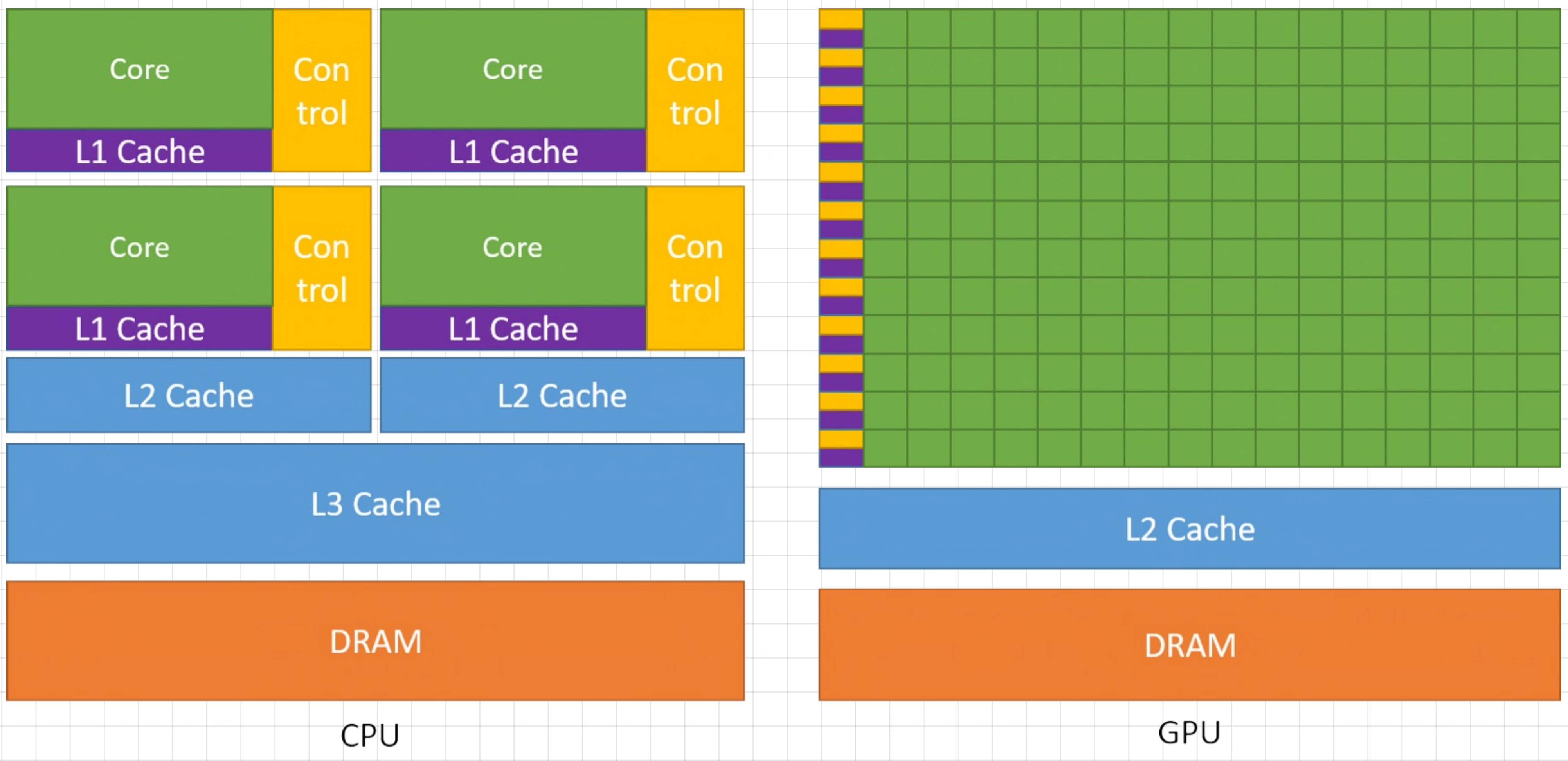
E-Portfolio Nvidia Cuda Script

Agenda

1.1 The Benefits of using GPUs

- GPU provides much higher instruction troughput and memory bandwith
- similar price and power envelope
- Designed with different goals:
  - CPU is designed to excel at executing a sequence of operations (in a thread) as fast as possible
  - GPU is designed to excel at executing thousands of sequences in parrallel
    ↳ specialized for highly parrallel computations
    ↳ more transistors are devoted to data processing rather than data caching and flow control
      ↳ results in higher memory access latencies
      ↳ gets hidden by the raw computational power of the GPU

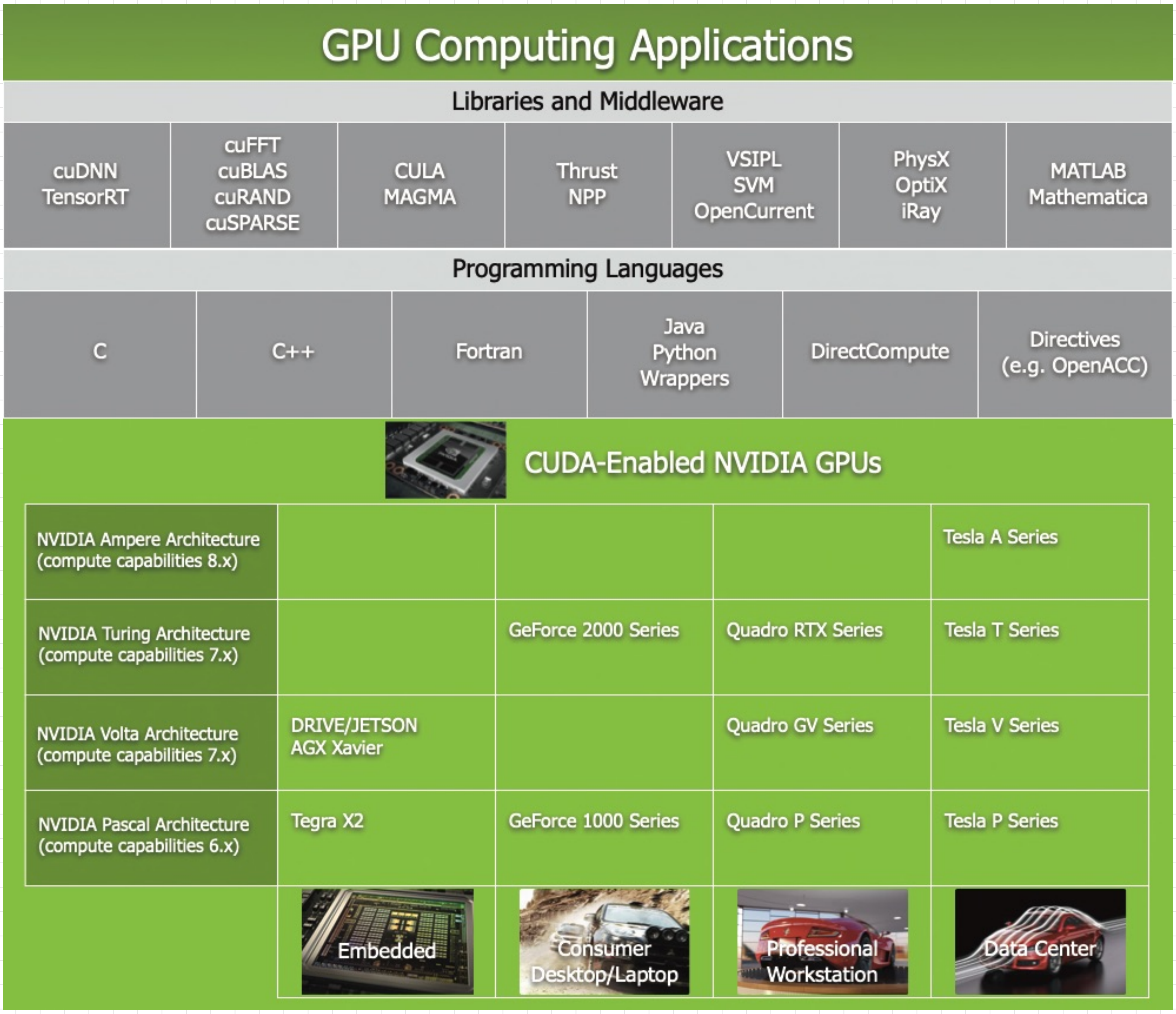


1.2 Performance Demo

- Matrix Multiplication Cuda vs C++

1.3 Cuda - Short Overview

- Introduced in 2006
- is a general purpose parallel computing plattform and programming model
- uses the parralel Compute engine of Nvidia GPUs



GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

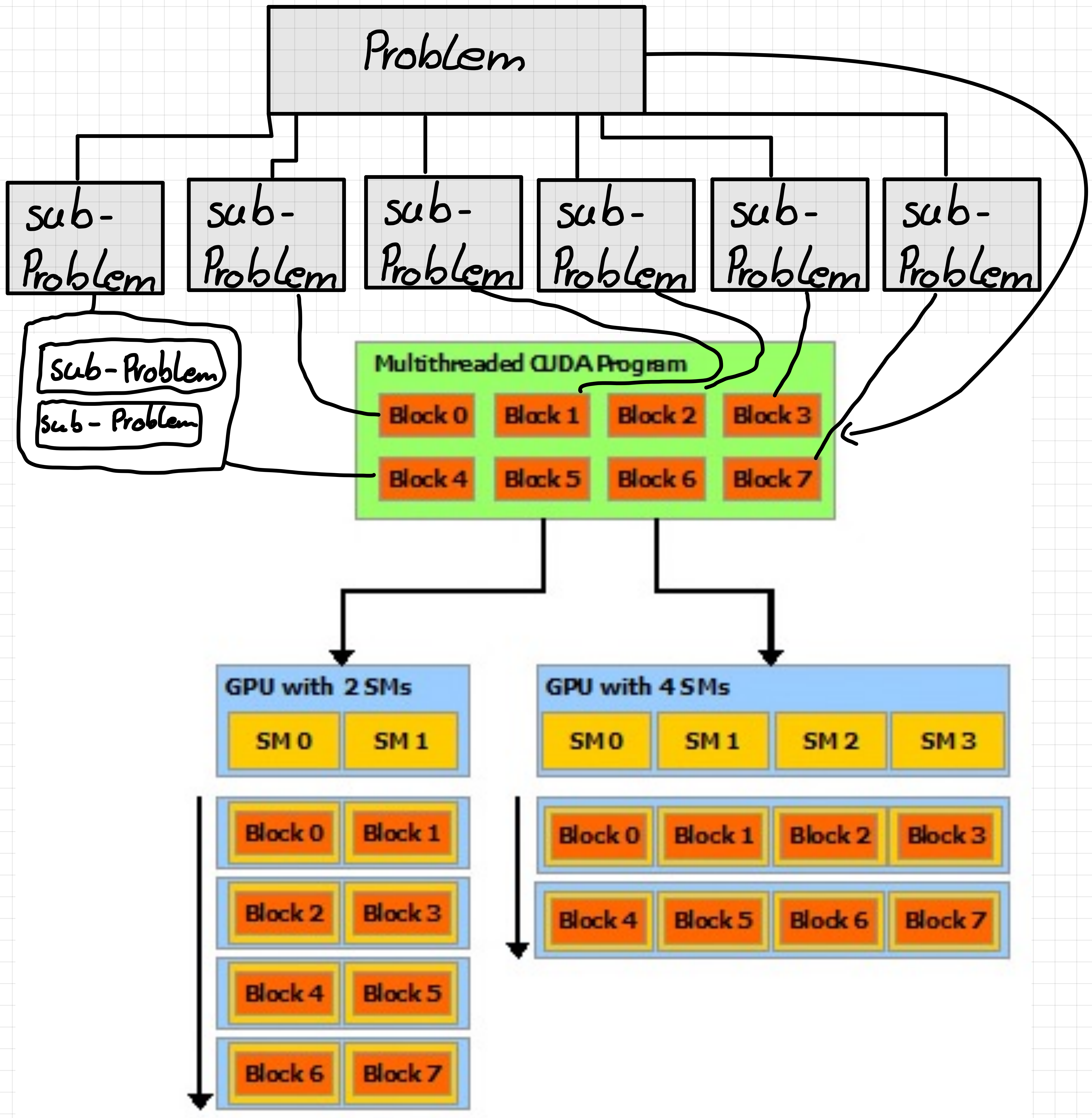
Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
	Embedded	Consumer Desktop/Laptop	Professional Workstation	Data Center

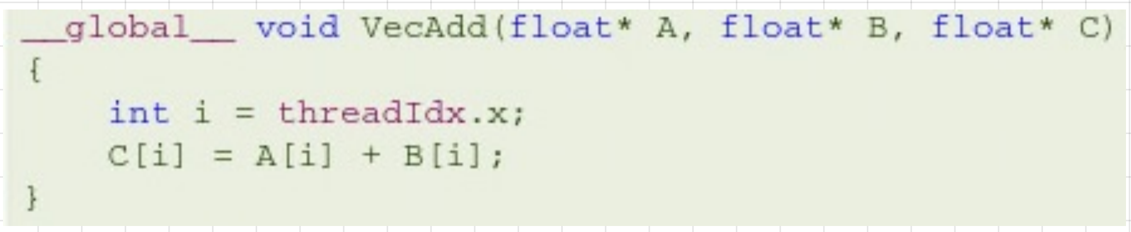
2.1 Programming Model



- Problem gets divided into sub-Problems wich can be solved in Blocks of threads
  ↳ these blocks solve these sub problems by allowing threads to cooperate
- blocks are scaled automatically over the number of SMs

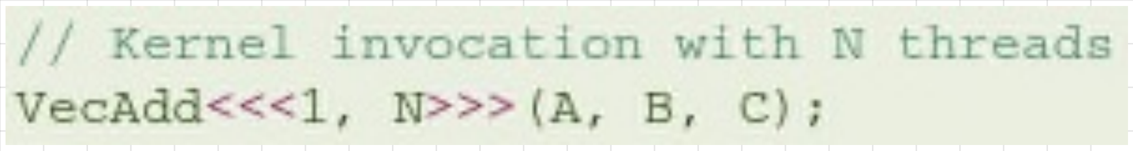
3.1 Kernels

- A Kernel represents the smallest possible sub Problem It is a C++ function wich gets executed n times by n different Cuda threads.
- It is defined by using the __global__ declaration specifier example:



```
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

- A Kernel can be executed by using the execution configuration Syntax:

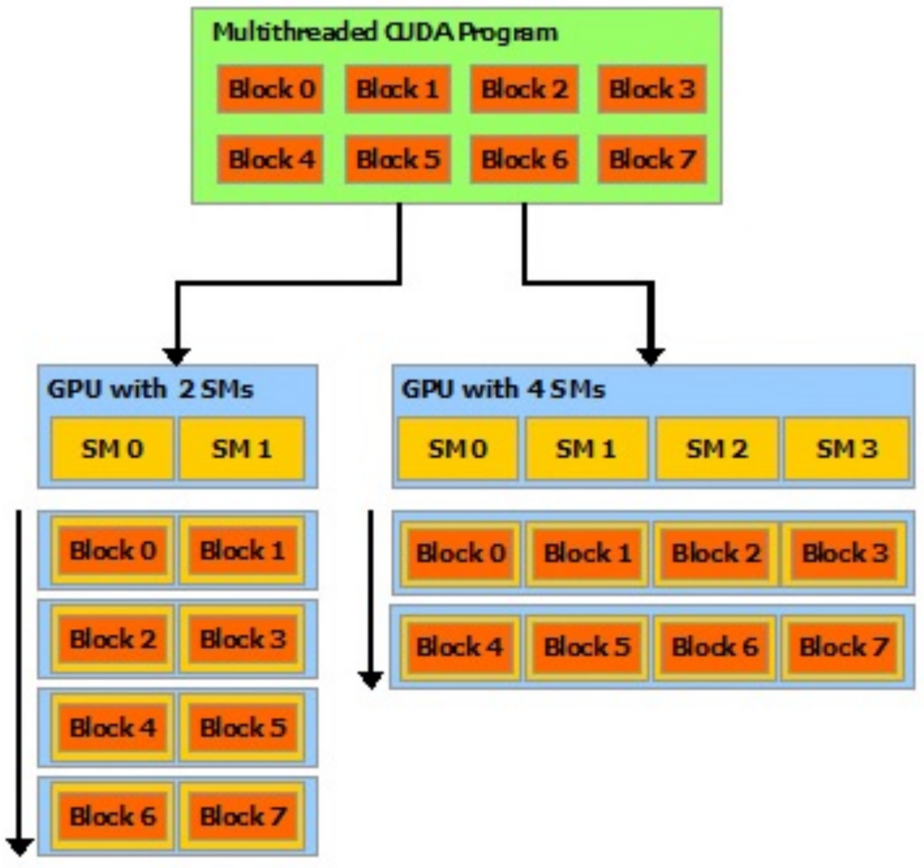


```
// Kernel invocation with N threads
VecAdd<<<1, N>>>(A, B, C);
```

- Each thread that executes the Kernel is given a unique threadID, that can be accessed through build in variables: int i = threadIdx.x

3.2. Thread Hierarchy

- threadIdx is a 3 component Vector
  - ↳ threads can be identified using a 1-, 2-, 3-dimensional thread Index forming a block of threads -> a thread Block
    - ↳ provides a natural way to invoke computatio across the elements in a domain



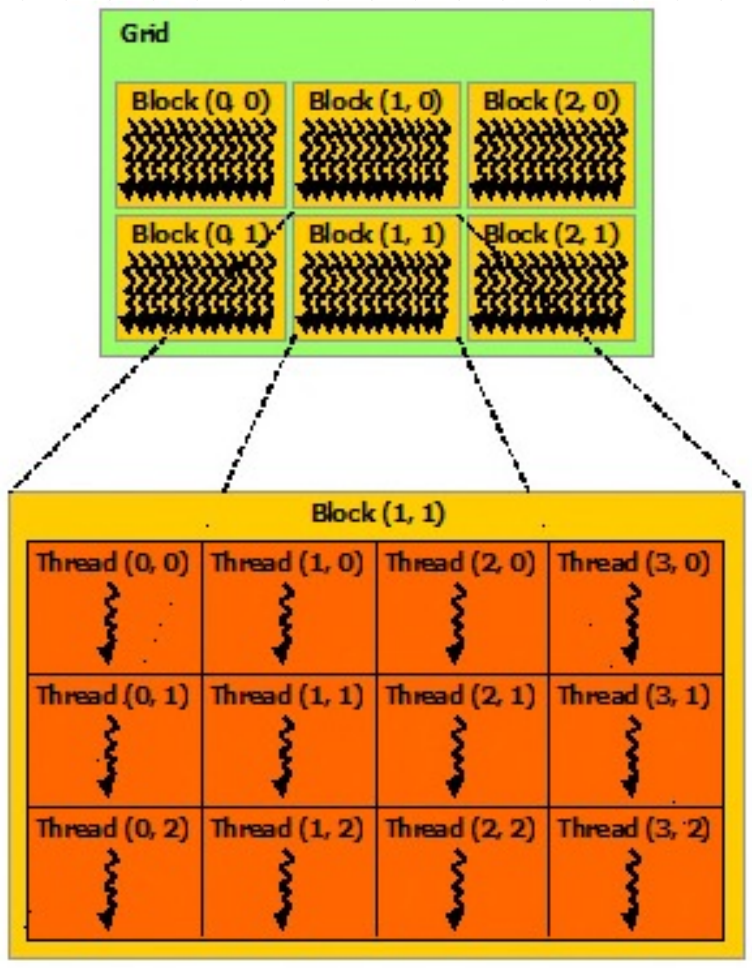
RTX 2070super
- 40 SM
- 2560 Cuda Cores
-> 64 per SM

- Number of threads in a Block is set by the user
- However to take full advantage of your GPU you should calculate the block size and the number of blocks to use.
- To calculate the block size / block Dim you divide the number of Cuda Cores by the number of streaming Multiprocessors (SMs), e.g. for a RTX 2070 super : blockSize = 2560 / 40 = 64

- After that you'll have to calculate the number of blocks to use for this you can use the following formular:

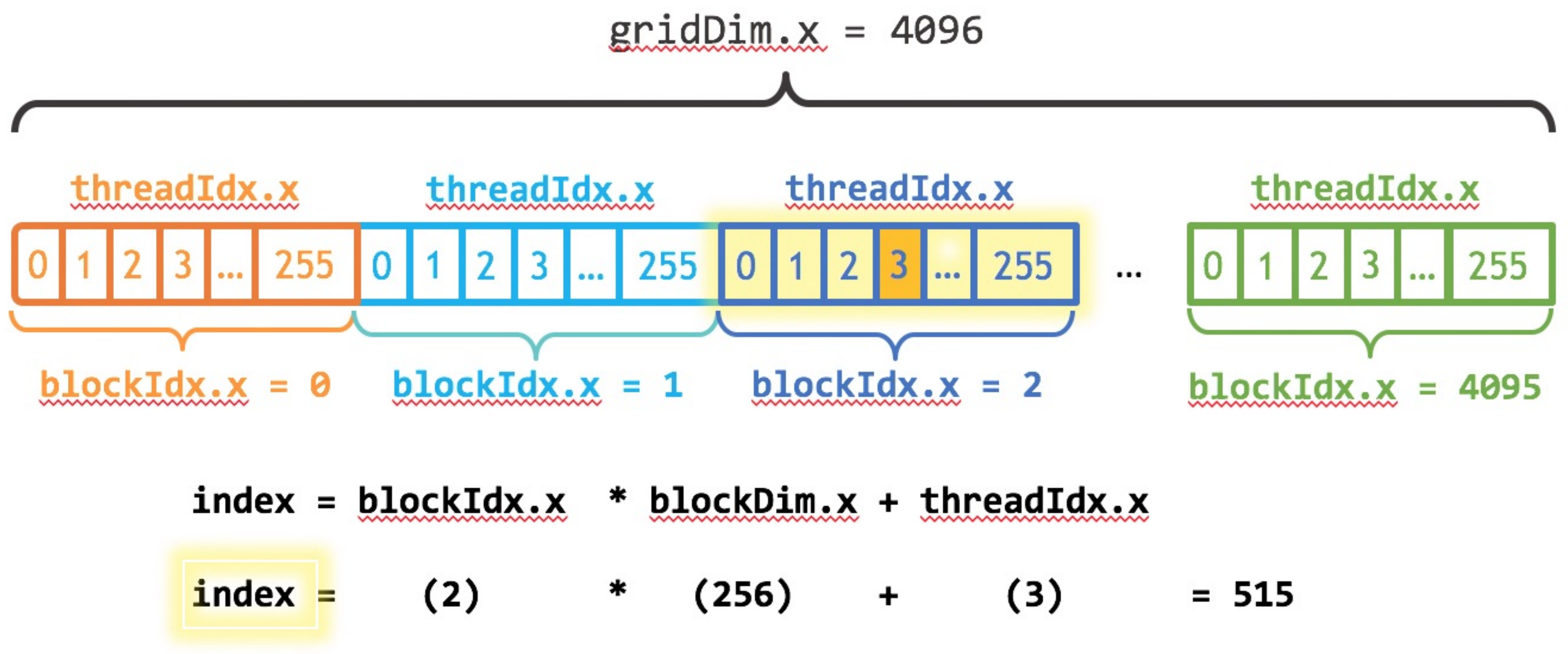
  numBlocks = (N + blockSize - 1) / blockSize

  with N = total Number of threads you need in your programm
- The resulting number of Blocks usually exceeds the number of available SMs on the GPU
- These Blocks get organized in a Grid, the number of blocks per Grid and the number of threads per block gets specified in the <<< >>> syntax



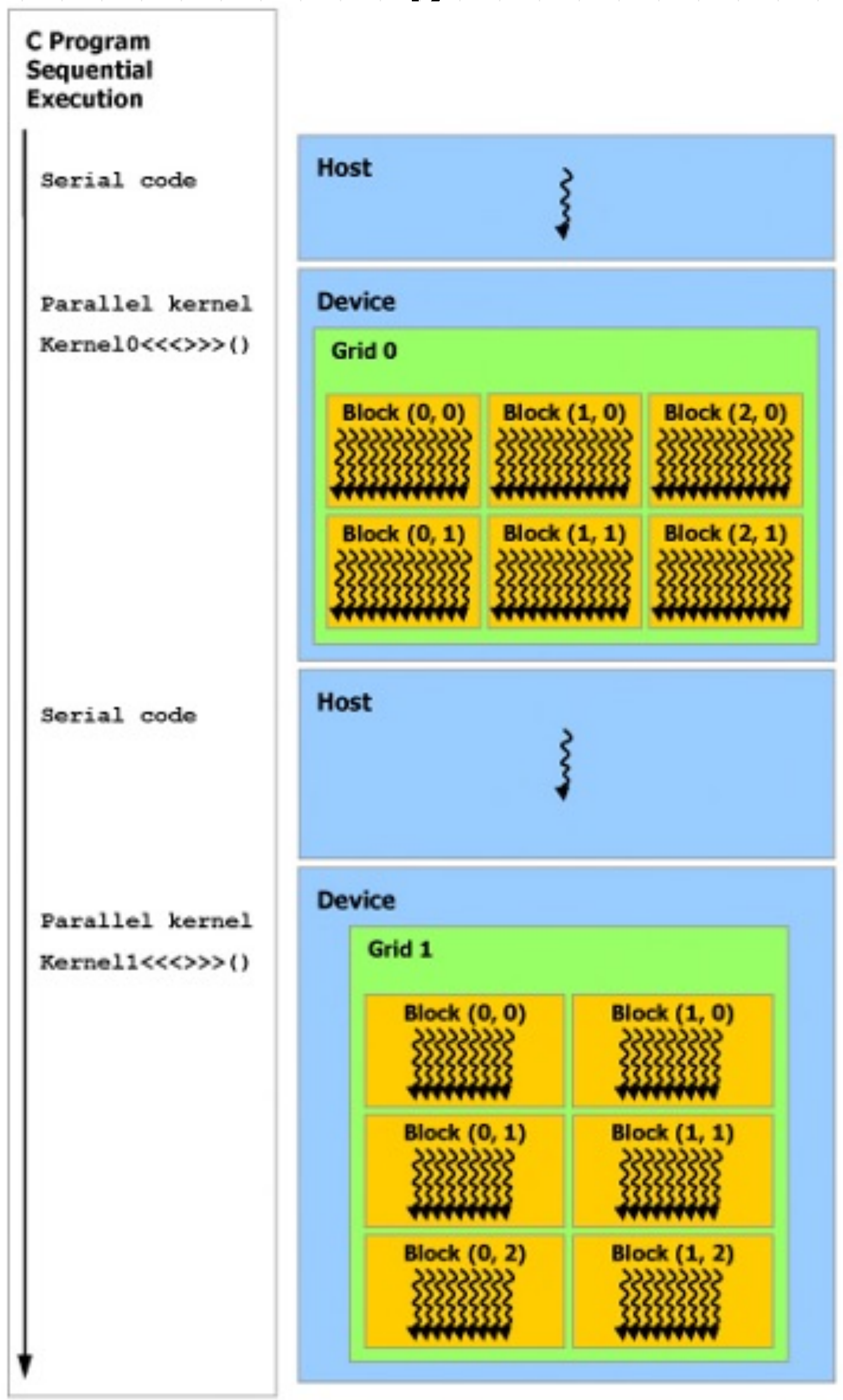
- besides the threadIdx, which represents the thread Index inside a Block, there is another built-in variable which represents the number of threads per block: blockDim and the current block ID: blockIdx

This allows us to index elements as follows :



3.3 Heterogeneous Programming

- The Cuda Programming Model assumes that the Cuda threads get executed on a physically seperate device that operates as a coprocessor to the host running the C++ Programm, e.g: Kernels execute on a GPU and the C++ programm on the CPU.
- It also assumes that both devices maintain their own seperate memory spaces in DRAM, which is refferved to as host memory and device memory. Therefor in the host programm device memory has to be allocated and deallocated and data needs to be transferred through calls to the Cuda Runtime



4.1 Installation

- Download & install the Cuda Toolkit from Nvidia Cuda Zone

4.2 IDEs & Tools

IDEs :
- Nsight Studio (eclipse based) $\rightarrow$ with debug
- CLion $\rightarrow$ without debugging tools
- Visual Studio $\rightarrow$ with debugging tools

Tools:

- Nvidia Visual Profiler: tool to run performance analysis on Cuda Kernels

5. Programming Demo

- Matrix Addition in C++

6. More Info

- Nvidia Cuda Zone
- Nvidia Developer Programm
- docs.nvidia.com