GPU Programming with CUDA

(Compute Unified Device Architecture)

Why?

How many Flops does a 200$ consumer GPU perform?

More than the number of people on earth?

More than the number of trees on earth?

More than the number of stars in Milky Way?

More than the number of days since the Big Bang?

More than the debt of Germany in Euro?

Why?

Consumer GPU GeForce RTX 3060 $\approx 10^{13}$ Flops (10 Teraflops FP32)

Number of people on earth: $\approx 8 \times 10^{9}$

Number of trees on earth: $\approx 3 \times 10^{12}$

Number of stars in Milky Way: $\approx 2\text{-}3 \times 10^{11}$

Number of days since the Big Bang: $\approx 5 \times 10^{12}$

Debt of Germany in Euro: $\approx 3 \times 10^{12}$

High end GPU Blackwell B200
$\approx 5 \times 10^{15}$ Flops FP16, tensor cores

Why?

Hight of a stack of paper with 10^{13} pages?

Higher than Mount Everest?

Higher than the altitude of a Starlink satellite?

Higher than the altitude of a geostationary satellite?

To the moon?

Why?

Thickness of paper: ≈ 0.1mm = 10^{-4} m

Height of a stack with **10^{13}** pages:

$10^{13} \times 10^{-4}$ m = 10^{9} m = **1 000 000 km** ≈ 24-fold earth circumference

Height of Mount Everest: 8 848 m

Altitude of Starlink satellite: 550 km

Altitude of geostationary satellite: 36 000 km

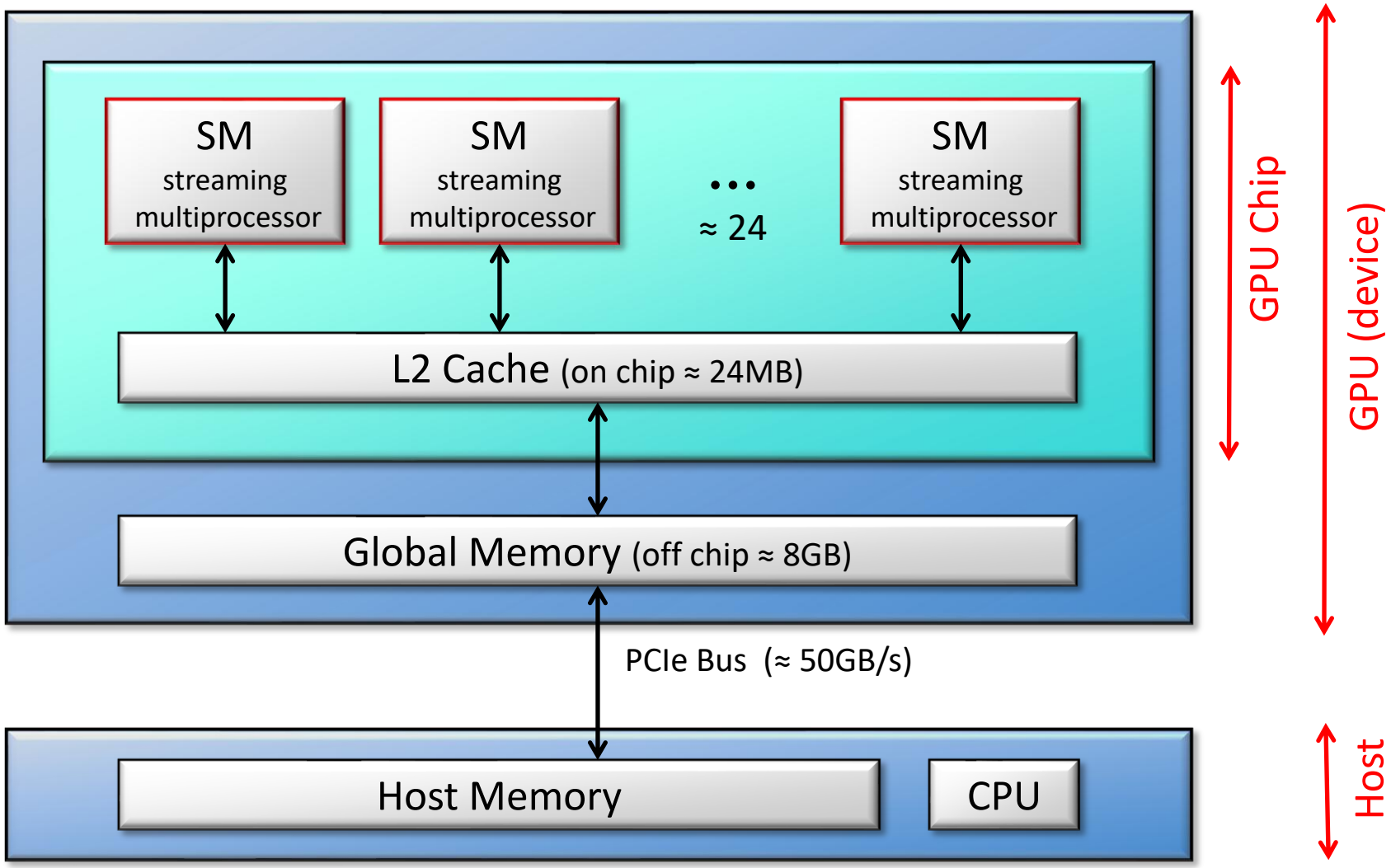
Distance to moon: 384 400 km

GPU's are really fast.

