Introduction to Linux and Supercomputers GPU

MA030366 (Term 2) R2-B5-2026

Zacharov Igor

i.zacharov@skoltech.ru

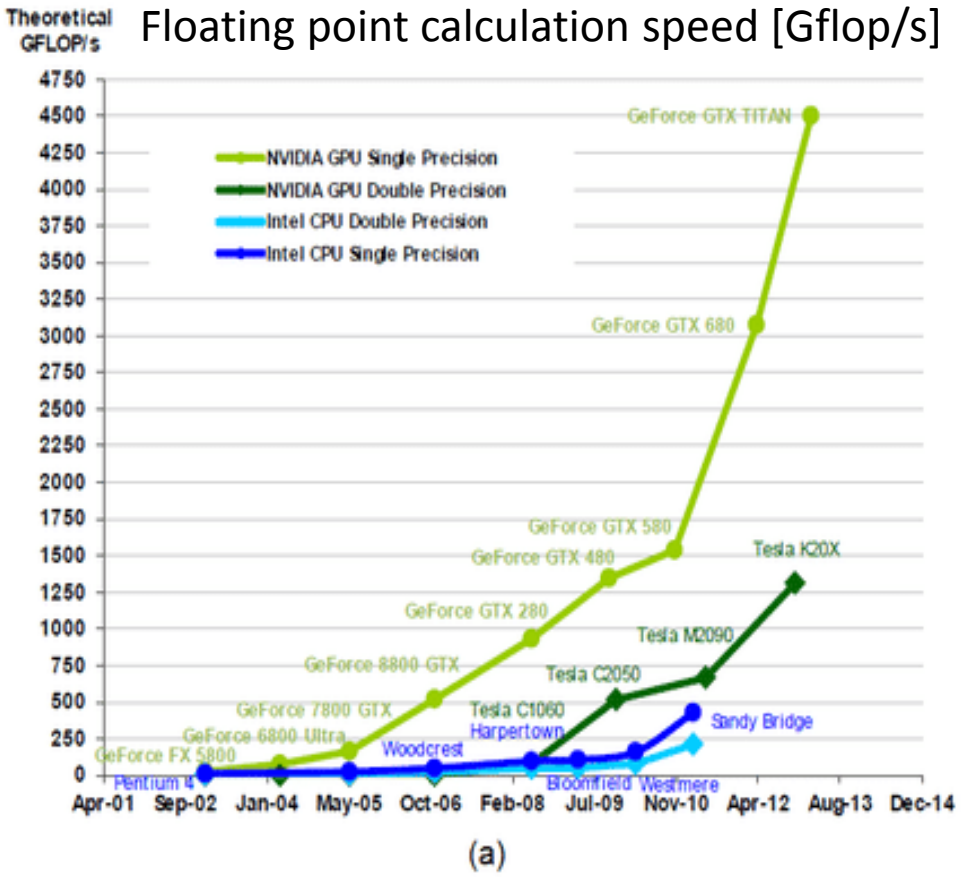
GPU

- GPGPU – general purpose graphical processing unit
- Originally used only to accelerate the screen drawing operations
  - Drawing on the screen consists of multiple independent (floating point) calculations
    - Geometry “engine”: tracing a path a light would take from an object to pixel on the screen
  - The GPUs have been designed to support massively parallel FP
    - Limited accuracy required (Single precision FP)

- With the advent of parallel computing the GPGPU is used just for its massively parallel computing capabilities
  - Accelerators of FP intensive calculations
    - Chemistry
    - Machine Learning
    - …. Etc.
- On the down side – the devices have to be programmed specially
  - Does not work “out of the box”
  - Porting of application is needed to use the GPGPUs
- The porting effort is justified:
  - by massive spread of GPGPU availability
  - Faster development of GPGPU hardware as compared to traditional CPU

