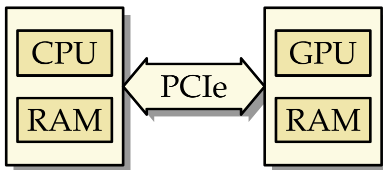
High Performance Computing I
WS 2019/2020
Session #28
Introduction to CUDA

Andreas F. Borchert and Michael C. Lehn
Ulm University

February 3, 2020

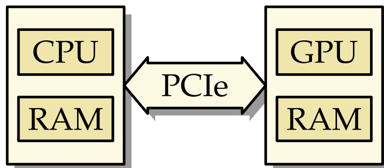
- Graphics accelerators were developed quite early to unburden the CPU.
- In March 2001, Nvidia introduced the GeForce 3 Series which supported programmable shading.
- Radeon R300 from ATI followed in August 2002 which extended the capabilities of the GeForce 3 series by adding mathematical functions and loops.
- In the following years GPUs developed step by step into the direction of so-called GPGPUs (*general purpose GPUs*).
- Multiple languages and interfaces were developed to support accelerating devices: OpenCL (Open Computing Language), DirectCompute (by Microsoft), CUDA (Compute Unified Device Architecture by Nvidia) and OpenACC (*open accelerators*, an open standard supported by Cray, CAPS, Nvidia, and PGI.

Relation between CPU and GPU



- Usually, GPUs are available as PCI Express cards.
- They operate separately from the CPU and its memory. Communications is done using the PCI Express bus.
- Most GPUs have very specialized processor architectures on base of SIMD (*single instruction multiple data*). One prominent exception are the Larrabee micro architecture and its successors by Intel which supports the x86 instruction set and which can no longer be considered as a GPU. Their only purpose is high performance computing.

Relation between CPU and GPU



- An application consists usually of two programs, one for the host (i.e. the CPU) and one for the accelerating device (i.e. the GPU).
- The program for the device is uploaded from the host to the device. Afterwards both communicate using the PCIe connection.
- The communication is done using a library which supports synchronous and asynchronous operations.

Programming models for GPUs

- High-level languages are supported for GPUs, typically variants that are specialized extensions of C and C++.
- The extensions are required to access all features of the GPU. On the other hand, support for C and C++ is not necessarily complete, in particular most libraries are not available on the GPU with the exception of some selected mathematical functions.
- The CUDA programming model allows program texts for the host and the device to be freely mixed, i.e. the same code can be compiled for both platforms. During compilation, the compiler separates the codes required for each platform and compiles to the corresponding architecture.
- In case of OpenCL programm texts for the host and the device have to be stored in different files.

CUDA is a package free of charge (but not open source) that is available by Nvidia for selected variants of Linux, MacOS, and Windows with following components:

- a device driver that supports the upload of GPU programs and the communication between host and device code,
- a language extension of C++ (CUDA C++) that allows to have a unified source for host and device,
- a compiler named *nvcc* that supports CUDA C++ (on base of C++14 in case of CUDA-10.1), and
- a runtime library, and
- more libraries (including the support of BLAS and FFT).

URL: `https://developer.nvidia.com/cuda-downloads`

A first contact with the GPU

properties.cu

```
#include <cstdlib>
#include <iostream>

inline void check_cuda_error(const char* cudaop, const char* source,
       unsigned int line, cudaError_t error) {
   if (error != cudaSuccess) {
      std::cerr << cudaop << " at " << source << ":" << line
         << " failed: " << cudaGetErrorString(error) << std::endl;
      exit(1);
   }
}

#define CHECK_CUDA(opname, ...) \
   check_cuda_error(#opname, __FILE__, __LINE__, opname(__VA_ARGS__))

int main() {
   int device; CHECK_CUDA(cudaGetDevice, &device);
   // ...
}
```

- All CUDA applications can freely access the CUDA runtime library. Error codes of all CUDA runtime library invocations shall be checked.

properties.cu

```
int device_count; CHECK_CUDA(cudaGetDeviceCount, &device_count);
struct cudaDeviceProp device_prop;
CHECK_CUDA(cudaGetDeviceProperties, &device_prop, device);
if (device_count > 1) {
   std::cout << "device " << device << " selected out of "
      << device_count << " devices:" << std::endl;
} else {
   std::cout << "one device present:" << std::endl;
}
std::cout << "name: " << device_prop.name << std::endl;
std::cout << "compute capability: " << device_prop.major
   << "." << device_prop.minor << std::endl;
// ...
```

