SIFT in CUDA

In the Horizon 2020 projects POPART and LADIO

project page: http://www.popartproject.eu http://ladioproject.eu

PopSift repository: https://github.com/alicevision/popsift

S - Scale

I - Invariant

F - Feature

T - Transform

Distinctive Image Features from Scale-Invariant Keypoints

David G. Lowe
Computer Science Department
University of British Columbia
Vancouver, B.C., Canada
lowe@cs.ubc.ca

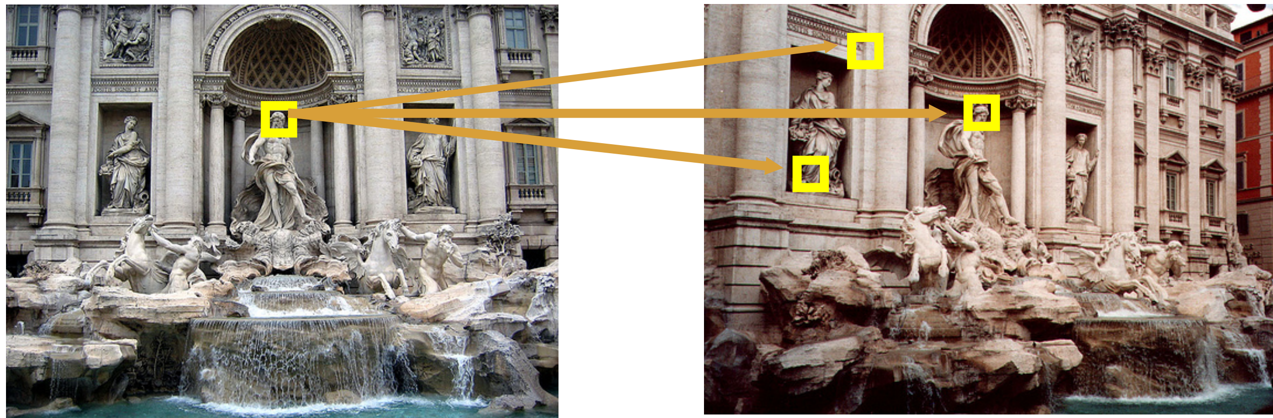
January 5, 2004

Abstract

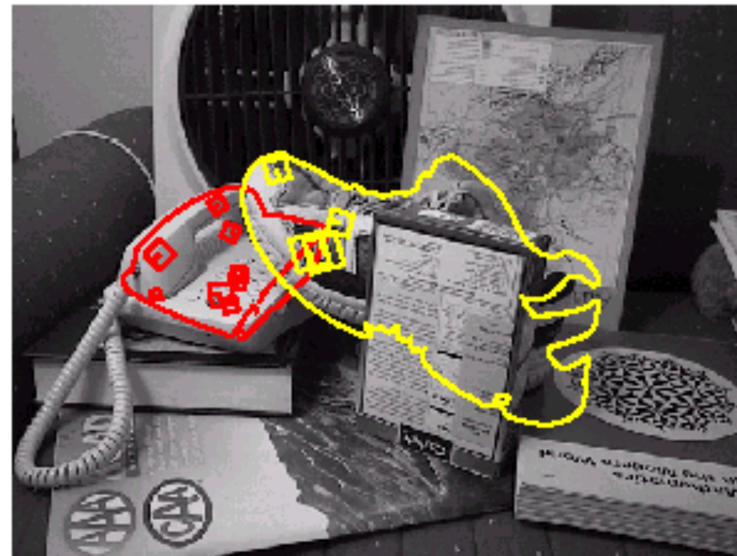
This paper presents a method for extracting distinctive invariant features from images that can be used to perform reliable matching between different views of an object or scene. The features are invariant to image scale and rotation, and are shown to provide robust matching across a a substantial range of affine distortion, change in 3D viewpoint, addition of noise, and change in illumination. The features are highly distinctive, in the sense that a single feature can be correctly matched with high probability against a large database of features from many images. This paper also describes an approach to using these features for object recognition. The recognition proceeds by matching individual features to a database of features from known objects using a fast nearest-neighbor algorithm, followed by a Hough transform to identify clusters belonging to a single object, and finally performing verification through least-squares solution for consistent pose parameters. This approach to recognition can robustly identify objects among clutter and occlusion while achieving near real-time performance.

Matching SIFT features

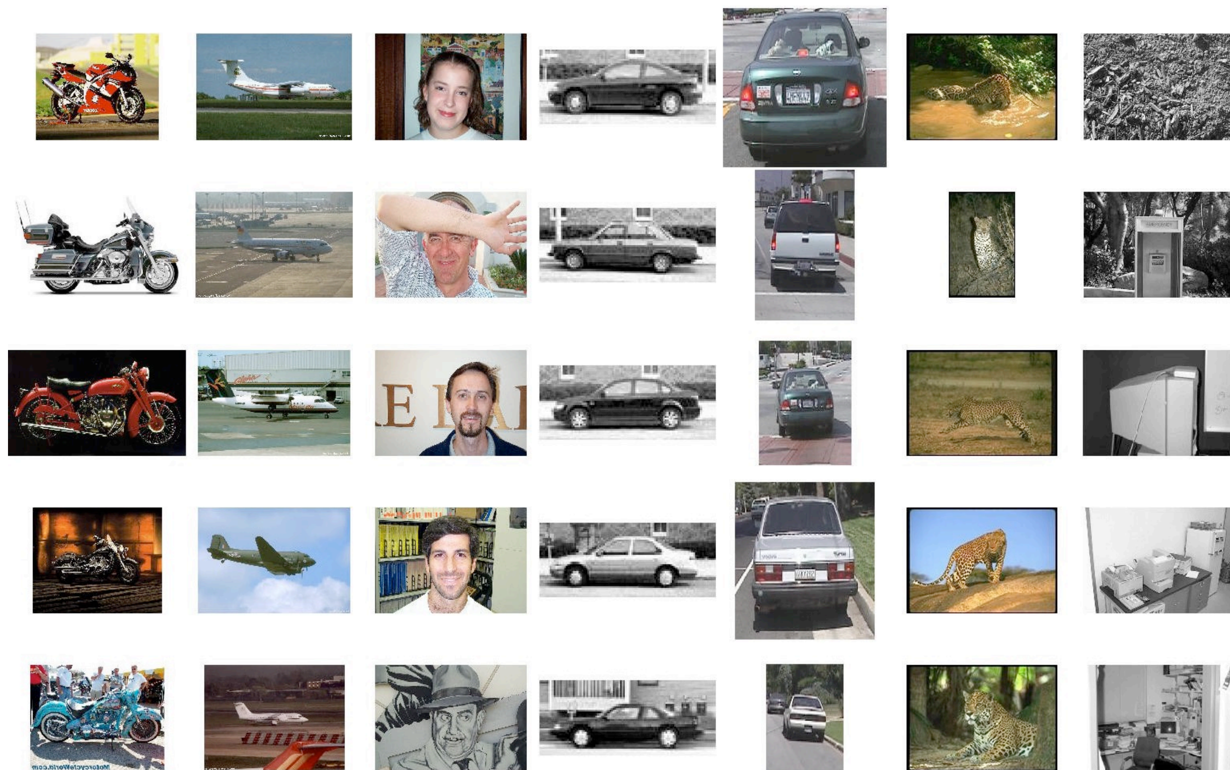
- Given a feature in I_1, how to find the best match in I_2?
  1. Define distance function that compares two descriptors
  2. Test all the features in I_2, find the one with min distance



Object Recognition



Object Categorization



Location recognition

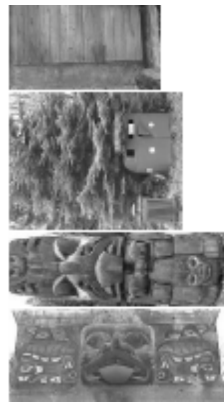


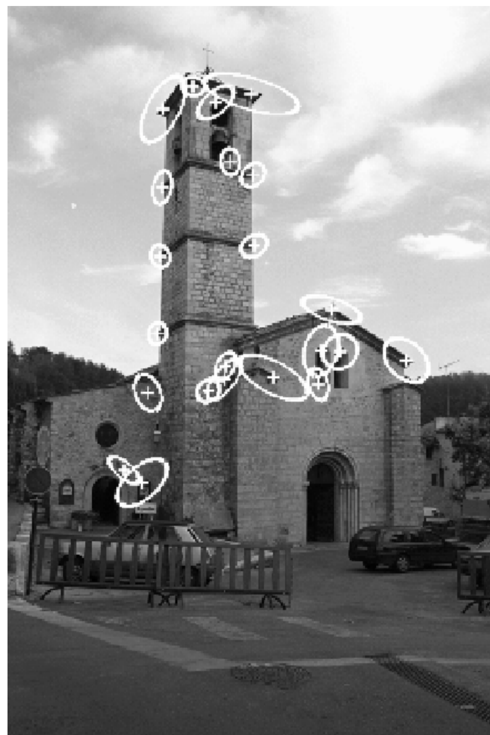
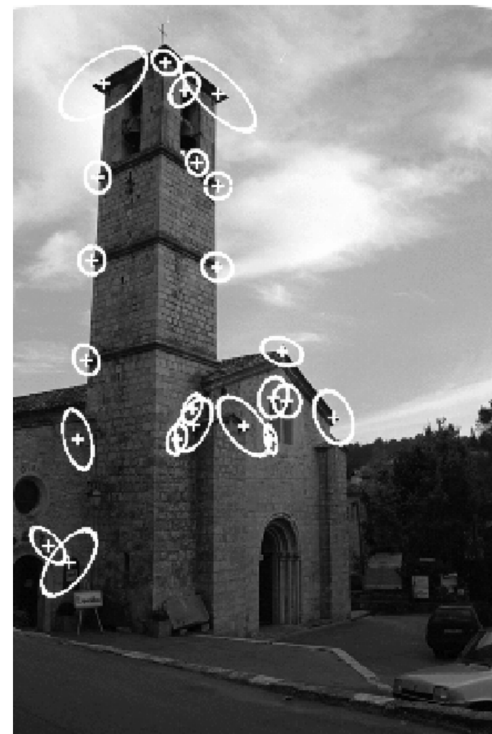
Image retrieval

change in viewing angle

• • •

> 5000 images

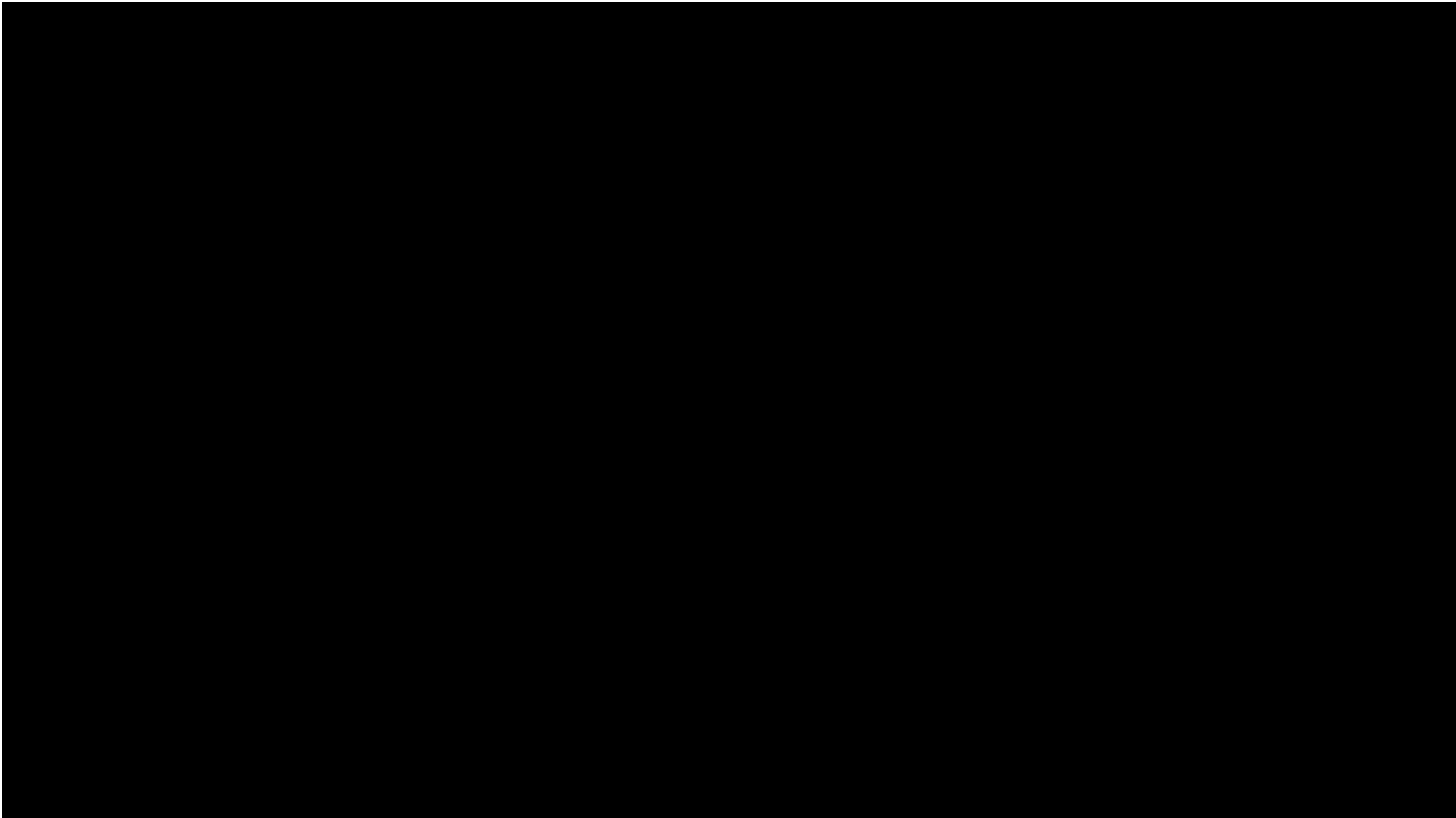
Matches

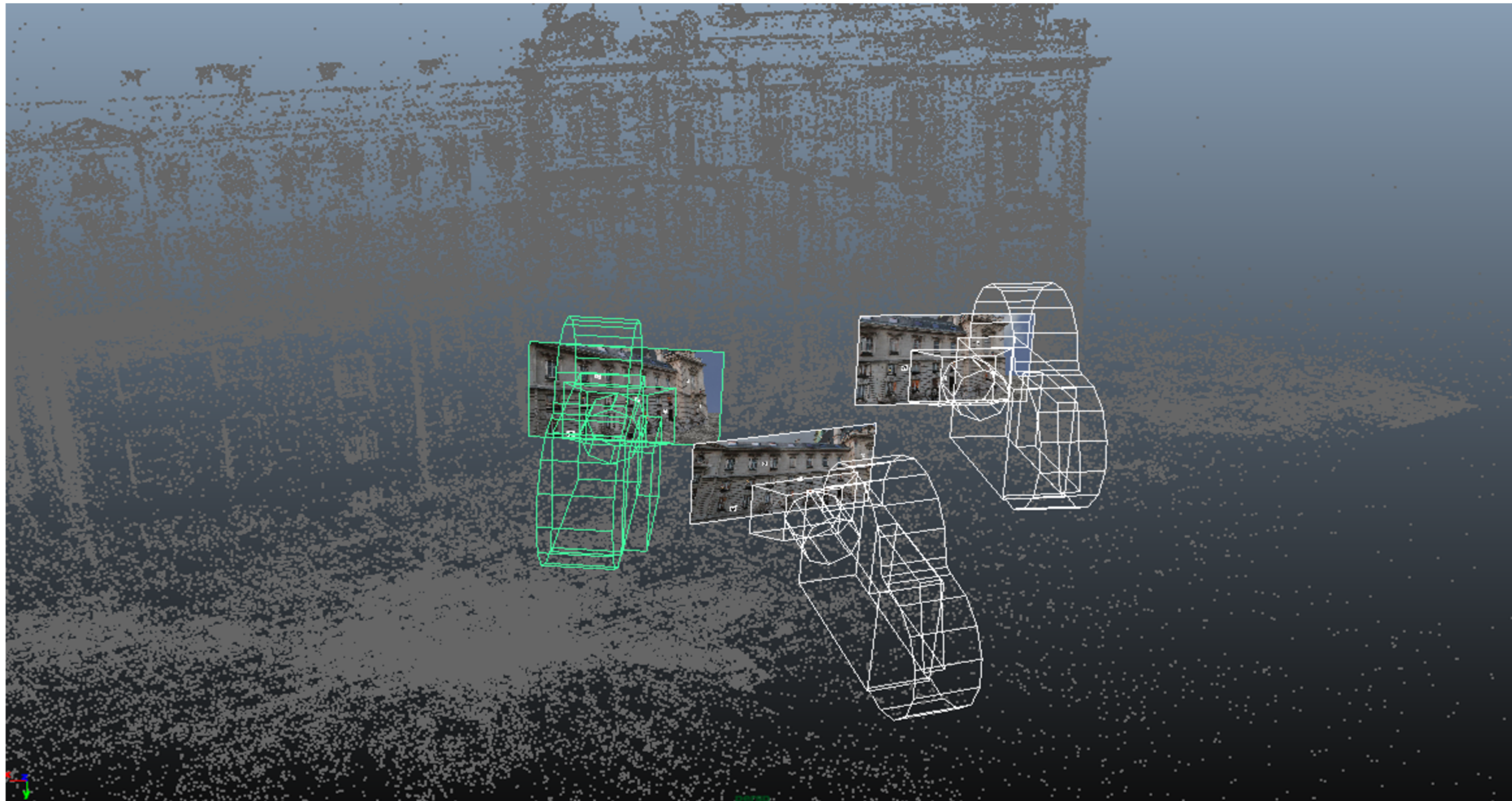
22 correct matches

#

3D reconstruction



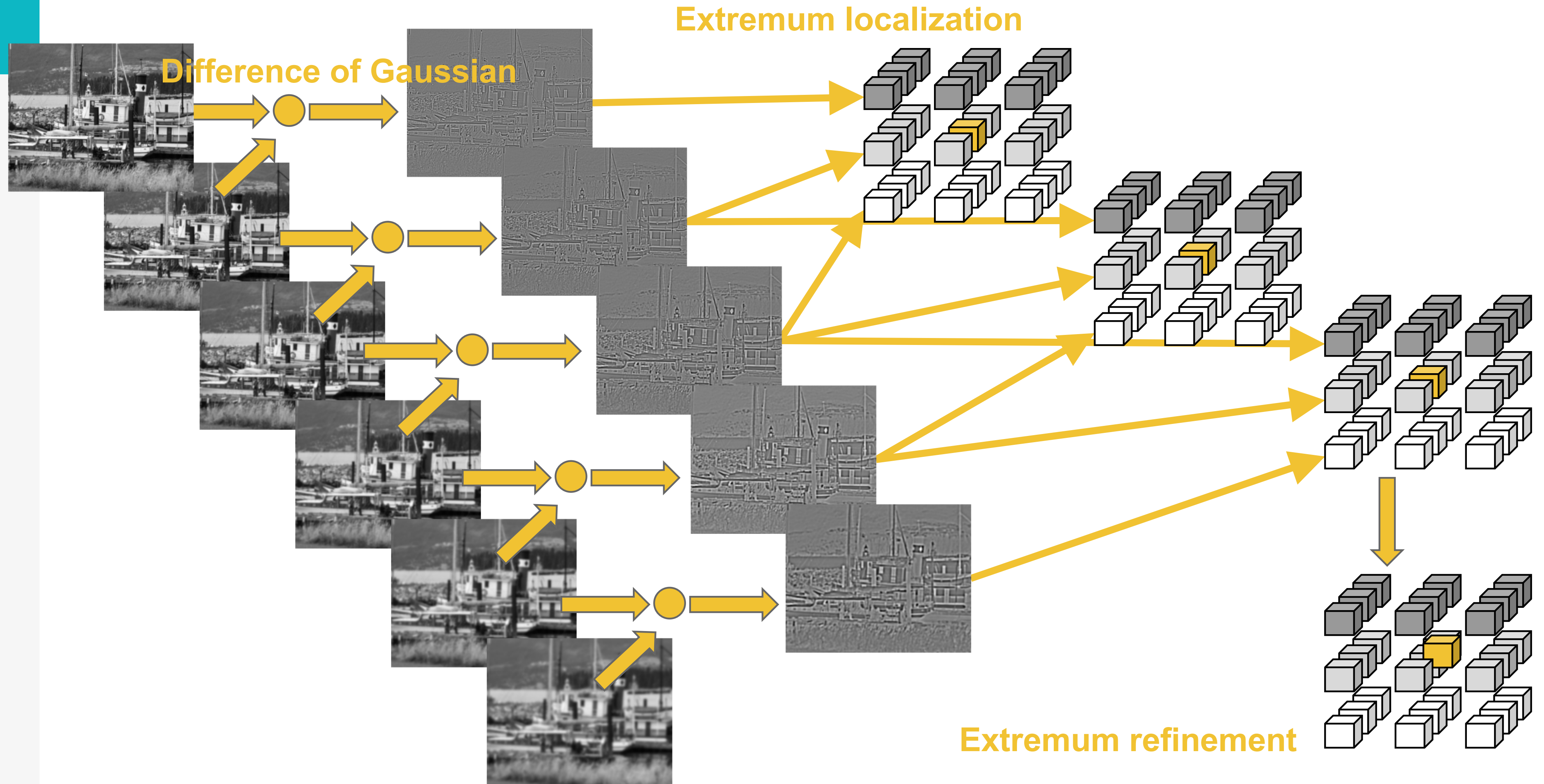
Camera tracking



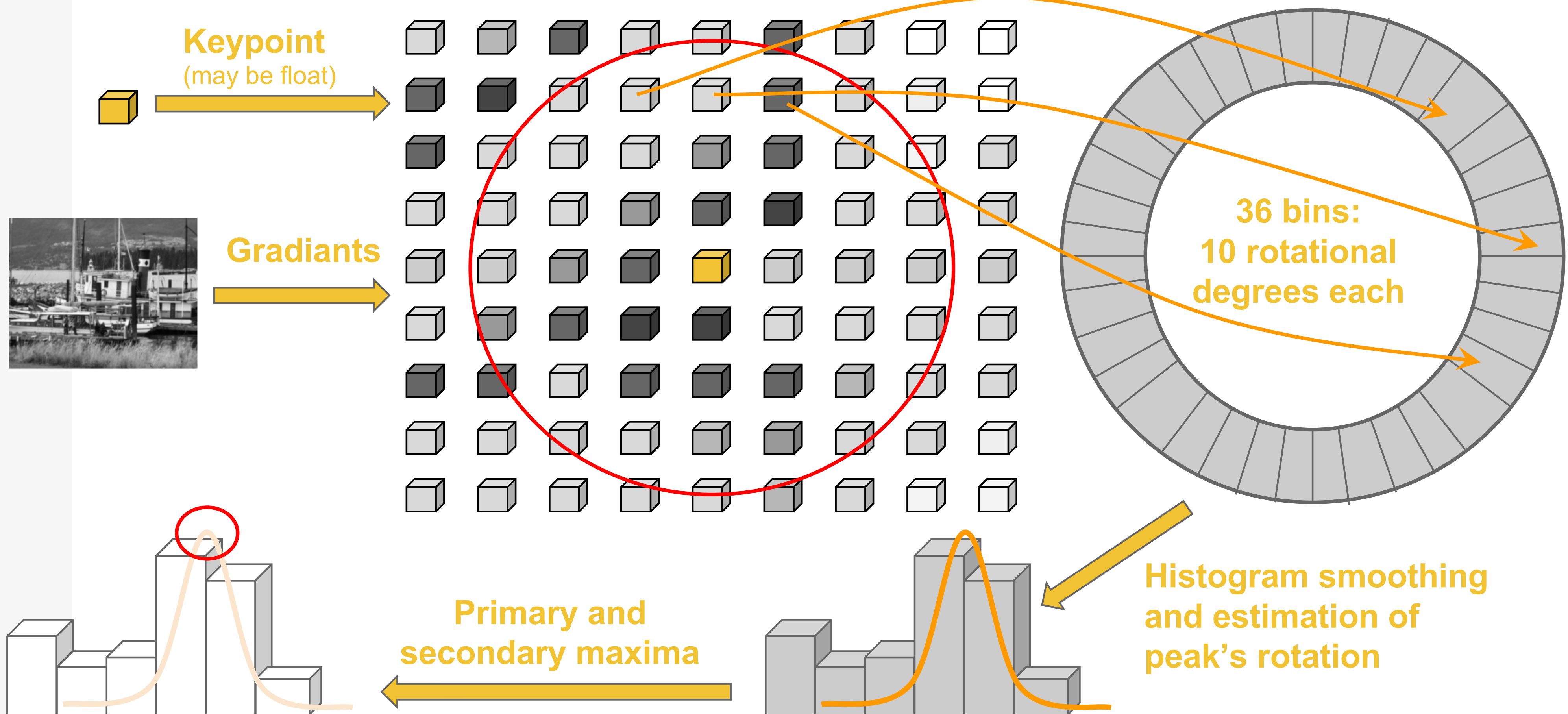
SIFT matching illustration

Overview of steps

Overview of steps

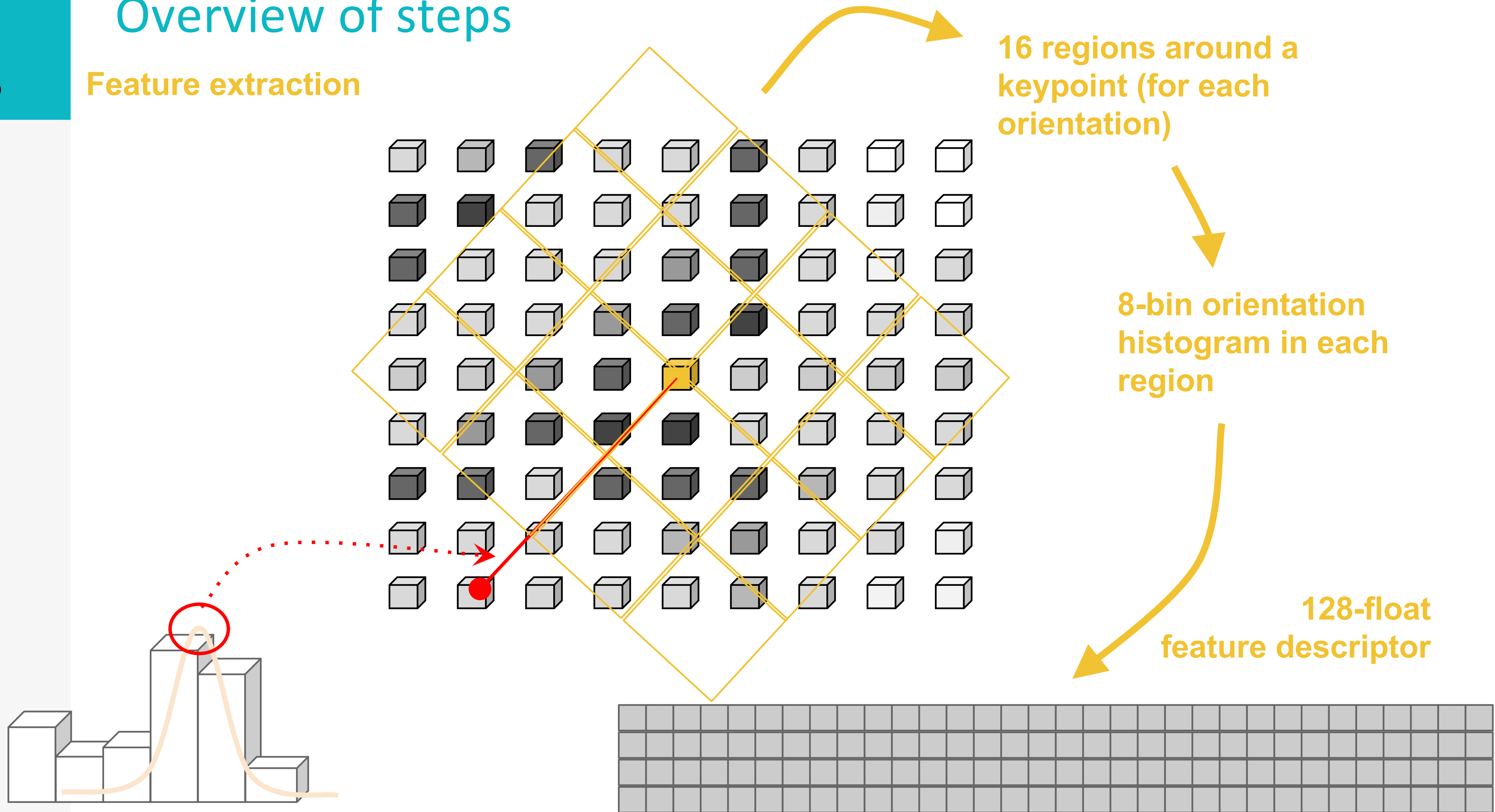


Overview of steps



Overview of steps

Feature extraction



CUDA tuning

Released under MPLv2

Feature vector output is compatible with VLFeat (drop-in replacement)

Arbitrary input scaling to trade speed for accuracy

Single-stream real-time extraction for 1920x1080 with upscaling on the GTX 980 Ti

Output value scaling by powers of 2

Very similar in terms of extracted and matched features to VLFeat (and better than OpenCV)

Streams for parallel kernels

Pinned memory for DMA transfer

CUDA texture engine

- uchar-float conversion
- bilinear interpolation
- scale-independent addressing
- automatic padding

CUDA “constant” for Gaussian filter parameters, edge threshold, contrast threshold ...

Reduce multiplications by using filter symmetry

#

Create pyramid

Input image: `struct ImageBase` in https://github.com/alicevision/popsift/blob/develop/src/popsift/s_image.h line 110

Pyramid construction in `Pyramid::build_pyramid`
https://github.com/alicevision/popsift/blob/develop/src/popsift/s_pyramid_build.cu line 460
case `conf.getGaussMode() == Config::VLFeat_Relative`
line 517

for the first image in the first octave, this call
`Pyramid::horiz_from_input_image`
which is in line 97

#

Creating image texture

```
struct Plane2D
{ …
  uint8_t* data;
  int rows, cols, pitchInBytes;
… };

struct Image
{ …
  Plane2D _image_d;
  cudaTextureObject_t _image_tex;
