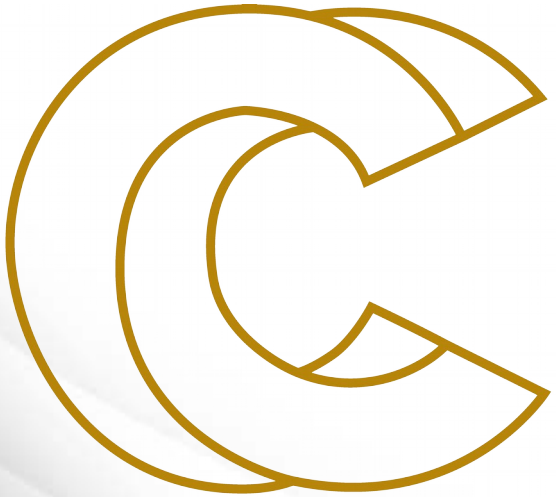
Introduction to parallel programming accelerators



EURO

Leon Bogdanović

University of Ljubljana, FME, LECAD lab

Graphics accelerators: definition

NVIDIA Tesla V100

Graphics accelerators or **graphics processing units** (**GPUs**) are devices with:

- many highly parallel streaming multiprocessors
- very high bandwidth memory

Applications:

- for intensive 3D graphical rendering, ray tracing etc. (**graphics** applications)
- for GPGPU (General Purpose GPU) computing (**scientific** and **engineering** applications)

Motivational example: CUDA ray tracing

- clone the repository from bitbucket to your viz.hpc.fs.uni-lj.si account:

```
$ git clone https://bitbucket.org/lecad-peg/eurocc-accelerators.git
```

- to build the executable with the CUDA ray tracer follow these steps:

```
$ cd eurocc-accelerators/CUDA_ray_tracing
$ source setupenv.sh
$ make
```

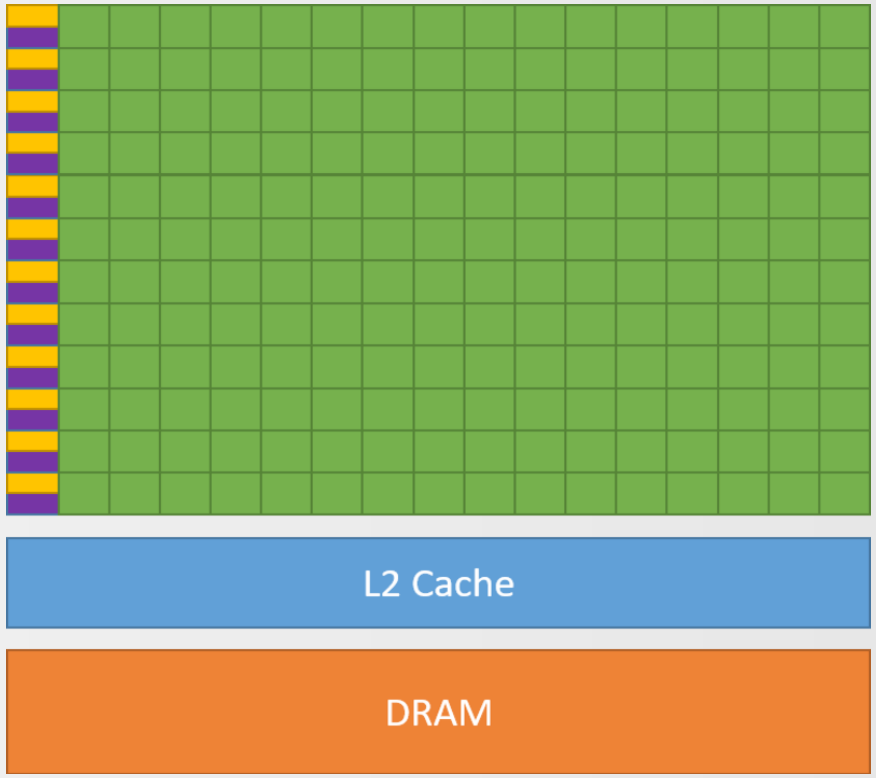
- to run the CUDA ray tracer execute:

```
$ ./run.sh
```

or

```
$ ./a.out
```

Graphics accelerators: architecture



Schematic of a GPU (source: nvidia.com)

GPU architecture:

- **global memory**: 0.5-80 GB, very high bandwidth (up to 2000 GB/s)
- **streaming multiprocessors** (SM):
  - groups of many parallely executing ALU cores
  - many registers (32-64 KB)
  - very fast shared memory
  - SIMT scheduling (older micro-architectures), Independent Thread Scheduling (Volta and Ampere micro-architecture on NVIDIA cards)

GPUs: consumer grade vs. high-end

NVIDIA GeForce 930MX

Output from *deviceQuery*:

Total amount of global memory:

2004 MBytes (2101870592 bytes)

(3) Multiprocessors, (128) CUDA Cores/MP:

384 CUDA Cores

GPU Max Clock rate: 1020 MHz (1.02 GHz)

Memory Bus Width: 64-bit

L2 Cache Size: 1048576 bytes

Output from *bandwidthTest*:

Device to Device Bandwidth, 1 Device(s)

Transfer Size (Bytes)	Bandwidth(MB/s)
33554432	13193.8

NVIDIA Tesla V100-SXM2-16GB

Output from *deviceQuery*:

Total amount of global memory:

16128 MBytes (16911433728 bytes)

(80) Multiprocessors, (64) CUDA Cores/MP:

5120 CUDA Cores

GPU Max Clock rate: 1530 MHz (1.53 GHz)

Memory Bus Width: 4096-bit

L2 Cache Size: 6291456 bytes

Output from *bandwidthTest*:

Device to Device Bandwidth, 1 Device(s)

Transfer Size (Bytes)	Bandwidth(GB/s)
32000000	713.5

GPUs for High Performance Computing (HPC)

SLING EURO

NVIDIA Tesla cards (flagship cards historically)

Model	No. of cores	Memory [GB]	Bandwidth [GB/s]	FP32 [TFlops]
Kepler K40	2280	12 (GDDR5)	240	4.3
Pascal P100	3584	16 (HBM2)	732	10.6
Volta V100	5120	32 (HBM2)	900	15.7
Ampere A100	6912	80 (HBM2)	2039	19.5

