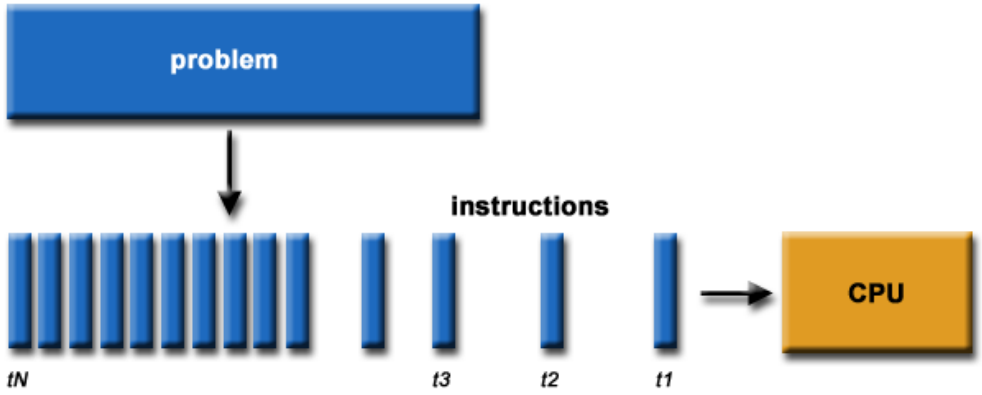
Parallel Programming using CUDA

Traditional Computing

Von Neumann architecture: instructions are sent from memory to the CPU

Serial execution: Instructions are executed one after another on a single Central Processing Unit (CPU)



Problems:

- More expensive to produce
- More expensive to run
- Bus speed limitation

Parallel Computing

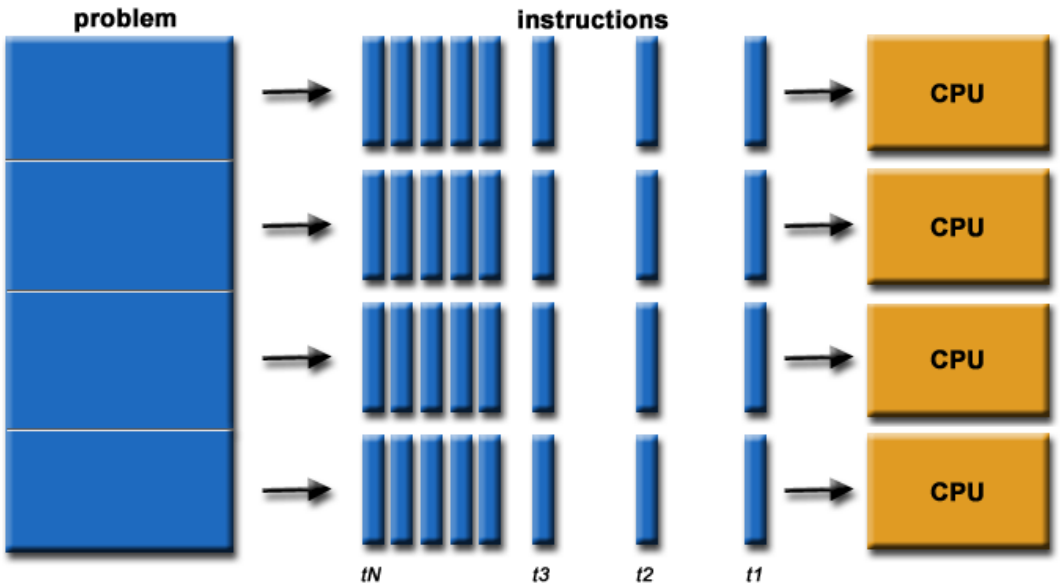
Official-sounding definition: The simultaneous use of multiple compute resources to solve a computational problem.

Benefits:

- Economical – requires less power and cheaper to produce
- Better performance – bus/bottleneck issue

Limitations:

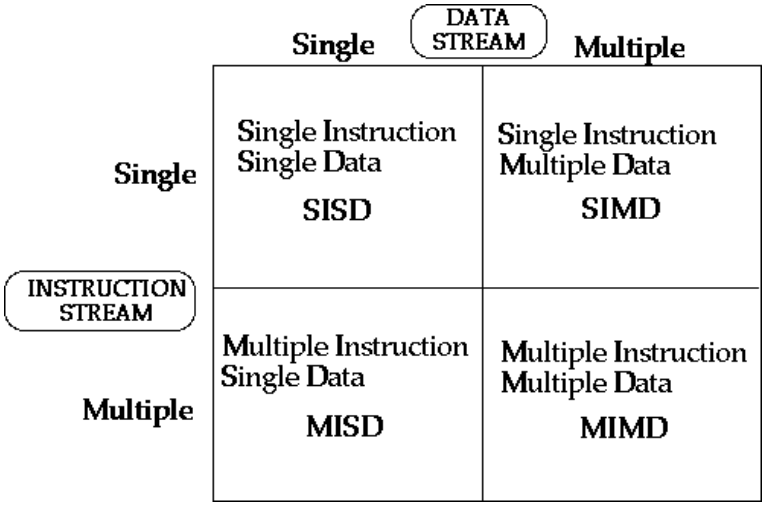
- New architecture – Von Neumann is all we know!
- New debugging difficulties – cache consistency issue



Flynn's Taxonomy

Classification of computer architectures, proposed by Michael J. Flynn

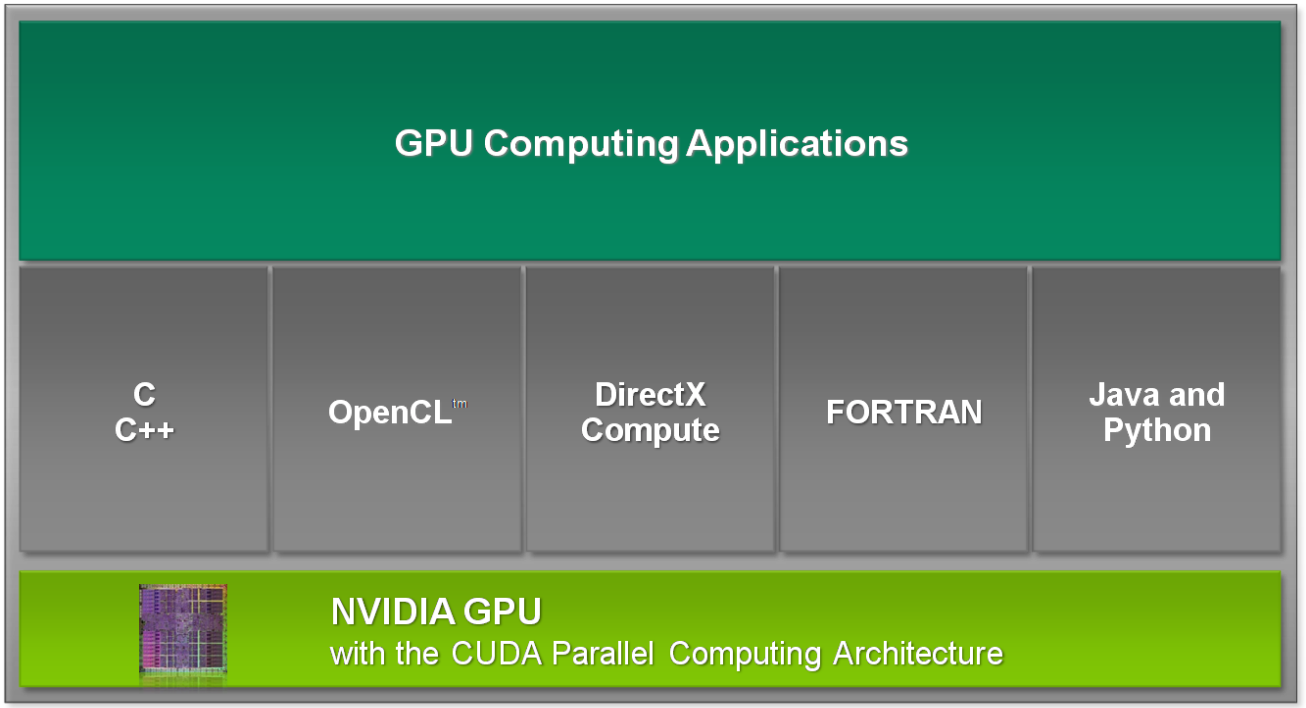
- SISD – traditional serial architecture in computers.
- SIMD – parallel computer. One instruction is executed many times with different data (think of a for loop indexing through an array)
- MISD - Each processing unit operates on the data independently via independent instruction streams. Not really used in parallel
- MIMD – Fully parallel and the most common form of parallel computing.



Enter CUDA

CUDA is NVIDIA's general purpose parallel computing architecture .

