

How to combine NVIDIAs **CUDA** and *Mathematica*.

Source-code for the presentation examples

Patrick Scheibe

September 2009

Contents

1	Introduction	2
2	Building and installation	3
3	Global structure of the file	4
4	The includes and the main function	5
5	Initialization of the CUDA-devices and information about them	6
6	A gaussian image filter	10
7	Rigid transformations for images	15
8	Distance measure for images	19
9	Initializing and freeing the global images	22
10	Reconstruction filter	25
11	Julia Sets	26
12	Appendix	31

1 Introduction

This is the `CWEB`-code for the *MathLink*-program which is required to run the CUDA-examples from the presentation I gave at the 2nd *Mathematica*-day in Leipzig 2009. It consists of function which are required to show examples for the julia-set and a rigid image registration. Furthermore there are some functions for handling the CUDA-device from *Mathematica*.

2 Building and installation

To build the program for your architecture you have to install the CUDA-software which typically includes a driver for your GPU, a software development kit (SDK) and the cuda examples. All of this can be found at www.nvidia.com/object/cuda_learn.html. It is not really necessary to have a CUDA-enabled graphics card since you can compile the stuff in emulation mode and run it on the CPU.

Furthermore you need the CWEB program which enables you to transform the *cudaDip.w* file into a source-code file which can be compiled by the CUDAcompiler *nvcc*. Donald Knuth's CWEB can be found for instance under www.literateprogramming.com/cweb_download.html.

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

```
ctangle -bh cudaDip.w - cudaDip.tm
```

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

```
mprep -o cudaDip.cu cudaDip.tm
```

mprep can be found in the *Mathematica*-install path under (on my OSX box)

/Applications/Mathematica.app/SystemFiles/Links/MathLink/DeveloperKit/CompilerAdditions.

This step is followed by the call of the CUDA-compiler.

The CUDA-compiler is called by

```
nvcc \  
--optimize 3 \  
-o cudaDip.exe \  
-I/Applications/Mathematica.app/SystemFiles/  
Links/MathLink/DeveloperKit/CompilerAdditions\  
-L/Applications/Mathematica.app/SystemFiles/  
Links/MathLink/DeveloperKit/CompilerAdditions\  
-lm \  
-lpthread \  
-lML64i3 \  
--linker-options "-rpath /usr/local/cuda/lib" \  
cudaDip.cu
```

To load the compiled *MathLink*-program with the CUDA.m package you have to put it in a subdirectory of the *Binaries* directory which is named exactly like your `$SystemID` variable in *Mathematica*. Then the whole ImageProcessing directory, containing the Binaries-dir with the executables and the CUDA.m file has to be placed where *Mathematica* can find it. Check the `$Path` to select such a place. You should prefer something like (on a OSX)

```
/Users/patrick/Library/Mathematica/Applications
```

3 Global structure of the file

```
3 #define NUM_OF_THREADS 32
#define FLOAT_EPS 1.19209290 · 10-7 F
#define TINY 1 · 10-6 F
    ⟨ Headers and main 4 ⟩
    ⟨ Reconstruction filter 29 ⟩
    ⟨ Functions for initialization and informations about the CUDAdevices 5 ⟩
    ⟨ Gaussian filter 11 ⟩
    ⟨ Rigid transformation for images 17 ⟩
    ⟨ Distance measure for images 24 ⟩
    ⟨ Initializing and freeing the global images 27 ⟩
    ⟨ Julia set 31 ⟩
```

4 The includes and the main function

Beside the includes, I define the global variables which are used for the rigid image registration. The images `D_SIMG` and `D_TIMG` are held on the gpu during the registration so that I don't have to transfer so much data.

```
4 <Headers and main 4> ≡
#include <mathlink.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
float *D_SIMG =  $\Lambda$ , *D_TIMG =  $\Lambda$ , *D_DIFF =  $\Lambda$ ;
size_t STRIDE_SIMG, STRIDE_TIMG;
int NX, NY;
const double SIGMA = 5.0;
int counter; /* only for testing purpose because of the NMinimize bug */
int main(int argc, char *argv[])
{
    return MLMain(argc, argv);
}
```

This code is used in chunk 3.

5 Initialization of the CUDA-devices and information about them

5 \langle Functions for initialization and informations about the CUDA devices 5 $\rangle \equiv$
 \langle Init CUDA 6 \rangle
 \langle Get information about the running cuda 7 \rangle
 \langle Deinitialize the runtime 10 \rangle

This code is used in chunk 3.

¶ This code is basically copied from the *cutil* library. First we check how many CUDA-enabled devices the system has and then we try to activate device number n . If n is larger then the number of devices-1 we use the first one.

6 \langle Init CUDA 6 $\rangle \equiv$

```

:Begin:
:Function:cudaInitDevice
:Pattern:mlCudaInitDevice[n_Integer]
:ReturnType:Manual
:Arguments:{n}
:ArgumentTypes:{Integer}
:End:

void cudaInitDevice(int n)
{
#ifdef __DEVICE_EMULATION__
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);
    if (deviceCount == 0) {
        fprintf(stderr, "ERROR: No CUDA-enabled devices detected.\n");
        MLPutString(stdlink, "ERROR: No CUDA-enabled devices detected.");
        MLPutSymbol(stdlink, "$Failed");
        return;
    }
    if (n > deviceCount - 1) n = deviceCount - 1;
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, n);
    if (deviceProp.major < 1) {
        fprintf(stderr,
            "ERROR: The selected device does not support at least CUDA v\
            ersion > 1.\n");
        MLPutFunction(stdlink, "List", 2);
        MLPutSymbol(stdlink, "$Failed");
        MLPutString(stdlink, "ERROR: The selected device does not sup\
            port at least CUDA version > 1.");
        return;
    }
    fprintf(stderr, "Using device: %s.\n", deviceProp.name);
    cudaSetDevice(n);

```

```

        MLPutFunction(stdlink, "List", 2);
        MLPutSymbol(stdlink, "$cudaSuccess");
        MLPutString(stdlink, deviceProp.name);
        return;
#else
        MLPutFunction(stdlink, "List", 2);
        MLPutSymbol(stdlink, "$cudaSuccess");
        MLPutString(stdlink,
            "This binary was compiled in \"device_emulation_mode\");
        return;
#endif
}

```

This code is used in chunk 5.

¶ Here we I'm collecting information about the installed devices.

