Practical Course: GPU Programming in Computer Vision

CUDA Basics

Björn Häfner, Robert Maier, David Schubert

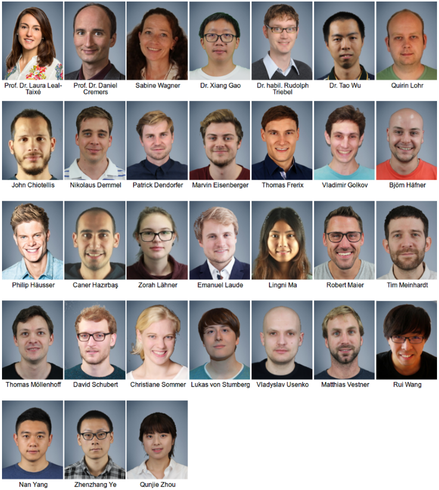
Technische Universität München
Department of Informatics
Computer Vision Group

Summer Semester 2018
September 17 - October 15

Outline

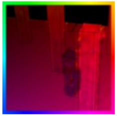
Outline

Computer Vision Group



Our Research Interests

Image-based 3D Reconstruction



Optical Flow Estimation

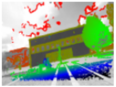
Shape Analysis

Robot Vision

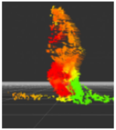
RGB-D Vision

Image Segmentation

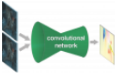
Convex Relaxation Methods



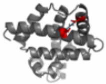
Visual SLAM



Scene Flow Estimation



Deep Learning



Biomedicine

Organizational Setup

What is this course about?

- Parallel Programming using CUDA
- Computer Vision Basics
- Work on a cool final project

What will you learn?

- How to program parallel processors
- Acquire the technical knowledge to understand how CUDA works
- Apply this knowledge efficiently to implement computer vision algorithms and gain a massive speedup

Organizational Setup

Time line:

- Lecture (September 17 - 21)
  - 2–3h lectures **!!!attendance is mandatory!!!**
  - Followed by programming exercises until open end
- Project (September 24 - October 12)
  - Implement an advanced application assigned to your group
  - Group of three students
- Demo day (October 15)
  - Prepare a presentation and demo
  - Showing off what your group achieved throughout the project phase

Organizational Setup

Lecture:

- Starts at 10 a.m. sharp!
- Don't forget: **!!!attendance is mandatory!!!**
- First part of lecture corresponds to CUDA
- Short break of 15 min
- Second part of lecture corresponds to mathematics/computer vision

Organizational Setup

Exercises:

- Starts after the second part of the lecture
- Will be supervised until 4 p.m.
- Stay as long as you want to solve the assignments
- Each day a new exercise sheet based on corresponding CUDA and math/cv lecture
- Grade bonus of 0.3 – 0.4:
  - Deadline: **Sunday 11.59 p.m.**
  - Hand in solution for all exercises
  - Each student has to hand in separately and code must be individual, i.e. copied code will not be graded and thus fail
  - Grade bonus achieved, if 80% or more are correct
  - Achieved grade bonus will be announced during project phase

Organizational Setup

Project Phase:

- Implement a computer vision algorithm in CUDA
- Form groups of three students per group, i.e. eight groups in total
- Pick one of the projects we suggest on Friday or
- Suggest your own project
- Let us know your group and your three preferred projects by **Friday 11.59 p.m.**
- Meet your advisor regularly
- If we detect cheating, everyone involved gets the grade 5.0

Organizational Setup

Demo day:

- Prepare a presentation of 15–20 minutes per group
- Explain the assigned problem/project
- How did you proceed to solve it
- Each group member presents and describes his/her task in the project
- Show your results

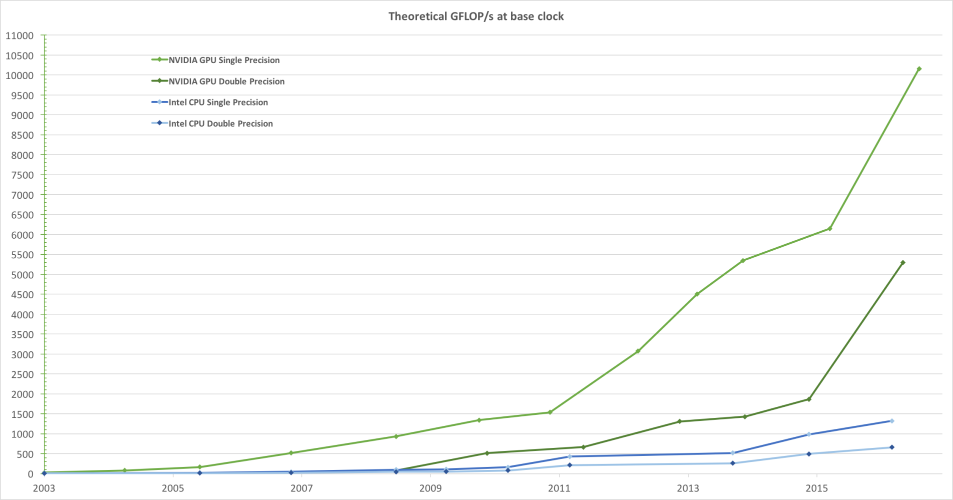
Organizational Setup

Work from home during project phase:

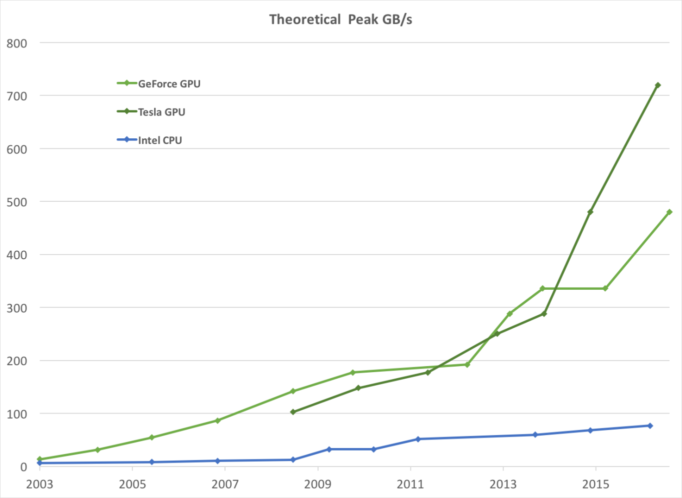
- Access your computer in the lab from home:
  `ssh -p 58022 a123@hostname.informatik.tu-muenchen.de`
- Replace `a123` with your login handed out by us
- Replace `hostname` with your computer name
  - type `hostname` in terminal to find out your computer name

Outline

Why using GPUs?



Why using GPUs?



GPU is available in every PC $\Longrightarrow$ Massive volume and impact!

Design Difference

CPU vs. GPU

- Different goals produce different designs
  - CPU must be good at everything, parallel or not
  - GPU assumes work load is highly parallel
- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic
- GPU: maximize throughput of all threads
  - skip big caches, multi-threading hides latency
  - share control logic across many threads: Single instruction, multiple data (SIMD)
  - create and run thousands of threads

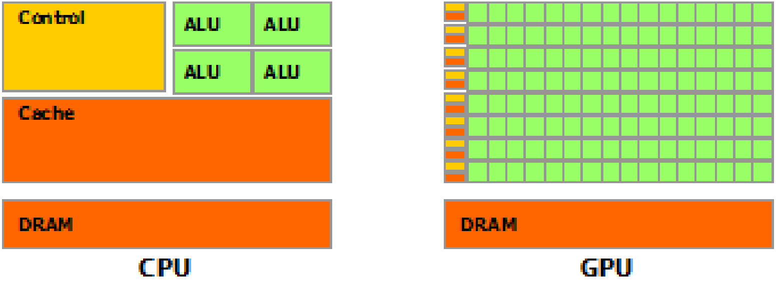
$\Longrightarrow$ Assumption: The problem is data parallel, i.e. same operations can be performed independently on many separate data elements.

Many computer vision problems fulfill this assumption.

Design Difference

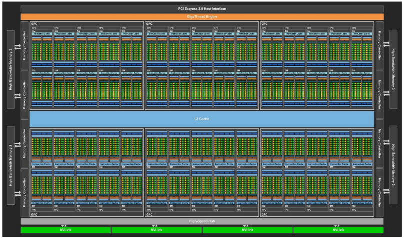
CPU vs. GPU

- Different goals produce different designs
  - CPU: Minimize latency using big cache and large control logic
  - GPU: Maximize throughput using SIMD and thousands of threads



GPU in Detail

Current Architecture



(a) Full GPU with 60 Streaming Multiprocessors (SMs)

(b) One SM; Each SM has 64 CUDA Cores

Figure: Pascal Architecture with $60 \cdot 64 = 3840$ cores

Pascal Architecture in the lab: 2×6 SMs with 64 CUDA cores each.

Entering CUDA

"Compute Unified Device Architecture"

- Scalable parallel programming model
  - is suitably efficient and practical when applied to large amount of data
  - thus exposes the computational horsepower of GPUs
- Abstractions for parallel computing
  - let programmers focus on parallel algorithms
  - not mechanics of a parallel programming language
- Minimal extensions to familiar C/C++ environment to run code on the GPU
  - Easy to learn
  - but hard to master

CUDA

Scalable Parallel Programming

- Provide straightforward mapping onto hardware
  - good fit to GPU architecture
  - thus programmer can focus on parallel algorithms
- Execute code by many threads in parallel
- Scale to 100s of cores and 10000s of threads
  - GPU threads are lightweight – create/switch is free
  - GPU needs 1000s of threads for full utilization

References

Good to know and almost mandatory to check it out

- CUDA has an excellent documentation:
  - `CUDA Toolkit Documentation v8.0`
  - `CUDA Programming Guide`
    - Provides detailed discussion of CUDA. Describes hardware implementation, provides guidance how to achieve maximum performance and much more in-depth explanations
  - `CUDA Runtime API`
    - List of all CUDA functions
  - `https://developer.nvidia.com/gpu-accelerated-libraries`
    - List of “official” (third party) libraries using of CUDA
  - `cd /usr/local/cuda-9.1/samples/1_Utilities/deviceQuery/`
    - Run `deviceQuery` sample to quickly see your hardware specifications

Outline of the course I

1. Basics (Monday; David)
   - Kernels and Thread Hierarchy
   - Execution on the GPU
   - Memory Management
   - Error Handling And Compiling
2. Memories (Tuesday; Robert)
   - Overview of Memory Spaces
   - Shared Memory
   - Texture Memory
   - Constant Memory
   - Common Strategy for Memory Accesses

Outline of the course II

1. Optimization (Wednesday; Robert)
   - Branch Divergence
   - Pitch Allocation for 2D Images
   - Host-Device Memory Transfer
   - Occupancy
   - Parallel reduction
2. Misc (Thursday; Björn)
   - Atomics
   - CUDA Streams and Events
   - Multi-GPU Programming
   - Third party libraries
3. Development Tools (Friday; Björn)
   - CMake
   - Nsight
   - `CUDA-MEMCHECK`

Outline

Example: CPU vs. GPU

- CPU - Processes subtasks serially one by one

```
for (int i = 0; i<n; i++)
{
  c[i] = a[i] + b[i];
}
```

- GPU - Processes each subtask in parallel

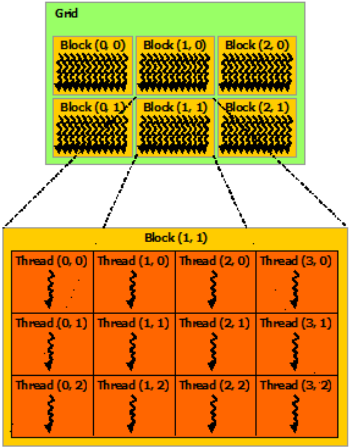
```
__global__ void g_vecAdd (float * a, float *b, float *c)
{
  int i = threadIdx.x + blockDim.x*blockIdx.x;
  c[i] = a[i] + b[i];
}
```

Thread Hierarchy

- Threads are grouped into blocks
  - Up to 512 or 1024 threads per block
  - Thread indices are unique within a block
- Note: Threads from the same block can cooperate
  - synchronize their execution
  - communicate via shared memory
  - threads from different blocks cannot cooperate
- All blocks together form a grid
  - Block indices are unique within a grid

Thread Hierarchy

- Blocks and grids can be 1D, 2D or 3D
- Dimensions of grids and blocks are set at launch
- Block dimensions can be different for each grid
- Built-in variables to access dimensions and indices:
  - `gridDim`, `blockDim`
  - `blockIdx`, `threadIdx`



Index Calculation

- Aim: mapping between threads and array elements
- 1D

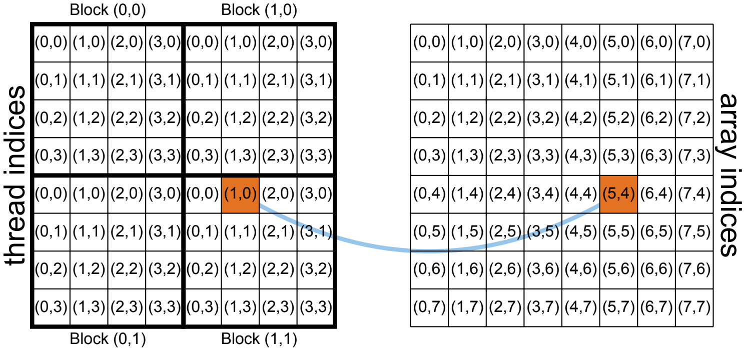
	Block 0								Block 1							
Thread Index	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
Array Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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

- Example: 11 = 3 + 8 * 1

Index Calculation

- 2D




```
int x = threadIdx.x + blockDim.x * blockIdx.x;
int y = threadIdx.y + blockDim.y * blockIdx.y;
```

- Example: 5 = 1 + 4 * 1   4 = 0 + 4 * 1

Index Calculation

- Use built-in variables to access unique indices

```
index = thread_in_block + threads_per_block * block_index;
```

- 1D

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

- 2D

```
int x = threadIdx.x + blockDim.x * blockIdx.x;
int y = threadIdx.y + blockDim.y * blockIdx.y;
```

- 3D

```
int x = threadIdx.x + blockDim.x * blockIdx.x;
int y = threadIdx.y + blockDim.y * blockIdx.y;
int z = threadIdx.z + blockDim.z * blockIdx.z;
```

Kernel Launch

- Usual C/C++ function call, with an additional specification of `grid` and `block` sizes:

```
myKernel <<< grid, block >>>( ... );
```

- `dim3 grid; dim3 block;`
  - access each dimension, e.g. in the variable `block`:
    `block.x; block.y; block.z;`
- CUDA kernels are launched from the CPU or GPU
- CUDA kernels are always executed on the GPU

Example: One-dimensional Kernel

```
__global__ void myKernel (int *a, int n)
{
  int ind = threadIdx.x + blockDim.x * blockIdx.x;
  if (ind<n) a[ind] += 1;
}

int main()
{
  dim3 block = dim3(128,1,1); // 128*1*1 threads per block
  // ensure enough blocks to cover n elements (round up)
  dim3 grid = dim3( (n + block.x -1) / block.x, 1, 1);
  myKernel <<<grid, block>>> (d_a, n);

  // Also possible:
  // launch 4 blocks, each with 128 threads per block
  myKernel <<<4,128>>> (d_a, n);
}
```

Example: Two-dimensional Kernel

```
__global__ void myKernel (int *a, int w, int h)
{
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;
  int ind = x + w*y; //derive linear index
  if (x<w && y<h) a[ind] += 1;
}

int main()
{
  dim3 block = dim3(32,8,1); // 32*8*1 = 256 threads per block

  // ensure enough blocks to cover w * h elements (round up)
  dim3 grid = dim3( (w + block.x -1) / block.x,
  (h + block.y - 1) / block.y, 1 );

  myKernel <<<grid,block>>> (d_A, w, h);
}
```

Why this `if`-statement?

- There may be more threads than array elements
  $\Longrightarrow$ Always test whether the indices are within bounds

```
__global__ void myKernel (int *a, int n)
{
  int ind = threadIdx.x + blockDim.x * blockIdx.x;
  if (ind<n) a[ind] += 1;
}

__global__ void myKernel (int *a, int w, int h)
{
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int y = threadIdx.y + blockDim.y * blockIdx.y;
  int ind = x + w*y; //derive linear index
  if (x<w && y<h) a[ind] += 1;
}
```

Exercise: IDs of Threads and Blocks

```
kernel<<<4,4>>>(d_a);
```

Exercise: IDs of Threads and Blocks

```
kernel<<<4,4>>>(d_a);
```

```
__global__ void kernel (int *a)
{
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  a[idx] = 7;
}
```

Exercise: IDs of Threads and Blocks

```
kernel<<<4,4>>>(d_a);
```

```
__global__ void kernel (int *a)
{
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  a[idx] = 7;
}
//Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
```

Exercise: IDs of Threads and Blocks

```
kernel<<<4,4>>>(d_a);
```

```
__global__ void kernel (int *a)
{
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  a[idx] = blockIdx.x;
}
```

Exercise: IDs of Threads and Blocks

```
kernel<<<4,4>>>(d_a);
```

```
__global__ void kernel (int *a)
{
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  a[idx] = blockIdx.x;
}

//Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
```

Exercise: IDs of Threads and Blocks

```
kernel<<<4,4>>>(d_a);
```

```
__global__ void kernel (int *a)
{
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  a[idx] = threadIdx.x;
}
```

Exercise: IDs of Threads and Blocks

```
kernel<<<4,4>>>(d_a);
```

```
__global__ void kernel (int *a)
{
  int idx = threadIdx.x + blockDim.x * blockIdx.x;
  a[idx] = threadIdx.x;
}
//Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
```

Code Executed on GPU

GPU Function Type Qualifiers

Terminology: **CPU** is called **host**!
GPU is called **device**!

- `__global__`: kernels
  - launched by CPU to run on the GPU must return `void`
- `__device__`: auxiliary GPU functions
  - launched by `__global__` or `__device__` functions to run on the GPU
- `__host__`: “normal” CPU C/C++ functions
  - launched by CPU to run on the CPU
- `__host__ __device__`: qualifiers can be combined
  - callable from CPU and from GPU

Code Executed on GPU

Crucial Restrictions

- On CPU: only access CPU memory
- On GPU: only access GPU memory
  - GPU can access CPU memory:
    - Page-Locked Host Memory (special allocation of host memory)
    - from CUDA 6: Unified Memory (managed memory space with coherent memory of device and host)
  - no access to host functions
  - no static variables in functions or classes
    - static variable for functions possible: `__device__ volatile` keyword
  - from CUDA 7: variadic templates variable number of arguments

Code Executed on GPU

Features

- Many C/C++ features available for GPU code
  - templates
  - recursion (CC $>=$ 2.0)
  - overloading
    - function overloading
    - operator overloading
  - classes
    - stack allocation
    - heap allocation (CC $>=$ 2.0)
    - inheritance, virtual functions (CC $>=$ 2.0)
  - function pointers (CC $>=$ 2.0)
  - `printf()` formatted output (CC $>=$ 2.0)
- Vector variants of basic types
  - `float2`, `float3`, `float4`, `double2`, `int4`, `char2`, etc.
  - `float2 a = make_float2(1,2); a.x = 10; a.y = a.x;`

Blocks

Must Be Independent

- Any possible ordering of blocks should be valid
  - Can run in any order (order is unspecified)
  - Can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
- Independence requirement gives scalability

Execution of Kernels

Asynchronous

- Kernel launches are asynchronous w.r.t. CPU
  - after kernel launch, immediately control returns
  - CPU is free to do other work while the GPU is busy
- Kernel launches are queued
  - kernel does not start until previous kernels are finished
  - concurrent kernels possible for CUDA >= 7.0: Streams (given enough resources)
- Explicit synchronization, if needed
  - Use `cudaDeviceSynchronize()`

Outline

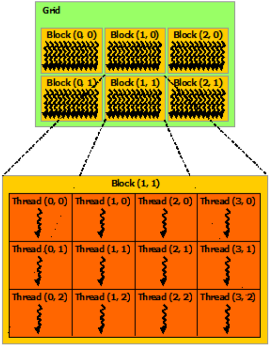
NVIDIA GPU Architecture

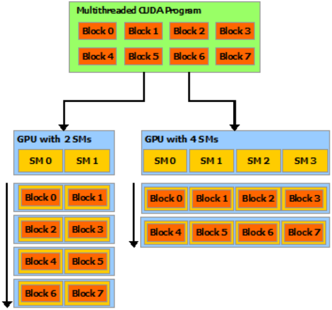
- Each GPU can have up to 10 (Tesla), 16 (Fermi), 15 (Kepler), 24 (Maxwell) or 60 (Pascal) independent Streaming Multiprocessors (SMs)
- No shared resources across SMs, except global memory
- No synchronization, always work in parallel
- Each SM can have 24 (Tesla), 32 (Fermi), 192 (Kepler), 128 (Maxwell) or 64 (Pascal) CUDA cores.
- In total a GPU can have 240 (Tesla), 512 (Fermi), 2880 (Kepler), 3072 (Maxwell) or 3840 (Pascal) cores

Execution of Kernels on the GPU

- Blocks are distributed across SMs
- Active blocks
  - are currently executed
  - reside on a multiprocessor
  - resources allocated
  - executed until finished
- Waiting blocks
  - wait to be executed
  - not yet assigned to a SM

Illustration of Architecture





Blocks Execute on Multiprocessors

- Each block is executed on one Multiprocessor (SM)
  - cannot migrate
  - reason for block independence
- Several blocks per SM possible
  - if enough resources available
  - SM resources are divided among all blocks
- Block threads share SM resources
  - SM registers are divided up among the threads
  - SM shared memory can be read/written by all threads

Warps

Key Architectural Idea

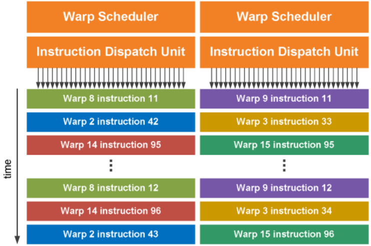
- SIMT (Single Instruction Multiple Thread) execution
  - threads run in groups of 32 called warps
- All 32 threads in a warp execute the same instruction
  - always, no matter what (even if threads diverge)
- Threads are executed warp-wise by the GPU
  - for each warp, the 32 threads are executed in parallel
  - warps are executed one after another
  - but several warps can run simultaneously

Warps in Multiprocessors

- Resources are allocated for all potential warps
  - the state of every potentially executable warp is always present on the Multiprocessor, until finished
  - overall many more potentially executable threads than CUDA Cores possible
- Switching between warps is free and any non-waiting warp can run
- At each clock cycle each warp scheduler chooses a single warp which is ready to be executed
- For each chosen warp the next instruction is executed for all 32 threads of the warp

Example

- Assume there are six blocks on one (out of four) SM(s). Each block has 128 threads
  - Threads from all blocks are divided into warps: 6(blocks)*128(threads/block)/32=24 warps, i.e. 4 warps from every block
  - Having two warp schedulers, two (out of 24) warps can be executed in parallel



Outline

GPU Memory

- CPU and GPU have separate memory spaces
  - data is moved across PCIe bus
  - use functions to allocate/set/copy memory on GPU
    - `cudaMalloc`, `cudaMemset`, `cudaFree`
- Pointers are just addresses
  - cannot tell from pointer if memory is on GPUs or CPU
    - but possible using unified virtual addressing
  - dereference with caution:
    - crash if GPU dereferences pointer to CPU memory and vice versa

Allocate and Release GPU Memory

- Host (CPU) manages device (GPU) memory:
  - `cudaMalloc(void **pointer, size_t nbytes)`
  - `cudaMemset(void *pointer, int value, size_t count)`
  - `cudaFree(void* pointer)`

```
int n = 1024;
size_t nbytes = (size_t)(n)*sizeof(int);
int *d_a = NULL;

cudaMalloc(&d_a, nbytes); //allocate memory on device
cudaMemset(d_a, 0, nbytes); //fill array with 0 valued !ints!
cudaFree(d_a); //free memory on device again
```

Copy Data between CPU and GPU

- `cudaMemcpy (void *dst, void *src, size_t nbytes, cudaMemcpyKind direction);`
  - blocks the CPU thread until all bytes have been copied
  - non-blocking variants are also available
  - doesn't start copying until all previous CUDA calls complete
- `cudaMemcpyKind`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

```
cudaMemcpy( dev_ptr,
            host_ptr,
            (size_t)(n)*sizeof(float),
            cudaMemcpyHostToDevice);
```

Example Host Code

```
// allocate and initialize host (CPU) memory
float *h_a = ..., *h_b = ...; *h_c = ...; (empty)

// allocate device (GPU) memory
float *d_a, *d_b, *d_c;
cudaMalloc( &d_a, n * sizeof(float) );
cudaMalloc( &d_b, n * sizeof(float) );
cudaMalloc( &d_c, n * sizeof(float) );

// copy host memory to device
cudaMemcpy( d_a, h_a, n * sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( d_b, h_b, n * sizeof(float), cudaMemcpyHostToDevice );

// launch kernel
dim3 block = dim3(128,1,1);
dim3 grid = dim3((n + block.x -1) / block.x, 1, 1);
vecAdd <<<grid,block>>> (d_a, d_b, d_c);

// copy result back to host (CPU) memory
cudaMemcpy( h_c, d_c, n * sizeof(float), cudaMemcpyDeviceToHost );

// do something with the result...

// free device (GPU) memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

Use `float` by Default!!!

- GPUs can handle double
- But `float` operations are still much faster
  - by an order of magnitude
  - so use `double` only if `float` is really not enough
- Avoid using `double`, unless necessary
  - Add 'f' suffix to `float` literals:
    - `0.f`, `1.0f`, `3.1415f` are of type `float`
    - `0.0`, `1.0`, `3.1415` are of type `double`
  - Use `float` version of math functions:
    - `expf` / `logf` / `sinf` / `sqrtf` / etc. take and return `float`
    - `exp` / `log` / `sin` / `sqrt` / etc. take and return `double`

Blocks Size

How to choose

- Number of threads per block should be multiple of 32
  - because threads are always executed in groups of 32 (buzzword: warps)
- Rules of thumb:
  - not too small or too big: between 128 and 256 threads
  - start with `dim3(32,8,1)`, i.e. 256 threads per block
  - experiment with similar sized "multiple-of-32"-blocks:
    - `dim3(64,4,1)`, `dim3(128,2,1)`, `dim3(32,4,1)`, `dim3(64,2,1)`
    - `dim3(32,16,1)`, `dim3(64,8,1)`, `dim3(128,4,1)`, `dim3(256,2,1)`
  - measure the run time and choose the best block size!

Outline

Error Handling

- Checking for errors is crucial for programming GPUs
- `cudaError_t cudaGetLastError()`
  - returns the code for the last error
  - resets the error flag back to `cudaSuccess`
  - `cudaPeekAtLastError()`: get error code without resetting it
  - if everything OK: `cudaSuccess`
- `char* cudaGetErrorString(cudaError_t code)`
  - returns a C-string describing the error

```
cudaMalloc(&d_a, n*sizeof(float));
cudaError_t e = cudaGetLastError();
if (e!=cudaSuccess)
{
  cerr << "ERROR: " << cudaGetErrorString(e) << endl;
  exit(1);
}
```

Error Handling

- Kernel execution is asynchronous
  - first force to wait for the kernel to finish by `cudaDeviceSynchronize()`
  - only then call `cudaGetLastError()`
    - otherwise it will be called too soon, the error may not have yet occurred
  - kernel launch itself may produce errors due to invalid configurations
    - too many threads/block, too many blocks, too much shared memory requested
- Kernels may produce subtle memory corruption errors
  - may get unnoticed even after `cudaDeviceSynchronize()`
  - subsequent CUDA calls may or may not fail because of such an error
  - if they do fail, they were not the origin of the error
- It helps to keep track of the previous $\{1, 2, ..., 10\}$ CUDA calls

Compiling

- CUDA files have ending `.cu`: `squareArray.cu`
- NVidia CUDA Compiler: `nvcc`
  - handles the CUDA part
  - hands over pure C/C++ part to host compiler

    `nvcc -o squareArray squareArray.cu`
- Additional info about the kernels using `--ptxas-options=-v`:

```
nvcc -o squareArray squareArray.cu --ptxas-options=-v
ptxas info: Compiling entry function '_Z18cuda_square_kernelPfi' for 'sm_10'
ptxas info: Used 2 registers, 28 bytes smem
```

Outline

Summary

Cheat Sheat

- Thread Hierarchy:
  - `thread`- smallest executable unit
  - `warp` - group of 32 threads
  - `block` - group of threads, shared memory for collaboration
  - `grid` - consists of several blocks
- Keyword extensions for C/C++:
  - `__global__` - kernel-function called by CPU, executed on GPU
  - `__device__` - function called by GPU and executed on GPU
  - `__host__` - [optional]-function called and executed by CPU
  - `<<<...>>>` - kernel launch, chevrons specify grid and block sizes
- Compilation:
  - `nvcc -o <executable> <filename>.cu`