Advanced Software Engineering Lecture TUM

Introduction to Parallel Computing & Parallel Programming Models

2021-12-10 Thanos Theo, SW Engineering Management & HPC/GPU Technical Leadership

Agenda

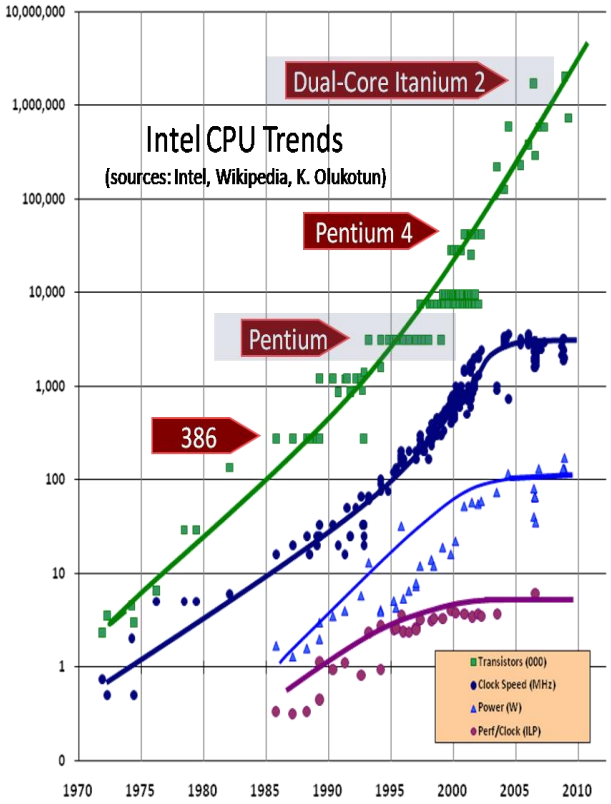
- *Advanced Software Engineering Lecture (Theory & Practical):*
  - *Let's make it a two-way interactive process with participants*
    - *1st Part (Theory, 30mins):* ***Introduction to Parallel Computing***
      - *History*
      - *Evolution of Hardware & Software Paradigm*
      - *Current Compute Architectures Types*
    - *2nd Part (Theory, 30mins):* ***Parallel Programming Model***
      - *CPU*
      - *GPU/CUDA*
    - *3rd Part (Practical, 15mins):* ***Open source demo code presented***
    - *Last 15mins Q&A & Open Discussion*

Part 1

Theory:
Introduction to Parallel Computing

The end of Moore's Law?

- Mid 2000s, chip manufacturers hit the 'Power Wall':
  - Moore's Law can't continue indefinitely for single core performance: chips get ***too hot*** at higher clock speeds
  - Herb Sutter famous article: The Free Lunch Is Over
    - Many declared Moore's Law being dead

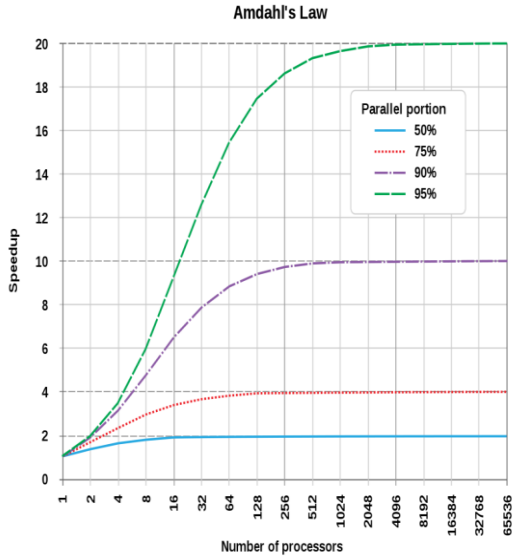


Emerging: Amdahl's Law

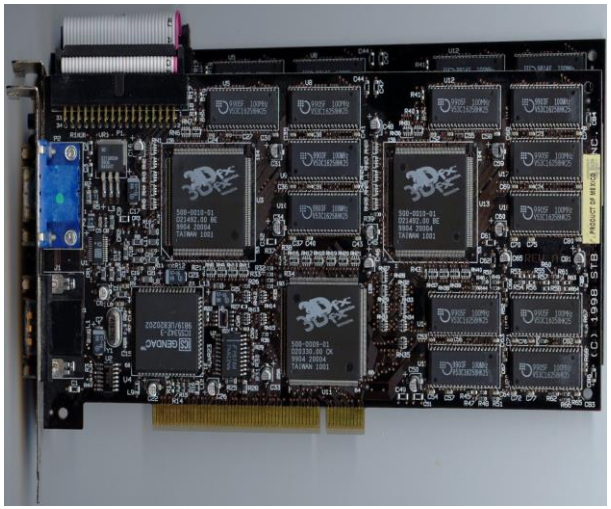
- Shift in Programming Paradigm:
  - Single Core performance not scaling anymore
  - Need to write parallel programs (end of '***free lunch***')
  - Equation for speed-up:

$$OverallSpeedup = \frac{1}{1 - FractionEnhanced + (FractionEnhanced / SpeedupEnhanced)}$$

  - Example: FractionEnhanced = 90% & with SpeedupEnhanced (N) -> limit of inf, 90%/N -> 0, then 1 / (1 - 0.90) -> only 10x speed-up with infinite cores!
  - ***Parallel computing with many processors is useful only for highly parallelizable programs***
  - ***Hardware design & support?***



A bit of 90's history: the Battle of the CG ASICs



- Silicon Graphics Inc R&D of 80's and their famous OpenGL 1.0 library (1992)
  - Multiple start-ups in Silicon Valley to bring dedicated cheap CG ASIC chips out
  - Ex- SGI employees start 3dfx (1994, fast ASIC 3D core only)
  - Rendition Verite (early 1990s, RISC/MIPS programmable core, vQuake)
  - Nvidia ex-AMD employees (1993, fast ASIC-based 2D/3D core)

  - End of the 1990's: the API Wars
    - 3dfx's Glide vs SGI's OpenGL vs MS's DirectX (also joint project Fahrenheit)
    - MS antitrust lawsuit, the 'triple E' phrase & their effort to kill OpenGL
    - 1999: Nvidia introduces Geforce 256 (HW T&L) and Geforce 3 (2001) 1st ***'programmable'*** GPU
    - Major turn: early 2000s 3dfx is defunct (Nvidia acquisition), SGI also slowly dies with the 2001 bubble bust, OpenGL in limbo for years

SiliconGraphics

3dfx

APIs: How to program a ‘programmable’ semi-ASIC?

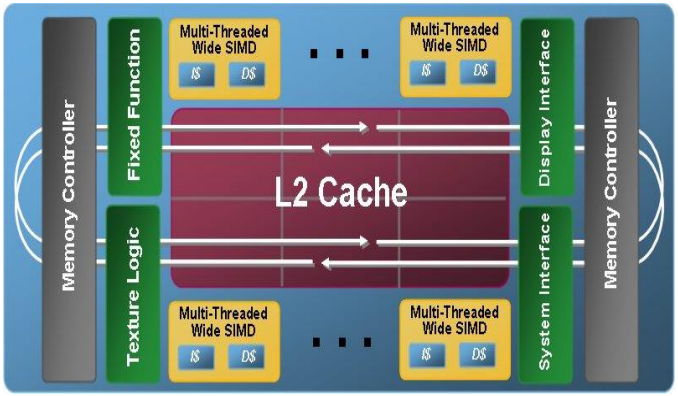
- The real-time CG community turned to the ***established CG film industry***, based on ***Stanford R&D***
  - ***Pixar’s Renderman*** & its ***C-based ‘Shader Language’*** (RSL, 1999)
  - This becomes the emerging ***Parallel Programming Paradigm***
- Reflected in Direct HLSL, OpenGL GLSL & Nvidia’s CG
  - OpenGL introduced GLSL in 2004
  - Original GPU Computing research efforts (GPGPU) in shader languages
    - The '***shader***' concept
- With the Geforce 8800 (8th Gen), GPUs become ***fully programmable***
  - C-based, low-level GPU Computing APIs introduced:
    - CUDA (2007, Nvidia, vendor-locked)
    - OpenCL (2009, from Apple/Nvidia & given to Khronos, multi-vendor)
      - The '***kernel***' concept

RENDERMAN

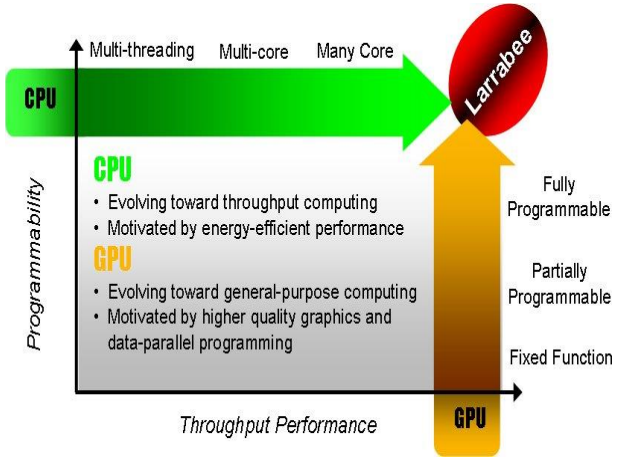
NVIDIA CUDA

OpenCL

The Intel approach: SIMD & AVX

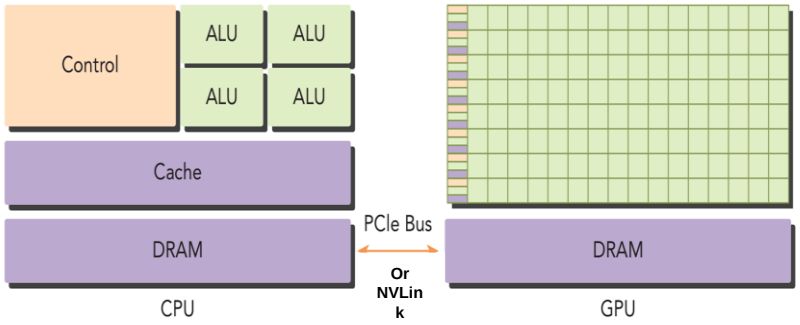


- Intel introduced ***SIMD approaches*** since late 90s
  - Pentium 3 had the 1st SIMD 128-bit registers (***packed float4***)
  - Mid 2000s dual-core CPUs where introduced
  - ***Difficult to program***, auto-vectorization compiler support limited
- In 2008 Intel decides to take on the GPU market
  - GPGPU chip Larrabee introduced, a ***hybrid*** between a multi-core CPU & a GPU, with ***ray-tracing*** in mind
  - ***AVX-512*** introduced with 512-bit registers (***packed float16*** or ***double8***) on top of a standard Atom/P54 core, up to ***80 Atom/P54*** cores per chip
  - ***No specific API*** support (intrinsics-level support, combined with multi-threaded code)
  - ***Failed*** to enter the market
  - ***But***: Intel re-entered the GPU market in late 2020!



CPU vs GPU architectures – Simplified Overview

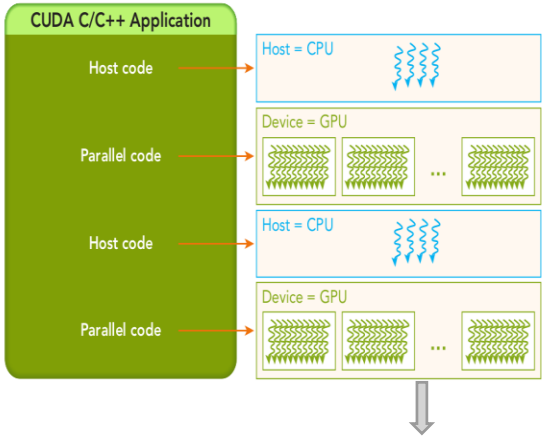
- Simplified overview



Intel / AMD CPU platforms

Nvidia / AMD GPU platforms
For Eg. Nvidia product family like
Tegra, Quadro, Tesla, Geforce
AMD product family like
Radeon, Firepro etc.

GPU hardware architecture – Modern Overview



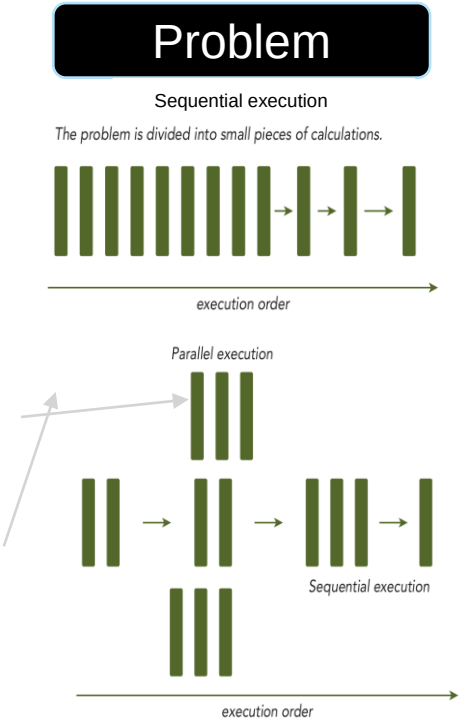
CPU Threads: Heavy weight entities. OS performs swap the threads on & off CPU execution channels to provide multithreading capability. Context switches are slow and expensive.

GPU Threads: Light weight threads. Typically, thousands of threads are queued up for work and if GPU is waiting on one group of threads, it simply begins executing the other group. No context save – restore.

Volta V100 Architecture

Parallel Computing - Overview

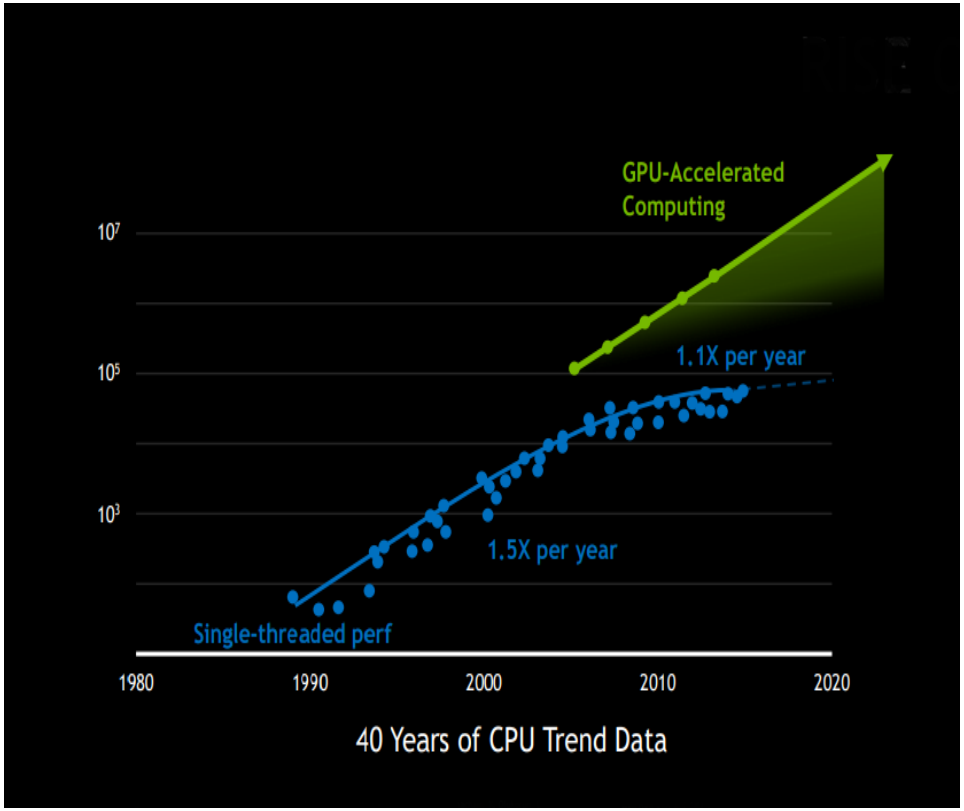
- Computational methodology in which many calculations are carried out simultaneously
  - Assumption : Large problems can be broken down into smaller ones, which are solved concurrently
  - Involves two area of computing technologies
    - ***Compute Platform Architecture*** (***hardware*** aspect) - focuses on supporting parallelism at chip/hardware architecture level
    - ***Parallel Programming Paradigm*** (***software*** aspect) - solving a problem by fully utilizing the compute power of the underlying architecture



- In a Task, **input** is applied to the function and **output** is produced
  - Relationship between various tasks can be categorized as dependent or independent.
  - Analysis of the data dependency is of prime focus in parallel implementation of algorithms as data dependencies are the primary inhibitors to ||ism.
- ***Task*** vs. ***Data Parallelism***
  - Task parallelism focuses on distributing task across multiple cores.
  - Data parallelism focuses on distributing data across multiple cores. CUDA programming is suited to address data parallel problems.

Is Moore's Law really dead?

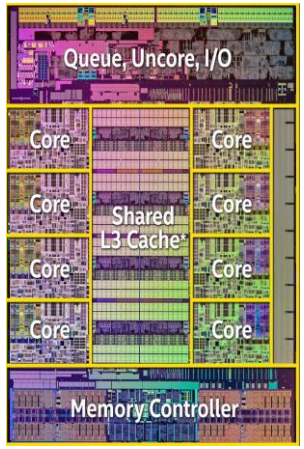
- The Nvidia CEO, Jensen Huang, has declared it dead, but:
  - Moore's Law ***continues*** (ie, is alive and well, don't believe the NV hype!):
    - With parallel hardware architectures from NVidia, AMD & Intel
    - With the new Parallel Programming Paradigm (the '***shader***' & '***kernel***' concepts)



Part 2

Theory
Parallel Programming Models

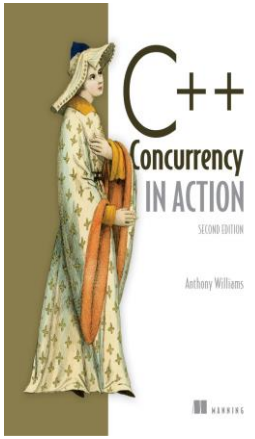
Parallel Programming Model – CPU



Modern 8-core i7 Intel CPU

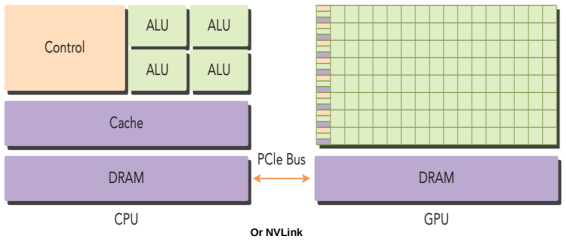
- Current multi-CPUs can have a lot of big cores (up to ***128*** -> ***64C / 128H*** for high-end CPUs):
  - Parallel Programming Model for CPUs follows the classic OS-based threading model
    - OS is responsible for thread scheduling (see also ***thread affinity***)
    - Standard languages & APIs can be used (like ***pthreads***, ***C11*** & ***C++11 threads*** with ***atomics*** support)
    - The programmer is responsible of introducing higher-level concepts (***ThreadBarrier*** / ***ThreadPool*** / ***parallel_For()***)
    - Unfortunately, ***SIMD / AVX / AVX-512*** support is not exposed via the ISO C/C++ languages (only via intrinsics)
      - Special compilers needed for proper automatic ***SIMD / AVX / AVX-512*** support (Intel's ***ispc***, ***icc***, etc)

- Modern C/C++ has moved towards CPU parallelization:
  - ***C11/C++11*** introduced ***threads*** & ***atomics*** in the language
  - ***C++11*** has been updated with a new ***memory model*** supporting concurrency
  - Promises for a fully featured ***concurrency library*** since C++11
    - Partially delivered with C++17 (***parallel std algorithms***)
    - ***Java*** & ***C#*** have advanced concurrency libraries available since the late 2000s
    - With ***C++20*** support still pending for concurrent containers (Microsoft with PPL, Google & Facebook)
      - See Williams, C++ Concurrency in Action



Heterogeneous Architecture

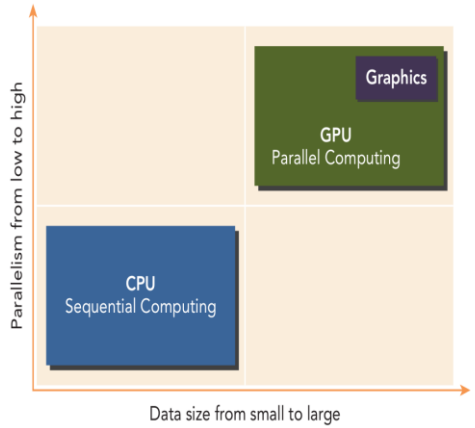
- ***Nomenclature***: typical heterogeneous compute node comprises of multi-core CPUs along with one or more many-core GPUs interfaced via PCIe/NVLink buses:
  - GPU(s) operate in conjunction with CPU(s) based host(s). In GPU computing terms, the CPU is always called the ***host*** and the GPU is the ***device***
- Heterogeneous application comprises of:
  - ***Host Code:*** runs on the CPU(s). Responsible for managing the environment, run-time, code & data for the device before launching the compute intensive tasks on the device
  - ***Device Code:*** running on the GPU(s)
- Understanding GPU's capabilities:
  - Number of ***CUDA cores***
  - ***Memory Size*** (also known as ***VRAM size***)
- Metrics for measuring GPU performance:
  - ***Peak Performance:*** how many single/double precision floating point calculations can be processed per second (***TFLOPS***)
  - ***Memory Bandwidth:*** ratio at which data can be read from or stored to memory (***GB/sec***)
  - ***Memory Bus Width***: width of the GPU on-chip memory controller (***bits***)

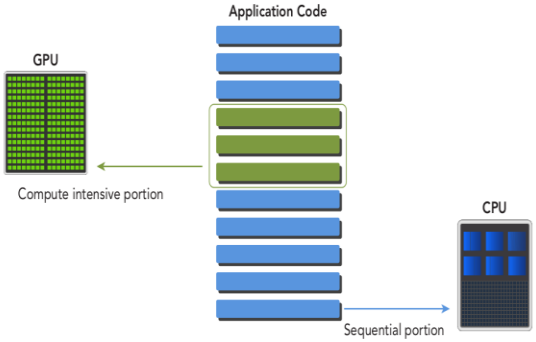


	Tesla C2050 (FERMI)	Tesla K40 (KEPLER)	Tesla P100 (PASCAL)	Tesla V100 (VOLTA)	NV Titan RTX (TURING)
CUDA Cores	448	2880	3584	5120	4608
Memory Size	6 GB	12 GB (GDDR5)	16 GB (HBM2)	16 GB (HBM2)	24 GB (GDDR6)
Peak Performance	1.03 TFLOPS	5.040 TFLOPS	10.609 TFLOPS	14.899 TFLOPS	16.312 TFLOPS
Memory Bandwidth	144 GB/s	288 GB/s	732 GB/s	900 GB/s	672 GB/s
Memory Bus Width	384 bits	384 bits	4096 bits	4096 bits	384 bits

Heterogeneous Architecture (cont.)

- Paradigm of heterogeneous architecture:
  - ***CPU Computing***: good for ***control intensive tasks***
  - ***GPU Computing***: good for ***data parallel computing tasks***
  - Two dimensions that differentiate the scope of applications for CPU and GPU
    - ***Parallelism Level***
    - ***Size of Data***
    - If the problem have small data size, sophisticated control logic/low level of ||ism, ***CPU is preferred*** (it can efficiently handle complex control logic)
    - Huge amount of data and massive data ||ism, ***GPU is the right choice*** (higher number of cores, support massive multithreading, has a larger peak memory bandwidth compared to the CPU)

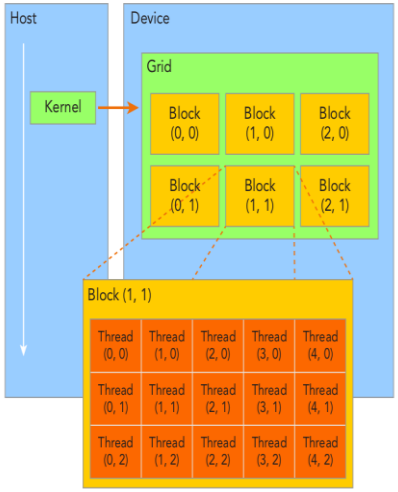




Parallel Programming Model – GPU/CUDA

- Typically the GPU is a scalable array of Streaming Multiprocessors (***SMs***)
  - Each ***SM***: supports concurrent execution of the several hundreds of threads; having multiple ***SM***s per GPU help concurrent execution of thousands of threads per GPU.
- ***CUDA*** (Compute Unified Device Architecture)
  - A ***C-based (with C++ extensions)*** programming model that leverages the parallel compute entities in the NVIDIA GPU's to solve complex compute intensive problems in an efficient way.

- Terminology:
  - ***Kernel***: Function that is targeted for execution on device
  - ***Thread***: Single instance of execution (on a CUDA Core)
  - ***Thread Block***: Group of threads (running on the same ***SM)***
  - ***Grid***: Group of thread blocks -> all the threads spawned by a single kernel launch



When a kernel Grid is launched for execution on the device, thread blocks of that kernel grid are distributed among available SMs for execution. Once scheduled, the threads of that ***thread block*** execute concurrently on that ***assigned SM***.

Simple Kernel Examples

```
__global__ void add (int* a, int* b,
int* c, int N)
{
    int tid = blockIdx.x;
    If (tid < N)
        C[tid] = a[tid] + b[tid];
}
```

To launch the Kernel Grid:

```
add<<<N, 1>>>(dev_a, dev_b, dev_c);
```

```
__global__ void add (int* a, int* b,
int* c, int N)
{
    int tid = threadIdx.x;
    If (tid < N)
        C[tid] = a[tid] + b[tid];
}
```

```
add<<<1, N>>>(dev_a, dev_b, dev_c);
```

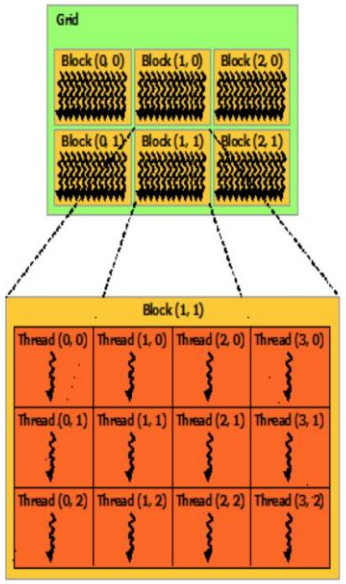
```
__global__ void add (int* a, int* b,
int* c, int N)
{
    int tid = blockIdx.x * blockDim.x +
threadIdx.x;
    If (tid < N)
        C[tid] = a[tid] + b[tid];
}
```

Kernel Grid Looks like :

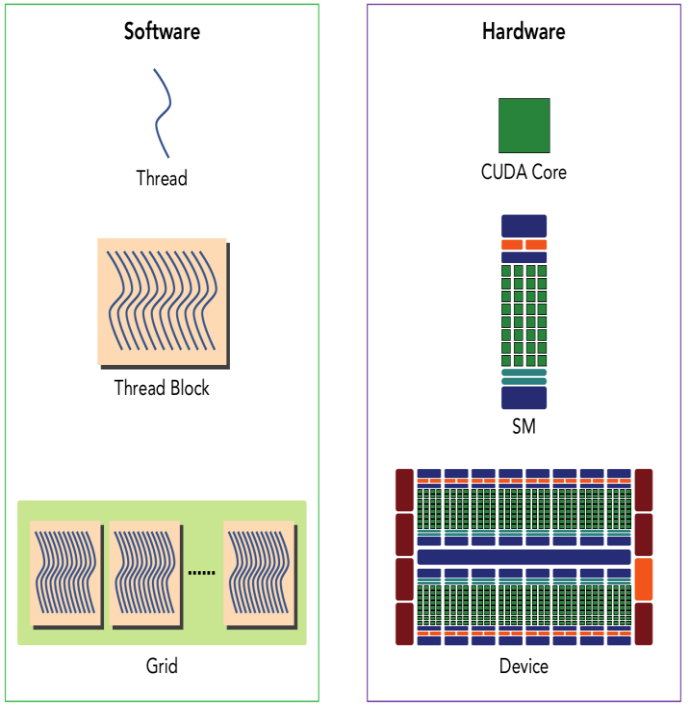
```
add<<<(N + 255) / 256, 256>>>(dev_a, dev_b, dev_c);
```

Parallel Programming Model – GPU/CUDA (cont.)

- Under the hood, when a Kernel Grid is launched
  - ***Thread blocks*** of that ***kernel grid*** are distributed among ***available SMs*** for execution. Once scheduled, the threads of that thread block execute concurrently on that assigned SM.
  - The threads belonging to the thread block distribute among themselves the available resources of the SM, like registers & shared memory (L1 cache), depending on the usage in the kernel.

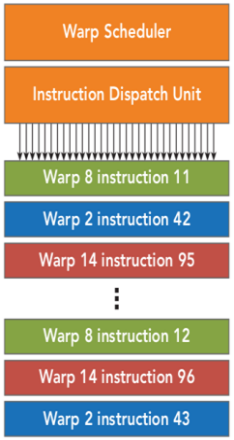


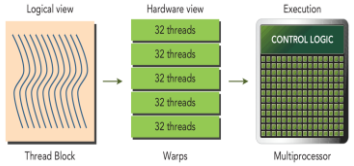
Mapping of CUDA terminology with the Underlying Compute Architecture



Terminology:

- CUDA employs a ***Single-Instruction-Multiple-Threads*** (***SIMT***) architecture to manage & execute threads in a group of ***32 threads*** called ***warps***.
- **Warps:** all the ***32 threads*** in a warp execute the same instruction at the same time. Every thread has its own instruction address counter & register state , carries out the current instruction on its own data. Every SM partitions the thread blocks assigned to it into 32-thread warps, which is then scheduled for execution on available compute resources.

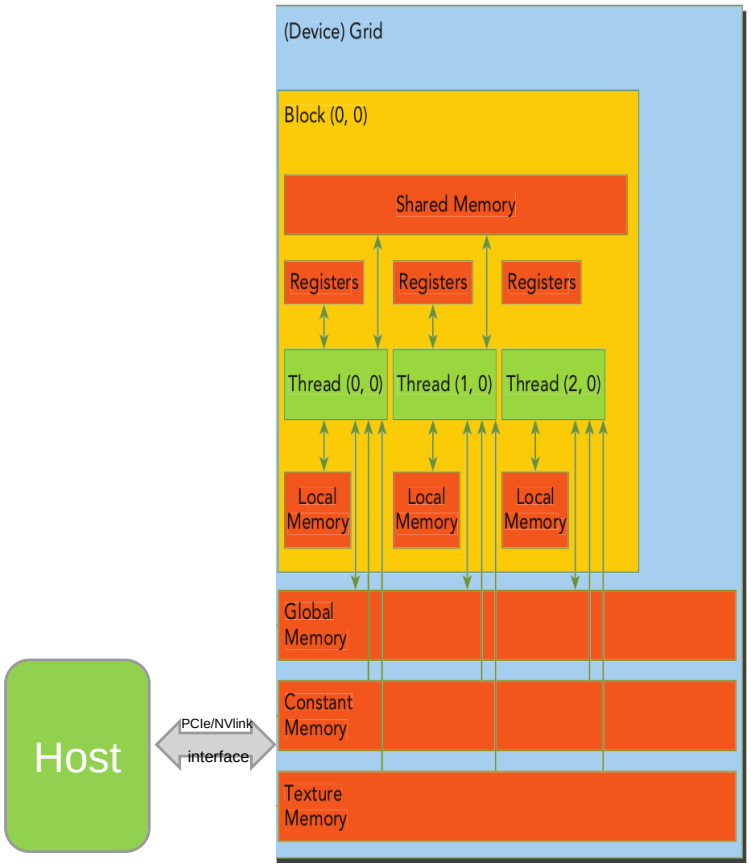




- Thread blocks can be configured as 1D/2D/3D.
  - Each thread has a unique ID in a block
- For 1D case, unique thread ID is stored in CUDA built-in variable ***threadIdx.x*** and threads with consecutive indices are grouped into warps. Eg. Warp 0 : Thread 0, 1, 2, …. thread 31 etc.
- For 2D case, unique thread ID is computed as: ***threadIdx.y * blockDim.x + threadIdx.x***
- For 3D case, unique thread ID is computed as: ***threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x***

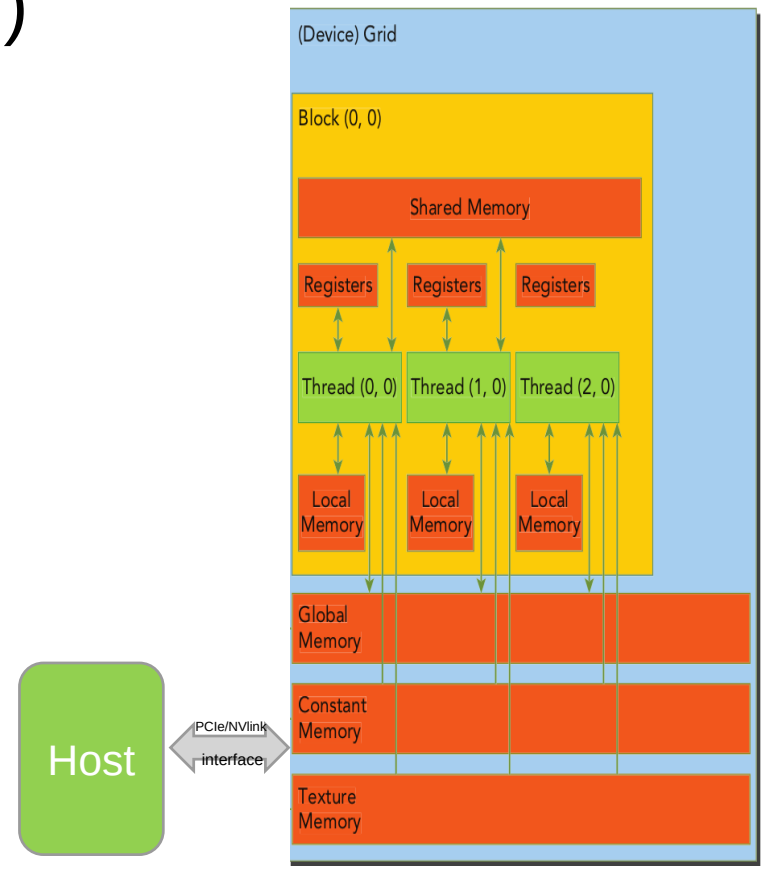
CUDA Memory Model

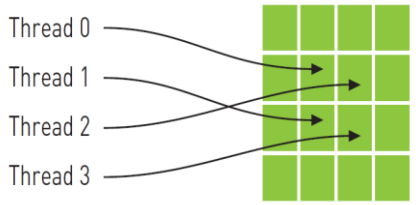
- ***Registers***
  - Fastest memory space on GPU
  - Automatic variable declared in a kernel without any qualifiers is stored in the registers
  - Private to each thread
  - Register variables share their lifetime with the kernel. Once the kernel execution is complete, the register variables are not accessible again
- ***Local memory***
  - Variables in the kernels that cannot be fitted in register space spills into the local memory (eg. large local structures or arrays)
  - Reside in same physical location as the global memory (so accesses are characterized by high latency)
- ***Shared memory***
  - Use __***shared***__ attribute to variables.
  - This is an ***on-chip L1 cache***, 10x lower latency compared to Local/Global memory.
  - Each SM has limited shared memory which is partitioned among all thread blocks (usually up to 196Kb with modern GPUs)
  - Serves as a means for inter-thread communication: use __***syncthreads()*** for synchronizing shared memory accesses
  - Has lifetime of the block, accessible only to all the threads within the block



CUDA Memory Model (cont.)

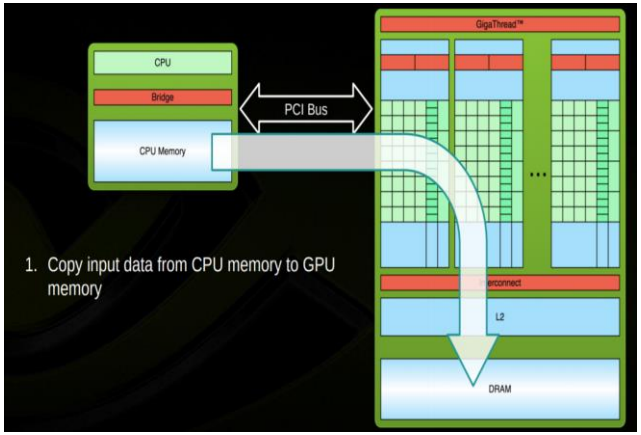
- ***Global memory***
  - Use ***__device__*** attribute
  - Resides in global memory space, lifetime of the CUDA context in which it is created
  - accessible from all the threads within the grid
- ***Constant memory***
  - Use ***__constant__*** attribute along with variable type, very limited amount of constant memory is generally present (re-routed via ***on-chip L1-L2 caches***)
  - lifetime of the CUDA context in which it is created
  - accessible from all the threads within the grid
- ***Texture memory***
  - Texture memory is read-only device memory
  - Can be accessed using the device functions described in Texture Functions, reading a texture using one of these functions is called a texture fetch
  - The texture memory system is designed to be efficient for spatially-localized accesses in 2D arrays; its lifetime is the CUDA context in which it is created
  - See also the ***const*** ***__restrict__*** type qualifier for fast texture-like cache fetches (re-routed via the ***on-chip L2 cache***) in our Argo CUDA Coding Standard



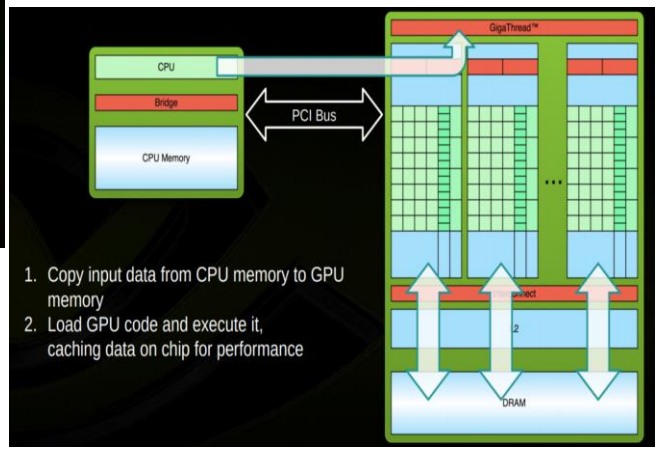


Texture Memory

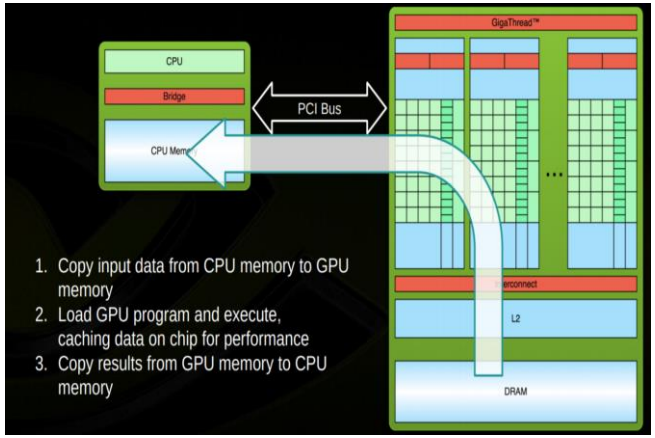
CUDA Usual Programming Workflow



// execute the Kernel on GPU
add<<<(N + 255) / 256, 256>>>(dev_a, dev_b, dev_c);



// allocate Memory on GPU
*int *dev_a, *dev_b, *dev_c;*
*err = cudaMalloc((void**)&dev_a, N * sizeof(int));*
….
// copy the host buffer content from Host-to-Device Memory
*err = cudaMemcpy(dev_a, src_a, N * sizeof(int),*
cudaMemcpyHostToDevice);
….

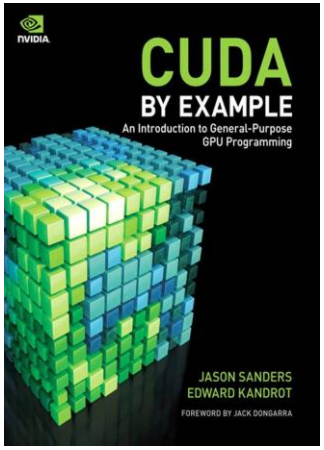


// copy the output from device buffer to host buffer (Device-to-Host)
*err = cudaMemcpy(src_c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);*
….

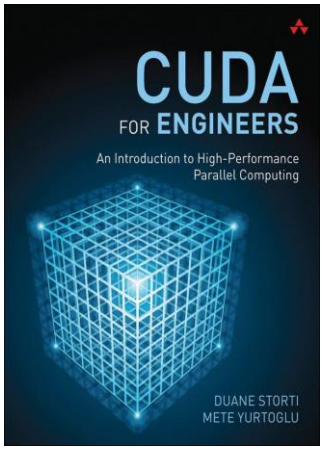
CUDA Introductory Books

- Good source material for our introductory course

- CUDA By Example: An Introduction (2010)
  - Introductory book recommended by Nvidia (free book)



- CUDA for Engineers (2015)
  - Introductory book with usable case studies & code

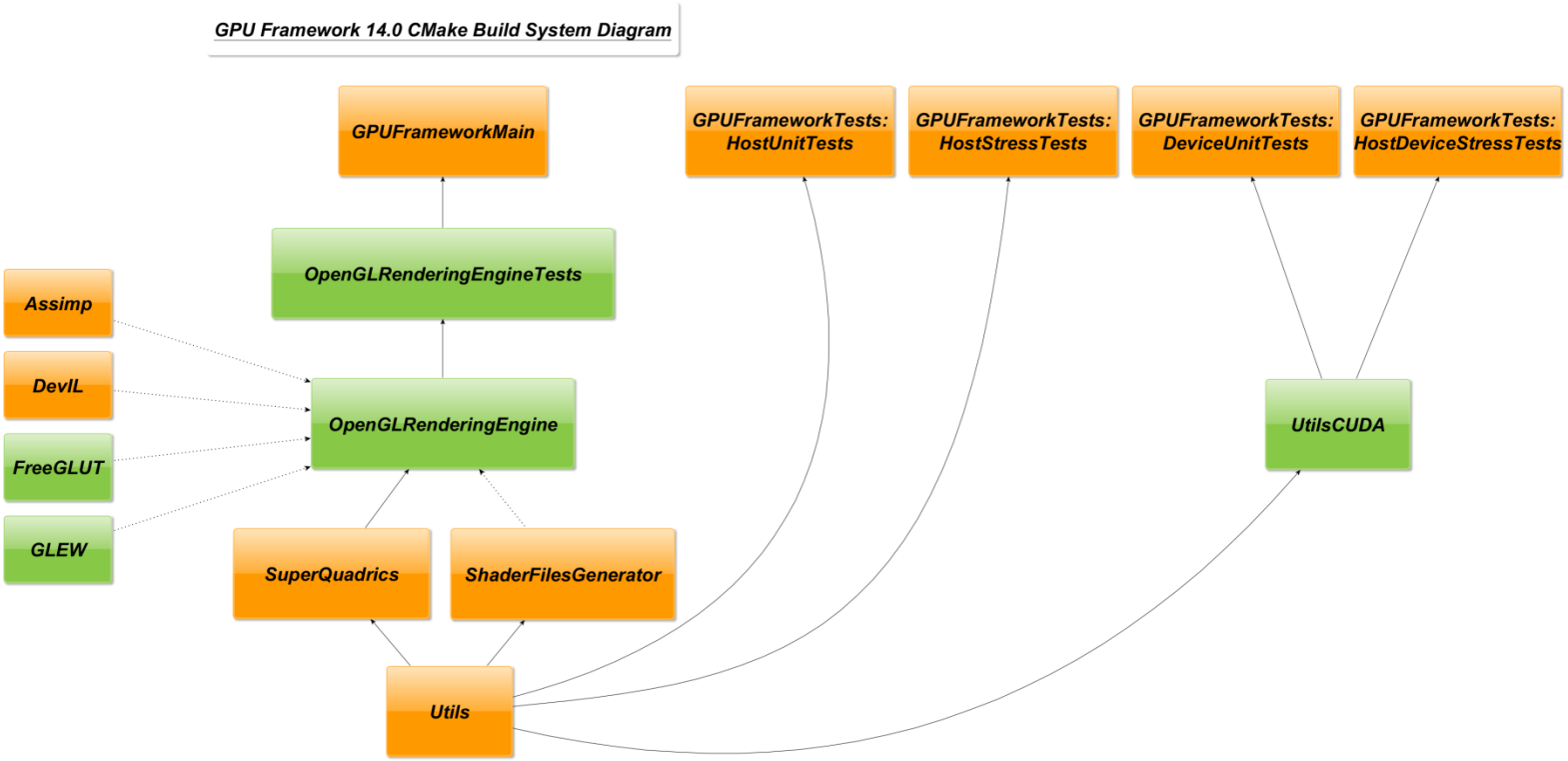


Part 3

Practical:
Open source demo code presented

The GPU Primer

- ***Simple Point Cloud Transformation example***
- *Let's call it the **GPU Primer***
- *Showcases a CPU vs. GPU implementation*
  - *A plain (bare-bone) CUDART C API implementation*
  - *A dotred GPU framework based implementation*
    - *Let's follow the on-screen presentation!*



Open discussion

Final audience discussion and Q&A

Thank you for participating!

Contact:
Thanos Theo
Mail: thanos.theo@dotredconsultancy.com

linkedin profile

dotred website

open source demo code