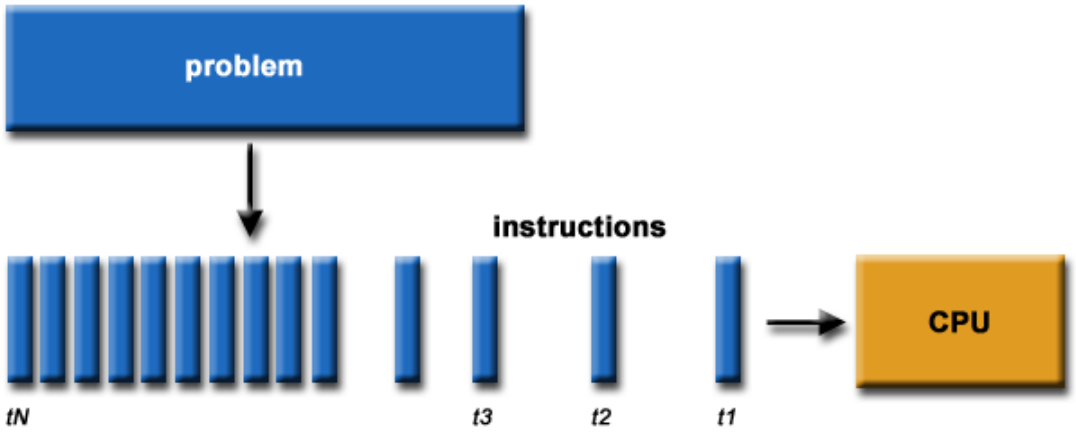
GPU Architecture and Programming

Andrei Doncescu inspired by NVIDIA

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

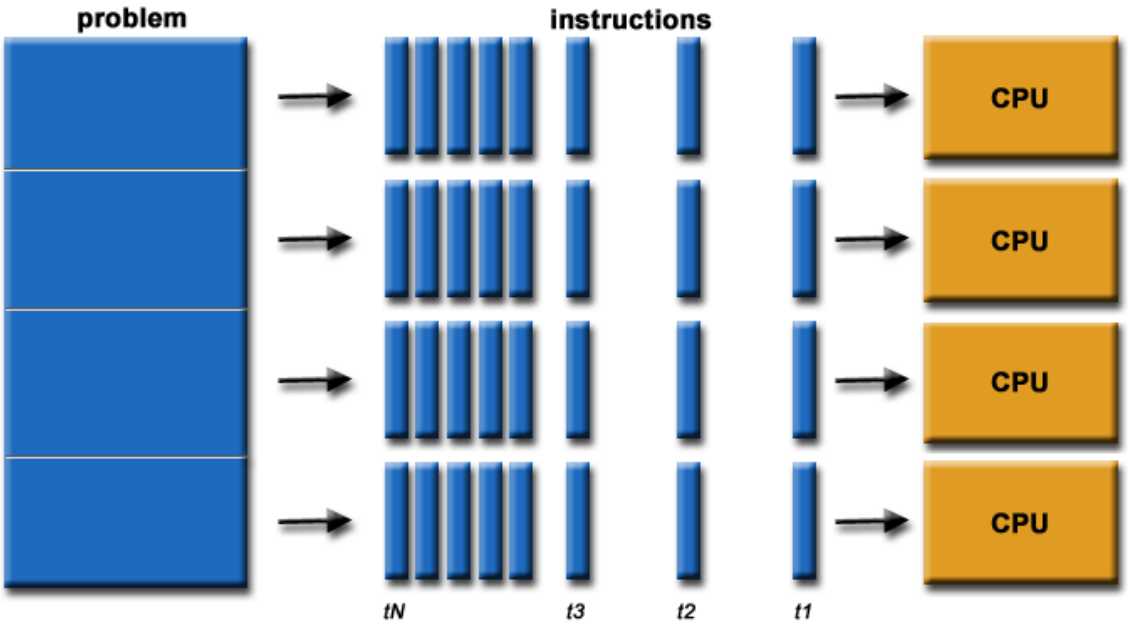
Official-sounding definition: The simultaneous use of multiple compute resources to solve a computational problem.

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

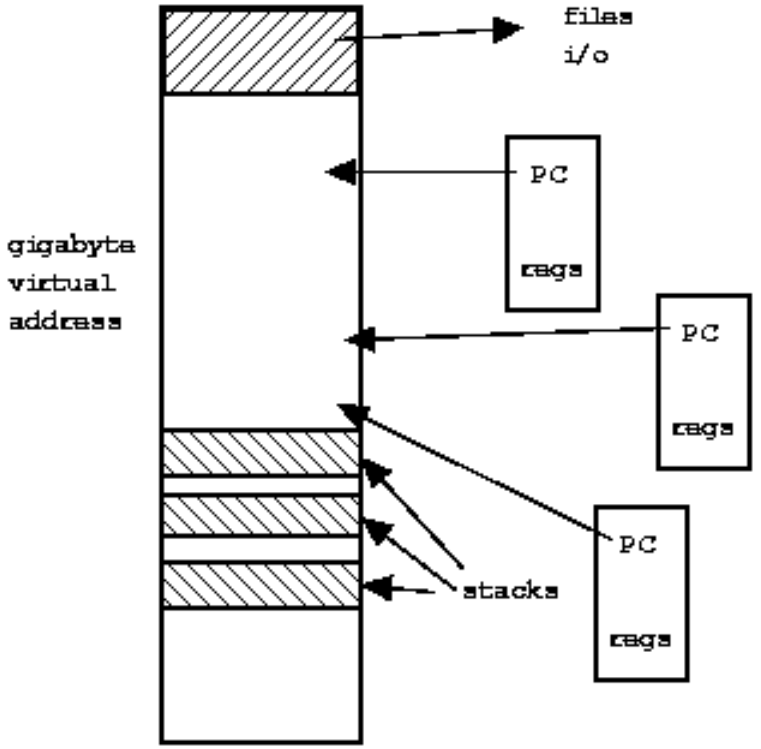


Processes and Threads

- Traditional process
  - One thread of control through a large, potentially sparse address space
  - Address space may be shared with other processes (shared mem)
  - Collection of systems resources (files, semaphores)
- Thread (light weight process)
  - A flow of control through an address space
  - Each address space can have multiple concurrent control flows
  - Each thread has access to entire address space
  - Potentially parallel execution, minimal state (low overheads)
  - May need synchronization to control access to shared variables

Threads

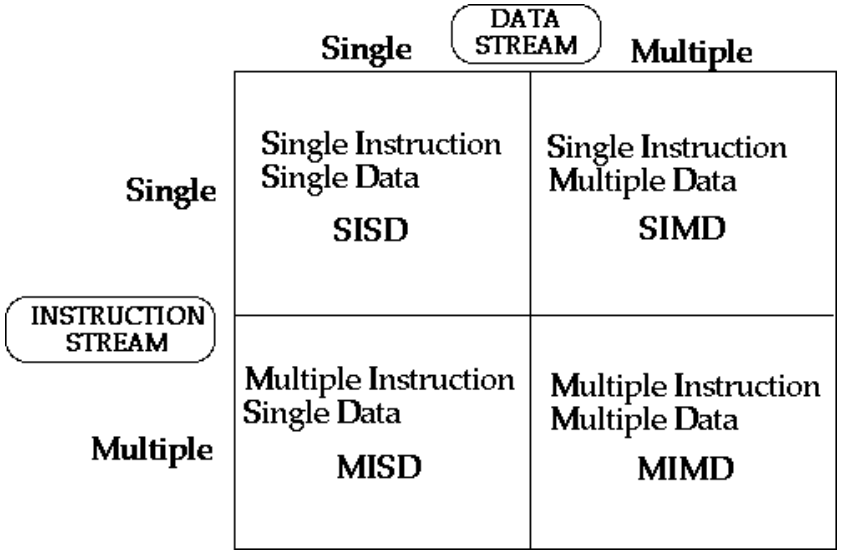
- Each thread has its own stack, PC, registers
  - Share address space, files,...



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.



What is GPGPU ?

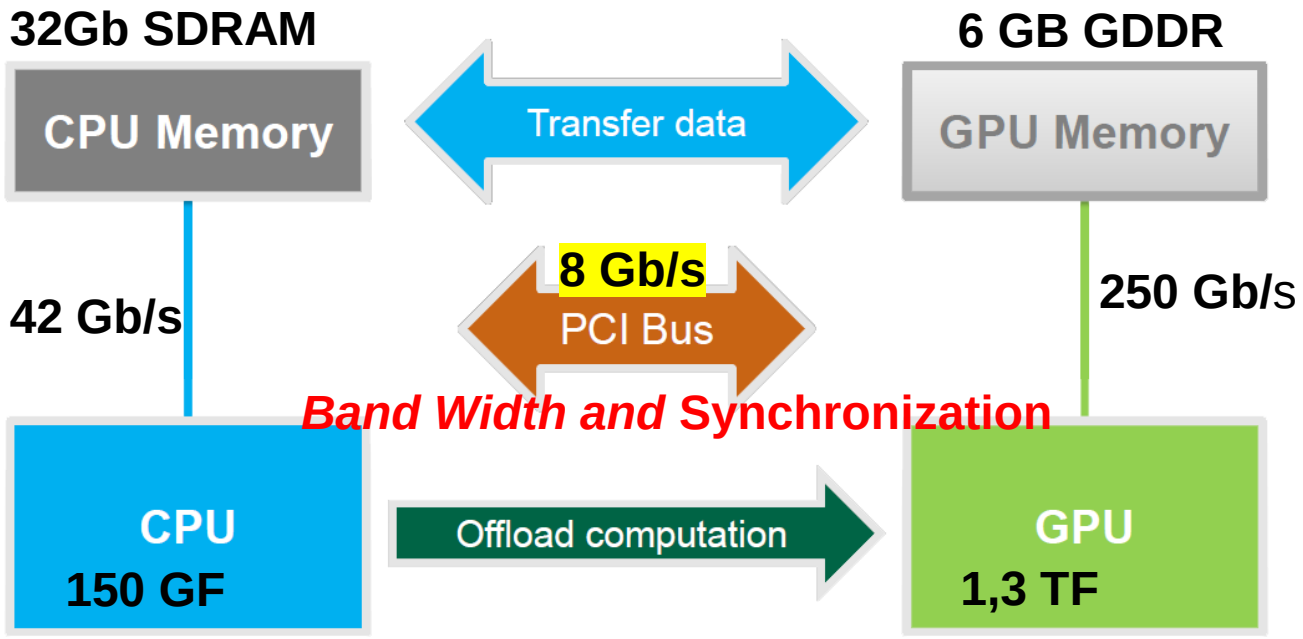
- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation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