AMD Radeon Instinct cards

Model	No. of cores	Memory [GB]	Bandwidth [GB/s]	FP32 [TFlops]
Radeon MI8	4096	4 (HBM)	512	8.2
Radeon MI25	4096	16 (HBM2)	484	12.3
Radeon MI50	3840	16 (HBM2)	1024	13.4
Radeon MI60	4096	32 (HBM2)	1024	14.7

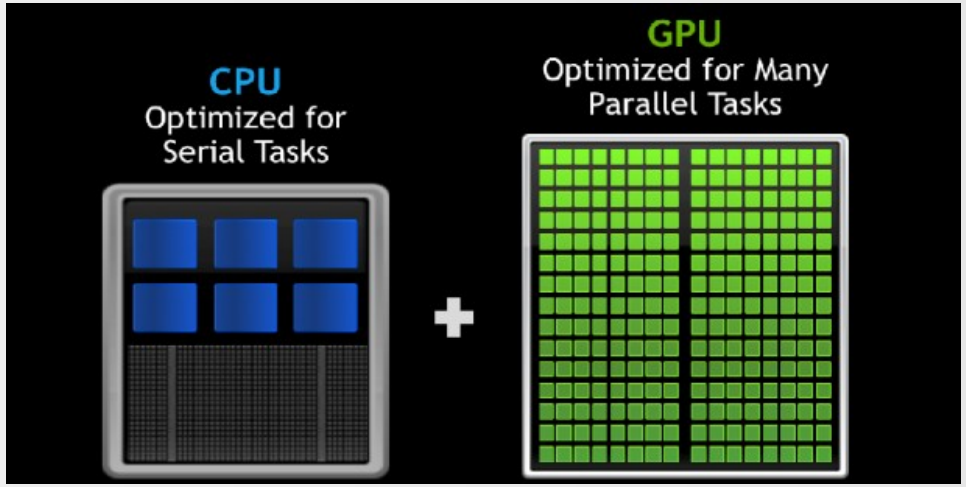
A consumer-grade GPU: just for comparison

Model	No. of cores	Memory [GB]	Bandwidth [GB/s]	FP32 [TFlops]
NVIDIA GeForce 930MX	384	2 (DDR3)	14.4	0.765

Exercise 1: Information and compute capabilities of a GPU

Login to your `viz.hpc.fs.uni-lj.si` account and complete the following tasks:

- find and load a suitable CUDA module (hint: use `module avail` and `module load`)
- find general info on the GPU available on the login node (hint: use `nvidia-smi`)
- find the diagnostic programs `deviceQuery` and `bandwidthTest` (hint: use `which nvcc` and navigate to the subdirectory `extras/demo_suite` of the main CUDA directory)
- execute the diagnostic programs to determine the main characteristics of the GPU (No. of SMs, No. of CUDA cores, global memory available, memory bandwidth)
- compile and run the OpenCL diagnostic program to determine the OpenCL compute capability of the GPU (hint: go to the `eurocc-accelerators/OpenCL_diagnostics` subdirectory of the cloned repository for this course and use the command `make`)



CPU vs. GPU (source: nvidia.com)

General Purpose GPU (GPGPU) programming:

- GPUs are used for **accelerating intensive computational tasks** rather than accelerating graphics tasks
- CPU and GPU are in principle **separate devices** with **separate memory space**
- **GPU is a co-processor to CPU:**
  - **CPU**: Optimized for **serial tasks** and **low-latency** access
  - **GPU**: Optimized for **many parallel tasks** and **throughput**

GPGPU languages

Many solutions exist for programming GPUs, the **two mostly used** are:

- **CUDA** (Compute Unified Device Architecture)
  - a set of extensions to higher level programming languages (C, C++ and Fortran) for
  - using GPU as a co-processor for heavy parallel tasks
  - provides a developer toolkit for compiling, debugging and profiling programs
  - **only supported by NVIDIA GPUs**
- **OpenCL** (Open Computing Language)
  - a standard open-source programming model initially developed by major manufacturers (Apple, Intel, ATI/AMD, NVIDIA), now maintained by Khronos
  - also provides extensions to C and C++ (**SYCL**) and a developer toolkit, more low-level than CUDA
  - **supported by many types of Processing Units** (CPUs, GPUs, FPGAs, MICs...):
  - de facto oriented to heterogeneous computing

Example 1: Hello world

Notebook with examples on Google Colaboratory:

https://colab.research.google.com/drive/1C6DAGC7fvfQWI-D5L7Cuc58-BjkFLGPk?usp=sharing

C for loop:

```
#define N 16
for(int i = 0; i < N; ++i){
    printf("Hello world! I'm
            Iteration %d\n", i);
}
```

- In a for loop every iteration of the code is run sequentially on a **CPU**
- the code will print messages in order from iteration 0 to 15

Example 1: Hello world (cont.)

CUDA kernel:

```
#define NUM_BLOCKS 16
#define BLOCK_SIZE 1
__global__ void hello(){
    int idx = blockIdx.x;
    printf("Hello world! I'm a
           thread in block
           %d\n", idx);
}
hello<<<NUM_BLOCKS, BLOCK_SIZE>>>();
```

From a for loop to a **CUDA** kernel:

- On a GPU the code in a kernel is run **in parallel** by independent **threads** organized in **blocks** (CUDA terminology)
- In CUDA a kernel is defined by the __global__ prefix: called by the CPU as a regular function by the **triple chevron** syntax <<<...>>>

Example 1: Hello world (cont.)

OpenCL kernel:

```
#define GLOBAl_SIZE 16
#define LOCAL_SIZE 1
__kernel void hello() {
        int gid = get_global_id(0);
        printf("Hello world! I'm a
                thread in
                block %d\n", gid);
}
size_t globalItemSize = GLOBAl_SIZE;
size_t localItemSize = LOCAL_SIZE;
cl_kernel kernel = clCreateKernel(program, "hello", &ret);
ret = clEnqueueNDRangeKernel(commandQueue,
         kernel, 1, NULL, &globalItemSize, &localItemSize,
         0, NULL, NULL);
```

From a for loop to an **OpenCL** kernel:

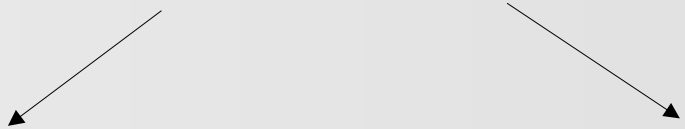
- On a GPU the code in a kernel is run **in parallel** by independent **work–items** organized in **work–groups** (OpenCL terminology)
- In OpenCL a kernel is defined by the __kernel prefix: called by the CPU with the clEnqueueNDRangeKernel() **function of the OpenCL API**

