Recursive Gaussian Filter for Mathematica using CUDA

Patrick Scheibe

December 2008

Contents

1 Introduction

This implementation bases on [YvV95]. It demonstrates how to combine *Mathematica* and CUDA and it is written in CWEB. Three parts are important

- I need a *MathLink*-template to call the C-function from *Mathematica*. This connects parameters and function names of the corresponding parts in C and in *Mathematica*.
- A C-function is required which allocates stores for arrays, does some simple precomputations of the gaussian parameters and calls then the CUDA-kernel. After the computation is finished the results are send to *Mathematica*. I will call this the *host-code* since all this does not run on the gpu.
- The most important part is the CUDA-kernel which define what is done on the gpu in parallel. This part is called the *device-code*.

1 **#define** NUM_OF_THREADS 32

⟨ Headers 2 ⟩
⟨ CUDA Kernel for the Gaussian Filter 8 ⟩
⟨ Mathematica MathLink Template for the Gaussian Filter 4 ⟩
⟨ Gaussian Filter 5 ⟩
⟨ Main Loop 3 ⟩

¶ The includes.

2 ⟨ Headers 2 ⟩ ≡
```
#include <mathlink.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cutil.h>
#include <math.h>
```
This code is used in chunk 1.

¶ The main-loop for the *MathLink*-program. Note that this is only for the linux OS (maybe OSX too).

3 ⟨ Main Loop 3 ⟩ ≡
```
int main(int argc, char *argv[])
{
  CUT_DEVICE_INIT(argc, argv);
  return MLMain(argc, argv);
}
```
This code is used in chunk 1.

2 *MathLink*-template code

The Mathematica Template connects the C-function, *cudaGaussianFilter*(...), and the Mathematica function, *CudaGaussianFilter*[...], which is a lowlevel function since it gets not an image but a list of the pixel-values and the dimensions. Later I will wrap this lowlevel function with a highlevel version which is called with an image. In the line starting with *Pattern* you can see that I get the graylevel-bitmap *bm* as one-dimensional list from *Mathematica*. Since I only test whether *bm* is a list and not whether it is one-dimensional, I flatten it in the *Arguments* line. Furthermore, I ensure that the sigma is evaluated to a numerical value.

4 ⟨ Mathematica MathLink Template for the Gaussian Filter 4 ⟩ ≡

```
:Begin:
:Function:cudaGaussianFilter
:Pattern:CudaGaussianFilter[bm_List,nx_Integer,ny_Integer,sigma_?NumericQ]
:Arguments:{Flatten[bm],nx,ny,N[sigma]}
:ArgumentTypes:{RealList,Integer,Integer,Real}
:ReturnType:Manual
:End:
```

This code is used in chunk 1.

3 The Host-Code

Before calling the CUDA-kernel I have to allocate and initialise the arrays. In this step I transfer the image into the memory of the graphics-card. After that the whole filter consists of only 4 sequential steps where every step runs in parallel on the gpu:

1. All rows are processed separate with the method described in [YvV95]. This consists basically of a forward run followed by a backward run through the row.
2. Now all columns need to be processed. Therefore, the bitmap is transposed and
3. the first step is called again, working now on the columns (Transposing a matrix means first row becomes first column, second row becomes second column and so on).
4. A final transpose rotates the bitmap to its original form.