- *cudaGetDeviceProperties* fills a voluminous struct with the properties of one of the available GPUs.
- There exist manifold Nvidia graphic cards with different micro architectures and configurations.
- CUDA program sources have the suffix ".cu".

```
livingstone$ make
nvcc -o properties properties.cu
livingstone$ ./properties
one device present:
name: Quadro P620
compute capability: 6.1
total global memory: 2096103424
total constant memory: 65536
total shared memory per block: 49152
registers per block: 65536
L2 Cache Size: 524288
warp size: 32
mem pitch: 2147483647
max threads per block: 1024
max threads dim: 1024 1024 64
max grid dim: 2147483647 65535 65535
multi processor count: 4
kernel exec timeout enabled: no
device overlap: yes
integrated: no
can map host memory: yes
unified addressing: yes
livingstone$
```

```
livingstone$ make
nvcc -o properties properties.cu
```

- We compile using the *nvcc* utility.
- The code which is required on the GPU is compiled for the corresponding architecture and stored as a byte-array in the host program, ready to be uploaded to the device. All other parts are forwarded to *gcc*.
- Options like "–gpu-architecture compute_60" allow to specify the architecture and enable the support of corresponding features.
- Options like "–code sm_60" ask to generate the code during compilation time, not later at runtime.

simple.cu

```
__global__ void add(std::size_t len, double alpha, double* x, double* y) {
   for (std::size_t i = 0; i < len; ++i) {
      y[i] += alpha * x[i];
   }
}
```

- Functions (or methods) that are to be run on the device but to be invoked from the host are called *kernel functions*. They are designated by using the CUDA keyword **__global__**.
- In this example, the function *add* computes $\vec{y} \leftarrow \vec{x} + \alpha\vec{y}$.
- All pointers refer to device memory.

A first program for the GPU

simple.cu

```
/* execute kernel function on GPU */
add<<<1, 1>>>(N, 2.0, device_a, device_b);
```

- Kernel functions can be invoked anywhere in code that is run on the host.
- Between the name of the function and the parameter list a kernel configuration has be given that is enclosed in “<<<...>>>”. This is strictly required and the parameters will be explained in the following.
- Elementary data types can be transmitted as usual (in this case N and the value 2.0). Pointers, however, must point to device memory. That means that the data of vectors and matrices has to be transfered first from host to device.

Copying between host and device

simple.cu

```
double a[N]; double b[N];
for (std::size_t i = 0; i < N; ++i) {
   a[i] = i; b[i] = i * i;
}

/* transfer vectors to GPU memory */
double* device_a;
CHECK_CUDA(cudaMalloc, (void**)&device_a, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_a, a, N * sizeof(double),
   cudaMemcpyHostToDevice);
double* device_b;
CHECK_CUDA(cudaMalloc, (void**)&device_b, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_b, b, N * sizeof(double),
   cudaMemcpyHostToDevice);
```

- The CUDA runtime library function *cudaMalloc* allows to allocate device memory.
- Fortunately, all elementary data types are represented in the same way on host and device memory. Hence, we can simply copy data from host to device without needing to convert anything.

Copying between host and device

simple.cu

```
double a[N]; double b[N];
for (std::size_t i = 0; i < N; ++i) {
   a[i] = i; b[i] = i * i;
}

/* transfer vectors to GPU memory */
double* device_a;
CHECK_CUDA(cudaMalloc, (void**)&device_a, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_a, a, N * sizeof(double),
   cudaMemcpyHostToDevice);
double* device_b;
CHECK_CUDA(cudaMalloc, (void**)&device_b, N * sizeof(double));
CHECK_CUDA(cudaMemcpy, device_b, b, N * sizeof(double),
   cudaMemcpyHostToDevice);
```

- *cudaMemcpy* supports copying in both directions. The parameters specify first the target address, then the source address, then the number of bytes, and finally the direction.
- In dependence of the direction, the addresses are interpreted to be addresses of host or device memory.