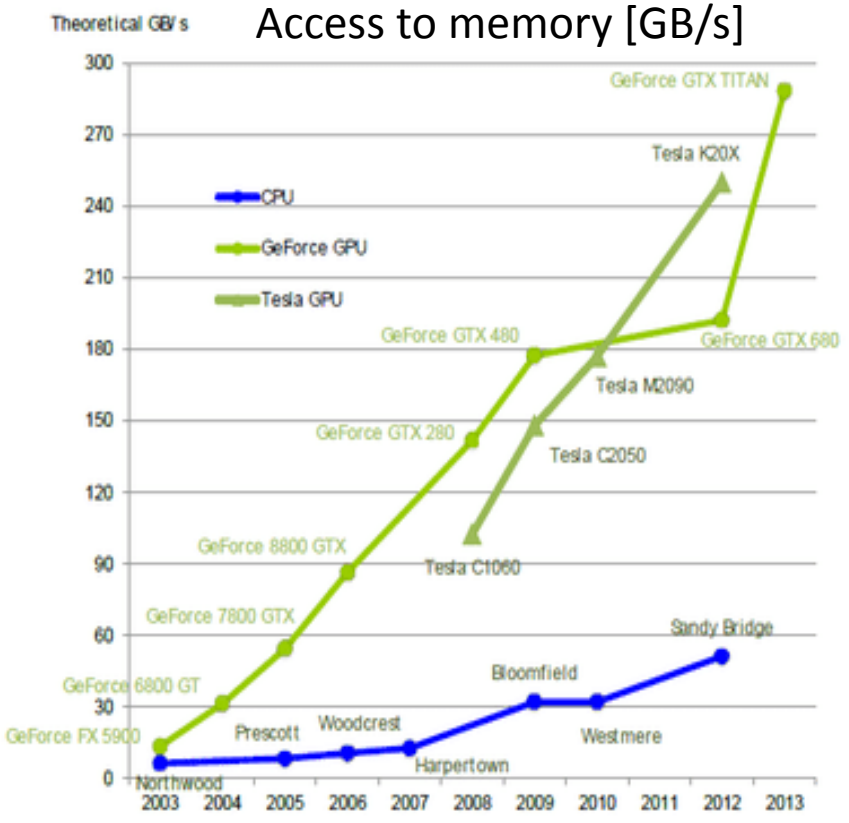
GPU vs CPU development

Floating point calculation speed [Gflop/s]



(a)

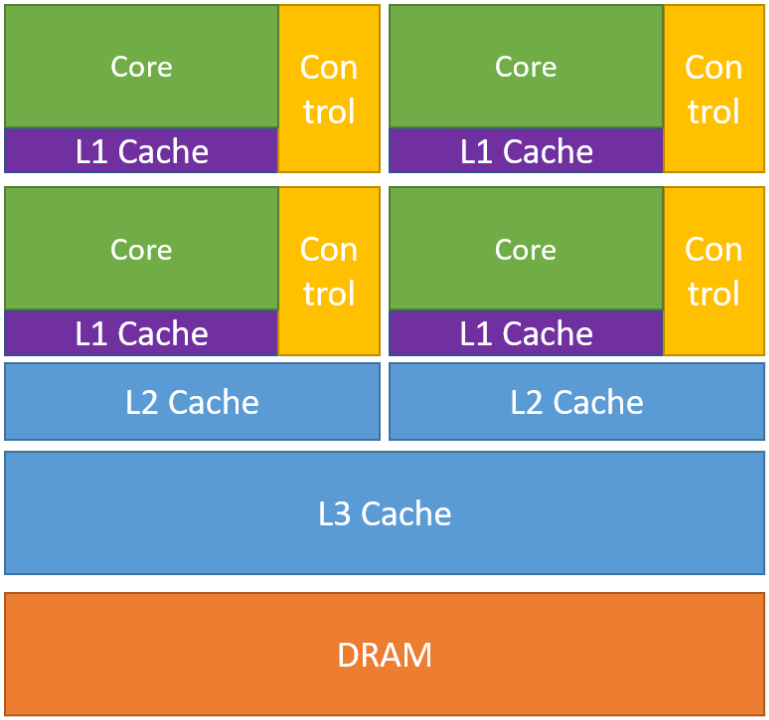
Access to memory [GB/s]



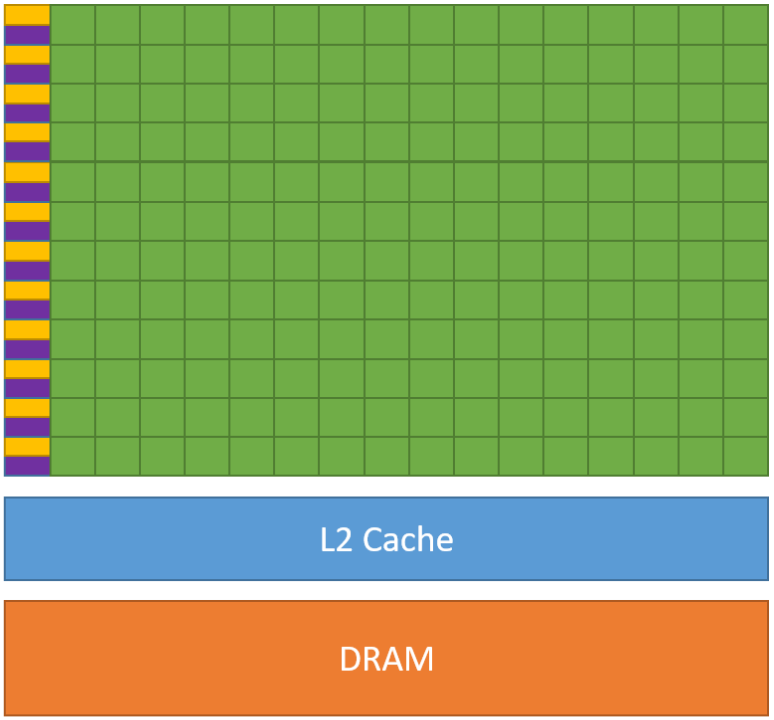
(b)

Pictures: A. Vieira et al.

GPU vs CPU comparison



CPU

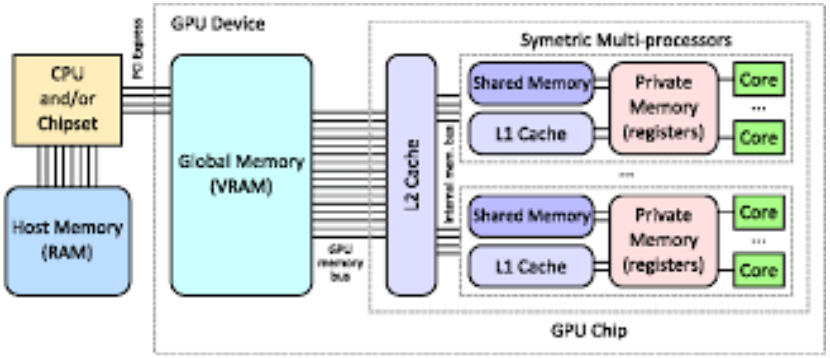


GPU

CPU	GPU
Few “fast” execution units (2 x 4 GHz)	Many “slow” execution units (1000 x 1 GHz)
Large amount of “slow” memory (400 GB, 0.1 TB/s)	Small amount of “fast” memory (40 GB, 1TB/s)
General purpose (1-10s parallel threads)	Specialized parallel only code (100s-1000s threads)
Runs OS and manages all resources	Attached device over a “slow” bus (30 GB/s)

GPU Architecture

- GPU has a special hardware with multiple symmetric multiprocessors (SM)
- Several layers of memory
  - The general architectural approach:



- A100 GPU:
  - The latest

    Nvidia Ampere architecture:

SM

L1 Cache

GA100 Full GPU with 128 SMs

GPU capabilities

GPU Model	DP [TF]	FP16 [TF]	Tensor [TF]	Err	Mem [GB/s]	Mem [GB]	Software capability			
							DMA	RDMA	HyprQ	Boost
GeForce GTX 1080 Ti	< 0.35	< 0.17			484	11	1	NA	Local	Td
GeForce Titan Xp	< 0.38	< 0.19			548	12	2	NA	Local	Td
GeForce Titan V	6.87	27.5	110		653	12	1	NA	Local	Td
GeForce RTX 2080 Ti	0.44	28.5	56.9		616	11	1	NA	Local	Td
Tesla P100 (16 GB)	4.7 – 5.3	18.7-21.2		ECC	732	16	2	Direct	Rem	Frq
Tesla V100 (16/32GB)	7-7.8	28-31.4	112-125	ECC	900	16/32	2	Direct	Rem	Frq
Quadro GV100	7.4	29.6	118.5	ECC	870	32	2	Direct	Rem	Frq
Quadro RTX 6k/8k	0.5	32.6	130.5		624	24/48	2	Direct		
Tesla T4	0.25	16.2	65	ECC	320	16	2	Direct	Rem	Frq

The different (Nvidia) cards address different markets

- most money is made in the games market
- HPC is on an edge and for us the selection is not easy

GPU programming

- CUDA is Nvidia's way to program the GPUs
  - Single vendor approach
  - Most advanced programming tool for largest market share of devices
  - Mature C-like programming with a lot of applications & libraries
  - Supports full frameworks for applications (eg. Machine Learning)

- ROCm is AMD's way to program the GPUs
  - Open-source
  - Due to smaller market share is less well developed
  - AMD has CUDA converters to ROCm

- OpenCL is a vendor independent programming language
  - Capable of producing code for Nvidia and AMD graphics chips
  - Targeting other accelerators in principle (eg. FPGA)
  - Performance hit (of about 30% with respect to CUDA)
    - Your mileage may vary
- OpenACC is set of compiler directives to express parallelism
  - Depends on the PGI compiler

You program for a GPU not to be flexible, but to get the highest performance

Programing GPU: CUDA

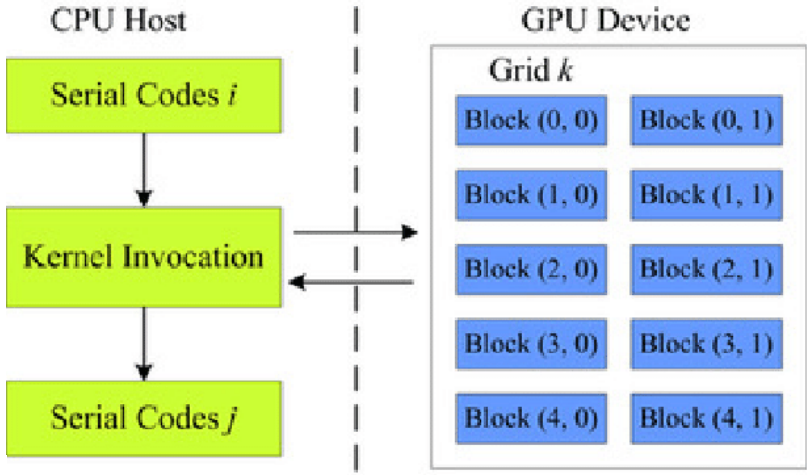
- Compute Unified Device Architecture (CUDA)
  - Introduced by Nvidia in late 2006
  - It is a compiler & runtime toolkit for programming Nvidia GPUs
  - CUDA API extends the C programming language
  - Runs on thousands of threads and is designed to be highly scalable

- CUDA objectives:
  - Express parallelism in C-type language
  - Give a high level abstraction from hardware

- CUDA introduced by Nvidia for developers to promote the hardware
  - Nvidia was successful with this marketing model: multi-billion $$ market
  - New GPU generation is introduced about ~18 months
  - Links: www.nvidia.com http://www.nvidia.com/cuda_home.html

Principle of CUDA programming

- Program starts on the CPU (host)
  - All of the serial code (eg. initialization) is running on the cpu
  - For specially written code (the “kernel”) the code is running on the GPU
    - Nvidia run-time arranges this run transparently to the programmer
    - The kernel has C-language description



- Data & code transfer over PCIe
  - Run dedicated mode on available hardware
  - Device memory (Device “global memory”)