- GPU – graphics processing unit
- Originally designed as a graphics processor
- Nvidia's GeForce 256 (1999) – first GPU
  - single-chip processor for mathematically-intensive tasks
  - transforms of vertices and polygons
  - lighting
  - polygon clipping
  - texture mapping
  - polygon rendering

Modern GPUs are present in

- ✓ Embedded systems
- ✓ Personal Computers
- ✓ Game consoles
- ✓ Mobile Phones
- ✓ Workstations

Basic Concepts

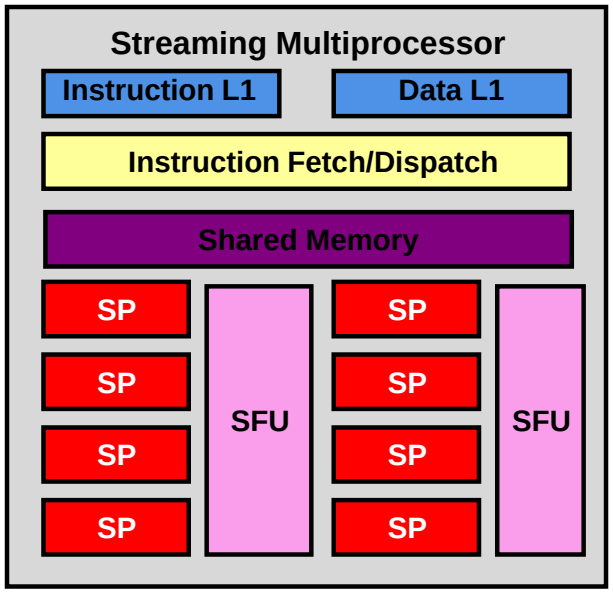


CUDA Processor Terminology

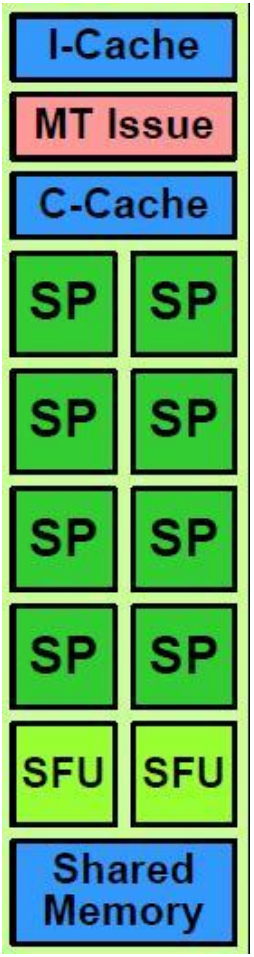
- SPA
  - Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800)
- TPC
  - Texture Processor Cluster (2 SM + TEX)
- SM
  - Streaming Multiprocessor (8 SP)
  - Multi-threaded processor core
  - Fundamental processing unit for CUDA thread block
- SP
  - Streaming Processor
  - Scalar ALU for a single CUDA thread

Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 512 threads active
  - Shared instruction fetch per 32 threads
  - Cover latency of texture/memory loads
- 20+ GFLOPS
- 16 KB shared memory
- texture and global memory access



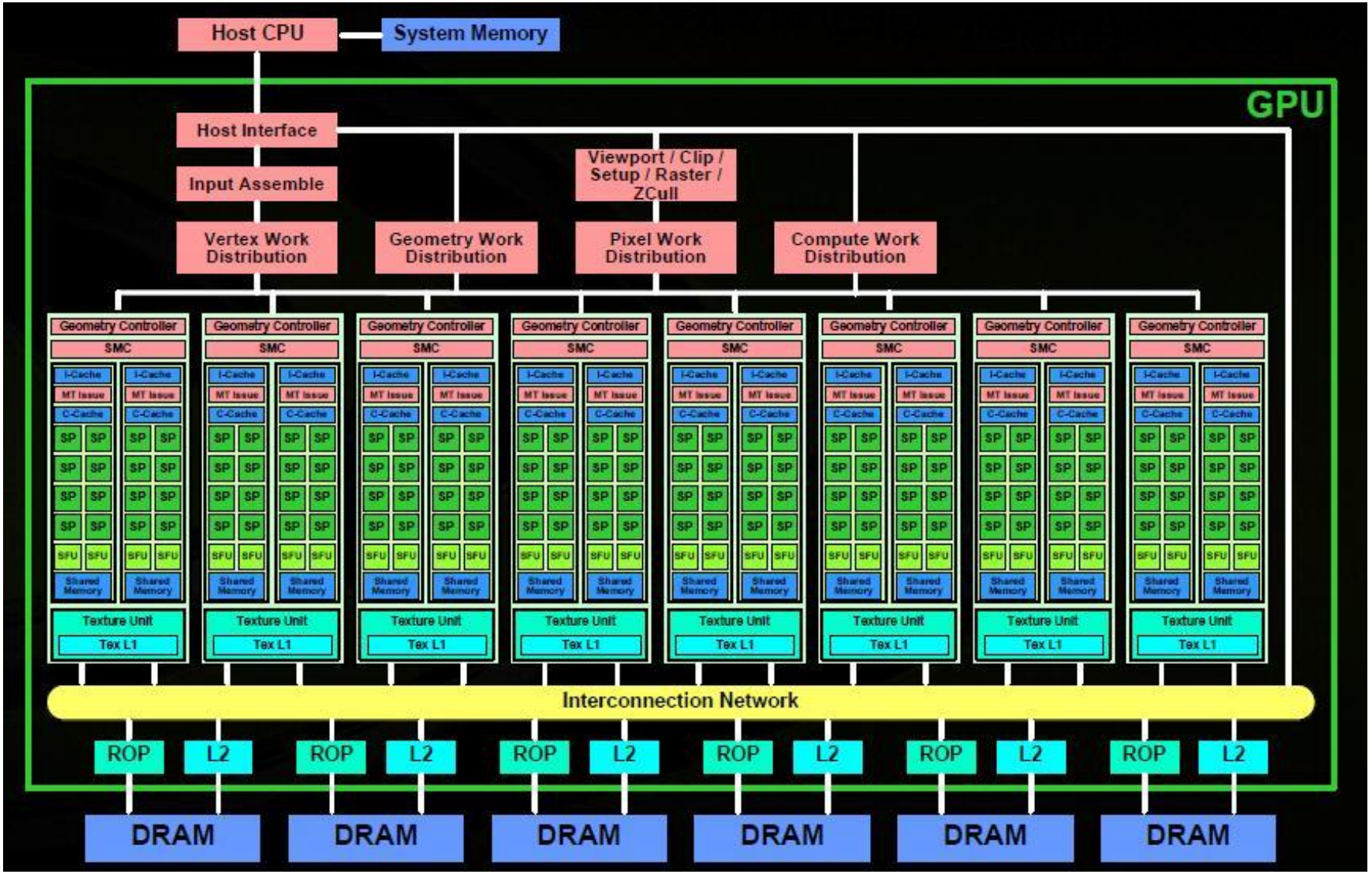
Streaming Multiprocessor (SM)



- Each SM has 8 Scalar Processors (SP)
- IEEE 754 32-bit floating point support (incomplete support)
- Each SP is a 1.35 GHz processor (32 GFLOPS peak)
- Supports 32 and 64 bit integers
- 8,192 dynamically partitioned 32-bit registers
- Supports 768 threads in hardware (24 SIMT warps of 32 threads)
- Thread scheduling done in hardware
- 16KB of low-latency shared memory
- 2 Special Function Units (reciprocal square root, trig functions, etc)

Each GPU has 16 SMs…

The GPU


Host CPU
System Memory
GPU
Host Interface
Input Assemble
Viewport / Clip / Setup / Raster / ZCull
Vertex Work Distribution
Geometry Work Distribution
Pixel Work Distribution
Compute Work Distribution
Geometry Controller
SMC
I-Cache
MT Issue
C-Cache
SP
SFU
Shared Memory
Texture Unit
Tex L1
Interconnection Network
ROP
L2
DRAM

Scalar Processor

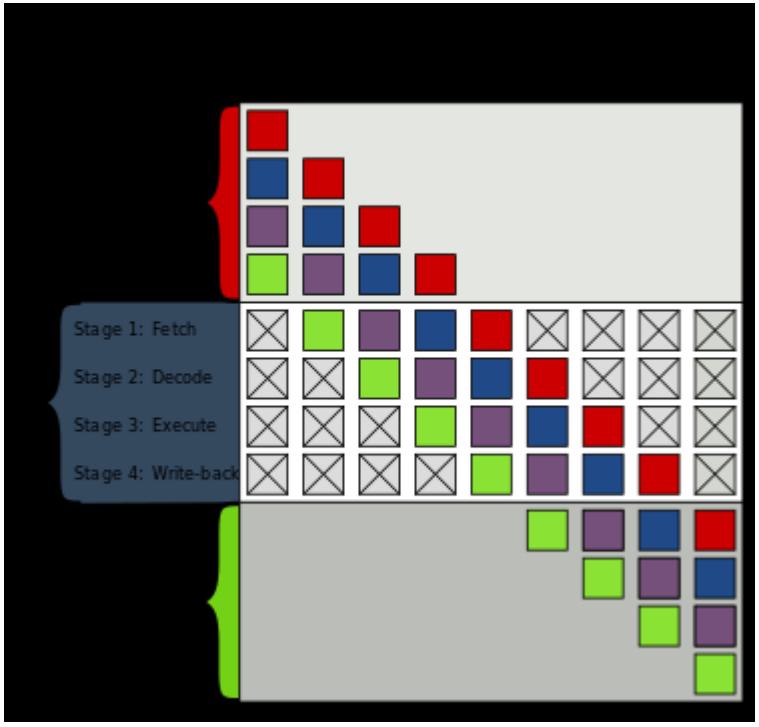
- Supports 32-bit IEEE floating point instructions:

  FADD, FMAD, FMIN, FMAX,
  FSET, F2I, I2F

- Supports 32-bit integer operations

  IADD, IMUL24, IMAD24,
  IMIN, IMAX, ISET, I2I, SHR,
  SHL, AND, OR, XOR

- Fully pipelined



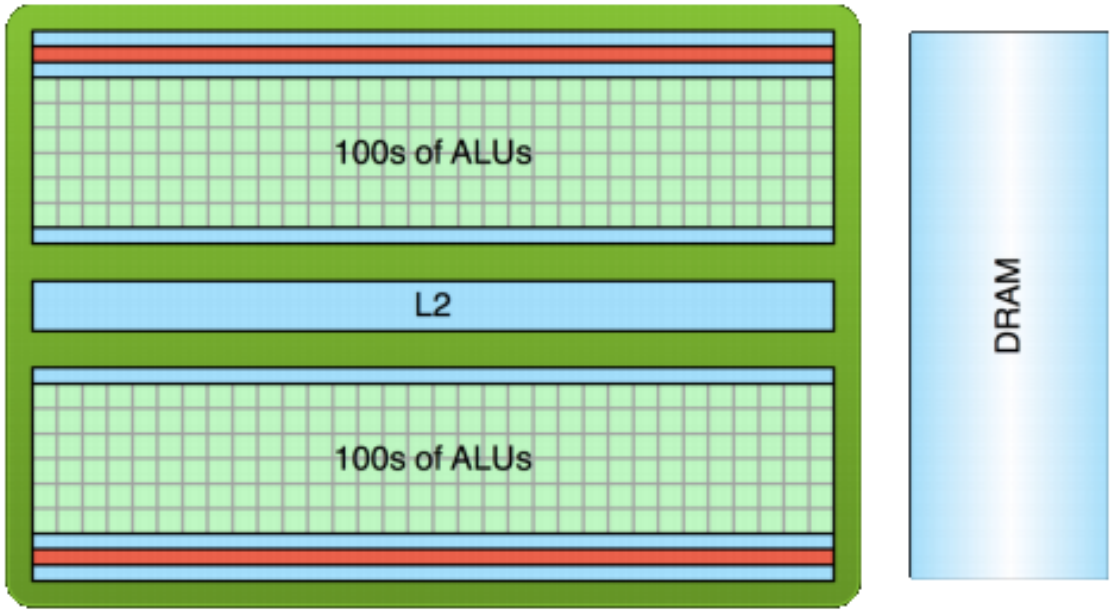
GPU computing

GPU: Graphics Processing Unit

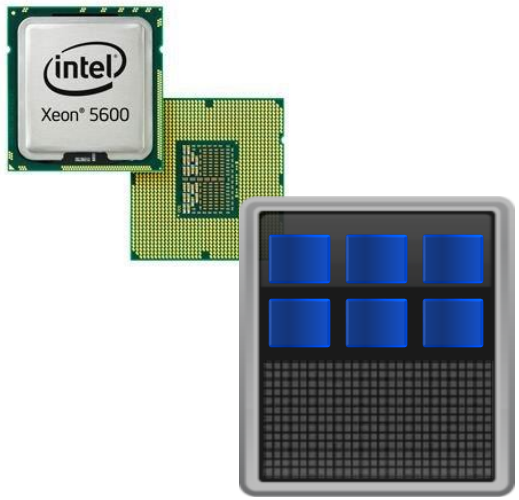
Traditionally used for real-time rendering

High Computational density and memory bandwidth

Throughput processor: 1000s of concurrent threads to hide latency

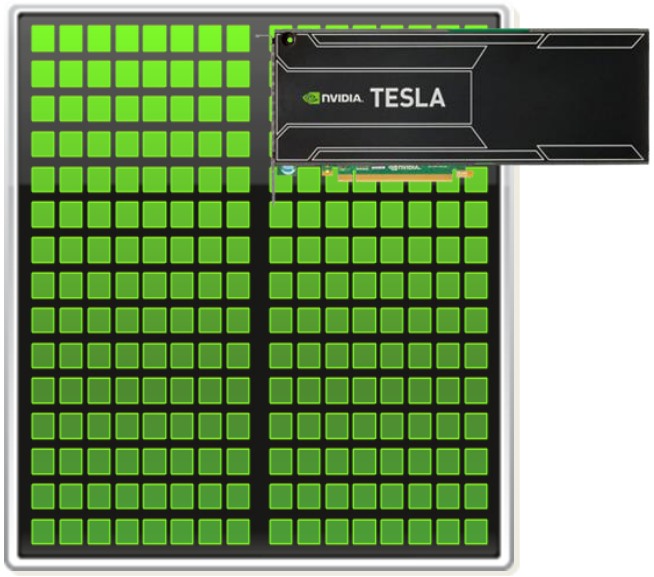


CPU



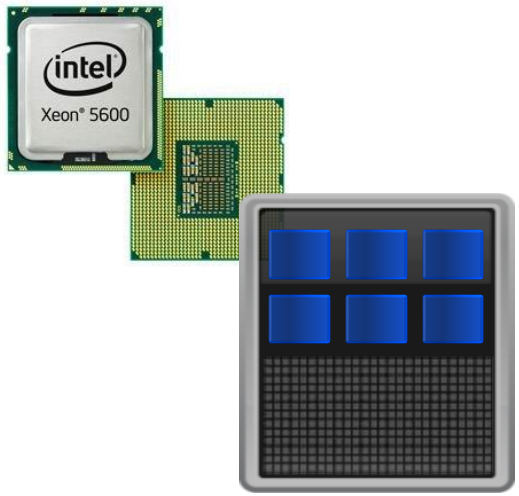
CPUs consist of a few cores optimized for serial processing

GPU

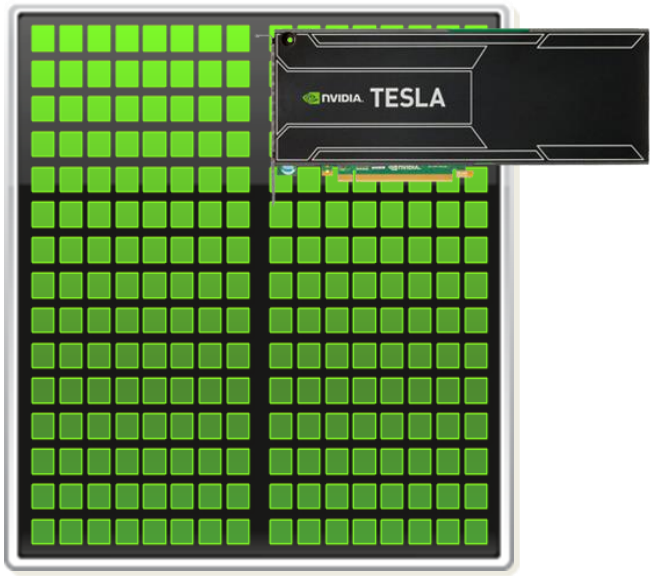


GPUs consist of hundreds or thousands of smaller, efficient cores designed for parallel performance

SCC CPU



SCC GPU



Intel Xeon E5-2670:

Clock speed: **2.6** GHz
4 instructions per cycle
CPU - 1**6** cores

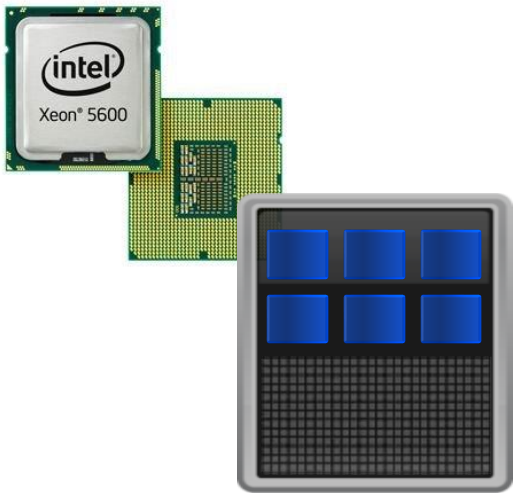
2.6 x 4 x 16 =
166.4 Gigaflops double precision

NVIDIA Tesla K40:

Single instruction
2880 CUDA cores

1.66 Teraflops double precision

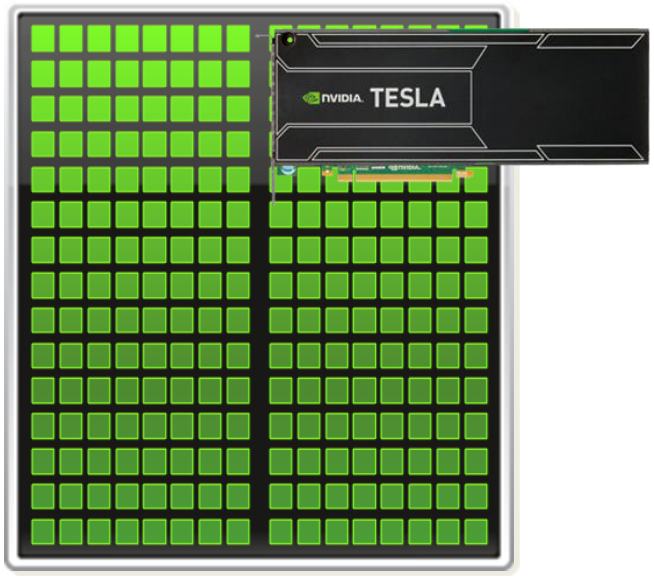
SCC CPU



Intel Xeon E5-2670 :

Memory size: **256** GB
Bandwidth: **32** GB/sec

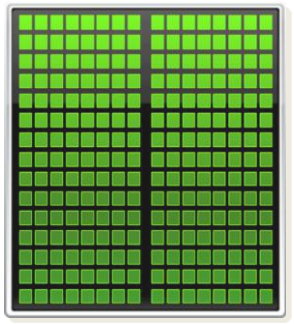
SCC GPU

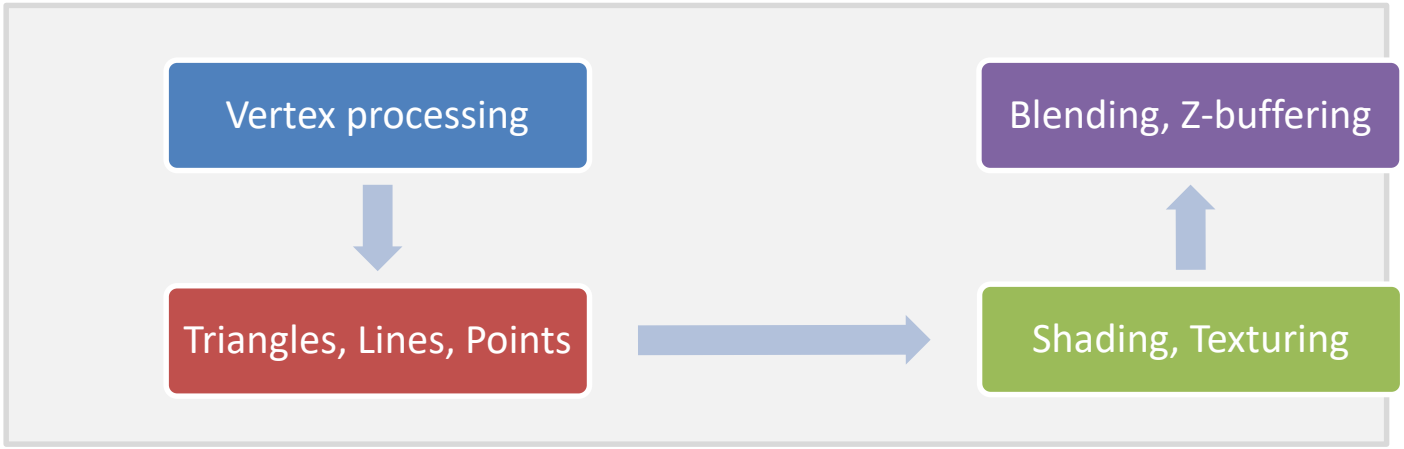


NVIDIA Tesla K40 :

Memory size: **12GB** total
Bandwidth: **288** GB/sec

Traditional GPU workflow

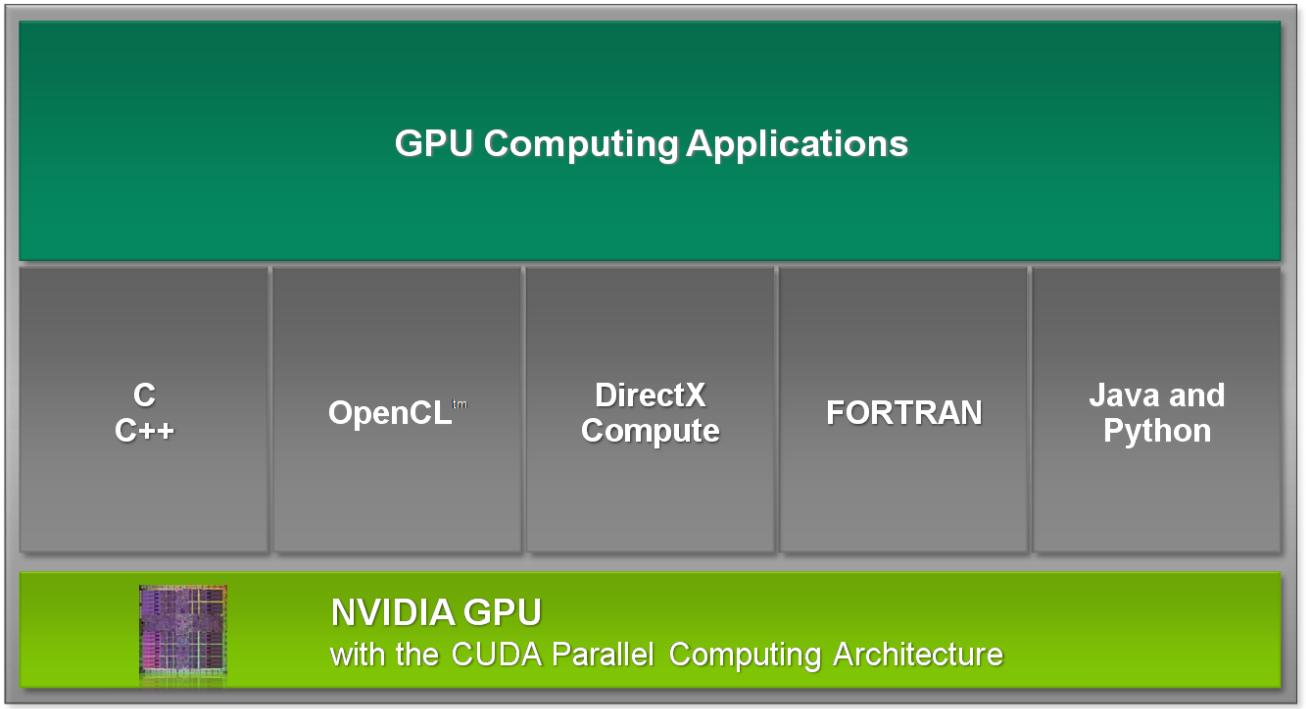




Enter CUDA

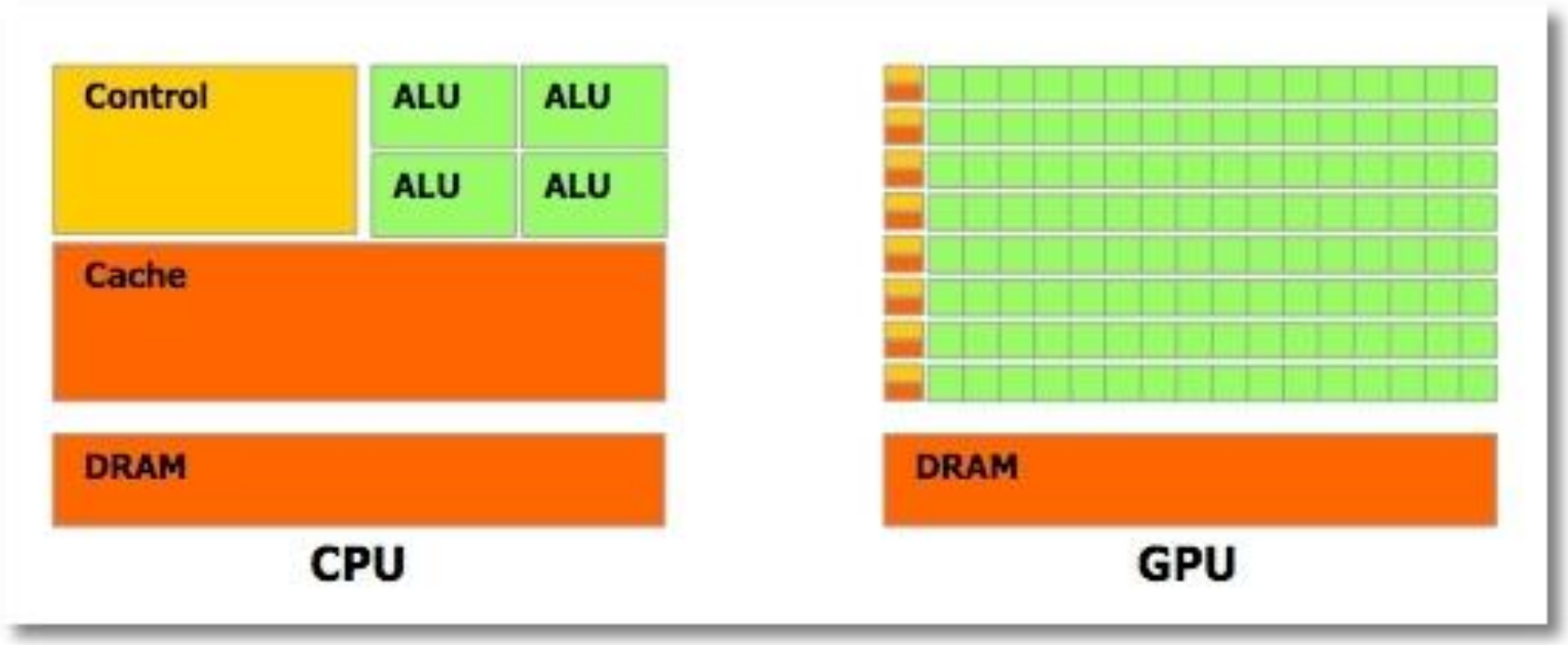
CUDA is NVIDIA's general purpose parallel computing architecture .