Outline

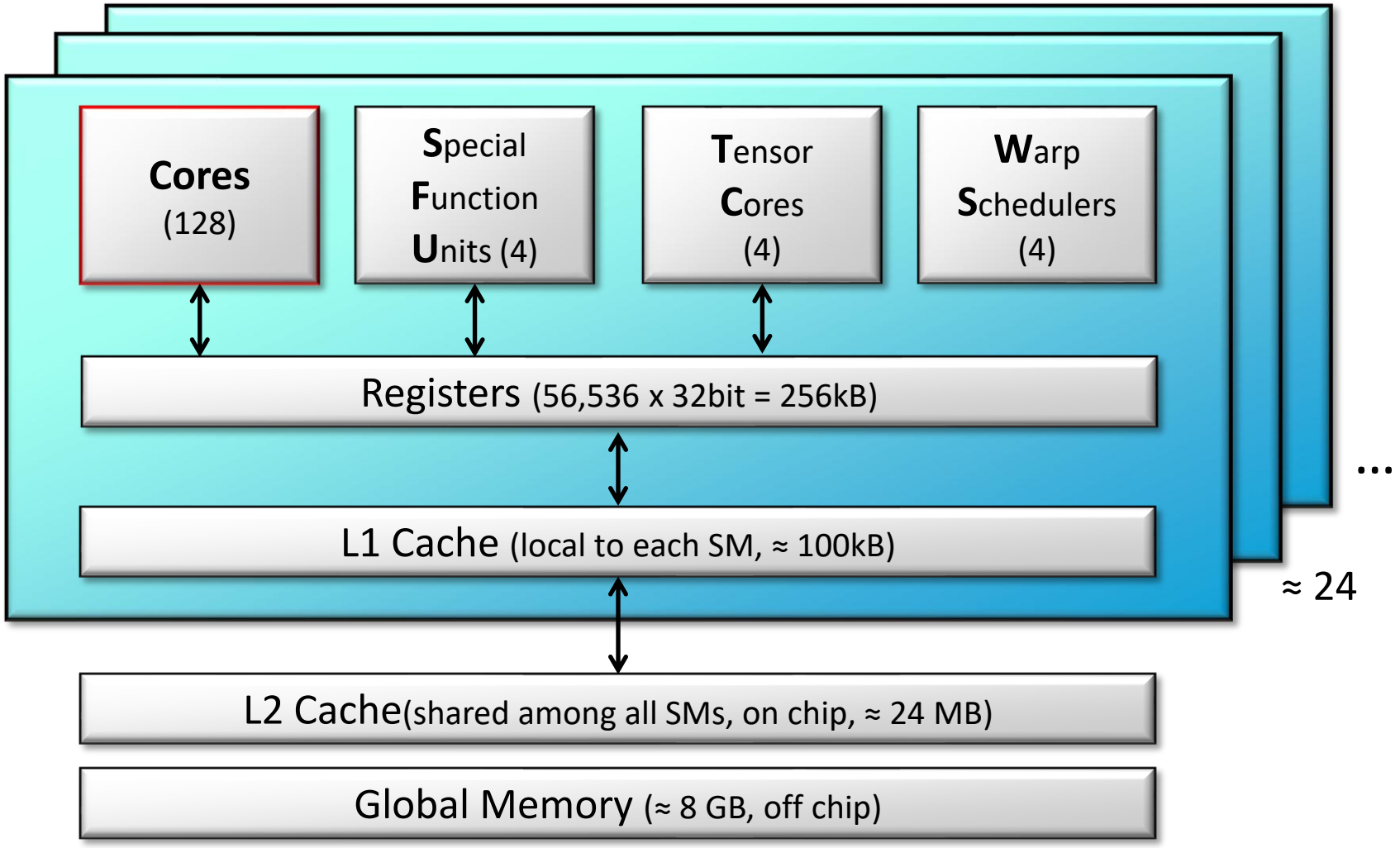
- Hardware architecture of Nvidia GPUs
- Threads
- Simple example of a CUDA program running on a GPU
- Google Colab: Experimenting with CUDA using a GPU
- Data transfer between host and GPU
- Debugging
- Time measurement with events
- Constant memory
- Parallel instruction streams
- Shared memory, thread communication

Simplified Architecture of a GPU (Graphics Processing Unit)



Values of the Nvidia Geforce RTX 4060 GPU

Simplified Architecture of a SM (Streaming Multiprocessor)



Values of the Nvidia Geforce RTX 4060 GPU

Architecture of a CUDA Core

32 bit floating point ALU

multiplier and adder
one multiplication and addition per cycle (pipelining)

32 bit integer ALU

64 bit floating point operations supported but very slow

Peak FP32 performance of Nvidia Geforce RTX 4060

24 SMs, 128 cores per SM → 3072 cores
2 Flops per cycle, boost clock frequency 2.46 GHz

⇨ **15 Tflops (15×10^{12} Flops)**

Memory Latency and Throughput

	Latency (Cycles)	Throughput (TB/s)
Registers	1	
L1 Cache	≈ 20-40	≈ 3-5
L2 Cache	≈ 200-300	≈ 1-2
Global Memory	≈ 400-1000	≈ 0.2 - 0.5

⇨ Memory latency is often the limiting factor for performance!

Latency Hiding

- Make sure that enough threads are ready to run.
- While one thread waits for memory, others can use the SMs.
- Thread rescheduling costs nothing!
  (Hardware registers remain allocated while threads are waiting)

Threads

Warp → *Scheduler*

- Group of 32 threads, handled by the warp schedulers.
  (4 warp schedulers for 128 cores on each SM.)

Thread Block → *SM*

- Group of threads.
  The threads of a block always run **on one and the** same SM.
  Communication within a block via shared memory of the SM.
- Threads of a warp always belong to the same block.

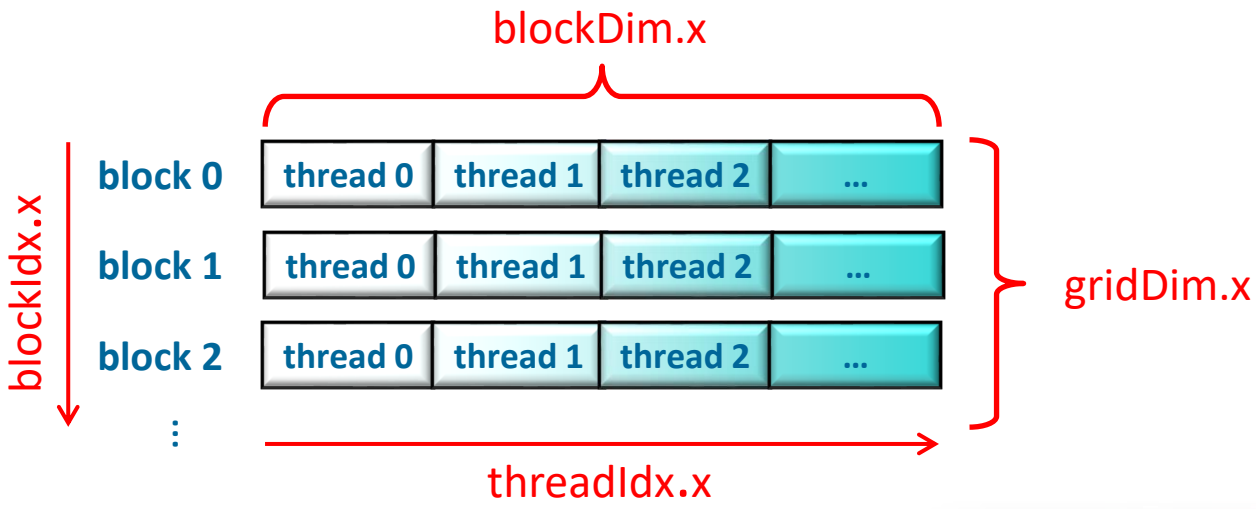
Grid → *GPU*

- Group of thread blocks.
  Each block in a grid has the same blocksize.

