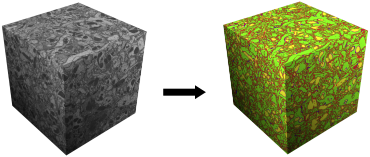
Infinite Impulse Response Filters

Judith Massa, Patrick Esser

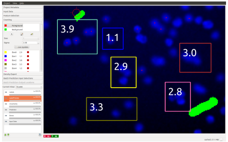
August 8, 2016

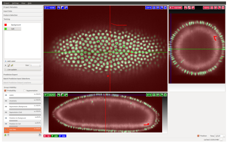
Overview

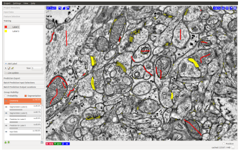
Ilastik



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

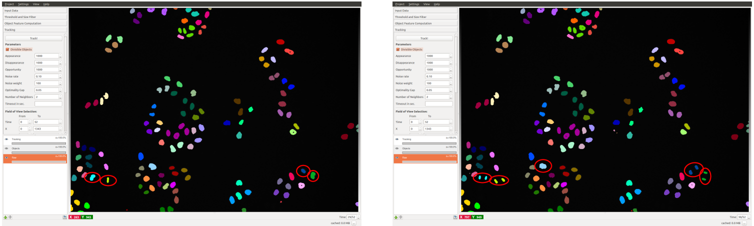






Tracking

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



Application in Biology: Cell Tracking

Gaussian Filtering

Convolution:

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

- discrete: $(f * g)(x) = \sum_{y=-\infty}^{\infty} f(x-y)\, g(y)$
- f ... image
- g ... filter, 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 (with $2M+1$ entries):

$$(f*g)(x) = \sum_{y=-\infty}^{\infty} f(x-y)\,g(y) = \sum_{y=-M}^{M} f(x-y)g(y)$$

  - multiple muliply-adds for one pixel
- Fast Fourier Transform (FFT)
  - $\mathcal{F}(f)(\omega) = \int_{\mathbb{R}^n} f(x)\,e^{-\mathrm{i}\omega x 2\pi}\,\mathrm{d}x$
  - discrete and finite: $\mathcal{F}(f)(\omega) = \sum_{x=0}^{N-1} f(x)\cdot e^{-\mathrm{i}\omega x \frac{2\pi}{N}}$
    $f(x) = \frac{1}{N}\sum_{\omega=0}^{N-1} \mathcal{F}(f)(\omega)\cdot e^{-\mathrm{i}\omega x \frac{2\pi}{N}}$ ($N\ldots$ number of pixels)

$$(f*g)(x) = \left(\mathcal{F}^{-1}\mathcal{F}(f*g)\right)(x) = (\mathcal{F}^{-1}[\underbrace{\mathcal{F}(g)}_{\text{precomputed}} \cdot \mathcal{F}(f)])(x)$$

  - after transformation 1 multiplication 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)

IIR Filters

Idea:

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

$$y(i) = \sum_{k=0}^{N-1} n(i)x(i-k) - \sum_{k=1}^{D} d(k)y(i-k)$$

Properties

- Approximate RMS error of IIR-filters: 10^{-4}
  $\rightarrow$ Implementations may vary approximately to this extent
  $\rightarrow$ single precision sufficient
- Computational cost does not depend on scale parameter
  $\rightarrow$ Cost is fixed, no filter comparison necessary
- Each row's column depends on previous column's result
  $\rightarrow$ Less parallelism than FIR

Causal and Anticausal pass

- Approximate a one dimensional, causal filter, i.e.

$$y_i = \sum_{k=0}^{N-1} h_k x_{i-k} \tag{1}$$

- Split a one dimensional non causal filter,

$$y_i = \sum_{k=-M}^{M} h_k x_{i-k} \tag{2}$$

into a causal and anticausal filter, i.e.

$$h_k^+ = \begin{cases} h_k & k \geq 0 \\ 0 & k < 0 \end{cases}, \quad h_k^- = \begin{cases} 0 & k \geq 0 \\ h_k & k < 0 \end{cases} \tag{3}$$

Horizontal and Vertical pass

- Approximate those two (anti-)causal filters $\rightarrow$ Causal (left to right) and Anticausal (right to left) passes
- Extend to two dimensions: One filter per row (Horizontal pass) and one filter per column (Vertical pass)

$\rightarrow$ total of 4 passes: Left-to-right (causal) and right-to-left (anticausal) in each row (horizontal) and top-to-bottom (causal) and bottom-to-top (anticausal) in each column (vertical). Details in [Der93].