7 <Get information about the running cuda 7> ≡

```

:Begin:
:Function:getDeviceInformation
:Pattern:mlCudaGetDeviceInformation[]
:ReturnType:Manual
:Arguments:{}
:ArgumentTypes:{}
:End:

void getDeviceInformation()
{
    cudaError_t err;
    int deviceCount;
    err = cudaGetDeviceCount(&deviceCount);
    <check for errors and when they happen return to Mathematica 37>
    if (err != cudaSuccess) return;
    MLPutFunction(stdlink, "List", deviceCount);
    for (int d = 0; d < deviceCount; ++d) {
        struct cudaDeviceProp prop;
        err = cudaGetDeviceProperties(&prop, d);
        <check for errors and when they happen return to Mathematica 37>
        <send CUDAdevice-information to Mathematica 9>
    }
}

```

See also chunk 8.

This code is used in chunk 5.

¶ This function is equivalent to *getDeviceInformation* but it extracts only information for the currently activated device.

8 <Get information about the running cuda 7> +≡

```

:Begin:

```



```

:Function:_getCurrentDevice :Pattern:_mlCudaGetCurrentDevice[]
:Arguments:{}
:ArgumentTypes:{}
:ReturnType:Manual
:End:
void getCurrentDevice()
{
    cudaError_t err;
    int dev = 0;
    struct cudaDeviceProp prop;
    err = cudaGetDevice(&dev);
    <check for errors and when they happen return to Mathematica 37>
    if (err != cudaSuccess) return;
    err = cudaGetDeviceProperties(&prop, dev);
    <check for errors and when they happen return to Mathematica 37>
    if (err != cudaSuccess) return;
    <send CUDAdevice-information to Mathematica 9>
}

```

¶ Sending the property struct to *Mathematica*.

```

9 <send CUDAdevice-information to Mathematica 9> ≡
    MLPutFunction(stdlink, "List", 16);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutString(stdlink, "Name");
    MLPutString(stdlink, prop.name);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutString(stdlink, "TotalGlobalMem");
    MLPutInteger(stdlink, prop.totalGlobalMem);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutString(stdlink, "SharedMemPerBlock");
    MLPutInteger(stdlink, prop.sharedMemPerBlock);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutString(stdlink, "RegsPerBlock");
    MLPutInteger(stdlink, prop.regsPerBlock);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutString(stdlink, "WarpSize");
    MLPutInteger(stdlink, prop.warpSize);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutString(stdlink, "MemPitch");
    MLPutInteger(stdlink, prop.memPitch);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutString(stdlink, "MaxThreadsPerBlock");
    MLPutInteger(stdlink, prop.maxThreadsPerBlock);
    MLPutFunction(stdlink, "Rule", 2);
    MLPutString(stdlink, "MaxThreadsDim");
    MLPutIntegerList(stdlink, prop.maxThreadsDim, 3);
    MLPutFunction(stdlink, "Rule", 2);

```

```

MLPutString(stdlink, "MaxGridSize");
MLPutIntegerList(stdlink, prop.maxGridSize, 3);
MLPutFunction(stdlink, "Rule", 2);
MLPutString(stdlink, "TotalConstMem");
MLPutInteger(stdlink, prop.totalConstMem);
MLPutFunction(stdlink, "Rule", 2);
MLPutString(stdlink, "Major");
MLPutInteger(stdlink, prop.major);
MLPutFunction(stdlink, "Rule", 2);
MLPutString(stdlink, "Minor");
MLPutInteger(stdlink, prop.minor);
MLPutFunction(stdlink, "Rule", 2);
MLPutString(stdlink, "ClockRate");
MLPutInteger(stdlink, prop.clockRate);
MLPutFunction(stdlink, "Rule", 2);
MLPutString(stdlink, "TextureAlignment");
MLPutInteger(stdlink, prop.textureAlignment);
MLPutFunction(stdlink, "Rule", 2);
MLPutString(stdlink, "DeviceOverlap");
MLPutInteger(stdlink, prop.deviceOverlap);
MLPutFunction(stdlink, "Rule", 2);
MLPutString(stdlink, "MultiProcessorCount");
MLPutInteger(stdlink, prop.multiProcessorCount);

```

This code is used in chunks 7 and 8.

¶ [cud, p. 323] Before uninstalling the *MathLink* program it's maybe a good idea to exit the CUDARuntime

10 < Deinitialize the runtime 10 > ≡

```

:Begin:
:Function:exitCudaRuntime
:Pattern:m!ExitCudaRuntime[]
:Arguments:{}
:ArgumentTypes:{}
:ReturnType:Manual
:End:

void exitCudaRuntime()
{
    cudaError_t err;
    err = cudaThreadExit();
    < check for errors and when they happen return to Mathematica 37 >
    if (err != cudaSuccess) {
        MLPutSymbol(stdlink, "$Failed");
    }
    MLPutSymbol(stdlink, "$Success");
}

```

This code is used in chunk 5.

6 A gaussian image filter

This is an implementation of the method by Young and van Vliet [1995]. The filter consists of 4 sequential steps where every step runs in parallel on the gpu:

1. All rows are processed separate with the method described in [Young and van Vliet, 1995]. 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.

```
11 < Gaussian filter 11 > ≡
    < CUDA-kernels for the gaussian filter 15 >
    < Mathematica template and wrapper for the gaussian filter 12 >
```

This code is used in chunk 3.

¶ Implementation for capability 1.3

```
12 < Mathematica template and wrapper for the gaussian filter 12 > ≡
    :Begin:
    :Function: cudaGaussianFilter
    :Pattern: mlCudaGaussianFilter[bm_List, nx_Integer, ny_Integer, sigma_?NumericQ]
    :Arguments: {Flatten[bm], nx, ny, N[sigma]}
    :ArgumentTypes: {RealList, Integer, Integer, Real}
    :ReturnType: Manual
    :End: 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 14 >
        cudaMallocPitch((void **) &d_bm, &stride, nx * sizeof(double), ny);
        cudaMallocPitch((void **) &d_bm_transposed, &stride_tr, ny * sizeof(double), nx);
        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);
    }
    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);
}

```

See also chunk 13.

This code is used in chunk 11.

¶ The interface for CUDAversion 1.1 where no doubles are available. Since *Mathematica* sends per default only doubles when giving lists of real-numbers I have to fetch the Real32List. A much smarter approach is possible here but since this is only for presentation, I keep the code small.

13 < *Mathematica* template and wrapper for the gaussian filter 12 > +≡

```

:Begin:
:Function:cudaGaussianFilter
:Pattern:mlCudaGaussianFilterFloat[bm_List,nx_Integer,ny_Integer,sigma_?NumericQ]
:Arguments:{Flatten[bm],nx,ny,N[sigma]}
:ArgumentTypes:{Real32List,Integer,Integer,Real}
:ReturnType:Manual
:End: void cudaGaussianFilter(float *h_bm,int n,int nx,int ny,double sigma)
{
    float *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 14 >
    cudaMallocPitch((void **) &d_bm,&stride, nx * sizeof(float), ny);
    cudaMallocPitch((void **) &d_bm_transposed,&stride_tr, ny * sizeof(float), nx);
    cudaMemcpy2D((void *) d_bm, stride, (void *) h_bm, nx * sizeof(float),
        nx * sizeof(float), ny, cudaMemcpyHostToDevice);
}

```

```

    h_res = (float *) calloc(sizeof(float), nx * ny);
    float fb0 = (float) b0, fb1 = (float) b1, fb2 = (float) b2, fb3 = (float) b3,
          fb4 = (float) b4;
    cudaGaussKernel <<< gridsize, blocksize >>> (d_bm, stride, nx, ny, fb0, fb1, fb2, fb3, fb4);
    err = cudaThreadSynchronize();
    <check for errors and when they happen return to Mathematica 37>
    cudaTransposeKernel <<< gridsize, blocksize >>> (d_bm, stride, nx, ny, d_bm_transposed,
        stride_tr);
    err = cudaThreadSynchronize();
    <check for errors and when they happen return to Mathematica 37>
    cudaGaussKernel <<< gridsize, blocksize >>> (d_bm_transposed, stride_tr, ny, nx, fb0,
        fb1, fb2, fb3, fb4);
    err = cudaThreadSynchronize();
    <check for errors and when they happen return to Mathematica 37>
    cudaTransposeKernel <<< gridsize, blocksize >>> (d_bm_transposed, stride_tr, ny, nx,
        d_bm, stride);
    err = cudaThreadSynchronize();
    <check for errors and when they happen return to Mathematica 37>
    err = cudaMemcpy2D((void *) h_res, nx * sizeof(float), (void *)
        d_bm, stride, nx * sizeof(float), ny, cudaMemcpyDeviceToHost);
    <check for errors and when they happen return to Mathematica 37>
    MLPutReal32List(stdlink, h_res, nx * ny);
    cudaFree(d_bm);
    cudaFree(d_bm_transposed);
    free(h_res);
}

```

¶ I start with the calculation of the q of (8c). Further details can be found in the section *Choosing q* in section 4 of [Young and van Vliet, 1995].

```

14 <Calculate the parameter for the gaussian filter 14> ≡
    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 chunks 12, 13, and 28.

¶ 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. There are two versions of the kernel since I already had the **double** version and for the MacBook CUDA-capability 1.1 (no doubles, only float) I just copied it.

```

15 < CUDA-kernels for the gaussian filter 15 > ≡
    __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;
    }
    __global__ void cudaGaussKernel(float *d_bm, size_t stride, int nx, int ny, float b0, float
        b1, float b2, float b3, float B)
    {
        int pos = blockIdx.x * blockDim.x + threadIdx.x;
        if (pos ≥ ny ∨ pos < 0) return;
        float pV; /* The value which is used for padding */
        /* Forward iteration. Calculating the boundary-elements by hand. */
        float *in = (float *)((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 16.

This code is used in chunk 11.

¶ 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*.

```

16 < CUDA-kernels for the gaussian filter 15 > +=
    __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];
        }
    }
    __global__ void cudaTransposeKernel(float *in, size_t stride1, int nx, int ny, float
        *out, size_t stride2)
    {
        int row = blockIdx.x * blockDim.x + threadIdx.x;
        if (row ≥ ny) return;
        float *r = (float *)((char *) in + stride1 * row);
        for (int i = 0; i < nx; ++i) {
            float *outRow = (float *)((char *) out + i * stride2);
            outRow[row] = r[i];
        }
    }
}

```

7 Rigid transformations for images

- 17 \langle Rigid transformation for images 17 $\rangle \equiv$
 \langle Rigid transformation CUDA-kernel 19 \rangle
 \langle Rigid transformation *Mathematica* template and wrapper 18 \rangle
This code is used in chunk 3.

¶ The wrapper for the rigid transform of an image. This takes an image from *Mathematica* and transforms it by

$$\vec{x}' = A\vec{x} + \vec{b} \quad (1)$$

The parameters of the transformation are stored in *param*. The content of the *param* list depends on the type of the transformation. The following is possible:

1. pure rotation
2. rotation and translation
3. rotation, scaling and translation
4. general affine

- 18 \langle Rigid transformation *Mathematica* template and wrapper 18 $\rangle \equiv$
- ```

:Begin:
:Function: mlRigidTransform
:Pattern: mlCudaRigidTransform[bm_List, nx_Integer, ny_Integer, nc_Integer, param_List, t_Integer]
:Arguments: {Flatten[bm], nx, ny, nc, param, t}
:ArgumentTypes: {Real32List, Integer, Integer, Integer, Real32List, Integer}
:ReturnType: Manual
:End: void mlRigidTransform(float *bm, int n, int nx, int ny, int nc, float *param, int
 paramLength, int transformation_type)
{
 float *d_bm, *h_result, *d_result, *d_param;
 size_t pitchBm, pitchResult;
 cudaError_t err;
 int blocksize = NUM_OF_THREADS;
 int 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 */
 err = cudaMallocPitch((void **) &d_bm, &pitchBm, nx * sizeof(float), ny);
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 err = cudaMallocPitch((void **) &d_result, &pitchResult, nx * sizeof(float), ny);
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 err = cudaMalloc((void **) &d_param, paramLength * sizeof(float));
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 err = cudaMemcpy(d_param, param, paramLength * sizeof(float),
 cudaMemcpyHostToDevice);
 h_result = (float *) malloc(nx * ny * nc * sizeof(float));
#ifdef __DEVICE_EMULATION__

```



```

 fprintf(stderr, "nx: %d ny: %d nc: %d\n", nx, ny, nc);
#endif
 for (int c = 0; c < nc; ++c) {
 err = cudaMemcpy2D(d_bm, pitchBm, (bm + c * nx * ny), nx * sizeof(float),
 nx * sizeof(float), ny, cudaMemcpyHostToDevice);
 <check for errors and when they happen return to Mathematica 37>
 cudaRigidTransform(<< gridsize, blocksize >>> (d_bm, pitchBm, d_result, pitchResult,
 nx, ny, d_param, transformation_type);
 err = cudaGetLastError();
 <check for errors and when they happen return to Mathematica 37>
 err = cudaThreadSynchronize();
 <check for errors and when they happen return to Mathematica 37>
 err = cudaMemcpy2D((h_result + c * nx * ny), nx * sizeof(float), d_result,
 pitchResult, nx * sizeof(float), ny, cudaMemcpyDeviceToHost);
 <check for errors and when they happen return to Mathematica 37>
 }
 #if defined (__DEVICE_EMULATION__) ^ defined (__PRINT__)
 for (int i = 0; i < nx * ny; ++i)
 fprintf(stderr, "(%lf->%lf)\n", bm[i], h_result[i]);
 #endif
 err = cudaFree(d_bm);
 <check for errors and when they happen return to Mathematica 37>
 err = cudaFree(d_result);
 <check for errors and when they happen return to Mathematica 37>
 err = cudaFree(d_param);
 <check for errors and when they happen return to Mathematica 37>
 if (!MLPutReal32List(stdlink, h_result, nx * ny * nc)) exit(EXIT_FAILURE);
 free(h_result);
}

```

This code is used in chunk 17.

¶ Transforming the src image. Here we transform the image by an affine transformation. This could either be a pure rotation, a rotation with translation, a rotation-scaling-translation or a general affine transformation. The parameters of the transformation are stored in *p* and are explained in the appropriate sections.

```

19 <Rigid transformation CUDA-kernel 19> ≡
 #define ROTATION 1
 #define ROTATION_TRANSLATION 2
 #define ROTATION_SCALING_TRANSLATION 3
 #define AFFINE 4
 __global__ void cudaRigidTransform(float *src, size_t pitchSrc, float *dst, size_t pitchDst, int
 nx, int ny, float *p, int transformation_type)
 {
 int yi = blockIdx.x * blockDim.x + threadIdx.x;
 if (yi ≥ ny) return;
 float *dstRow = (float *)((char *) dst + pitchDst * yi);
 switch (transformation_type) {
 case ROTATION: <Calculate row with a pure rotation 20>
 break;

```

```

 case ROTATION_TRANSLATION: < Calculate row with a rotation and translation 21 >
 break;
 case ROTATION_SCALING_TRANSLATION:
 < Calculate row with a rotation, scaling and translation 22 >
 break;
 case AFFINE: < Calculate row with a general affine transformation 23 >
 break;
}
}

```

This code is used in chunk 17.

¶ Rotation. The angle phi is stored in p[0].

```

20 < Calculate row with a pure rotation 20 > ≡
{
 float y, cosphi, sinphi, x, xt, yt;
 y = yi;
 cosphi = cos(p[0]);
 sinphi = sin(p[0]);
 for (int xi = 0; xi < nx; ++xi) {
 x = xi;
 xt = 0.5 * (nx - cosphi * nx - ny * sinphi + 2 * cosphi * x + 2 * sinphi * y);
 yt = 0.5 * (ny + sinphi * (nx - 2 * x) - cosphi * (ny - 2 * y));
 dstRow[xi] = interpolate(src, pitchSrc, nx, ny, xt, yt);
 }
}

```

This code is used in chunk 19.

¶ Rotation, translation. The angle phi is stored in p[0] and the the translation-vector is (p[1],p[2]).

```

21 < Calculate row with a rotation and translation 21 > ≡
{
 float y, cosphi, sinphi, x, xt, yt, tx, ty;
 y = yi;
 cosphi = cos(p[0]);
 sinphi = sin(p[0]);
 tx = p[1];
 ty = p[2];
 for (int xi = 0; xi < nx; ++xi) {
 x = xi;
 xt = 0.5 * (nx - cosphi * (nx + 2 * tx - 2 * x) - sinphi * (ny + 2 * ty - 2 * y));
 yt = 0.5 * (ny + sinphi * (nx + 2 * tx - 2 * x) - cosphi * (ny + 2 * ty - 2 * y));
 dstRow[xi] = interpolate(src, pitchSrc, nx, ny, xt, yt);
 }
}

```

This code is used in chunk 19.

¶ Rot., Transl and Scaling. Here we have phi in p[0], scaling factors in p[1], p[2] and translation in p[3] and p[4]. For the scaling  $s_x$  and  $s_y$  I have to check whether they are too small since they are used in the denominator of the formula.

22  $\langle$  Calculate row with a rotation, scaling and translation 22  $\rangle \equiv$

```
{
 float y, cosphi, sinphi, x, xt, yt, sx, sy, tx, ty;
 y = yi;
 cosphi = cos(p[0]);
 sinphi = sin(p[0]);
 sx = fabs(p[1]) < FLOAT_EPS ? TINY : p[1];
 sy = fabs(p[2]) < FLOAT_EPS ? TINY : p[2];
 tx = p[3];
 ty = p[4];
 for (int xi = 0; xi < nx; ++xi) {
 x = xi;
 xt = 0.5 * (nx - (cosphi * (nx + 2.0 * tx - 2.0 * x)) / sx - (sinphi * (ny + 2.0 * ty - 2.0 * y)) / sy);
 yt = 0.5 * (ny * sx * sy + sinphi * sy * (nx + 2.0 * tx - 2.0 * x) - cosphi * sx * (ny + 2.0 *
 ty - 2.0 * y)) / (sx * sy);
 dstRow[xi] = interpolate(src, pitchSrc, nx, ny, xt, yt);
 }
}
```

This code is used in chunk 19.

¶ General affine transformation by completely inverting formula.

23  $\langle$  Calculate row with a general affine transformation 23  $\rangle \equiv$

```
{
 float det, y, x, xt, yt;
 float a11 = p[0], a12 = p[1], a21 = p[2], a22 = p[3], tx = p[4], ty = p[5];
 det = a11 * a22 - a12 * a21;
 det = fabs(det) < FLOAT_EPS ? TINY : det;
 y = yi;
 for (int xi = 0; xi < nx; ++xi) {
 x = xi;
 xt = (a12 * a21 * nx + a22 * nx - a11 * a22 * nx - a12 * ny + 2 * a22 * tx - 2 * a12 * ty -
 2 * a22 * x + 2 * a12 * y) / (-2.0 * det);
 yt = -(a21 * nx - a11 * ny - a12 * a21 * ny + a11 * a22 * ny + 2 * a21 * tx - 2 * a11 *
 ty - 2 * a21 * x + 2 * a11 * y) / (-2 * det);
 dstRow[xi] = interpolate(src, pitchSrc, nx, ny, xt, yt);
 }
}
```

This code is used in chunk 19.

## 8 Distance measure for images

24  $\langle$ Distance measure for images 24 $\rangle \equiv$   
 $\langle L^2$  norm CUDA-kernel 26 $\rangle$   
 $\langle L^2$  norm *Mathematica* template and wrapper 25 $\rangle$   
This code is used in chunk 3.

¶ This applies a rigid transformation to the global template image D\_TIMG and calculates with the result the  $L^2$ -norm of the two images.

25  $\langle L^2$  norm *Mathematica* template and wrapper 25 $\rangle \equiv$

```

:Begin:
:Function:_mlCudaGetDifference
:Pattern:_mlCudaGetDifference[param_List,_type_Integer]
:Arguments:{param,_type}
:ArgumentTypes:{Real32List,Integer}
:ReturnType:Real32
:End:

float mlCudaGetDifference(float *param,int n,int type)
{
 float *h_diff, *d_param;
 cudaError_t err;
 if (! (D_SIMG ^ D_TIMG ^ D_DIFF)) return -1.0;
 int blocksize = NUM_OF_THREADS; /* How many thread per block */
 int gridsize = (NY % blocksize == 0) ? NY/blocksize : NY/blocksize + 1;
 /* How many blocks in all */
 float *d_result;
 size_t pitchResult;
 err = cudaMalloc((void **) &d_param,n * sizeof(float));
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 err = cudaMemcpy((void *) d_param,(void *) param,n * sizeof(float),
 cudaMemcpyHostToDevice);
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 err = cudaMallocPitch((void **) &d_result,&pitchResult,NX * sizeof(float),NY);
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 fprintf(stderr,"Exec_Config:_%d,_%d\n",gridsize,blocksize);
 cudaRigidTransform <<< gridsize,blocksize >>> (D_TIMG,STRIDE_TIMG,d_result,
 pitchResult,NX,NY,d_param,type);
 err = cudaGetLastError();
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 err = cudaThreadSynchronize();
 \langle check for errors and when they happen return to Mathematica 37 \rangle
#ifdef __DEVICE_EMULATION__
 fprintf(stderr,"Image:\n");
 for (int j = 0; j < NY; j++) {
 fprintf(stderr,"\n");
 for (int i = 0; i < NX; i++) {

```

```

 fprintf(stderr, "%f", d_result[j * NX + i]);
 }
}
#endif
 cudaDifferenceKernel <<< gridsize, blocksize >>> (D_SIMG, STRIDE_SIMG, d_result,
 pitchResult, D_DIFF, NX, NY);
 err = cudaGetLastError();
 <check for errors and when they happen return to Mathematica 37>
 err = cudaThreadSynchronize();
 <check for errors and when they happen return to Mathematica 37>
 h_diff = (float *) calloc(NY, sizeof(float));
 err = cudaMemcpy((void *) h_diff, (void *) D_DIFF, NY * sizeof(float),
 cudaMemcpyDeviceToHost);
 <check for errors and when they happen return to Mathematica 37>
 float difference = 0.0;
 for (int i = 0; i < NY; ++i) {
 difference += h_diff[i];
 }
 free(h_diff);
 err = cudaFree(d_result);
 <check for errors and when they happen return to Mathematica 37>
#ifdef __DEVICE_EMULATION__
 counter++;
 fprintf(stderr, "%d->%lf\n", counter, difference);
#endif
 return difference;
}

```

This code is used in chunk 24.

¶ Calculating the  $L^2$  norm of the stored *simg* and *timg*. In every *cudaDifferenceKernel* the sum of squared difference of one row of *simg* and *timg* is calculated.

26 <  $L^2$  norm CUDA-kernel 26 >  $\equiv$

```

__global__ void cudaDifferenceKernel(float *simg, size_t pitchSimg, float *timg, size_t
 pitchTimg, float *diffVector, int nx, int ny)
{
 int yi = blockIdx.x * blockDim.x + threadIdx.x;
 if (yi ≥ ny) return;
 float *simplRow = (float *)((char *) simg + pitchSimg * yi);
 float *tmplRow = (float *)((char *) timg + pitchTimg * yi);
 float d, diff = 0.0, w = 1.0/(nx * ny);
 for (int x = 0; x < nx; ++x) {
 d = simplRow[x] - tmplRow[x];
 diff += d * d * w;
 }
#ifdef __DEVICE_EMULATION__
 fprintf(stderr, "\nprocessing line %d (%d, %d) and %f (%f)\n", yi, blockIdx.x,
 threadIdx.x, diff, w);
#endif
}

```

```
 diffVector[yi] = diff;
}
```

This code is used in chunk 24.

## 9 Initializing and freeing the global images

27  $\langle$  Initializing and freeing the global images 27  $\rangle \equiv$   
 $\langle$  Initialization *Mathematica* template and wrapper 28  $\rangle$   
This code is used in chunk 3.

¶ Mathlink function definition for initialisation of two images. This takes the images from Mathematica and copies them to the graphics card. Note that we are storing a smoothed version of the images by applying a gaussian filter with strength SIGMA

28  $\langle$  Initialization *Mathematica* template and wrapper 28  $\rangle \equiv$

```

:Begin:
:Function:_cudaInitImages
:Pattern:_mlCudaInitImages[simg_List,timg_List,nx_Integer,ny_Integer]
:Arguments:{Flatten[simg],Flatten[timg],nx,ny}
:ArgumentTypes:{Real32List,Real32List,Integer,Integer}
:ReturnType:Manual
:End:

void cudaInitImages(float *simg,int ns,float *timg,int nt,int nx,int ny)
{
 float *d_bm_transposed;
 double q; /* An adapted version of the sigma */
 size_t stride_tr; /* The stride I have to use instead of ny */
 int blocksize, /* How many thread per block */
 gridsize; /* How many blocks in all */
 double sigma = SIGMA;
 cudaError_t err;
 NX = nx;
 NY = ny;
 counter = 0;
 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 */
 \langle Calculate the parameter for the gaussian filter 14 \rangle
 err = cudaMallocPitch((void **) &D_SIMG,&STRIDE_SIMG,nx * sizeof(float),ny);
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 err = cudaMallocPitch((void **) &D_TIMG,&STRIDE_TIMG,nx * sizeof(float),ny);
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 err = cudaMallocPitch((void **) &d_bm_transposed,&stride_tr,ny * sizeof(float),nx);
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 err = cudaMemcpy2D((void *) D_SIMG,STRIDE_SIMG,(void *) simg,
 nx * sizeof(float),nx * sizeof(float),ny,cudaMemcpyHostToDevice);
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 err = cudaMemcpy2D((void *) D_TIMG,STRIDE_TIMG,(void *) timg,
 nx * sizeof(float),nx * sizeof(float),ny,cudaMemcpyHostToDevice);
 \langle check for errors and when they happen return to Mathematica 37 \rangle
 err = cudaMalloc((void **) &D_DIFF,ny * sizeof(float));

```

```

 cudaGaussKernel <<< gridsize, blocksize >>> (D_SIMG, STRIDE_SIMG, nx, ny, b0, b1, b2,
 b3, b4);
 err = cudaGetLastError();
 <check for errors and when they happen return to Mathematica 37>
 cudaTransposeKernel <<< gridsize, blocksize >>> (D_SIMG, STRIDE_SIMG, nx, ny,
 d_bm_transposed, stride_tr);
 err = cudaGetLastError();
 <check for errors and when they happen return to Mathematica 37>
 cudaGaussKernel <<< gridsize, blocksize >>> (d_bm_transposed, stride_tr, ny, nx, b0, b1,
 b2, b3, b4);
 err = cudaGetLastError();
 <check for errors and when they happen return to Mathematica 37>
 cudaTransposeKernel <<< gridsize, blocksize >>> (d_bm_transposed, stride_tr, ny, nx,
 D_SIMG, STRIDE_SIMG);
 err = cudaGetLastError();
 <check for errors and when they happen return to Mathematica 37>
 cudaGaussKernel <<< gridsize, blocksize >>> (D_TIMG, STRIDE_TIMG, nx, ny, b0, b1, b2,
 b3, b4);
 err = cudaGetLastError();
 <check for errors and when they happen return to Mathematica 37>
 cudaTransposeKernel <<< gridsize, blocksize >>> (D_TIMG, STRIDE_TIMG, nx, ny,
 d_bm_transposed, stride_tr);
 err = cudaGetLastError();
 <check for errors and when they happen return to Mathematica 37>
 cudaGaussKernel <<< gridsize, blocksize >>> (d_bm_transposed, stride_tr, ny, nx, b0, b1,
 b2, b3, b4);
 err = cudaGetLastError();
 <check for errors and when they happen return to Mathematica 37>
 cudaTransposeKernel <<< gridsize, blocksize >>> (d_bm_transposed, stride_tr, ny, nx,
 D_TIMG, STRIDE_TIMG);
 err = cudaGetLastError();
 <check for errors and when they happen return to Mathematica 37>
 err = cudaThreadSynchronize();
 <check for errors and when they happen return to Mathematica 37>
 cudaFree(d_bm_transposed);
 MLPutSymbol(stdlink, "Null");
}
:Begin:
:Function:_cudaDeinitializeImages
:Pattern:_mlCudaDeinitializeImages[]
:Arguments:{}
:ArgumentTypes:{}
:ReturnType:Manual
:End:
void cudaDeinitializeImages(void)
{
 if (D_SIMG) {
 cudaFree(D_SIMG);

```



```

 D_SIMG = Λ ;
 }
 if (D_TIMG) {
 cudaFree(D_TIMG);
 D_TIMG = Λ ;
 }
 if (D_DIFF) {
 cudaFree(D_DIFF);
 D_DIFF = Λ ;
 }
 MLPutSymbol(stdlink, "Null");
}
:Begin:
:Function: \square cudaGetImages
:Pattern: \square mlCudaGetImages []
:Arguments: {}
:ArgumentTypes: {}
:ReturnType: Manual
:End:

void cudaGetImages()
{
 cudaError_t err;
 float *simg = (float *) malloc(NX * NY * sizeof(float));
 float *timg = (float *) malloc(NX * NY * sizeof(float));
 err = cudaMemcpy2D((void *) simg, NX * sizeof(float), (void *)
 D_SIMG, STRIDE_SIMG, NX * sizeof(float), NY, cudaMemcpyDeviceToHost);
 < check for errors and when they happen return to Mathematica 37 >
 err = cudaMemcpy2D((void *) timg, NX * sizeof(float), (void *)
 D_TIMG, STRIDE_TIMG, NX * sizeof(float), NY, cudaMemcpyDeviceToHost);
 < check for errors and when they happen return to Mathematica 37 >
 /* TODO: check Mma functions for errors */
 MLPutFunction(stdlink, "List", 2);
 MLPutReal32List(stdlink, simg, NX * NY);
 MLPutReal32List(stdlink, timg, NX * NY);
 if (MLError(stdlink) \neq MLEOK) {
 fprintf(stderr, "%s\n", MLErrorMessage(stdlink));
 MLClearError(stdlink);
 }
 return;
}

```

This code is used in chunk 27.

## 10 Reconstruction filter

29  $\langle$  Reconstruction filter 29  $\rangle \equiv$

```
__device__ float interpolate(float *d_bm, size_t pitch, int nx, int ny, float xt, float yt)
{
 if (xt < 0.0 \vee xt \geq nx - 1.0 \vee yt < 0.0 \vee yt \geq ny - 1.0) {
#ifdef __DEVICE_EMULATION__NOOO
 fprintf(stderr,
 "Position to interpolate lies outside the image: (%lf,%lf) a\
 nd image dim (%d,%d)", xt, yt, nx, ny);
#endif
 return 0.0;
 }
 xt = xt \equiv nx - 1.0 ? xt - TINY : xt;
 yt = yt \equiv ny - 1.0 ? yt - TINY : yt;
 int x1 = floor(xt);
 int y1 = floor(yt);
 int x2 = x1 + 1;
 int y2 = y1 + 1;
 float xx = xt - x1;
 float yy = yt - y1;
 float v11 = *((float *)((char *) d_bm + pitch * y1) + x1);
 float v21 = *((float *)((char *) d_bm + pitch * y1) + x2);
 float v12 = *((float *)((char *) d_bm + pitch * y2) + x1);
 float v22 = *((float *)((char *) d_bm + pitch * y2) + x2);
 return (v11 * (1.0 - xx) + v21 * xx) * (1.0 - yy) + (v12 * (1.0 - xx) + v22 * xx) * yy;
}
```

This code is used in chunk 3.

## 11 Julia Sets

¶ Mathlink function definition for the julia set

```
31 <Julia set 31> ≡
 #define BLOCK_SIZE_2D 16
 <Type for the julia set parameters 32>
 <CUDA-kernel for the julia set 35>
 <Initialize the parameters of the julia set and wrapper for the cuda call 33>
 <Change the parameter for the Julia-Set on the fly 36>
```

This code is used in chunk 3.

¶ The parameters and the bitmap of the julia-set are stored in a struct. I'm using a square of the complex plane for calculation which is centered at  $(centX, centY)$  and has a side length of  $2 \cdot radius$ .

```
32 <Type for the julia set parameters 32> ≡
 struct JuliaSet {
 float centX, centY, radius, delta;
 int imgSize, maxIter;
 int grid, block;
 size_t pitch;
 float rc, ic; /* the complex constant c in the formula $z = z^2 + c$ */
 int *data, *d_data; /* the result matrices */
 };

 inline void initJuliaSet(struct JuliaSet *j, int size, float rrc, float iic, int iter)
 {
 j->centX = 0;
 j->centY = 0;
 j->radius = 0;
 j->imgSize = size;
 j->rc = rrc;
 j->ic = iic;
 j->maxIter = iter;
 j->data = (int *) calloc(size * size, sizeof(int));
 int gs = size % NUM_OF_THREADS ? (size / NUM_OF_THREADS + 1) : (size / NUM_OF_THREADS);
 j->grid = gs;
 j->block = NUM_OF_THREADS;
 }
 ;

 struct JuliaSet JS; __device__ constant__
 struct JuliaSet d_js;
```

This code is used in chunk 31.

¶ I will first have a function which sets the parameter for the iteration formula and allocates the memory on the device. To retrieve an image I then have only to call a second function and give the parameters of the wanted frame of the complex-plane.

```
33 <Initialize the parameters of the julia set and wrapper for the cuda call 33> ≡
 :Begin:
```

```

:Function:_cudaInitJuliaSet
:Pattern:_mlCudaInitJuliaSet[imgSize_Integer,rc_?NumericQ,
_ _ _ _ic_?NumericQ,maxIter_Integer]
:Arguments:{imgSize,N[rc],N[ic],maxIter}
:ArgumentTypes:{Integer,Real32,Real32,Integer}
:ReturnType:Manual
:End: void cudaInitJuliaSet(int imgSize,float rc,float ic,int maxIter)
{
 cudaError_t err;
 initJuliaSet(&JS,imgSize,rc,ic,maxIter);
 err = cudaMallocPitch((void **) &(JS.d_data),&JS.pitch,JS.imgSize*sizeof(int),
 JS.imgSize);
 <check for errors and when they happen return to Mathematica 37>
 MLPutSymbol(stdlink,"Null");
 return;
}
:Begin:
:Function:_cudaDeinitJuliaSet
:Pattern:_mlCudaDeinitJuliaSet[]
:Arguments:{}
:ArgumentTypes:{}
:ReturnType:Manual
:End: void cudaDeinitJuliaSet()
{
 if (JS.data) free(JS.data);
 if (JS.d_data) cudaFree(JS.d_data);
 MLPutSymbol(stdlink,"Null");
}

```

See also chunk 34.

This code is used in chunk 31.

¶ This gets the bitmap of the initialize julia-set in the square which is defined by the parameters *centx*, *centy* and *radius*.

34 <Initialize the parameters of the julia set and wrapper for the cuda call 33> +=

```

:Begin:
:Function:_cudaGetJuliaSet
:Pattern:_mlCudaGetJuliaSet[centx_?NumericQ,centy_?NumericQ,radius_?NumericQ]
:Arguments:_{N[centx],N[centy],N[radius]}
:ArgumentTypes:_{Real32,Real32,Real32}_
:ReturnType:_Manual_
:End:

```

```

void cudaGetJuliaSet(float cx,float cy,float r)
{
 cudaError_t err;
 JS.centX = cx;
 JS.centY = cy;
 JS.radius = r;
 JS.delta = 2 * r / (JS.imgSize - 1); /* the distance between every pixel */
 err = cudaMemcpyToSymbol("d_js", &JS, sizeof(struct JuliaSet), 0,
 cudaMemcpyHostToDevice);
 <check for errors and when they happen return to Mathematica 37>
 dim3 blockSize(BLOCK_SIZE_2D, BLOCK_SIZE_2D, 1);
 int gs = (JS.imgSize % BLOCK_SIZE_2D) ? JS.imgSize / BLOCK_SIZE_2D + 1 :
 JS.imgSize / BLOCK_SIZE_2D;
 dim3 gridSize(gs, gs, 1);
#ifdef __DEVICE_EMULATION__
 fprintf(stderr, "changed: \tgrid(%d,%d,%d) \tblock(%d,%d,%d) \n", gridSize.x,
 gridSize.y, gridSize.z, blockSize.x, blockSize.y, blockSize.z);
#endif
 juliaSetKernelWithSharedMem << gridSize, blockSize >>> ();
 err = cudaGetLastError();
 <check for errors and when they happen return to Mathematica 37>
 err = cudaThreadSynchronize();
 <check for errors and when they happen return to Mathematica 37>
 err = cudaMemcpy2D((void *) JS.data, JS.imgSize * sizeof(int), (void *)
 JS.d_data, JS.pitch, JS.imgSize * sizeof(int), JS.imgSize, cudaMemcpyDeviceToHost);
 <check for errors and when they happen return to Mathematica 37>
 MLPutIntegerList(stdlink, JS.data, JS.imgSize * JS.imgSize);
 if (MLError(stdlink) ≠ MLEOK) {
 fprintf(stderr, "%s\n", MLErrorMessage(stdlink));
 MLClearError(stdlink);
 }
}

```

¶ Two implementations of the julia-set function are given. One which uses the cuda implementation of the arithmetic operations and one which doesn't. The CUDA-functions are supposed to be faster but sometimes less precise. Check appendix of the CUDA programming guide for detailed information.

```

35 <CUDA-kernel for the julia set 35> ≡
 __device__ int juliaIteration(float rz,float iz)
 {
 int iter = 0;
 float tmpRz, tmpIz;
 while (iter ≤ d_js.maxIter) {
 float rzSq = __fmul_rn(rz, rz), izSq = __fmul_rn(iz, iz);
 if (__fadd_ru(rzSq, izSq) ≥ 4) return iter;
 tmpRz = __fadd_rn(__fadd_rn(rzSq, -izSq), d_js.rc);
 tmpIz = __fadd_rn(d_js.ic, __fmul_rn(__fmul_rn(2.0, iz), rz));
 rz = tmpRz;

```

```

 iz = tmpIz;
 iter++;
 }
 return iter;
}
__device__ int juliaIterationSlow(float rz, float iz)
{
 int iter = 0;
 float tmpRz, tmpIz;
 while (iter ≤ d_js.maxIter) {
 float rzSq = rz * rz, izSq = iz * iz;
 if (rzSq + izSq ≥ 4) return iter;
 tmpRz = rzSq - izSq + d_js.rc;
 tmpIz = d_js.ic + 2.0 * iz * rz;
 rz = tmpRz;
 iz = tmpIz;
 iter++;
 }
 return iter;
}
__global__ void juliaSetKernelWithSharedMem() { int tx = threadIdx.x,
 ty = threadIdx.y, bx = blockIdx.x, by = blockIdx.y;
 int px = bx * blockDim.x + tx, py = by * blockDim.y + ty;
#ifdef __DEVICE_EMULATION__
 fprintf(stderr, "thread(%d,%d) □ block(%d,%d) □ pos(%d,%d) □->□", tx, ty,
 bx, by, px, py);
#endif

 __shared__
 int res[BLOCK_SIZE_2D][BLOCK_SIZE_2D]; if
 (px < d_js.imgSize ∧ py < d_js.imgSize) { float
 d = d_js.delta, x1 = d_js.centX - d_js.radius,
 y1 = d_js.centY - d_js.radius;
 res[ty][tx] = juliaIteration(x1 + d * px, y1 + d * py); #
 ifdef __DEVICE_EMULATION__
 fprintf(stderr, "(%f,%f,\\t%d)\\n", x1 + d * px, y1 + d * py, res[ty][tx]);
 #endif

 __syncthreads();
 *((int *)((char *) d_js.d_data + py * d_js.pitch) + px) = res[ty][tx]; } }

```

This code is used in chunk 31.

¶ Changing the parameter c. This changes the parameter c for an already initialized Julia-Set. This maybe good for demonstration purposes when you can change the whole look. I just set the parameter in the global JS struct which is copied anyway to the device when *cudaGetJuliaSet* is called.

36 < Change the parameter for the Julia-Set on the fly 36 > ≡

```

:Begin:
:Function: □setJuliaSetParameter
:Pattern: □□mlCudaSetJuliaSetParameter[rc_?NumericQ,ic_?NumericQ]

```

```

:Arguments: {N[rc],N[ic]}
:ArgumentTypes: {Real32,Real32}
:ReturnType: Manual
:End:
void setJuliaSetParameter(float rc,float ic)
{
 JS.rc = rc;
 JS.ic = ic;
 MLPutFunction(stdlink,"List",2);
 MLPutReal32(stdlink,rc);
 MLPutReal32(stdlink,ic);
 return;
}

```

This code is used in chunk 31.

¶ Checking for CUDA-errors printing them to *stderr* and returning them to *Mathematica*. We do not exit the binary or return from the function call. Returning might be a bigger problem than writing the error and going on while maybe returning a wrong result. The alternative would be to return go *Mathematica* without cleaning up used memory.

Note that the return value of the CUDA-call has to be stored in the variable *err*.

37 <check for errors and when they happen return to *Mathematica* 37> ≡

```

if (err != cudaSuccess) {
 fprintf(stderr,"An error occurred.\n");
 const char *errStr = cudaGetErrorString(err);
 fprintf(stderr,"%s\n",errStr);
 MLPutFunction(stdlink,"List",2);
 MLPutSymbol(stdlink,"$Failed");
 MLPutString(stdlink,errStr);
}

```

This code is used in chunks 7, 8, 10, 13, 18, 25, 28, 33, and 34.

## 12 Appendix

### References

*NVIDIA CUDA – Reference Manual Version 2.3.*

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



## List of Refinements

- ⟨  $L^2$  norm CUDA-kernel 26 ⟩ Used in chunk 24.
- ⟨  $L^2$  norm *Mathematica* template and wrapper 25 ⟩ Used in chunk 24.
- ⟨ Calculate row with a general affine transformation 23 ⟩ Used in chunk 19.
- ⟨ Calculate row with a pure rotation 20 ⟩ Used in chunk 19.
- ⟨ Calculate row with a rotation and translation 21 ⟩ Used in chunk 19.
- ⟨ Calculate row with a rotation, scaling and translation 22 ⟩ Used in chunk 19.
- ⟨ Calculate the parameter for the gaussian filter 14 ⟩ Used in chunks 12, 13, and 28.
- ⟨ Change the parameter for the Julia-Set on the fly 36 ⟩ Used in chunk 31.
- ⟨ Deinitialize the runtime 10 ⟩ Used in chunk 5.
- ⟨ Distance measure for images 24 ⟩ Used in chunk 3.
- ⟨ Functions for initialization and informations about the CUDA devices 5 ⟩ Used in chunk 3.
- ⟨ Gaussian filter 11 ⟩ Used in chunk 3.
- ⟨ Get information about the running cuda 7, 8 ⟩ Used in chunk 5.
- ⟨ Headers and main 4 ⟩ Used in chunk 3.
- ⟨ Init CUDA 6 ⟩ Used in chunk 5.
- ⟨ Initialization *Mathematica* template and wrapper 28 ⟩ Used in chunk 27.
- ⟨ Initialize the parameters of the julia set and wrapper for the cuda call 33, 34 ⟩ Used in chunk 31.
- ⟨ Initializing and freeing the global images 27 ⟩ Used in chunk 3.
- ⟨ Julia set 31 ⟩ Used in chunk 3.
- ⟨ Reconstruction filter 29 ⟩ Used in chunk 3.
- ⟨ Rigid transformation CUDA-kernel 19 ⟩ Used in chunk 17.
- ⟨ Rigid transformation *Mathematica* template and wrapper 18 ⟩ Used in chunk 17.
- ⟨ Rigid transformation for images 17 ⟩ Used in chunk 3.
- ⟨ Type for the julia set parameters 32 ⟩ Used in chunk 31.
- ⟨ CUDA-kernel for the julia set 35 ⟩ Used in chunk 31.
- ⟨ CUDA-kernels for the gaussian filter 15, 16 ⟩ Used in chunk 11.
- ⟨ *Mathematica* template and wrapper for the gaussian filter 12, 13 ⟩ Used in chunk 11.
- ⟨ check for errors and when they happen return to *Mathematica* 37 ⟩ Used in chunks 7, 8, 10, 13, 18, 25, 28, 33, and 34.
- ⟨ send CUDA device-information to *Mathematica* 9 ⟩ Used in chunks 7 and 8.