Kernel → *Compiler*

- C-Function which is executed by threads. Runs on the GPU.
- Declared with __global__

Threads



Local variables in each kernel:

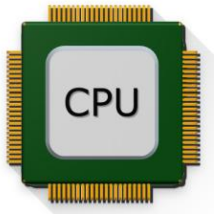
blockDim.x	Number of threads in a block
gridDim.x	Number of blocks in the grid
threadIdx.x	ID of the thread within its block.
blockIdx.x	ID of its block within its grid.

Total number of threads: blockDim.x * gridDim.x

Unique ID of the thread: blockDim.x * blockIdx.x + threadIdx.x

Example: Startup of Threads

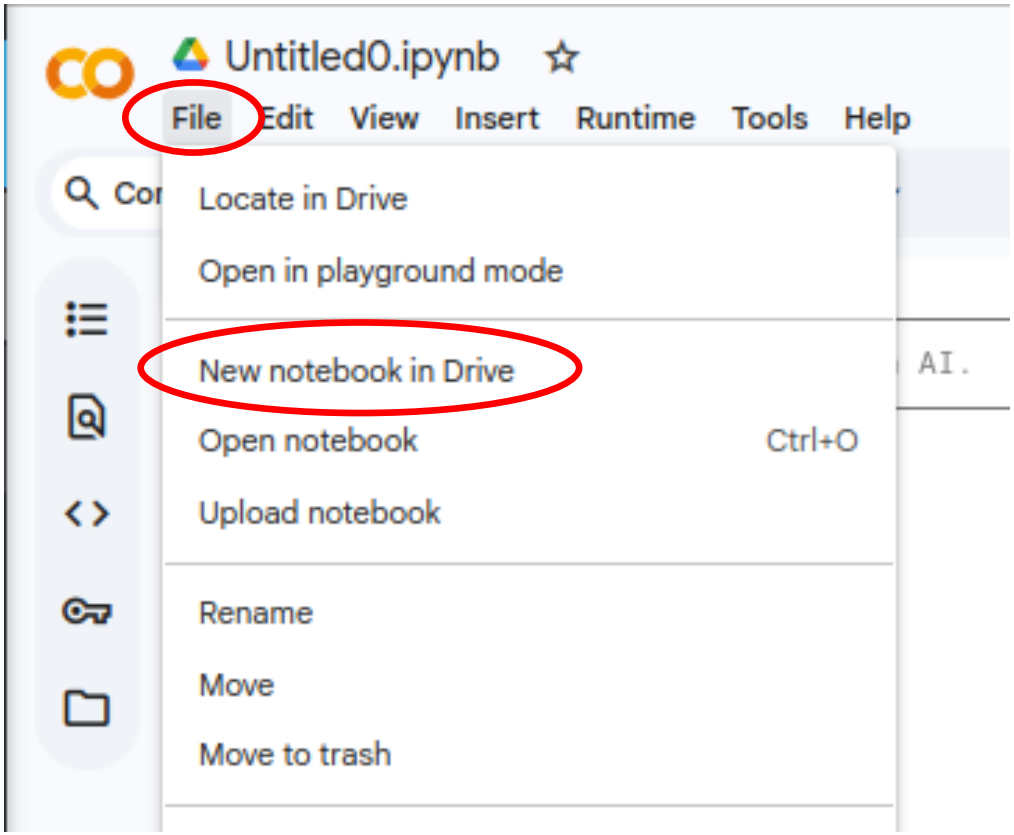
```
#include <stdio.h>

// Kernel function runs on device.
__global__ void mykernel( )
{
   // Every thread prints its block and thread ID.
   printf( "number of blocks in grid %d\n", gridDim.x );
   printf( "number of threads in block %d\n", blockDim.x );
   printf( "block index %d, thread index %d\n", blockIdx.x, threadIdx.x );
}

// Host function runs on CPU.
int main( void )
{
   int griddim = 2,  blockdim = 3;

   // Start 2 blocks with 3 threads each running function mykernel.
   mykernel <<<griddim, blockdim>>> ( );
   return 0;
}
```



