

Image Processing on Graphics Processing Unit with CUDA and C++

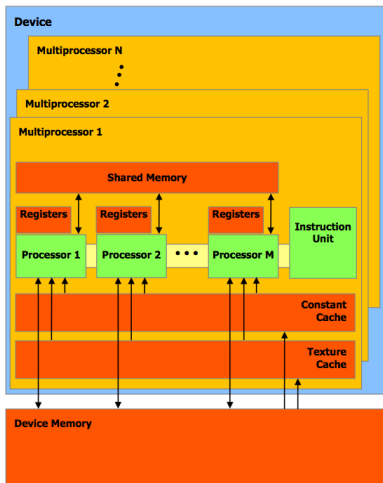
Matthieu Garrigues <matthieu.garrigues@gmail.com>
ENSTA-ParisTech

June 15, 2011

Outline

- 1 The Graphic Processing Unit
- 2 Benchmark
- 3 Real Time Dense Scene tracking
- 4 A Small CUDA Image Processing Toolkit
- 5 Dige

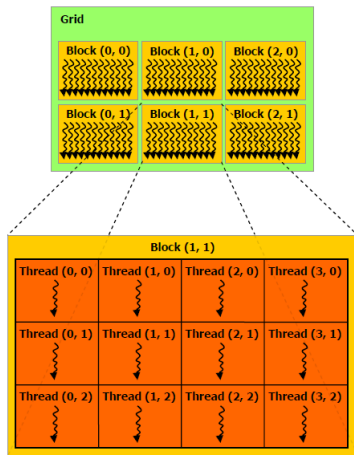
Nvidia CUDA Architecture



Schema from CUDA documentation [NVI]

- GeForce GTX 580: 16 multiprocessors \times 32 processors
- 1 multiprocessor runs simultaneously 32 threads
- Threads local to a multiprocessor communicate via fast on-chip shared memory
- Two dimensional texture cache

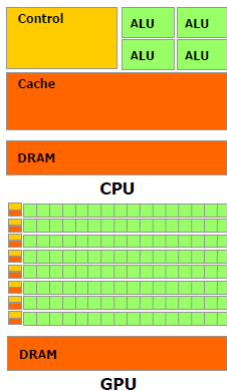
The CUDA Programming Model



Schema from CUDA documentation [NVI]

- Every thread execute the same programs on different data according to a unique thread id
- For example: One thread handles one pixel
- Grid size and block size are defined by the user to fit the data dimensions
- One block runs on one multiprocessor
- Intra block synchronization and communication via shared memory

Parallelism hides high latency



Schema from CUDA documentation [NVI]

- The GPU devotes more transistors to data processing
- Absence of branch prediction, out of order execution, etc implies high read after write register latency (about 20-25 cycles)
- A multiprocessor must overlap execution of at least 6 thread blocks to hide this latency
- \Rightarrow At least $30 \times 6 \times 16 = 2880$ threads needed to properly feed the Geforce GTX 280

The PCI Bus

- Communication between host and device is done via the PCI bus which can be a bottleneck
- Need to minimize data transfers
- However:
 - PCI express 2.0 16x theoretical bandwidth is 8GB/s
 - Next generation PCI express 3.0 16x theoretical bandwidth is 16GB/s
 - CUDA driver pins memory for direct memory access from the device

Benchmarking host device communications

Benchmark

```
image2d<i_char4>    ima(1024, 1024);
host_image2d<i_char4> h_ima(1024, 1024);
for (unsigned i = 0; i < 100; i++)
    copy(h_ima, ima); // Host to device
for (unsigned i = 0; i < 100; i++)
    copy(ima, h_ima); // Device to host
```

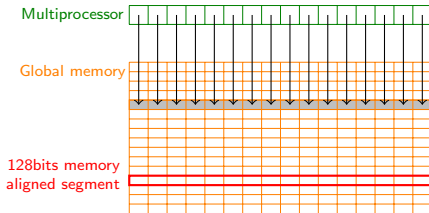
Table: Results using PCI-Express 2.0 16x / Geforce GTX 280

	classical memory	pinned-memory
host to device	1.44 GB/s	5.5 GB/s
device to host	1.25 GB/s	4.7 GB/s