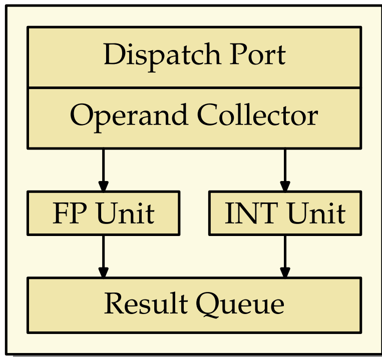
simple.cu

```
/* transfer result vector from GPU device to host memory */
CHECK_CUDA(cudaMemcpy, b, device_b, N * sizeof(double),
   cudaMemcpyDeviceToHost);
/* free space allocated at GPU memory */
CHECK_CUDA(cudaFree, device_a); CHECK_CUDA(cudaFree, device_b);
```

- We transfer the result from device to host after invoking the kernel function.
- Kernel functions run asynchronously, i.e. the host continues computing when the kernel function runs on the device. However, by default requests for the device (like copying or kernel function invocations) are serialized. Thereby, the call of *cudaMemcpy* that copies the result back from device to the host waits implicitly until the kernel function is finished.
- If the kernel function couldn't be started or failed for some reason, the associated error code is delivered with the next serialized CUDA operation.
- Finally, the allocated device memory ought to be released.

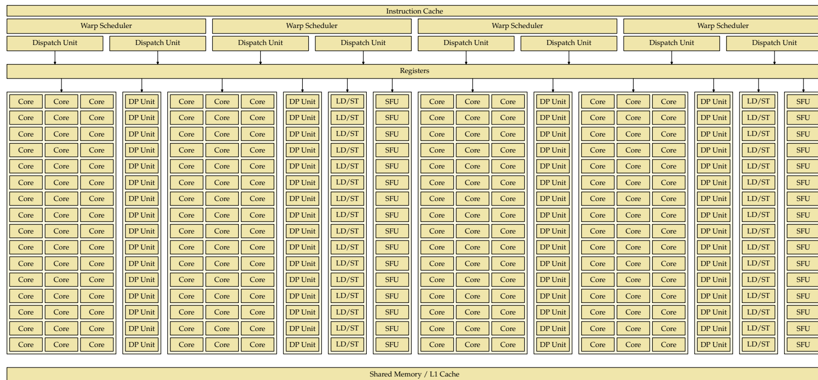
Simple GPU application pattern

- The first example shows the typical order of execution of many applications:
  - Allocate device memory
  - Transfer input data from host to device
  - Invoke kernel function
  - Copy results from device to host
  - Release device memory
- We had no real parallelization in this example as the host waited for the device and the device employed one single core only.



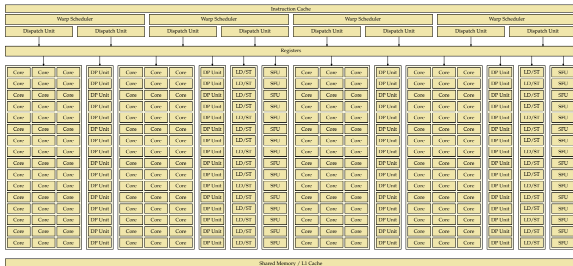
- The stripped-down cores of a GPU consist mainly of two units, one supporting arithmetic operations for floating point numbers, the other for integers.
- GPU cores don't do much beyond that. Instructions and data are delivered (*Dispatch Port*, *Operand Collector*), and the result is passed to the *Result Queue*.

Multiprocessors of a GPU



- This diagram shows a single multiprocessor of the Kepler microarchitecture with 192 cores operating in single precision ("Core") and 64 cores operating in double precision ("DP Unit").

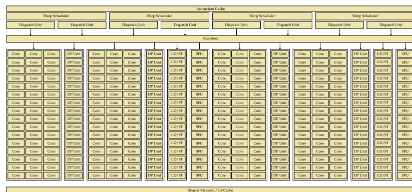
Multiprocessors of a GPU



- The GPU at the Livingstone has a Pascal microarchitecture where each multiprocessor provides 128 cores.
- In addition to the single and double precision cores, a specialized units are provided: load/store units that copy data between L1 and registers (“LD/ST”) and special function units for mathematical functions (“SFU”).
- (The diagram depicts the older Kepler architecture.)

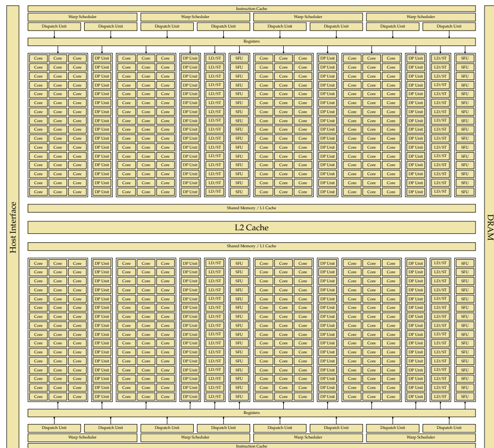
- The individual cores do not operate independently from each other.
- Usually, 32 cores are unified to a so-called warp. (Some architectures support also so-called half warps with 16 cores.)
- All cores of a warp execute synchronously the same instruction at the same time but get different data (SIMD architecture: array processor).
- At the beginning of each instruction, the input data are fetched in parallel by "LD/ST" units and stored into registers which can be used by the cores.

Multiprocessors of a GPU



- Each warp scheduler takes care of a warp (32 threads).
- The Kepler microarchitecture provides each warp scheduler with two dispatch units where each of them is able to launch an instruction for a warp on a group of cores or special units (in dependence of the instruction).
- This means that two instructions can be executed in parallel for a thread.
- Double precision instructions must be executed twice, once for each half warp (16 threads) as each warp scheduler has only 16 double precision units at its disposal.
- Each Kepler multiprocessor is equipped with four warp schedulers.

GPU with multiple multiprocessors



- Each GPU consists of multiple multiprocessors.
- The Nvidia Quadro P620 GPU of Livingstone provides four multiprocessors with 512 single and 16 double precisions cores in total.

- The instruction set of the Nvidia GPUs is proprietary and has not been published yet – just a summary is available.
- The utility *cuobjdump* allows to disassemble code.
- The instructions have a length of 32 or 64 bits, 64-bit instructions must be on 8-byte-boundaries.
- Arithmetic operations can have up to three source operands and a target where the result is to be stored. For example, the instruction *FMAD* computes $d = a * b + c$ in single precision.

How are conditional jumps implemented? Remember, all cores of a warp executes always the same instruction. What is to be done if we hit an if statement where different cores would like to continue differently?

- We have two stacks:
- One stack provides masks consisting of 32 bit which tell which core of a warp has to execute the instruction. (Others will simply ignore it.)
- The other stack maintains target addresses (of conditional jumps).
- In case of conditional jumps, every core sets its bit in the mask to signal if the following instructions are to be executed or not. The joined mask is then pushed onto the corresponding stack.
- The target address of the conditional jump is pushed onto the second stack.
- When the target address is reached, the top-most elements of both stacks are removed.

- A block is an abstraction that joins multiple virtual warps. (At least one physical warp is required, we can have more virtual warps as long we have sufficient registers to store the current state of all virtual warps, allowing us to assign the physical warps to varying virtual warps).
- A block is always connected with a multiprocessor. The *Warp Scheduler* of a warp decides which virtual warp is to be run at which time on which physical warp.
- All threads that belong to a block have access to shared memory.
- Threads within a block are free to synchronize using barriers and to communicate using the shared memory of the multiprocessor.
- Typically, one block supports up to 32 warps (or 1024 threads).

vector.cu

```
__global__ void add(double alpha, double* x, double* y) {
   std::size_t tid = threadIdx.x;
   y[tid] += alpha * x[tid];
}
```

- The **for**-loop is gone. Instead each core executes just one addition.
- Using *threadIdx.x* we learn who we are, i.e. which thread within a block (beginning at 0).
- The kernel function will then be executed for each element of the vector. And this will be parallelized to the extent possible by one multiprocessor.

Parallelization of the vector addition

vector.cu

```
/* execute kernel function on GPU */
add<<<1, N>>>(2.0, device_a, device_b);
```

- The configuration of the kernel function has been adapted.
- $<<<1, N>>>$ configures one block with N threads.
- Hence, the individual threads operate with *threadIdx.x* values in the range from 0 to $N-1$.
- N must not exceed the maximal number of threads per block.

Compilation process: Generation of PTX

vector.ptx

```
        ld.param.f64    %fd1, [_Z3adddPdS__param_0];
        ld.param.u64    %rd1, [_Z3adddPdS__param_1];
        ld.param.u64    %rd2, [_Z3adddPdS__param_2];
        cvta.to.global.u64      %rd3, %rd2;
        cvta.to.global.u64      %rd4, %rd1;
        mov.u32         %r1, %tid.x;
        mul.wide.u32    %rd5, %r1, 8;
        add.s64         %rd6, %rd4, %rd5;
        ld.global.f64   %fd2, [%rd6];
        add.s64         %rd7, %rd3, %rd5;
        ld.global.f64   %fd3, [%rd7];
        fma.rn.f64      %fd4, %fd2, %fd1, %fd3;
        st.global.f64   [%rd7], %fd4;
        ret;
```

- PTX is an acronym for *parallel thread execution* and stards for an assembly language of a virtual GPU processor. This is the target language of the *nvcc* for the device code.

- The PTX instruction set is public:
  `http://docs.nvidia.com/cuda/pdf/ptx_isa_6.1.pdf`
- The virtual machine of PTX has been developed to have an intermediate architecture-independent language which can be efficiently translated to a particular architecture.
- PTX can be generated using following command:
  `nvcc -o vector.ptx --ptx vector.cu`

Compilation process: Generation of CUBIN

vector.disas

```
/*0008*/                     MOV R1, c[0x0][0x20] ;
/*0010*/                     S2R R0, SR_TID.X ;
/*0018*/                     SHL R4, R0.reuse, 0x3 ;
/*0028*/                     SHR.U32 R0, R0, 0x1d ;
/*0030*/                     IADD R6.CC, R4, c[0x0][0x148] ;
/*0038*/                     IADD.X R7, R0, c[0x0][0x14c] ;
/*0048*/          {          IADD R4.CC, R4, c[0x0][0x150] ;
/*0050*/                     LDG.E.64 R2, [R6]          }
/*0058*/                     IADD.X R5, R0, c[0x0][0x154] ;
/*0068*/                     LDG.E.64 R8, [R4] ;
/*0070*/                     DFMA R2, R2, c[0x0][0x140], R8 ;
/*0078*/                     STG.E.64 [R4], R2 ;
/*0088*/                     EXIT ;
```

- *ptxas* translates PTX for a particular architecture into machine code (CUBIN):
  `ptxas --output-file vector.cubin --gpu-name sm_30 vector.ptx`

Compilation process: Generation of CUBIN

- Nvidia provides *cuobjdump* which allows to disassemble generated CUBIN code:
  `cuobjdump --dump-sass vector.cubin >vector.disas`
- The instruction summary of the Pascal architecture is to be considered for the GPU on Livingstone.

Working with multiple blocks

bigvector.cu

```
unsigned int max_threads_per_block() {
   int device; CHECK_CUDA(cudaGetDevice, &device);
   struct cudaDeviceProp device_prop;
   CHECK_CUDA(cudaGetDeviceProperties, &device_prop, device);
   return device_prop.maxThreadsPerBlock;
}
```

- We need to spread our task across multiple blocks if the number of elements exceeds the maximal number of threads per block.

bigvector.cu

```
/* execute kernel function on GPU */
size_t max_threads = hpc::cuda::get_max_threads_per_block();
if (N <= max_threads) {
   add<<<1, N>>>(2.0, device_a, device_b);
} else {
   add<<<(N+max_threads-1)/max_threads, max_threads>>>(2.0,
      device_a, device_b);
}
```

bigvector.cu

```
__global__ void add(double alpha, double* x, double* y) {
   std::size_t tid = threadIdx.x + blockIdx.x * blockDim.x;
   if (tid < N) {
      y[tid] += alpha * x[tid];
   }
}
```

- As *threadIdx.x* just identifies the thread within a block, it no longer works as a global index within the range of 0 to $N - 1$.
- *blockIdx.x* gives the index of the current block and *blockDim.x* the number of threads per block.
- Hence, we will have idling threads if *max_threads* is not a divisor of N. An **if**-statement is necessary to avoid out-of-bound accesses of the arrays.

Configuration of a kernel

In the most general form, a kernel configuration accepts four parameters where the last two are optional:
$<<< Dg, Db, Ns, S >>>$

- *Dg* specifies the dimensions of the grid (in one, two, or three dimensions).
- *Db* specifies the dimensions of a block (in one, two, or three dimensions).
- *Ns* specifies the size of the shared memory per block (by default 0 which causes the compiler to allocate shared memory automatically).
- *S* allows to associate the kernel with a stream (by default none).
- Both, *Dg* and *Db* are of type *dim3* whose constructor accepts up to three whole numbers.
- The limits of the current device are to be taken into consideration. If the parameters exceed the limits, the kernel function will not start.

