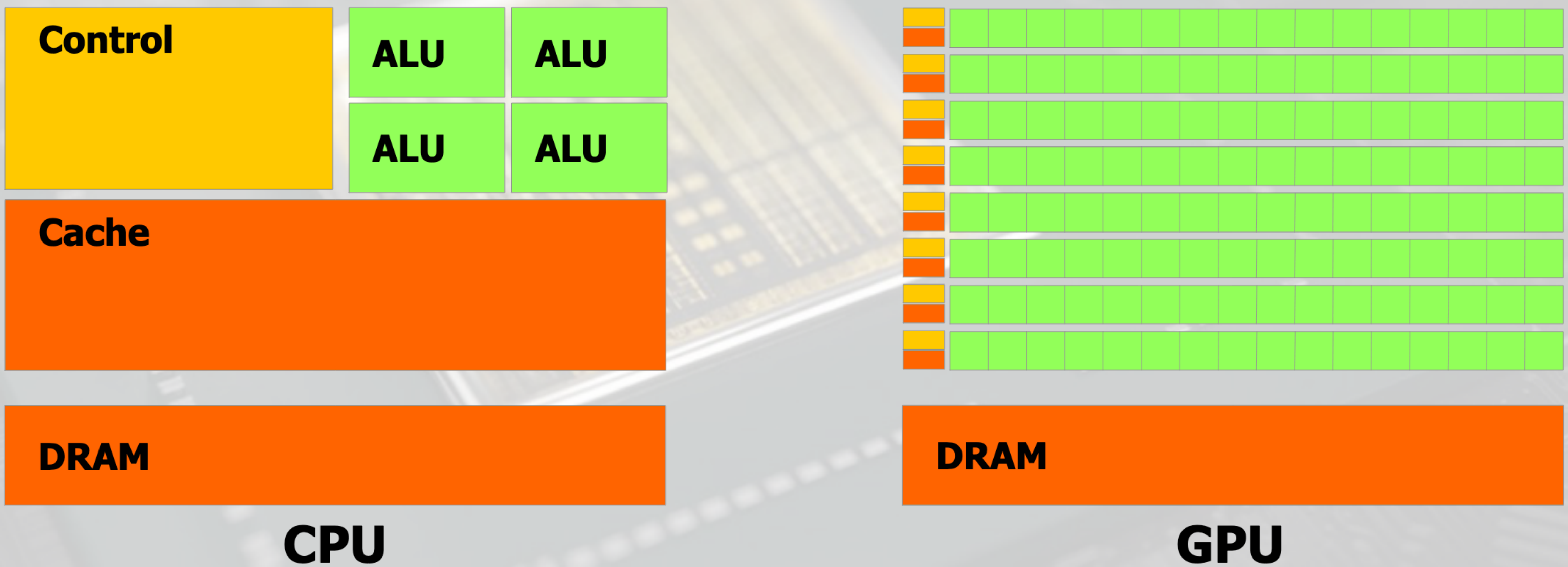
HOW AND WHEN TO USE

GPUS

WHAT IS A GPU?

