CUDA

G. Lochmann

Institute of Astronomy,
University of Vienna,
Vienna, Austria.

Practical in numerical astronomy
Vienna

April 2010

Why do we want to use the graphic chip? I

- Optimized for floating point operations per second (FLOPS) for the video game industry
- Computation in parallel, called single instruction multiple data architecture
- Comparison graphic chips and motherboard (single data single instruction)

	Floating point operations per second
NVIDIA Geforce 8800M GTX	360 GFlops
NVIDIA Geforce 8600M GT	43,2 GFlops
Pentium 4,3 GHz	6 GFlops

Why do we want to use the graphic chip? II

- Higher memory bandwidth

- Comparison graphic chips and motherboard

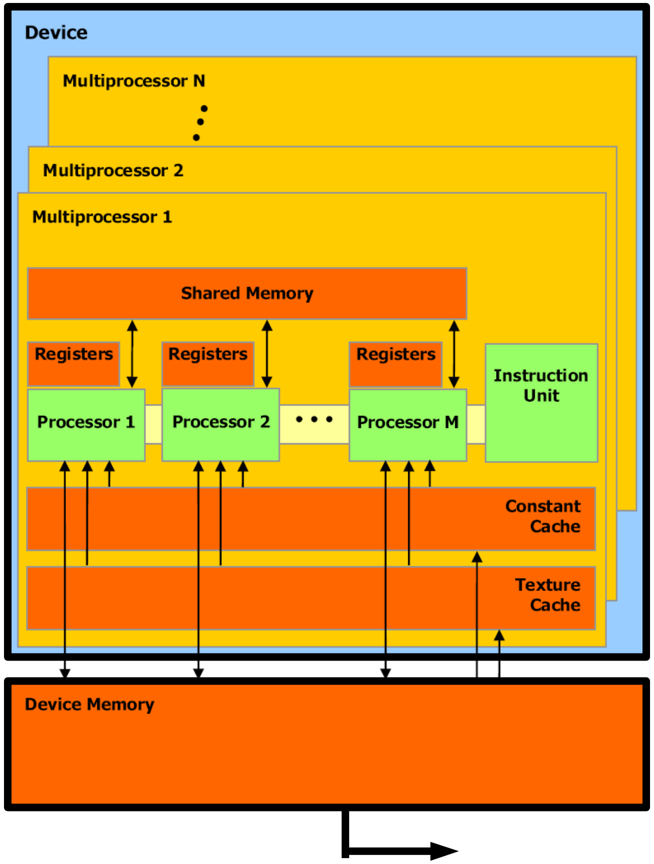
	Memory bandwidth
NVIDIA Geforce 8800M GTX	51,2 GB/s
NVIDIA Geforce 8600M GT	19,2 GB/s
Pentium 4,3 GHz	5,9 GB/s

Why do we want to use the graphic chip? III

- Distribute the working flow to both the motherboard and the graphic chip
- Restricted for complicated evaluations over a small set of initial conditions
- Basic methods derived from the methematic:
  A problem is parallelizable if it can be described as matrix multiplication

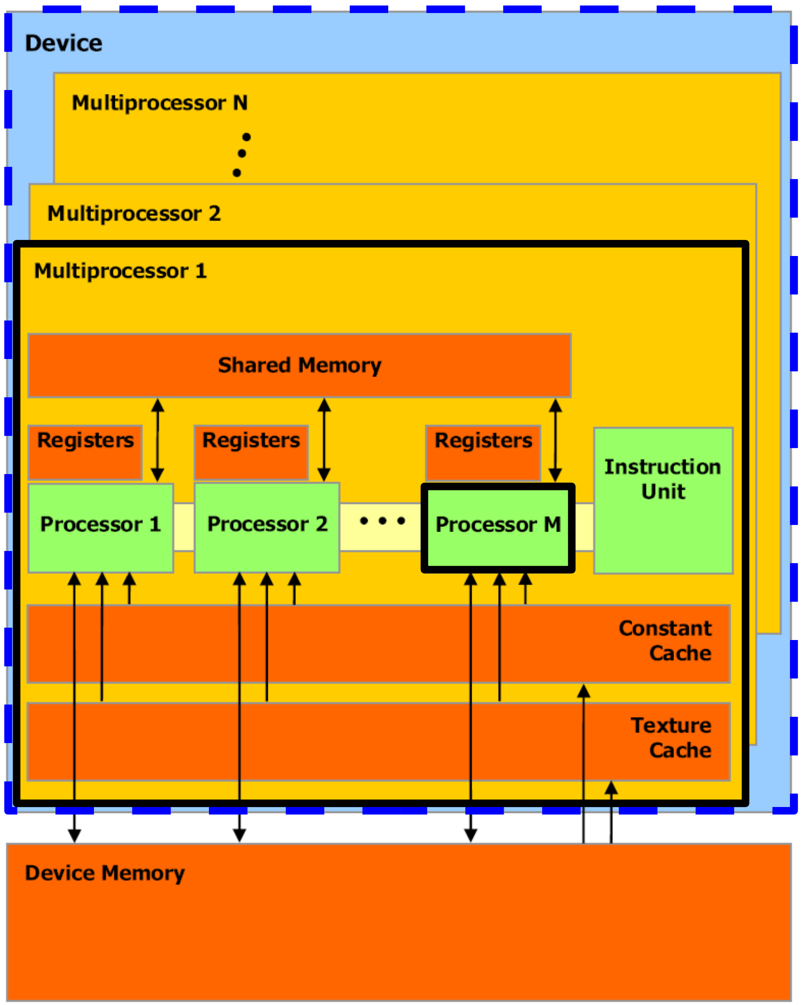
Architecture of the GPU I

- Basically the GPU consists of:



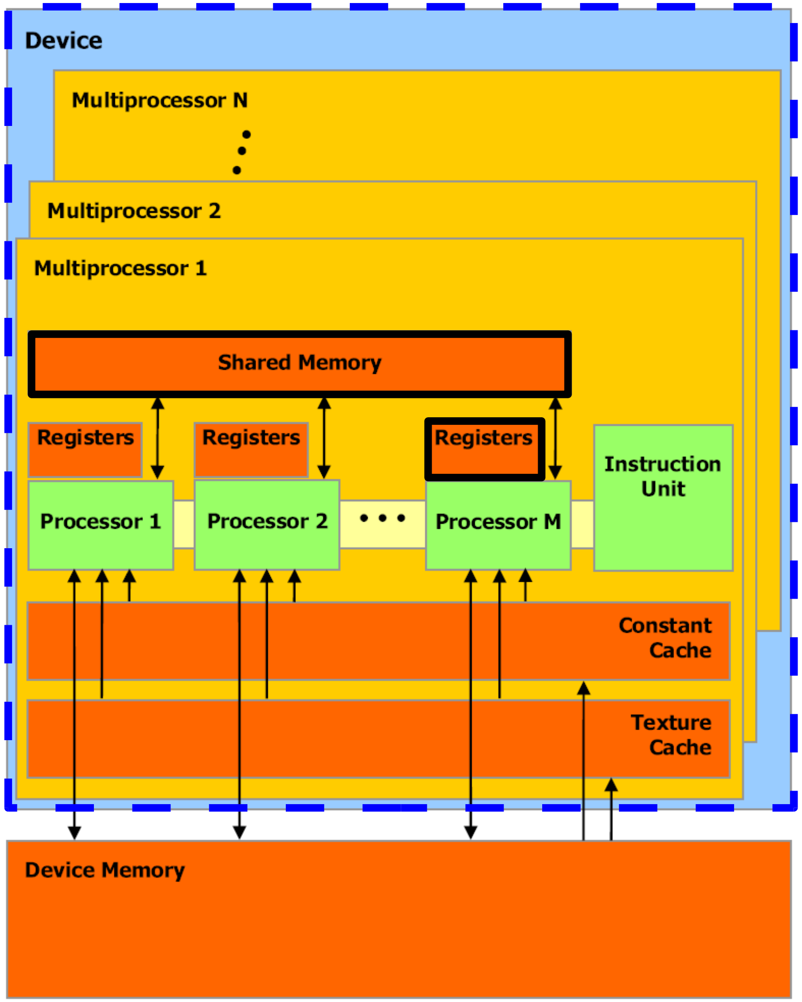
- The processor
- DRAM memory, called global or device memory
- Linkage to the motherboard through a PCI express BUS

