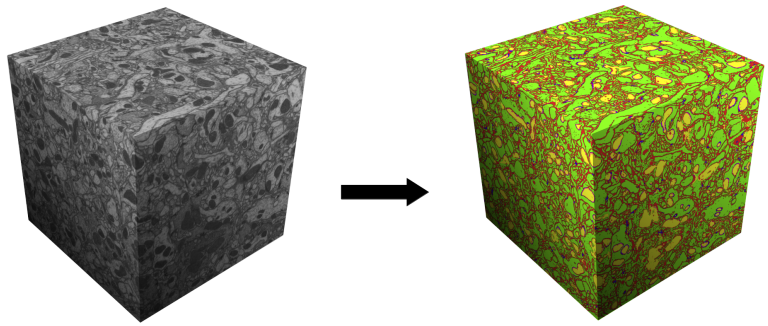
Infinite Impulse Response Filters

Judith Massa, Patrick Esser

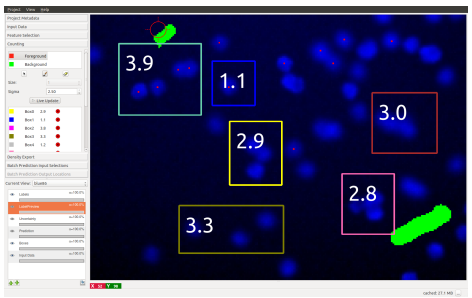
August 2, 2016

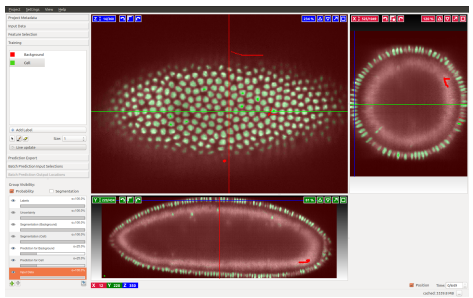
Overview

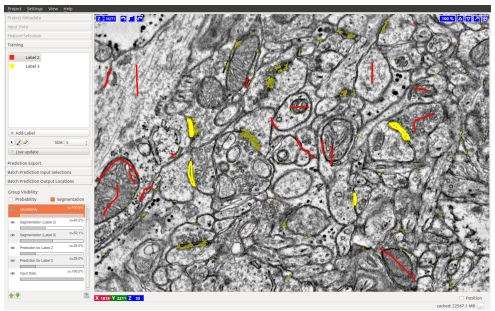
Ilastik



- Ilastik: toolkit for interactive image classification and segmentation
- Algorithms rely on precomputed features of the image