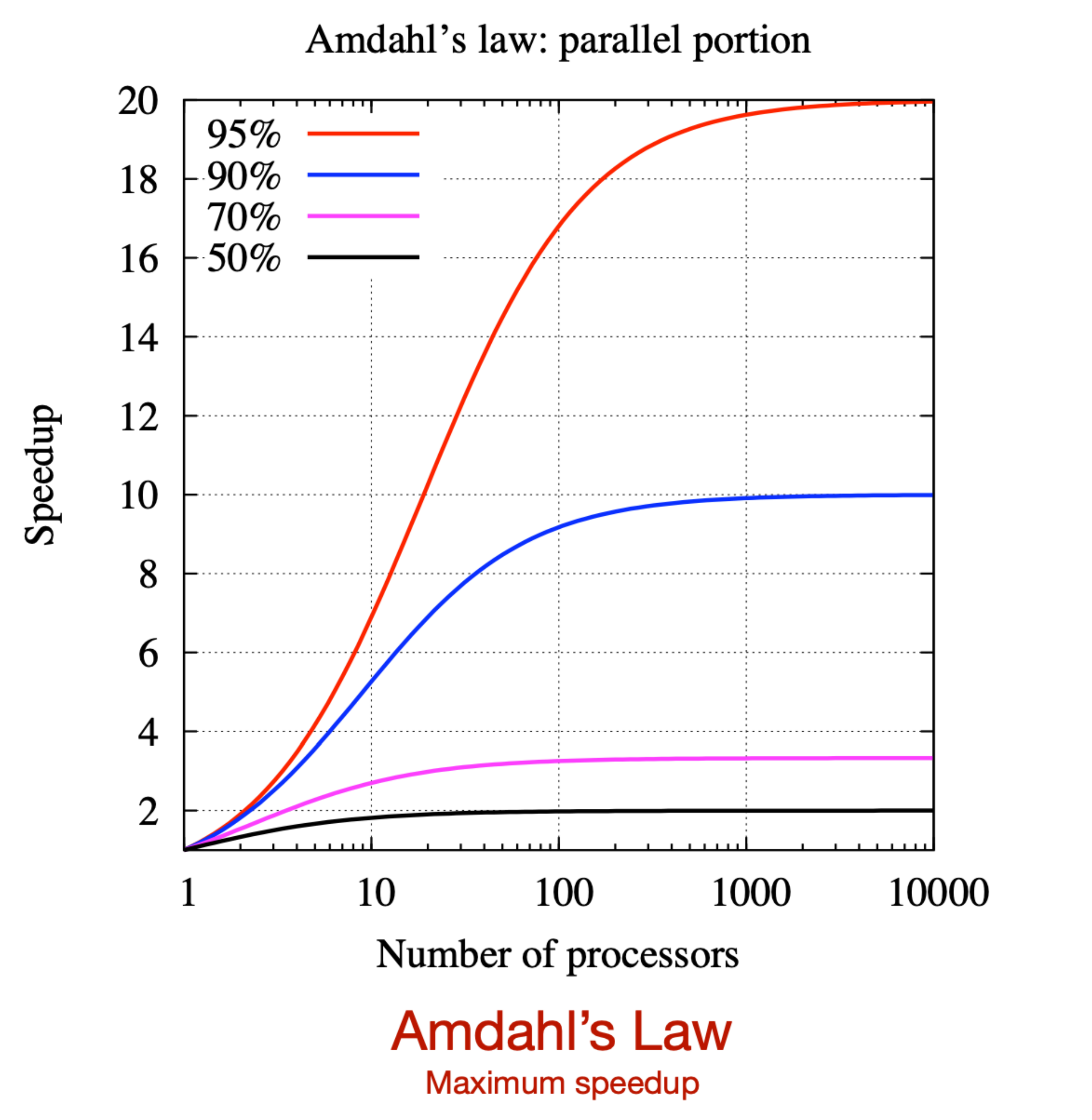


- Few processing cores
- Highly flexible
- Low latency, moderate throughput

- Many processing cores
- Limited flexibility
- High latency, High throughput

AMDAHL'S LAW

- The more of your problem that is parallel the faster a GPU will make it.
- The more data you have to process, the more likely it is to be paralleliseable.
- Good examples: FFTs, particle simulations, linear algebra, etc.
- Bad examples: Unknown problem in CS, does NC=P?



OVERVIEW

- In this tutorial you will learn:
  - How to check the GPUs on a node
  - How to write a CUDA kernel
  - How GPU memory management works
  - How CUDA threads, blocks and grids are arranged
  - Memory access rules
  - Tools that make it all easy

GLOSSARY

- **Host**: The server that hosts the GPU
- **Device**: The GPU accelerator card
- **Kernel**: A program that is executed by the GPU
- **Profiler**: A tool for measuring the performance of a piece of code

HELLO WORLD

- Log into Numerix0
- Add CUDA bin/ to PATH
  - export PATH=$PATH:/usr/local/cuda/bin/
- Make yourself a working directory
- Run **nvidia-smi** to check GPUs
- Open editor of your choice…

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.39       Driver Version: 418.39       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| 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 K20m          On   | 00000000:04:00.0 Off |                    0 |
| N/A   23C    P8    13W / 225W |      0MiB /  4743MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K20m          On   | 00000000:09:00.0 Off |                    0 |
| N/A   29C    P8    14W / 225W |      0MiB /  4743MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla K40m          On   | 00000000:83:00.0 Off |                    0 |
| N/A   26C    P8    21W / 235W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla K40m          On   | 00000000:84:00.0 Off |                    0 |
| N/A   23C    P8    19W / 235W |      0MiB / 11441MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