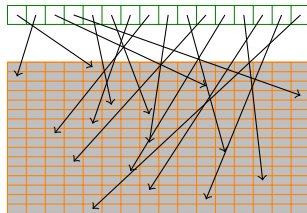
- About 1ms to transfer a RGBA 8bits 512x512 2D image using pinned memory
- Transfers and CUDA kernels execution can overlap with CPU computation

Global Memory Access

1 memory transaction

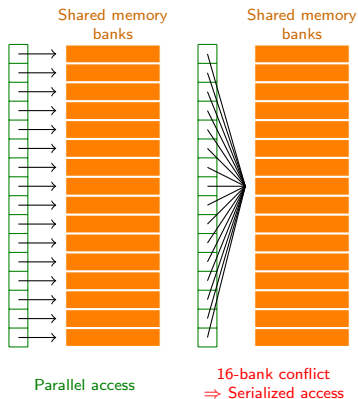


16 memory transactions



- Theoretical maximum bandwidth: 115GB/s (GTX 460)
- Accesses inside a multiprocessor coalesce if they fall in the same 128 aligned segment.

Shared Memory Access



- On-chip memory
- 100x faster than global memory if no bank conflicts
- Successive 32 bits words are assigned to successive banks
- Bank bandwidth: 32 bits per cycle
- Bank conflicts are serialized

GPUs are not magic...

- Host device communications should be minimum
- Global memory accesses inside a block should coalesce
- The instruction flow should not diverge within a block
- Shared memory should be used with care (bank conflicts)
- Enough thread blocks should run to hide register latency

Convolution Benchmark: CPU vs GPU

Benchmark

- Convolution on row and columns
- Image size: $3072 \times 3072 \times \text{float } 32\text{bits} = 36\text{MB}$

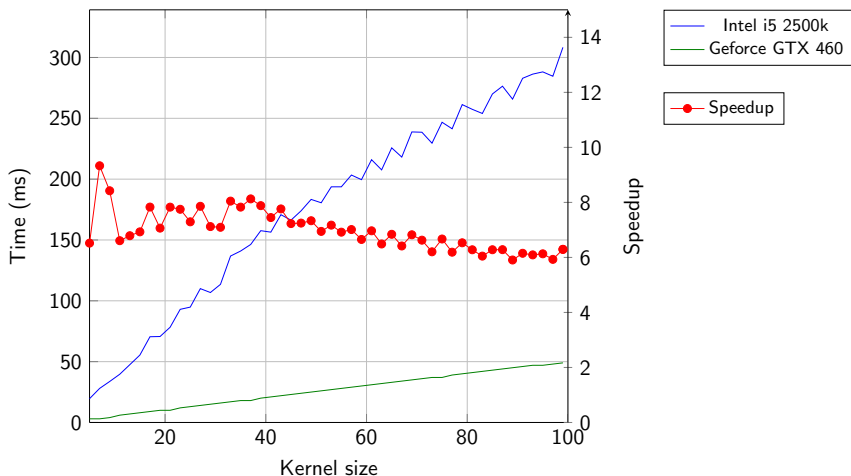
CPU version

- Intel i5 2500k 4-cores@3.3GHz (May 2011: 150 euros)
- Use of OpenMP to feed all the cores

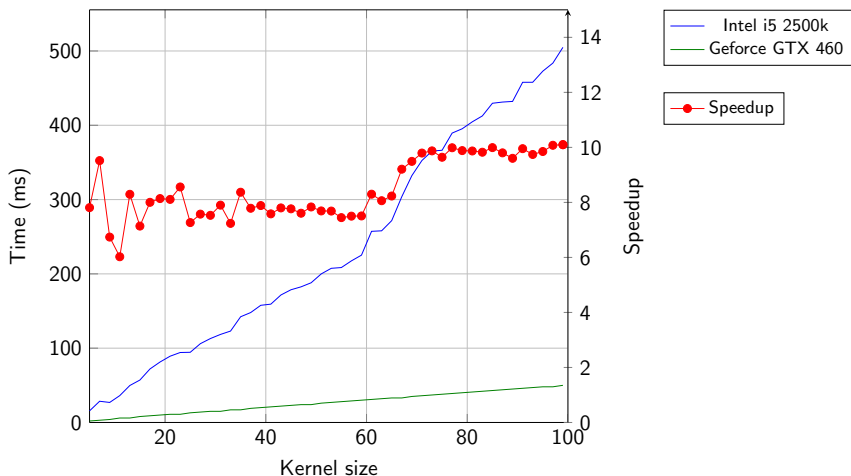
GPU version

- Hardware: GeForce GTX 460 (May 2011: 160 euros)
- CUDA and C++
- 2D texture cache