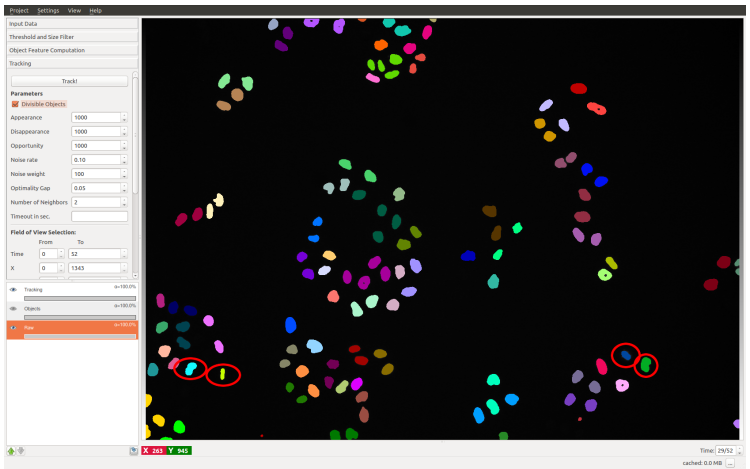


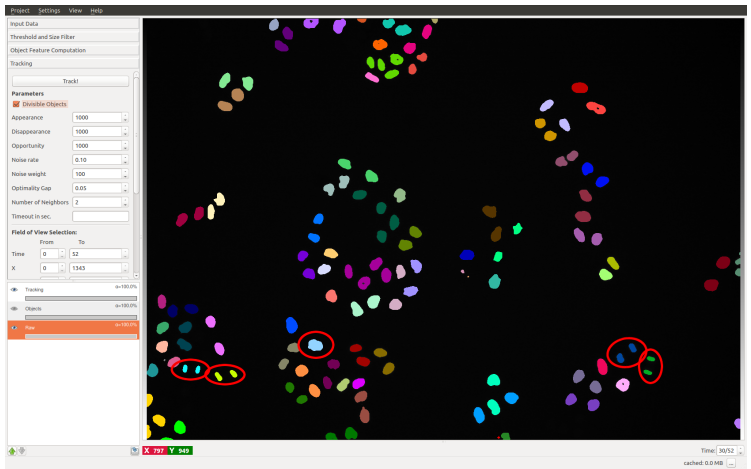


Tracking

Example:

- Project supervisor Sven Peter is working on object tracking
  $\rightarrow$ Requires classification scale space and edge detection





Gaussian Filtering

Convolution:

$$(f * g)(x) := \int_{\mathbb{R}^n} f(\tau) g(x - \tau) \mathrm{d}\tau$$

- discrete: $(f * g)[n] = \sum_{m=-\infty}^{\infty} f[n - m]\, g[m]$
- f ... image
- g ... signal, e.g. Gaussian

Gaussian:

- $g = \frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
- Convolution with Gaussian gives scale space representations
- Convolution with Gaussian derivative gives derivative of smoothed image (edge features)

Implementations of Filters

- Finite Impulse Response Filters (FIR):
  - if g finite:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[n-m]\, g[m] = \sum_{m=-M}^{M} f[n-m] g[m]$$

  - multiple muliply-adds for one pixel
- Fast Fourier Transform (FFT)
  - $\mathcal{F}(f)(t) = \frac{1}{(2\pi)^{\frac{n}{2}}} \int_{\mathbb{R}^n} f(x)\, e^{-\mathrm{i}t\cdot x}\, \mathrm{d}x$
  - $\mathcal{F}(f * g) = (2\pi)^{\frac{n}{2}}\, \mathcal{F}(f) \cdot \mathcal{F}(g)$
  - after transformation: 2 multiplications per pixel

Implementations of Filters

Problems:

- suffer from increasing stencils
  - e.g. wide Gaussian (needed for coarse-scale representation)
- FFT does not always outperform FIR on GPUs [FC06]

Alternative: Infinite Impulse Response Filters (IIR)

The idea

Instead of using possibly wide stencil
$\rightarrow$ Approximate by a fixed size stencil and recursion
$\rightarrow$ Recursion makes filter infinite:
all previous values taken into account

The consequences

- Approximation error: not too important here
  $\rightarrow$ result should be visually plausible.
- Fixed computational cost: does not depend on scale parameter
- Less parallelism than FIR: each row's column depends on previous column's result

Original Algorithm

- Deriche coefficients [Der93] $\rightarrow$ 2 passes: causal and anticausal

```
1         // causal pass
2         for (unsigned int i = 0; i < n_pixels; ++i) {
3             float sum = 0.0;
4
5             xtmp[0] = cur_line[i];
6             for (unsigned int j = 0; j < 4; ++j)
7                 sum += coefs.n_causal[j] * xtmp[j];
8             for (unsigned int j = 0; j < 4; ++j)
9                 sum -= coefs.d[j] * ytmp[j];
10            for (unsigned int j = 3; j > 0; --j) {
11                xtmp[j] = xtmp[j - 1];
12                ytmp[j] = ytmp[j - 1];
13            }
14
15            tmpbfr[i] = sum;
16            ytmp[0] = sum;
17        }
```

- $y_i = [x_i,\ x_{i-1},\ x_{i-2},\ x_{i-3}] \cdot \mathbf{c} - [y_{i-1},\ y_{i-2},\ y_{i-3},\ y_{i-4}] \cdot \mathbf{d}$
- anticausal pass and equivalent to causal

Original Algorithm - Border treatment

- mirroring: $[\ldots, x_2, x_1, x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \ldots]$

```
1          for (unsigned int i = 0; i < 4; ++i)
2              xtmp[i] = ytmp[i] = 0.0;
3
4          // left border
5          for (unsigned int i = 0; i < coefs.n_border; ++i) {
6              float sum = 0.0;
7
8              xtmp[0] = cur_line[coefs.n_border - i];
9              for (unsigned int j = 0; j < 4; ++j)
10                 sum += coefs.n_causal[j] * xtmp[j];
11             for (unsigned int j = 0; j < 4; ++j)
12                 sum -= coefs.d[j] * ytmp[j];
13             for (unsigned int j = 3; j > 0; --j) {
14                 xtmp[j] = xtmp[j - 1];
15                 ytmp[j] = ytmp[j - 1];
16             }
17
18             ytmp[0] = sum;
19         }
```

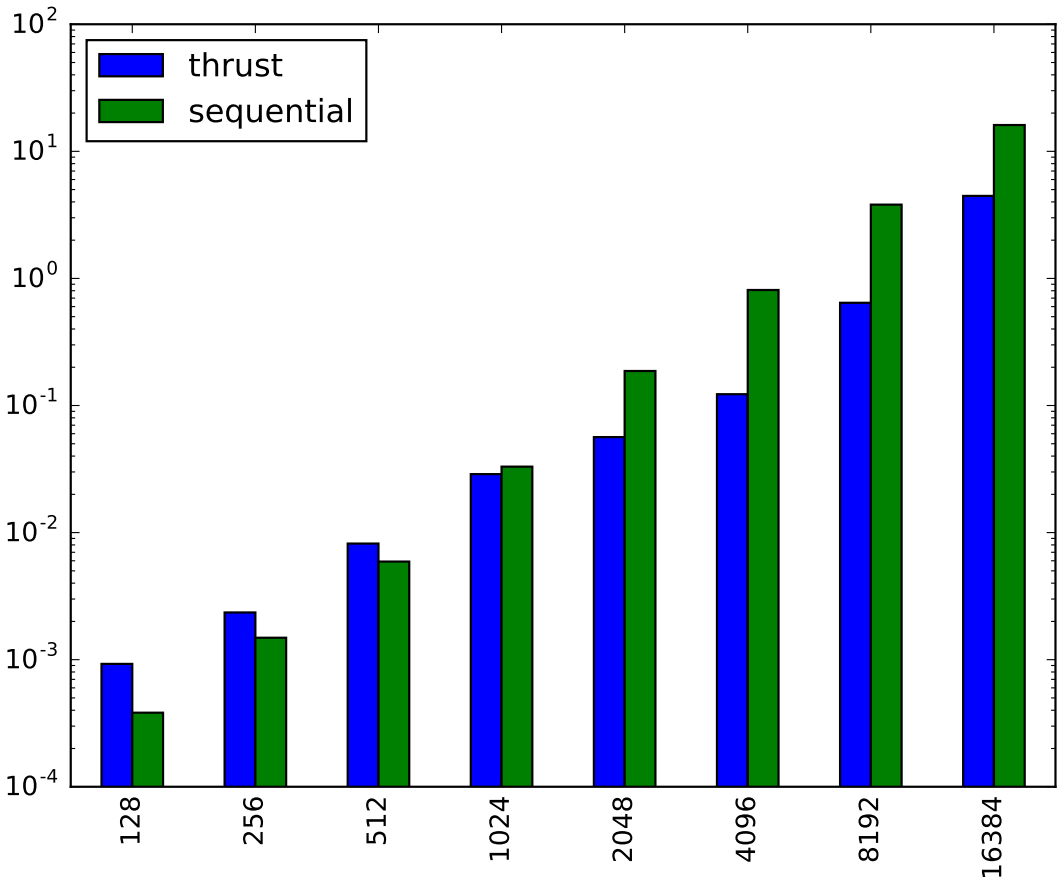
Possibilities and challenges

1. All rows (columns) independent
2. Causal and anticausal pass independent
3. Either horizontal or vertical pass will cause bad memory layout
4. Parallelize recurrence relation within each row
5. Compute multiple features for multiple images concurrently

Causal pass

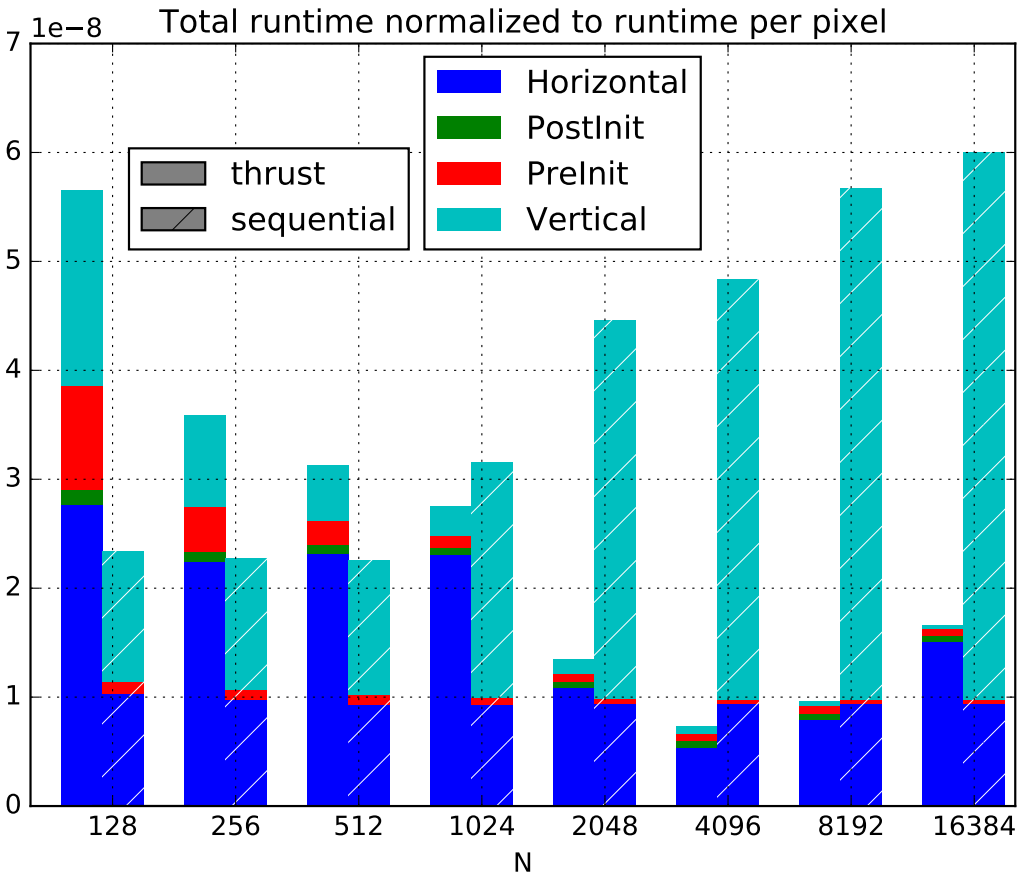
```
1  thrust::for_each_n(
2    thrust::counting_iterator<int>(0), height,
3    [buffer_begin, src_begin, row_stride, column_stride, width, c]
4    __device__ (int n) {
5      auto row = buffer_begin + n * row_stride;
6      auto src = src_begin + n * row_stride;
7      // init recursion ... then recurse
8      for(int i = 4; i < width; ++i)
9      {
10       row[i * column_stride] =
11         c.b_causal[0] * src[(i - 0) * column_stride]
12       + c.b_causal[1] * src[(i - 1) * column_stride]
13       + c.b_causal[2] * src[(i - 2) * column_stride]
14       + c.b_causal[3] * src[(i - 3) * column_stride]
15       - c.a[1] * row[(i - 1) * column_stride]
16       - c.a[2] * row[(i - 2) * column_stride]
17       - c.a[3] * row[(i - 3) * column_stride]
18       - c.a[4] * row[(i - 4) * column_stride];
19     }
20   });
```

Results



- code handles horizontal and vertical passes by adjusting strides
- results do not look very promising

A closer look



- Initialization and data transfer not limiting
- Sequential version's performance degrades for the vertical pass - stride causes cache misses
- Thrust version is limited by performance of horizontal pass - results in non-coalesced memory access (consecutive threads access strided data)

Streams

Just run them on two different streams!

```
1 thrust::for_each_n(
2   thrust::cuda::par.on(stream),
3   thrust::counting_iterator<int>(0), height,
4   ...
```

Improves performance but horizontal pass is still the bottleneck.

Reducing causality passes

- Synchronize streams
- Add causal and anticausal buffers into result

Remember: Anticausal pass goes from right to left - yet the original implementation writes into the buffer from left to right:

```
1  for(int i = 4; i < width; ++i) {
2    row[i * column_stride] =
3      c.b_anticausal[1] * src[((width - 1) - i + 1) * column_stride]
4    + c.b_anticausal[2] * src[((width - 1) - i + 2) * column_stride]
5    + c.b_anticausal[3] * src[((width - 1) - i + 3) * column_stride]
6    + c.b_anticausal[4] * src[((width - 1) - i + 4) * column_stride]
7    - c.a[1] * row[(i - 1) * column_stride]
8    - c.a[2] * row[(i - 2) * column_stride]
9    - c.a[3] * row[(i - 3) * column_stride]
10   - c.a[4] * row[(i - 4) * column_stride];
11 }
```

Reducing causality passes

$\Longrightarrow$ need to reverse anticausal buffer in each row:

```
cudaStreamSynchronize(s1);
cudaStreamSynchronize(s2);
thrust::for_each_n(
  thrust::counting_iterator<int>(0), height,
  [...] __device__ (int n) {
    auto dest = dest_begin + n * row_stride;
    auto row_l = buffer_l_begin + n * row_stride;
    auto row_r = buffer_r_begin + n * row_stride;
    for(int i = 0; i < width; ++i) {
      dest[i * column_stride] =
      row_l[i * column_stride] + row_r[(width - 1 - i) * column_stride];
    }
  });
```

Instead, write buffer from right to left and perform fully parallel summation:

```
thrust::transform(
  buffer_l_begin, buffer_l_begin + width * height,
  buffer_r_begin,
  dest_begin,
  thrust::plus<T>());
```

Idea

1. Transpose
2. Vertical pass
3. Transpose
4. Horizontal pass

But:

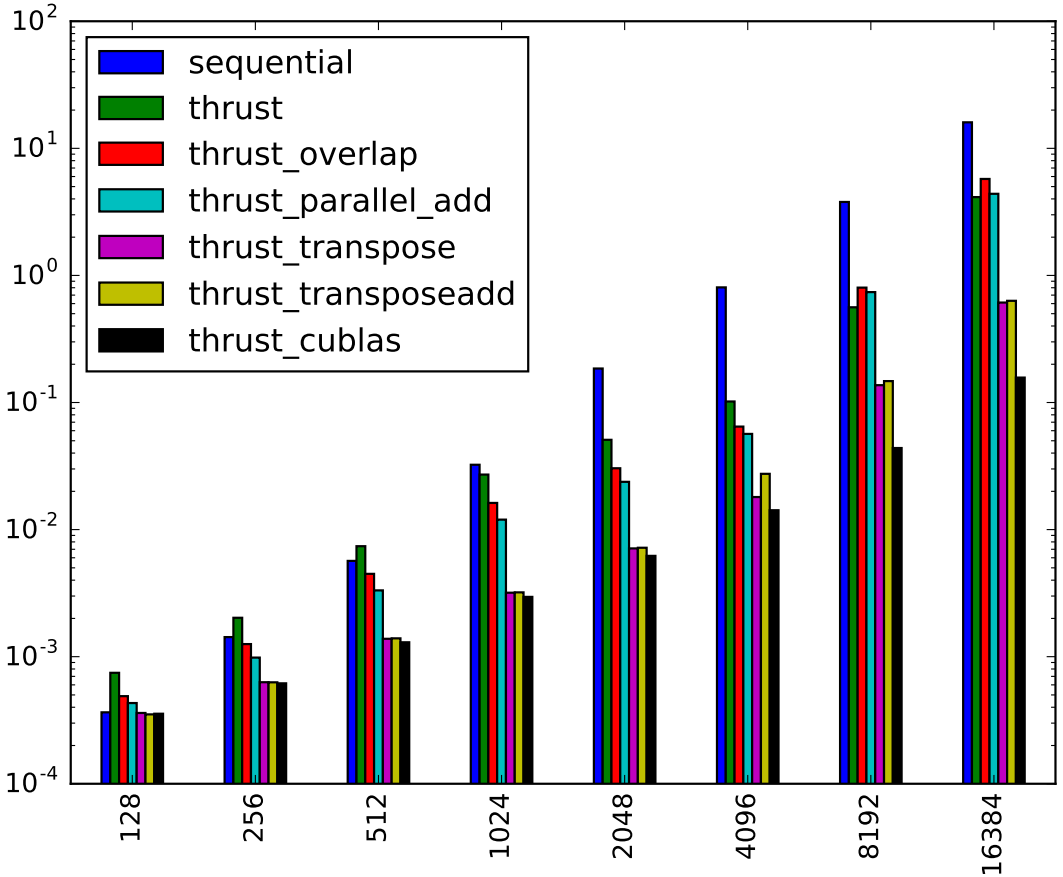
- Two additional transpose operations which are non-trivial to implement efficiently on GPUs
- Additional buffer for transpose operation required

Even better

- Even with a simple thrust implementation this achieves speedups
- We can avoid the additional buffer by performing the transposition during the addition of the causality buffers ($\Longrightarrow$ perform vertical pass first, i.e. work on mirror image or assume column major format)
- Exactly this, adding two transposed matrices, is also implemented in cublas!

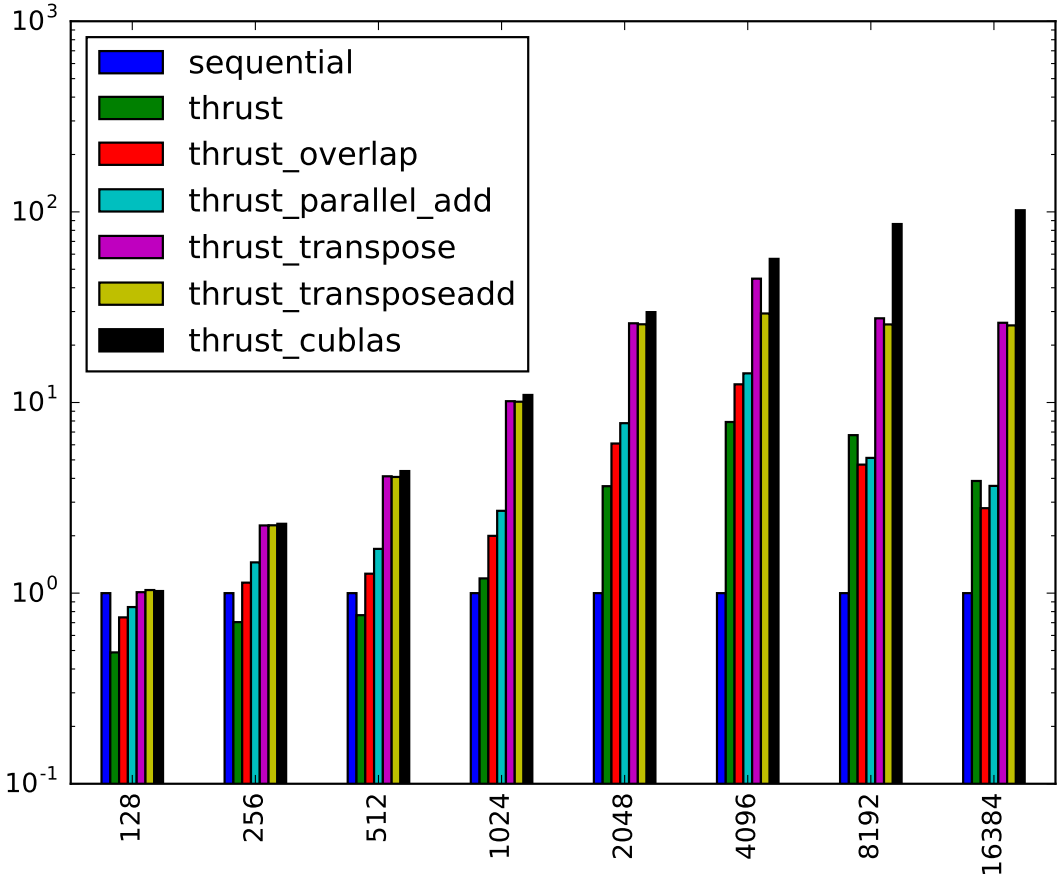
```
1 T* buffer_l_ptr  = thrust::raw_pointer_cast(&*buffer_l_begin);
2 T* buffer_r_ptr  = thrust::raw_pointer_cast(&*buffer_r_begin);
3 T* dest_ptr = thrust::raw_pointer_cast(&*dest_begin);
4 T alpha = beta = 1.;
5 cublasSgeam(
6   *handle, CUBLAS_OP_T, CUBLAS_OP_T, height, width,
7   &alpha, buffer_l_ptr, width,
8   &beta, buffer_r_ptr, width,
9   dest_ptr, height);
```

Results - Times



- Improving memory access most important factor
- Optimized cuBLAS geam gives additional boost for larger data

Results - Speedups



- Maximum speedup of 100
- For $1024x1024$ images: Speedup of 11 without transfers and 7 with.
- 360 FPS @ $1024x1024$

Further considerations

Different data and applications need different optimizations. Here: Image sequences of approximately 1024x1024 pixels.

- Parallelization of higher order recurrence relations is more complicated (because prefix sum operators are required to be associative for parallelization) [Ble90]
- It must be evaluated whether a parallelization of the recurrence relation within rows pays off for the relatively small number of 1024 elements.
- Multiple features can be calculated concurrently.
- Features of multiple images (from a video sequence) can be calculated concurrently.

Alternatives

- Evaluate for which parameters FIR and FFT perform better/worse.
- Alternative parallelization strategy: blocked parallelism.

References


Guy E Blelloch, *Prefix sums and their applications*.


Rachid Deriche, *Recursively implementating the Gaussian and its derivatives*, Research Report RR-1893, INRIA, 1993.


O. Fialka and M. Cadik, *Fft and convolution performance in image filtering on gpu*, Tenth International Conference on Information Visualisation (IV'06), July 2006, pp. 609–614.