Working with Google Colab

https://colab.research.google.com/

Create a new Jupyter Notebook on Google Drive



test.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Connect T4

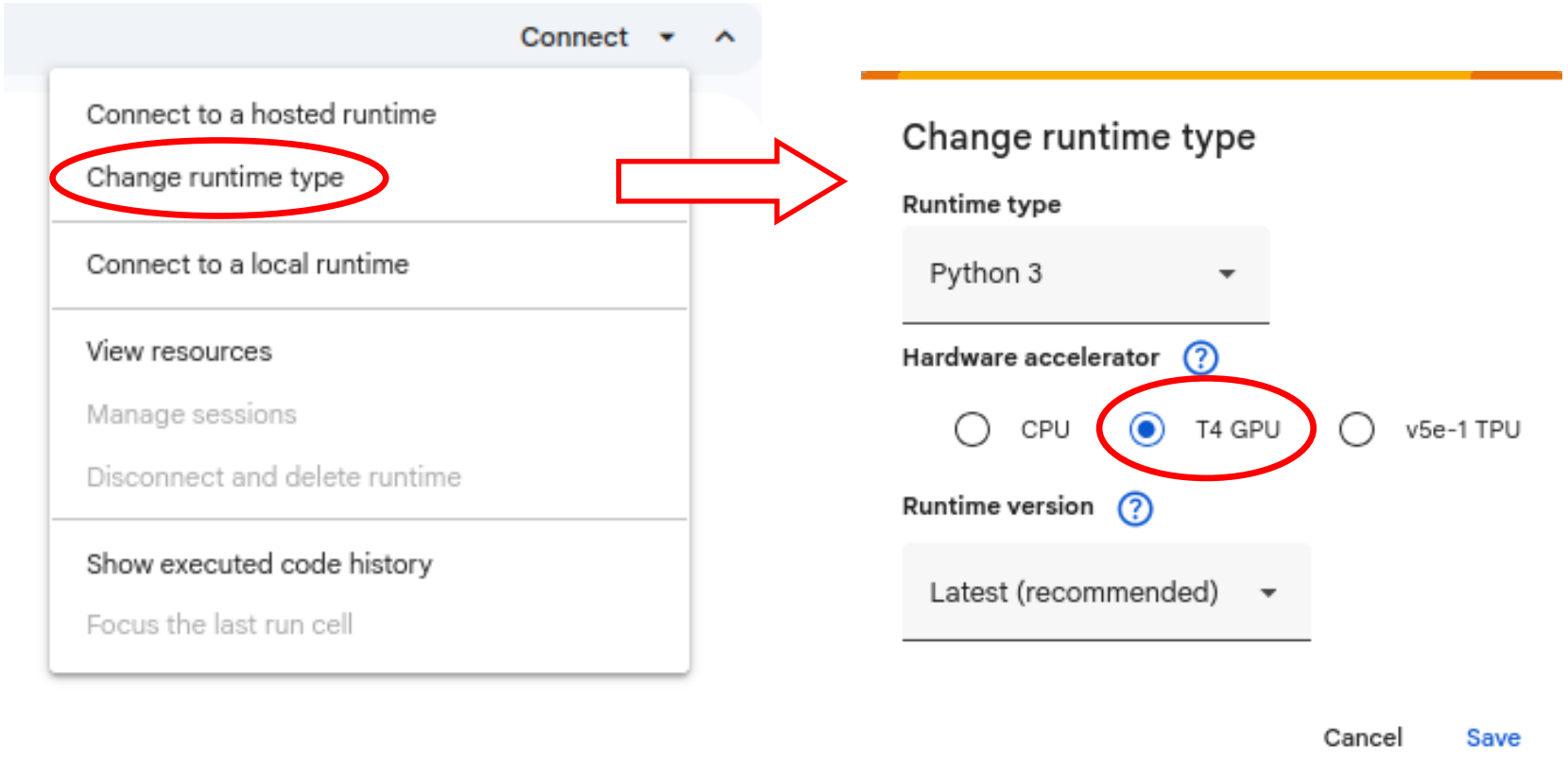
```
%%writefile test.cu

#include <stdio.h>

// Kernel function runs on GPU
__global__ void mykernel()
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;

  printf("this is thread %d\n",index);
}

int main(void)
{
  mykernel<<<2,3>>>();  // 2 blocks, 3 threads per block

  // Wait for GPU to finish.
  cudaDeviceSynchronize();

  return 0;
}
```

```
!nvcc -arch=sm_75 test.cu
```

```
!./a.out
```

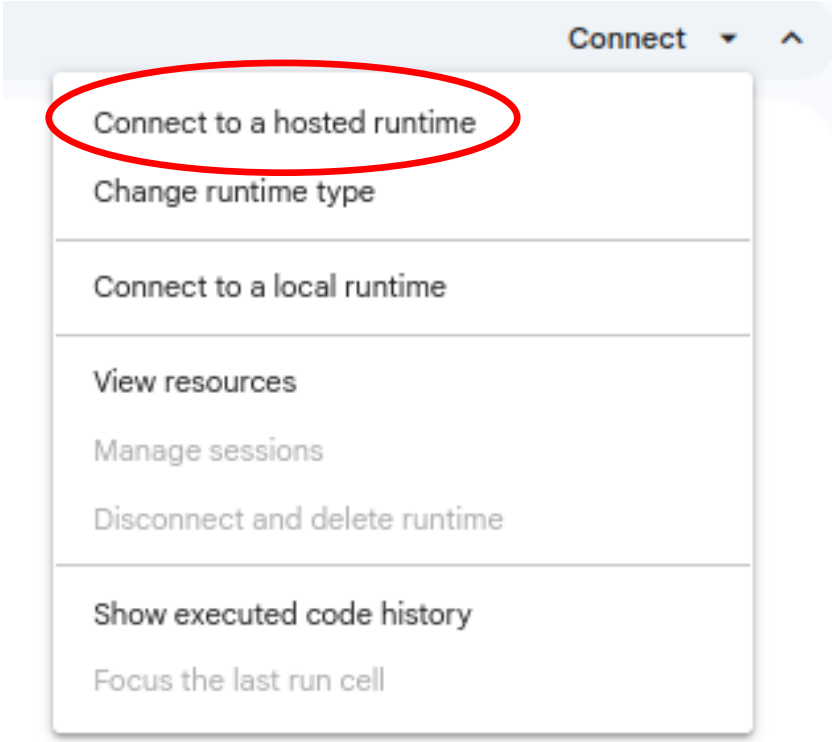
Working with Google Colab

Select „runtime type“ as T4 GPU.