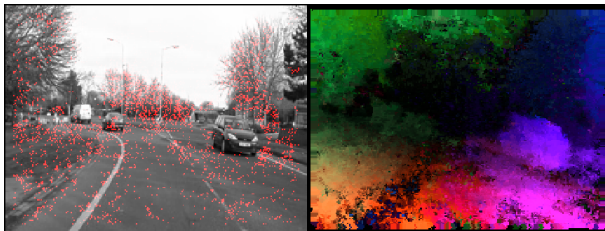
Convolution on rows: CPU vs GPU



Convolution on columns: CPU vs GPU



Real Time Dense Scene tracking



- Track scene points through the frames
- Estimate apparent motion of each points
- In real time

Real Time Dense Scene tracking: Overview

Descriptor

- 128bits local jet descriptors [[Man11]]
- Partial derivatives, several scales
- 16 x 8 bits components

Matching

Search of the closest point according to a distance on the descriptor space

Tracking

- Move the trackers according to the matches
- Filter bad matches (Occlusions)
- Avoid trackers drifts (Sub-pixel motion)
- Estimate speed of the trackers

Real Time Dense Scene tracking: Overview

Descriptor

- 128bits local jet descriptors [[Man11]]
- Partial derivatives, several scales
- 16 x 8 bits components

Matching

Search of the closest point according to a distance on the descriptor space

Tracking

- Move the trackers according to the matches
- Filter bad matches (Occlusions)
- Avoid trackers drifts (Sub-pixel motion)
- Estimate speed of the trackers

Real Time Dense Scene tracking: Overview

Descriptor

- 128bits local jet descriptors [[Man11]]
- Partial derivatives, several scales
- 16 x 8 bits components

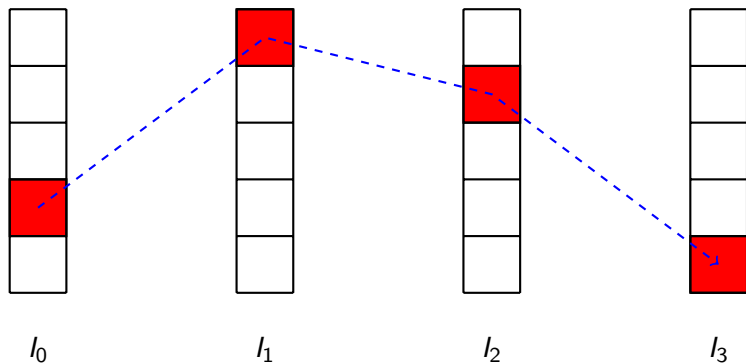
Matching

Search of the closest point according to a distance on the descriptor space

Tracking

- Move the trackers according to the matches
- Filter bad matches (Occlusions)
- Avoid trackers drifts (Sub-pixel motion)
- Estimate speed of the trackers

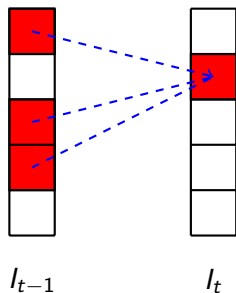
First Pass: Matching



- For each pixel of frame I_{t-1} find its closest point on frame I_t
- We limit our search to a fixed 2D neighborhood \mathcal{N}

$$\text{match}_{t-1}(p) = \arg \min_{q \in \mathcal{N}(p)} \mathcal{D}(I_{t-1}(p), I_t(q))$$

Second Pass: Tracking



For each point p of I_t

- 1 Find its counterparts in I_{t-1}

$$\mathcal{C}_p = \{q \in I_{t-1}, p = \text{match}_{t-1}(q)\}$$

- 2 Threshold bad matches

$$\mathcal{F}_p = \{q \in \mathcal{C}_p, \mathcal{D}(I_{t-1}(p), I_t(q)) < \mathcal{T}\}$$

- 3 Update tracker speed and age
- 4 If no counterpart, tracker is discarded

Temporal Filters

Filtering descriptor

Instead of matching descriptors between I_{t-1} and I_t , we use a weighted mean of the trackers history. Each tracker x computes at time t :

$$x_v^t = (1 - \alpha)x_v^{t-1} + \alpha I_t \quad \alpha \in [0, 1] \quad \text{the forget factor}$$

A low α minimize tracker drifts

Filtering tracker speed

Likewise, the filtered tracker speed is given by:

$$x_s^t = (1 - \beta)x_s^{t-1} + \beta S_x^t \quad \beta \in [0, 1]$$

- $\beta \in [0, 1]$ is the forget factor
- S_x^t is the speed estimate of tracker x at time t

Results

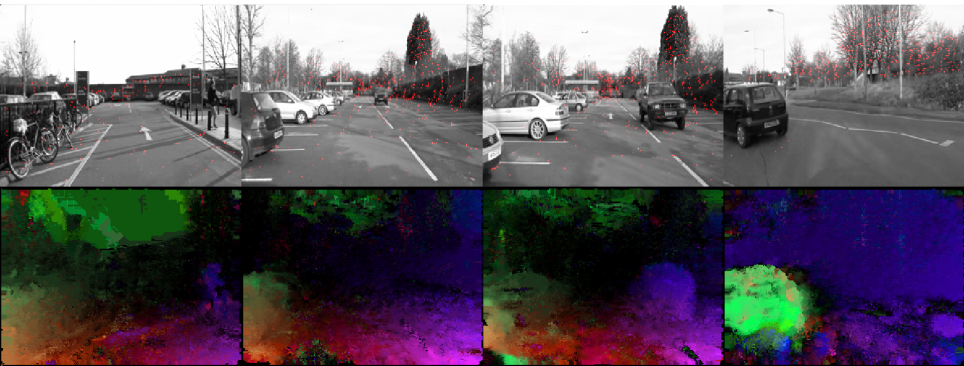


Figure: Motion vector field: Color encodes speed orientation, luminance is proportional to speed modulus

240x180 pixels @ 30 fps (GeForce GTX 280)

Culmg: A Small CUDA Image Processing Toolkit

Goals

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

Image Types

`host_image{2,3}d`

- Instances and pixel buffer lives on main memory
- Memory management using reference counting

`image{2,3}d`

- Instances lives on main memory, buffer on device memory
- Memory management using reference counting
- Implicitly converts to `kernel_image{2,3}d`

`kernel_image{2,3}d`

- Instances and pixel buffer lives on device memory
- Argument of CUDA kernel
- No memory management needed

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     image2d<float4> img_h(img.domain()); // light copy.
15
16     // Host device transfers
17     copy(img_h, img); // CPU -> GPU
18     copy(img, img_h); // GPU -> CPU
19
20 } // Images are freed automatically here.
21
22 // View a slice as a 2d image:
23 image3d<float4> img3d(100, 100, 100);
24 image2d<float4, simple_ptr> slice = img3d.slice(42);
```


Improved Built-In Types

CUDA built-in types

- `{char|uchar|short|ushort|int|uint|long|ulong|float}[1234]`
- `{double|longlong|ulonglong}[12]`
- Access to coordinates via: `bt.x`, `bt.y`, `bt.z`, `bt.w`
- No arithmetic operator

Culmg built-in improved types

- Inherit from CUDA built-in types
- Provide static type information such as `T::size` and `T::vtype`
- Access to coordinates via: `get<N>(bt)`
- Generic arithmetic operators
- No memory or run-time overhead
- Use `boost::typeof` to compute return type of generic operators

Improved Built-In Types: Example

CUDA built-ins

```
float4 a = make_float4(1.5f, 1.5f, 1.5f);  
int4 b   = make_int4(1, 2, 3);
```

```
float3 c;  
c.x = (a.x + b.x) * 2.5f;  
c.y = (a.y + b.y) * 2.5f;  
c.z = (a.z + b.z) * 2.5f;  
c.w = (a.w + b.w) * 2.5f;
```

Improved built-ins

```
i_float4 a(1.5, 1.5, 1.5);  
i_int4   b(2.5, 2.5, 2.5);  
  
i_float4 c = (a + b) * 2.5f;
```

Inputs

- Wrap OpenCV for use with Cimg image types
- Image and video

```
1 // Load 2d images.
2 host_image2d<uchar3> img = load_image("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
```

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 ptx 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