I end with returning the result to *Mathematica* and freeing all allocated memory.

```
5  ⟨ Gaussian Filter 5 ⟩ ≡
     void cudaGaussianFilter(double *h_bm, long n, int nx, int ny, double sigma)
     {
       double *d_bm, *d_bm_transposed, *h_res;      /* The array where the result is stored */
       double q;      /* An adapted version of the sigma */
       size_t stride, stride_tr;      /* The stride I have to use instead of ny */
       int blocksize,      /* How many thread per block */
       gridsize;      /* How many blocks in all */
       cudaError_t err;
       blocksize = NUM_OF_THREADS;
       gridsize = (ny % blocksize ≡ 0) ? ny/blocksize : ny/blocksize + 1;      /* make enough blocks
           if ny is not divisible by the choosen blocksize which is usually 32 */
       ⟨ Calculate the parameter for the gaussian filter 7 ⟩
       CUDA_SAFE_CALL(cudaMallocPitch((void **) &d_bm, &stride, nx * sizeof(double), ny));
       CUDA_SAFE_CALL(cudaMallocPitch((void **) &d_bm_transposed, &stride_tr,
           ny * sizeof(double), nx));
       CUDA_SAFE_CALL(cudaMemcpy2D((void *) d_bm, stride, (void *) h_bm,
           nx * sizeof(double), nx * sizeof(double), ny, cudaMemcpyHostToDevice));
       h_res = (double *) calloc(sizeof(double), nx * ny);
       cudaGaussKernel <<< gridsize, blocksize >>> (d_bm, stride, nx, ny, b0, b1, b2, b3, b4);
       cudaTransposeKernel <<< gridsize, blocksize >>> (d_bm, stride, nx, ny, d_bm_transposed,
           stride_tr);
       cudaGaussKernel <<< gridsize, blocksize >>> (d_bm_transposed, stride_tr, ny, nx, b0, b1, b2,
           b3, b4);
       cudaTransposeKernel <<< gridsize, blocksize >>> (d_bm_transposed, stride_tr, ny, nx, d_bm,
           stride);
       err = cudaGetLastError();
       if (cudaSuccess ≠ err) {
         fprintf(stderr, "Cuda error: %s.\n", cudaGetErrorString(err));
         exit(EXIT_FAILURE);
       }
       CUDA_SAFE_CALL(cudaMemcpy2D((void *) h_res, nx * sizeof(double), (void *)
           d_bm, stride, nx * sizeof(double), ny, cudaMemcpyDeviceToHost));
       MLPutRealList(stdlink, h_res, nx * ny);
       cudaFree(d_bm);
```

```
    cudaFree(d_bm_transposed);
    free(h_res);
}
```
This code is used in chunk 1.

4 The Device Code

This part consists of two *Kernels* (functions running on the gpu), one for the transposition of a matrix and one for the gaussian filter. Note that a kernel is (here) a chunk of code which processes exactly one row (column) of the bitmap. CUDA will run many instances of the kernel in parallel. You may ask how CUDA know which kernel should process which row. This is done through the *blockIdx*, *threadIdx* and *blockDim* variables which "enumerate" every gpu-processor uniquely (this is very unprecise in terms of CUDAs Processor, Thread, Block and Grid vocabulary).

¶ I start with the calculation of the q of (8c). Further details can be found in the section *Choosing q* in section 4 of [YvV95].

7 ⟨ Calculate the parameter for the gaussian filter 7 ⟩ ≡

```
if (sigma < 0.5) {      /* The quotient q would be 0, so we can stop. */
  MLPutSymbol(stdlink, "$Failed");
  return;
}
else if (sigma ≥ 0.5 ∧ sigma ≤ 2.5) {
  q = 3.97156 − 4.14554 * sqrt(1.0 − 0.26891 * sigma);
}
else if (sigma > 2.5) {
  q = −0.9633 + 0.98711 * sigma;
}
else q = sigma;
double b0 = 1.57825 + q * (2.44413 + (1.4281 + 0.422205 * q) * q);
double b1 = q * (2.44423 + q * (2.85619 + 1.26661 * q));
double b2 = (−1.4281 − 1.26661 * q) * q * q;
double b3 = 0.422205 * q * q * q;
double b4 = 1.0 − (b1 + b2 + b3)/b0;
```

This code is used in chunk 5.

¶ The algorithm for one row is straight forward. Since the formula for a pixel needs the (already calculated) last three pixel values, the boundary must be handled separately. I assumed missing pixel to be of the same value like the boundary pixel and I process these three by hand.