Architecture of the GPU II – The processor I



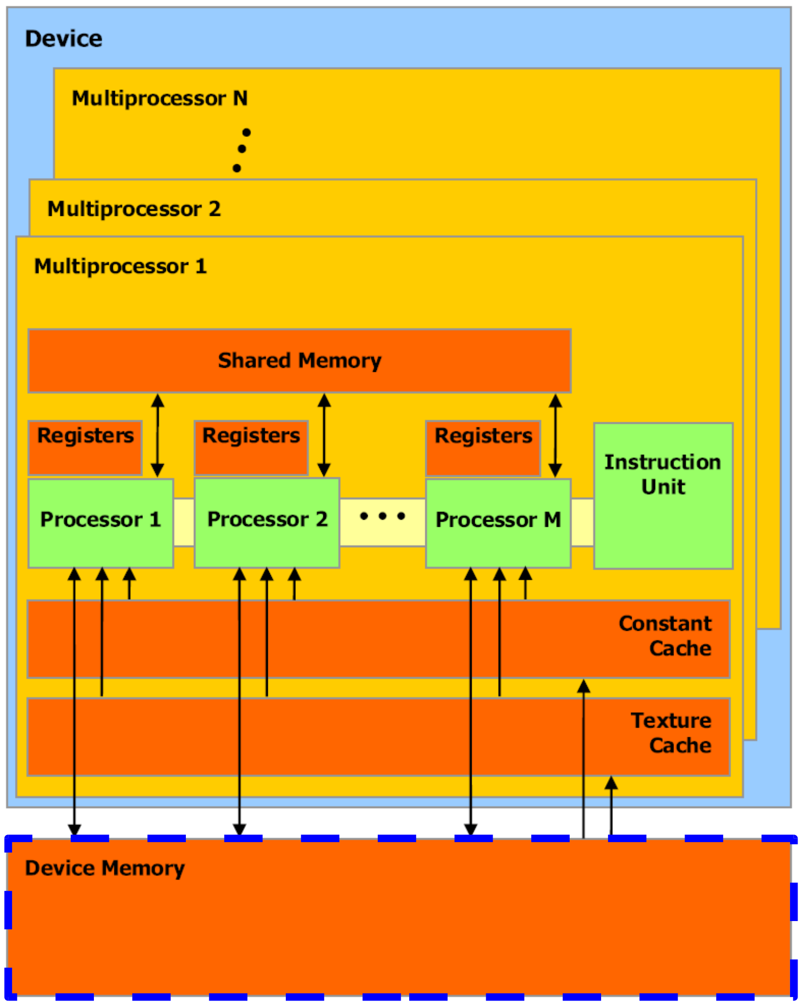
- The processor contains up to 16 streaming multiprocessors
- Every streaming multiprocessor contains 8 streaming processors
- Each streaming processor evaluates 4 executions pro step
- Computes up to 512 values in the same time

Architecture of the GPU II – The processor II



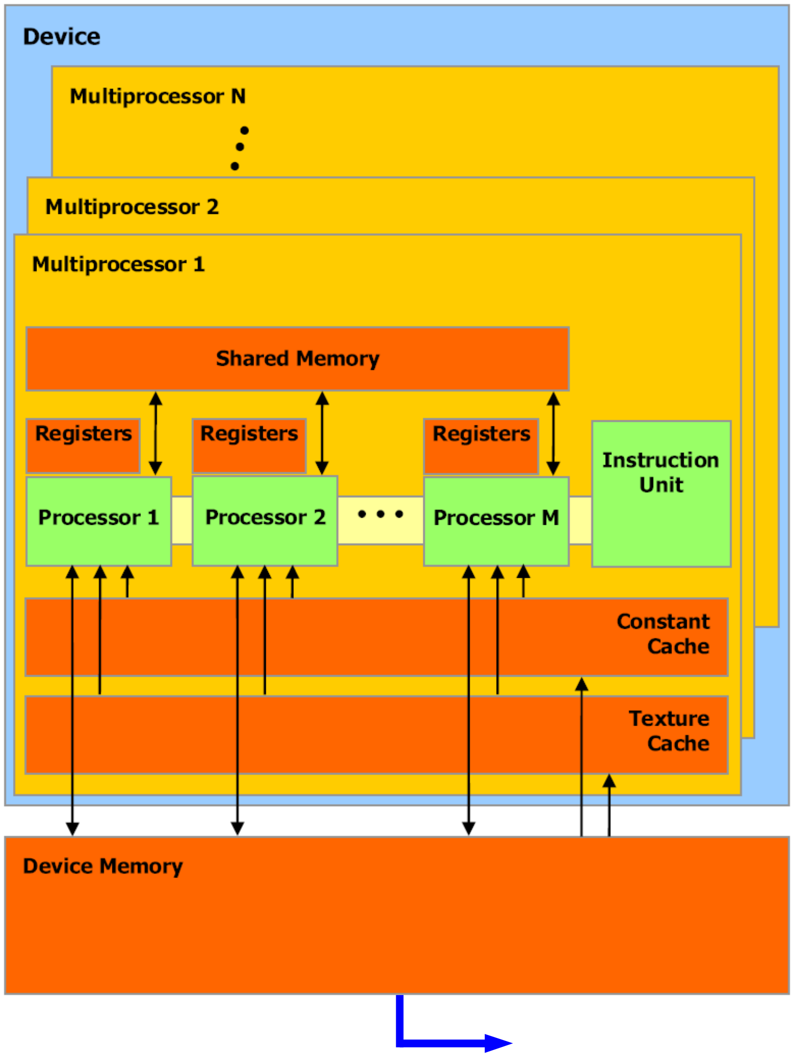
- On-chip memory consists of:
  - Register space:
    Up to 5000 float-values
  - Shared memory:
    Up to 4000 float-values

