CUDA programming

PRACE Summer of HPC 2022 - Day 4

Leon Bogdanović

University of Ljubljana, FME, LECAD lab

CUDA history and current state

https://docs.nvidia.com/cuda/

History:

- developed by NVIDIA (2007) – first solution for GPU programming
- latest stable version: 11.7.0 (2022)

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**

EXAMPLE 1

CUDA Hello world on GPU

Hello world example

Notebook:

`02_Hello_World_C.ipynb`

C for loop:

```
#define N 4
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 3

Notebook **on Google Colaboratory**:

https://colab.research.google.com/drive/1VVqwQS6ui1BlKQiSuq2TTy7pEAR2x2DB

Hello world example (cont.)

Notebook:

`03_Hello_World_CUDA.ipynb`

CUDA kernel:

```
#define NUM_BLOCKS 4
#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 <<<...>>>

Notebook **on Google Colaboratory**:

https://colab.research.google.com/drive/1oyh0XGep61-NJha7vei3022wS3Imm-fC

CUDA kernel launch

Triple chevron launch syntax <<< >>> contains

"kernel launch parameters"

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

defines the number of blocks to use = 4

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>>>();
```

Exercise 2: Hello World extended

Notebook:

`05_Hello_World_GPU_extended.ipynb`

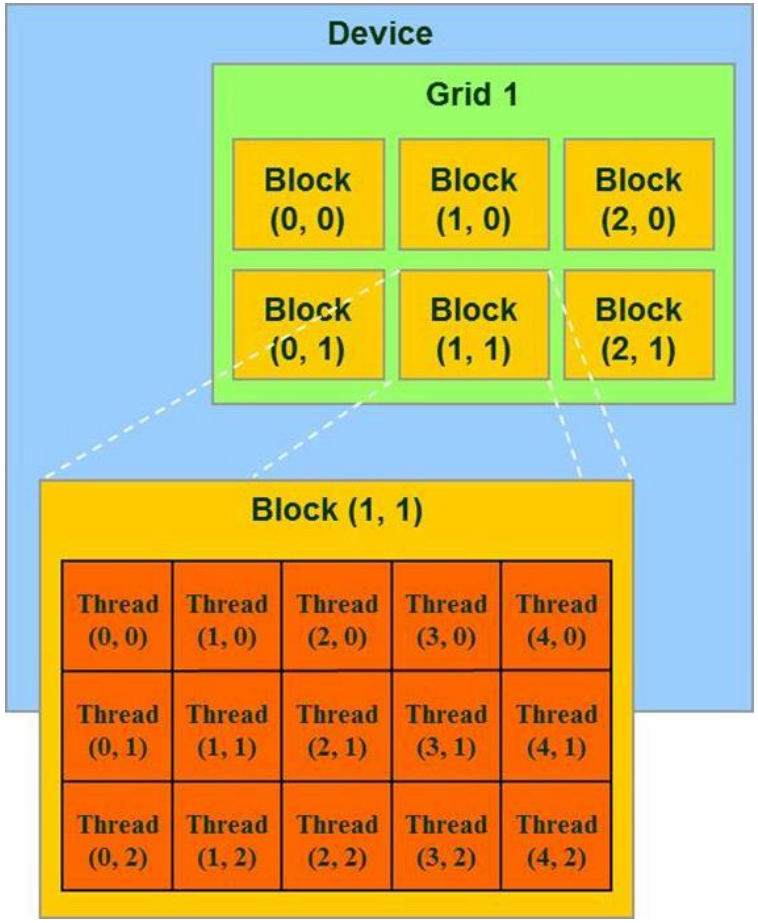
Modify the Hello World CUDA code from Example 1 in the following way:

- 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`)

Notebook **on Google Colaboratory**:

https://colab.research.google.com/drive/1Zy6BA-yBo2DJrSgMmkaybe75AuVot3TC

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:

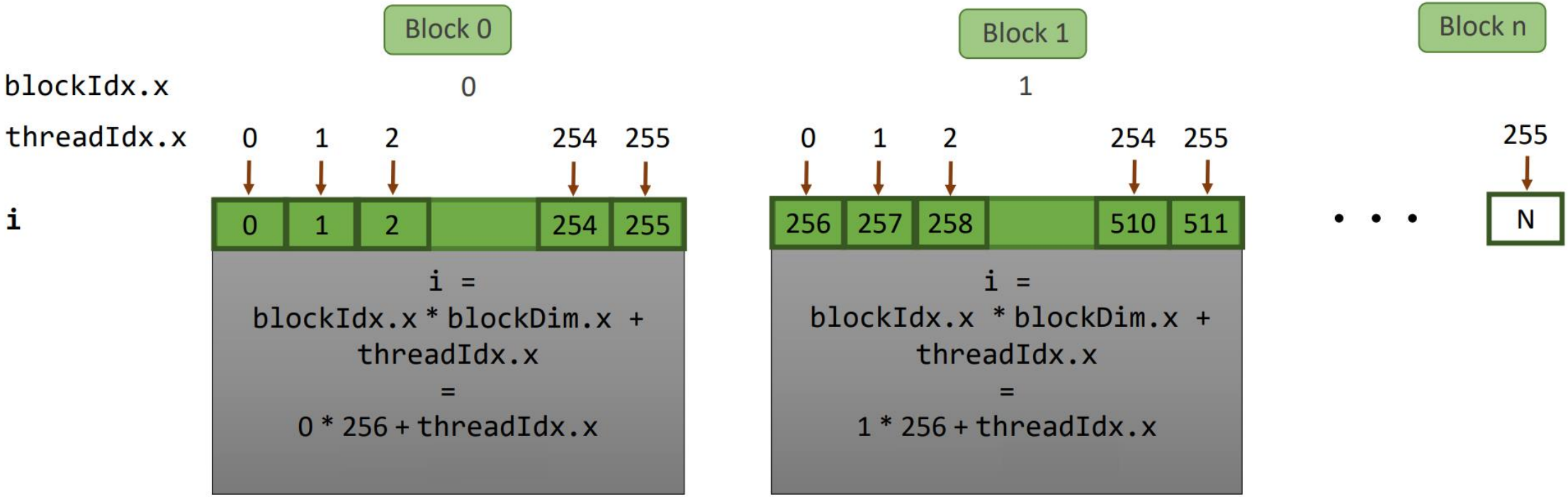
  grid dimension in block units

CUDA threads hierarchy: examples

1D kernel:

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

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



CUDA threads hierarchy: examples (cont.)

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

EXAMPLE 2

Vector addition on GPU

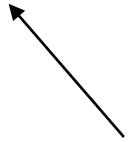
Vector addition CPU code

Notebook:

`06_Vector_addition_GPU.ipynb`

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!

Notebook **on Google Colaboratory**:

https://colab.research.google.com/drive/1vx1FVQ3xLgAAkndJD7mK5sebh1tjNsyj

Vector addition CUDA code

Notebook:

`06_Vector_addition_GPU.ipynb`

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**!

Notebook **on Google Colaboratory**:

https://colab.research.google.com/drive/18G29WIWRwaPgT_cg-rorHM1joPusiSml

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()` wrapper (optional):

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

```
static void CUDA_ERROR(cudaError_t err)
{
    if (err != cudaSuccess) {
        printf("CUDA ERROR: %s,
                exiting\n",
                cudaGetErrorString(err));
        exit(-1);
    }
}
```

- 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 memory

- 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
```

EXAMPLE 3

Numerical integration (Riemann sum)

Numerical integration: Riemann sum with trapezoids

Approximation of the definite 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

Riemann sum with trapezium rule:

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=0}^{N-1} \frac{f(\frac{i}{N}+h)+f(\frac{i}{N})}{2} \cdot h$$

where:

$$h = \frac{b-a}{N}$$

For $a = 0$ and $b = 1$:

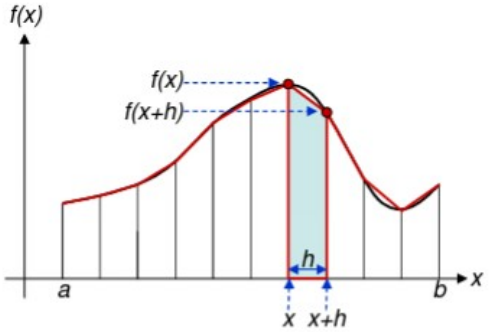
$$\int_a^b f(x)\mathrm{d}x \approx \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(\frac{i+1}{N})+f(\frac{i}{N})}{2}$$

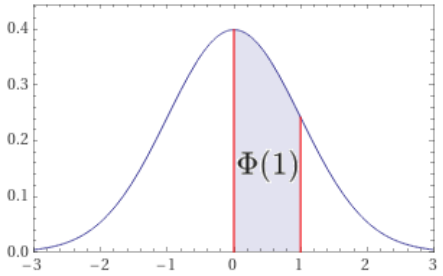
Standard normal distribution:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}\,\mathrm{d}x$$

Normal distribution function $\Phi(1)$:

$$\Phi(1) = \frac{1}{\sqrt{2\pi}} \int_0^1 e^{-x^2/2}\,\mathrm{d}x \approx 0.3413447460685429$$





CPU code: Calculation of the Riemann sum

Notebook:

07_Riemann_sum_C.ipynb

```
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
- execution time for N = 100 millions: about **2.8 s** (about **2.1 s** with -O3 optimization level)

Notebook **on Google Colaboratory**:

https://colab.research.google.com/drive/1LBqOcLlPN1zAczCh5uaA1C2Xa_gLoy0x

GPU code: Riemann sum with one CUDA kernel

Notebook:

`08_Riemann_sum_CUDA_one_kernel.ipynb`

```
__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**
- an array of trapezoid medians is returned to host (CPU)
- the **trapezoid sums** are **calculated on host**
- execution time for N = 100 millions: about 0.55 s (a **speed up of 4x** compared to the CPU code)

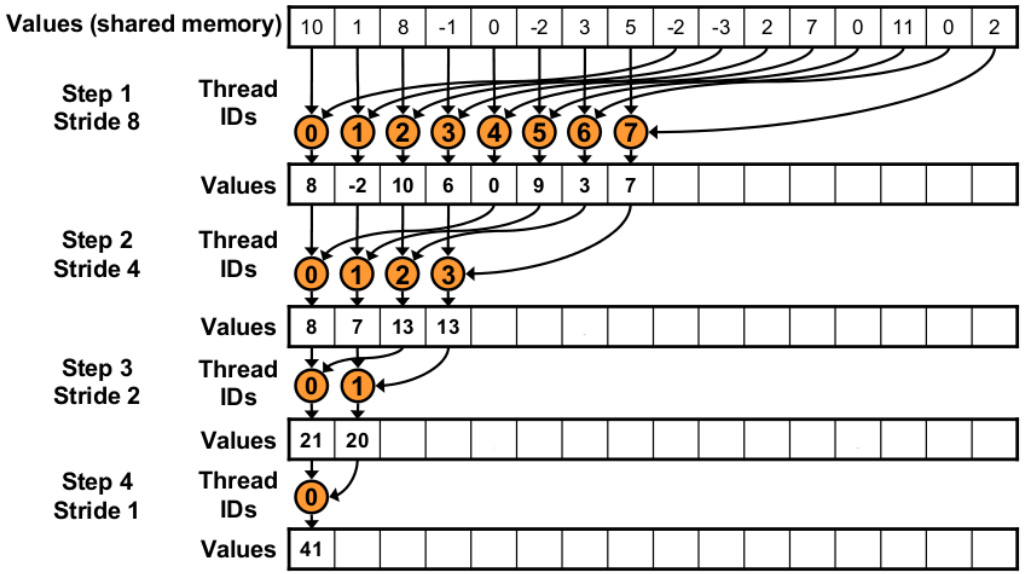
Notebook **on Google Colaboratory**:

https://colab.research.google.com/drive/17XdBdkOWpiToW-Hcdb8MjMdtf7FHnBNR

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**
- type of parallel reduction: **sequential addressing**, with an array of N elements done in $\log_2 N$ steps



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];
}
```

Notebook:

`10_Riemann_sum_CUDA_two_kernels.ipynb`

- an additional CUDA **kernel "reducerSum"** for **calculating the trapezoid sums**
- execution time for N = 100 millions: about 0.1 s (a **speed up of 21x** against the CPU code)

Notebook **on Google Colaboratory**:

https://colab.research.google.com/drive/199tA6Df9xMbI7YgdIu0IPpnuA1CV-Nhi

What about Fortran?

Notebook:

`14_CUDA_Fortran_examples.ipynb`

- **Fortran wrapper of CUDA** is available in **NVIDIA HPC SDK**
- CUDA Fortran codes reside in `*.cuf` files
- `nvfortran` compiler is used to compile the codes, e.g.:

```
nvfortran -r8 -o riemann_cuda_fortran riemann_cuda.cuf
```

- on Google Colaboratory NVIDIA HPC SDK is not available by default

What about Python?

Notebook:

`15_Riemann_sum_GPU_Python.ipynb`

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

- **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)
- for running scripts on Google Colaboratory prior installation of libraries is needed:

```
!pip -q install pycuda
!pip -q install pyopencl
```

Notebook **on Google Colaboratory**:

https://colab.research.google.com/drive/1bBC9O2qLTGbaQ0uSwpv0QRCfrmhx-nbi