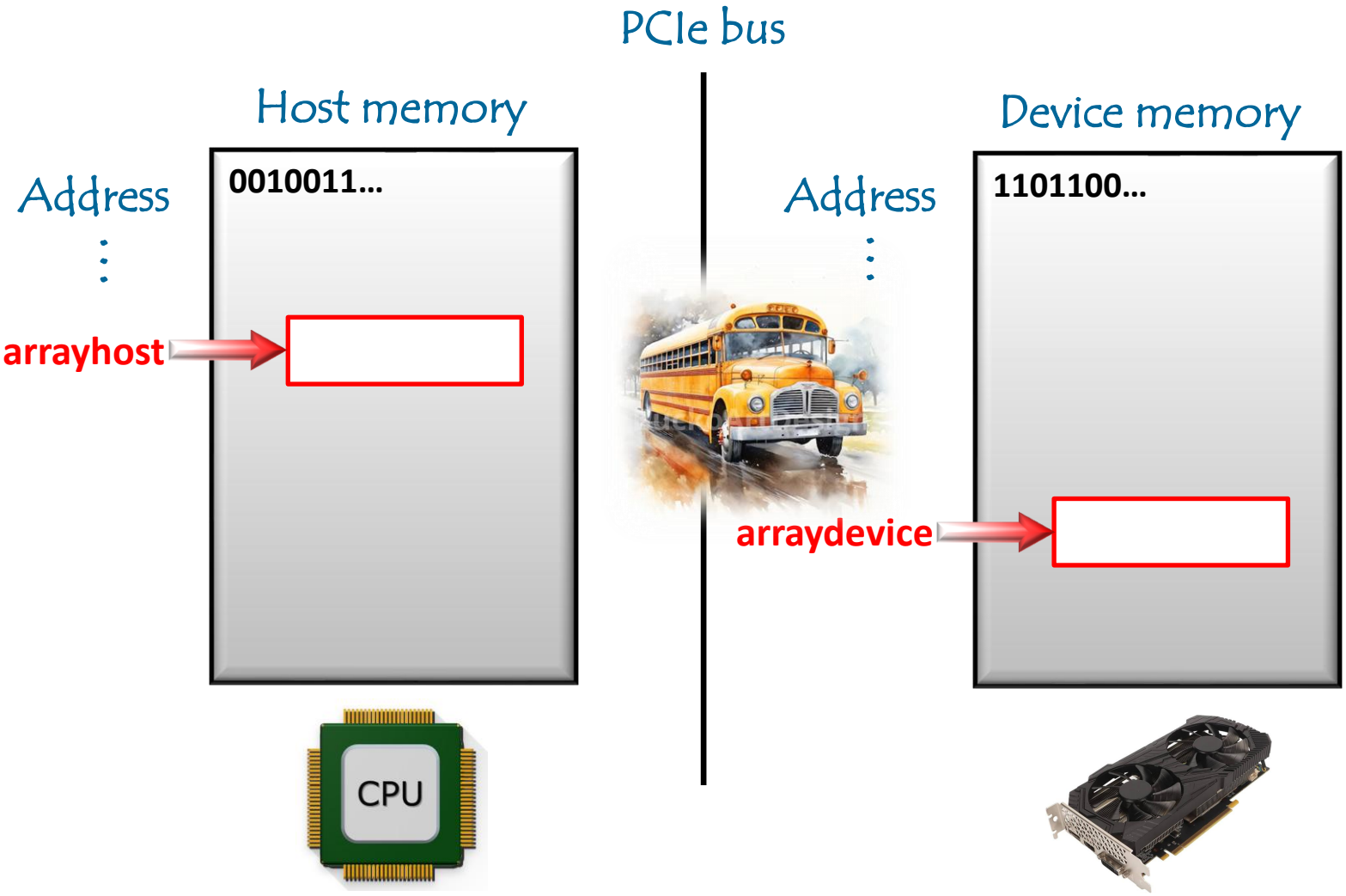
Working with Google Colab

Before running your program: „Connect to a hosted runtime“



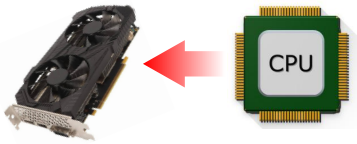
„Disconnect and delete runtime“ afterwards!
(time limit for free access to GPU)

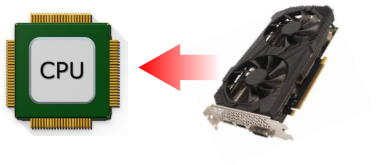
Data Transfer between Host and GPU



Data Transfer between Host and GPU

```
__global__ void mykernel( float * arraydevice ) { ... }

int main( void )
{
   int size = 1024*sizeof( float );
   float * arrayhost;      // pointer in host memory.
   float * arraydevice;    // pointer in device memory.

   // Allocate memory on host and on device.
   arrayhost = (float *)malloc( size );
   cudaMalloc( &arraydevice, size );

   // Copy memory from host to device.
   cudaMemcpy( arraydevice, arrayhost, size, cudaMemcpyHostToDevice );

   mykernel <<<3, 128>>> ( arraydevice );   // Start threads.
   cudaDeviceSynchronize( );                // Host waits until device has finished computations.

   // Copy memory from device to host.
   cudaMemcpy( arrayhost, arraydevice, size, cudaMemcpyDeviceToHost );
}
```





Example: Vector addition

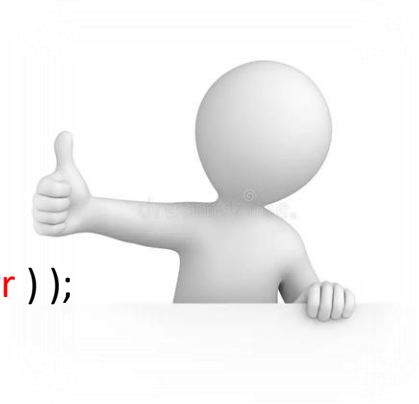
Debugging CUDA Programs

- Every CUDA function returns an error code.

```
cudaError_t err = cudaMalloc( &arraydevice, size );
if( err != cudaSuccess )
    printf( "CUDA Error %s\n ", cudaGetErrorString( err ) );
```

- At the end of a programm check for errors.

```
cudaError_t err = cudaGetLastError( );
if ( err != cudaSuccess )
   printf( "CUDA Error: %s\n", cudaGetErrorString( err ) );
```



Debugging CUDA Programs

- Enclose all CUDA function calls in a macro.

```
#define CHECK( call )                                                                    \
do                                                                                       \
{                                                                                        \
    cudaError_t err = call;                                                              \
    if( err != cudaSuccess )                                                             \
    {                                                                                    \
        printf( "Error at %s %d : %s\n", __FILE__, __LINE__, cudaGetErrorString( err ) );  \
        exit( EXIT_FAILURE );                                                            \
    }                                                                                    \
}                                                                                        \
while( 0 )  // only one iteration to create local scope for variable err.
```

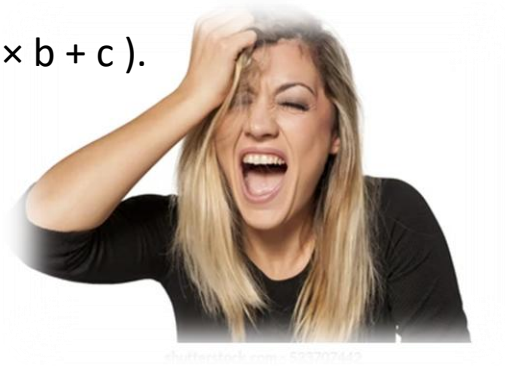

```
CHECK( cudaMalloc( &arraydevice, size ) );
```

Debugging CUDA Programs

- Compare floating point results of GPU with CPU

Floating point standard IEEE 754 guarantees identical results on CPU and GPU.