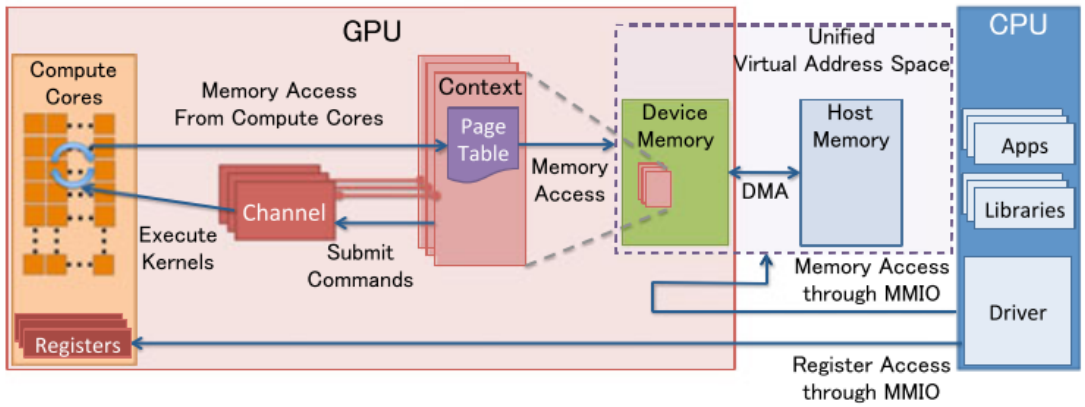
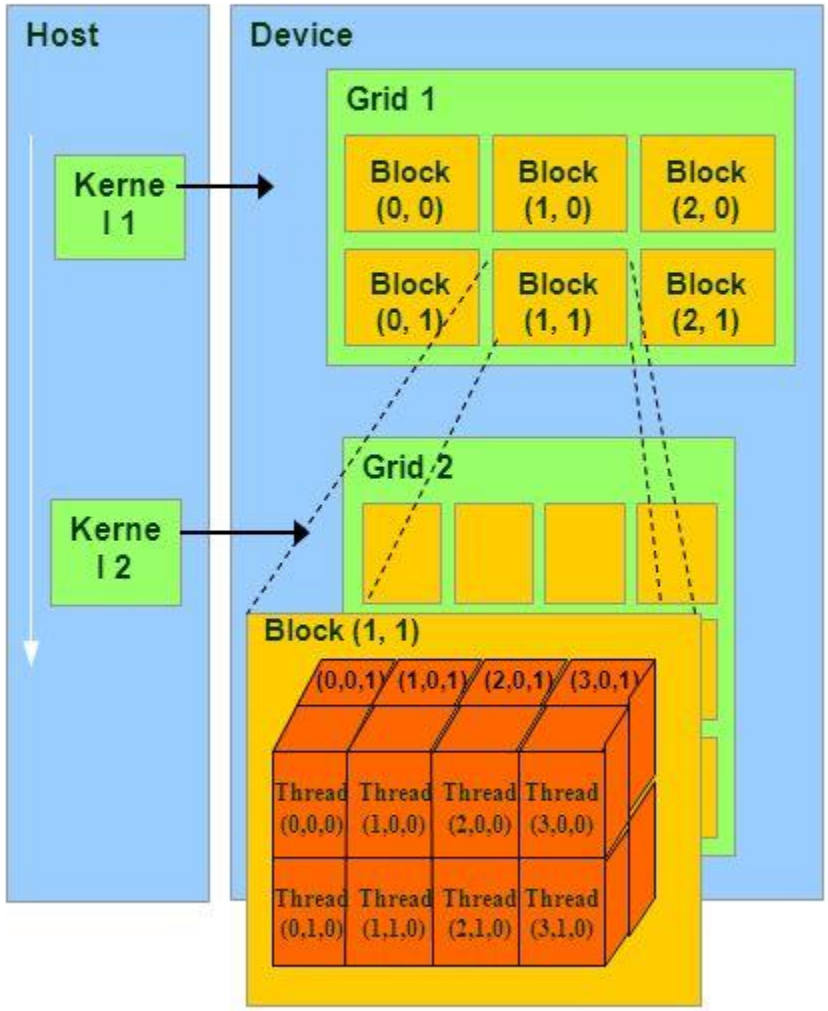


Fig. 1. GPU resource management model.

CUDA Programming Model

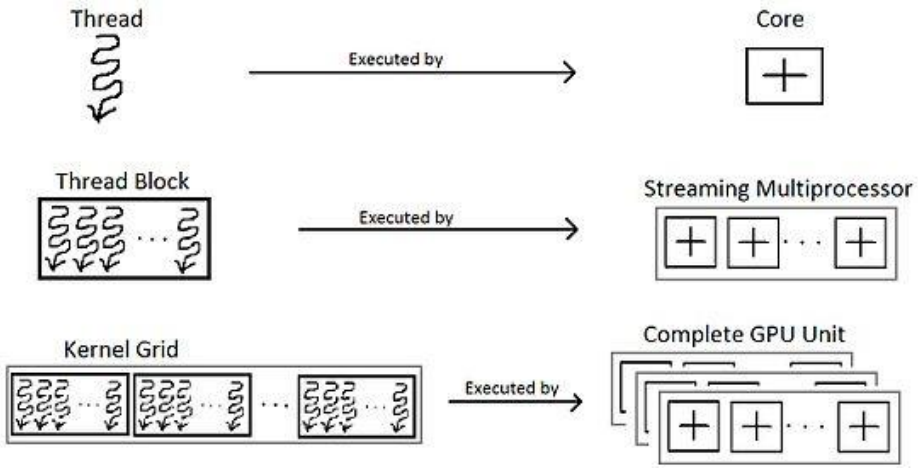
- A kernel is executed as a grid of thread blocks
  - Grid of blocks can be 1 or 2-dimentional
  - Thread blocks can be 1, 2, or 3-dimensional
- Different kernels can have different grid/block configuration
- Threads from the same block have access to a shared memory and their execution can be synchronized