CUDA kernel launch

Triple chevron launch syntax <<< >>> contains

"kernel launch parameters"

```
hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();
```



defines the number of blocks to use = 16

defines the number of threads per block = 1

```
__global__ void hello(){
    int idx = blockIdx.x;
    printf("Hello world! I'm a thread in block %d\n", idx);
}
hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();
```

OpenCL kernel launch

This function of the OpenCL API contains

“kernel launch parameters”

clEnqueueNDRangeKernel(commandQueue, kernel,

1, NULL, &globalItemSize, &localItemSize, 0, NULL,

NULL);

defines the number of work–items times work–groups

1 x 16 = 16

defines the number of work–items

= 1

```
__kernel void hello() {
        int gid = get_global_id(0);
        printf("Hello world! I'm a thread in block %d\n", gid);
}
cl_kernel kernel = clCreateKernel(program, "hello", &ret);
ret = clEnqueueNDRangeKernel(commandQueue, kernel, 1,
        NULL, &globalItemSize, &localItemSize, 0, NULL, NULL);
```

Exercise 2: Hello World extended

1. **Modify the Hello World CUDA code from Example 1** in the following way:

- define 8 blocks with 2 threads each
- print the "Hello World" message to reflect also information on the thread number from each block (hint: use the built-in variable `threadIdx.x`)

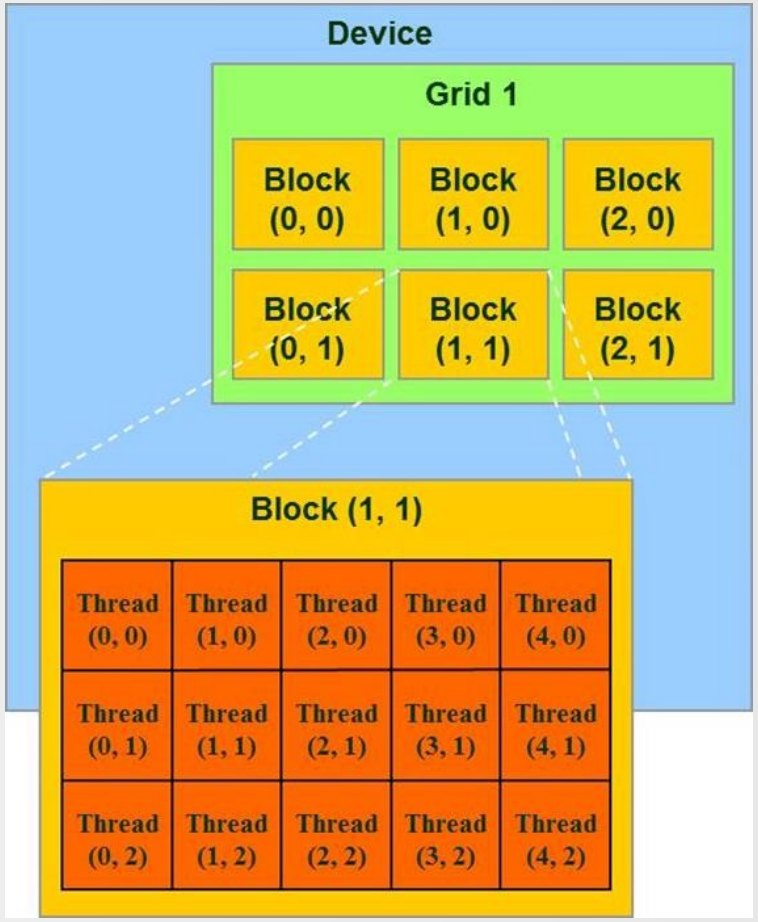
2. **Modify the Hello World OpenCL code from Example 1** in the following way:

- define 8 blocks (work-groups) with 2 threads (work-items) each
- print the "Hello World" message to reflect also information on the thread (work-item) number from each block (work-group) (hint: use the built-in variables `get_group_id(0)` for work-groups and `get_local_id(0)` for work-items)

The skeleton codes for this exercise can be found on Google Colab on this link:
https://colab.research.google.com/drive/1eqQoXbmL7kPjS3dwZGtxfYY3fmETTBrK?usp=sharing

Replace ??? in the code to complete the tasks listed above.

GPU CUDA threads hierarchy



Threads hierarchy (source: nvidia.com)

- **threads** are organized into blocks:

  blocks can be 1D, 2D, 3D
- **blocks** are organized into a grid:

  grids can also be 1D, 2D, 3D
- each block or thread has a unique ID:

  .x, .y, .z are components in every dimension

  `threadIdx`:

  thread coordinate inside the block

  `blockIdx`:

  block coordinate inside the grid

  `blockDim`:

  block dimension in thread units

  `gridDim`:

  grid dimension in block units

CUDA threads hierarchy examples

- **1D kernel:**

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

`idx` ... global thread index in one dimension

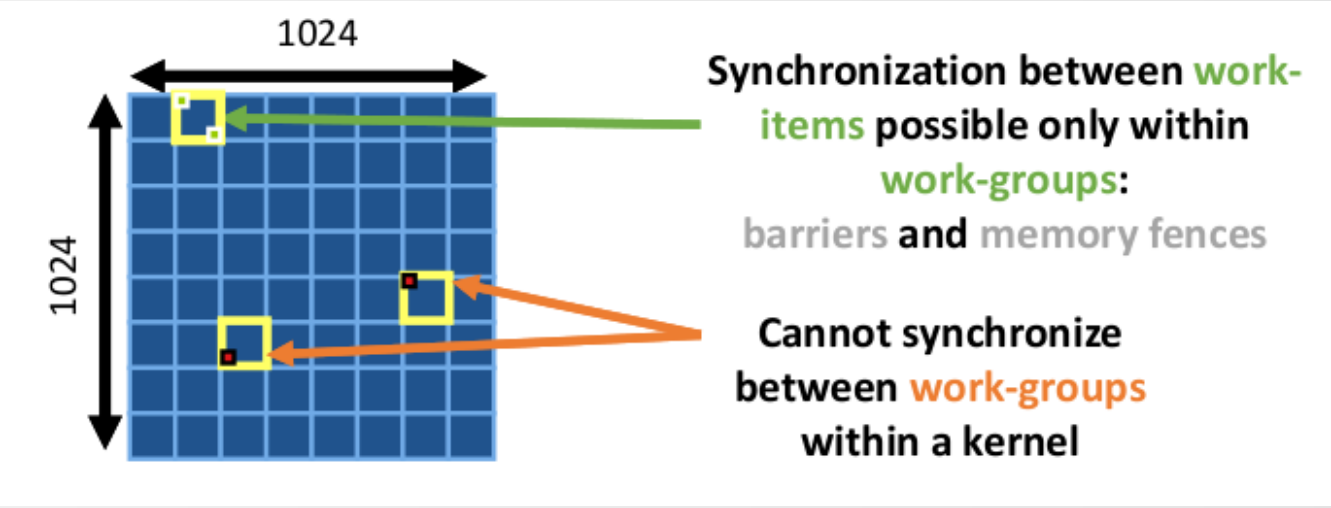
- **2D kernel:**

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

`i` ... global thread index in first dimension

`j` ... global thread index in second dimension

GPU OpenCL work-items hierarchy



work-items hierarchy (source: khronos.org)

- work-items are grouped into **work-groups**
- work-items within a work-group can **share local memory** and can **synchronize**
- the number of work-items can be specified in a work-group – this is called the **local (work-group) size**
- the OpenCL run-time can choose the **work-group size automatically** (usually not optimal)

OpenCL work-items hierarchy examples

- **1D kernel:**

```
int idx = get_global_id(0);
```

or alternatively:

```
int idx = get_group_id(0) * get_local_size(0) + get_local_id(0)
```

`idx` ... global work-item index in one dimension

- **2D kernel:**

```
int i = get_global_id(0);
int j = get_global_id(1);
```

or alternatively:

```
int i = get_group_id(0) * get_local_size(0) + get_local_id(0)
int j = get_group_id(1) * get_local_size(1) + get_local_id(1)
```

`i` ... global work-item index in first dimension

`j` ... global work-item index in second dimension

Example 2: Vector addition on CPU

Notebook with examples on Google Colaboratory:

https://colab.research.google.com/drive/1C6DAGC7fvfQWI-D5L7Cuc58-BjkFLGPk?usp=sharing

Vector addition is done in a **for** loop:

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

part of code for **parallelization** on GPU!

Example 2: Vector addition with CUDA

CUDA **kernel** for vector addition:

```
__global__ void vector_add(double *out, double *a, double *b, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < n)
        out[i] = a[i] + b[i];
}
```

vector components of a and b are **added in parallel**!

each thread `i` runs **in parallel**!

CUDA program flow

A typical flow of a CUDA program:

- **Allocate** GPU memory
- **Populate** GPU memory with inputs from the host
- **Execute** a GPU kernel on those inputs
- **Transfer** outputs from the GPU back to the host
- **Free** GPU memory

Recent Nvidia GPUs (Pascal microarchitecture or newer) support:

- unified memory invoked with `cudaMallocManaged()`
- single-pointer-to-data model, CPUs and GPUs use the same memory address space hence transfers from/to GPU memory no longer needed

CUDA step by step: 1. Initialize device

- CUDA **initialization** (optional):

```
CudaSetDevice(0);
```

- CUDA **initialization** through `CUDA_ERROR()` API call (optional):

```
CUDA_ERROR(cudaSetDevice(0));
```

- getting device (first available) **properties** through `cudaGetDeviceProperties()` (optional):

```
cudaDeviceProp prop;
CUDA_ERROR(cudaGetDeviceProperties(&prop,0));
printf("Found GPU '%s' with %g GB of global memory, max %d threads per
       block, and %d multiprocessors\n", prop.name,
       prop.totalGlobalMem/(1024.0*1024.0*1024.0),
       prop.maxThreadsPerBlock,prop.multiProcessorCount);
```

CUDA step by step: 2. Allocate GPU memory

- allocating **memory**:

```
cudaMalloc((void**)&d_a, sizeof(double) * N);
cudaMalloc((void**)&d_b, sizeof(double) * N);
cudaMalloc((void**)&d_out, sizeof(double) * N);
```

- naming **convention**(optional):

  “d” indicating device in d_a or a_d

CUDA step by step: 3. Transfer data from host to device memory

- copy **from host to device memory**:

```
cudaMemcpy(d_a, a, sizeof(double) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(double) * N, cudaMemcpyHostToDevice);
```

- host and device variables **must be of same size and type!**

CUDA step by step: 4. Execute kernel on device variables as inputs

- defining **kernel block size** and **threads per block size**:

```
int threadsPerBlock = 1024;
int blocksPerGrid = N/threadsPerBlock + (N % threadsPerBlock == 0 ? 0:1);
```

- or **alternatively**:

```
int threadsPerBlock = 1024;
int blocksPerGrid =(N + threadsPerBlock - 1) / threadsPerBlock;
```

- **executing kernel**:

```
vector_add<<<blocksPerGrid, threadsPerBlock>>>(d_out, d_a, d_b, N);
```

- **integers** and **constant type** variables can be passed to the kernel without device memory allocation

CUDA step by step: 5. Transfer data back from device to host

- copy **from device to host memory**:

```
cudaMemcpy(out, d_out, sizeof(double) * N, cudaMemcpyDeviceToHost);
```

- the counterpart host variable (e.g., `out`) must be of the **same size and type**!

CUDA step by step: 6. Deallocate (free) device memory

- **free** device memory:

```
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_out);
```

CUDA step by step: 7. Compiling the code

- CUDA codes reside in `*.cu` files
- `nvcc` compiler is used to compile the codes, e.g.:

  ```
  $ nvcc -o vector_add_cuda vector_add_cuda.cu
  ```

- execution of the codes in command line, e.g.:

  ```
  $ ./vector_add_cuda
  ```

- hardware design, number of cores, cache size, and supported arithmetic instructions are different for different versions of compute capability
- compiling the codes for different compute capabilities, e.g. for maximum compatibility with cards predating Volta microarchitecture:

```
$ nvcc -arch=sm_30 -gencode=arch=compute_20,code=sm_20 \
-gencode=arch=compute_30,code=sm_30 -gencode=arch=compute_50,code=sm_50 \
-gencode=arch=compute_52,code=sm_52 -gencode=arch=compute_60,code=sm_60 \
-gencode=arch=compute_61,code=sm_61 -gencode=arch=compute_61,code=compute_61 \
-o vector_add_cuda vector_add_cuda.cu
```

Nvidia Kepler cards compute capabilities

- **`SM30 or SM_30, compute_30`**

  Kepler architecture (e.g. generic Kepler, GeForce 700, GT-730).

  Adds support for unified memory programming

  Completely dropped from CUDA 11 onwards.

- **`SM35 or SM_35, compute_35`**

  Tesla K40.

  Adds support for dynamic parallelism.

  Deprecated from CUDA 11, will be dropped in future versions.

- **`SM37 or SM_37, compute_37`**

  Tesla K80.

  Adds a few more registers.

  Deprecated from CUDA 11, will be dropped in future versions.

  More info on other Nvidia cards compute capabilities:

  https://arnon.dk/matching-sm-architectures-arch-and-gencode-for-various-nvidia-cards/

Example 2: Vector addition with OpenCL

OpenCL **kernel** for vector addition:

```
__kernel void vector_add(__global double *a, __global double *b,
__global double *out, int n) {

    int i = get_global_id(0);
    if(i < n)
      out[i] = a[i] + b[i];
}
```

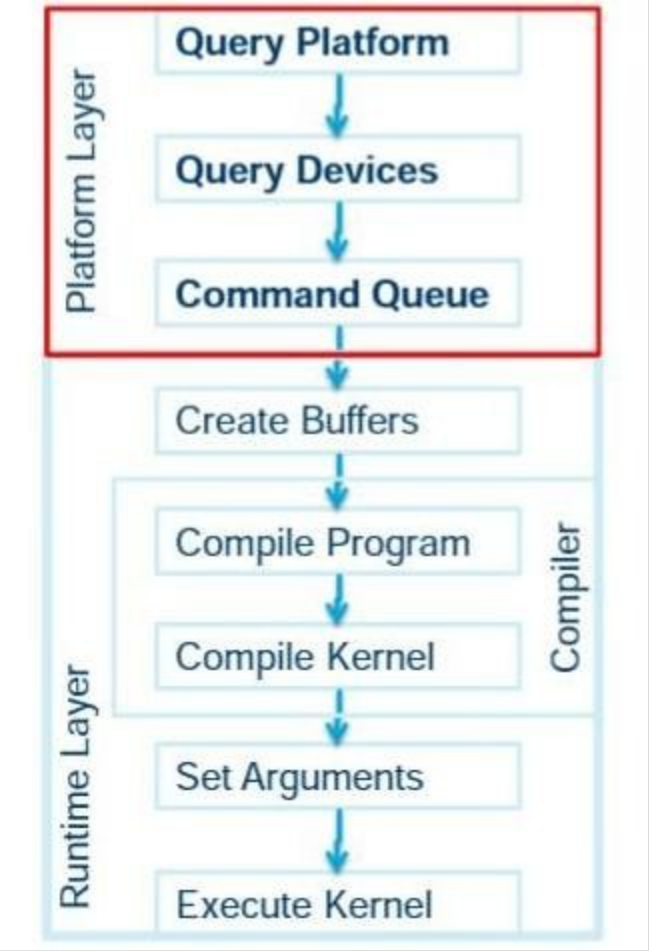
vector components of a and b are **added in parallel**!

each thread `i` runs **in parallel**!

OpenCL step by step: 1. Initialize device

- **declare** context
- **choose** a device from context
- **create** a command queue with device and context

```
cl_context context = clCreateContext(NULL, 1,
&device_id, NULL, NULL, &ret);


ret = clGetDeviceIDs(platform_id,
                     CL_DEVICE_TYPE_ALL,
                     1, &device_id, &ret_num_devices);


cl_command_queue command_queue = clCreateCommandQueue
                                 (context, device_id,
                                  0, &ret);
```

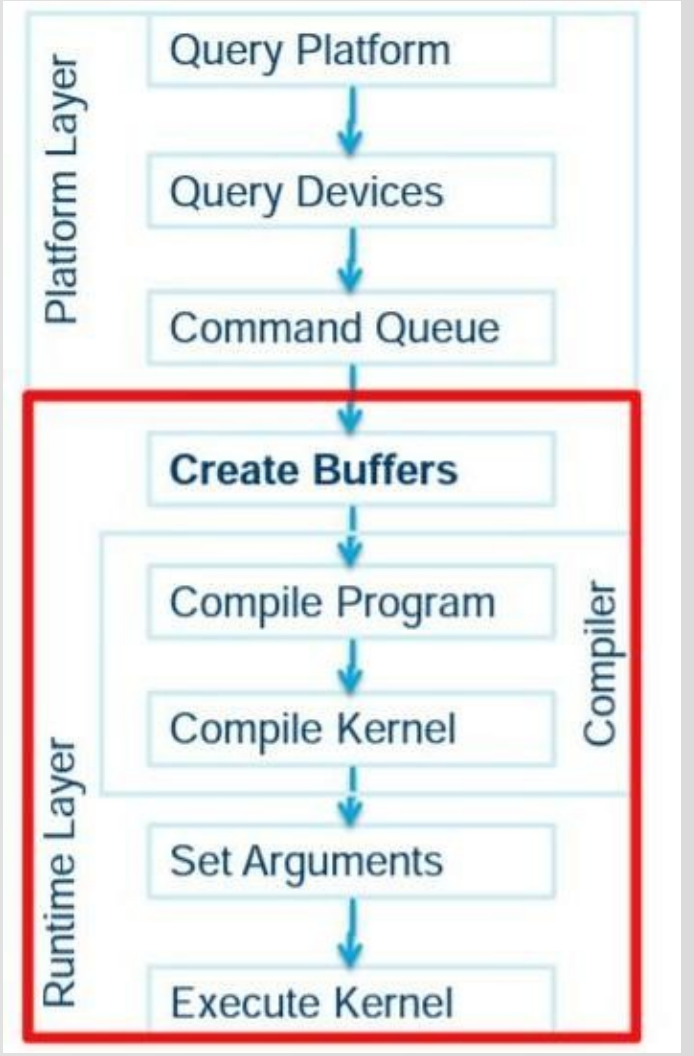


OpenCL program flow
(source: KU Leuven)

OpenCL step by step: 2. Create buffers

- **create** buffers on device
- **transfer** host data to device

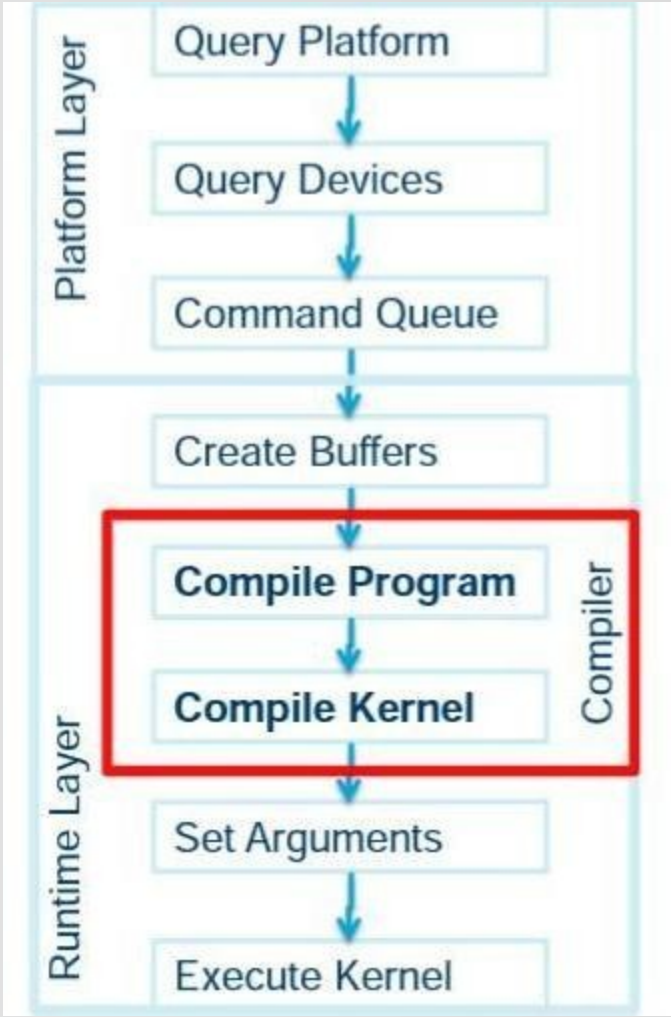
```
cl_mem a_mem_obj = clCreateBuffer(context,
                       CL_MEM_READ_ONLY,
                       N * sizeof(double), NULL,
                       &ret);

cl_mem out_mem_obj = clCreateBuffer(context,
                         CL_MEM_WRITE_ONLY,
                         N * sizeof(double), NULL,
                         &ret);

ret = clEnqueueWriteBuffer(command_queue, a_mem_obj,
                           CL_TRUE, 0,
                           N * sizeof(double), a, 0,
                           NULL, NULL);
```



OpenCL step by step: 3. Build program and select kernel

- **create** program
- **build** program
- **create** kernel

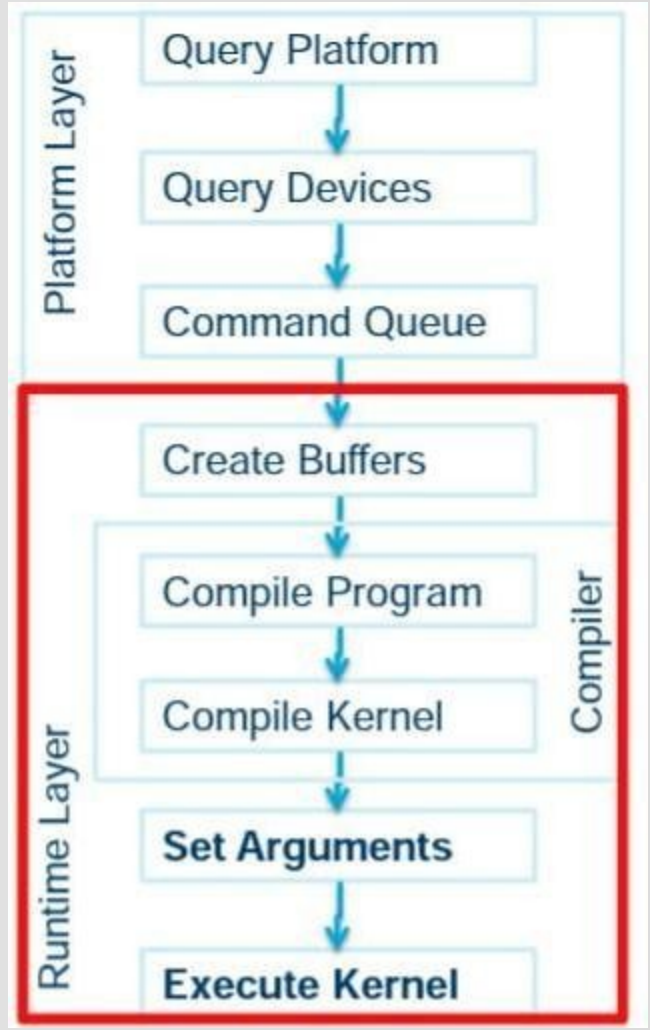
```
cl_program program = clCreateProgramWithSource
                     (context, 1,
                      (const char **)&source_str,
                      (const size_t *)&source_size,
                      &ret);

ret = clBuildProgram(program, 1, &device_id,
                     NULL, NULL, NULL);


cl_kernel kernel = clCreateKernel(program, "vector_add",
                                  &ret);
```



OpenCL step by step: 4. Set arguments and enqueue kernel

- **set** arguments
- **set** local and global work-group sizes
- **execute** kernel

```
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                     (void *)&a_mem_obj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem),
                     (void *)&b_mem_obj);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem),
                     (void *)&out_mem_obj);
ret = clSetKernelArg(kernel, 3, sizeof(cl_int),
                     (void *)&n);

size_t local_item_size = 64;
int n_blocks = n/local_item_size + (n % local_item_size
                                    == 0 ? 0:1);
size_t global_item_size = n_blocks * local_item_size;

ret = clEnqueueNDRangeKernel(command_queue, kernel, 1,
                             NULL, &global_item_size,
                             &local_item_size, 0, NULL,
                             NULL);
```



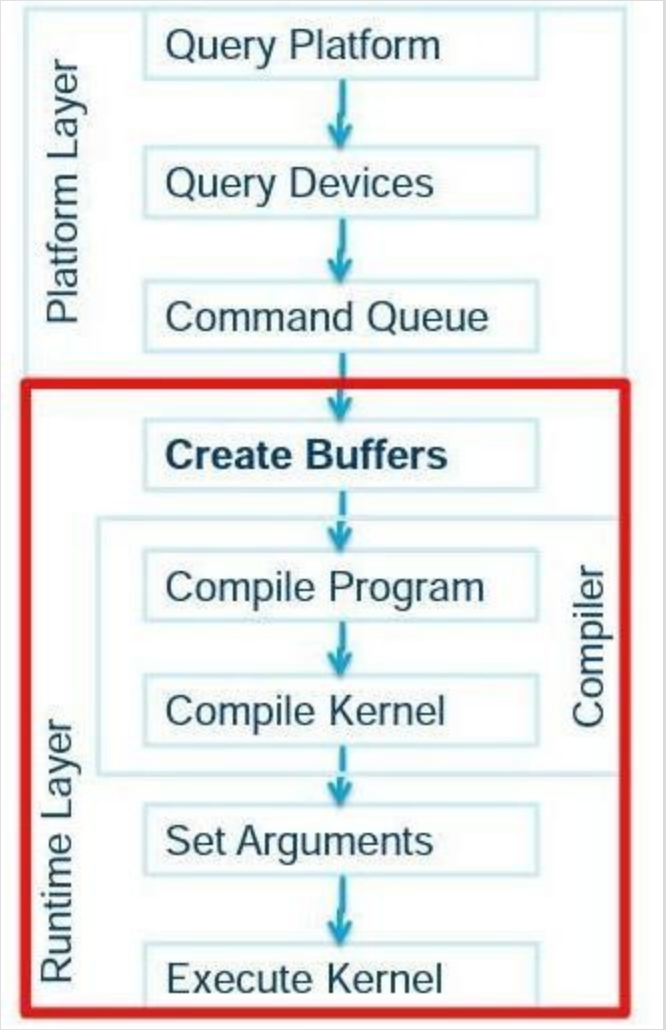
OpenCL step by step: 5. Transfer back result

- **transfer** of results if needed on the host:

  avoid unnecessary transfers from/to host!

- **data from one kernel** can be used by another kernel

```
ret = clEnqueueReadBuffer(command_queue, out_mem_obj,
                          CL_TRUE, 0,
                          N * sizeof(double), out, 0,
                          NULL, NULL);
```



OpenCL step by step: 6. Compiling the code

- OpenCL codes reside in `*.c` files and `*.cl` files (kernels)
- `gcc` (or `nvcc`) compiler is used to compile the codes, e.g.:

```
$ nvcc -o vector_add_opencl vector_add_opencl.c -lOpenCL
```

- execution of the codes in command line, e.g.:

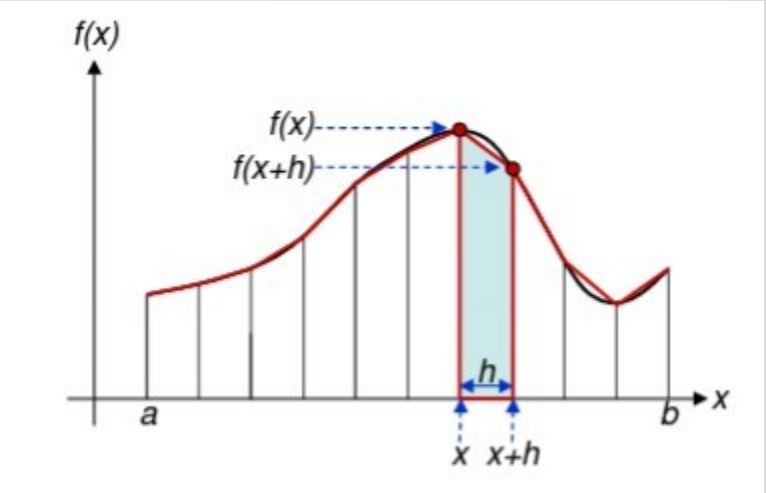
```
$ ./vector_add_opencl
```

- OpenCL drivers available in CUDA (most cards support OpenCL 1.2)

Example 3: Numerical integration (Riemann sum with trapezoids)

Approximation of the integral of a function using the **trapezoid rule**:

- divide area under the function from a to b into N trapezoids
- area of trapezoid: **median of the trapezoid** (f(x+h)+f(x))/2 multiplied with **sub-interval width** (b-a)/N
- **sum** of the **trapezoid areas**: **approximation** of the **definite integral** from a to b
- **for simplicity**: a = 0 and b = 1
- numerical evaluation of the **normal distribution function** from 0 to 1: **0.341345**



$$\Phi(1) = \frac{1}{\sqrt{2\pi}} \int_0^1 e^{-x^2/2}\, dx$$

CPU code: Calculation of the Riemann sum

```
double riemann(int n)
{
  double sum = 0;
  for(int i = 0; i < n; ++i)
  {
    double x = (double) i / (double) n;
    double fx = (exp(-x * x / 2.0) +
                 exp(-(x + 1 / (double)n) *
                 (x + 1 / (double)n) / 2.0)) / 2.0;
    sum += fx;
  }
  sum *= (1.0 / sqrt(2.0 * M_PI)) / (double) n;
  return sum;
}
```

All the computation is done in a **for loop**:

- trapezoid medians and trapezoid sums
- **non-optimized** CPU code:

  riemann_cpu_double.c
- Execution time for N = 1 billion:

  about **90 s** (about **40 s** with -O3 optimization level)

GPU code: Riemann sum with one CUDA kernel

```
__global__ void medianTrapezoid(double *a, int n)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  double x = (double)idx / (double)n;

  if(idx < n)
    a[idx] = (exp(-x * x / 2.0) + exp(-(x + 1 /
             (double)n) * (x + 1 / (double)n) / 2.0))
             / 2.0;
}
```

- trapezoid medians are calculated on device (GPU) with the CUDA **“medianTrapezoid” kernel**
- GPU code: *riemann_cuda_double.cu*
- an array of trapezoid medians is returned to host (CPU)
- the **trapezoid sums** are **calculated on host**
- a **speed up of 10x** for N = 1 billion

GPU code: Riemann sum with one OpenCL kernel

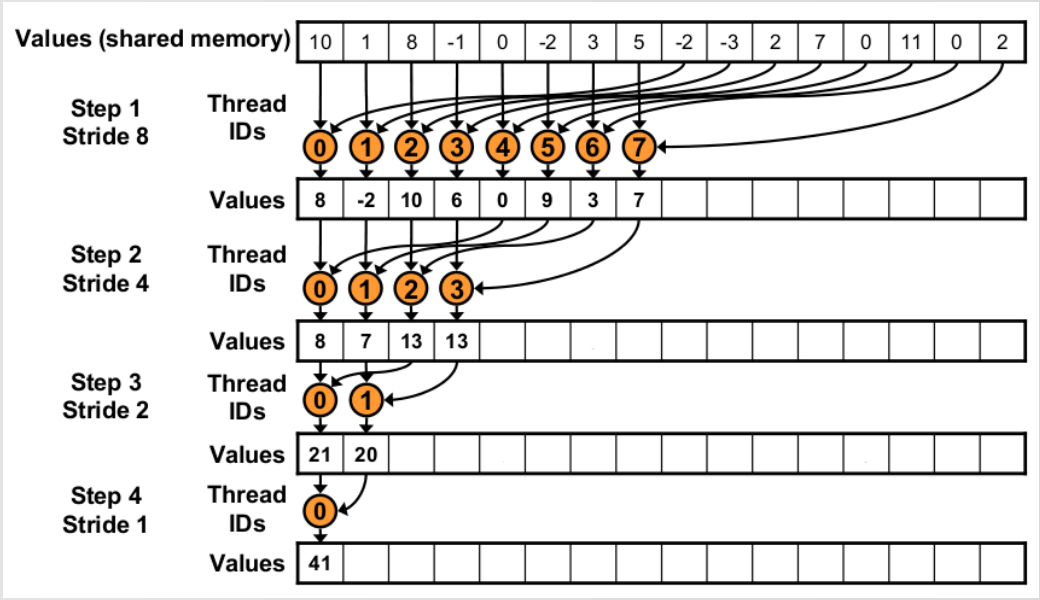
```
__kernel void medianTrapezoid(__global double *a, int n)
{
    int idx = get_global_id(0);
    double x = (double)idx / (double)n;

    if(idx < n)
       a[idx] = (exp(-x * x / 2.0) +
                 exp(-(x + 1 / (double)n) * (x + 1 /
                 (double)n) / 2.0)) / 2.0;
}
```

- trapezoid medians are calculated on device (GPU) with the OpenCL **“medianTrapezoid” kernel**
- GPU code: *riemann_opencl_double.c*
- an array of trapezoid medians is returned to host (CPU)
- the trapezoid sums are **calculated on host**
- also a **speed up of 10x** for N = 1 billion

Numerical integration: reduction of trapezoid sums

Calculation of trapezoid sums is done with **sum reduction**:

- the array of calculated trapezoid medians is used by **another kernel** for **sum reduction**
- a kernel with **one block of multiple threads** is used
- sum reduction of partial sums is done in **shared memory** which is **faster than global memory**



Sum reduction (source: nvidia.com)

GPU code: Riemann sum with two CUDA kernels

```
__global__ void reducerSum(double *a, double *out,
int n, int block_size) {
    int idx = threadIdx.x;
    double sum = 0;
    for (int i = idx; i < n; i += block_size)
        sum += a[i];
    extern __shared__ double r[];
    r[idx] = sum;
    __syncthreads();
    for (int size = block_size/2; size>0; size/=2)
{
        if (idx<size)
            r[idx] += r[idx+size];
        __syncthreads();
    }
    if (idx == 0)
        *out = r[0];
}
```

- an additional CUDA **kernel “reducerSum”** for **calculating the trapezoid sums**
- GPU code: *riemann_cuda_double_reduce.cu*
- a **speed up of 8x** against the GPU code with one kernel and of **80x** against the non-optimized CPU code for N = 1 billion

GPU code: Riemann sum with two OpenCL kernels

```
__kernel void reducerSum(__global double *a, __global
double *out, __local double *r, int n, int block_size)
{
    int idx = get_local_id(0);
    double sum = 0;
    for (int i = idx; i < n; i += block_size)
        sum += a[i];
    r[idx] = sum;
    barrier(CLK_LOCAL_MEM_FENCE);

    for (int size = block_size/2; size>0; size/=2) {
        if (idx<size)
            r[idx] += r[idx+size];
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if (idx == 0)
        *out = r[0];
}
```

- an additional OpenCL **kernel “reducerSum”** for **calculating the trapezoid sums**
- GPU code: *riemann_opencl_double_reduce.cu*
- also a **speed up of 8x** against the GPU code with one kernel and of **80x** against the non-optimized CPU code for N = 1 billion

Exercise 3: Block size performance analysis

1. **Analyze the performance of the `riemann_cuda_double_reduce.cu` code by varying the block size** (32, 64, 128, 192, 256, 512 and 1024) **of the kernel `medianTrapezoid`** and setting the block size of the kernel `reducerSum` to 1024:

- Use the prepared shell script for the analysis.
- For which block size the code performs the best?

2. **Analyze the performance** for the best performing block size of the kernel `medianTrapezoid` from the previous analysis and **by varying the block size** (32, 64, 128, 192, 256, 512 and 1024) **of the kernel `reducerSum`**:

- Replace ??? in the second shell script to complete the analysis due to the requirements.
- For which block size the code performs the best?
- Are there any anomalous results for a chosen block size? Can you identify the reason? Also run the anomalous case separately with `nvprof` and `cuda-memcheck` (replace ??? with the appropriate block size values) to check for any errors.

The codes for this exercise can be found on Google Colab on this link:
https://colab.research.google.com/drive/1lxW-QaIg66_BTMoGWA0vtERfJouYkbJS?usp=sharing

What about Python?

Python wrappers of CUDA and OpenCL exist although not officially suported:

- **pyCUDA**:

  ```
  $ pip install pycuda
  ```

- **pyOpenCL**:

  ```
  $ pip install pyopencl
  ```

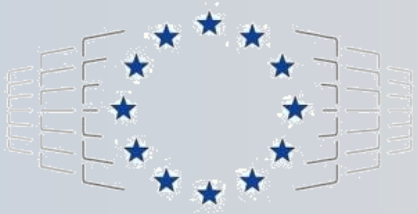
- both use **numpy** for array and data manipulation
- PyOpenCL is somewhat easier to use than OpenCL (no low-level programming needed)

Riemann sum codes in pyCUDA (`riemann_cuda_double.py`) and pyOpenCL (`riemann_opencl_double.py`) available in jupyter notebook on Google Colab – for running the scripts prior installation of libraries is needed:

- `!pip -q install pycuda`
- `!pip -q install opencl`

SLING EURO

Thanks!



EuroHPC
Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro