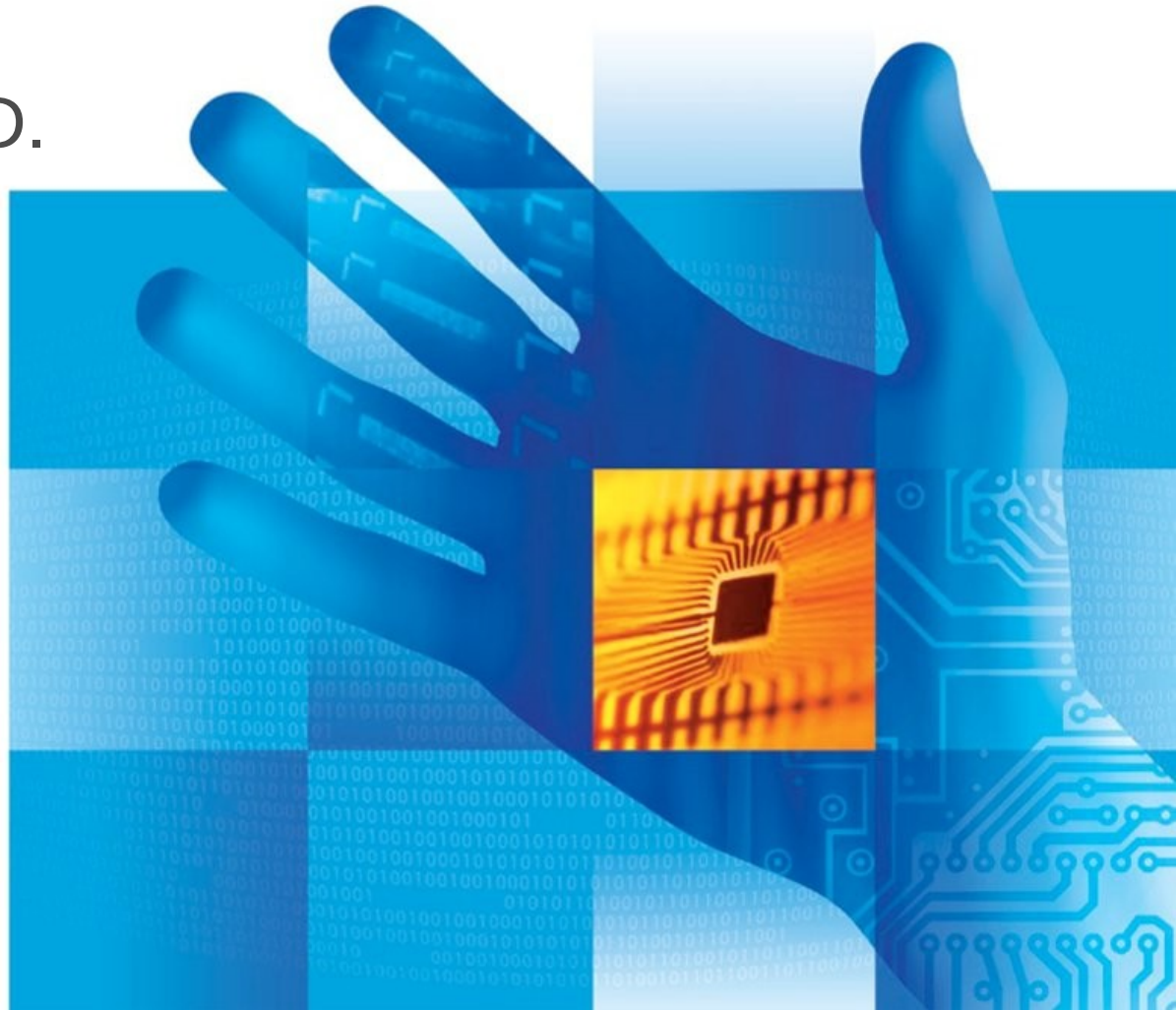
Architecture of Computers and Parallel Systems
Part 10: GP-GPU Computing, CUDA

Ing. Petr Olivka, Ph.D.

petr.olivka@vsb.cz

Department of Computer Science

FEECS VSB-TUO



(GP)GPU – Motivation (for APPS?)

The General Purpose Graphics Processing Unit is one of the most modern parallel system with the high computing performance. But there are more reasons why to become familiar with this technology:

- Massive parallelism (division of algorithms into elementary steps and its programming).
- Correct data manipulation (transfer between GPU and CPU).
- Digital image handling (processing) and graphical data representation in a computer.
- Representation of colors (RGB, BGR, RGBA, B&W)
- Representation of digital images data – matrices, linear sequence of bytes in memory.
- Introduction of Open Source Computer Vision Library (OpenCV).
- Last but not least: Object programming.

(GP)GPU Computing History

The GPU stands for Graphical Processing Unit and it allows users to use it for General Purpose calculations, thus abbreviation GP-GPU. Let's have a look at a brief overview of the GPU computing history. Because our department is a partner of NVIDIA, we will mainly focus on that manufacturer.

- 1970 – ANTIC in an 8-bit Atari computer.
- 1980 – IBM 8514.
- 1993 – Nvidia Co founded.
- 1994 – 3dfx Interactive founded.
- 1995 – first chip NV1 introduced by Nvidia.
- 1996 - 3dfx released Voodoo Graphics.
- 1999 - GeForce 256 from Nvidia offered geometrical transformations support.

(GP)GPU Computing History

- 2000 - Nvidia acquires 3dfx Interactive.
- 2002 - GeForce 4 equipped with pixel and vertex shaders.
- 2006 - GeForce 8 – unified computing architecture (not distinguishing pixels and vertex sharers) – Nvidia CUDA.
- 2008 - GeForce 280 – supports computing in double FPU precision
- 2010 - GeForce 480 (Fermi) – first GPU designed directly for general purpose GPU computing.

Department's labs for this study subject are equipped with Nvidia graphics cards with the Pascal architecture. Therefore it is possible to use latest technology under Windows and Linux operating systems.

This presentation is focused on the Fermi / Kepler / Maxwell / Pascal / Volta architecture and CUDA.

(GP)GPU Programming History

- The first GPU computing was the indirect computing over the OpenGL programming interface.
- Programs and problems were formulated as calculations using points and textures.
- Game developers needed more universal hardware, therefore pixel shaders were developed. It is a simple graphical processor to work with points. It is usually able to compute FPU numbers with single precision. Code is limited to a few instructions. Let's imagine the following simple example:
  We have two textures A and B with dimension 100x100 points. We put them on the screen on the same position with the given alpha channel. Thus the result on the screen is:
  $Screen_{ij} = \alpha_A.A_{ij} + \alpha_B.B_{ij}$
  The result of this simple example is a matrix addition and matrix multiplication by a constant.

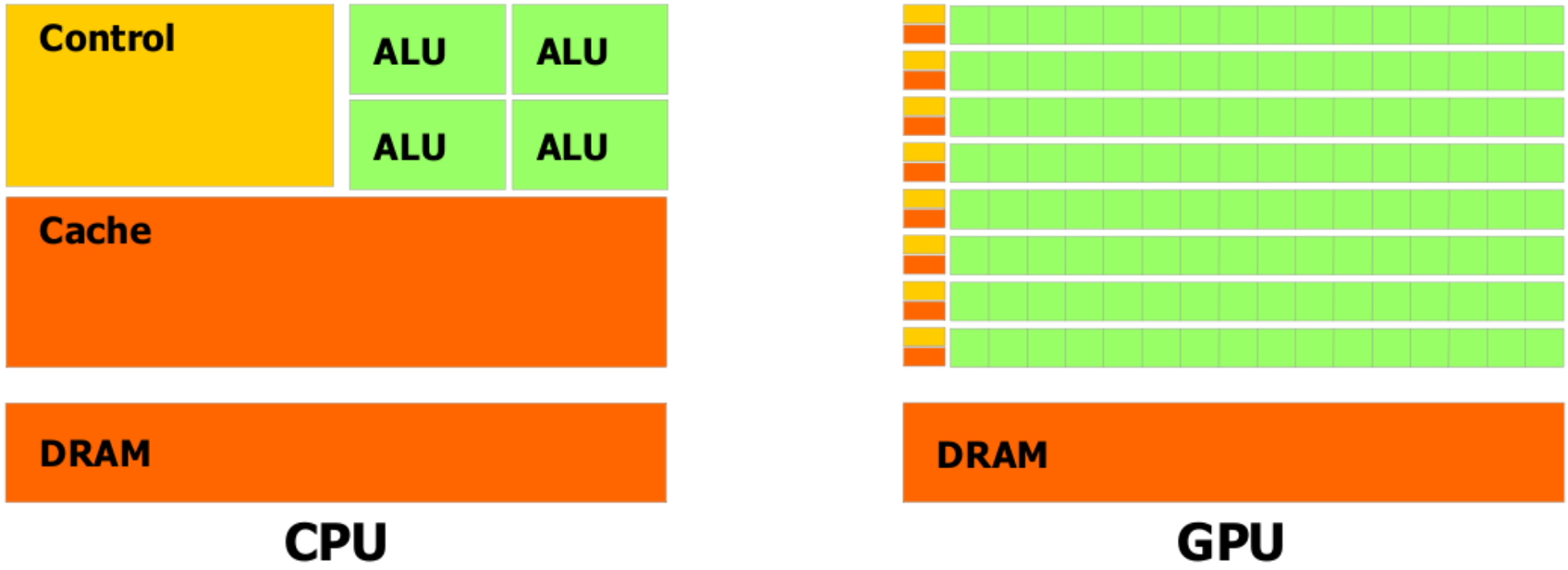
(GP)GPU Programming History - CUDA

- In 2/2007 Nvidia introduced the new computing architecture known as CUDA – Compute Unified Device Architecture.
- This architecture unified the programming interface of all graphical cards produced by Nvidia at that time.
- The programming is made easier. It is not necessary to know the architecture of each graphics card.
- It completely removed the need to use OpenGL. The definition of problems by textures is also eliminated.
- CUDA is small extension of C/C++ language.
- The disadvantage is functionality only on Nvidia cards.
- The programming in/by CUDA is very simple and every programmer can do that. But the effective code writing needs a deep knowledge of GPU architecture to use whole GPU computing power.

Advantages of GPU Computing

- GPUs are designed to run parallelly many hundreds of threads. They can virtually execute hundreds of thousands of threads. Therefore this architecture is called Massively Parallel System.
- All threads have to be independent. The GPU does not guarantee the order of thread execution.
- GPU is designed to process the compute intensive code with a limited number of conditional jumps. Better option is the code without “if” conditions.
- GPU does not support execution out of order (like Intel CPU).
- GPU is optimized for the sequential access to the main (global) memory of graphical card. The transfer speed by the bus is up to hundreds of GB/s.
- Most of transistors in GPU are designed for computing. Other auxiliary circuits are minimized. The scheme on the next slide compares the CPU and GPU architecture:

CPU and GPU Comparison



The scheme above shows the ratio of chip parts in a typical modern CPU and GPU. Green areas represent computing units. Yellow areas are controllers and orange ones are memories.