  cudaTextureDesc _texDesc;
  cudaResourceDesc _image_resDesc;
… };
```

```
void Image::createTexture()
{
  memset(&_texDesc,0,
    sizeof(cudaTextureDesc) );
  _texDesc.normalizedCoords = 1;
  _texDesc.addressMode[0] =
  _texDesc.addressMode[1] =
  _texDesc.addressMode[2] =
    cudaAddressModeClamp;
  _texDesc.readMode =
    cudaReadModeNormalizedFloat;
  _texDesc.filterMode =
    cudaFilterModeLinear;

…
```

Creating image texture

```
struct Plane2D
{ …
  uint8_t* data;
  int rows, cols, pitchInBytes;
… };

struct Image
{ …
  Plane2D _image_d;
  cudaTextureObject_t _image_tex;
  cudaTextureDesc _texDesc;
  cudaResourceDesc _resDesc;
… };
```

```
void Image::createTexture()
{
  …
  memset(&_resDesc, 0,
    sizeof(cudaResourceDesc) );
  _resDesc.resType=cudaResourceTypePitch2D;
  _resDesc.res.pitch2D.devPtr=_image_d.data;
  _resDesc.res.pitch2D.desc.f=
    cudaChannelFormatKindUnsigned;
  _resDesc.res.pitch2D.desc.x=8;
  _resDesc.res.pitch2D.desc.y=
  _resDesc.res.pitch2D.desc.z=
  _resDesc.res.pitch2D.desc.w=0;
  _resDesc.res.pitch2D.pitchInBytes=
    _image_d.pitchInBytes;
  _resDesc.res.pitch2D.width=_image_d.cols;
  _resDesc.res.pitch2D.height=_image_d.rows;
  …
```

Creating image texture

```
struct Plane2D
{ …
  uint8_t* data;
  int rows, cols, pitchInBytes;
… };

struct Image
{ …
  Plane2D _image_d;
  cudaTextureObject_t _tex;
  cudaTextureDesc _texDesc;
  cudaResourceDesc _resDesc;
… };
```

```
void Image::createTexture()
{ …
  cudaError_t err;
  err = cudaCreateTextureObject(
    &_tex,
    &_resDesc,
    &_texDesc, 0 );
 POP_CUDA_FATAL_TEST(err,
    "Could not create texture object: ");
}
```

Initialize the Gaussian filter

```
template<int LEVELS>
struct GaussTable
{
  float filter[LEVELS*GAUSS_ALIGN];
  float sigma [LEVELS];
  int   span  [LEVELS];
};

__device__ __constant__
GaussInfo d_gauss;
__align__(128)
GaussInfo h_gauss;
```

```
void init_filter(float sigma0, int levels)
{ …
  h_gauss.clearTables();
  int stages = levels + 3;
  sigma[0] = sigma0;
  for( int lvl=1; lvl<stages; lvl++ ) {
    const float P = sigma0 * pow(
      2.0f, (float)(lvl-1)/(float)levels );
    const float S = sigma0 * pow(
      2.0f, (float)(lvl)/(float)levels );
    sigma[lvl]=sqrt(S*S - P*P);
    span[lvl]=std::min<int>(
      ceilf(4.0f*sigma[lvl])+1,
      GAUSS_ALIGN+1);
    …
```

Initialize the Gaussian filter

```
template<int LEVELS>
struct GaussTable
{
  float filter[LEVELS*GAUSS_ALIGN];
  float sigma [LEVELS];
  int   span  [LEVELS];
};

__device__ __constant__
GaussInfo d_gauss;
__align__(128)
GaussInfo h_gauss;
```

```
void init_filter(float sigma0, int levels)
{ …
    double sum = 1.0;
    filter[level*GAUSSL-ALIGN+0] = 1.0f;
    for(int x=1; x<span[lvl]; x++= {
      const float val = exp(
        -0.5*(pow(double(x)/sigma[lvl],
                   2.0)));
      filter[level*>GAUSS_ALIGN+1] = val;
      sum += 2.0f * val;
    }
    for(int x=1; x<span[lvl]; x++= {
      filter[level*GAUSS_ALIGN+x] /= sum;
    }
  }
  …
}
```

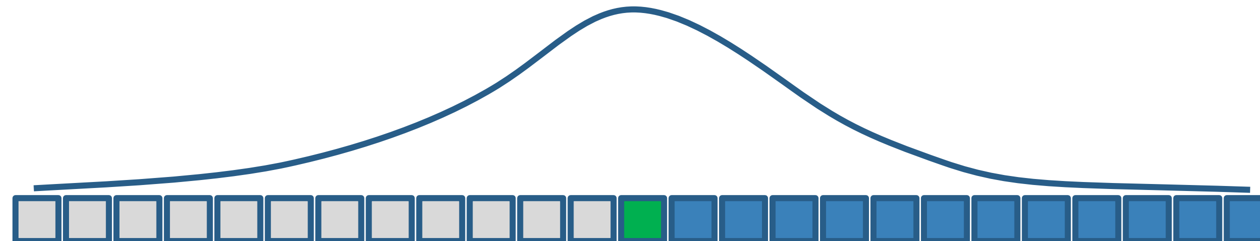
#

#

Initialize the Gaussian filter

```
template<int LEVELS>
struct GaussTable
{
  float filter[LEVELS*GAUSS_ALIGN];
  float sigma [LEVELS];
  int   span  [LEVELS];
};

__device__ __constant__
GaussInfo d_gauss;
__align__(128)
GaussInfo h_gauss;
```

```
void init_filter(float sigma0, int levels)
{ …
  cudaError_t err;
  err = cudaMemcpyToSymbol(
    d_gauss,
    &h_gauss,
    sizeof(GaussInfo), 0,
    cudaMemcpyHostToDevice );
  POP_CUDA_FATAL_TEST(err,
    "cudaMemcpyToSymbol failed: ”);
}
```



Initialize the Gaussian filter

```
template<int LEVELS>
struct GaussTable
{
  float filter[LEVELS*GAUSS_ALIGN];
  float sigma [LEVELS];
  int   span  [LEVELS];
};

__device__ __constant__
GaussInfo d_gauss;
__align__(128)
GaussInfo h_gauss;
```

```
void init_filter(float sigma0, int levels)
{ …
  cudaError_t err;
  err = cudaMemcpyToSymbol(
      d_gauss,
      &h_gauss,
      sizeof(GaussInfo), 0,
      cudaMemcpyHostToDevice );
  POP_CUDA_FATAL_TEST(err,
    "cudaMemcpyToSymbol failed: ”);
}
```

#

Create pyramid

calls `Pyramid::horiz_from_input_image`
starts the kernel

`gauss::normalizedSource::horiz`
with blocks of 128 threads in one dimensions, grid configuration determines that we use one for every pixel in the image!
https://github.com/alicevision/popsift/blob/develop/src/popsift/s_pyramid_build_ra.cu in line 18

for the all other images, `Pyramid::build_pyramid` calls
`Pyramid::horiz_from_prev_level`
which is in line 250

Fill the first image of the Gaussian pyramid

```
class Octave
{
  int w, h;
  cudaArray_t data;
  cudaArray_t intm;
  cudaArray_t dog;
  cudaSurfaceObject_t surf;
  cudaTextureObject_t tex;
  …
};

class Pyramid
{
  int _num_octaves, _levels;
  Octave* _octaves;
  …
```

```
__device__ __host__
inline int grid_divide( int val, int div )
{
  return val  / div + ( val%div==0?0:1);
}

__host__
void Pyramid::horiz_from_input_image(
  Image* base)
{ …
  dim3 block( 128, 1 );
  dim3 grid;
  grid.x  = grid_divide( width,  128 );
  …
  gauss::normalizedSource::horiz
        <<<grid,block,0,stream>>>
        ( base->_tex,
          octaves[0].surf,
          width, height, … );
}
```

Fill the first image of the Gaussian pyramid

```
class Octave
{
  int w, h;
  cudaArray_t data;
  cudaArray_t intm;
  cudaArray_t dog;
  cudaSurfaceObject_t surf;
  cudaTextureObject_t tex;
  …
};

class Pyramid
{
  int _num_octaves, _levels;
  Octave* _octaves;
  …
```

```
__global__
void horiz(cudaTextureObject_t tex,
  cudaSurfaceObject_t dst, int w, int h, …)
{
  const int write_x =
    blockIdx.x * blockDim.x + threadIdx.x;
  const int    write_y = blockIdx.y;
  if( write_x >= dst_w ) return;

  const int span=d_gauss.span[0];
  const float* filter =&d_gauss.filter[0];
  const float  read_x = (blockIdx.x *
blockDim.x+threadIdx.x+shift ) / w;
  const float  read_y =
    (blockIdx.y + shift ) / h;

  float out = 0.0f;
```

Side effect of texture: free choice of scale for the first image!

Fill the first image of the Gaussian pyramid

```
class Octave
{
  int w, h;
  cudaArray_t data;
  cudaArray_t intm;
  cudaArray_t dog;
  cudaSurfaceObject_t surf;
  cudaTextureObject_t tex;
  …
};

class Pyramid
{
  int _num_octaves, _levels;
  Octave* _octaves;
  …
```

```
  for(int o=span; o>0; o-- ) {
    const float& g=filter[o];
    const float  rel=float(o) / w;
    const float  v1=tex2D<float>(
      tex,read_x-rel,read_y);
    const float  v2=tex2D<float>(
      tex,read_x+rel,read_y );
    out += ( ( v1 + v2 ) * g );
  }
  const float& g = filter[0];
  const float v3 = tex2D<float>(
    tex, read_x, read_y );
  out += ( v3 * g );

  surf2DLayeredwrite(
    out * 255.0f, dst_data,
    write_x*4, write_y,
    0, cudaBoundaryModeZero );
}
```

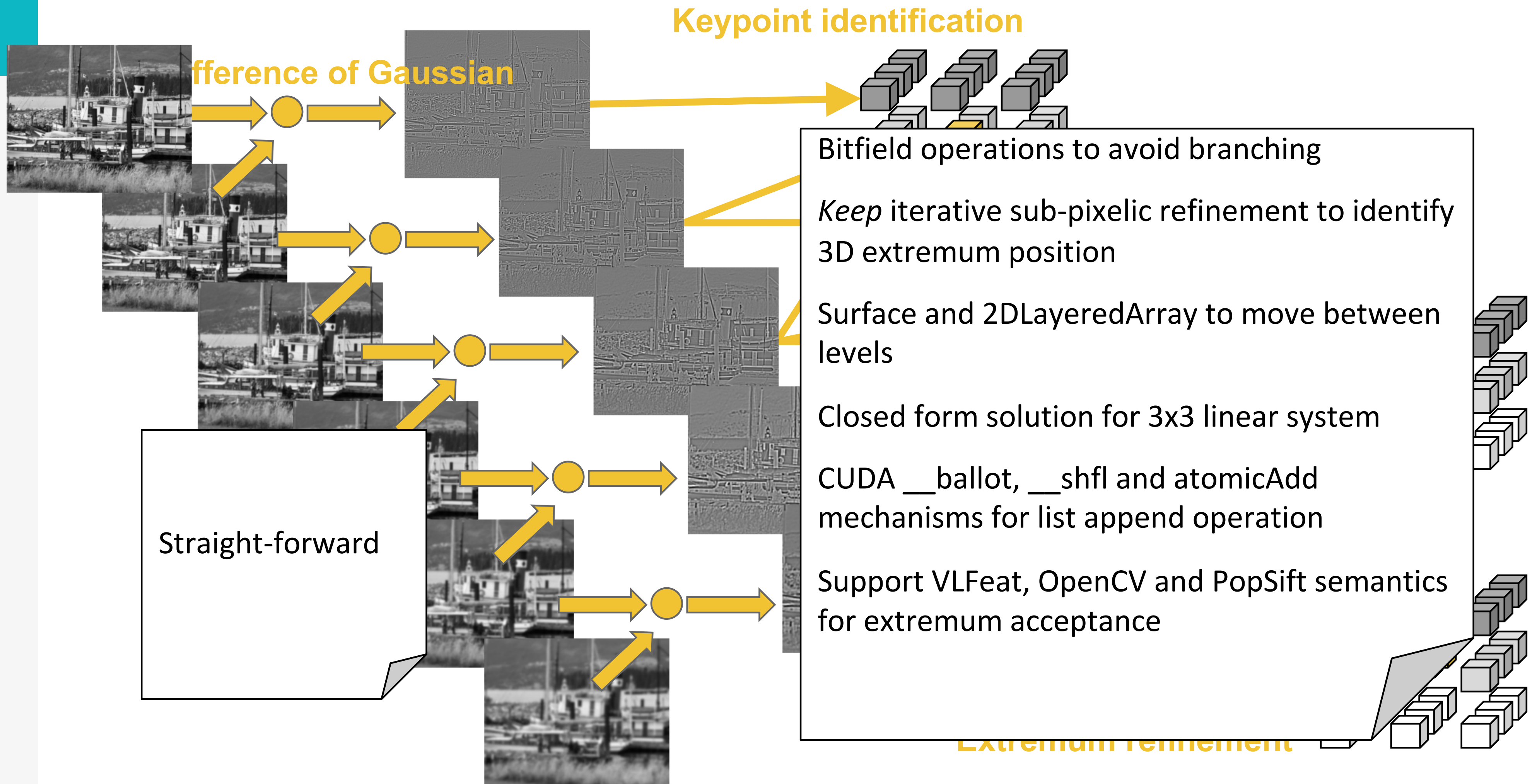
#

Create pyramid

calls `Pyramid::horiz_from_prev_level`
starts the kernel

`gauss::absoluteSourceInterpolated::horiz`
with blocks of 128 threads in one dimensions, grid configuration determines that we use one for every pixel in the image!
https://github.com/alicevision/popsift/blob/develop/src/popsift/s_pyramid_build_ai.cu in line 18

CUDA tuning



#

Find extrema

https://github.com/alicevision/popsift/blob/develop/src/popsift/s_extrema.cu

`find_extrema_in_dog`: line 512
`find_extrema_in_dog_sub`: line 302
`is_extremum`: line 58

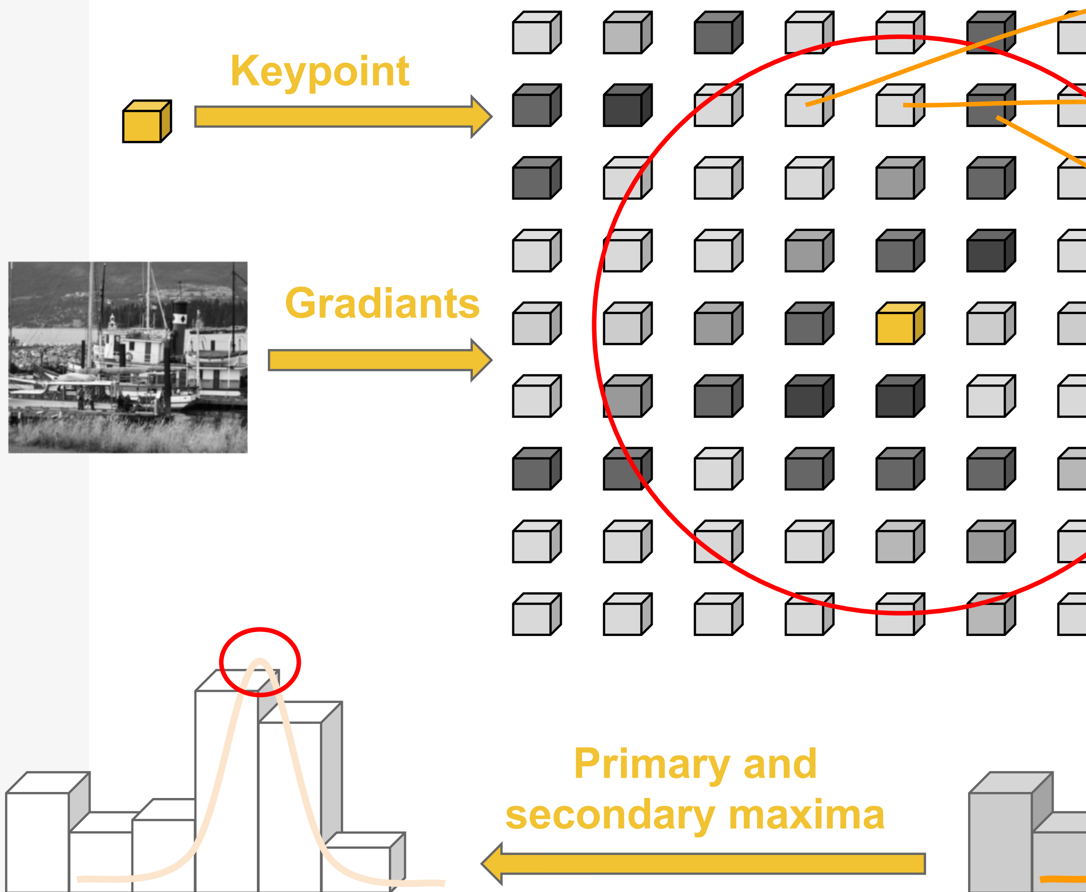
#

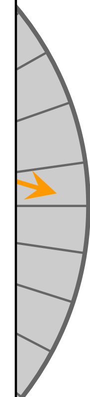
Fill the first image of the Gaussian pyramid

```
class Octave
{
  int w, h;
  cudaArray_t data;
  cudaArray_t intm;
  cudaArray_t dog;
  cudaSurfaceObject_t surf;
  cudaTextureObject_t tex;
  …
};

class Pyramid
{
  int _num_octaves, _levels;
  Octave* _octaves;
  …
```

```
  for(int o=span; o>0; o-- ) {
    const float& g=filter[o];
    const float  rel=float(o) / w;
    const float  v1=tex2D<float>(
      tex,read_x-rel,read_y);
    const float  v2=tex2D<float>(
      tex,read_x+rel,read_y );
    out += ( ( v1 + v2 ) * g );
  }
  const float& g = filter[0];
  const float v3 = tex2D<float>(
    tex, read_x, read_y );
  out += ( v3 * g );

  surf2DLayeredwrite(
    out * 255.0f, dst_data,
    write_x*4, write_y,
    0, cudaBoundaryModeZero );
}
```

CUDA tuning





- Gradiants not pre-computed due to keypoint sparsity
- CUDA __shared__ memory for histogram
- Support VLFeat, OpenCV and PopSift histogram smoothing methods
- Warp shuffle operations to implement no-overhead parallel bitonic sort, finding all accepted orientations in parallel
- optional CUDA Dynamic Parallelism for cards with Compute Capability >= 3.5

#

Find orientation

https://github.com/alicevision/popsift/blob/develop/src/popsift/s_orientation.cu

`Pyramid::orientation`: line 248
`ori_par`: line 61

CUDA tuning

Feature extraction

Obviously, speed depends on the scene

Correlated with number of extrema

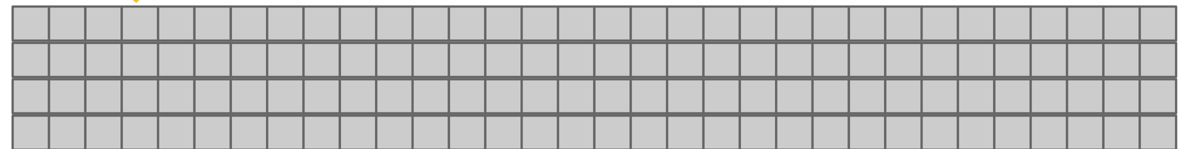
Upscaling:	(ms) yes	(ms) no
765 x 512	12.5	5.9
850 x 680	16.9	5.9
1000 x 700	18.0	7.7
1920 x 1080	24.8	9.4

No gradient pre-computation due to feature sparsity

Assign thread block to each group of 8 values in the feature vector

Support rootSift and L2 normalization

Scale by powers of 2 before exporting



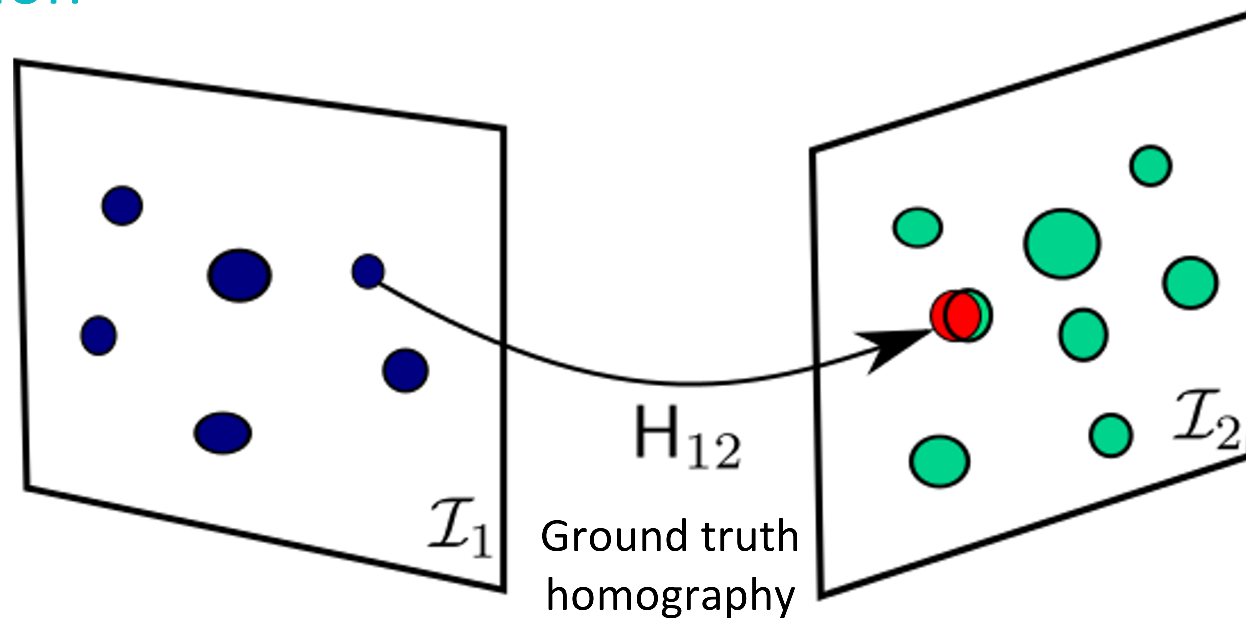
#

Compute descriptors

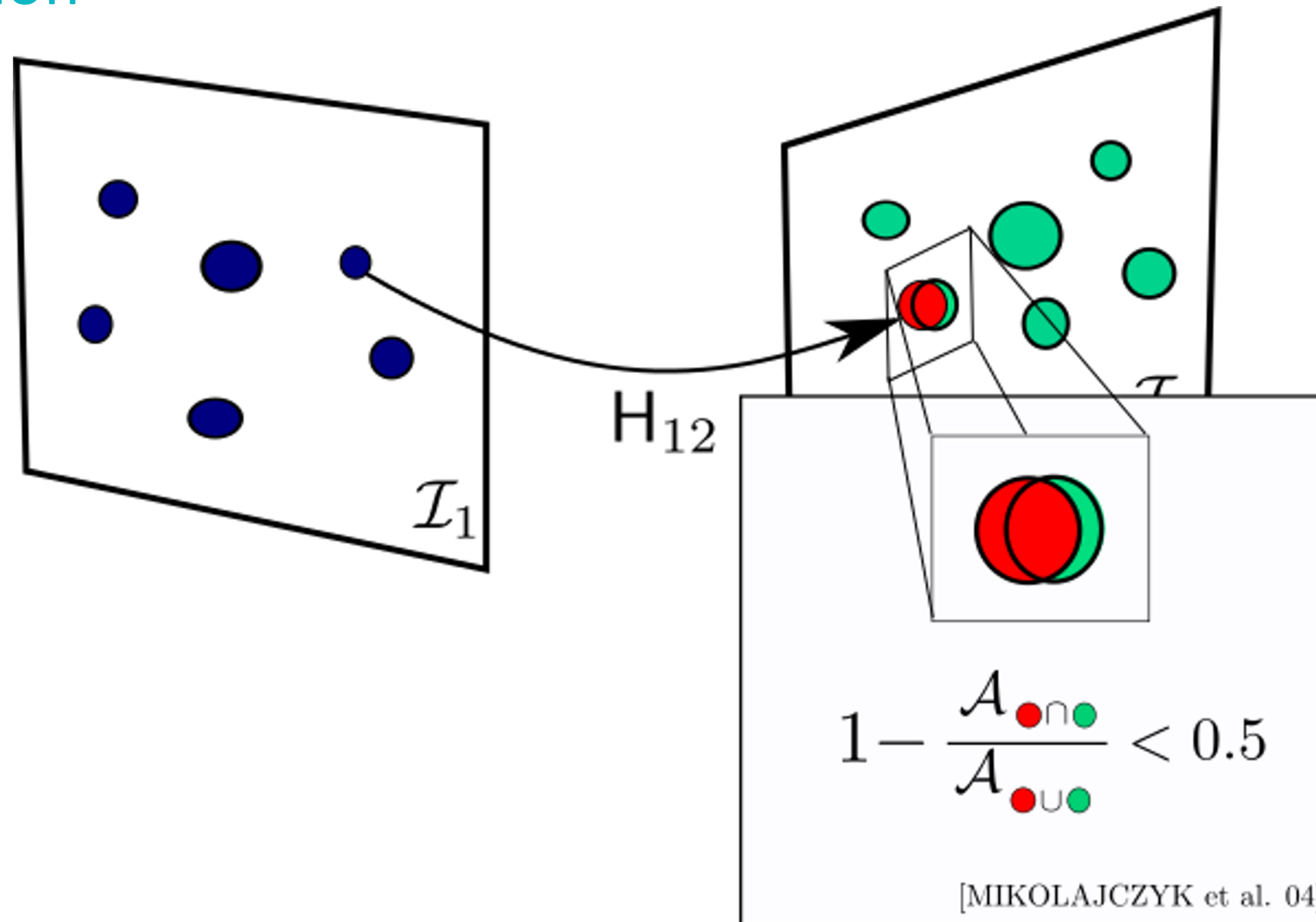
https://github.com/alicevision/popsift/blob/develop/src/popsift/s_desc_igrid.cu

`ext_desc_igrid`: line 61

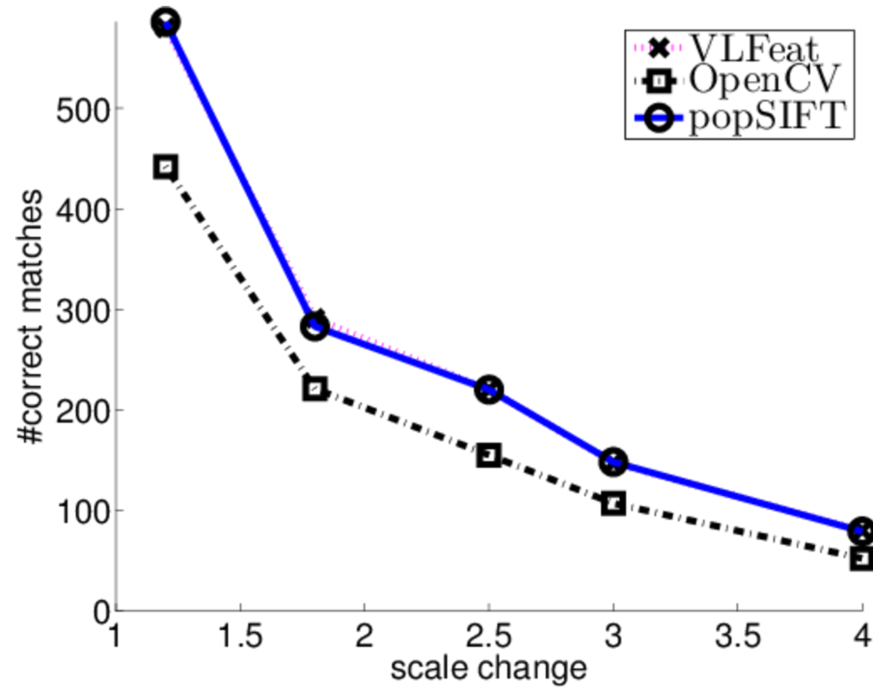
Evaluation

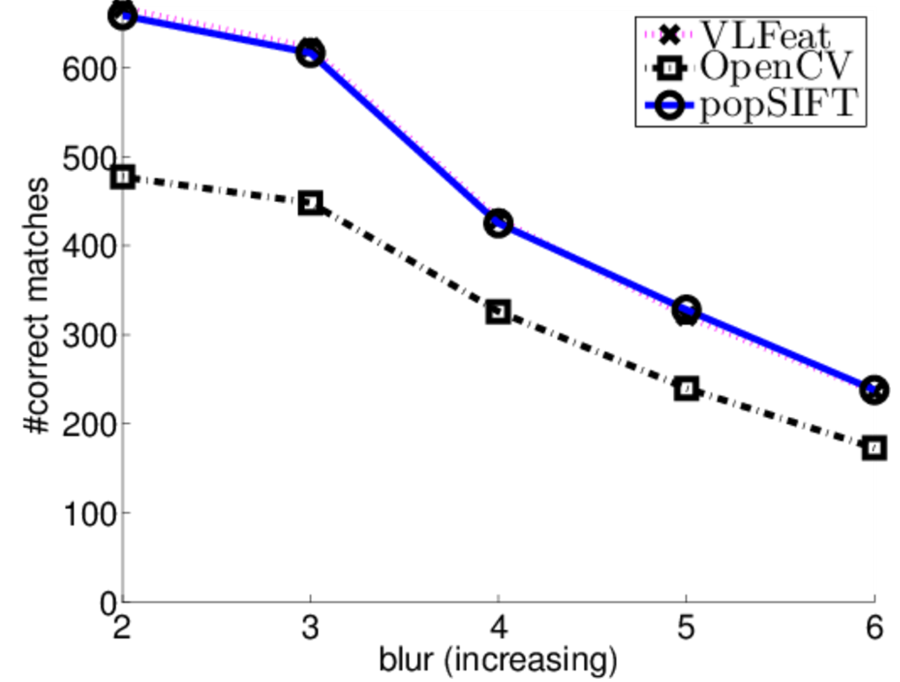


Evaluation

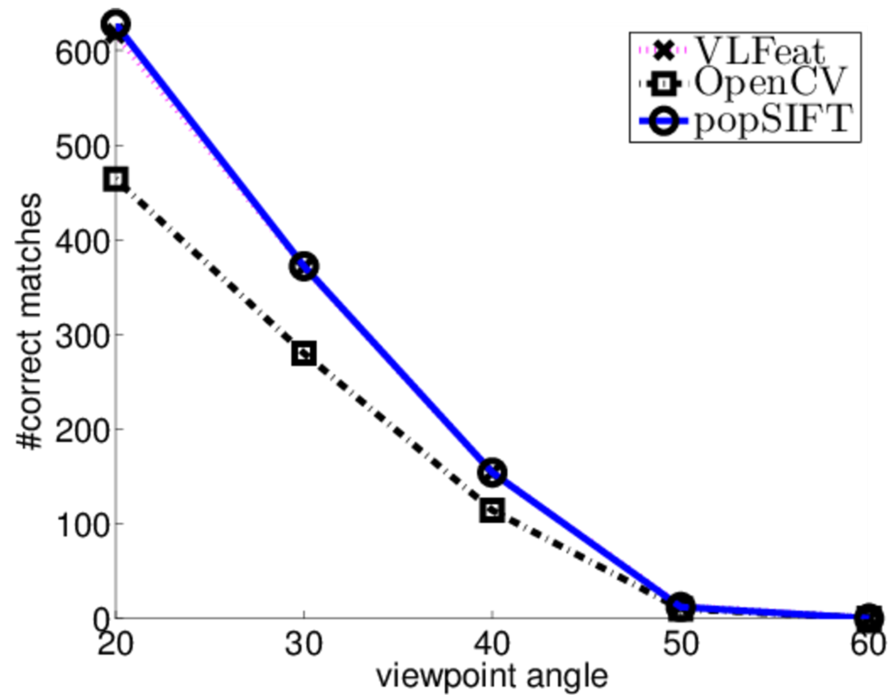


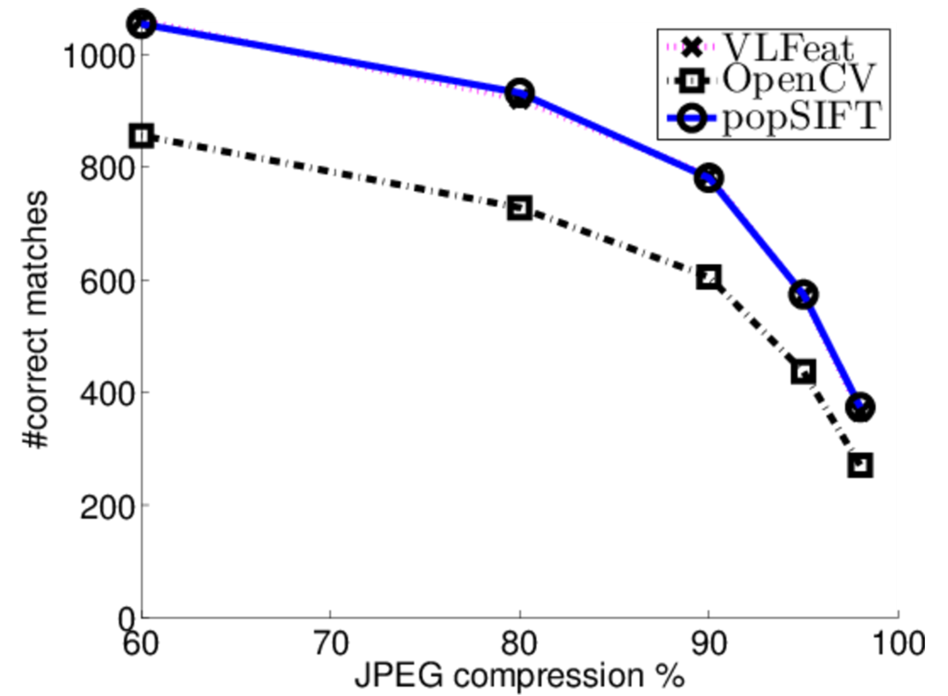
Evaluation





Evaluation





Some details on CUDA functions

#

memories

- host
- device
- shared
- local
- constant
- texture
  - TextureObject / TextureReference (deprecated)
- surface

- read-only caches:
  - L1 cache – on some CCs, shared with __shared__
  - L2 cache

calling

- `__host__`
- `__global__`
- `__device__`

- C++ templates!
- limited stack depth
- can malloc in kernel – but slow
- can start kernels in kernel – but
complicated – except tail recursion!

Hints & Tips

- CUDA can handle memory between CPU and GPU in 3 ways
- copy explicitly between CPU and GPU using `cudaMallocHost, cudaMAlloc, cudaMemcpy`
  - DMA transfer – you know where memory is
  - use if GPU is on a card or Windows

```
cudaError_t err;
YourType *hst, *dev;
size_t sz = 10*sizeof(YourType);

err=cudaMallocHost(&hst, sz,
    cudaMemAttachGlobal);
… test and fill memory …

err=cudaMalloc(&dev, sz,
    cudaMemAttachGlobal);
… test …

err=cudaMemcpy(dev, hst, sz,
    cudaMemcpyHostToDevice);
… test …
```

```
if(err != cudaSuccess) {
    std::cerr << "Failed to allocate managed mem: "
              << cudaGetErrorString(err) << std::endl;
    exit(-1);
}
```

Hints & Tips

- CUDA can handle memory between CPU and GPU in 3 ways
- reserve a shared memory range: `cudaMallocManaged`
  - device driver guesses who needs memory next
  - use if GPU is in a SOC+Linux
- Careful: CUDA must guess when you need something on CPU or GPU. Safest to use with stream sync instructions.

```
cudaError_t err;
YourType* ptr;
size_t sz = 10*sizeof(YourType);

err=cudaMallocManaged(&ptr, sz,
    cudaMemAttachGlobal);
… test …
```

Hints & Tips

- CUDA can handle memory between CPU and GPU in 3 ways
- better on Tegra than MallocManaged
- reserve a shared memory range: `cudaMallocHost` and register in GPU with `cudaHostRegister` and `cudaHostGetDevicePointer`
  - device driver guesses who needs memory next
  - use if GPU is in a SOC+Linux

```
cudaError_t err;
YourType *hst, *dev;
size_t sz = 10*sizeof(YourType);

err=cudaMallocHost(&hst, sz);
… test …

err=cudaHostRegister(hst, sz,
    cudaHostRegisterDefault);
… test …

err=cudaHostGetDevicePointer(
    dev, hst, 0);
… test …
```

#

Hints & Tips

- Allocate well-aligned memory only on GPU

```
cudaError_t err;
YourType *dev;
size_t pitch;
size_t sx = 10;
size_t sy = 20;

err=cudaMallocPitch(
    &dev,
    &pitch,
    sz * sizeof(YourType),
    sy );
… test …
```

```
__inline__ float get(YourType* ptr,
int x, int y, int pitch)
{
    YourType* p =
        (YourType*)(
            ((char*)ptr)+y*pitch );
    return p[x];
}
```

#

indexing

- starting a kernel
- void kernel<<<blocks,threads,sharedmem,stream>>>(params);

Datatype: dim3
threadIdx – blockDim
blockIdx – gridDim

```
int myidx = threadIdx.x + threadIdx.y * blockDim.y;
```

max 1024 threads per block
WARPs !

lockstep

```
float a = bigmem[myidx];
float b = moremem[myidx];
if( threadIdx.x % 2 == 0 )
{
    float c = a;
    c /= b;
}
else
{
    float c = a;
    c *= b;
}
c = sqrtf( c );
```

#

synchronize

memory sync automatic at end of kernel

cudaDeviceSynchronize() – CPU/GPU
cudaStreamSynchronize() – partial CPU/GPU

__syncthreads() – between threads on an SM
__any_sync(mask,predicate), __all_sync, __ballot_sync – between threads in a warp

new stuff: Thread groups, C++ objects

Hints & Tips

- Integer multiplication with powers of 2 is well-known:

```
int k = <>;
int z = k * 16;

int k = <>;
int z = k << 4;
```

- shifts the bit to achieve multiplication and division

- Float multiplication with powers of 2 is less known:

```
float k = <>;
float z = k * 16.0f;

float k = <>;
float z = scalbnf(k,4);
```

- adds and subtracts the exponent to achieve multiplication and division

#

Hints & Tips

- Fast exchange of 4-byte values between *lanes* within a single SM
- Groups of 2, 4, 8, 16 or 32 threads with identical `threadIdx.{y,z}` and `blockIdx.{x,y,z}` can exchange data very fast

- *good thing: this is also a barrier*
- under the hood, this is a SIMD register shuffle

adding 32 integers:

OLD

```
int k = <>;
int k = __shfl_down(k,16);
int k = __shfl_down(k,8);
int k = __shfl_down(k,4);
int k = __shfl_down(k,2);
int k = __shfl_down(k,1);
int k = __shfl(k,0);
```

NEW

```
int k = <>;
int m = 0xffffffff; /* mask */
int k = __shfl_down_sync(m,k,16);
int k = __shfl_down_sync(m,k,8);
int k = __shfl_down_sync(m,k,4);
int k = __shfl_down_sync(m,k,2);
int k = __shfl_down_sync(m,k,1);
int k = __shfl_sync(m,k,0);
```

Hints & Tips

- CUDA compiler does not try to second-guess that **you** know what **you** are doing

- C's implicit typecast rules are followed strictly
- cannot don't forget that constants value are float values

Which of these is fastest, which is slowest?

```
int k = 4;
```

```
float x = k * 7;
```

- (1) i*i (2) c(i->f) (3) f=f

```
float y = k * 7.0;
```

- (1) c(i->d) (2) d*d (3) c(d->f) (4) f=f

```
float z = k * 7.0f;
```

- (1) c(i->f) (2) f*f (3) f=f

texture access

- texture access
- +0.5f
- interpolation

- linear interpolation – or next neighbour in floor(coord)
- normalize 0-255 → 0.0f-1.0f (or not)

- float v = tex2D<float>(x+0.5f, y+0.5f);

#

registers required by kernel

- compiler decides how many registers every kernel gets
- you have 64K register memory
- each int or float eats 4 bytes – so 16*1024=16384 values
- assume 1024 threads: only 16 int or floats per thread!!!!!

- Calling any kind of function does usually exceed the available registers, leading to register spill to local
- AVOID!
- compiler has warning flags

NVTX

- add nvtx functions to log time on CPU as well