Slide is courtesy of NVIDIA

CUDA programmer effort

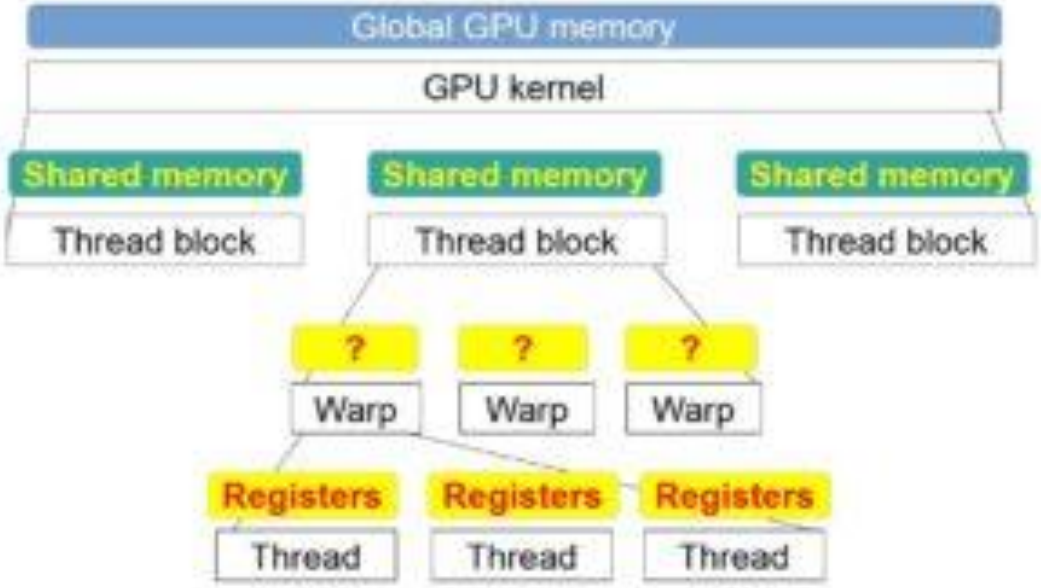
- Analyze algorithm for exploiting parallelism
  - Block size (data that can be grouped together)
  - Number of threads
- Challenge: “keep the machine busy” – efficiency
  - Put global data set on the GPU for an efficient data transfer
  - Local data set & Register set have limited on-chip memory
- Hierarchies of abstractions:
  - Kernels are executed by threads
    - Kernel is a simple C function
    - Each thread has its own ID
    - Thousands of threads execute same kernel
    - Threads are grouped into blocks
    - Threads in a block can synchronize execution
    - Blocks are grouped in a grid
    - Blocks are independent and
      - must be able to be executed in any order



- The Amdahl law is in full force: Scalability is limited by serial code
  - Data transfer to/from Host/Device

GPU memory hierarchy

- Three types of memory on a graphics card:
  - Global memory (~16 GB)
    - latency ~500 cycles
    - Purpose: I/O for grid
  - Shared memory (~64 KB)
    - latency ~10 cycles
    - Thread collaboration
  - Registers (64 KB)
    - latency ~1 cycle
    - Thread local stack



- The warp is a collection of (32) threads
  - executed simultaneously by an SM
  - Multiple warps can be active at the same time (concurrently) on an SM
    - Block is a logical collection of threads, while warps is its physical implementation
    - Threads are divided into warps: WarpsPerBlock =(ThreadsPerBlock + WarpSize-1)/WarpSize

Basic C extensions in CUDA

- Function modifiers
  - Called by host code, executed by host: __host__
  - Called by host code, executed on the GPU: __global__
  - Called by device code, executed on the GPU: __device__
    - Function with __device__ qualifier can be called from other such functions or from __global__ functions (not from __host__ functions)

- Kernel launch parameters
  - Block size (x,y,z)
  - Grid size (x,y) : Maximum of 1000s threads

- Variable modifiers, per block
  - variables private to threads in a block, shared within that block
  - Variable in shared memory: __shared__
  - Synchronization of threads within a block: __syncthreads()

Example: Hello, World!