Architecture of the GPU III – The memory I



- DRAM off-chip memory, called global or device memory
- Used by:
  - Every streaming multiprocessor
  - The CPU to copy forth and back
- 1.5 GByte, circa 0.4 Mrd. float-values
- Needs much more cycles to read or write in contrast to the on-chip memory

Architecture of the GPU IV – The link I



- PCI express BUS with a memory bandwidth of 4 GB per second
- Bottleneck of the computation

Recapitulate

- Two devices, the motherboard and the graphic chip, that communicate through a „small BUS“

- Two different processors:
  - The CPU working sequentially
  - The GPU working parallel

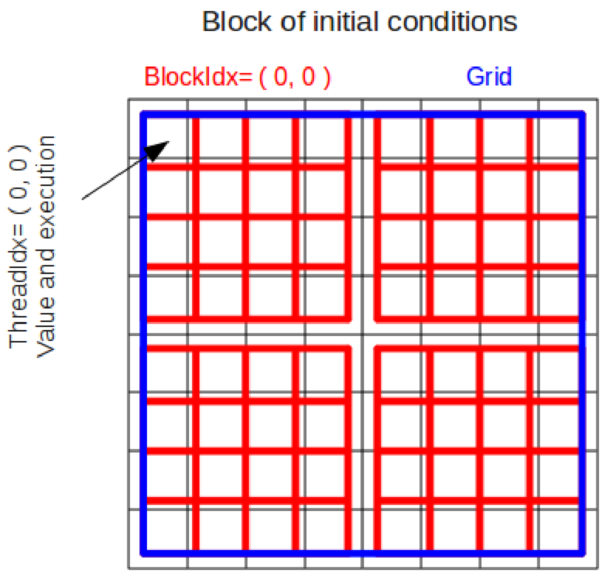
- Two distinguished memories on which the processors work

Structure seen through CUDA I

- General:
  - The motherboard is called host
  - The graphic chip is called device
  - A whole computation is called kernel
- With the kernel call the architecture of the computation has to be set through the parameters grid and block

Structure seen through CUDA II

- The two parameters:

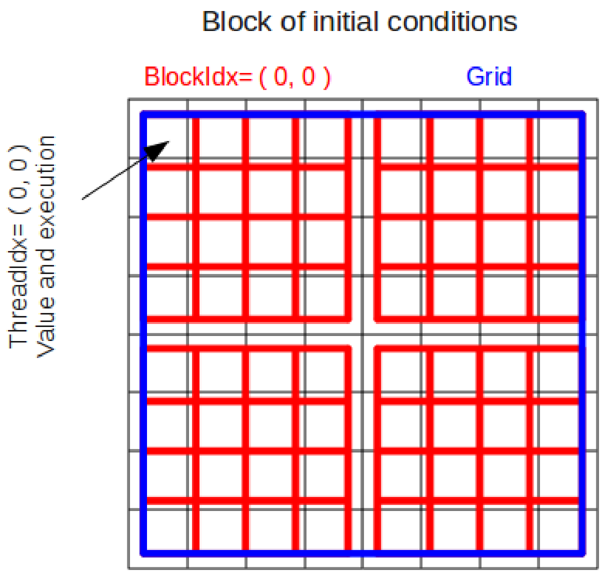


- Grid:
  - Whole executions over all streaming multiprocessors
  - Related to the whole set of initial conditions
- Block:
  - Executions of one streaming multiprocessor
  - Related to the shared memory

- The grid size tells the GPU how many blocks it has to distribute over the streaming multiprocessors
- The block size tells the GPU how many computations it has to perform on one streaming multiprocessor

Structure seen through CUDA III

- The Thread:

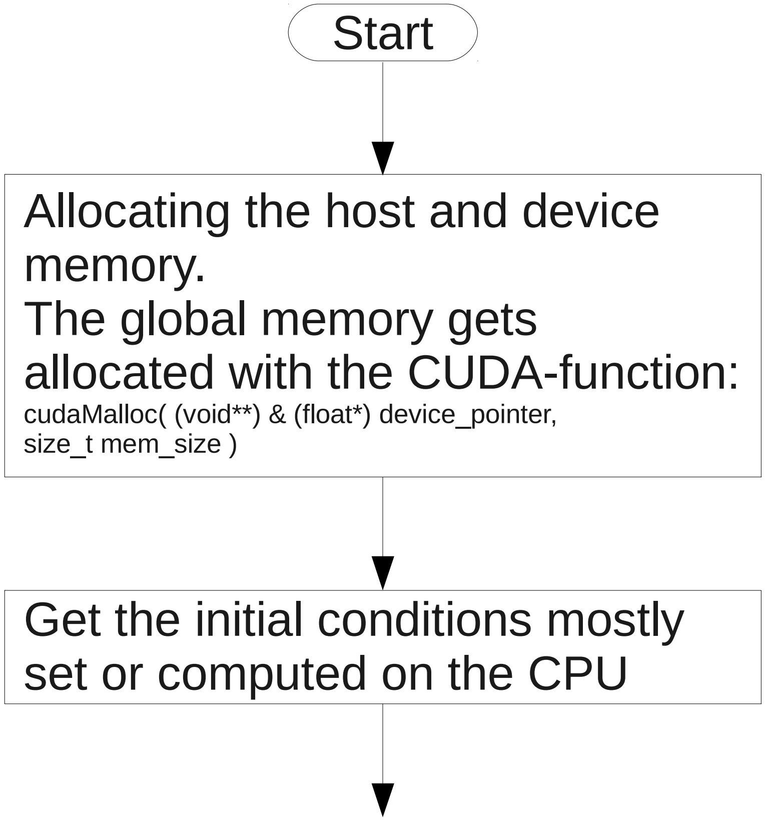


- Corresponds to one execution
- Closest relation to the data, can symbolize one value
- Related to the registers too

- The Warp:
  - Stands for all threads that get executed simultaneously on one streaming multiprocessor
  - Warp size is an important value for time issues

Fluxdiagram I

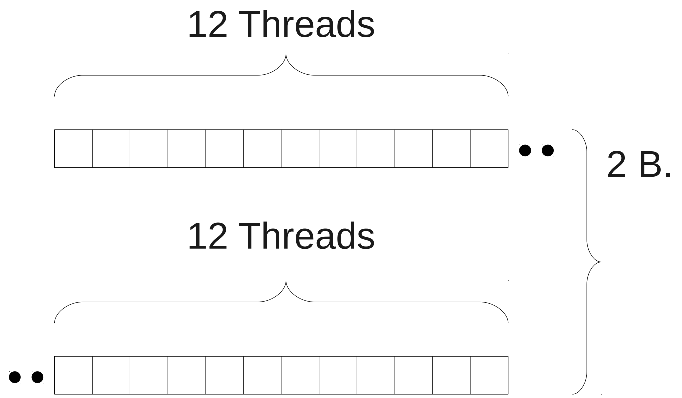
- General fluxdiagram:



Fluxdiagram II

Copy the initial conditions (referred to through the pointer host_pointer) from the RAM to the global memory. The used CUDA-function is:

```
cudaMemcpy( (float*) device_pointer, (float*) host_pointer,
size_t mem_size, cudaMemcpyHostToDevice )
```



Set the kernel architecture with 2 dim3 variables, common use grid, block

Start the kernel with the call:

```
kernel_name<<<dim3 grid, dim3 block>>>(
(float*) device_pointer(s), everything else )
```

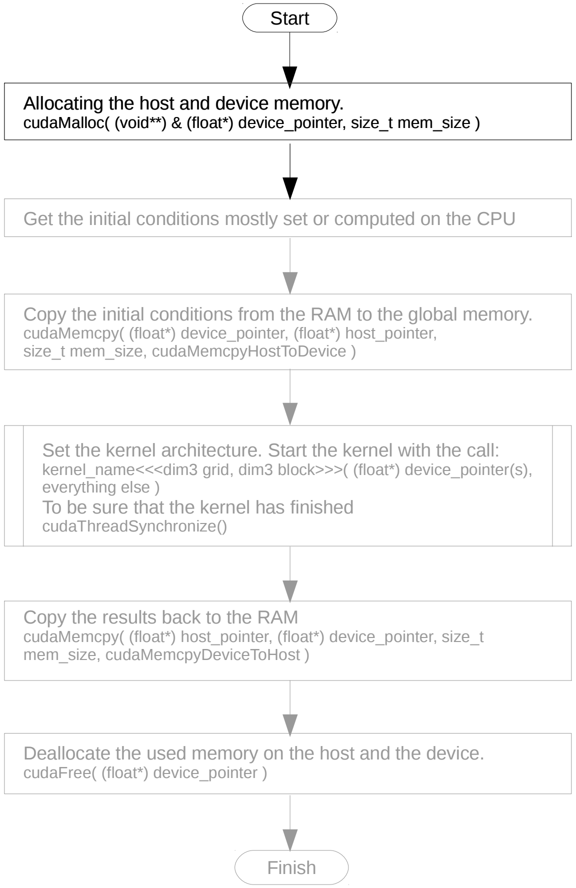
To be sure that the kernel has finished before copying the results back use the CUDA-function

```
cudaThreadSynchronize()
```

Fluxdiagram III

Sample program: matrix multiplication I

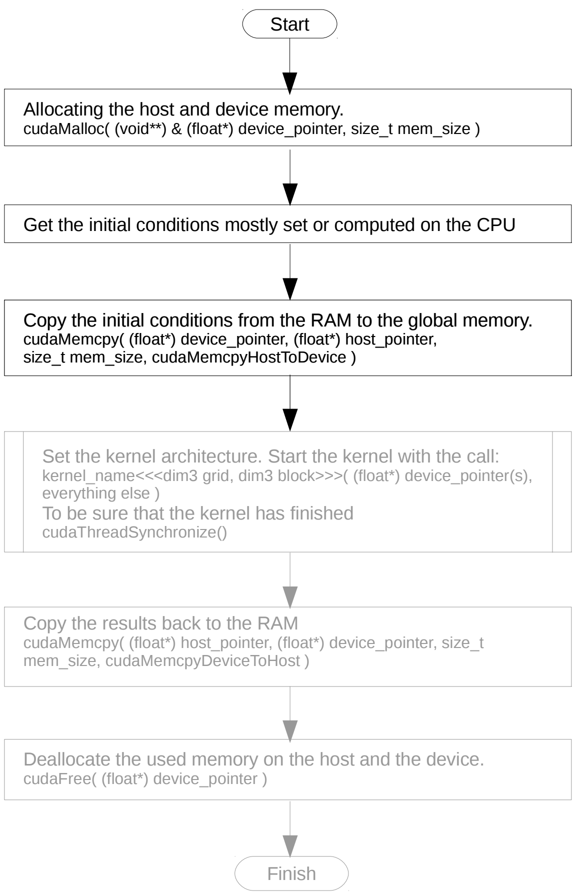
- Begin of the sample CUDA-program




```
    int width, height;

    size_t mem_size;


// Declaration of the pointers to the memory on the RAM (A, B, C)
// and on the global memory (d_A, d_B, d_C)

    float *A, *d_A;
    float *B, *d_B;
    float *C, *d_C;


// dim3 bases on the vectortype with 3 unsigned int arguments, new
// with CUDA. Describes the architecture of the kernelexecution

    dim3 block, grid;
```

```
    mem_size= sizeof(float)*width*height;


// Allocating the memory on the RAM (linear alligned memory)

    A= (float*) malloc(mem_size);
    B= (float*) malloc(mem_size);
    C= (float*) malloc(mem_size);


// Allocating the memory on the global memory (device)
// with the function cudaMalloc (linear alligned memory)

    cudaMalloc((void**) &d_A, mem_size);
    cudaMalloc((void**) &d_B, mem_size);
    cudaMalloc((void**) &d_C, mem_size);
```

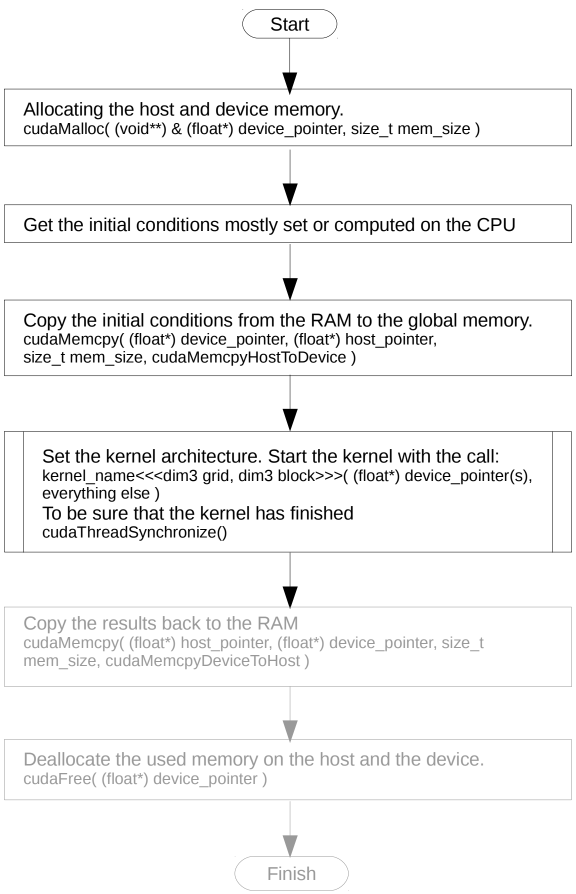
Sample program: matrix multiplication II

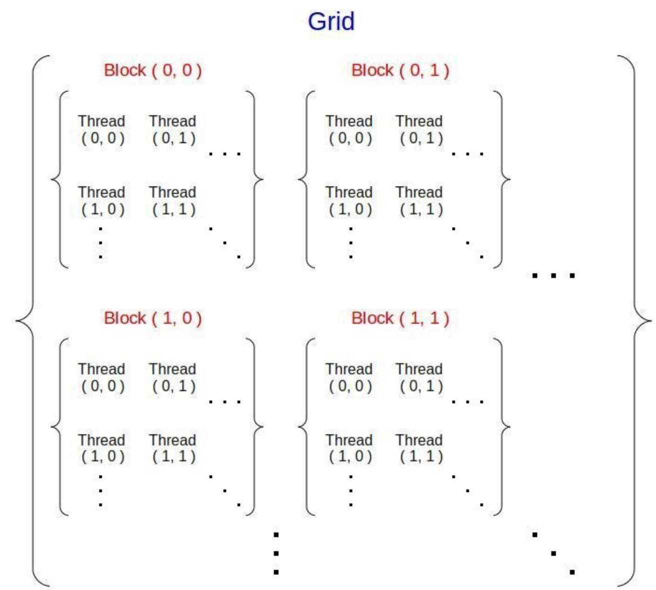



```
// Initialisation of the given matrices A and B that will get
// multiplied. The resulting matrices are 2 identity matrices

    for( i = 0 ; i < height ; i++ )
    {

        for( j = 0 ; j < width ; j++ )
        {

            if( j==i )
            {

                A[j+i*width]= 1;
                B[j+i*width]= 1;

            }
            else
            {

                A[j+i*width]= 0;
                B[j+i*width]= 0;

            }

        }

    }
```

```
// Copy the given matrices to the reserved space on
// the global memory. Using the function cudaMemcpy with
// the option cudaMemcpyHostToDevice

    cudaMemcpy( d_A, A, mem_size, cudaMemcpyHostToDevice );
    cudaMemcpy( d_B, B, mem_size, cudaMemcpyHostToDevice );
```

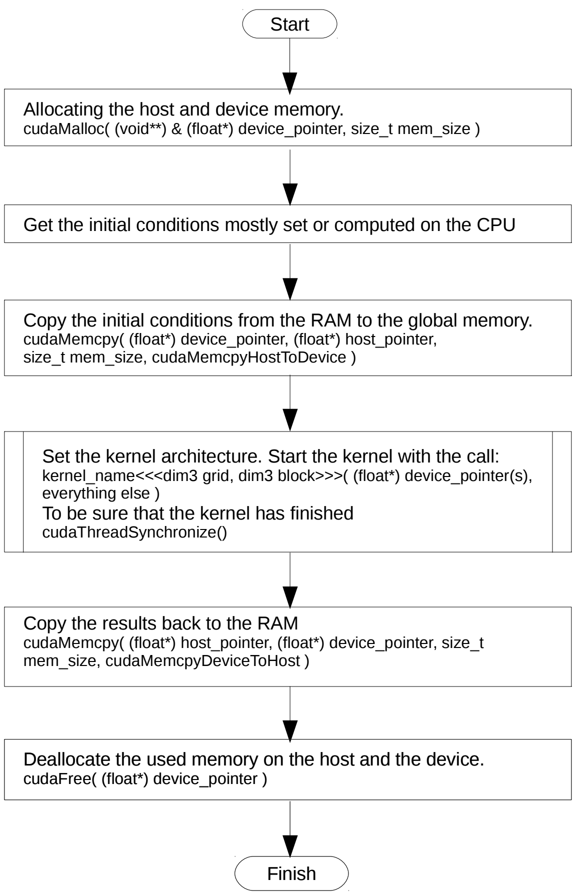
Sample program: matrix multiplication III






```
// Call for the kernelexecution, using the architecture
// that was set before.
// The kernel needs the pointers wich point to the memory
// on the gpu and the width, for one loop in the execution

    matrixmul_kernel<<<grid,block>>>( d_A, d_B, d_C, width );


    cudaThreadSynchronize();
```

Sample program: matrix multiplication III




```
// Copy the result back to the RAM. Again using
// the function cudaMemcpy but with the option
// cudaMemcpyDeviceToHost

    cudaMemcpy( C, d_C, mem_size, cudaMemcpyDeviceToHost );

    cudaThreadSynchronize();
```

```
// Deallocate the used memory; on the RAM with free
// on the global memory with cudaFree

    free( A ); free( B ); free( C );
    cudaFree( d_A ); cudaFree( d_B ); cudaFree( d_C );
```

Sample program: matrix multiplication IV

- Matrix multiplication in C:

```
// Standard matrixmultiplication on a cpu

void matrixmul_c( float *A, float *B, float *C,
int width, int height )
{

    int i, j, k;
    float sum;

    for( i = 0 ; i < height ; i++ )
    {

        for( j = 0 ; j < width ; j++ )
        {

            sum= 0.0;

            for( k = 0 ; k < width ; k++ )
            {
                sum= sum+A[k+i*width]*B[k*width+j];
            }

            C[i*width+j]= sum;

        }
    }
```

- Needs 3 loops:
  - i-loop for the rows
  - j-loop for the columns
  - k-loop for every value in the corresponding rows and columns

Sample program: matrix multiplication V

CUDA
G.Lochmann

- Matrix multiplication in CUDA:

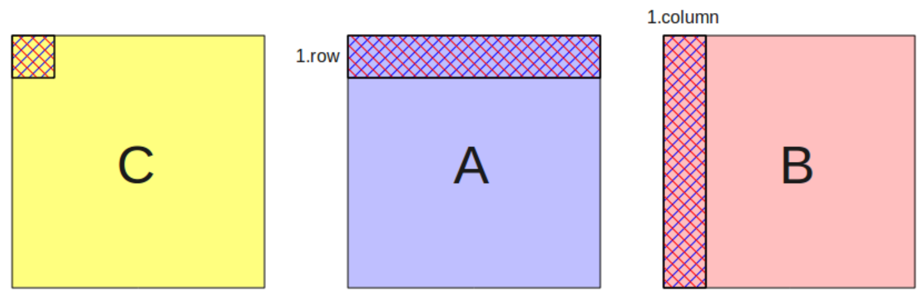
```
// Matrixmultiplication on the gpu

__global__ void matrixmul_kernel( float *d_A, float *d_B,
float *d_C, int width )
{

    int i;
    float sum;

    int col= threadIdx.x+blockIdx.x*blockDim.x;

    int row= threadIdx.y+blockIdx.y*blockDim.y;

    sum= 0.0;

// Each execution takes one row and one column and multiplies
// the corresponding values, walked through with this loop

    for( i = 0 ; i < width ; i++ )
    {
        sum= sum+d_A[i+row*width]*d_B[i*width+col];
    }

    d_C[row*width+col]= sum;

}
```

- Needs 1 loops for the multiplications in the corresponding rows and columns
- Each thread gets this instruction and the parameters:
  - ThreadIdx
  - BlockIdx
  - BlockDim

Sample program: matrix multiplication VI

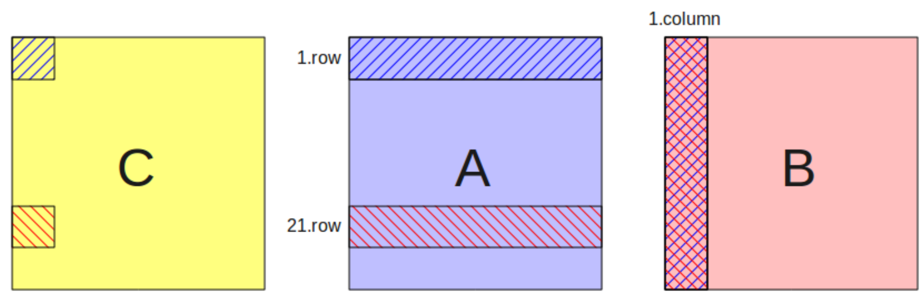
WRONG

Block (0, 0) : Thread (0, 0) => row = 0 ; col = 0 => 1st row * 1st column
Block (1, 0) : Thread (0, 0) => row = 0 ; col = 0 => 1st row * 1st column



RIGHT

Block (0, 0) : Thread (0, 0) => row = 0 ; col = 0 => 1st row * 1st column
Block (1, 0) : Thread (0, 0) => row = 20 ; col = 0 => 21st row * 1st column



Sample program: matrix multiplication VII

- Execution for the thread (10, 1) within the block (0, 1)

```
int i;
float sum;

int col= 10+0*20    //threadIdx.x+blockIdx.x*blockDim.x;

int row= 1+1*20     //threadIdx.y+blockIdx.y*blockDim.y;

sum= 0.0;

for( i = 0 ; i < width ; i++ )
{
    sum= sum+d_A[i+21*width]*d_B[i*width+10];
}

d_C[21*width+10]= sum;
```