```

Pixel-Wise Kernels Generator

Goal

Use C++ to generate pixel-wise kernels

Classic CUDA code

```
// Declaration
template <typename T, typename U,
          typename V>
__global__
void simple_kernel(kernel_image2d<T> a,
                  kernel_image2d<U> b,
                  kernel_image2d<V> out)
{
    point2d<int> p = thread_pos2d();
    if (out.has(p))
        out(p) = a(p) * b(p) / 2.5f;
}

// Call
dim3 dimblock(16, 16);
dim3 dimgrid(divup(a.ncols(), 16),
             divup(a.nrows(), 16));
simple_kernel<<<dimgrid, dimblock>>>
            (a, b, c);
```

Using expression templates

```
image2d<i_int4>      a(200, 100);
image2d<i_short4>   b(a.domain());
image2d<i_float4>   c(a.domain());

// Generates and calls simple_kernel.
c = a * b / 2.5;

// BGR <=> RGB conversion
a = aggregate<int>::run(get_z(a),
                       get_y(a),
                       get_x(a), 1.0f);
```

Pixel-Wise Kernels Generator: Implementation

Expression templates build an expression tree

Type of `a * b / 2.5f` is `div<mult<image2d<A>, image2d >, float>`

Recursive tree evaluation is inlined by the compiler

```
template <typename I, typename E>
__global__ void pixel_wise_kernel(kernel_image2d<I> out, E e)
{
    i_int2 p = thread_pos2d();
    if (out.has(p))
        out(p) = e.eval(p); // E holds expression tree
}
```

operator= handles the kernel call

```
template <typename T, typename E>
inline void
image2d<T>::operator=(E& expr)
{
    [...]
    pixel_wise_kernel<<<<dimgrid, dimblock>>>>(*this, expr);
}
```

Future works

- Add convolutions to expression templates
- Image and video output
- Documentation and release?

Dige: Fast Prototyping of Graphical Interface

Our problem with image processing experiments

- Visualize image is not straightforward
- `mln::display` is convenient but is not well suited to visualize images in real time
- Interact with our code is hard. For example, tuning thresholds on the fly is hard or impossible without a GUI

In short

- Graphical interface creation
- Image visualization
- Event management
- No additional thread
- Non intrusive!
- Based on the Qt framework

Create GUI with only one C++ statement

```
Window("my_window", 400, 400) <<=
  vbox_start <<
    (hbox_start <<
      ImageView("my_view") / 3 <<
      Slider("my_slider", 0, 42, 21) / 1 <<
      hbox_end) / 3 <<
    Label("Hello LRDE") / 1 <<
  vbox_end;
```



Display your images with only one C++ statement...

```
dg::Window("my_window", 400, 400) <<=
    ImageView("my_view");

// Milena image
mln::image2d<rgb8> lena =
    io::ppm::load<rgb8>("lena.ppm");

ImageView("my_view") <<=
    dl() - lena - lena + lena;
```



... But

Dige is not magic, it does not anything about Milena image types. So we need to provide it with a bridge between its internal image type and `mln::image2d<rgb8>`

```
namespace dg
{
    image<trait::format::rgb, unsigned char>
    adapt(const mln::image2d<mln::value::rgb8>& i)
    {
        return image<trait::format::rgb, unsigned char>
            (i.ncols() + i.border() * 2, i.nrows() + i.border() * 2,
             (unsigned char*)i.buffer());
    }
}
```

Events management

- Wait for user interaction:

```
wait_event(key_press(key_enter));
```

- Create and watch event queues:

```
event_queue q(PushButton("button").click() | key_release(key_x));  
while (true)  
    if (!q.empty())  
    {  
        any_event e = q.pop_front();  
        if (e == PushButton("button").click())  
            std::cout << "button clicked" << std::endl;  
        if (e == key_release(key_x))  
            std::cout << "key x released" << std::endl;  
    }
```

- Event based loops:



```
for_each_event_until(PushButton("button").click(),  
                    key_release(key_enter))  
    std::cout << "button clicked" << std::endl;
```

The library

- Available: <https://gitorious.org/dige>
- Still under development
- 7k lines of code
- Portable (depends on Qt and boost)
- Should be released this year (2011)

Thank you for your attention.
Any questions?

References I

-  Antoine Manzanera, *Image representation and processing through multiscale local jet features*.
-  NVIDIA, *Nvidia cuda c programming guide*.