- designed for calculation-intensive computation on GPU hardware
- CUDA is not a language, it is an API
- we will mostly concentrate on the C implementation of CUDA

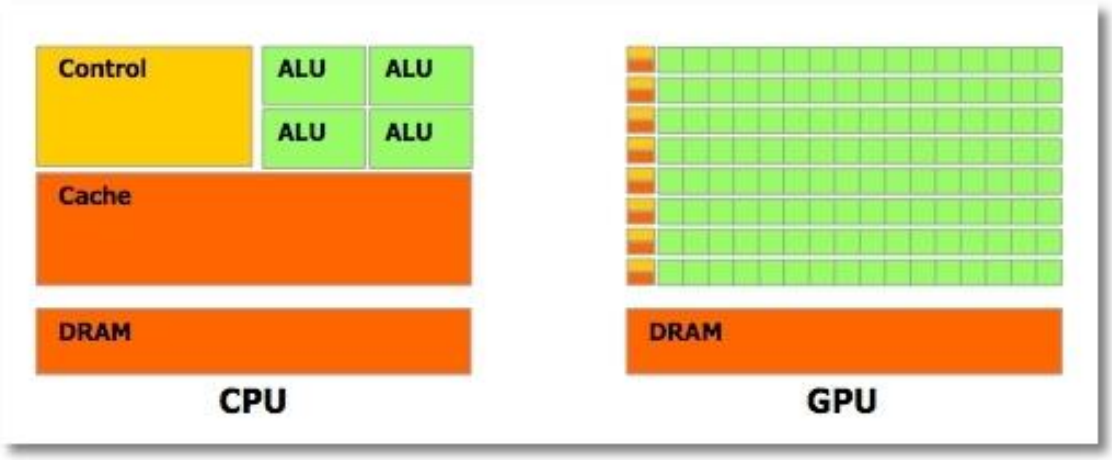


What is GPGPU ?

- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

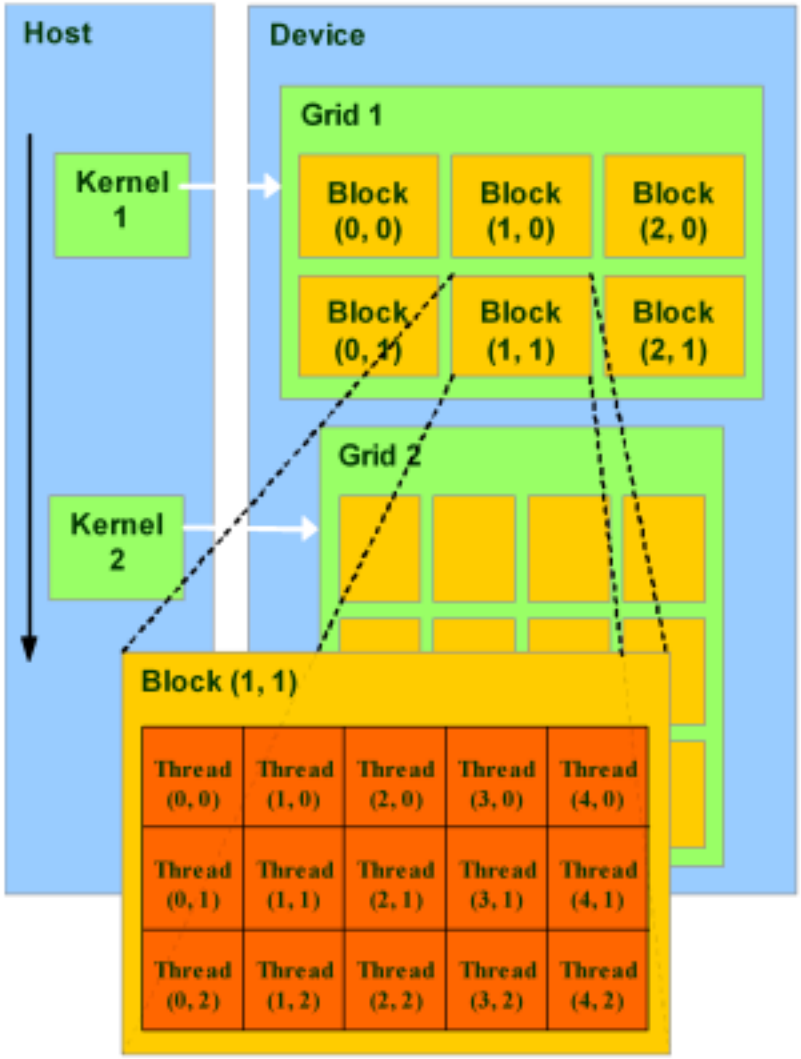
CUDA Goals

- Scale code to hundreds of cores running thousands of threads
- The task runs on the gpu independently from the cpu



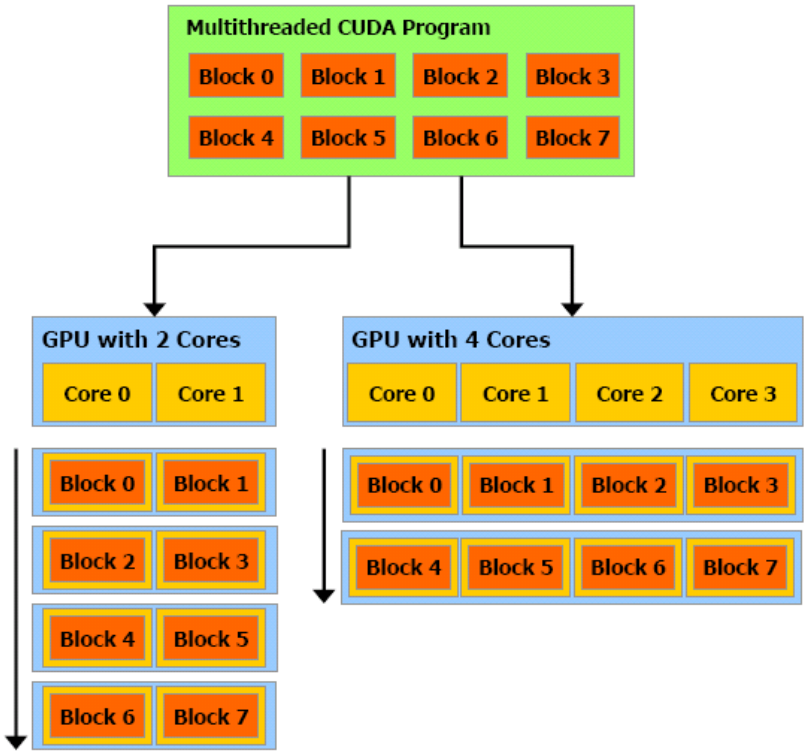
CUDA Structure

- Threads are grouped into thread blocks
- Blocks are grouped into a single grid
- The grid is executed on the GPU as a kernel



Scalability

- Blocks map to cores on the GPU
- Allows for portability when changing hardware



Terms and Concepts

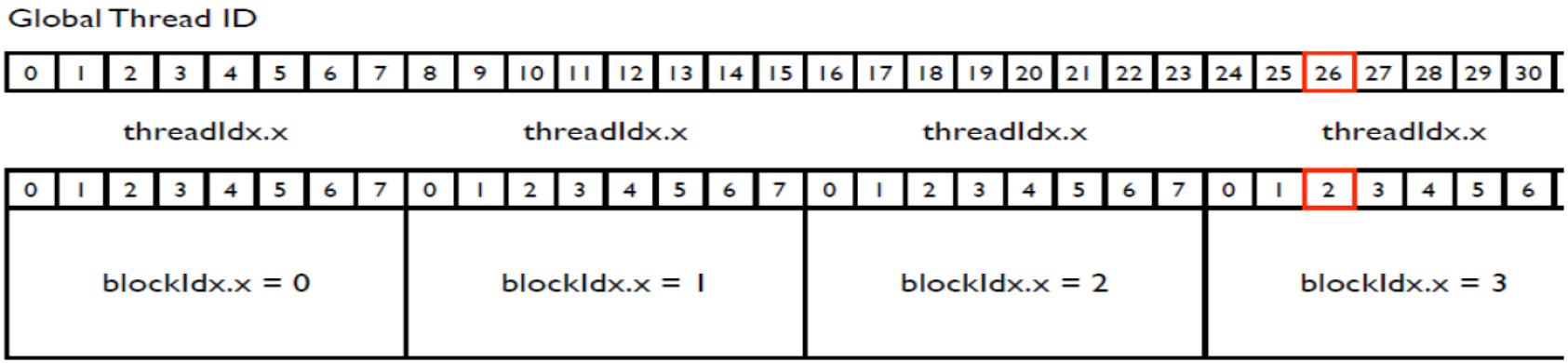
Each block and thread has a unique id within a block.

- threadIdx – identifier for a thread
- blockIdx – identifier for a block
- blockDim – size of the block

Unique thread id:

(blockIdx*blockDim)+threadIdx.x

Terms and Concepts

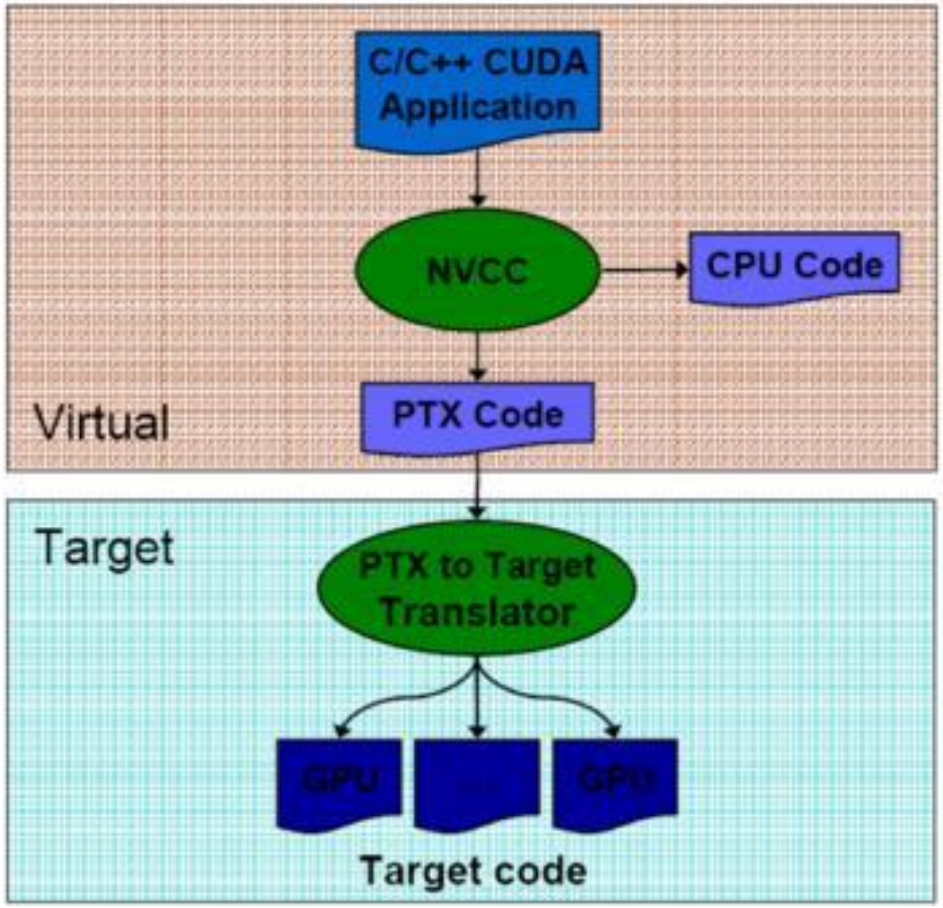


- Assume a hypothetical 1D grid and 1D block architecture: 4 blocks, each with 8 threads.

- For Global Thread ID 26:
  - gridDim.x = 4 x 1
  - blockDim.x = 8 x 1
  - Global Thread ID = blockIdx.x * blockDim.x + threadIdx.x
  - = 3 x 8 + 2 = 26

NVCC compiler

- Compiles C or PTX code (CUDA instruction set architecture)
- Compiles to either PTX code or binary (cubin object)



Development: Basic Idea

1. Allocate equal size of memory for both host and device
2. Transfer data from host to device
3. Execute kernel to compute on data
4. Transfer data back to host

Kernel Function Qualifiers

- __device__
- __global__
- __host__

Example in C:

CPU program

void increment_cpu(float *a, float b, int N)

CUDA program

__global__ void increment_gpu(float *a, float b, int N)

Variable Type Qualifiers

- Specify how a variable is stored in memory
- __device__
- __shared__
- __constant__

Example:

```
__global__ void increment_gpu(float *a, float b, int N)
{
    __shared__ float shared[];
}
```

Calling the Kernel

- Calling a kernel function is much different from calling a regular function

```
Void main(){
Int blocks = 256;
Int threadsperblock = 512;
    mycudafunc<<<blocks,threadsperblock>>>(some parameter);
}
```

GPU Memory Allocation / Release

Host (CPU) manages GPU memory:

- cudaMalloc (void ** pointer, size_t nbytes)
- cudaMemset (void * pointer, int value, size_t count);
- cudaFree (void* pointer)

```
Void main(){
    int n = 1024;
    int nbytes = 1024*sizeof(int);
    int * d_a = 0;
    cudaMalloc( (void**)&d_a,  nbytes );
    cudaMemset( d_a, 0, nbytes);
    cudaFree(d_a);
}
```

Memory Transfer

cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);

- returns after the copy is complete blocks CPU
- thread doesn't start copying until previous CUDA calls complete

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice