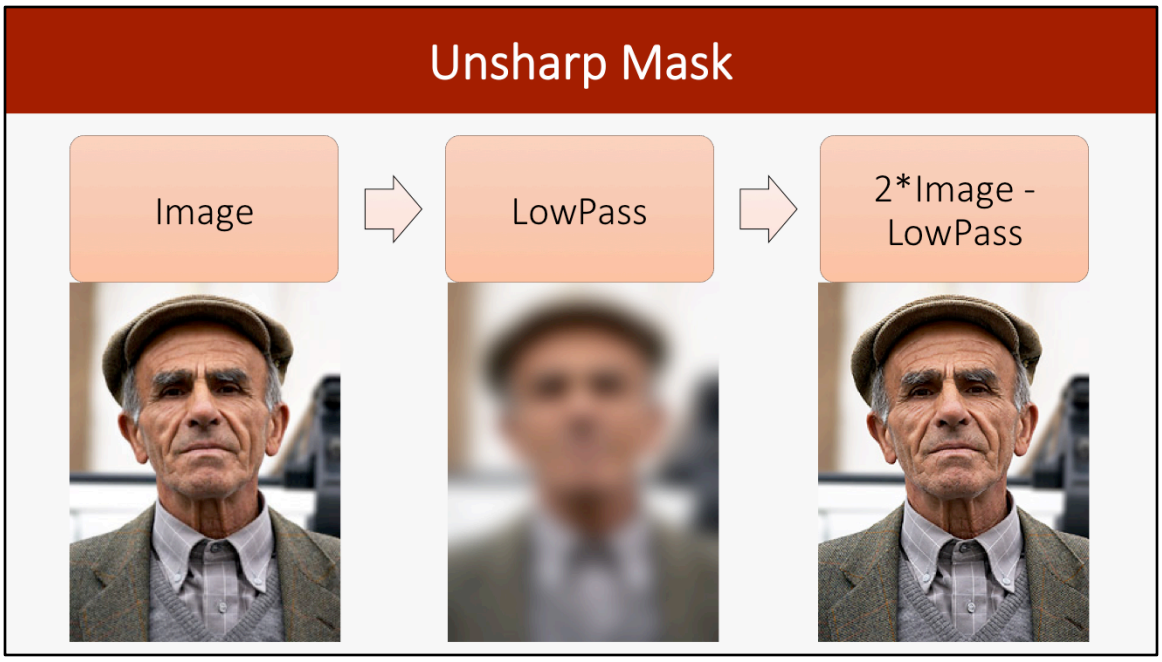
Vectorize dimension with zero stride

30

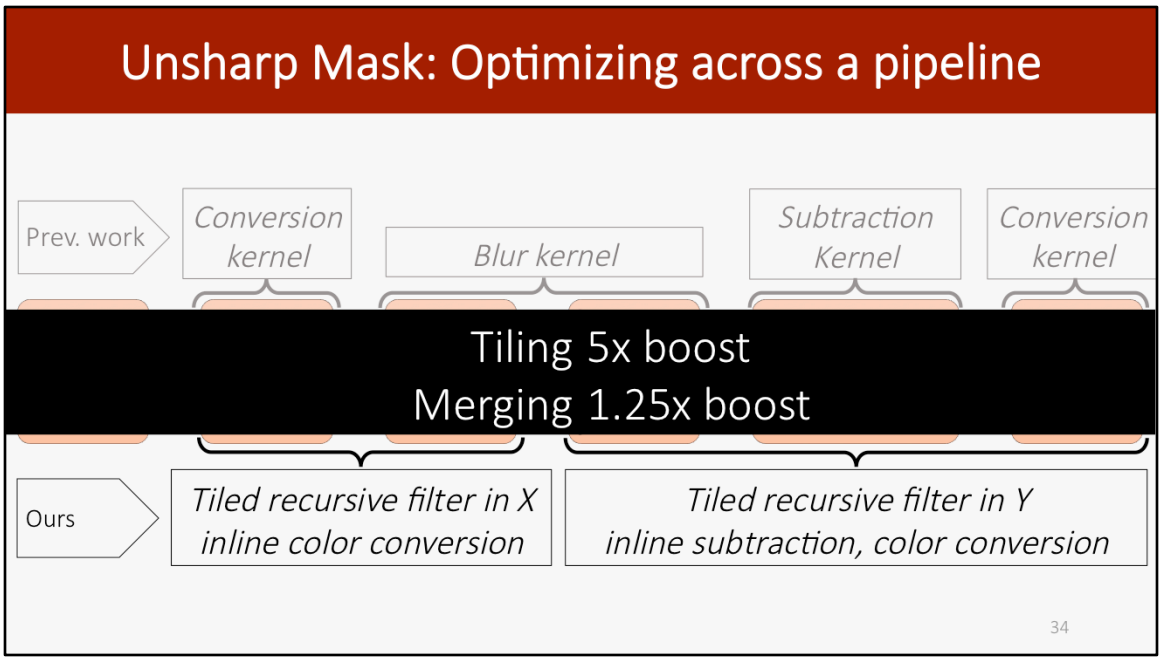
Results



We first compare our performance for the simplest test case of a summed area table. We are an order of magnitude faster than NVIDIA Thrust. This is because Thrust implementation applies prefix sum in x, then transposes the image, then applies the same in y and transposes the image back. The whole image is read and written 4 times. In comparison, our tiled implementation computed both x and y prefix sums in a joint fashion and we only write the full image once. Our implementation also matches the hand tuned implementation of Nehab et al. which uses the same algorithm. This shows the advantage of tiling, while also proving that our automatically generated tiling and scheduling can match hand tuned code.

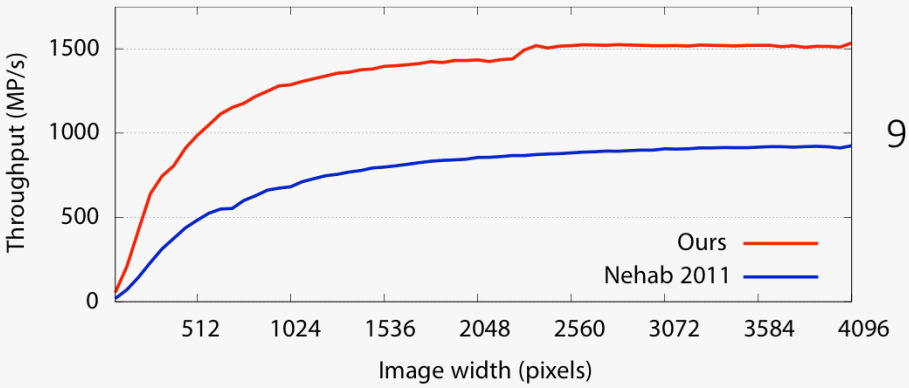


A more interesting example is a detail enhancement filter called Unsharp Mask. This filter blurs the image, then subtracts the blurred version from the original image to give a high pass image. It finally adds the high pass back to the input image to boost detail.



You can implement unsharp mask using a bunch of precompiled kernels. You can find well tuned implementations of RGB to YUV conversion. Then you can use an precompiled kernels which blur the image in both dimensions. Then you can use a subtraction kernel to subtract the low pass form the image and finally convert back to RGB. First of all, you are reading and writing the image at each stage. So the IO is very high. In contrast we reshuffle the computation. RGB to YUV conversion is a pixel to pixel process. We inline this step into the low pass filter along x. We write the result to memory and then start the y low pass filter. The subtraction kernel is also a pixel to pixel process, and so is the subsequent RGB to YUV conversion. We inline both these steps into the y filter. We also wrote both x and y blurs as tiled recursive filters. Here tiling gives almost 5x speedup and reorganizing computation and merging different pixel to pixel operations gives additional 25% in performance on top of everything. Having a compiler to generate the final code makes these optimizations easy.

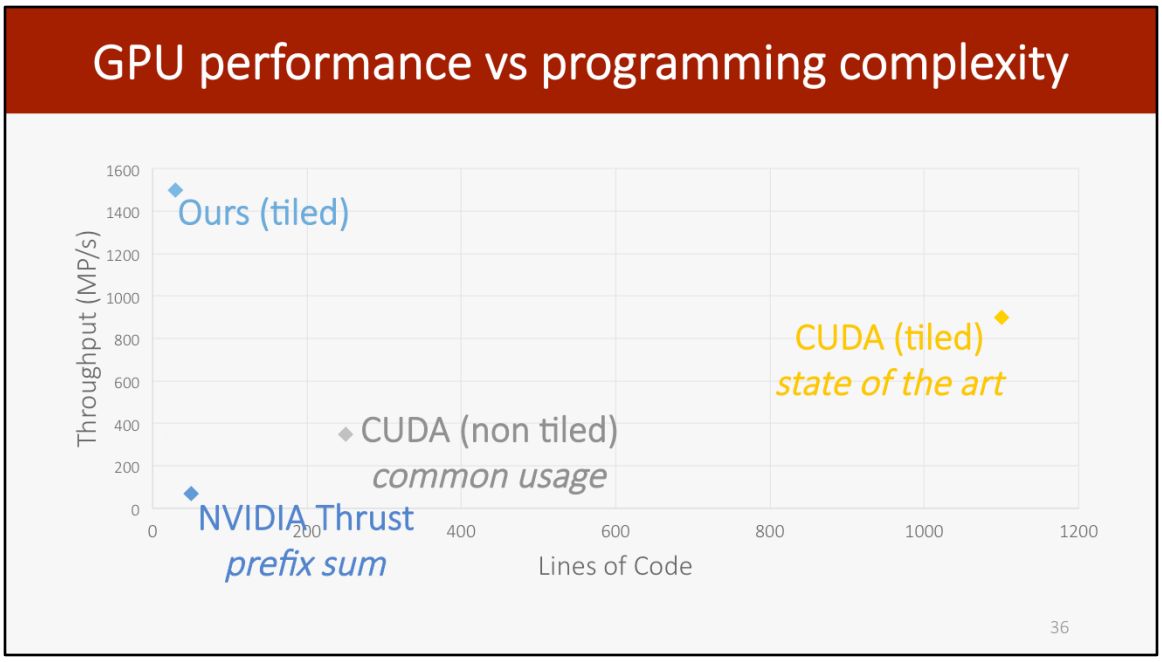
Difference of Gaussians



1.5x faster
90x less code
than tiled
CUDA

35

Difference of Gaussians shows even greater speedup due to optimization across the entire pipeline. This filter computes two Gaussians on the input image and then computes the difference to extract edges form the image. Here we compare two implementations both of which use tiled recursive filters. The difference is that we perform global optimizations in our approach e.g. merging successive stages of computation. This gives almost 1.5 times performance boost. These examples that a DSL can buy extra optimization by simply allowing programmers to experiment with more alternatives easily.



This graph shows all the options for writing recursive filters on GPUs. Here we can see throughput versus number of lines of code, which is a rough indicator of implementation effort.

The easiest is to use a precompiled lib like NVIDIA Thrust. This also gives the worst performance because it involves transposing 2D data and repeated read/write of the full image.

The next option is to implement them yourself in CUDA – this gets rid of transposing but it is still slow because of repeated image IO. This is still very doable, and also the most commonly used option. The third option is to use tiled recursive filters – this exploits the GPU in a much better fashion but the implementation effort is much higher. Our approach is presents the best tradeoff – the implementation effort is very low and we get the best performance because it gives tiled implementations and we can optimize across the whole pipeline.

Conclusions

Tiling & overlapping recursive filters gives up to 10x boost

Precompiled libraries compose into inefficient pipelines

DSL allows easy exploration of transformations

Generic tool for recursive filters, not just 2D images

Open source, free for commercial use

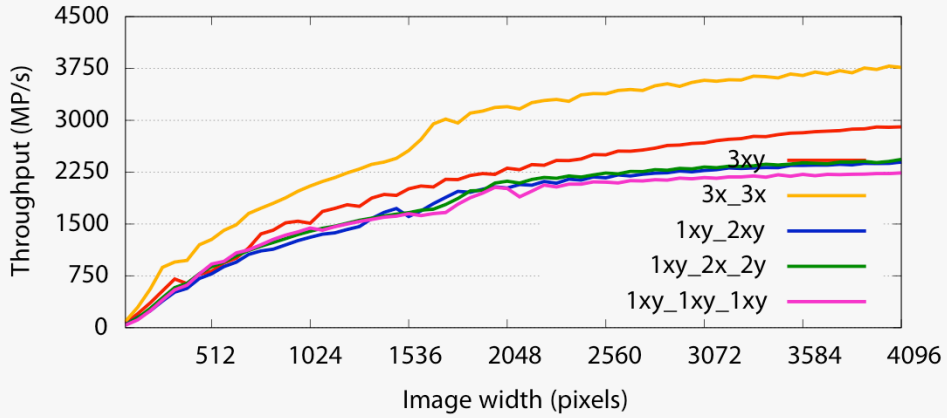
37

The take home message is that tiling transformations give an order of magnitude improvement in to performance. Precompiled libraries for image filters are generally not the optimal solution because they compose into inefficient pipelines. Our DSL makes it very easy to apply these transformations. Programmers can experiment with all possibilities and find the optimal implementation.
Lastly, we are not proposing a specific algorithm of a specific filter. Our work addresses the generic class of recursive filters. You can write arbitrary recursive filter pipelines on 1D, 2D or n-dimensional data for CPUs or GPUs. We give 4-5x speedup on 1D high- order filters on CPUs; currently there are no implementations that can achieve such speedups. We hope our unified approach will help rethink for recursive filters are written.

Thank you

http://mit-gfx.github.io/recfilter/

Performance impact of joint tiling combinations



39