Original Algorithm

```
        // causal pass
        for (unsigned int i = 0; i < n_pixels; ++i) {
            float sum = 0.0;

            xtmp[0] = cur_line[i];
            for (unsigned int j = 0; j < 4; ++j)
                sum += coefs.n_causal[j] * xtmp[j];
            for (unsigned int j = 0; j < 4; ++j)
                sum -= coefs.d[j] * ytmp[j];
            for (unsigned int j = 3; j > 0; --j) {
                xtmp[j] = xtmp[j - 1];
                ytmp[j] = ytmp[j - 1];
            }

            tmpbfr[i] = sum;
            ytmp[0] = sum;
        }
```

- $y_i = [x_i,\ x_{i-1},\ x_{i-2},\ x_{i-3}] \cdot \mathbf{n} - [y_{i-1},\ y_{i-2},\ y_{i-3},\ y_{i-4}] \cdot \mathbf{d}$
- Anticausal pass similar to causal pass

Border Treatment

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

```
for (unsigned int i = 0; i < 4; ++i)
    xtmp[i] = ytmp[i] = 0.0;

// left border
for (unsigned int i = 0; i < coefs.n_border; ++i) {
    float sum = 0.0;

    xtmp[0] = cur_line[coefs.n_border - i];
    for (unsigned int j = 0; j < 4; ++j)
        sum += coefs.n_causal[j] * xtmp[j];
    for (unsigned int j = 0; j < 4; ++j)
        sum -= coefs.d[j] * ytmp[j];
    for (unsigned int j = 3; j > 0; --j) {
        xtmp[j] = xtmp[j - 1];
        ytmp[j] = ytmp[j - 1];
    }

    ytmp[0] = sum;
}
```

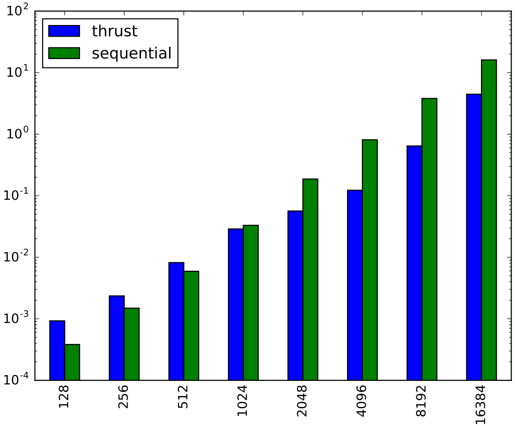
Possibilities and challenges

1. All rows (columns) independent in horizontal (vertical) pass
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

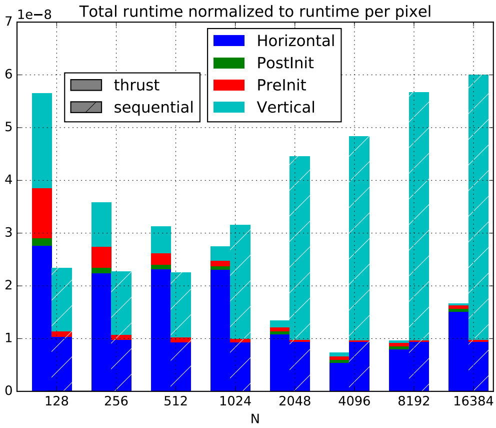
```
thrust::for_each_n(
  thrust::counting_iterator<int>(0), height,
  [buffer_begin, src_begin, row_stride, column_stride, width, c]
  __device__ (int n) {
    auto row = buffer_begin + n * row_stride;
    auto src = src_begin + n * row_stride;
    // init recursion ... then recurse
    for(int i = 4; i < width; ++i)
    {
      row[i * column_stride] =
        c.b_causal[0] * src[(i - 0) * column_stride]
      + c.b_causal[1] * src[(i - 1) * column_stride]
      + c.b_causal[2] * src[(i - 2) * column_stride]
      + c.b_causal[3] * src[(i - 3) * column_stride]
      - c.a[1] * row[(i - 1) * column_stride]
      - c.a[2] * row[(i - 2) * column_stride]
      - c.a[3] * row[(i - 3) * column_stride]
      - c.a[4] * row[(i - 4) * column_stride];
    }
  });
```

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)

Parallelizing causal and anti-causal pass

Just run them on two different streams!

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

Improves performance but horizontal pass is still the bottleneck.

Combining 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:

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

Combining 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>());
```

Horizontal and Vertical memory access patterns

Horizontal pass:

- Parallelized over rows
- Two threads working on consecutive rows access strided data
- $\rightarrow$ wasted memory bandwidth

Vertical pass:

- Parallelized over columns
- Two threads working on consecutive columns access consecutive data
- $\rightarrow$ fully coalesced memory access

Idea to improve access pattern

Transpose data instead of performing horizontal pass.

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

But:

- Two additional transpose operations which are non-trivial to implement efficiently on GPUs
- Additional buffer for transpose operation required
- Assumes either column-major storage or it works on image mirrored along diagonal

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
- Exactly this, adding two transposed matrices, is also implemented in cublas!

```
T* buffer_l_ptr  = thrust::raw_pointer_cast(&*buffer_l_begin);
T* buffer_r_ptr  = thrust::raw_pointer_cast(&*buffer_r_begin);
T* dest_ptr = thrust::raw_pointer_cast(&*dest_begin);
T alpha = beta = 1.;
cublasSgeam(
  *handle, CUBLAS_OP_T, CUBLAS_OP_T, height, width,
  &alpha, buffer_l_ptr, width,
  &beta, buffer_r_ptr, width,
   dest_ptr, height);
```

Performance evaluation

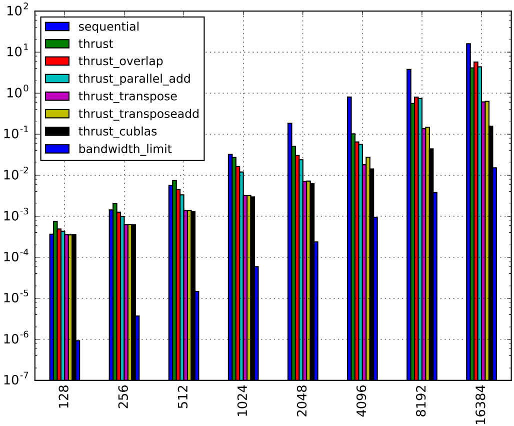
The following evaluations were performed on a:

- NVIDIA GeForce GTX 980M
- Compute Capability 5.2
- Bandwidth: 160 GB/s

and a:

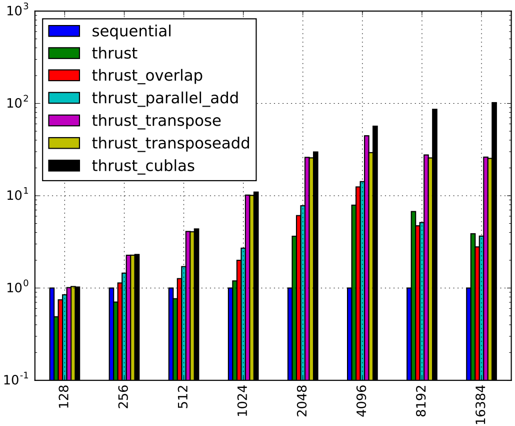
- Intel Core i7-4810MQ @ 2.8 Ghz
- Boost up to 3.8 Ghz

Times



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

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.
- Thrust implementation can also run on different backends - e.g. OpenMP

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.