BUT…



GPU uses „fused multiply add“ by default: fma(a,b,c) = round(a × b + c).

CPU rounds after each instruction: round(round(a × b) + c).

Solution 1: switch off fma on **GPU**. Compile with

```
nvcc --fmad false
```

Solution 2: use fma on **CPU**. Compile with

```
nvcc -Xcompiler="-mfma -march=native"
```

```
#include<math.h>
fmaf(a,b,c);        // instead of a*b+c
```

Optimizing compiler might still change order of floating point operations causing differences!

Time Measurement with CUDA Events

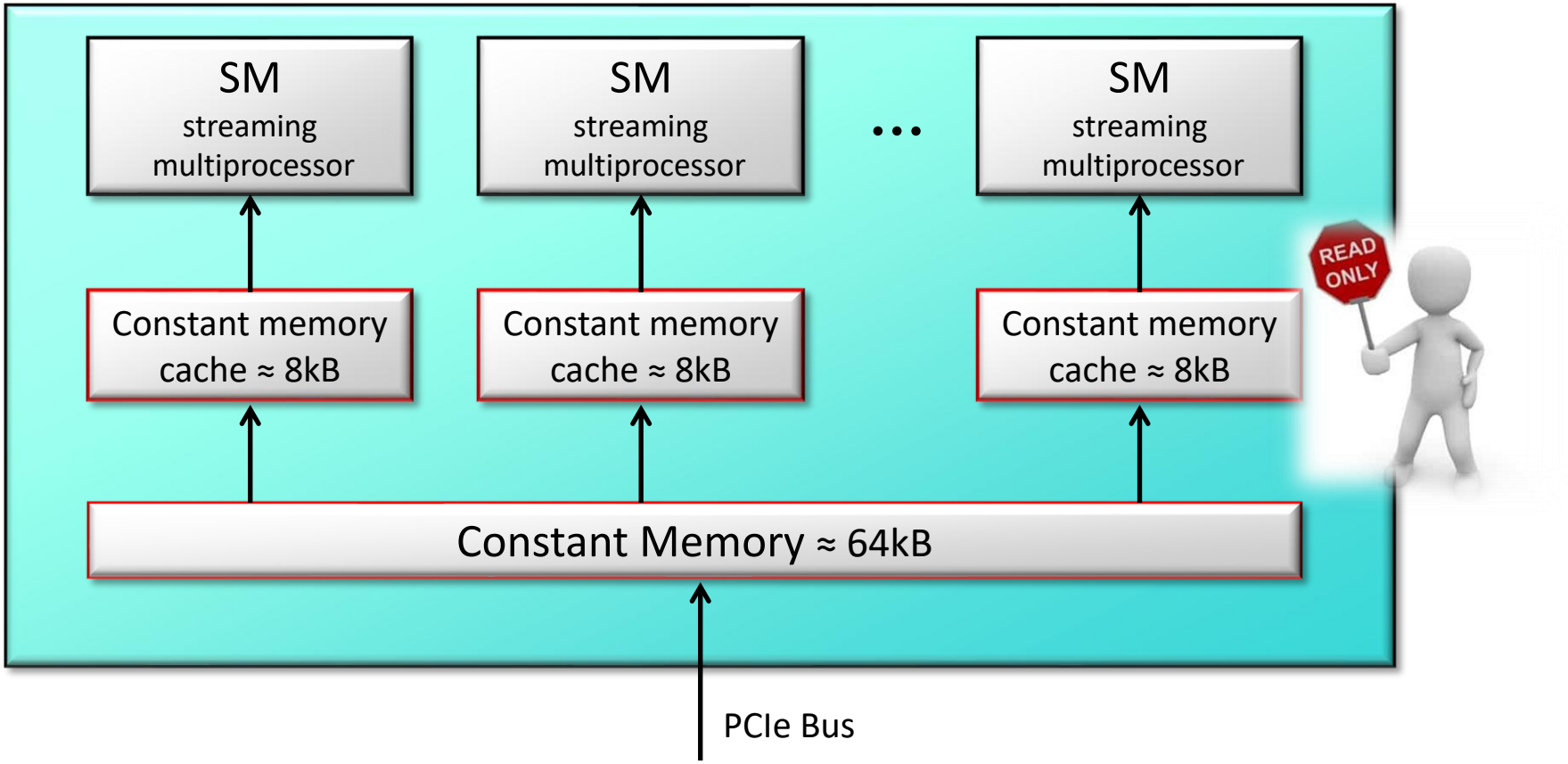
In host function:

```
// Create two CUDA events.
cudaEvent_t event1, event2;
cudaEventCreate( &event1 );
cudaEventCreate( &event2 );

// Add events and kernel to instruction stream for GPU.
cudaEventRecord( event1 );
mykernel<<< 16, 128 >>> ( ... );
cudaEventRecord( event2 );

// Host waits until event2 has been processed by GPU.
cudaEventSynchronize( event2 );

// Elapsed time in milliseconds.
float milliseconds;
cudaEventElapsedTime( &milliseconds, event1, event2 );
printf( "Elapsed time for kernel in ms: %f\n", milliseconds );
```

Example: Vector addition with events

Constant Memory

Fast read-only memory with local cache on each SM (unfortunately very small).

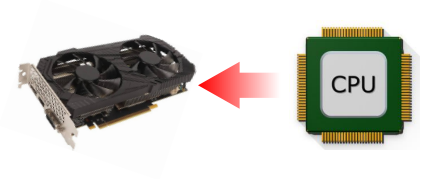


Function parameters to a kernel are also passed via constant memory.