HELLO WORLD

C

```c
void c_hello(){
    printf("Hello World!\n");
}

int main() {
    c_hello();
    return 0;
}
```

CUDA

```c
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

nvcc -o hello_world hello_world.cu

https://cuda-tutorial.readthedocs.io/en/latest/tutorials/tutorial01/

HELLO WORLD

Execution space specifier
__host__, __device__ or __global__

C

Function

```c
void c_hello(){
    printf("Hello World!\n");
}

int main() {
    c_hello();
    return 0;
}
```

Function call

CUDA

Kernel

```c
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

Kernel launch

https://cuda-tutorial.readthedocs.io/en/latest/tutorials/tutorial01/

GPU ASYNCHRONISITY

- Host and Device code are asynchronous
- Good because GPU can do work at the same time as CPU does work
- CUDA kernel launches return immediately
- It is users responsibility to synchronise and to check errors
- Do this using **cudaDeviceSynchronise()**

```
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    cudaDeviceSynchronise();
    return 0;
}
```

MEMORY MANAGEMENT

- **new**: Allocates memory on host
- **cudaMalloc**: Allocates memory on device
- **cudaMemcpy**: Copies data to/from host/device
- **cudaFree**: Free memory on the device
- **delete**: Free memory on the host

ADDING TWO VECTORS

Worked example

ERROR HANDLING

- CUDA code can fail silently at runtime: VERY BAD, WTF IS HAPPENING, WHY DOES IT NOT WORK!
- Users have responsibility to check for errors
- Many CUDA functions return **cudaError_t** values
- When **error == cudaSuccess** everything is good
- The rest of the time us **cudaGetErrorString()** to find out what went wrong
- Copy */media/scratch/gpu-tutorial/examples/errors/error_checker.cu into all your codes and use the **CUDA_ERROR_CHECK** macro.*