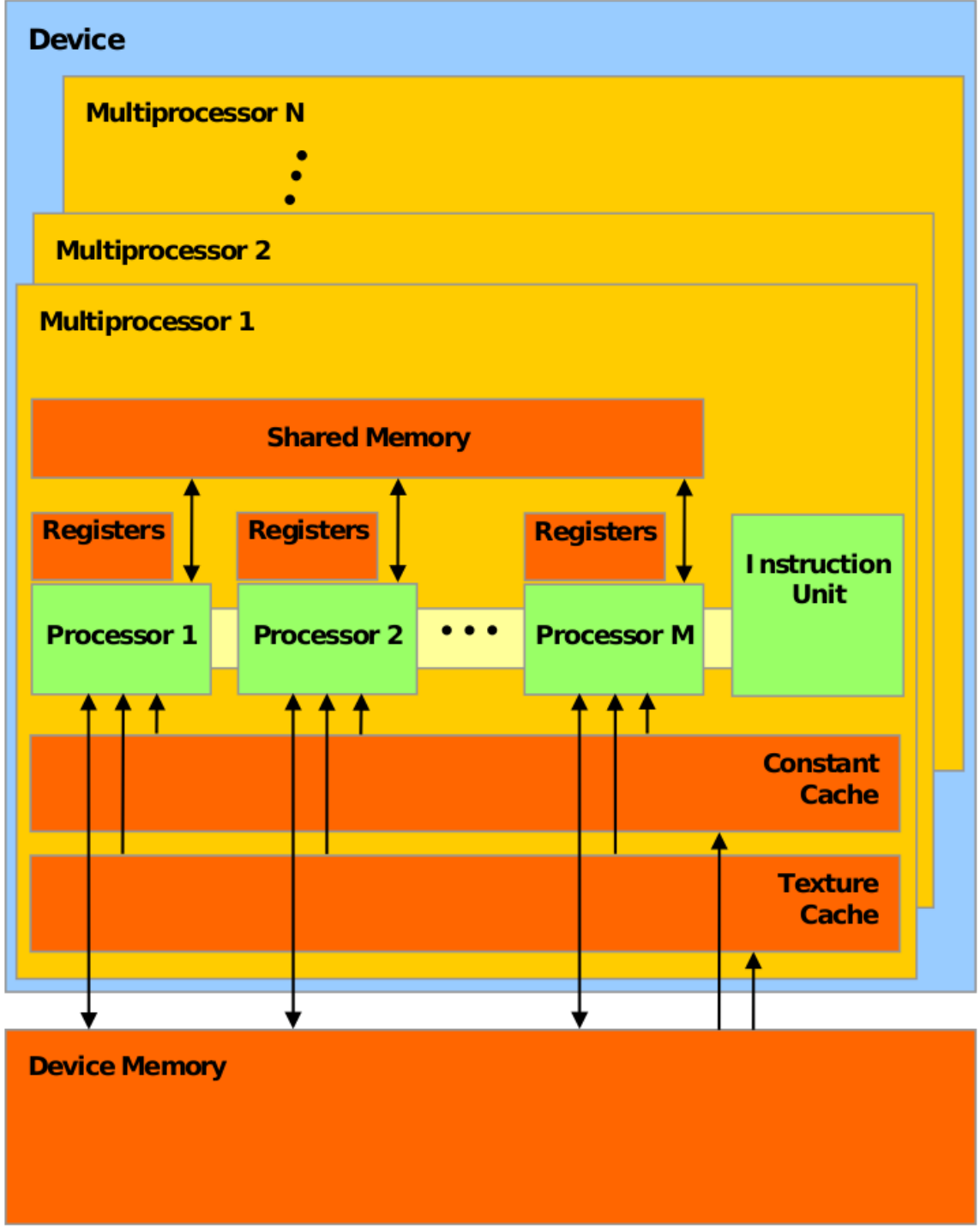
The DRAM memory is not part of GPU nor CPU. But it has to be used to store data for the computing.

… CPU and GPU Comparison

The comparison of two specific products, Nvidia GeForce 580 and Intel i7 with 6 cores, is in the table below. From the table it is clear that Nvidia GPU uses three times more transistors and has up to ten times higher computing performance. The bus transfer speed (throughput) is also significantly higher.

	Nvidia GeForce 580	Intel i7-960 6xCore
Transistors	3000 * 10^6	1170 * 10^6
Frequencty	1.5 GHz	3.5 GHz
Num. of threads	512	12
Performance	1.77 Tflops	~200 GFLops
Throughput	194 GB/s	26 GB/s
RAM	1.5 GB	~48GB
Load	244W	130W

CUDA Architecture



CUDA Architecture

CUDA architecture unified the internal architecture of all Nvidia GPUs. The previous scheme shows that on the highest level there is a graphical card divided to **Multiprocessors**. Every graphics card contains a different number of multiprocessors.

The device memory is shared between all multiprocessors. It consists of three parts: the Global memory, Texture memory and Constants memory.

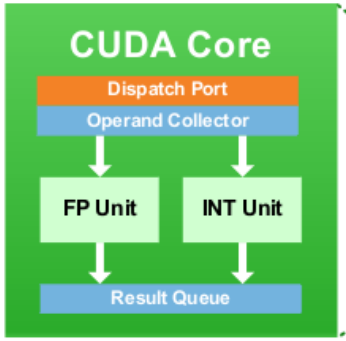
All multiprocessors can share their data only in device memory!

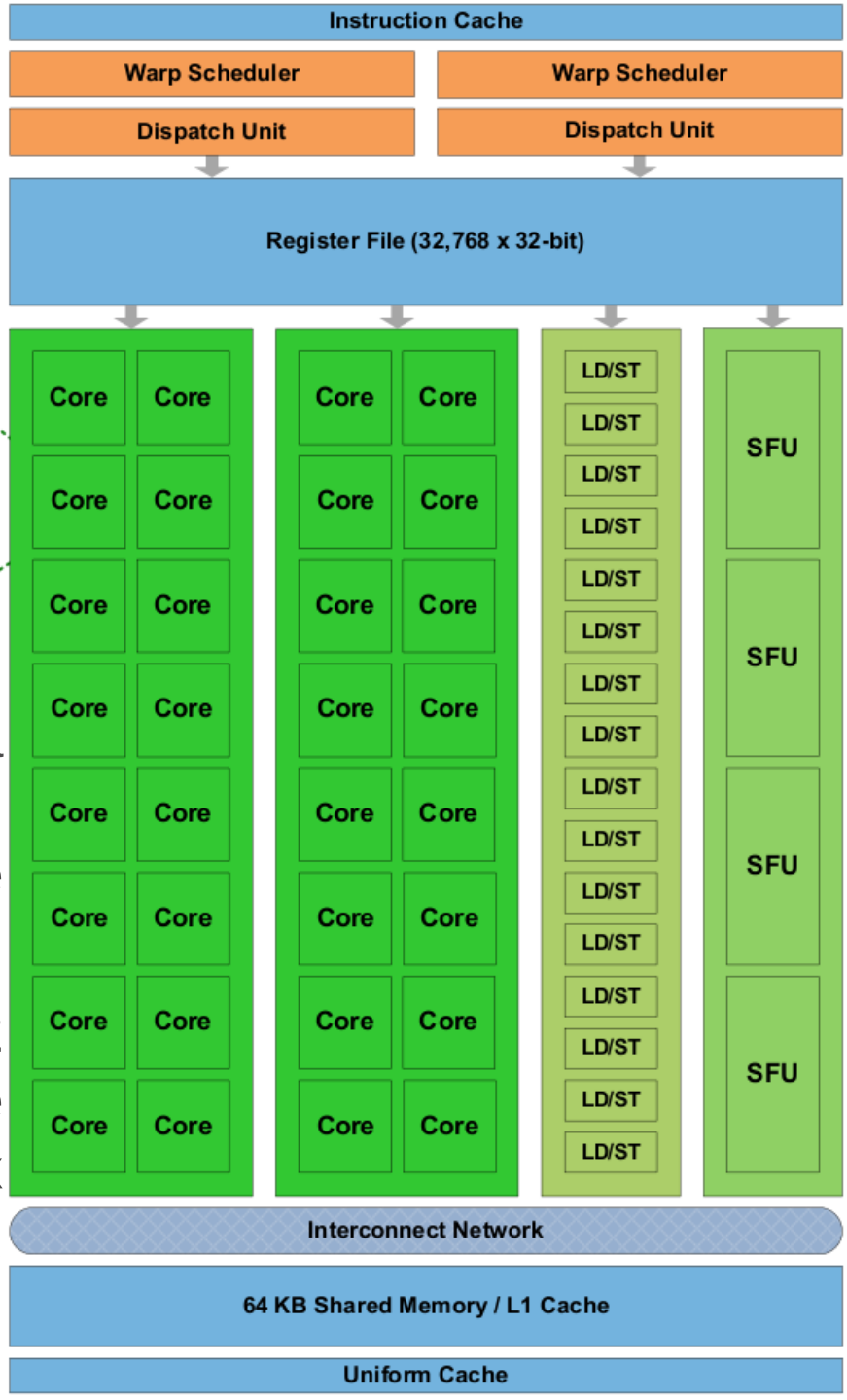
The design of all microprocessors is the same. All of them contain **shared memory** for all **processors** in the single multiprocessor. All processors have their own bank of **registers**.

Every multiprocessor has a cache to speed up the access to texture and constant memories. Both caches are read only.

This unified architecture and terminology facilitates programming.

Fermi Architecture





As mentioned earlier, in 2010 Nvidia introduced its newest GPU architecture. The scheme of one (streaming) multiprocessor is here →

The multiprocessor contains 32 (FPU double) cores, 16 load/store units, 4 Special function units, 32K x 32bit registers and 64 kB shared memory.

Fermi Architecture

The newest GPU architecture is primarily designed for the general computing, not only for graphics cards. The previous scheme shows only one multiprocessor.

This multiprocessor contains 2x16 cores. Each core has one ALU and FPU unit. One group of 16 cores is called half-warp. One half-warp has one instruction decoder.

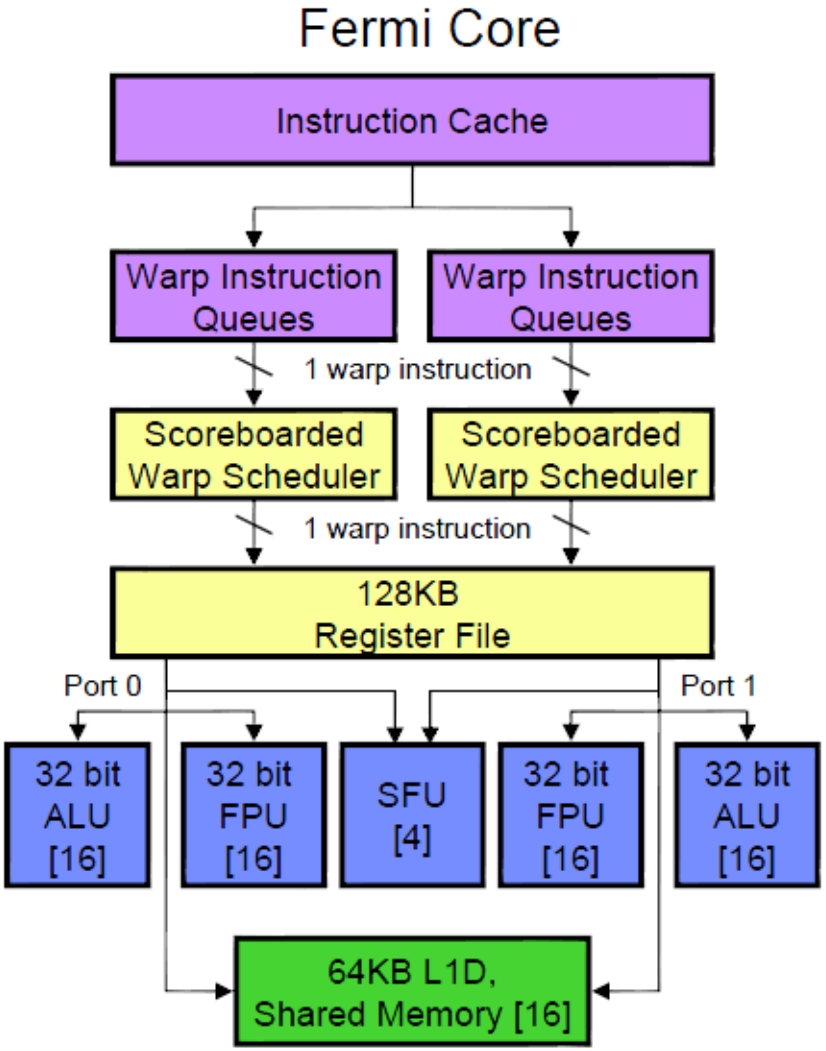
All cores are implemented with 32 kB of registers and 64 kB shared memory and L1 data cache.

Four SFU – Special Floating Point Unit are also available to handle transcendental and other special operations such as sin, cos, exp... Four of these operations can be issued per cycle in each multiprocessor.

FPU Units can compute in single and double precision. ALU units support computation with 32 and 64bits integers.

Fermi Architecture

Here you can see the Fermi architecture in simpler diagram than the manufacturer officially presents (on previous scheme).



Fermi Successors – Kep., Max., Pas., Volta

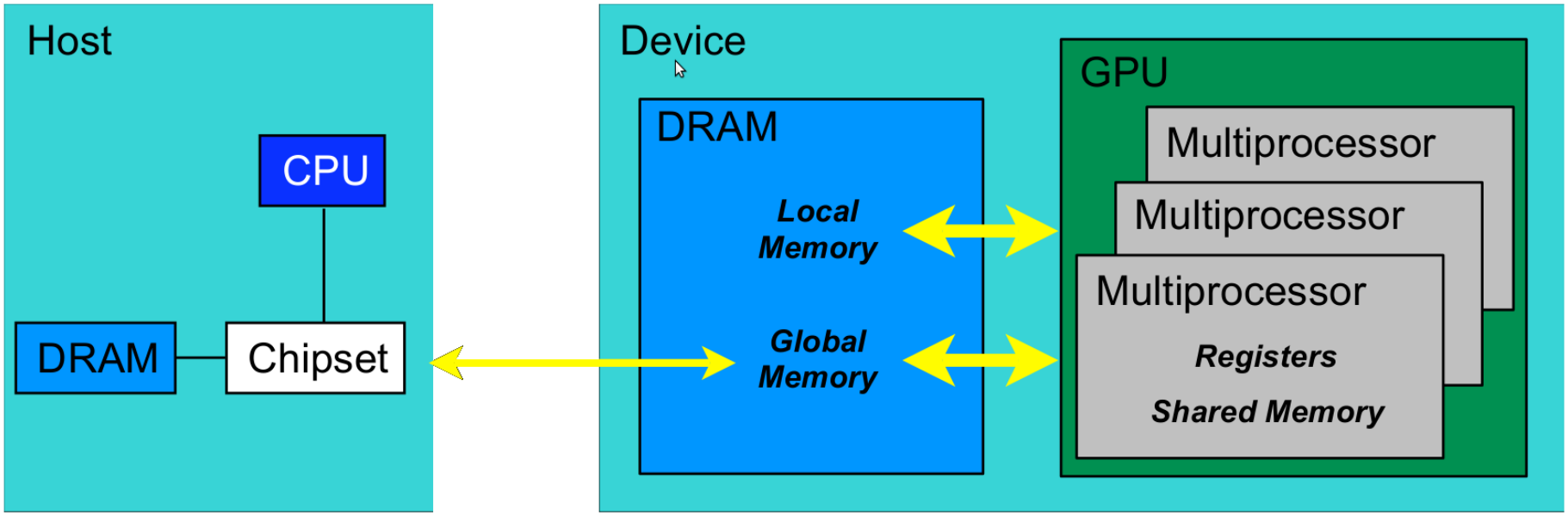
Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm^2	601 mm^2	610 mm^2	815 mm^2
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

[1] Peak TFLOPS rates are based on GPU Boost Clock

Memories in Computer

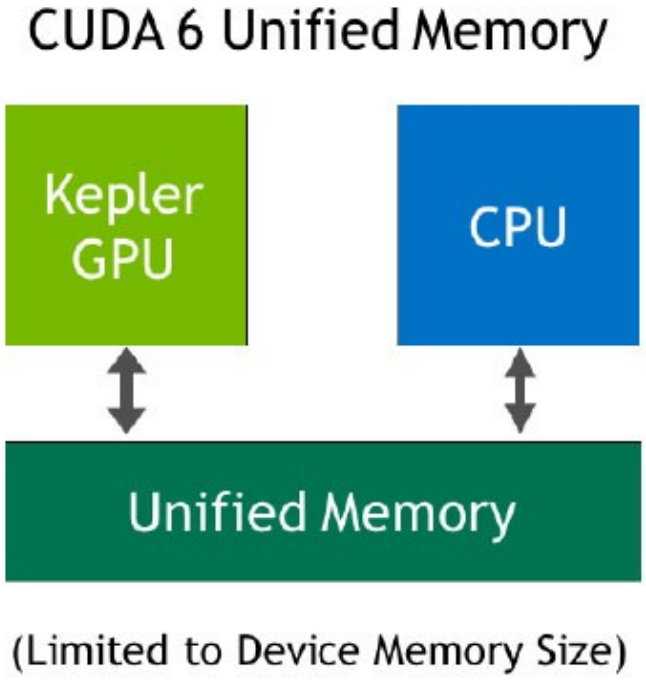
Up to now we have spoken only about GPU and graphics card. These devices are installed in computer and it is necessary to be familiar with memory organization in computer (see below).

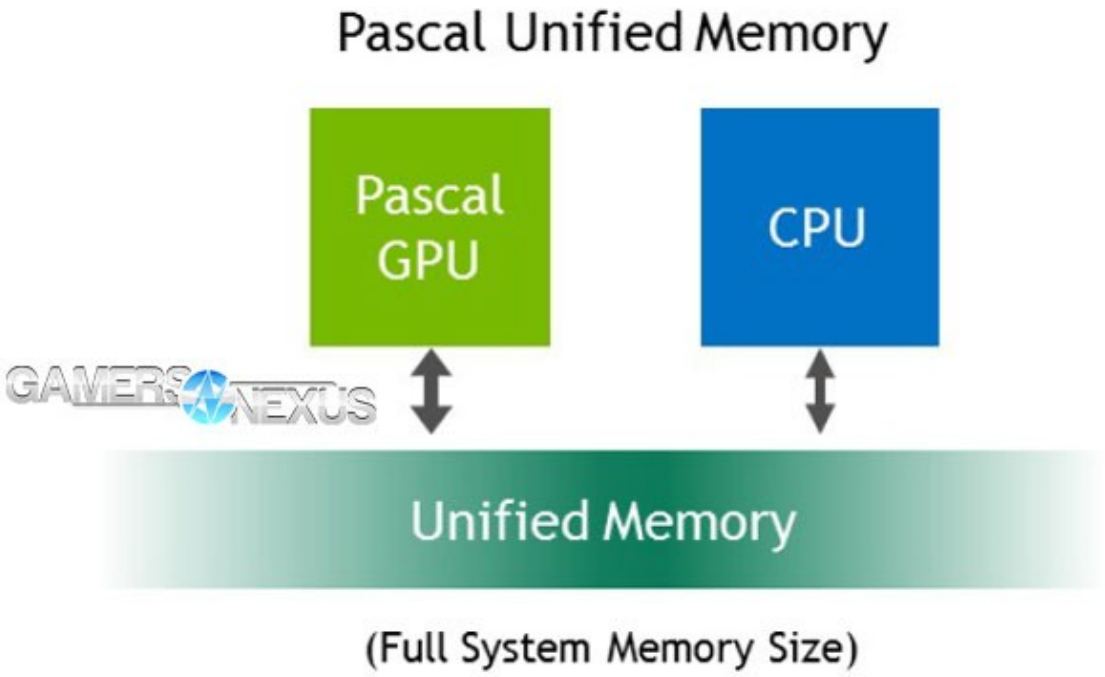
The Device is a graphical card with the DRAM memory and GPU multiprocessors. **The Host** is any computer with installed device. The memory in the device and the memory in the host are connected only by bus. They are not shared!



Unified Memory

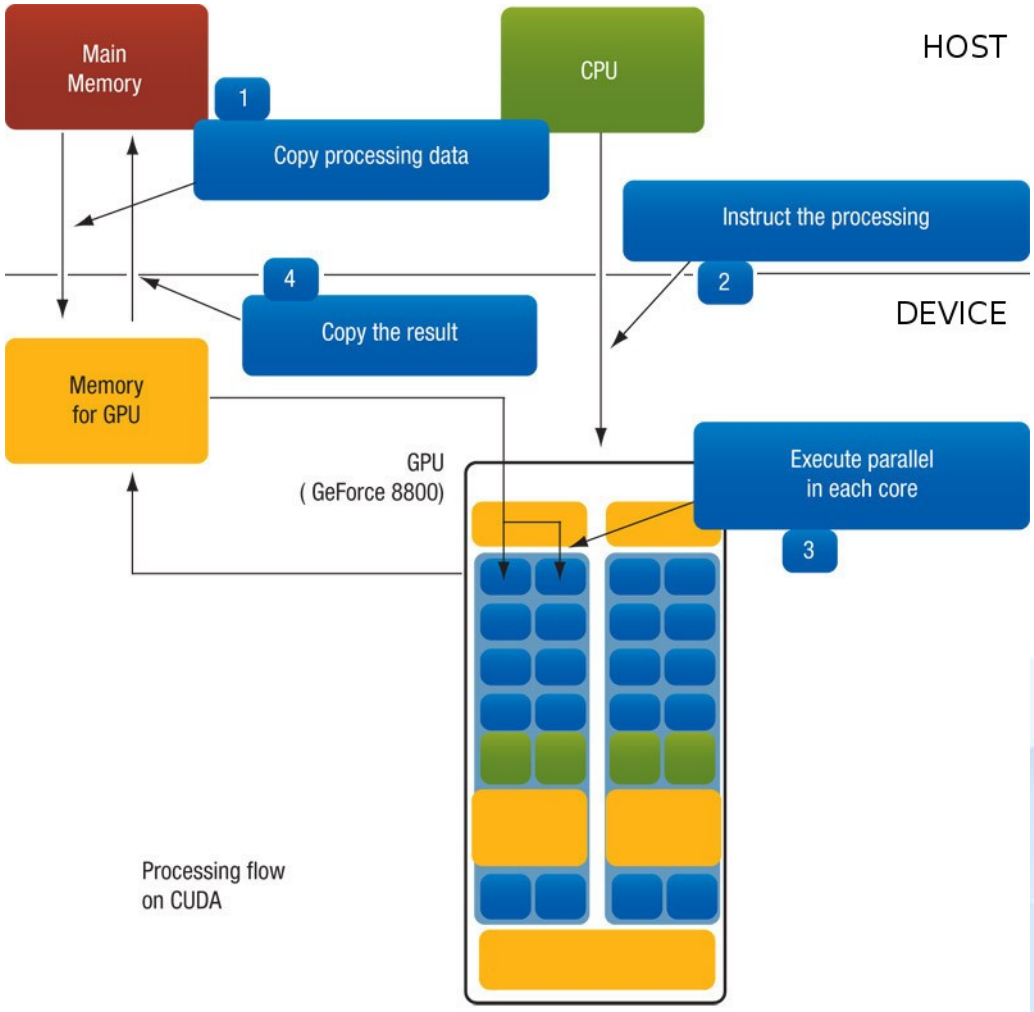
Unified Memory and memory sharing are driving features of modernized GPU architectures. This extends beyond system resources and reaches into L1/L2 Cache and texture cache with GPU.





Computing Process

1. Copy data from the HOST memory to the DEVICE memory.
2. Start threads in DEVICE
3. Execute threads in GPUs multiprocessors.
4. Copy results back from the DEVICE memory to the HOST memory.



… Computing Process

- The typical GPU computing process is shown on the previous diagram. Before the GPU computing is started, it is necessary to move data from the computer main memory to the global device memory. Data is passed by the computer bus. This bus is much slower than memory buses in the computer and graphics card. Today the graphics card uses usually the PCI-Express bus with transfer speed less than 5 GB/s.
- The second step is to transfer code to the GPU and start the computing.
- The third phase is the massively parallel execution of threads. During the GPU computing the CPU can continue its own processes.
- The last step of computing is transfer of computed data from the device memory back to the computer main memory. Then the CPU in computer can evaluate and use the results.

GPU Computing Efficiency

It is important to keep in mind some basic rules to maximize efficiency of GPU computing:

- Minimize data transfer between host and device. In ideal case transfer data only twice. Before and after computing.
- Use GPU only for task with very intensive calculations.
- GPU with shared memory on the board would be more suitable.
- For intensive data transfer between CPU-GPU use pipelining.
- GPU computing can be used alongside data transfer GPU-CPU or CPU computing.
- Optimize access to shared memory. Sequential access is much faster then random access.
- Reduce divergent threads.
- Select optimal thread grid.

CUDA Programming - Extensions

CUDA is not only a unified GPU architecture. CUDA is mainly programming interface that allows to use GPU computing. Today CUDA is available for C/C++, Fortran, OpenCL, Direct computing, Java and Python.

Now we will shortly deal with the programming in C/C++. CUDA introduced a few language extensions to the standard.

- The **kernel** is a function for GPU threads. The C/C++ is extended for kernel function execution by command **“name<<<...>>>(...)”**.
- **__device__** is a function modifier. This function will be executed in the device and it can be called only from the device.
- **__host__** is opposite function modifier than **__device__**. Functions marked with this modifier are only for the CPU.
- **__global__** is modifier for kernels. Function will be executed in GPU, but called (started) is from CPU.

… CUDA Programming - Extensions

Variable type qualifier:

- __**device**__ declares a variable that resides in the device as long as application is running.
- __**constant**__ is used for variables places in constant memory.
- __**shared**__ variable resides in shared memory of the block, where kernel is running.

New types are defined for data shared between the GPU and CPU:

All common data types – **char, uchar, int, uint, short, ushort, long, ulong, float and double** are used **as structures** with suffix **1, 2, 3** or **4**. For example **int3** is a structure with 3 items. All members in structures are accessible by **x, y, z** and **w** fields. The grid dimension uses type **dim3** = **uint3.**

```
int3 myvar;
myvar.x = myvar.y = myvar.z = 0;
```

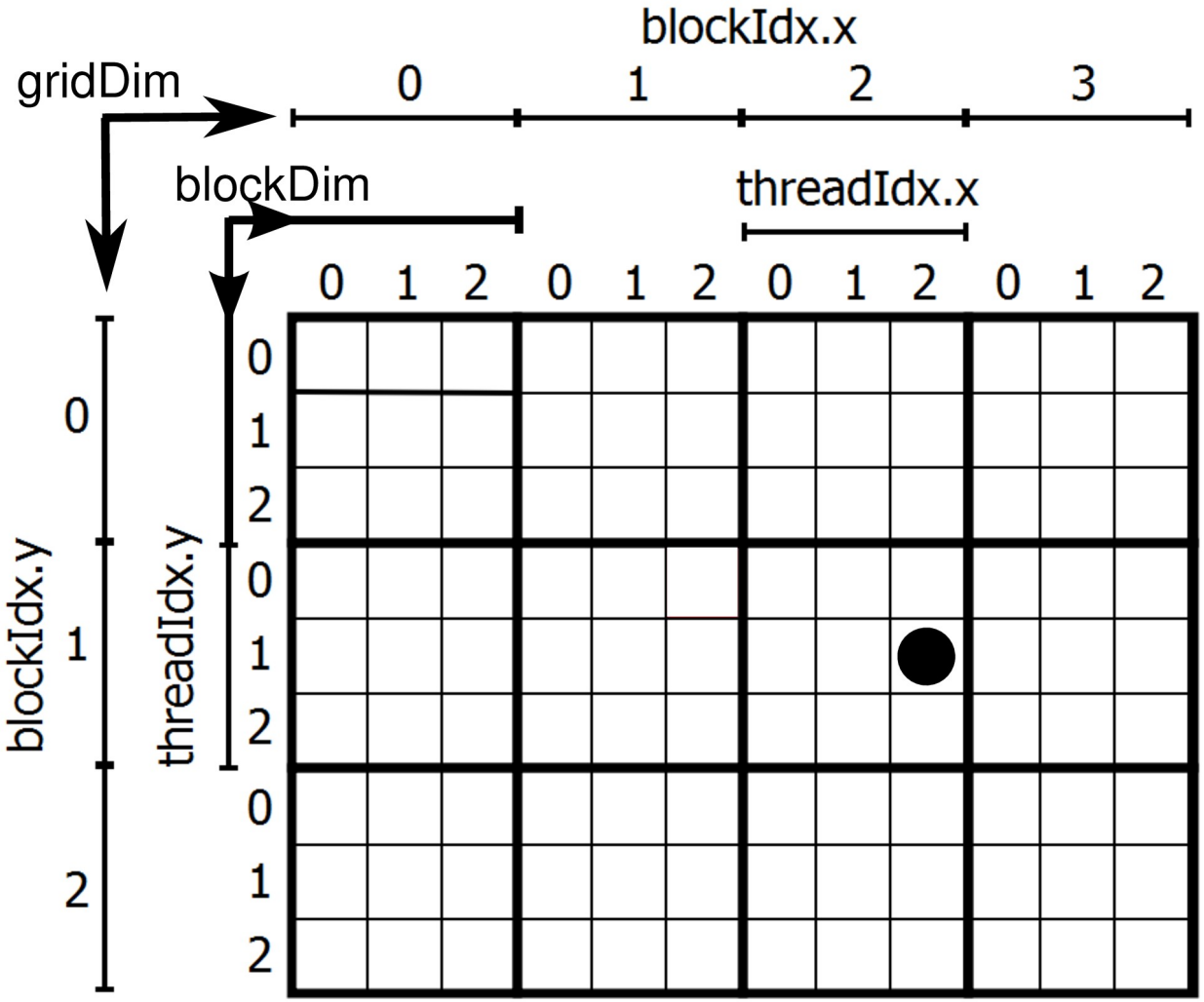
… CUDA Programming - Extension

All threads are organized in a grid (see below) and every thread in this grid has the exact position. Every thread has predefined variables to identify itself among others.

Predefined variables:

- **dim3 gridDim** contains dimension of the whole grid.
- **uint3 blockIdx** contains block position in grid.
- **dim3 blockDim** contains dimension of block (all blocks in the grid have the same dimension).
- **uint3 threadIdx** position of thread in block.
- **int warpSize** contains the warp size in threads.

CUDA – Grid Organization



x = blockIdx.x * blockDim.x + threadIdx.x

y = blockIdx.y * blockDim.y + threadIdx.y

CUDA – Grid Organization

The grid organization is shown on the previous scheme. It is very important for programmers to understand it! But not only understand it. Programmers have to propose the grid before starting the kernel. The kernel code has to be adjusted to the proposed grid. Here, everything relates to everything.

When some thread from the defined grid needs to know its own position, it has to use predefined variables to correctly identify, which part of problem will be computed just in this thread and kernel.

The method of calculation was introduced in the previous diagram. Every thread has to use predefined variables **blockDim, blockIdx** and **threadIdx**. Then it is necessary to use the following formulas:

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

The *x* and *y* are the exact positions in the grid (2D).

CUDA – API

CUDA API contains up to one hundred functions. Here are but a few for starters:

- **cudaDeviceReset()** - function (re)initialize the device.
- **cudaDeviceSynchronize()** - synchronize device and host.
- **cudaGetLastError()** - returns **cudaSuccess** or error code.
- **CudaGetErrorsString(...)** returns string for error code.
- **CudaMalloc(...)** - allocates memory in device.
- **CudaMallocManaged(...)** - allocates unified memory.
- **CudaFree(...)** - releases allocated memory.
- **CudaMemcpy(...)** - copies memory between host and device. Direction of copying is given by the value **cudaMemcpyHostToDevice** or **cudaMemcpyDeviceToHost**.
- **printf()** - CUDA capabilities 2.0 and higher allows to use printf. The output is not displayed on-line, but at the end of computation.

CUDA – Example

```
// printf() is only supported
// for devices of compute capability 2.0 and above

__global__ void helloCUDA(float f)
{
  printf(“Hello thread %d, f=%f\n”, threadIdx.x, f);
}

void main()
{
  helloCUDA<<<1, 5>>>(1.2345f);

  if ( cudaGetLastError() != cudaSuccess) {
    printf( “Error: %s\n”, cudaErrorString(
                              cudaGetLastError() ) );
  }
  else
    printf( “Kernel finished successfuly\n” );

  cudaDeviceSynchronize();
}
```

… CUDA – Example

More examples are in ZIP archive on the web pages for this subject.