Values of the Nvidia Geforce RTX 4060 GPU

Constant Memory

```
// Global variable for constant memory.
__constant__ float  constarray[1024];

// GPU can only read (not write) constant memory.
__global__ void mykernel( … )
{
   float value = constarray[ … ];
}

// Host copies local memory to constant memory on GPU.
int main()
{
   float hostarray[1024];
   …
   cudaMemcpyToSymbol( constarray, hostarray, 1024*sizeof(float) );
   mykernel<<< 16, 128 >>>( … );
}
```



Example: Vector addition with constant memory

Parallel Instruction Streams

CUDA stream:

Sequence of operations runing on GPU
(kernel launches, events, memory transfers)

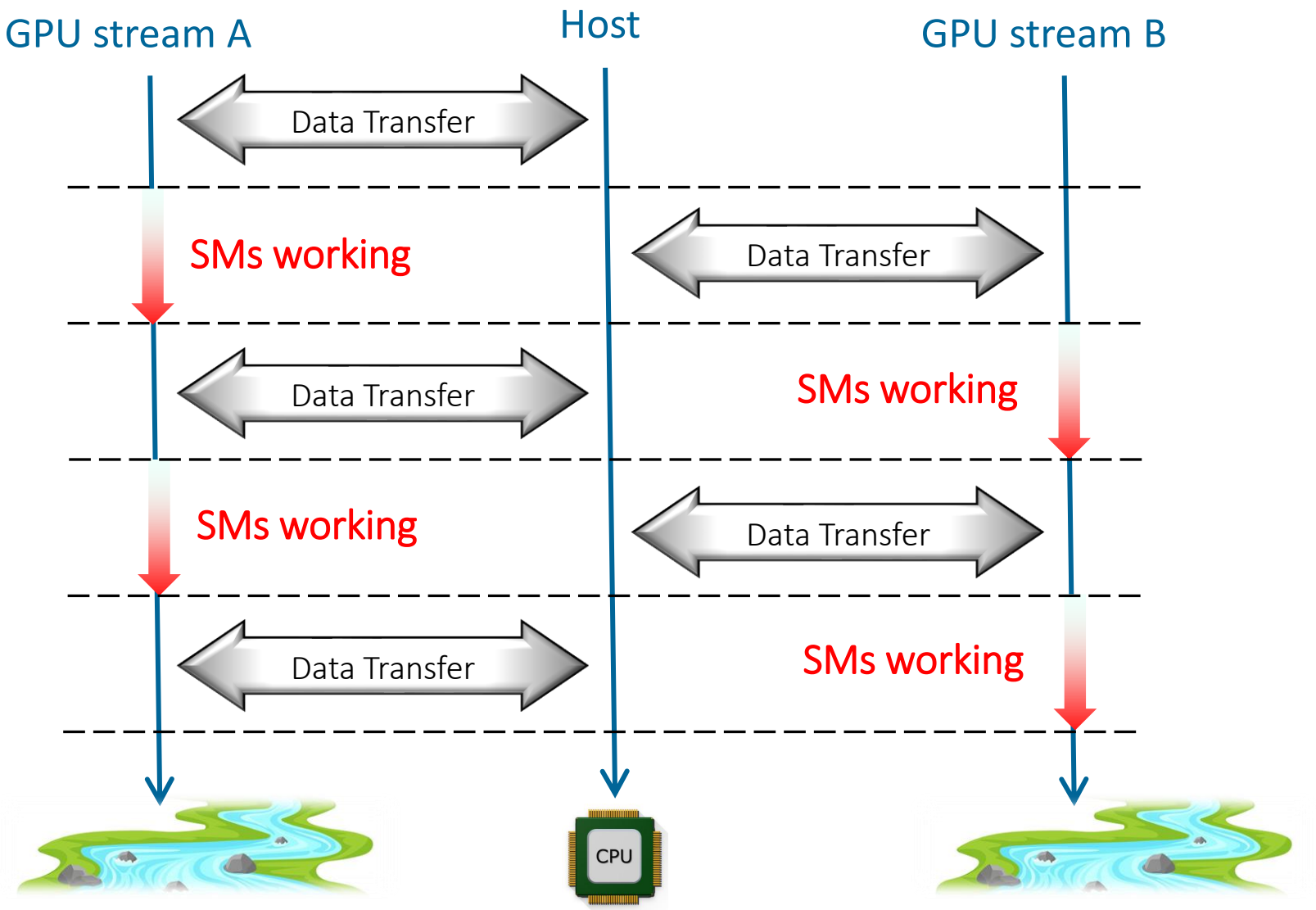
Operations within a stream run ***sequentially***
Operations in different streams run ***concurrently***

Purpose:

- Execute several kernels in parallel streams.
- Interleave computation and data transfer with host:
  one stream waits for data transfer while other stream uses SM‘s.

⇨ Higher throughput.

Parallel Instruction Streams



„latency hiding": With a single stream only half the utilization!

Parallel Instruction Streams

```
// Create two parallel streams.
cudaStream_t streamA, streamB;
cudaStreamCreate( &streamA );
cudaStreamCreate( &streamB );

// Add kernel1 and kernel2 to stream A.
kernel1<<< 32, 256, 0*, streamA >>> ( ... );
kernel2<<< 10, 128, 0*, streamA >>> ( ... );

// Add kernel3 to stream B.
kernel3<<< 16, 512, 0*, streamB >>> ( ... );

// Stream A and B run concurrently.

// Wait for streams to finish.
cudaStreamSynchronize( streamA );
cudaStreamSynchronize( streamB );
```

If no stream argument is provided, default stream 0 is used.

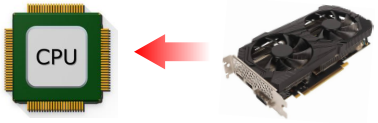
*Third parameter in <<< … >>> is used for shared memory

Parallel Instruction Streams

Asynchronous data transfer between host and a stream

```
// Memory allocation (pinned on host!)
float *arrayhost, *arraydevice;
cudaMallocHost( &arrayhost, size );  // Memory is not swapped out by host OS!
cudaMalloc( &arraydevice, size );

// Create stream.
cudaStream_t stream;    cudaStreamCreate( &stream );

// Add data transfer host → device to stream (non-blocking for host!)
cudaMemcypAsync( arraydevice, arrayhost, size, cudaMemcpyHostToDevice, stream );
```


```
// Add kernel invocation to stream.
mykernel<<< 32, 256, 0, stream >>> ( );

// Add data transfer device → host to stream (non-blocking for host!)
cudaMemcypAsync( arrayhost, arraydevice, size, cudaMemcpyDeviceToHost, stream );
```




```
// Host waits until stream is processed.
cudaStreamSynchronize( stream );
```

Example: Vector addition on two parallel streams

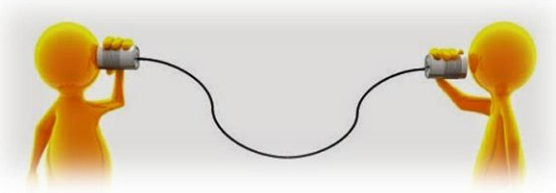
Parallel Instruction Streams

Events can be added to a streams and used for synchronization

```
// Create event.
cudaEvent_t event;
cudaEventCreate( &event );

// Create streams.
cudaStream_t streamA, streamB;
cudaStreamCreate( &streamA );
cudaStreamCreate( &streamB );

// Add event to streamA.
cudaEventRecord( event, streamA );
kernel1<<< 3, 64, 0, streamA >>> ( … );
```


```
// streamB launches kernel2 only after streamA has processed event.
cudaStreamWaitEvent( streamB, event, 0 );
kernel2<<< 2, 32, 0, streamB >>>( … );
```

Shared Memory / Thread Communication



- Declaration of a shared variable in a kernel.

  __global__ int x;

  Variable is shared among *all threads of the grid*.
  Stored in *global* memory (slow!).

  __shared__ int x;

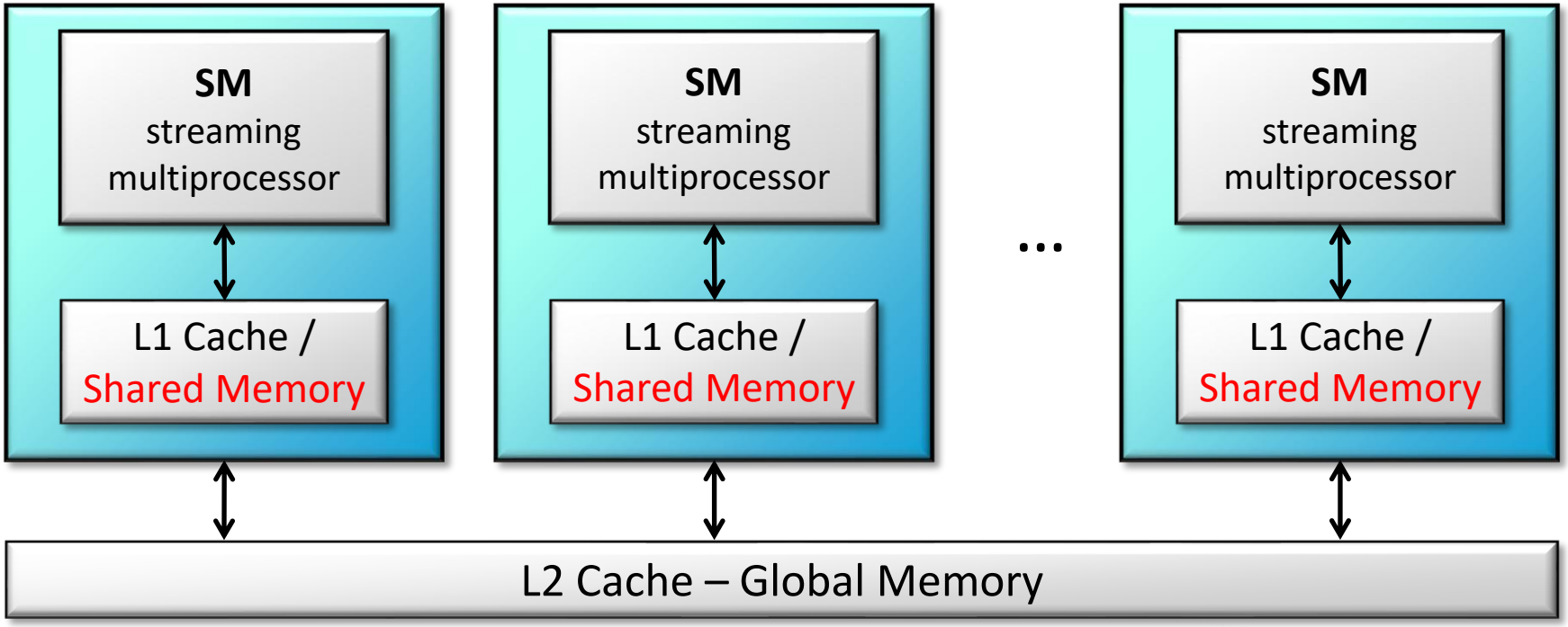
  Variable is shared among *all threads of the same block*.
  Stored in *local memory* of the SM (L1 cache / shared memory).
  Remember: all threads of a block run on the same SM.

- Synchronization of threads.

  __syncthreads( );

  Synchronize all threads of a block.
  Thread waits until all other threads of its block are at this point.

Shared Memory / Thread Communication



- Shared memory uses the same physical memory chip as L1 cache.
- Organized in 32 memory banks.
  Fast access of consecutive addresses by a warp.

Shared Memory / Thread Communication

```
__global__ void mykernel( )
{
   __shared__ int x = 17;       // Initialize shared variable x with 17.
   __syncthreads( );            // Wait until all threads are here.
-------------------------------------------------------------------
   if( threadidx.x == 0 )
      x = 42;                   // Thread 0 writes 42 to shared variable x.
   __syncthreads( );            // Wait until all threads are here.
-------------------------------------------------------------------
   if( threadidx.x == 1 )
      printf( "x = %d\n", x );  // Thread 1 reads shared variable x.
}

int main( )
{
   mykernel<<<1, 2>>>( );       // Start one block with two threads.
}
```

Shared Memory / Thread Communication

Dynamic shared memory: Size is determined *at runtime*!

```
__global__ mykernel( ... )
{
   extern __shared__ int mem[ ];        // Shared memory block.

   // All threads of this thread block have access to the fast shared memory mem.
}

int main( )
{
   int lengt = 10;                      // Length of shared memory block (dynamic)
   int size = length*sizeof( int );     // Size of shared memory block in bytes

   // size bytes are reserved in L1 cache / shared memory for each thread block.
   mykernel<<<16, 128, size>>>( ... );
}
```

Example: Parallel median filter

Project Proposals

Discrete Convolution with CUDA

Use constant memory for impulse response (< 64kB)

Parallel streams to hide latency of PCIe bus

- Split input signal in overlapping blocks
- Real time capability

Use fast shared memory for input signal to hide latency of global memory

---

DFT of a sequence of vectors with CUDA

Use constant memory for B-Matrix (Phasors < 64kB)

Parallel streams for consecutive vectors

Split dot product of a large vectors into several parallel threads

Timings and speedup curves with different block/grid size