BENCHMARKING

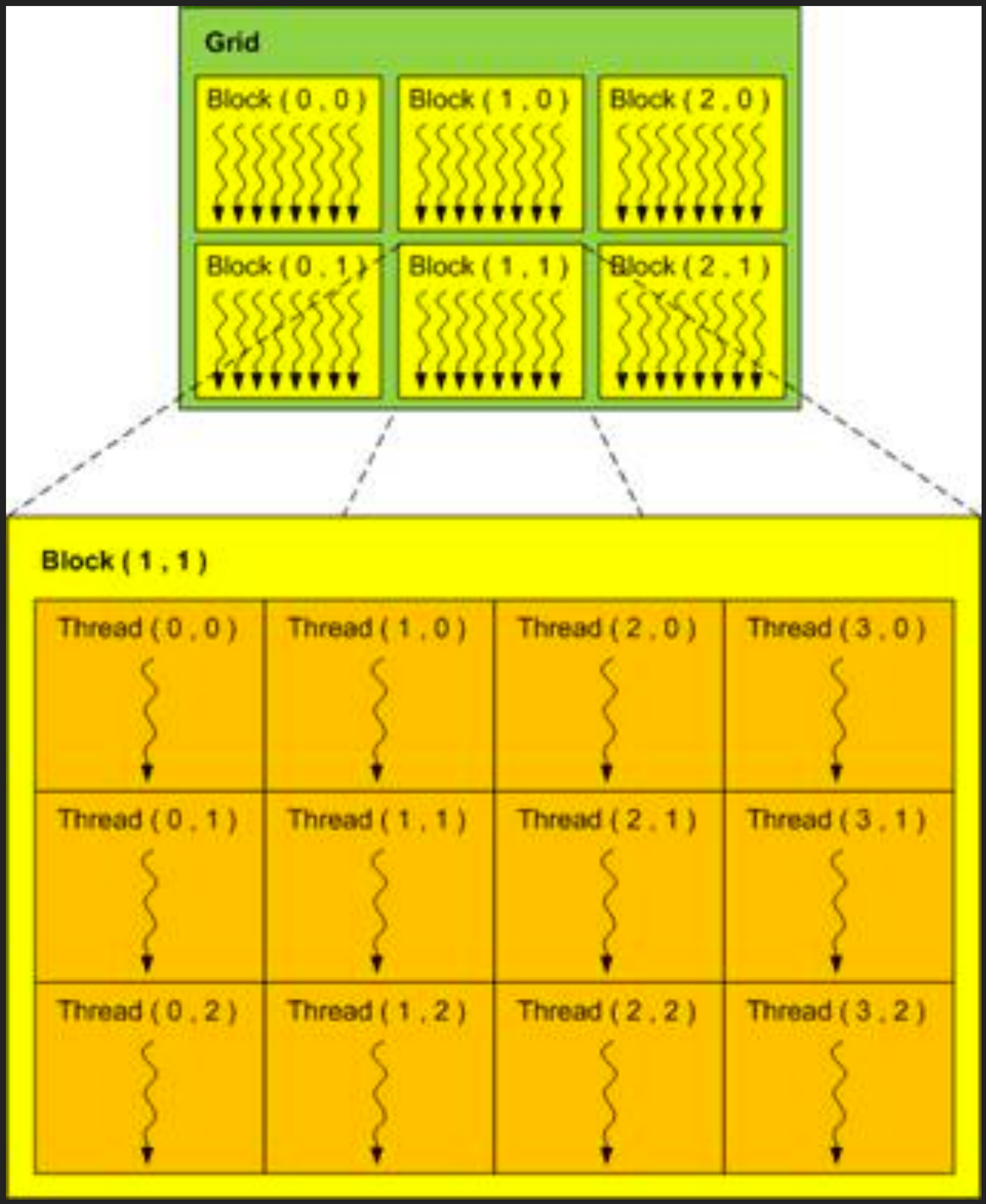
- Use **NVPROF** (command line) or **NVVP** (GUI) to benchmark code

```
(py3) ebarr@numerix0:/media/scratch/gpu-tutorial/examples/hello_world$ nvprof ./hello_world
==1852== NVPROF is profiling process 1852, command: ./hello_world
Hello world!
==1852== Profiling application: ./hello_world
==1852== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  100.71us         1  100.71us  100.71us  100.71us  hello_world(void)
      API calls:   96.71%  313.61ms         1  313.61ms  313.61ms  313.61ms  cudaLaunchKernel
                    1.94%  6.2803ms       388  16.186us     312ns  655.13us  cuDeviceGetAttribute
                    1.13%  3.6717ms         4  917.91us  584.01us  1.2852ms  cuDeviceTotalMem
                    0.21%  665.27us         4  166.32us  132.66us  258.91us  cuDeviceGetName
                    0.01%  22.135us         4  5.5330us  3.9950us  7.5830us  cuDeviceGetPCIBusId
                    0.00%  8.9100us         8  1.1130us     434ns  2.0600us  cuDeviceGet
                    0.00%  4.0870us         3  1.3620us     327ns  2.0800us  cuDeviceGetCount
                    0.00%  2.1250us         4     531ns     443ns     753ns  cuDeviceGetUuid
```

