

# GPU Programming

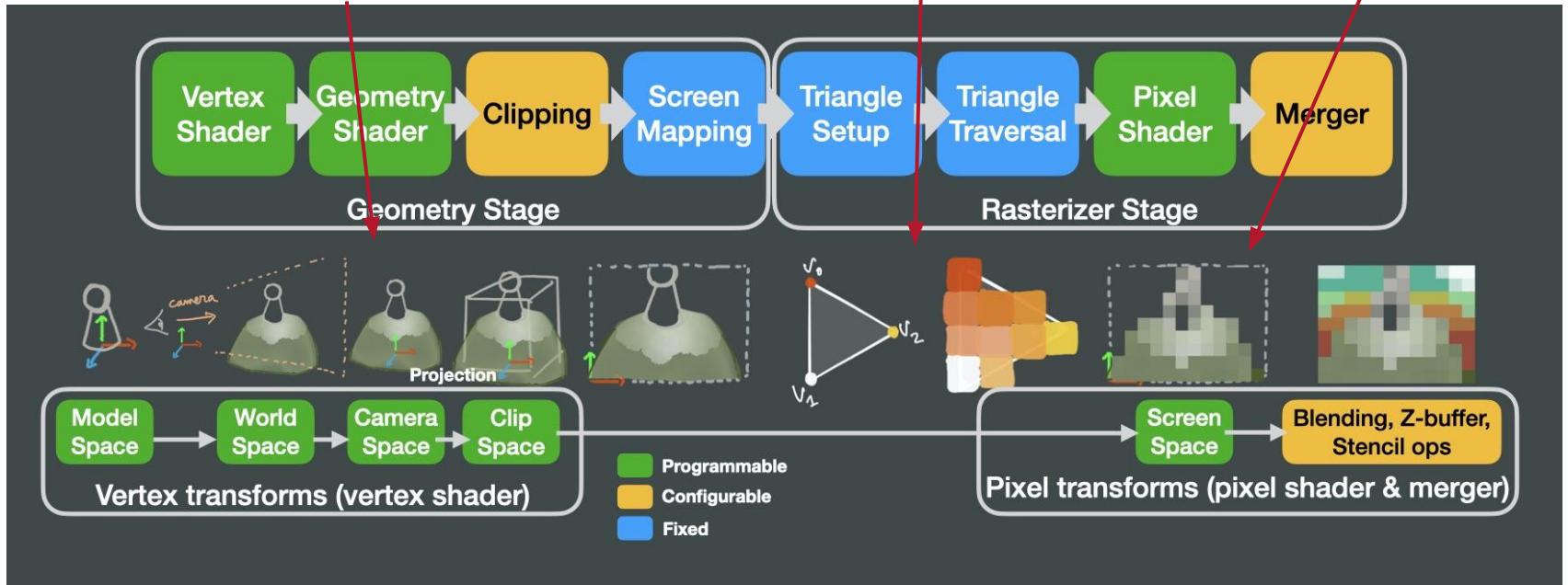
Stephan Hageboeck  
August 2023

# The Roots of GPU Computing

Linear algebra / matrix multiplications (floating-point computations)

Interpolations / Texture mapping

Integer operations on pixel arrays



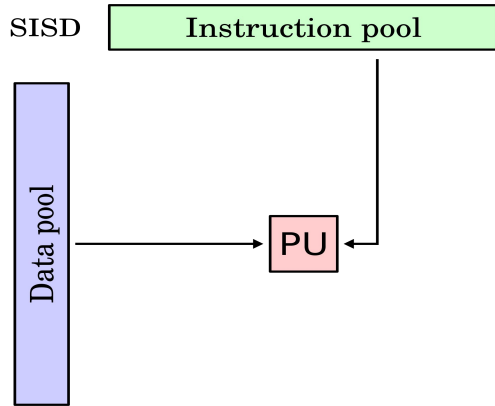
<https://tech.metal.com/the-stencil-buffer-and-how-to-use-it-to-visualize-volume-intersections/>

# General-Purpose GPU Computing

- ▶ The beginning: Abuse shader programs for GPU computing
- ▶ 2007: Nvidia's CUDA released
  - Compute Unified Device Architecture
  - Oldest and most frequent GPU programming interface
  - Exclusively for Nvidia GPUs
- ▶ 2016: AMD's ROCm (HIP)
  - Similar to CUDA, but targets AMD GPUs
- ▶ ~ now: Intel GPUs with SYCL
- ▶ Abstraction frameworks:
  - Program abstract kernels, translate to native GPU language later
  - Target multiple vendors
  - Some well-known candidates: OpenCL, SYCL, Alpaka, Kokkos

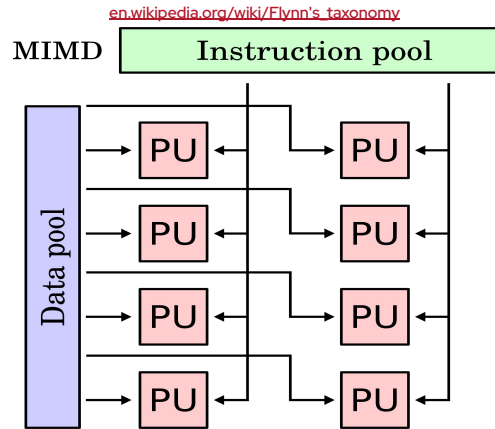


# GPUs in Flynn's Taxonomy



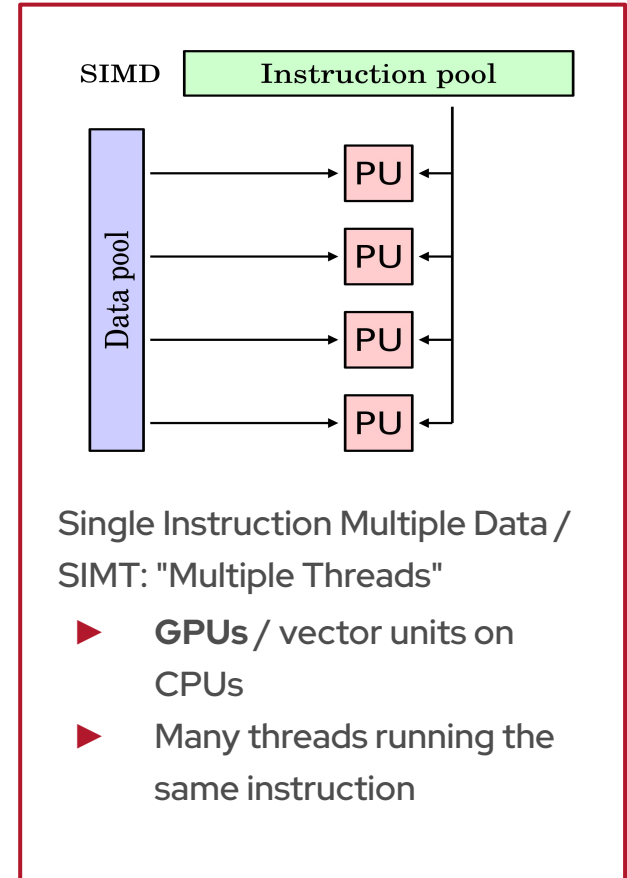
Single Instruction Single Data

- ▶ Single core executing one instruction after the other



Multiple Instruction Multiple Data

- ▶ E.g. multi-core processor
- ▶ Multiple independent threads of execution



Single Instruction Multiple Data /  
SIMT: "Multiple Threads"

- ▶ **GPUs** / vector units on CPUs
- ▶ Many threads running the same instruction

# CPU vs GPU

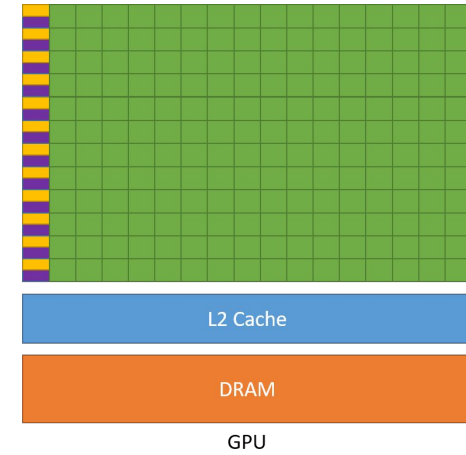
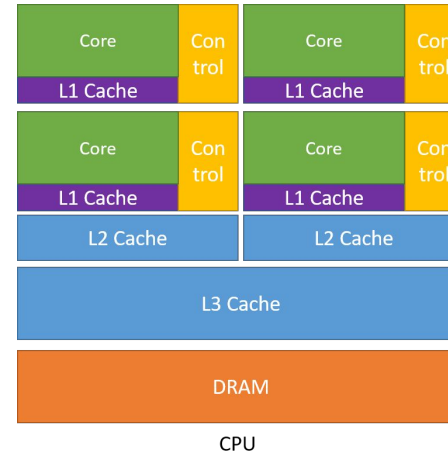
## ▶ CPUs:

- Try to reduce latency of execution
- Every core (~ thread) has its own cache and control unit, several levels of caches
- Lots of energy and hardware spent on latency reduction

## ▶ GPUs:

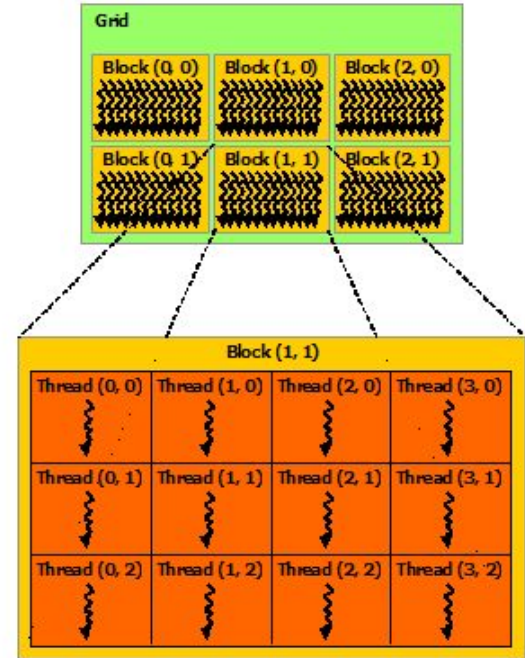
- Threads **share** control units and a lot of cache space
- More transistors devoted to data processing
- → Massive data parallelism
- → Higher latency

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/>



# CUDA Execution Model

- ▶ **Kernel:** Function invoked on the GPU
- ▶ **Grid:** Set of blocks running for a specific kernel
- ▶ **Block:** Set of threads on the same "streaming multiprocessor"
- ▶ **Thread:** Set of instructions to be executed

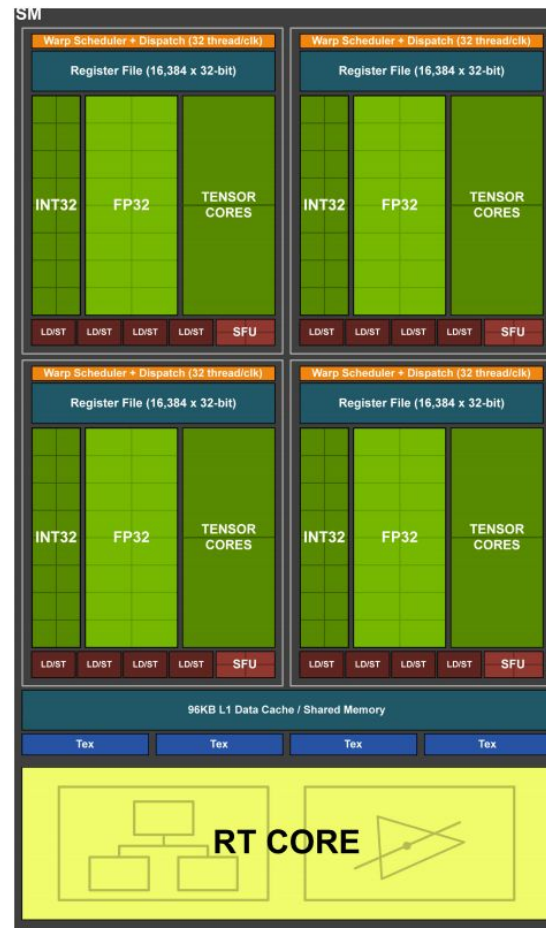


# CUDA Execution Model II

- ▶ All threads in a block run on the same Streaming Multiprocessor
- ▶ In every cycle, SMs can execute **one** instruction simultaneously on 32 threads ("warp")
- ▶ Blocks run in unspecified order



## Turing SM



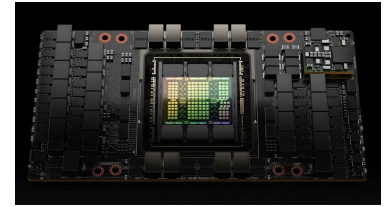
# Some Hardware Examples



**Nvidia RTX 2070**



**Nvidia Tesla T4**



**Nvidia Tesla H100  
PCIe**

<b>CUDA Cores</b>	2304 (36 SMs)	2560 (40 SMs)	7296 (114 SMs)
<b>Max GPU Clock Rate</b>	1710 MHz	1590 MHz	1755 MHz
<b>Single Precision Perf.</b>	7.5 TFLOPS (32:1)	8.1 TFLOPS (32:1)	51 TFLOPS (2:1)
<b>VRAM</b>	8GB GDDR6	16 GB GDDR6 / ECC	80 GB HBM2 / ECC
<b>Memory Bandwidth</b>	448 GB/s	300 GB/s	2000 GB/s
<b>TDP</b>	185 W	70 W	350 W





# Our First CUDA Program

```
#include <cstdio>

// kernel definition
__global__ void HelloWorld() {
    printf("Hello world\n");
}

int main() {
    const auto nBlock = 1;
    const auto nThread = 1;
    HelloWorld<<< nBlock , nThread >>> ();
    return 0;
}
```

- ▶ A function compiled for the GPU is a "kernel"
- ▶ It supports a limited subset of C++ (e.g. no standard library)
- ▶ Mark with `__global__` and invoke as `KernelName<<< , >>>(args)`
- ▶ Compile as  
`nvcc -std=c++17 helloWorld.cu -o helloWorld`

# Our First CUDA Program

```
#include <cstdio>

// kernel definition
__global__ void HelloWorld() {
    printf("Hello world\n");
}

int main() {
    const auto nBlock = 1;
    const auto nThread = 1;
    HelloWorld<<< nBlock , nThread >>> ();
    cudaDeviceSynchronize();
    return 0;
}
```

- ▶ Kernels run asynchronously on the GPU
- ▶ We need to wait for the kernel to complete

# Our First CUDA Program

```
#include <cstdio>
#include <iostream>

// kernel definition
__global__ void HelloWorld() {
    printf("Hello world\n");
}

int main() {
    const auto nBlock = 1;
    const auto nThread = 1;

    HelloWorld<<< nBlock , nThread >>> ();

    if (auto errorCode = cudaDeviceSynchronize();
        errorCode != cudaSuccess) {
        std::cerr << "Encountered cuda error '"
            << cudaGetErrorName(errorCode)
            << "' with description:"
            << cudaGetErrorString(errorCode) << "\n";
        return 1;
    }

    return 0;
}
```

- ▶ The host doesn't get notified about errors in kernels
- ▶ We need to check manually
- ▶ Everything is asynchronous. Errors might even be from previous kernel launches!
- ▶ Usually, each framework has their own "CheckCuda" macro/function, e.g.  
`CHECK_CUDA( cudaDeviceSynchronize() )`

# What happens on the GPU

## Nvidia Tesla T4

▶ HelloWorld<<<1,1>>>();



# What happens on the GPU

## Nvidia Tesla T4

▶ HelloWorld<<<1,1>>>();



# What happens on the GPU

## Nvidia Tesla T4

- ▶ HelloWorld<<< 1, 32 >>>();
- ▶ Recommendation:
  - Block sizes should be multiples of the warp size, 32
  - For modern SMs, use minimum 64



# What happens on the GPU

- ▶ HelloWorld<<< 1, 128 >>>();
- ▶ Why would you send more threads than cuda cores?  
→ Memory access (later)

## Nvidia Tesla T4





# What happens on the GPU

## Nvidia Tesla T4

► HelloWorld<<< 4, 64 >>>();



# What happens on the GPU

- ▶ HelloWorld<<< 40, 64 >>>();
- ▶ On a Tesla T4, we need 2560 threads to have "one full wave" of warps that can span across the entire GPU
- ▶ In general, go even higher to fully fill the GPU pipelines
- ▶ NB: We are still not using the floating-point units (nor tensor cores / ray-tracing cores)