The kernel and device functions can access the following variables that allow to identify the own thread and to retrieve the dimensions of the blocks and the grid:

threadIdx.x	x index within the own block
threadIdx.y	y index within the own block
threadIdx.z	z index within the own block
blockDim.x	first dimension of the block
blockDim.y	second dimension of the block
blockDim.z	third dimension of the block
blockIdx.x	x index of the own block within the grid
blockIdx.y	y index of the own block within the grid
blockIdx.z	z index of the own block within the grid
gridDim.x	first dimension of the grid
gridDim.y	second dimension of the grid
gridDim.z	third dimension of the grid

simpson.cu

```
// to be integrated function
__device__ double f(double x) {
   return 4 / (1 + x*x);
}

// numerical integration according to the Simpson rule
// for f over the i-th subinterval of [a,b]
__global__ void simpson(double a, double b, double* sums) {
   const std::size_t N = blockDim.x * gridDim.x;
   const std::size_t i = threadIdx.x + blockIdx.x * blockDim.x;
   double xleft = a + (b - a) / N * i;
   double xright = xleft + (b - a) / N;
   double xmid = (xleft + xright) / 2;
   sums[i] = (xright - xleft) / 6 * (f(xleft) + 4 * f(xmid) + f(xright));
}
```

- The kernel function applies Simpson's rule for one subinterval. Both, the blocks and the grid have one dimension only. Hence, *blockDim.x* ∗ *gridDim.x* gives the total number of subintervals.
- Functions that can be invoked on the device (but not as kernel function) can be designated using the keyword **__device__**.

Simpson's rule on the GPU

simpson.cu

```
double* device_sums;
CHECK_CUDA(cudaMalloc, (void**)&device_sums, N * sizeof(double));
simpson<<<nof_blocks, blocksize>>>(a, b, device_sums);

double sums[N];
CHECK_CUDA(cudaMemcpy, sums, device_sums,
   N * sizeof(double), cudaMemcpyDeviceToHost);
CHECK_CUDA(cudaFree, device_sums);
double sum = 0;
for (std::size_t i = 0; i < N; ++i) {
   sum += sums[i];
}
```

- Aggregation has to be done manually – there are no aggregation operators.

simpson2.cu

```
constexpr std::size_t THREADS_PER_BLOCK = 1024; // must be a power of 2

// numerical integration according to the Simpson rule
// for f over the i-th subinterval of [a,b]
__global__ void simpson(double a, double b, double* sums) {
   /* compute approximative sum for our sub-interval */
   const std::size_t N = blockDim.x * gridDim.x;
   const std::size_t i = threadIdx.x + blockIdx.x * blockDim.x;
   // ...
   double sum = (xright - xleft) / 6 * (f(xleft) +
      4 * f(xmid) + f(xright));

   /* store it into the per-block shared array of sums */
   std::size_t me = threadIdx.x;
   __shared__ double sums_per_block[THREADS_PER_BLOCK];
   sums_per_block[me] = sum;

   // ...
}
```

- Synchronization and shared memory are supported within a block.
- This opens the option of a block-wise aggregation.

simpson2.cu

```
/* store it into the per-block shared array of sums */
unsigned int me = threadIdx.x;
__shared__ double sums_per_block[THREADS_PER_BLOCK];
sums_per_block[me] = sum;
```

- Variables designated with **__shared__** are shared among all threads of a block.
- Access of shared memory is much faster than access of global memory.
- Usually the capacity is very limited. The GPU on Livingstone provides 48 KiB per block.

Block-wise aggregation

simpson2.cu

```
/* aggregate sums within a block */
std::size_t index = blockDim.x / 2;
while (index) {
   __syncthreads();
   if (me < index) {
      sums_per_block[me] += sums_per_block[me + index];
   }
   index /= 2;
}
/* publish result */
if (me == 0) {
   sums[blockIdx.x] = sums_per_block[0];
}
```

- Individual threads of a block are grouped to warps that are run synchronously (SIMD architecture). However, threads of the same block that belong to different warps are not necessarily executed synchronously.
- *__syncthreads* is a barrier function that suspends the thread (along with its warp) until all threads of the same block reached the barrier by invoking this function.