Massively Parallel Computing with CUDA

Antonino Tumeo
Politecnico di Milano

“GPUs have evolved to the point where many real world applications are easily implemented on them and run significantly faster than on multi-core systems.

Future computing architectures will be hybrid systems with parallel-core GPUs working in tandem with multi-core CPUs.”

Jack Dongarra
Professor, University of Tennessee; Author of “Linpack”

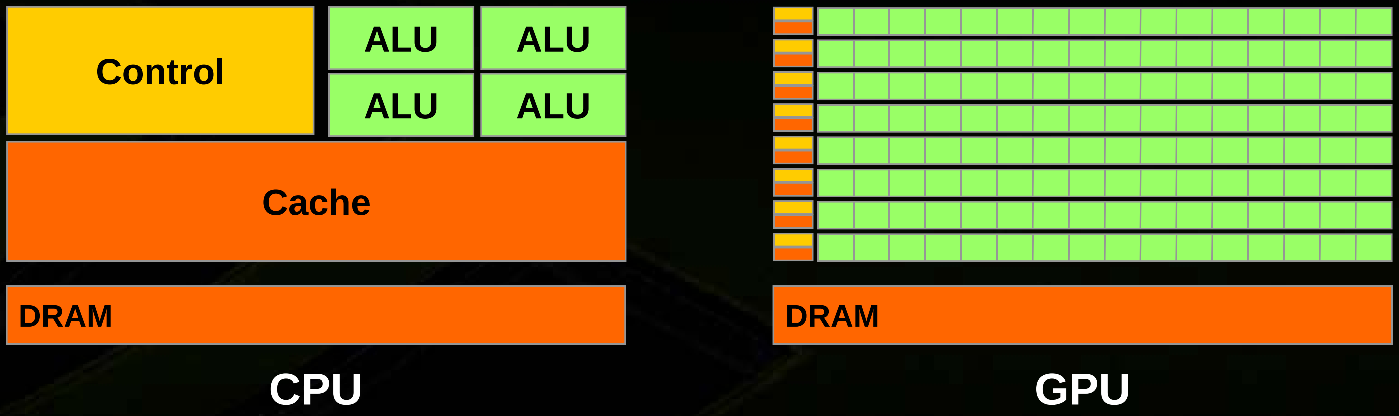
Why Use the GPU?

- The GPU has evolved into a very flexible and powerful processor:
  - It's programmable using high-level languages
  - It supports 32-bit and 64-bit floating point IEEE-754 precision
  - It offers lots of GFLOPS:

- GPU in every PC and workstation

What is behind such an Evolution?

- The GPU is specialized for compute-intensive, highly parallel computation (exactly what graphics rendering is about)
  - So, more transistors can be devoted to data processing rather than data caching and flow control



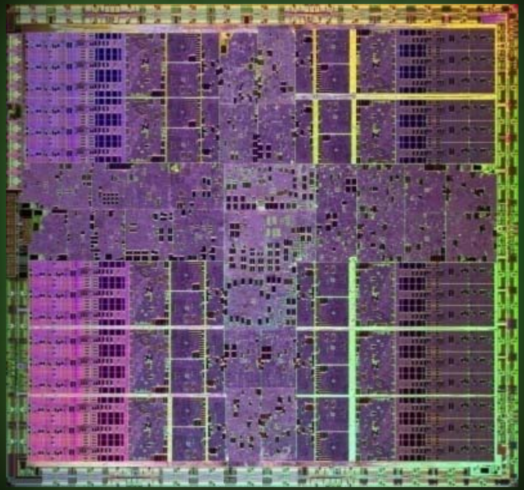
- The fast-growing video game industry exerts strong economic pressure that forces constant innovation

GPUs

- Each NVIDIA GPU has 240 parallel cores
- Within each core
  - Floating point unit
  - Logic unit (add, sub, mul, madd)
  - Move, compare unit
  - Branch unit
- Cores managed by thread manager
  - Thread manager can spawn and manage 12,000+ threads per core
  - Zero overhead thread switching

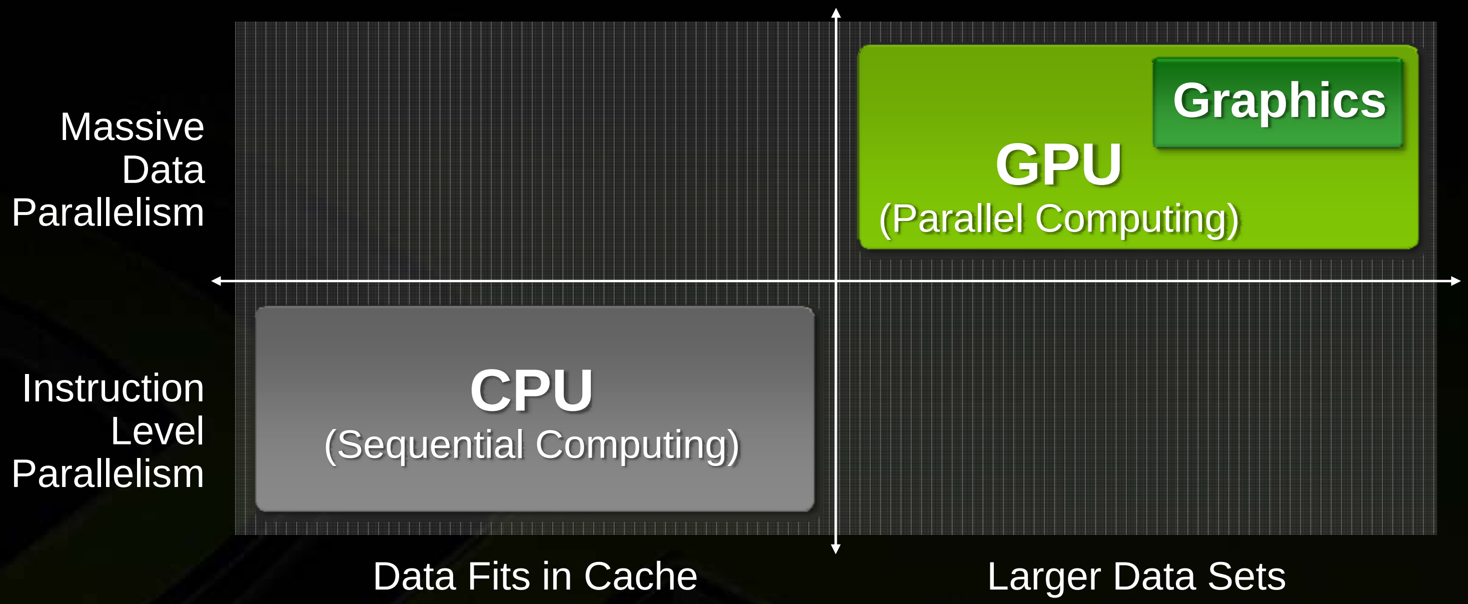
NVIDIA GPU

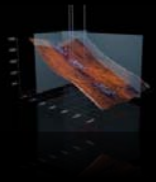
1.4 Billion Transistors

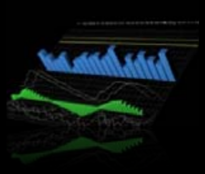


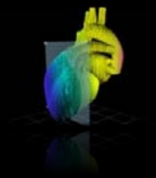
1 Teraflop of processing power

Heterogeneous Computing Domains

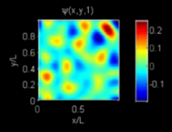



Oil & Gas


Finance


Medical


Biophysics


Numerics


Audio


Video


Imaging

CUDA

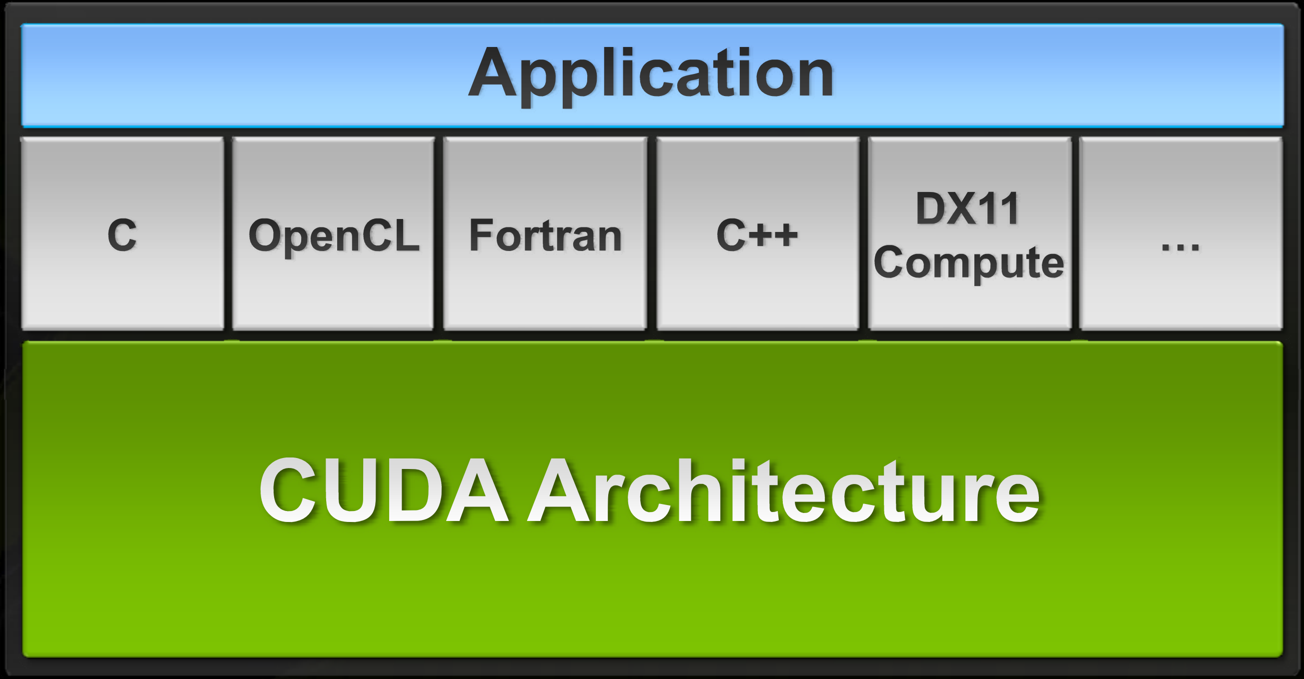
CUDA Parallel Programming Architecture and Model
Programming the GPU in High-Level Languages

CUDA is C for Parallel Processors

- CUDA is industry-standard C with minimal extensions
  - Write a program for one thread
  - Instantiate it on many parallel threads
  - Familiar programming model and language
- CUDA is a scalable parallel programming model
  - Program runs on any number of processors without recompiling
- CUDA parallelism applies to both CPUs and GPUs
  - Compile the same program source to run on different platforms with widely different parallelism
  - Map to CUDA threads to GPU threads or to CPU vectors

CUDA Parallel Computing Architecture

- Parallel computing architecture and programming model
- Includes a C compiler plus support for OpenCL and DX11 Compute
- Architected to natively support all computational interfaces (standard languages and APIs)
- NVIDIA GPU architecture accelerates CUDA
  - Hardware and software designed together for computing
  - Expose the computational horsepower of NVIDIA GPUs
  - Enable general-purpose GPU computing

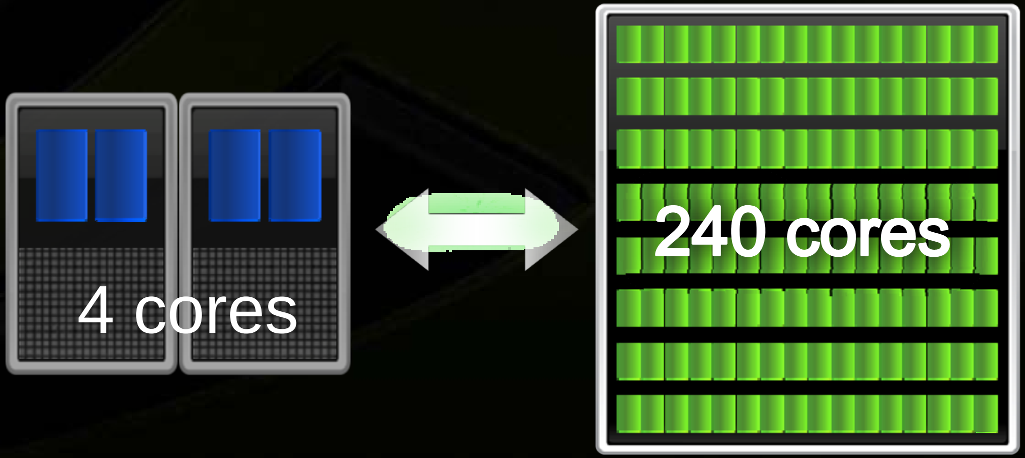


Application Software (written in C)

CUDA Libraries

cuFFT | cuBLAS | cuDPP

CPU Hardware	CUDA Compiler	CUDA Tools
1U PCI-E Switch	C Fortran	Debugger Profiler



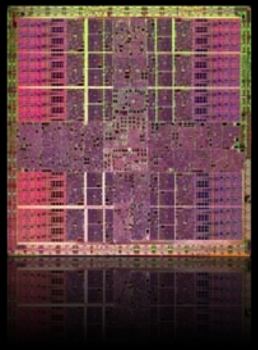
Pervasive CUDA Parallel Computing

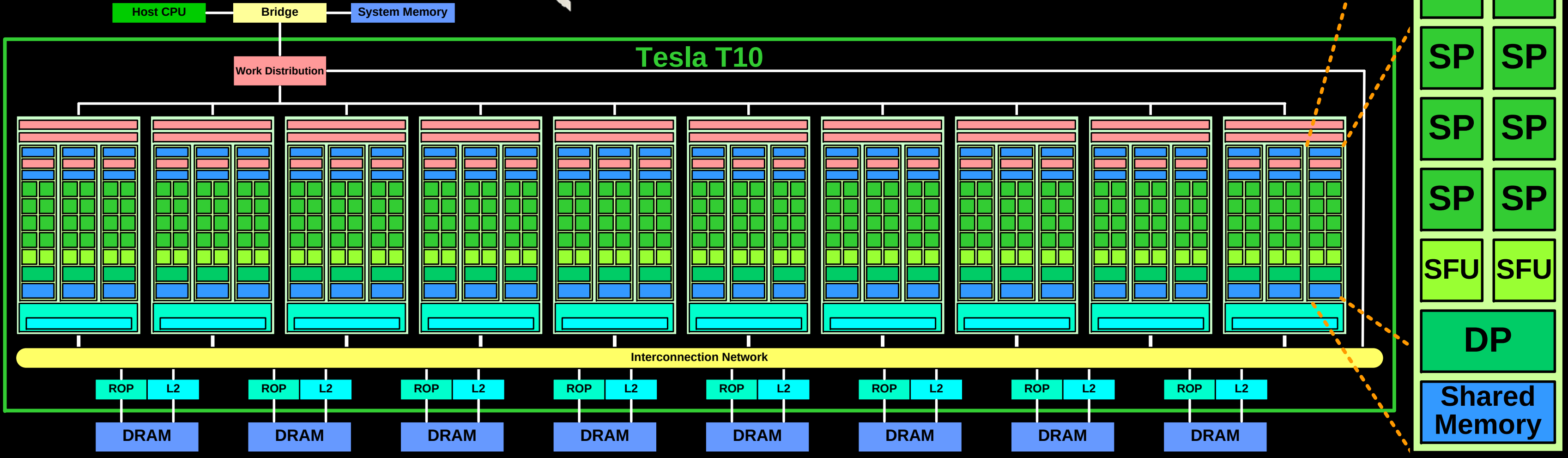
- CUDA brings data-parallel computing to the masses
  - Over 100M CUDA-capable GPUs deployed since Nov 2006
- Wide developer acceptance
  - Over 150K CUDA developer downloads (CUDA is free!)
  - Over 25k CUDA developers . . . and growing rapidly
  - A GPU "developer kit" costs ~ $200 for 500 GFLOPS
  - Now available on any new Macbook
- Data-parallel supercomputers are everywhere!
  - CUDA makes this power readily accessible
  - Enables rapid innovations in data-parallel computing

Massively parallel computing has become a commodity technology!

CUDA Computing with Tesla

- 240 SP processors at 1.5 GHz: 1 TFLOPS peak
- 128 threads per processor: 30,720 threads total
- Tesla PCI-e board: C1060 (1 GPU)
- 1U Server: S1070 (4 GPUs)





CUDA Uses Extensive Multithreading

- CUDA threads express fine-grained data parallelism
  - Map threads to GPU threads
  - Virtualize the processors
  - You must rethink your algorithms to be aggressively parallel
- CUDA thread blocks express coarse-grained parallelism
  - Blocks hold arrays of GPU threads, define shared memory boundaries
  - Allow scaling between smaller and larger GPUs
- GPUs execute thousands of lightweight threads
  - (In graphics, each thread computes one pixel)
  - One CUDA thread computes one result (or several results)
  - Hardware multithreading & zero-overhead scheduling

CUDA Computing Sweet Spots

Parallel Applications

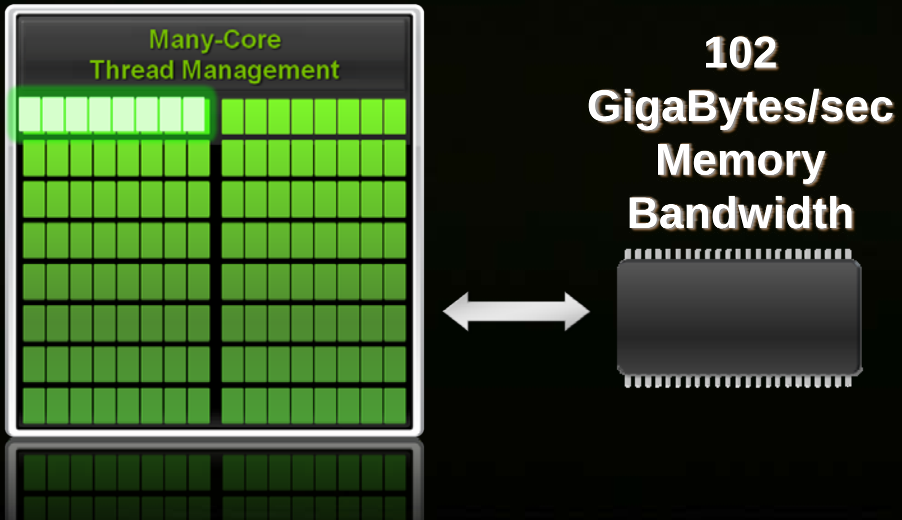
- **High bandwidth:**
  Sequencing (virus scanning, genomics), sorting, database, …

- **Visual computing:**
  Graphics, image processing, tomography, machine vision, …

- **High arithmetic intensity:**
  Dense linear algebra, PDEs, *n*-body, finite difference, …

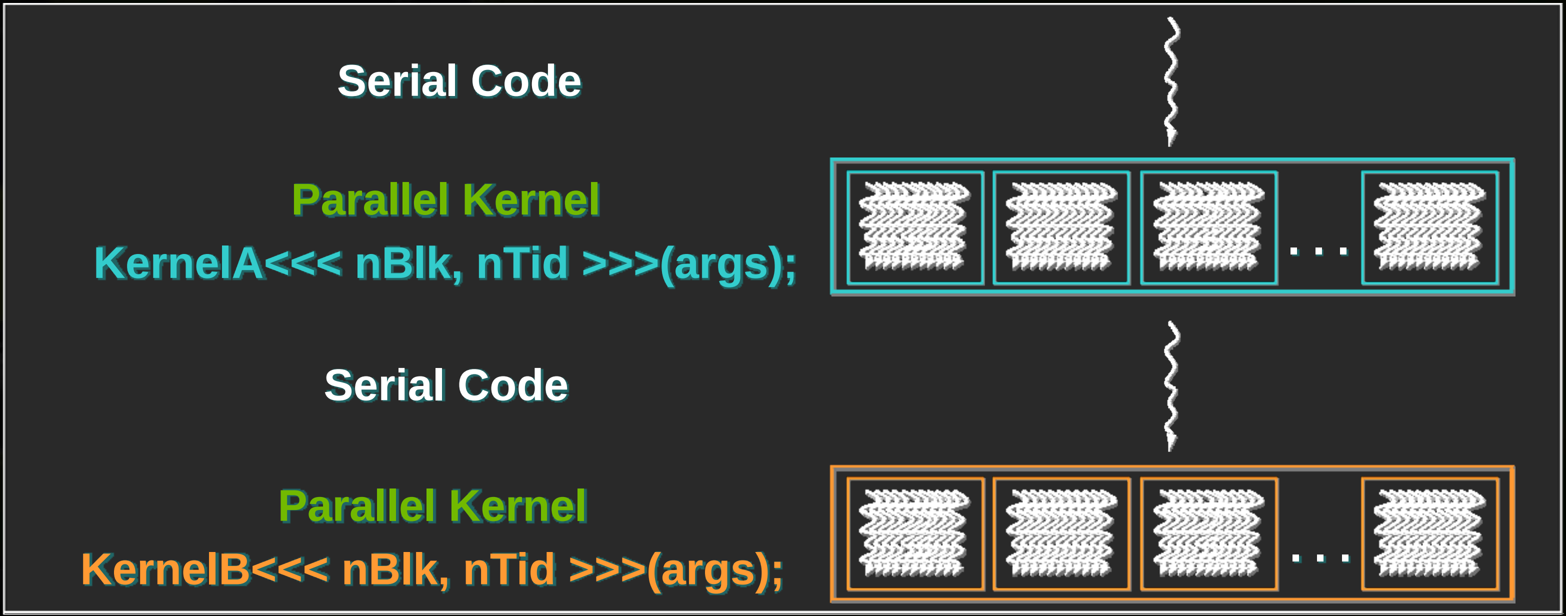
A Highly Multithreaded Coprocessor

- The GPU is a highly parallel compute coprocessor
  - serves as a coprocessor for the host CPU
  - has its own device memory with high bandwidth interconnect
- The application run its parallel parts on GPU, via kernels.
  - Many threads execute same kernel
  - SIMT = Single Instruction Multiple Threads
  - GPU Threads are extremely lightweight
    - Very little creation overhead,
    - Instant switching
  - GPU uses 1000s of threads for efficiency



Heterogeneous Programming

- CUDA application = serial program executing parallel kernels, all in C
  - Serial C code executed by a CPU thread
  - Parallel kernel C code executed by GPU, in *threads (grouped in blocks)*



Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same program, SIMT (Singe Instruction multiple threads)
  - Each thread uses its ID to compute addresses and make control decisions

threadID

0	1	2	3	4	5	6	7

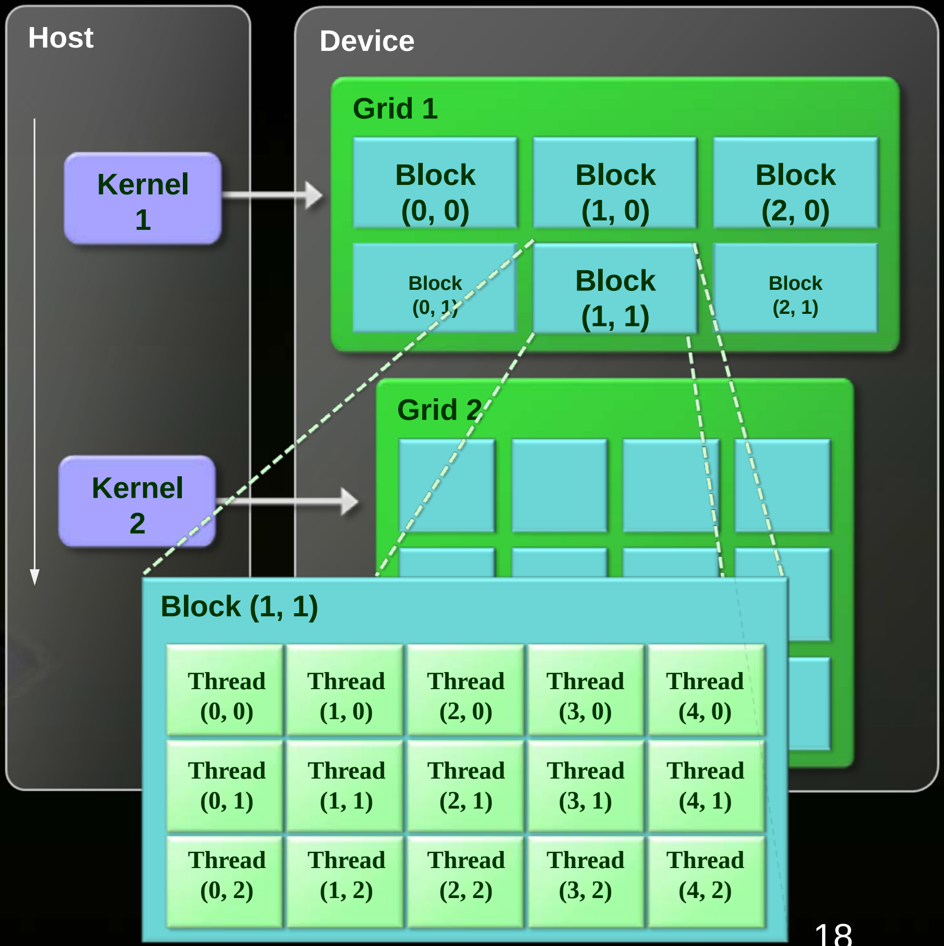
```
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

CUDA Programming Model

A kernel is executed by a grid, which contain blocks.

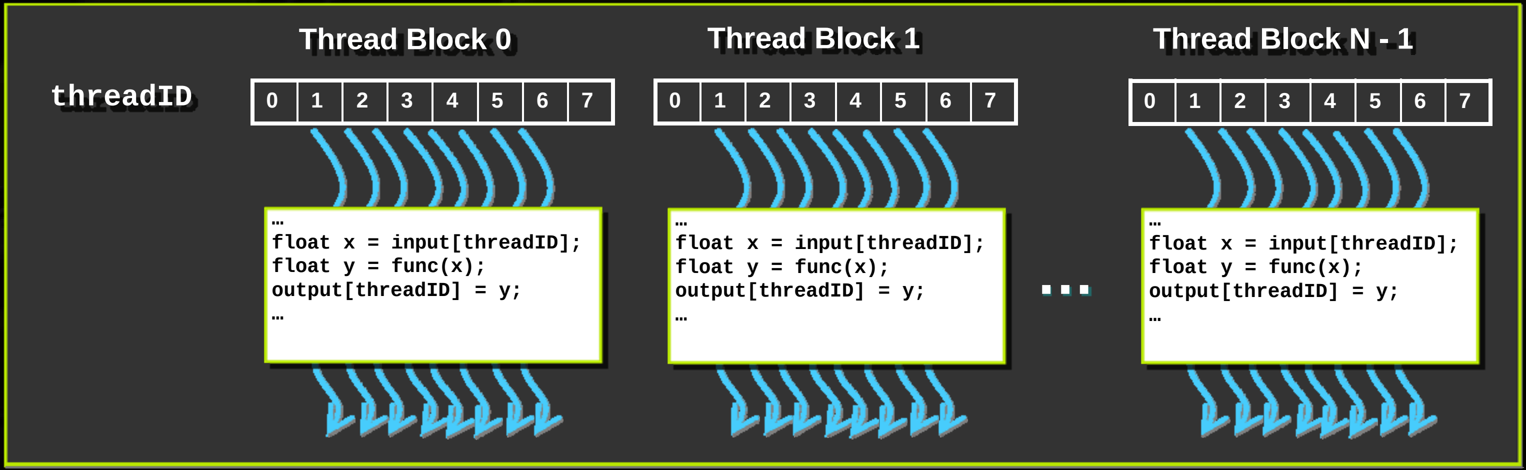
These blocks contain our threads.

- A thread block is a batch of threads that can cooperate:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks operate independently



Thread Blocks: Scalable Cooperation

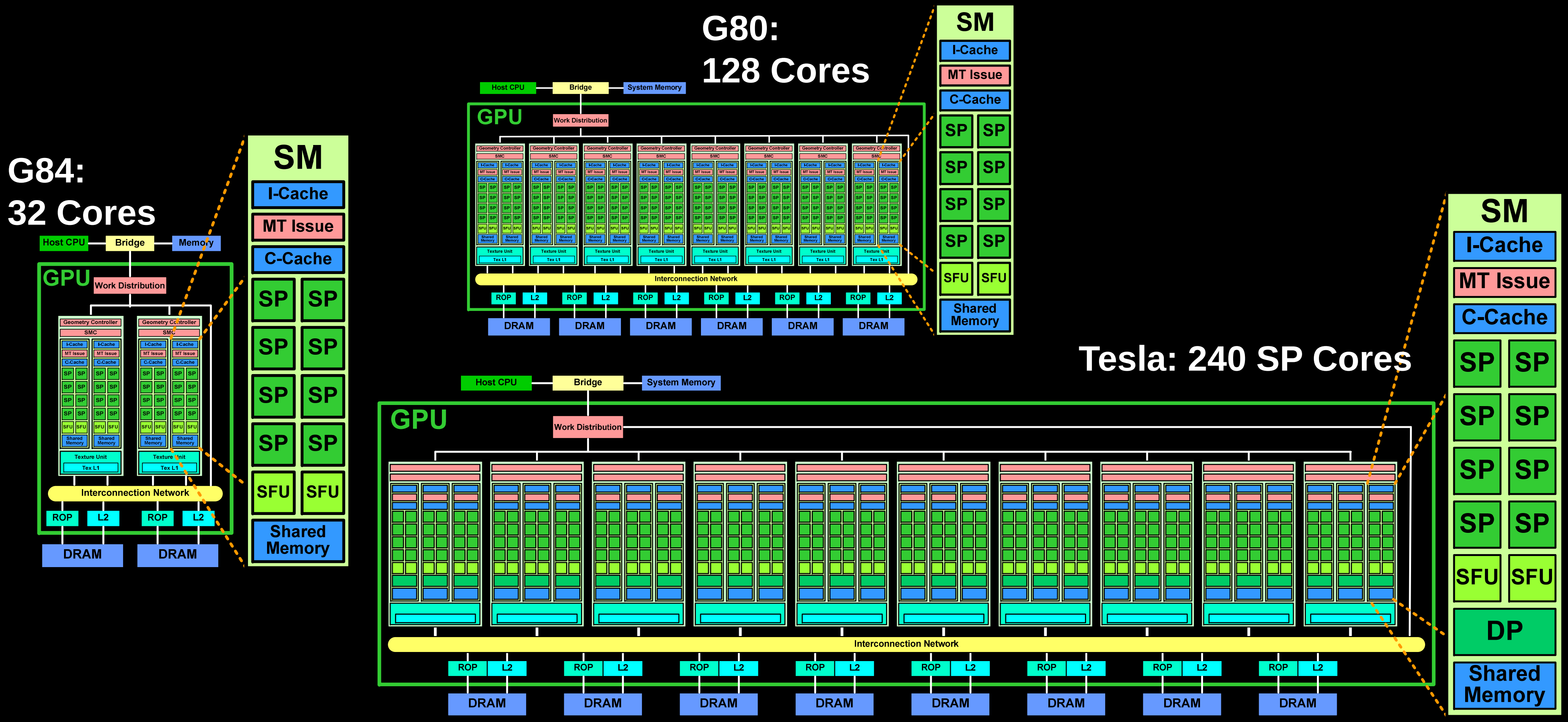
- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via shared memory
  - Threads in different blocks cannot cooperate
- **Enables programs to transparently scale to any number of processors!**



Thread Cooperation

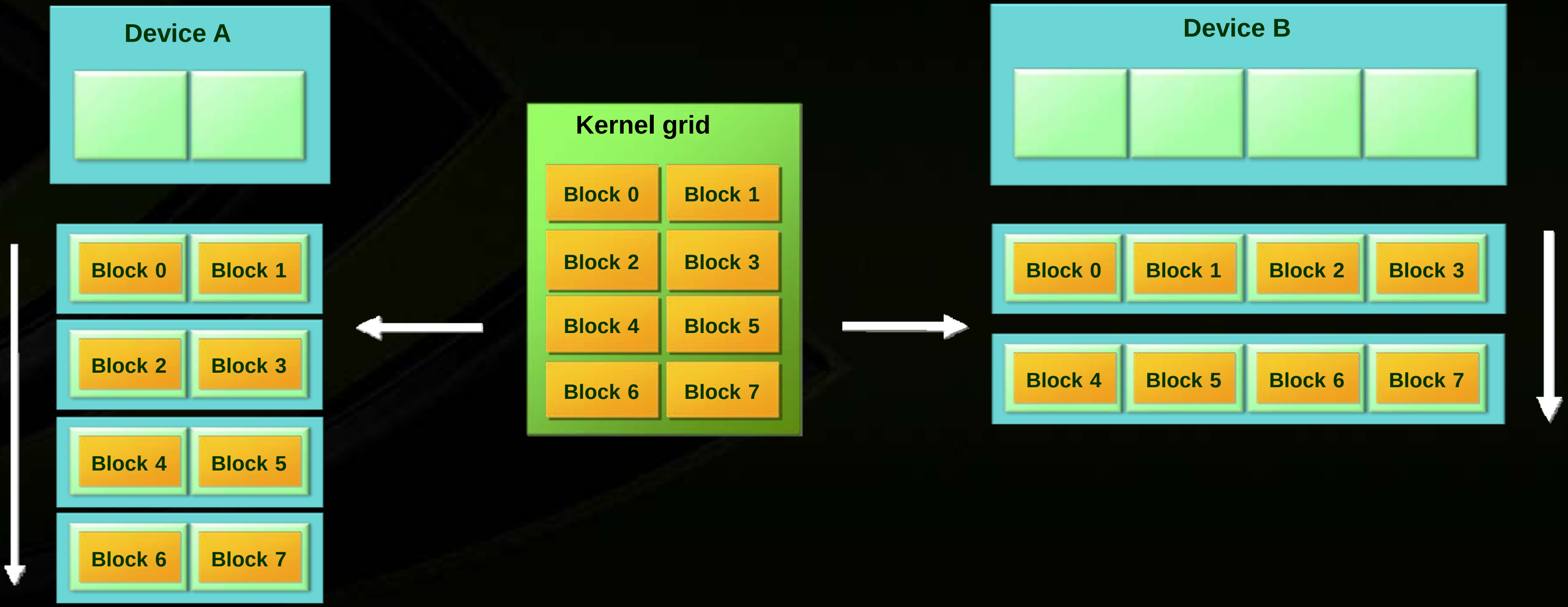
- Thread cooperation is a powerful feature of CUDA
    - Threads can cooperate via on-chip shared memory and synchronization
- The on-chip shared memory within one block allows:
    - Share memory accesses, drastic *memory bandwidth reduction*
    - Share intermediate results, thus: *save computation*
- Makes algorithm porting to GPUs a *lot* easier
  (vs. GPGPU and its strict stream processor model)

Reason for blocks: GPU scalability



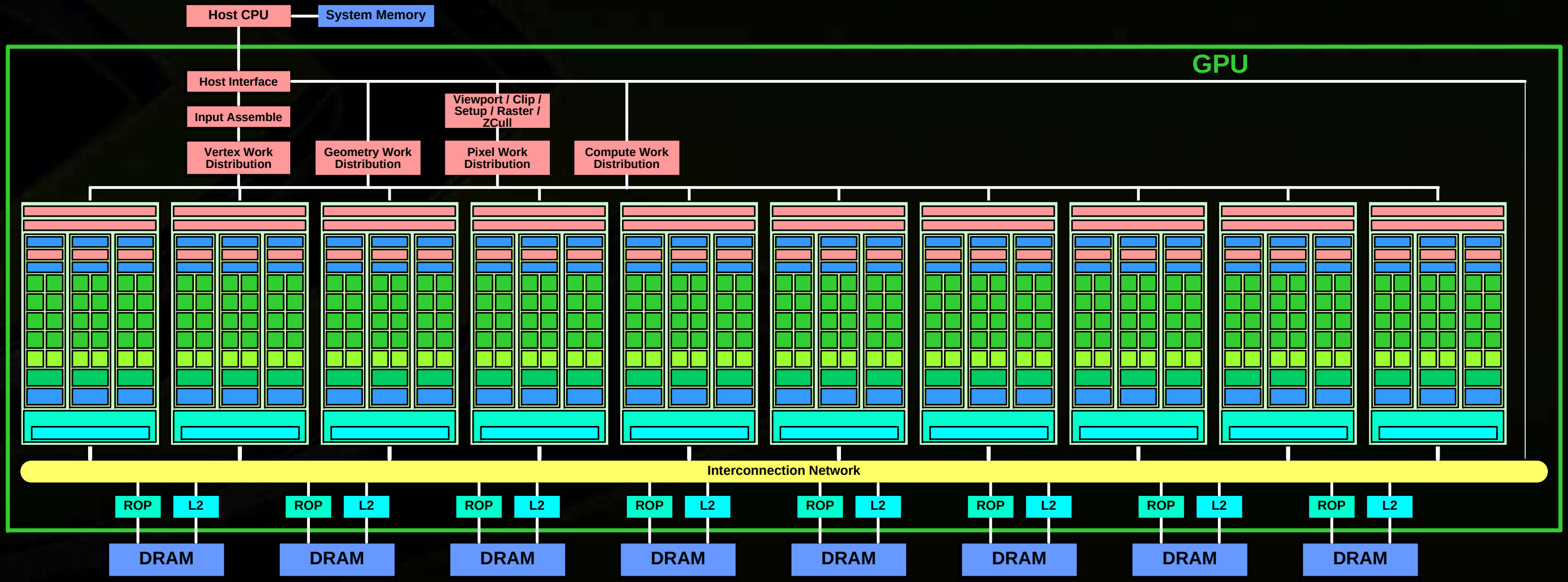
Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
  - Kernels scale to any number of parallel multiprocessors



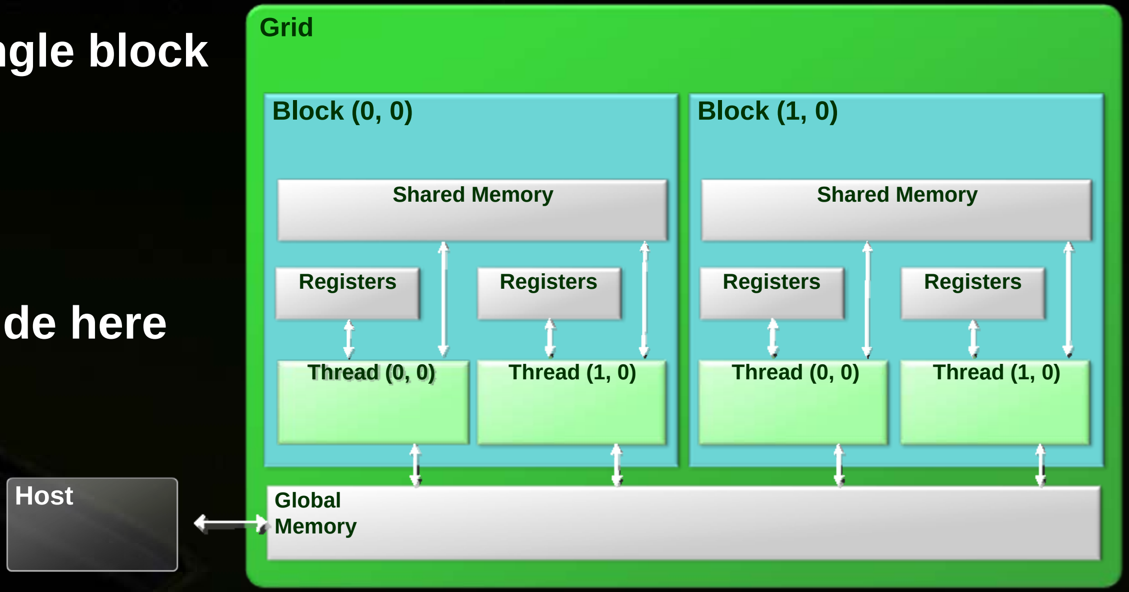
Parallel Core GPU – Block Diagram

- Tesla T10 chip (Tesla C1060 / one GPU of Tesla S1070)
- 240 Units execute kernel threads, grouped into 10 multiprocessors
- Up to 30,720 parallel threads active in the multiprocessors
- Threads are grouped in blocks, providing shared memory: Scalability!!



Memory model seen from CUDA Kernel

- Registers (per thread)
- Shared Memory
    - Shared among threads in a single block
    - On-chip, small
    - As fast as registers
- Global Memory
    - Kernel inputs and outputs reside here
    - Off-chip, large
    - Uncached (use coalescing)



- Note: The host can read & write global memory but not shared memory

Simple "C" Extensions to Express Parallelism

Standard C Code	*CUDA C Code*

Standard C Code

```
void
saxpy_serial(int n, float a,
             float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

CUDA C Code

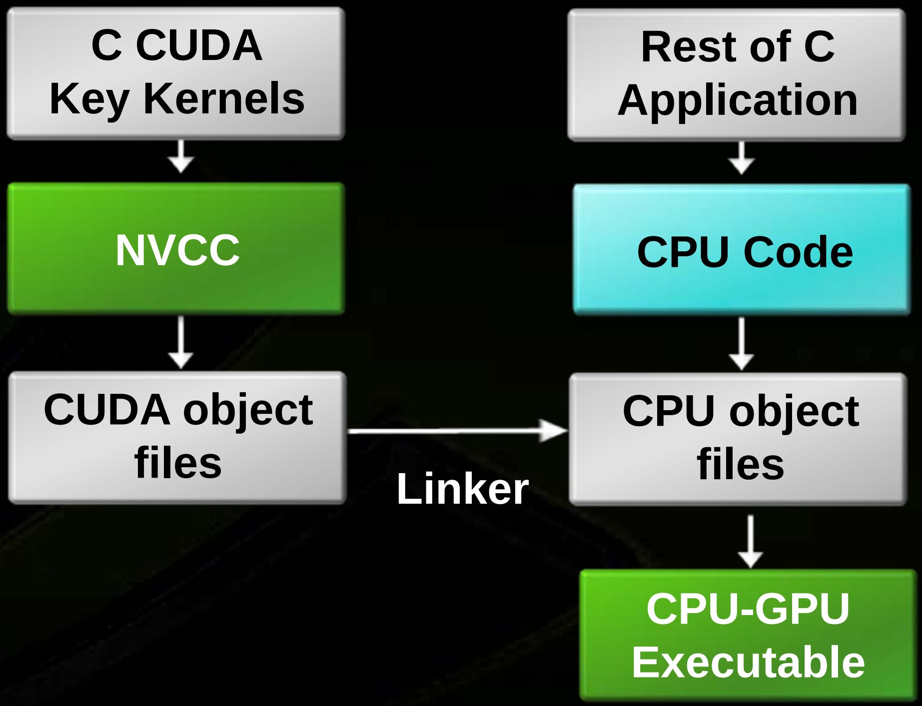
```
__global__ void
saxpy_parallel(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with
// 256 threads/block
int nblocks = (n + 255) / 256;

saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Compilation

- Any source file containing CUDA language extensions must be compiled with nvcc
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC can output:
  - Either C code (CPU Code)
    - That must then be compiled with the rest of the application using another tool
  - Or PTX object code directly
- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (cuda)

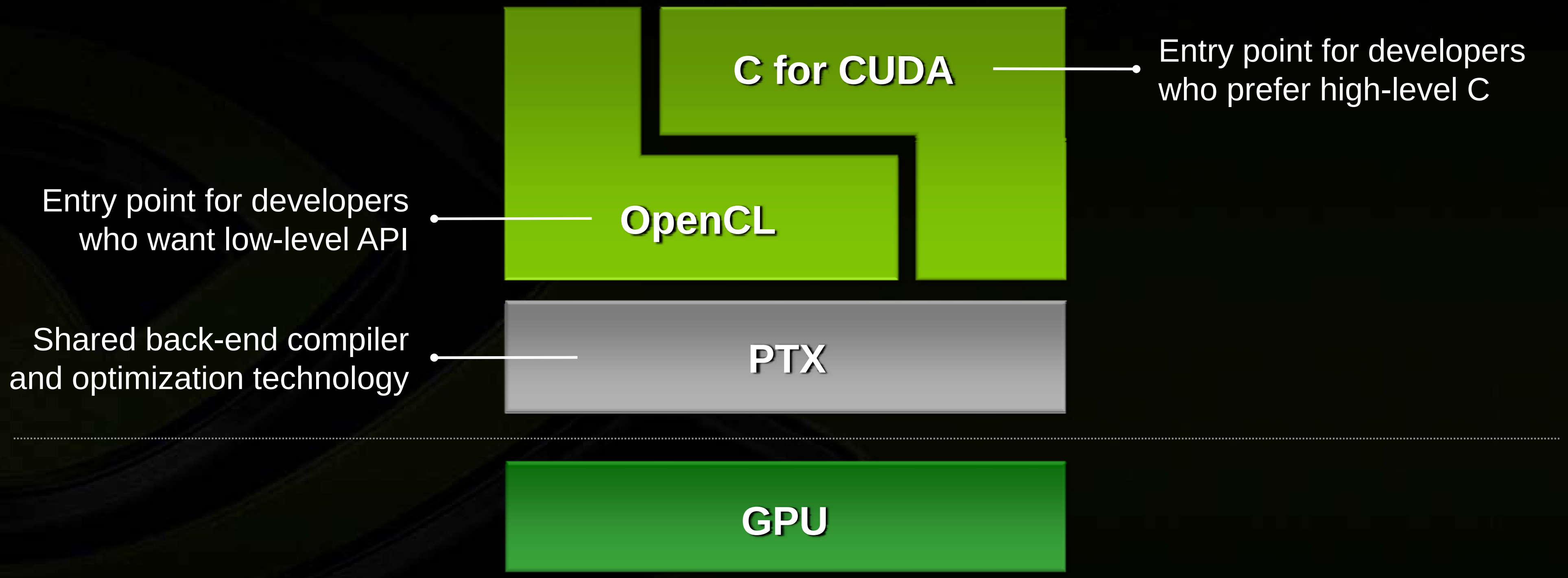
Compiling C for CUDA Applications



Keys to GPU Computing Performance

- Hardware Thread Management
  - Thousands of lightweight concurrent threads
  - No switching overhead
  - Hide instruction and memory latency
- On-Chip Shared Memory
  - User-managed data cache
  - Thread communication / cooperation within blocks
- Random access to global memory
  - Any thread can read/write any location(s)
  - Direct host access

NVIDIA C for CUDA and OpenCL

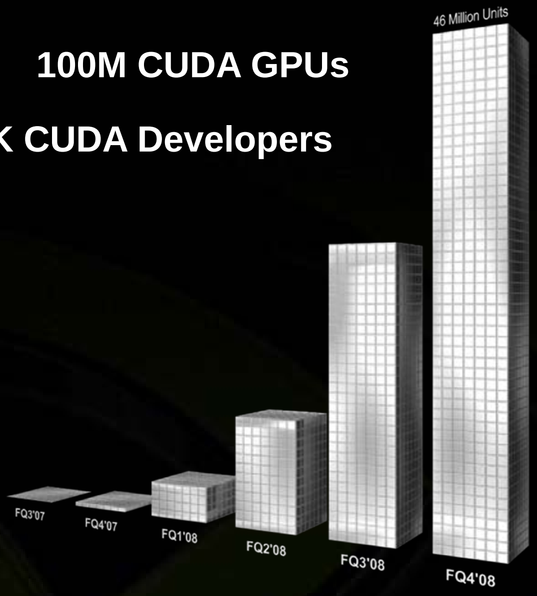


Different Programming Styles

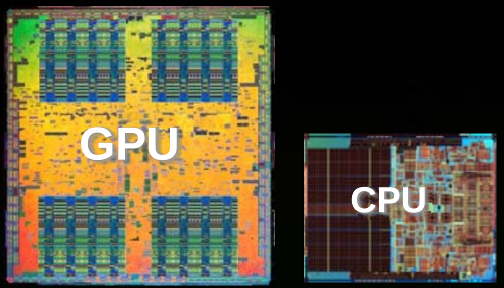
- C for CUDA
  - C with parallel keywords
  - C runtime that abstracts driver API
  - Memory managed by C runtime
  - Generates PTX
- OpenCL
  - Hardware API - similar to OpenGL and CUDA driver API
  - Programmer has complete access to hardware device
  - Memory managed by programmer
  - Generates PTX

100M CUDA GPUs

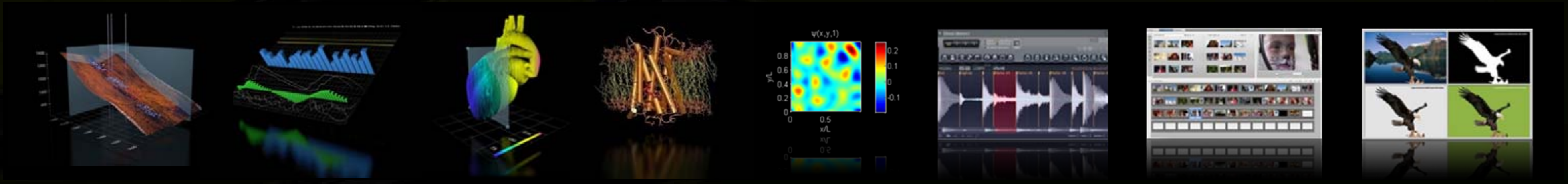
30K CUDA Developers



CUDA



Heterogeneous Computing



Oil & Gas | Finance | Medical | Biophysics | Numerics | Audio | Video | Imaging

Resources

- NVIDIA CUDA Zone (www.nvidia.com/cuda)
  - SDK
  - Manuals
  - Papers
  - Forum
  - Courses