## Nvidia Tesla T4



# Getting info on devices

"System management interface"

```
[1]shageboe@lxplus764:julia$ nvidia-smi
Fri Jul 22 18:22:47 2022

+-----+
| NVIDIA-SMI 515.48.07      Driver Version: 515.48.07      CUDA Version: 11.7      |
+-----+-----+
| GPU   Name           Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|=====+=====+
| 0   Tesla T4            Off          | 00000000:00:08.0 Off  |                    0 |
| N/A   57C    P0      29W / 70W | 2MiB / 15360MiB |      5%      Default |
|=====+=====+
+-----+-----+

Processes:
+-----+-----+
| GPU   GI    CI          PID   Type   Process name                      GPU Memory |
| ID   ID   ID              |              |           | Usage |
+-----+-----+
|
| No running processes found
|
+-----+-----+
[1]shageboe@lxplus764:julia$
```

```
[1]shageboe@lxplus764:julia$ /usr/local/cuda/extras/demo_suite/deviceQuery
/usr/local/cuda/extras/demo_suite/deviceQuery Starting...
```

```
CUDA Device Query (Runtime API) version (CUDA static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla T4"
  CUDA Driver Version / Runtime Version      11.7 / 11.3
  CUDA Capability Major/Minor version number: 7.5
  Total amount of global memory:            14964 MBytes (15691415552 bytes)
  (40) Multiprocessors, ( 64) CUDA Cores/MP: 2560 CUDA Cores
  GPU Max Clock rate:                       1590 MHz (1.59 GHz)
  Memory Clock rate:                         5001 Mhz
  Memory Bus Width:                          256-bit
  L2 Cache Size:                             4194304 bytes
  Maximum Texture Dimension Size (x,y,z)    1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
  Total amount of constant memory:          65536 bytes
  Total amount of shared memory per block:  49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                 32
  Maximum number of threads per multiprocessor: 1024
  Maximum number of threads per block:      1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                     2147483647 bytes
  Texture alignment:                         512 bytes
  Concurrent copy and kernel execution:      Yes with 3 copy engine(s)
  Run time limit on kernels:                 No
  Integrated GPU sharing Host Memory:        No
  Support host page-locked memory mapping:   Yes
  Alignment requirement for Surfaces:        Yes
  Device has ECC support:                    Enabled
  Device supports Unified Addressing (UVA):  Yes
  Device supports Compute Preemption:        Yes
  Supports Cooperative Kernel Launch:        Yes
  Supports MultiDevice Co-op Kernel Launch:  Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 0 / 8
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

If you don't know where  
the cuda installation is, try  
"which nvcc" first

```
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.7, CUDA Runtime Version = 11.3, NumDevs = 1, Device0 = Tesla T4
Result = PASS
```

```
[1]shageboe@lxplus764:julia$
```

## Question:

I want to add two vectors with 1000 elements.

CPU or GPU?

## **Question:**

I want to add two vectors with 2560 elements.

CPU or GPU?

# Memory Management

```
int main() {
    constexpr size_t N = 1'000'000;

    float * data;
    cudaMallocManaged(&data, N * sizeof(float));

    kernel<<< 100, 1024 >>>(data);
    cudaDeviceSynchronize();

    // Touching the pointer triggers a
    // copy to the host
    data[i];

    cudaFree(data);
    // ...
}
```

- ▶ GPUs require C-style memory management with `cudaMalloc` and `cudaFree`
- ▶ Your data should fit in arrays for best performance
- ▶ Pascal (2016) and later architectures support unified addressing in host and kernel code
  - Data is copied on demand when accessed
  - Don't touch it repeatedly in kernel *and* host! Slow!

# Memory Management II

```
int main() {
    constexpr size_t N = 1'000'000;

    float * data_gpu;
    float * data_cpu;
    cudaMalloc(&data_gpu, N * sizeof(float));
    cudaMallocHost(&data_cpu, N * sizeof(float));

    kernel<<< 100 , 1024 >>>(data_gpu);
    cudaMemcpy(data_cpu, data_gpu, N * sizeof(float),
               cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();

    // ...
    data_cpu[i];

    cudaFree(data_gpu);
    cudaFree(data_cpu);
    // ...
}
```

## Classic Style:

- ▶ Manage host and device memory separately
- ▶ Explicitly request transfers
- ▶ More explicit, more control

## Recommendations:

- ▶ Compute work on GPU >> copy work (Host→Dev / Dev→Host)
- ▶ Prefer copying few large arrays over many small pieces



# Thread and Block Indexing

```
__global__ void kernel() {  
    printf("I am thread %d block %d\n",  
          threadIdx.x, blockIdx.x);  
}  
  
// ...  
kernel<<< 4, 256 >>>();
```

- ▶ How does a thread know what it has to do?
  - Implicit variables threadIdx and blockIdx
  - x, y, z dimension; often y = z = 1
  - Maximum block size: 1024 threads
  - Maximum grid size: depends on device (e.g. 2 billion)
- ▶ Submit kernels either with integers (x dimension) or tuples of type `dim3`

# Thread and Block Indexing

```
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1
    // threads
    int numBlocks = 1;
    dim3 threadsPerBlock(10, 10);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

- ▶ How does a thread know what it has to do?
  - Implicit variables threadIdx and blockIdx
  - x, y, z dimension; often y = z = 1
  - Maximum block size: 1024 threads
  - Maximum grid size: depends on device (e.g. 2 billion)
- ▶ Submit kernels either with integers (x dimension) or tuples of type **dim3**

# Addressing any Array on the GPU

```
__global__  
void add(int n, float *x, float *y)  
{  
    int index = threadIdx.x;  
  
    for (int i = index; i < n; ++i)  
        y[i] = x[i] + y[i];  
}
```

```
add<<< 1 , 256 >>>(N, x, y);
```

Let's assume we have arrays of arbitrary size

How do we access them?

1. Use `threadIdx.x` as offset
2. Exit if  $i \geq n$



# Addressing any Array on the GPU

```
__global__  
void add(int n, float *x, float *y)  
{  
    int index = threadIdx.x;  
  
    for (int i = index; i < n; ++i)  
        y[i] = x[i] + y[i];  
}
```

```
add<<< 1, 256 >>>(N, x, y);
```

**Works, but slow!**  
**Only 1 SM running!**

Let's assume we have arrays of arbitrary size

How do we access them?

1. Use `threadIdx.x` as offset
2. Exit if `i >= n`



# Addressing any Array on the GPU

```
__global__  
void add(int n, float *x, float *y)  
{  
    int index = blockIdx.x * blockDim.x  
              + threadIdx.x;  
  
    for (int i = index; i < n; )  
        y[i] = x[i] + y[i];  
}
```

```
add<<< 4096 , 256 >>>(N, x, y);
```

This is `blockDim.x`

3. Use multiple blocks, and distribute them as  $\text{blockIdx.x} * \text{blockDim.x}$



$$\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

$$\text{index} = (2) * (256) + (3) = 515$$

<https://developer.nvidia.com/blog/even-easier-introduction-cuda/>

# Addressing any Array on the GPU

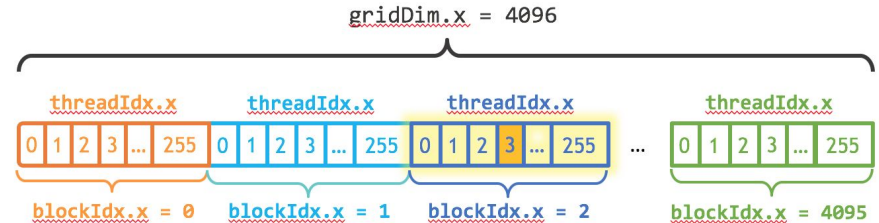
```
__global__  
void add(int n, float *x, float *y)  
{  
    int index = blockIdx.x * blockDim.x  
              + threadIdx.x;  
    int stride = blockDim.x * gridDim.x;  
    for (int i = index; i < n; i += stride)  
        y[i] = x[i] + y[i];  
}
```

```
add<<< 4096 , 256 >>>(N, x, y);
```

4. Step through the array with  $\text{block} * \text{grid}$  stride until done

► This is the **grid-strided loop**

- Number of blocks will only determine execution speed, *not* correctness!



`index = blockIdx.x * blockDim.x + threadIdx.x`

`index = (2) * (256) + (3) = 515`

<https://developer.nvidia.com/blog/even-easier-introduction-cuda/>

# Some more CUDA

```
__host__ __device__  
void processTrack(Track & track, OtherData * data) {  
    // ...  
}
```

```
__global__  
void kernel(Tracks * tracks, unsigned int N,  
            OtherData * data)  
{  
    for (unsigned int i = blockIdx.x * blockDim.x  
        + threadIdx.x;  
         i < N; i += blockDim.x * gridDim.x)  
    {  
        processTrack(tracks[i], data);  
    }  
}
```

- ▶ Only kernels (`__global__`) can be invoked from the host
- ▶ Kernels can invoke `__device__` functions
- ▶ `__host__ __device__` functions are compiled twice: once for cuda, once for C++
- ▶ Use same code for CPU/GPU

## Question:

Why would you send more threads than  
cuda cores ( $>64$ ) to one SM?



# Memory Latency and GPUs

- ▶ GPUs are made for throughput, not latency
- ▶ Hundreds of GPU cycles from load instruction to data arrival

```
for (int i = index; i < n; i += stride)  
    y[i] = x[i] + y[i];
```

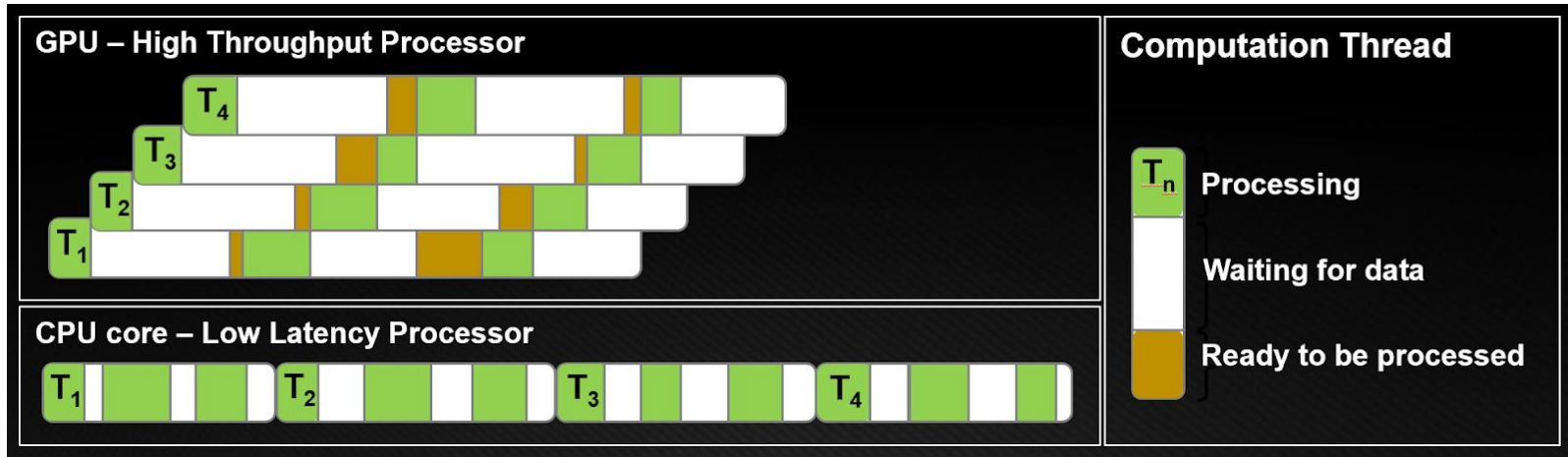


[developer.nvidia.com/blog/cuda-refresher-revisiting-the-origins-of-gpu-computing/](https://developer.nvidia.com/blog/cuda-refresher-revisiting-the-origins-of-gpu-computing/)

# Memory Latency and GPUs

- ▶ If a warp stalls, other warps step in
  - Requires multiple warps / SM
- ▶ **Recommendation:** Start with 256 threads – 8 warps – and try other numbers

```
for (int i = index; i < n; i += stride)  
    y[i] = x[i] + y[i];
```

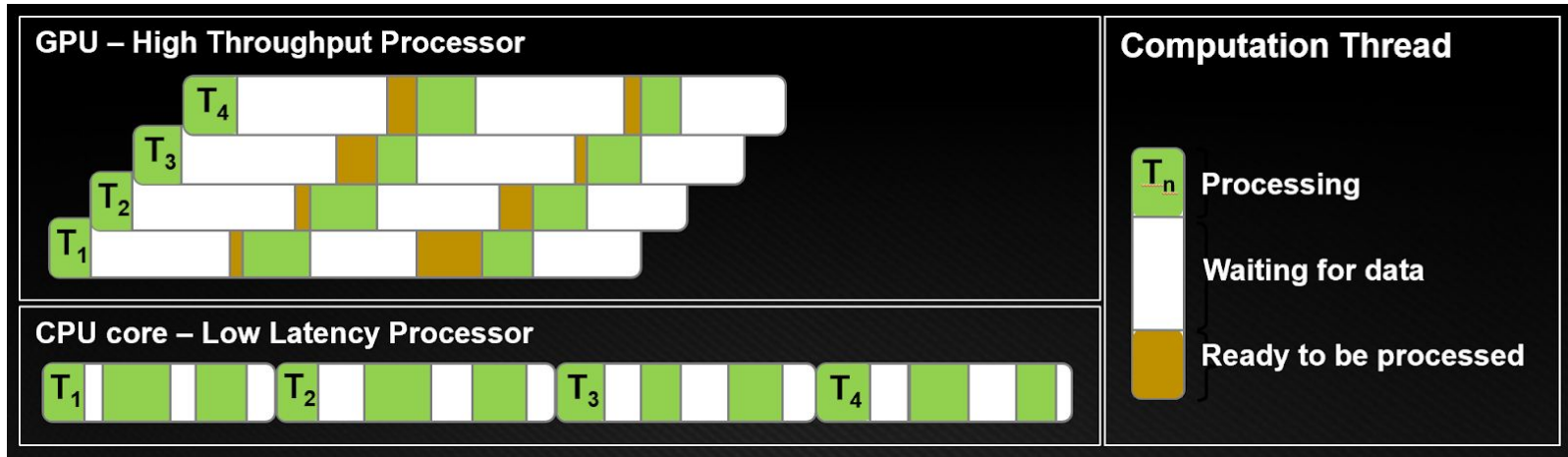


# Memory Latency and GPUs

- ▶ If a warp stalls, other warps step in
  - Requires multiple warps / SM
- ▶ **Recommendation:** Start with 256 threads – 8 warps – and try other numbers

```
for (int i = index; i < n; i += stride)  
    y[i] = x[i] + y[i];
```

- ▶ CPUs minimise latency, GPUs hide it
- ▶ You need enough work for the GPU to do this successfully



# Efficient memory access

*// Access with stride 1*

**\_\_global\_\_**

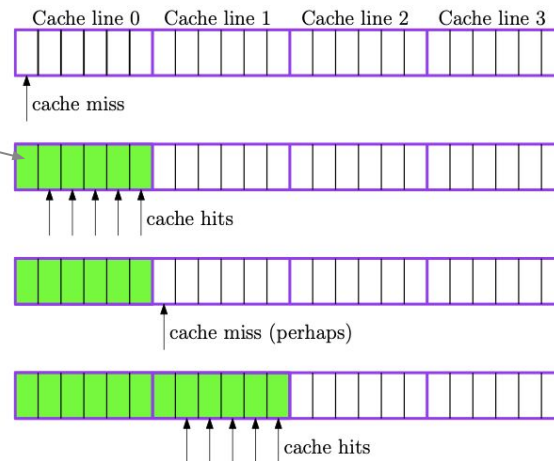
```
void k_copy(double* x, double* y, int n) {  
    for(int i = threadIdx.x + blockDim.x*blockIdx.x;  
        i < n; i += blockDim.x * gridDim.x)  
        y[i] = x[i];  
}
```

*// Access with larger stride*

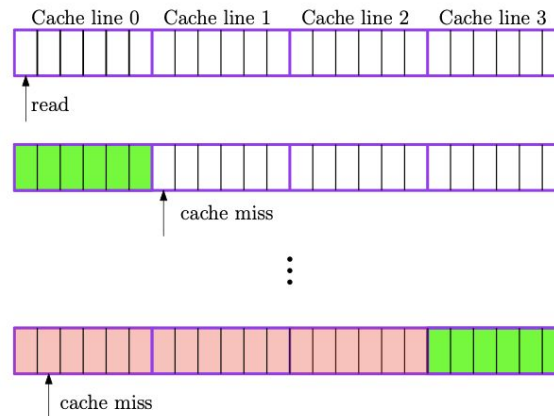
**\_\_global\_\_**

```
void k_copy(double* x, double* y, int n) {  
    for(int i = threadIdx.x * gridDim.x + blockIdx.x;  
        i < n; i += blockDim.x * gridDim.x)  
        y[i] = x[i];  
}
```

DRAM is read in bursts  
(e.g. 16 float/int in GDDR6)



L. Einkemmer



# Efficient memory access

// Access with stride 1

**\_\_global\_\_**

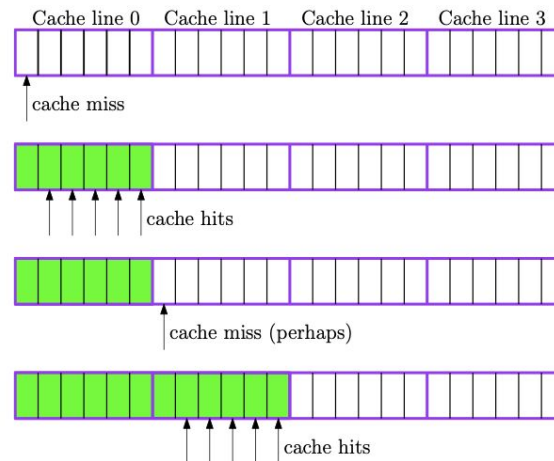
```
void k_copy(double* x, double* y, int n) {  
    for(int i = threadIdx.x + blockDim.x*blockIdx.x;  
        i < n; i += blockDim.x * gridDim.x)  
        y[i] = x[i];  
}
```

Neighbouring threads should read  
neighbouring memory locations  
"coalesced access"

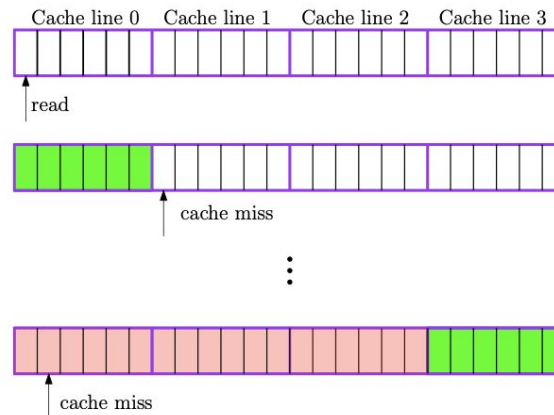
// Access with larger stride

**\_\_global\_\_**

```
void k_copy(double* x, double* y, int n) {  
    for(int i = threadIdx.x * gridDim.x + blockIdx.x;  
        i < n; i += blockDim.x * gridDim.x)  
        y[i] = x[i];  
}
```



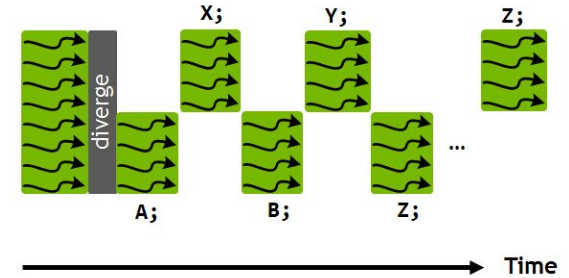
L. Einkemmer



# Branches

- ▶ If all threads in a warp have to run the same instruction, how do we branch?
  - The warp is split
  - Threads that don't take the branch are disabled
    - Throughput reduced
- ▶ Branching is not the fastest way of using a GPU, but it's OK on modern GPUs

```
if (threadIdx.x < 4) {  
    A;  
    B;  
} else {  
    X;  
    Y;  
}  
Z;
```



[developer.nvidia.com/blog/inside-volta/](https://developer.nvidia.com/blog/inside-volta/)

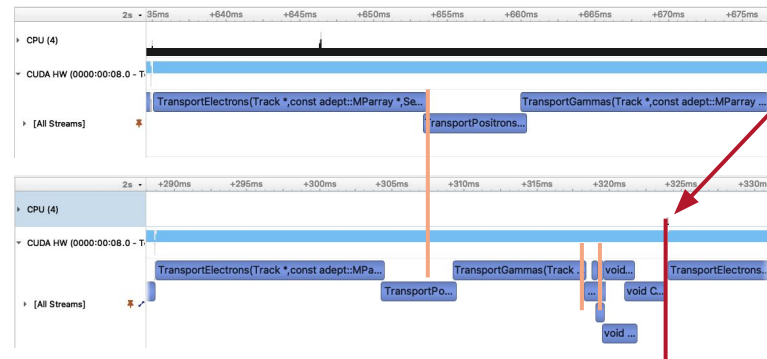
# Take-Home Messages

- ▶ GPUs are devices that require a high degree of data parallelism
  - They excel when the same operations have to be applied to many pieces of data (SIMD)
  - GPUs hide memory latency instead of minimising it
- ▶ Minimise frequent data copies between host/device for best performance
- ▶ GPUs work best with arrays and coalesced data access
- ▶ To max out a GPU, start with
  - $nBlock \geq 2 * nSM$
  - 256 threads / block
- ▶ Learn to use the grid-strided loop
- ▶ AMD GPUs work very similar to what we learned today

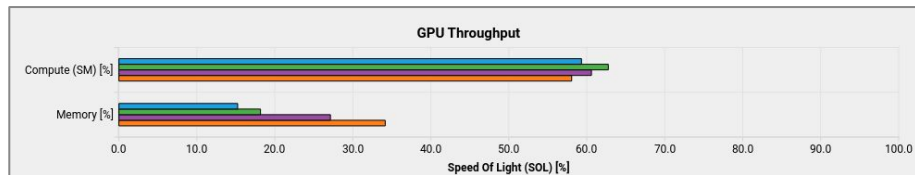
# What we didn't talk about ...

- ▶ Profiling
  - Understand kernel performance in detail
  - [Nsight systems](#) / [Nsight compute](#)
- ▶ Concurrent kernel execution  
→ streams
- ▶ Memory hierarchies
  - Registers, shared memory, L1/L2 cache, global memory
- ▶ Tuning device occupancy, minimising warp divergence
- ▶ Atomics and synchronisation

Screenshot of a particle simulation in Nsight systems



Throughput measurement in Nsight compute





# Resources

- ▶ [Cuda C++ Programming Guide](#)
- ▶ Profiling: [Nsight systems](#) / [Nsight compute](#)
- ▶ Insights into CUDA programming: NVidia technical blog  
e.g.:
  - [An even easier introduction to CUDA](#)
  - [Efficient memory access](#)
  - [Efficient matrix transpose](#)

# Hands on

# The Exercises

## 1. helloWorld

- Write first kernel
- Start it from the host

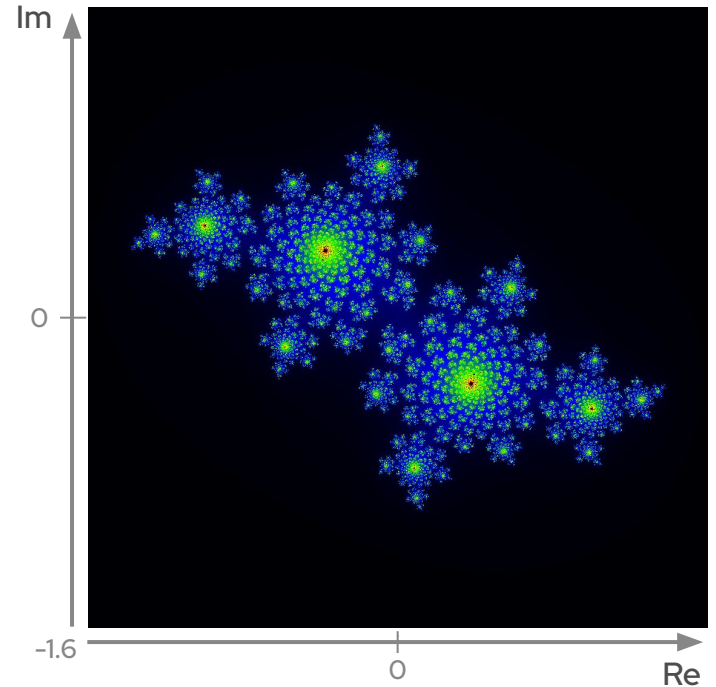
## 2. vectorAdd

- Write a proper grid-strided loop
- Test different launch configurations
- Measure execution speed

# The Exercises

## 3. Compute a Julia set

- Repeatedly apply
$$f_c(z) = z^2 + c,$$
To each number in the complex plane.
  - ▷ If it stays bounded: Julia set
  - ▷ If it diverges: Fatou set
- Colour each pixel by how fast it diverges
- Run for 1024\*1024 pixels in parallel



# Via ssh

- ▶ Find a machine with CUDA and a GPU

- Own machine?
- `ssh -X lxplus-gpu.cern.ch`
- Also `lxplus8-gpu`, `lxplus9-gpu`


- ▶ Git clone

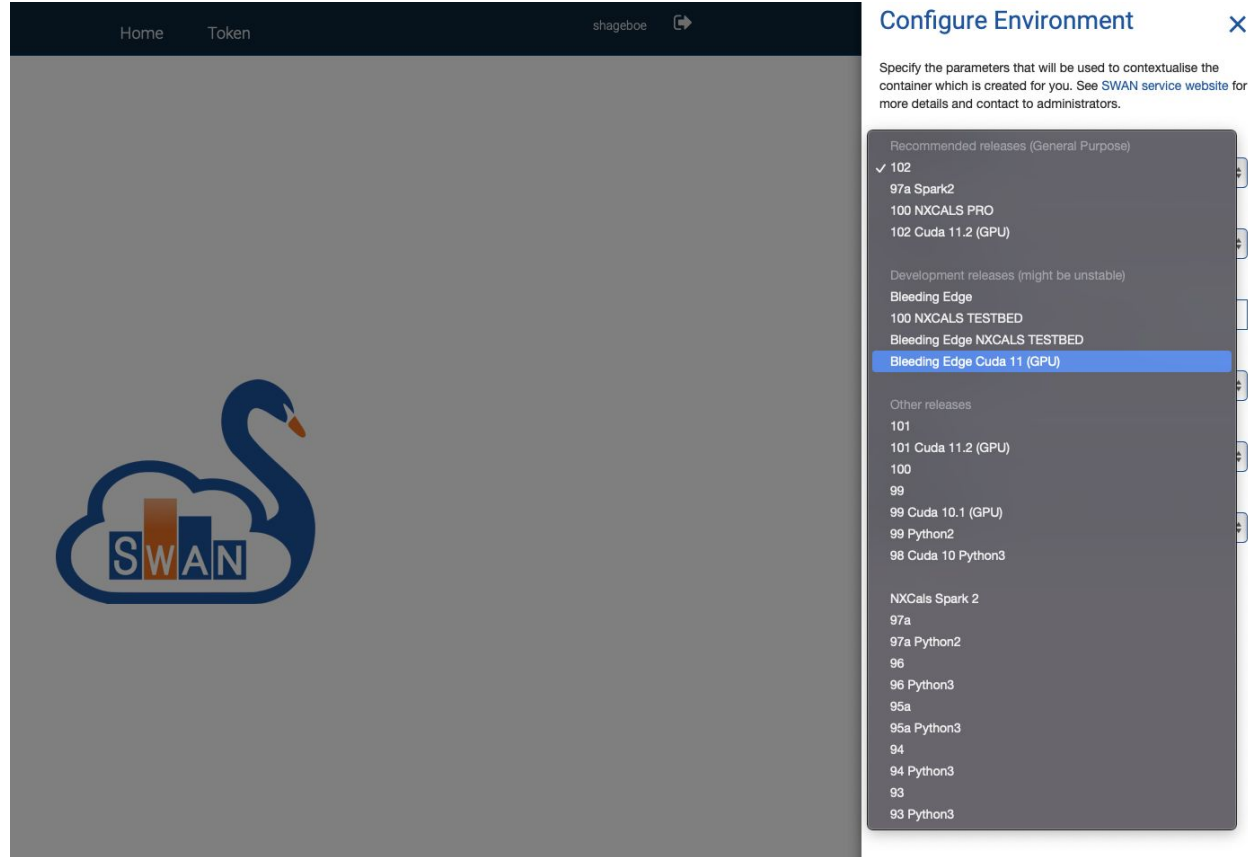
<https://github.com/hageboeck/OpenlabLecture.git>

- ▶ `cd OpenlabLecture`

- ▶ Follow [README.md](#)

# Via SWAN

- ▶ If you don't have a cernbox yet:  
<https://cernbox.cern.ch>
- ▶ Then 
- ▶ Select bleeding edge cuda
- ▶ Use the notebooks "First steps" and "Julia"



The screenshot shows the SWAN web interface. At the top, there are navigation links for 'Home' and 'Token', and a user profile 'shageboe'. The main content area features the SWAN logo, which consists of a blue swan silhouette with a bar chart inside its body. On the right side, a 'Configure Environment' panel is open, displaying a list of environment configurations. The 'Recommended releases (General Purpose)' section includes '102' (selected), '97a Spark2', '100 NXCALS PRO', and '102 Cuda 11.2 (GPU)'. Below this is the 'Development releases (might be unstable)' section, which includes 'Bleeding Edge', '100 NXCALS TESTBED', 'Bleeding Edge NXCALS TESTBED', and 'Bleeding Edge Cuda 11 (GPU)'. The 'Other releases' section lists various versions like '101', '101 Cuda 11.2 (GPU)', '100', '99', '99 Cuda 10.1 (GPU)', '99 Python2', '98 Cuda 10 Python3', 'NXCals Spark 2', '97a', '97a Python2', '96', '96 Python3', '95a', '95a Python3', '94', '94 Python3', '93', and '93 Python3'.