- Try testing the vector addition code…

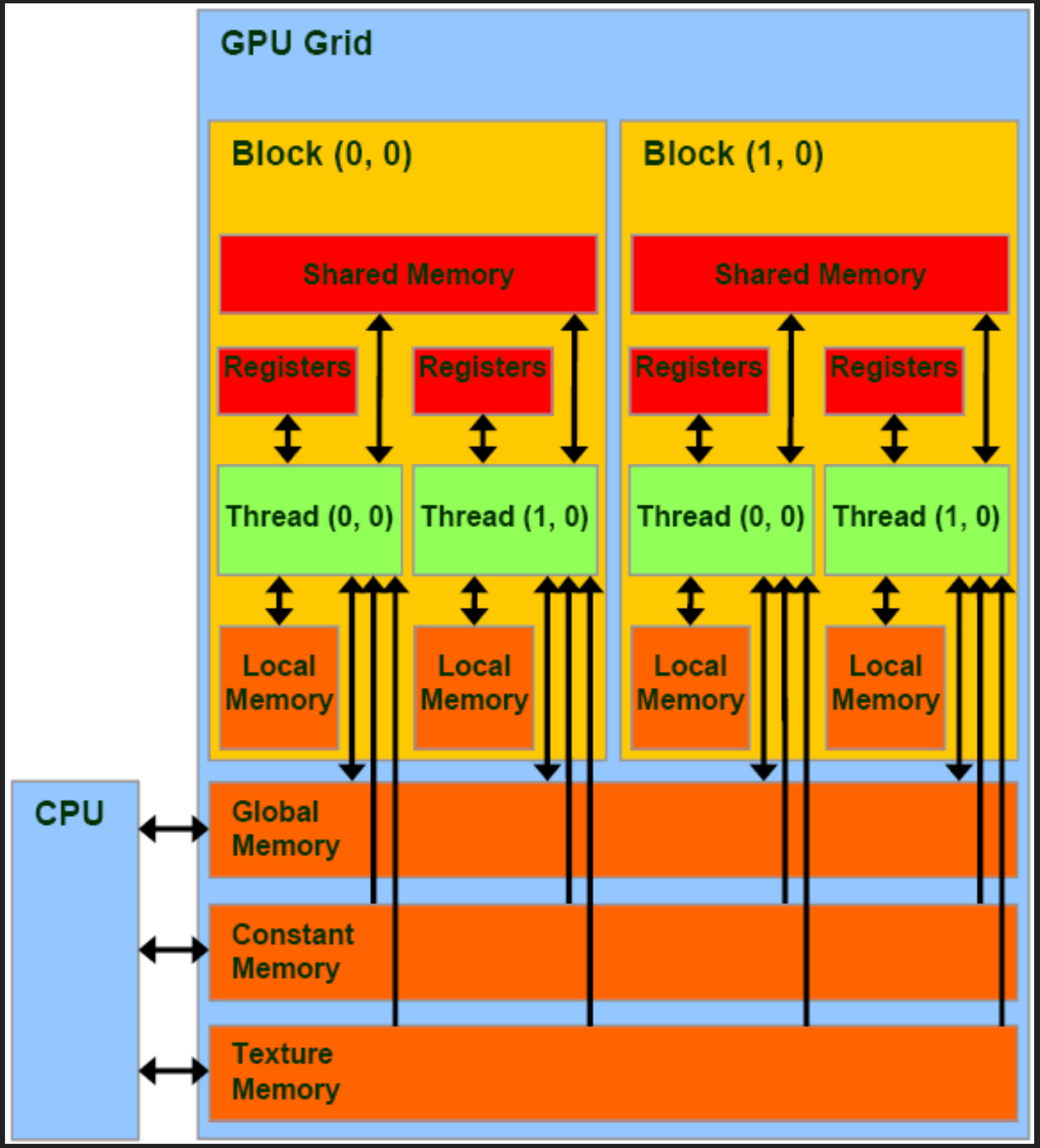
CUDA ARCHITECTURE

- GPU has many processing threads available, but they do not all work independently.
- **Threads** are mapped into **blocks**, which are in turn mapped into **grids**.
- One grid per kernel
- Blocks and grids can be 3D (X, Y, Z indexing)
- We write general code that is parameterised by the thread and block coordinates
- Groups of 32 threads (a **warp**) work in lock-step



MEMORY HIERARCHY

- Different types of memory available (fastest to slowest):
  - **Registers:** 256 kB, thread local
  - **Shared memory:** 64 kB, block local
  - **Constant memory:** 64 KB, global, read-only, broadcast
  - **Texture memory:** Huge, global, read-only, hardware interpolation
  - **Global memory:** Huge, global



MAPPING A CODE TO CUDA THREADS

- Which parts of the code are independent?
- Can the code be broken up into separate tasks?
- Can I do the most work possible per byte of memory at one time?
- Can I write code that doesn't care how many threads or blocks I have?

MAPPING A CODE TO CUDA THREADS

- How do I map the following?

```
__global__
void vector_add(float *out, float *a, float *b, int n)
{
    for(int i = 0; i < n; i++)
    {
        out[i] = a[i] + b[i];
    }
}
```

- **Clue:** CUDA will tell me which thread is executing the code by the following variables:
  - **gridDim.x,** gridDim.y, gridDim.z (how many blocks in each grid axis)
  - **blockIdx.x,** blockIdx.y, blockIdx.z (the block index)
  - **blockDim.x,** blockDim.y, blockDim.z (how many threads in each block axis)
  - **threadIdx.x,** threadIdx.y, threadIdx.z (the block index)
- Consider only the X axis

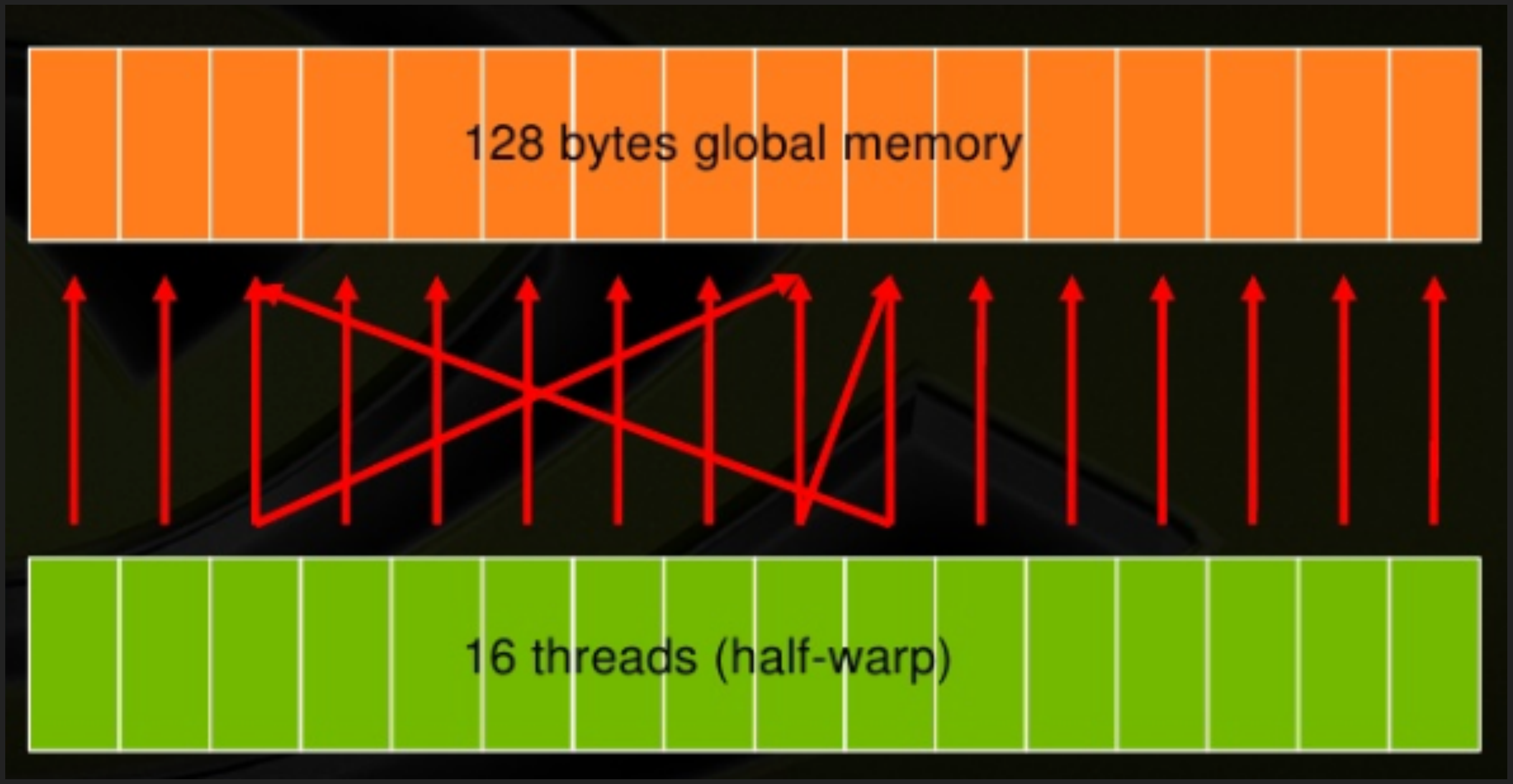
MAPPING A CODE TO CUDA THREADS

```
__global__
void vector_add(float *out, float *a, float *b, int n)
{
  int total_threads = gridDim.x * blockDim.x;
  int n_per_thread = (n / total_threads) + 1;
  int idx = n_per_thread * (blockDim.x * blockIdx.x + threadIdx.x);
  for (int ii = idx;
       (ii < idx + n_per_thread) && (ii < n);
       ++ii)
    {
      out[ii] = a[ii] + b[ii];
    }
}
```

- Here each thread does **n_per_thread** calculations
- Code works, but it has problems: unnecessary calculations, and a **uncoalesced memory access pattern**

MEMORY ACCESS PATTERNS

- CUDA likes it when neighbouring threads read neighbouring data
- Threads in same **half-warp (16 threads)** should try to read data in 32-, 64- or 128-byte aligned cache lane
- Can affect per performance



MAPPING A CODE TO CUDA THREADS

```
__global__
void vector_add(float *out, float *a, float *b, int n)
{
    for (idx = blockDim.x * blockIdx.x + threadIdx.x;
        idx < n;
        idx += gridDim.x * blockDim.x)
    {
        out[idx] = a[idx] + b[idx];
    }
}
```

- Memory access is now **coalesced** (neighbouring threads access neighbouring data)
- Threads still do multiple indices, but without unnecessary extra calculations

KERNEL LAUNCHING

- CUDA uses **<<<>>>** triple angle bracket notation for kernel launches
- Arguments are:
    1. Grid dimensions
    2. Block dimensions
    3. Size of desired dynamic shared memory (optional)
    4. Stream ID (optional)
- Dimensions can be described with a **dim3** struct
    - e.g. **<<<dim3(4,4,4), dim3(5,5,5)>>>** would give a 4 by 4 by 4 grid of blocks (64), each with 5 by 5 by 5 threads (125)
- Maximum number of threads per block is 1024 (there are also limits for each dimension and the same for blocks)

KERNEL LAUNCHING

- As we have written our *vector_add* code to be thread-block agnostic, we can choose any combination of threads and blocks (but only x-axis).
  - **`vector_add<<<1024,128>>>(d_out, d_a, d_b, N);`**
  - **`vector_add<<<dim3(1024,1,1), dim3(128,1,1)>>>(d_out, d_a, d_b, N);`**
- After the kernel call we can synchronise to wait for it to finish (and we should check the error code returned)
  - **`CUDA_ERROR_CHECK(cudaDeviceSynchronize());`**

TOOLS

- Lots of libraries for CUDA:
  - Mathmatical functions with **CUDA Math Library**
  - Fast Fourier Transforms with **cuFFT**
  - Deep Neural Networks with **cuDNN**
  - Linear algebra with **cuBLAS**, **cuSPARSE**, **cuSOLVER** and **cuTENSOR**
  - Random number generators with **cuRAND**

MAKING THINGS EASY

- Lots of high-level language abstractions:
  - **Thrust**: C++ STL-like library that provides easy interface for people already familiar with C++
  - **PyCUDA**: Python wrappers for CUDA Driver API that provide extensive functionality with the ability to embed raw CUDA code that can be **JIT** compiled.

THRUST: VECTOR ADD

```
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/transform.h>
#define N 1000000

int main()
{
  thrust::host_vector<float> a(N);
  thrust::host_vector<float> b(N);
  thrust::device_vector<float> d_a = a;
  thrust::device_vector<float> d_b = b;
  thrust::device_vector<float> d_out(N);
  thrust::transform(d_a.begin(), d_a.end(), d_b.begin(),
                    d_out.begin(), thrust::plus<float>());
  thrust::host_vector<float> out = d_out;
  return 0;
}
```

https://docs.nvidia.com/cuda/thrust/index.html

PYCUDA: VECTOR ADD

```python
import pycuda.gpuarray as gpuarray
import pycuda.driver as cuda
import pycuda.autoinit
import numpy as np

a = np.random.normal(0, 1, 1000000)
b = np.random.normal(0, 1, 1000000)
a_gpu = gpuarray.to_gpu(a.astype(np.float32))
b_gpu = gpuarray.to_gpu(b.astype(np.float32))
a_plus_b = (a_gpu + b_gpu).get()
```

https://documen.tician.de/pycuda/tutorial.html