- Nvidia uses its own compiler environment: nvcc
  - Wrapper around the default gcc compiler
  - File name extension: cu
    - Ex.: hello.cu
  - Compile with nvcc

```
module load compilers/gcc-7.3.1
module load gpu/cuda-11.3
nvcc –o hello hello.cu
```

  - run with:

```
srun -p gpu --gpus=1 -c 1 --mem=1G -t 1 hello
```

- Runtime parameters

```
kernel<<<B,T>>>( args );
```

B – Threads per grid = number of blocks

T – Threads per block

  - Example of a distribution:
    - N – vector length

```
ThreadsPerBlock = 256;
ThreadsPerGrid  = (N+ThreadsPerBlock -1)
                        /ThreadsPerBlock
```

```
#include <stdio.h>
#include "commoncuda.h“  //def of CUDA_CHECK macro

/* kernel definitions */
__global__ void add( int a, int b, int *c )
{
   *c = a + b;
}

int main()
{
   int c;
   int *dev_c;
   CUDA_CHECK( cudaMalloc((void**) &dev_c,sizeof(int)));

   add<<<1,1>>>( 2,9, dev_c );

   CUDA_CHECK( cudaPeekAtLastError() );
   CUDA_CHECK( cudaMemcpy( &c, dev_c, sizeof(int),
                                cudaMemcpyDeviceToHost ));

   printf( "Hello, World! --- Result: 2 + 7 = %d\n", c);
   cudaFree( dev_c );
   return 0;
}
```

Example: Vector addition

Function to add vectors
Will be executed in parallel on the GPU

GPU runs N copies
of the kernel code

Memory allocation on the GPU
For all 3 arrays

Transfer data to the GPU
for two arrays a, b

Execute kernel on the GPU with number of blocks equal to vector length N

Transfer data back to HOST
for the resulting array c

Free memory on the device

```
#define  N  10      // static vector length

__global__ void add( int *a, int *b, int *c )
{
  int i = blockIdx.x; // buildin CUDA variable, unique within a block
  if ( i < N )        // being paranoid, since MAX(blockIdx.x) = N-1
        c[i] = a[i] + b[i];
}

int main()
{
  int a[N], b[N], c[N];          // arrays on the HOST
  int *dev_a, *dev_b, *dev_c;    // pointers representing arrays in GPU
  const int size = N * sizeof(int);

  CUDA_CHECK( cudaMalloc( (void**) &dev_a, size ) ); // allocate memory
  CUDA_CHECK( cudaMalloc( (void**) &dev_b, size ) ); // on the GPU
  CUDA_CHECK( cudaMalloc( (void**) &dev_c, size ) ); // dev_ is pointer

  for (j=1, i=0; i<N; i++, j++) { a[i] = 123.0 * j; b[i] = 456.0 *j; }

  CUDA_CHECK( cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice ) );
  CUDA_CHECK( cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice ) );

  add<<<N,1>>>( dev_a, dev_b, dev_c ); // N blocks, 1 thread per block
    /* N is the number of parallel blocks = grid size */

  CUDA_CHECK( cudaPeekAtLastError() );
  CUDA_CHECK( cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost ) );

  for(i=0; i<N; i++) printf( "%d + %d = %d\n", a[i], b[i], c[i]);

  cudaFree( dev_a ); cudaFree( dev_b ); cudaFree( dev_c );
  return 0;
}
```

Runtime parameters

- The kernel invocation: kernel<<<B,T>>>(args);
  - B and T have hardware limitation (eg. for VT100 and GTX 1080)
    - Number of threads: Max threads per block: 1024
    - Number of blocks in a single launch: 65535
- The <<<B,T>>> specification can be scalar or 3D:

```
dim3  grid(N,N);                // dim3 is CUDA defined Nvidia structure, 3rd dim = 1
kernel<<<grid,1>>>( args );     // kernel has access to blockIdx.x, blockIdx.y
```

- Or (also in combination)

```
dim3  grid(N,N);                // dim3 is CUDA defined Nvidia structure, 3rd dim = 1
kernel<<<1,grid>>>( args );     // kernel has access to threadIdx.x, threadIdx.y
```

- For longer vectors
  - iterate inside the kernel
  - strides can be calculated :
  - Call: add<<<128,128>>> (args);
    - blockDim.x = 128, gridDim.x = 128

```
__global__ void add(int *a, int *b, int *c) {
        int tid =  threadIdx.x +
                   blockIdx.x * blockDim.x;
        while ( tid < N) {
            c[tid] = a[tid] + b[tid];
            tid += blockDim.x * gridDim.x;
        }
}
```