Cuimg: A small image processing framework based on CUDA

Matthieu Garrigues <matthieu.garrigues@ensta.fr>

May 26, 2011

Outline

Goals

- Write efficient image processing applications
- With CUDA
- Reduce the size of the application code
- ... and the number of potential bugs

Image types

```
1  // Host types.
2  template <typename T> host_image2d;
3  template <typename T> host_image3d;
4  // CUDA types.
5  template <typename T> image2d;
6  template <typename T> image3d;
7  // Types used inside CUDA kernels.
8  template <typename T> kernel_image2d;
9  template <typename T> kernel_image3d;
10
11 {
12   image2d<float4> img(100, 100); // creation.
13   image2d<float4> img2 = img; // light copy.
14 } // Images are freed automatically here.
15
16 // See a slice as a 2d image:
17 image3d<float4> img3d(100, 100, 100);
18 image2d<float4> slice = img3d.slice(42);
```

Improved builtins types

```
1   i_float3 a(1.5, 1.5, 1.5);
2   i_float3 b(2.5, 2.5, 2.5);
3
4   // Classical arithmetic operators
5   i_float3 c = (a + b) * 2.5f;
6
7   // Interoperability
8   i_char2 x = i_int2(0,1) + i_float2(0,1) * 5.0f;
9   // Type of (i_int2 + i_float2) is i_float2 as
10  // (int + float) returns a float.
```

Inputs

```
1   // Load 2d images.
2   host_image2d<uchar3> img = load("test.jpg");
3
4   // Read USB camera
5   video_capture cam(0);
6   host_image2d<uchar3> cam_img(cam.nrows(), cam.ncols());
7   cam >> cam_img; // Get the camera current frame
8
9   // Read a video
10  video_capture vid("test.avi");
11  host_image2d<uchar3> frame(vid.nrows(), vid.ncols());
12  vid >> v; // Get the next video frame
```

Data tranfers

```
1
2   host_image2d<uchar3> img_h = load("test.jpg"); // Lives in CPU RAM
3   image2d<uchar3> img(img_h.domain()); // Lives in GPU RAM
4
5   copy(img_h, img); // CPU -> GPU
6   copy(img, img_h); // GPU -> CPU
7
8   // Also works on 3d images.
```

Fast gaussian convolutions code generation

- Heavy use of C++ templates for loop unrolling
- Gaussian kernel is known at compile time and injected directly inside the assembly
- Used by the local jet computation
- Cons: Large kernels slow down the compilation.

```
mov.f32 %f182, 0f3b1138f8; // 0.00221592
mul.f32 %f183, %f169, %f182;
mov.f32 %f184, 0f39e4c4b3; // 0.000436341
mad.f32 %f185, %f184, %f181, %f183;
mov.f32 %f186, 0f3c0f9782; // 0.00876415
mad.f32 %f187, %f186, %f157, %f185;
mov.f32 %f188, 0f3cdd25ab; // 0.0269955
mad.f32 %f189, %f188, %f145, %f187;
mov.f32 %f190, 0f3d84a043; // 0.0647588
mad.f32 %f191, %f190, %f133, %f189;
mov.f32 %f192, 0f3df7c6fc; // 0.120985
mad.f32 %f193, %f192, %f121, %f191;
mov.f32 %f194, 0f3e3441ff; // 0.176033
mad.f32 %f195, %f194, %f109, %f193;
mov.f32 %f196, 0f3e4c4220; // 0.199471
mad.f32 %f197, %f196, %f97, %f195;
mov.f32 %f198, 0f3e3441ff; // 0.176033
mad.f32 %f199, %f198, %f85, %f197;
mov.f32 %f200, 0f3df7c6fc; // 0.120985
mad.f32 %f201, %f200, %f73, %f199;
mov.f32 %f202, 0f3d84a043; // 0.0647588
mad.f32 %f203, %f202, %f61, %f201;
mov.f32 %f204, 0f3cdd25ab; // 0.0269955
mad.f32 %f205, %f204, %f49, %f203;
mov.f32 %f206, 0f3c0f9782; // 0.00876415
mad.f32 %f207, %f206, %f37, %f205;
mov.f32 %f208, 0f3b1138f8; // 0.00221592
mad.f32 %f209, %f208, %f25, %f207;
mov.f32 %f210, 0f39e4c4b3; // 0.000436341
```

A simple CUDA kernel

```
1
2   // Declaration
3   template <typename T, typename U, typename V>
4   __global__ void simple_kernel(kernel_image2d<T> a,
5                                 kernel_image2d<U> b,
6                                 kernel_image2d<V> out)
7   {
8     point2d<int> p = thread_pos2d();
9     if (out.has(p))
10      out(p) = a(p) * b(p) / 2.5f;
11  }
12
13  // Call
14  dim3 dimblock(16, 16);
15  dim3 dimgrid(divup(a.ncols(),  16), divup(a.nrows(),  16));
16  simple_kernel<<<dimgrid, dimblock>>>(a, b, c);
```

Simple kernels generator

Goal

Automatically generate simple kernels

Examples

```
1  image2d<i_int4> a(200, 100);
2  image2d<i_short4> b(a.domain());
3  image2d<i_float4>   c(a.domain());
4
5  // Generates and calls simple_kernel.
6  c = a * b / 2.5;
7
8  // BGR <-> RGB conversion
9  a = aggregate<int>::run(get_z(a), get_y(a), get_x(a), 1.0f);
```

Simple kernels generator

```
1   // C++ templates handle the syntax tree of the expression
2   // Type of
3   a * b / 2.5
4   // is
5   div<mult<image2d<i_int4>,
6            image2d<i_short4> >,
7       float>
8
9
10  //operator= handles the kernel call
11  template <typename I, typename E>
12  __global__ void assign_kernel(kernel_image2d<I> out, E e)
13  {
14    i_int2 p = thread_pos2d();
15    if (out.has(p))
16      out(p) = e.eval(p);
17      // Recursivelly evaluates the expression, inlining of the recursive
18         evaluation of the tree is done by the compiler.
19  }
20
21  template <typename A, template <class> class AP, typename E>
22  inline void
23  image2d::operator=(E& expr)
24  {
25    assign_kernel<<<dimgrid, dimblock>>>(*this, e);
26  }
```

Question?