8 ⟨ CUDA Kernel for the Gaussian Filter 8 ⟩ ≡

```
__global__ void cudaGaussKernel(double *d_bm, size_t stride, int nx, int ny, double
        b0, double b1, double b2, double b3, double B)
{
  int pos = blockIdx.x * blockDim.x + threadIdx.x;
  if (pos ≥ ny ∨ pos < 0) return;
  double pV;      /* The value which is used for padding */
    /* Forward iteration. Calculating the boundary-elements by hand. */
  double *in = (double *)((char *) d_bm + pos * stride);
  pV = *in;
  in[0] = B * pV + (b1 * pV + b2 * pV + b3 * pV)/b0;
  in[1] = B * in[1] + (b1 * in[0] + b2 * pV + b3 * pV)/b0;
  in[2] = B * in[2] + (b1 * in[1] + b2 * in[0] + b3 * pV)/b0;
```

```
  for (int i = 3; i < nx; ++i)
    in[i] = B * in[i] + (b1 * in[i - 1] + b2 * in[i - 2] + b3 * in[i - 3])/b0;
      /* Backward iteration. Calculating the boundary-elements by hand. */
  int r = nx - 1;
  pV = in[r];
  in[r] = B * pV + (b1 * pV + b2 * pV + b3 * pV)/b0;
  in[r - 1] = B * in[r - 1] + (b1 * in[r] + b2 * pV + b3 * pV)/b0;
  in[r - 2] = B * in[r - 2] + (b1 * in[r - 1] + b2 * in[r] + b3 * pV)/b0;
  for (int i = r - 3; i ≥ 0; --i)
    in[i] = B * in[i] + (b1 * in[i + 1] + b2 * in[i + 2] + b3 * in[i + 3])/b0;
}
```

See also chunk 9.

This code is used in chunk 1.

¶ The last part is the transposition of the bitmap. Here I just allocated another array and every row is copied into the appropriate column. If you don't know why the *strides* are needed, check the documentation of *cudaMallocPitch*.

9 ⟨ CUDA Kernel for the Gaussian Filter 8 ⟩ +≡

```
__global__ void cudaTransposeKernel(double *in, size_t stride1, int nx, int ny, double
        *out, size_t stride2)
{
  int row = blockIdx.x * blockDim.x + threadIdx.x;
  if (row ≥ ny) return;
  double *r = (double *)((char *) in + stride1 * row);
  for (int i = 0; i < nx; ++i) {
    double *outRow = (double *)((char *) out + i * stride2);
    outRow[row] = r[i];
  }
}
```

5 Appendix

¶ The compilation of the `CWEB` file into an executable constists of several steps. First you need to *tangle* your .w file

```
ctangle -bh cudaRecursiveGaussianFilter.w - cudaRecursiveGaussianFilter.tm
```

This step produces a *Mathematica*-template file which needs to be processed by *mprep*

```
mprep -o cudaRecursiveGaussianFilter.cu cudaRecursiveGaussianFilter.tm
```

mprep can be found in the *Mathematica*-install path under (on my 64 bit Linux box) `SystemFiles/Links/MathLink/DeveloperKit/Linux-x86-64/CompilerAdditions`. This step is followed by the call of the CUDA-compiler. I added several options

- `-arch compute_13` and `-gpu-code sm_13` forces CUDA to take the version 1.3. In older versions and on other graphic cards the **double** data type is not supported. Therefore, this option seems essential
- `--optimize 3` is for optimized code.
- `--machine 64` to build a 64-bit program.

The CUDA-compiler is called by

```
nvcc \
-arch compute_13 \
--optimize 3 \
--machine 64 \
--gpu-code sm_13 \
-o fastgauss.exe \
-I/usr/local/Wolfram/Mathematica/6.0/SystemFiles/\
Links/MathLink/DeveloperKit/Linux-x86-64/CompilerAdditions \
-L/usr/local/Wolfram/Mathematica/6.0/SystemFiles/Links/\
MathLink/DeveloperKit/Linux-x86-64/CompilerAdditions \
-lm \
-lpthread \
-lML64i3 \
-I ~/NVIDIA_CUDA_SDK/common/inc/ \
-L ~/NVIDIA_CUDA_SDK/lib \
-lcutil \
cudaRecursiveGaussianFilter.cu
```

¶

References

[YvV95] I.T. Young and L.J. van Vliet. Recursive implementation of the Gaussian filter. *Signal Processing*, 44(2):139–151, 1995.

Index

List of Refinements

⟨ CUDA Kernel for the Gaussian Filter 8, 9 ⟩ Used in chunk 1.
⟨ Calculate the parameter for the gaussian filter 7 ⟩ Used in chunk 5.
⟨ Gaussian Filter 5 ⟩ Used in chunk 1.
⟨ Headers 2 ⟩ Used in chunk 1.
⟨ Main Loop 3 ⟩ Used in chunk 1.
⟨ Mathematica MathLink Template for the Gaussian Filter 4 ⟩ Used in chunk 1.