- 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



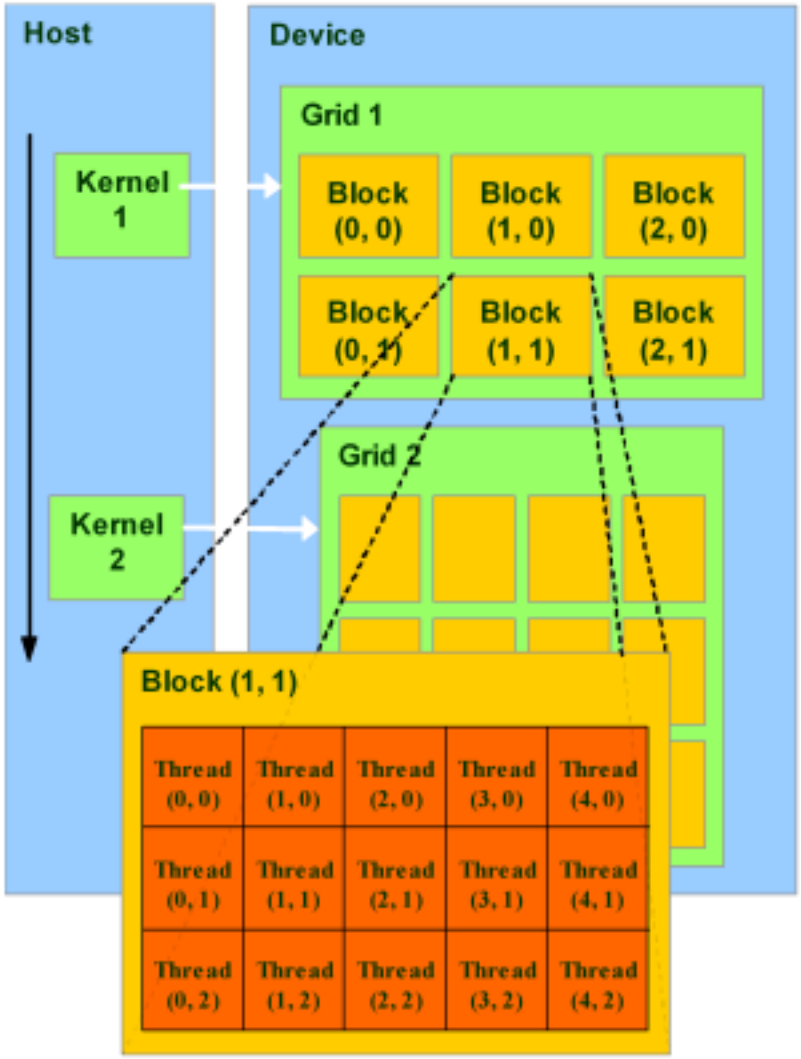
CUDA Goals

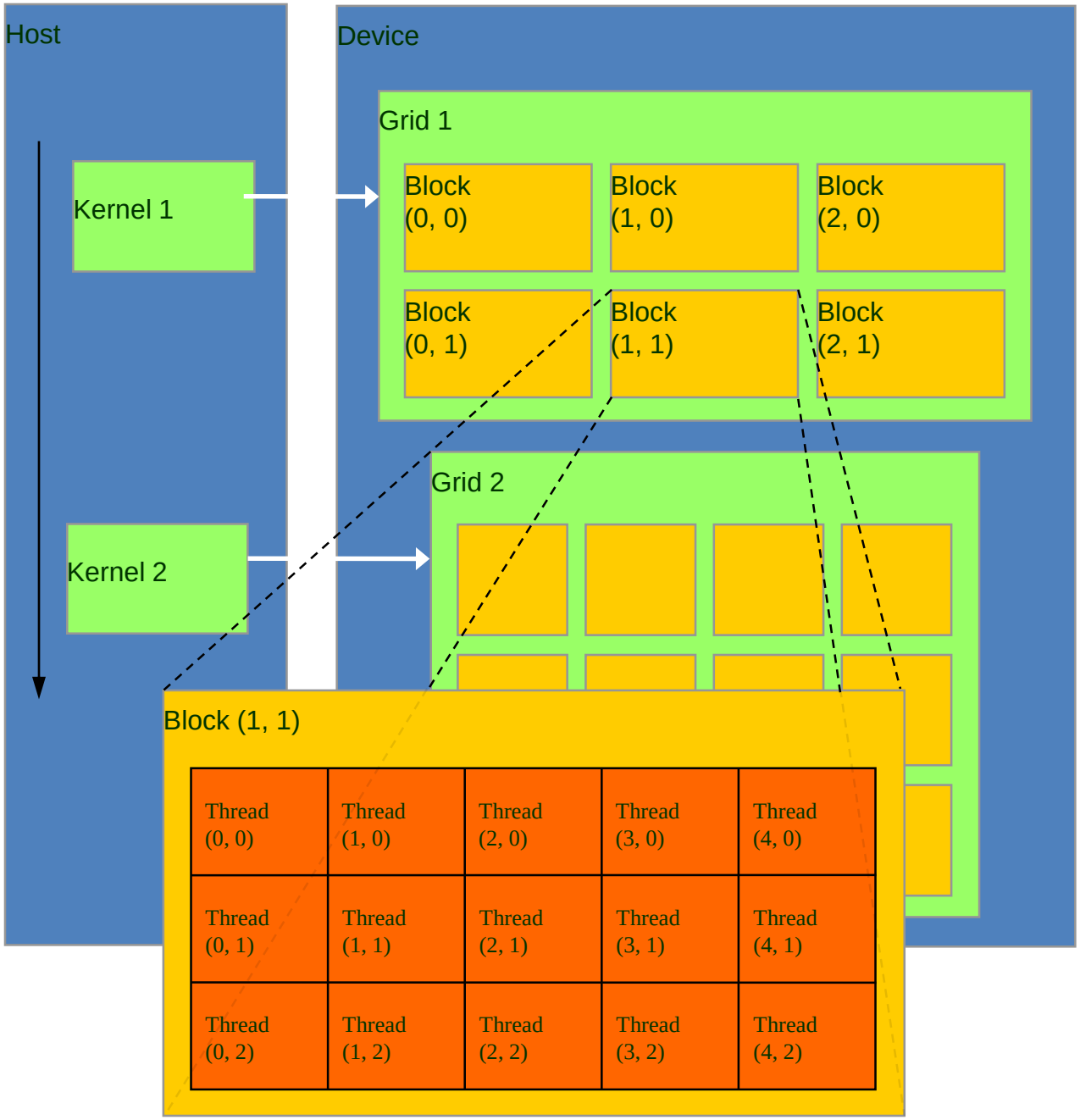
- Scale code to hundreds of cores running thousands of threads
- The task runs on the gpu independently from the cpu



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




Host
Device
Grid 1
Kernel 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Kernel 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

Global and Shared Memory

Global memory not cached on G8x GPUs

- **High latency, but launching more threads hides latency**
- **Important to minimize accesses**
- **Coalesce global memory accesses (more later)**

Shared memory is on-chip, very high bandwidth

- **Low latency (100-150times faster than global memory)**
- **Like a user-managed per-multiprocessor cache**
- **Try to minimize or avoid bank conflicts (more later)**

Texture and Constant Memory

Texture partition is cached

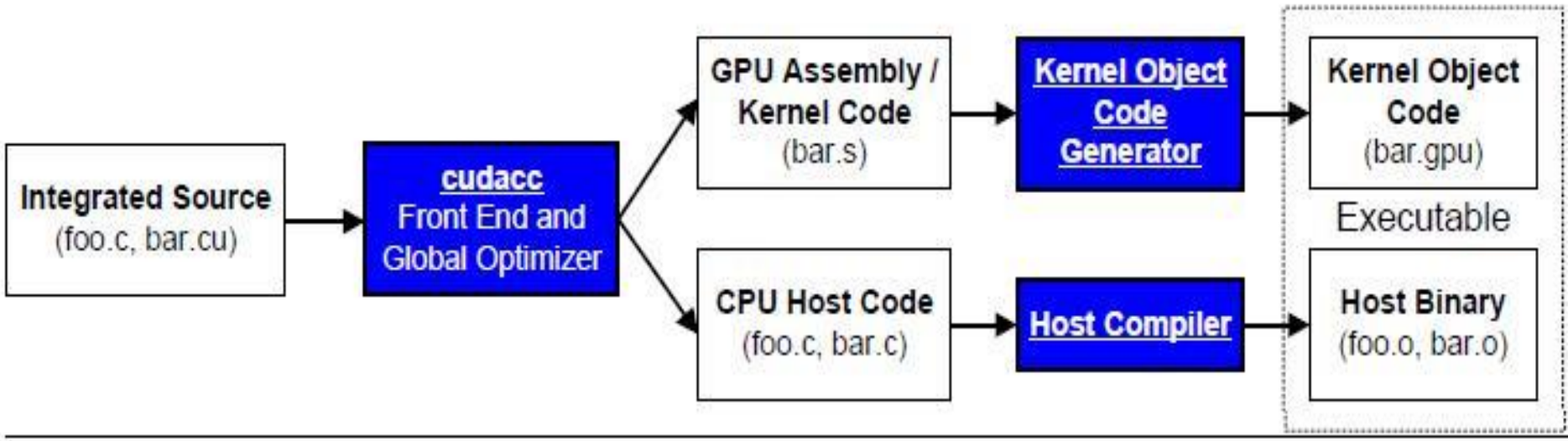
- **Uses the texture cache also used for graphics**
- **Optimized for 2D spatial locality**
- **Best performance when threads of a warp read locations that are close together in 2D**

Constant memory is cached

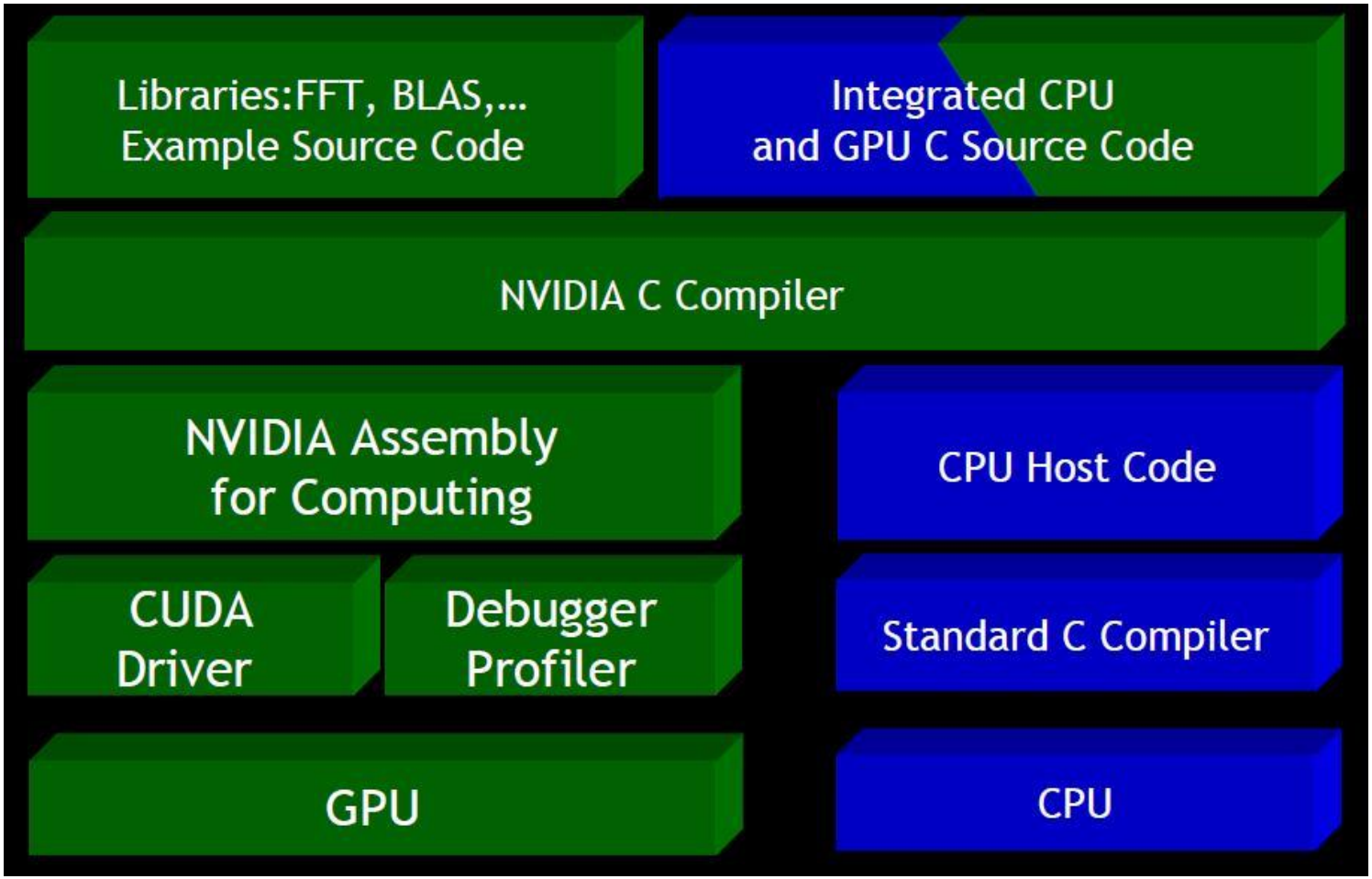
- **4 cycles per address read within a single warp**
- **Total cost 4 cycles if all threads in a warp read same address**
- **Total cost 64 cycles if all threads read different addresses**

CUDA Compilation

- As a programming model, CUDA is a set of extensions to ANSI C
- CPU code is compiled by the host C compiler and the GPU code (kernel) is compiled by the CUDA compiler. Separate binaries are produced



CUDA Stack


Libraries:FFT, BLAS,...
Example Source Code
Integrated CPU
and GPU C Source Code
NVIDIA C Compiler
NVIDIA Assembly
for Computing
CPU Host Code
CUDA
Driver
Debugger
Profiler
Standard C Compiler
GPU
CPU

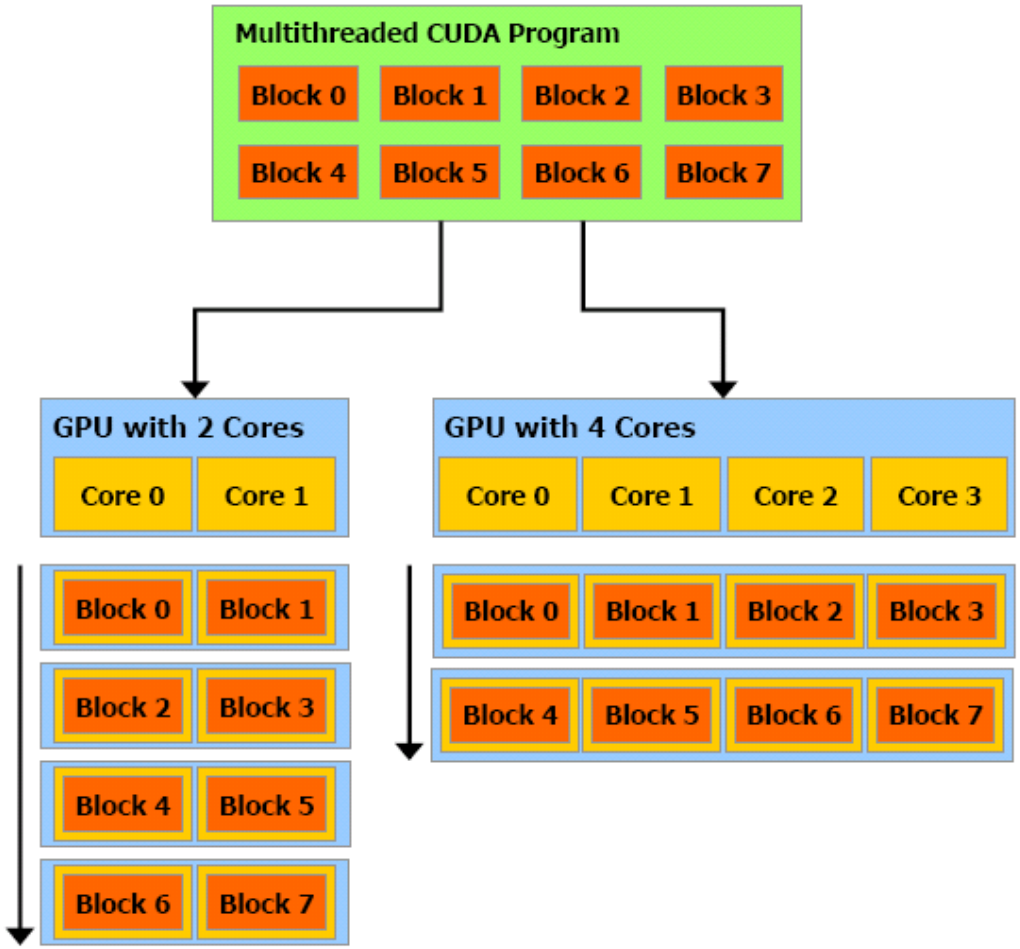
Limitations of CUDA

- Tesla does not fully support IEEE spec for double precision floating point operations
- Code only supported on NVIDIA hardware
- No use of recursive functions (can workaround)
- Bus latency between host CPU and GPU

(Although double precision will be resolved with Fermi)

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

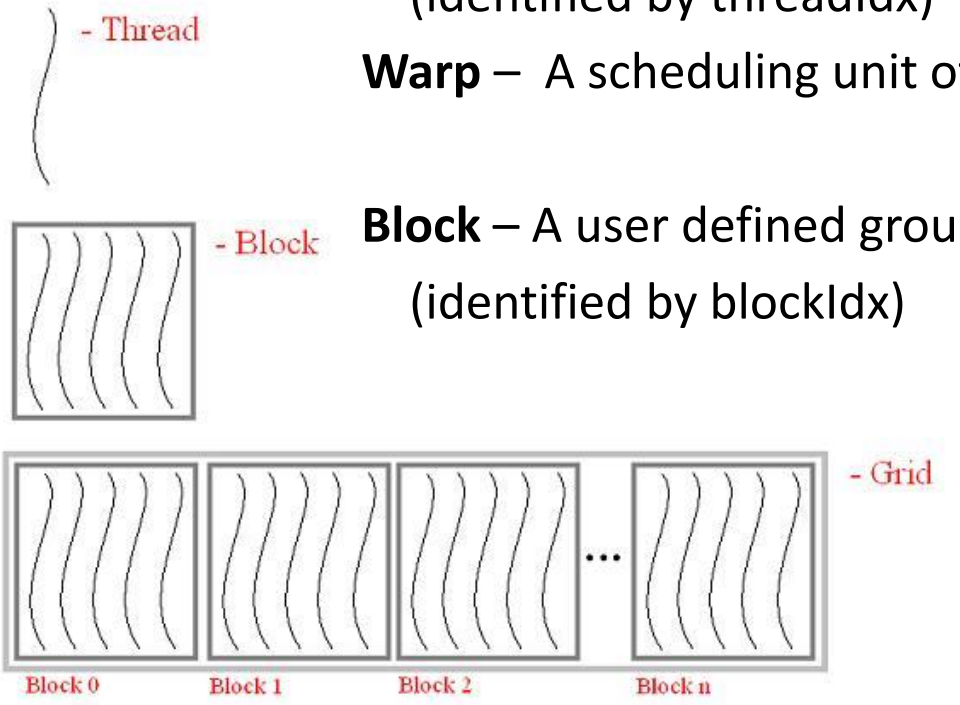
Unique thread id:

(blockIdx*blockDim)+threadIdx

Thread Hierarchy

Thread – Distributed by the CUDA runtime
(identified by threadIdx)

Warp – A scheduling unit of up to 32 threads



Block – A user defined group of 1 to 512 threads.
(identified by blockIdx)

Grid – A group of one or more blocks. A grid is created for each CUDA kernel function

CUDA - Memory Model

- Shared memory much much faster than global
- Don’t trust local memory
- Global, Constant, and Texture memory available to both host and cpu

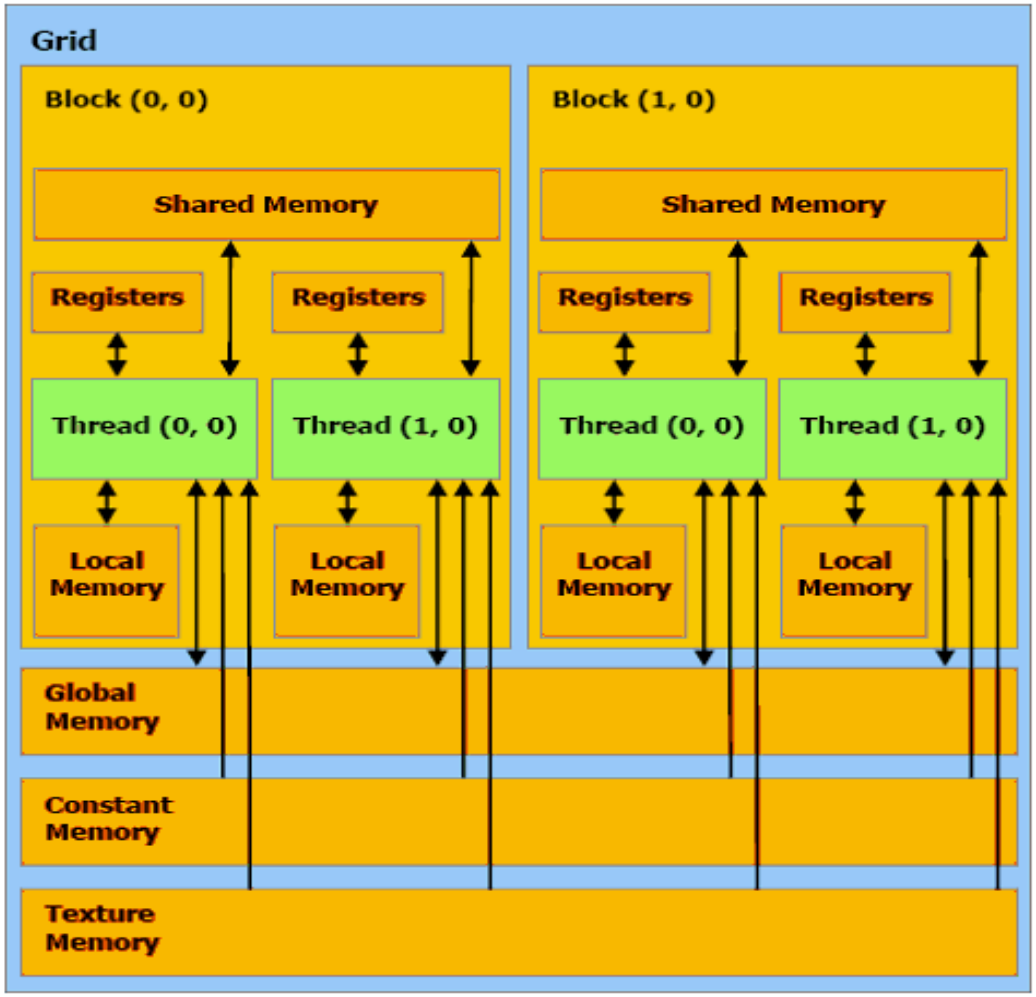


Diagram depicting memory organization.

(Rob Farber, "CUDA, Supercomputing for the Masses: Part 4", Dr.Dobbs, http:http://www.ddj.com/architect/208401741?pgno=3//www.ddj.com/hpc-high-performance-computing/207402986)

CUDA Memory Hierarchy

- The CUDA platform has three primary memory types

  Local Memory – per thread memory for automatic variables and register spilling.

  Shared Memory – per block low-latency memory to allow for intra-block data sharing and synchronization. Threads can safely share data through this memory and can perform barrier synchronization through _ _syncthreads()

  Global Memory – device level memory that may be shared between blocks or grids

CUDA - Memory Model (continue)

- Each block contain following:
  - Set of local registers per thread.
  - Parallel data cache or shared memory that is shared by all the threads.
  - Read-only constant cache that is shared by all the threads and speeds up reads from constant memory space.
  - Read-only texture cache that is shared by all the processors and speeds up reads from the texture memory space.

- Local memory is in scope of each thread. It is allocated by compiler from global memory but logically treated as independent unit.

CUDA - Memory Units Description

- Registers:
  - Fastest.
  - Only accessible by a thread.
  - Lifetime of a thread

- Shared memory:
  - Could be as fast as registers if no bank conflicts or reading from same address.
  - Accessible by any threads within a block where it was created.
  - Lifetime of a block.

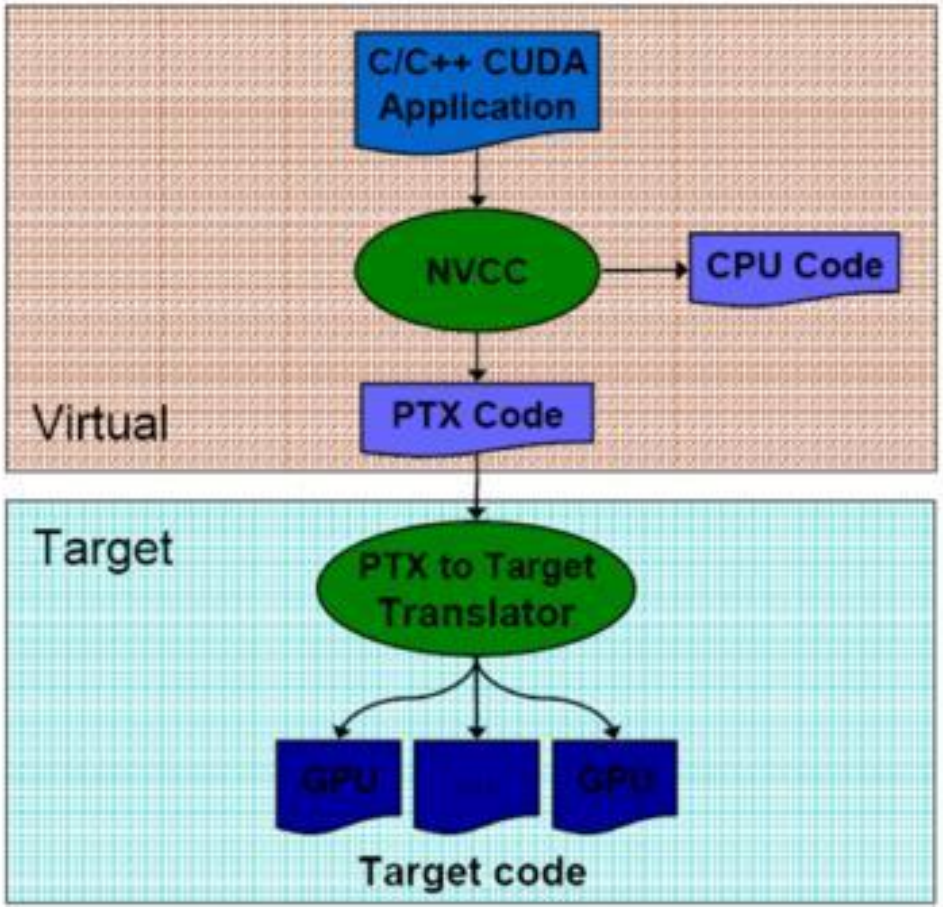
CUDA - Memory Units Description (continue)

- Global Memory:
  - Up to 150x slower then registers or share memory.
  - Accessible from either host or device.
  - Lifetime of an application.

- Local Memory
  - Resides in global memory. Can be 150x slower then registers and shared memory.
  - Accessible only by a thread.
  - Lifetime of a thread.

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);
}
```

CUDA: Hello, World! example

```
/* Main function, executed on host (CPU) */
int main( void) {

     /* print message from CPU */
     printf( "Hello Cuda!\n" );

     /* execute function on device (GPU) */
     hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();

     /* wait until all threads finish their job
*/
     cudaDeviceSynchronize();

     /* print message from CPU */
     printf( "Welcome back to CPU!\n" );

     return(0);
}
```

Kernel:
A parallel function that runs on the GPU

CUDA: Hello, World! example

```
/* Function executed on device (GPU */
__global__ void  hello( void) {

  printf( "\tHello from GPU: thread %d and block %d\n",
                                        threadIdx.x,
blockIdx.x );

}
```

CUDA: Hello, World! example

Compile and build the program using NVIDIA's **nvcc** compiler:

```
nvcc -o helloCuda helloCuda.cu -arch sm_20
```

Running the program on the GPU-enabled node:

helloCuda

```
Hello Cuda!
        Hello from GPU: thread 0 and block 0
        Hello from GPU: thread 1 and block 0
        . . .

        Hello from GPU: thread 6 and block 2
        Hello from GPU: thread 7 and block 2
Welcome back to CPU!
```

Note:
Threads are executed on "first come, first serve" basis. Can not expect any order!

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

Host Synchronization

All kernel launches are asynchronous

- control returns to CPU immediately
- kernel starts executing once all previous CUDA calls have completed

Memcopies are synchronous

- control returns to CPU once the copy is complete
- copy starts once all previous CUDA calls have completed

cudaThreadSynchronize()

- blocks until all previous CUDA calls complete

Asynchronous CUDA calls provide:

- non-blocking memcopies
- ability to overlap memcopies and kernel execution

The Big Difference

CPU program

```
void increment_cpu(float *a, float b, int N)
{
 for (int idx = 0; idx<N; idx++)
a[idx] = a[idx] + b;
}

void main()
{
…
increment_cpu(a, b, N);
}
```

GPU program

```
__global__ void increment_gpu(float *a, float b, int
      N)
 {
int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if( idx < N)                a[idx] = a[idx] + b;
}
void main() {
…..
dim3 dimBlock (blocksize);
 dim3 dimGrid( ceil( N / (float)blocksize)  )
      increment_gpu<<<dimGrid, dimBlock>>>(a, b,
      N);
}
```

CUDA: Vector Addition example

```
/* Main function, executed on host (CPU) */
int main( void) {


      /* 1. allocate memory on GPU */

      /* 2. Copy data from Host to GPU */

      /* 3. Execute GPU kernel */

      /* 4. Copy data from GPU back to Host */

      /* 5. Free GPU memory */

      return(0);
}
```

CUDA: Vector Addition example

```
/* 1. allocate memory on GPU */

float *d_A = NULL;
if (cudaMalloc((void **)&d_A, size) != cudaSuccess)
   exit(EXIT_FAILURE);

float *d_B = NULL;
cudaMalloc((void **)&d_B, size); /* For clarity we'll not check
for err */

float *d_C = NULL;
cudaMalloc((void **)&d_C, size);
```

CUDA: Vector Addition example

```
/* 2. Copy data from Host to GPU */

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);


cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

CUDA: Vector Addition example

```
/* 3. Execute GPU kernel */

/* Calculate number of blocks and threads */
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1) /
threadsPerBlock;

/* Launch the Vector Add CUDA Kernel */
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B,
d_C, numElements);

/* Wait for all the threads to complete */
cudaDeviceSynchronize();
```

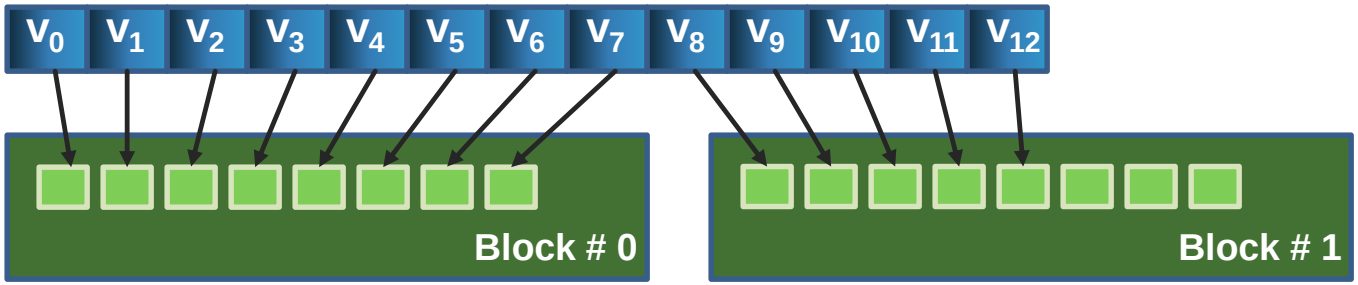
CUDA: Vector Addition example

```
/* 4. Copy data from GPU back to Host */

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

CUDA: Vector Addition example

```
/* 5. Free GPU memory */

cudaFree(d_A);

cudaFree(d_B);

cudaFree(d_C);
```

CUDA: Vector Addition example




```
/* CUDA Kernel */
__global__ void vectorAdd( const float *A,
                           const float *B,
                                 float *C,
                                 int numElements) {

        /* Calculate the position in the array */
        int i = blockDim.x * blockIdx.x + threadIdx.x;

        /* Add 2 elements of the array */
        if (i < numElements) C[i] = A[i] + B[i];
}
```

CUDA: Vector Addition example

```
/* To build this example, execute Makefile */
> make

/* To run, type vectorAdd: */
> vectorAdd

[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads *
Copy output data from the CUDA device to the host memory
Done

* Note: 196 x 256 = 50176 total threads
```

intro_gpu/vectorAdd/vectorAdd.cu

Mechanics of Using Shared Memory

- __shared__ type qualifier required
- Must be allocated from global/device function, or as "extern"
- Examples:

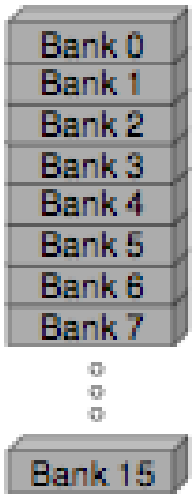
```
extern __shared__ float  d_s_array[];

/* a form of dynamic allocation */
/* MEMSIZE is size of per-block  */
/* shared memory*/
__host__ void outerCompute() {
   compute<<<gs,bs,MEMSIZE>>>();
}
__global__ void compute() {
    d_s_array[i] = …;
}
```

```
__global__ void compute2() {
   __shared__ float d_s_array[M];

   /* create or copy from global memory */
   d_s_array[j] = …;

  /* write result back to global memory */
   d_g_array[j] =  d_s_array[j];
}
```

Optimization using Shared Memory

Parallel Memory Architecture

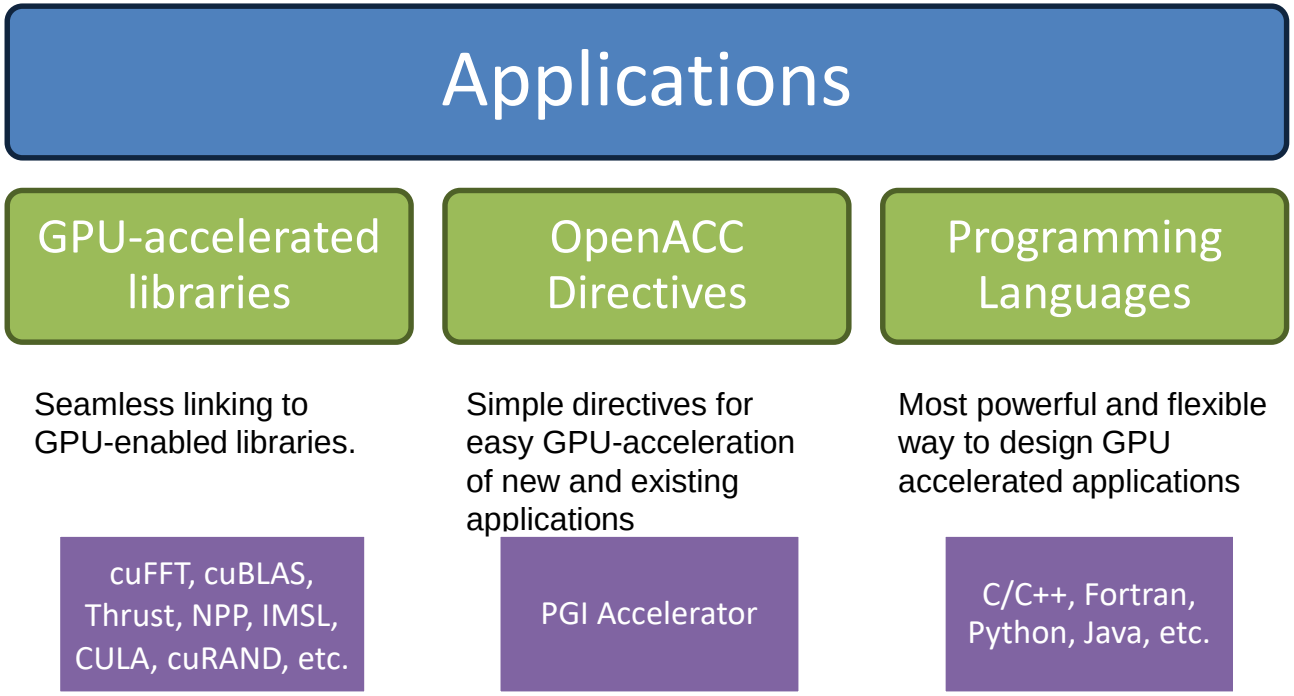
- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



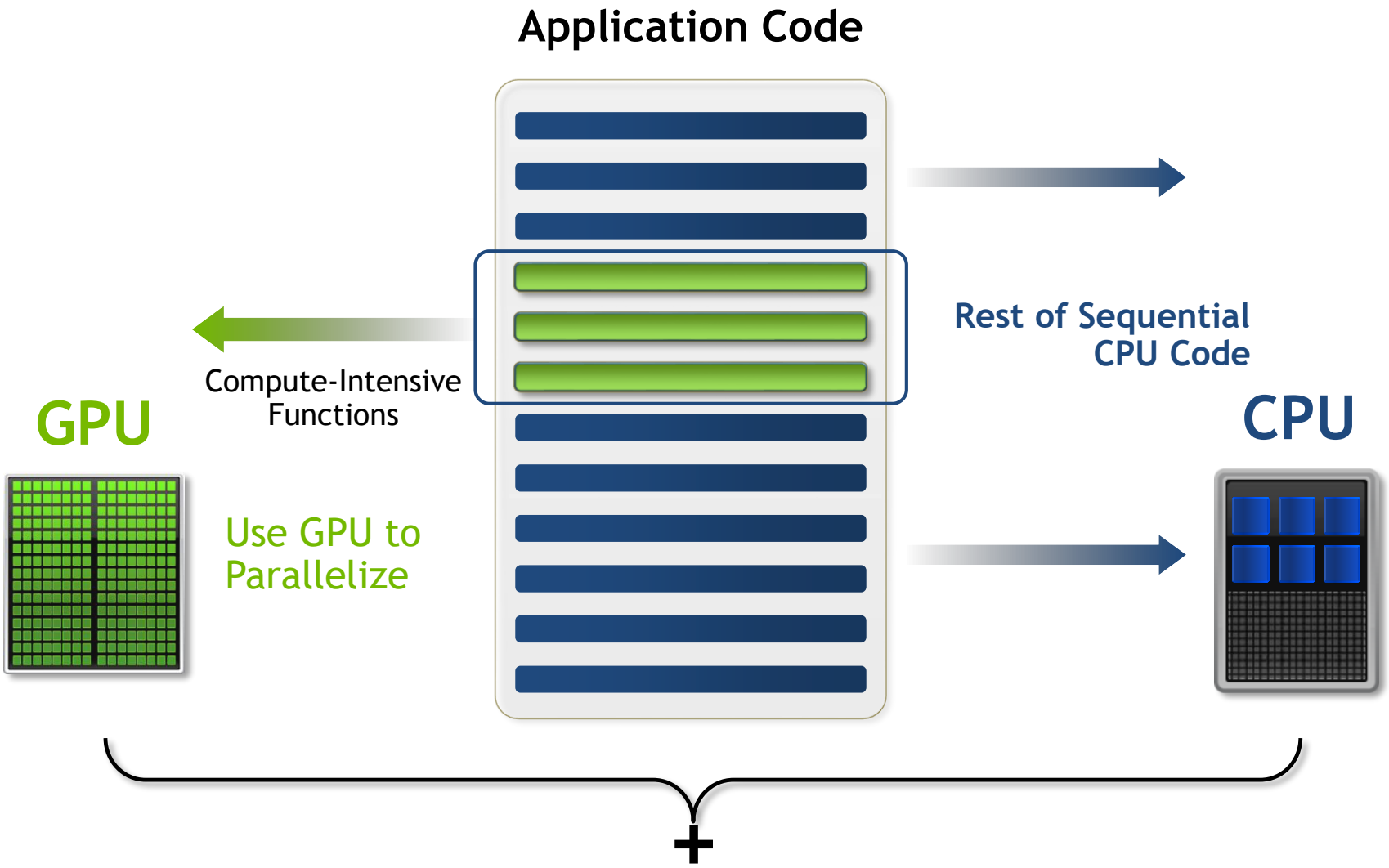
10x GPU Computing Growth

2008	2015
6,000 Tesla GPUs	**450,000** Tesla GPUs
150K CUDA downloads	**3M** CUDA downloads
77 Supercomputing Teraflops	**54,000** Supercomputing Teraflops
60 University Courses	**800** University Courses
4,000 Academic Papers	**60,000** Academic Papers

GPU Acceleration



Minimum Change, Big Speed-up



Will Execution on a GPU Accelerate My Application?

Computationally intensive—The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory.

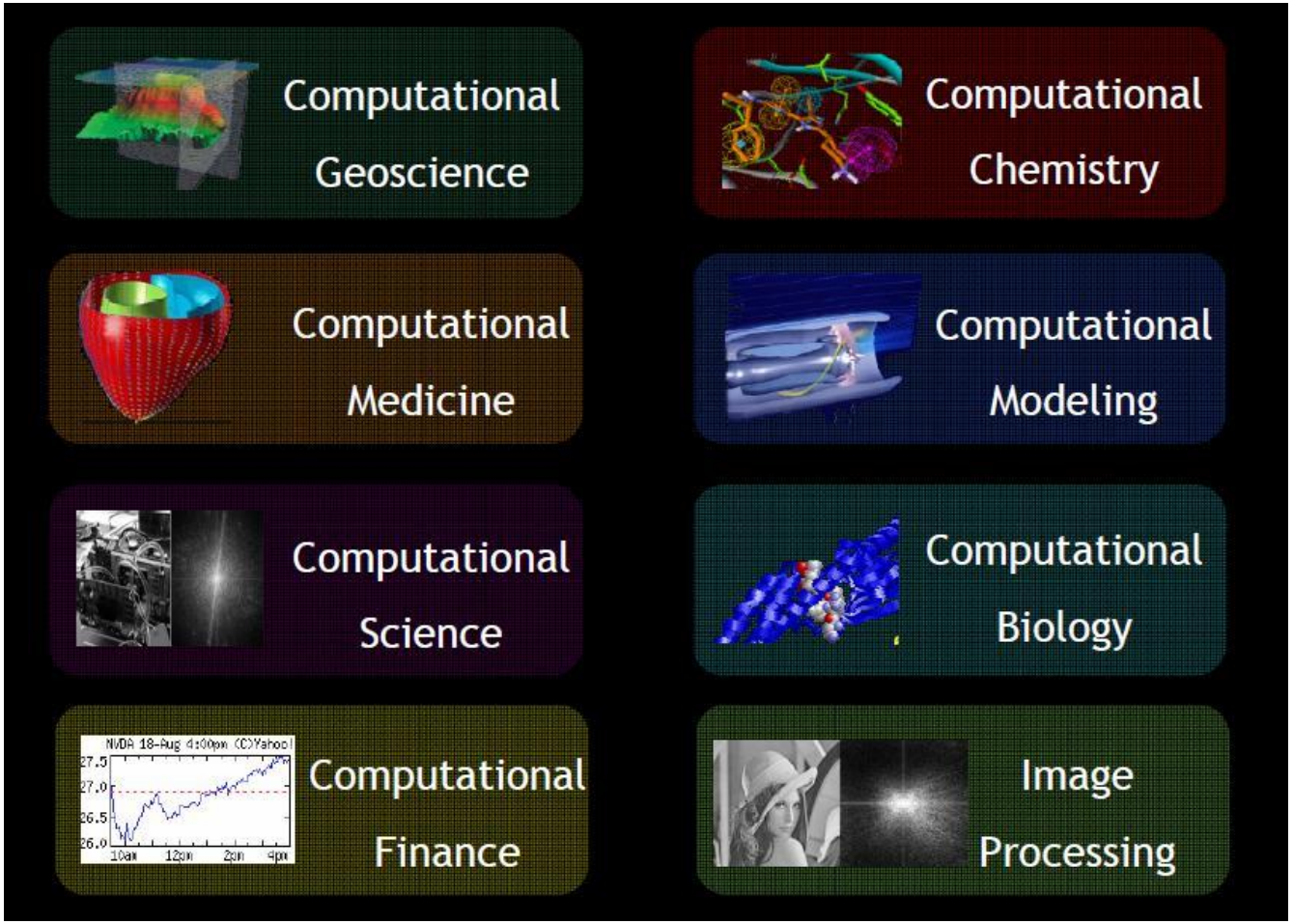
Massively parallel—The computations can be broken down into hundreds or thousands of independent units of work.

C	OpenACC, CUDA
C++	Thrust, CUDA C++
Fortran	OpenACC, CUDA Fortran
Python	PyCUDA
Numerical analytics	MATLAB, Mathematica

Myths About CUDA

- GPUs are the only processors in a CUDA application
  - The CUDA platform is a co-processor, using the CPU and GPU
- GPUs have very wide (1000s) SIMD machines
  - No, a CUDA Warp is only 32 threads
- Branching is not possible on GPUs
  - Incorrect.
- GPUs are power-inefficient
  - Nope, performance per watt is quite good
- CUDA is only for C or C++ programmers
  - Not true, there are third party wrappers for Java, Python, and more

Different Types of CUDA Applications



CUDA - Uses

- CUDA provided benefit for many applications. Here list of some:
  - Seismic Database - 66x to 100x speedup http://www.headwave.com.
  - Molecular Dynamics - 21x to 100x speedup http://www.ks.uiuc.edu/Research/vmd
  - MRI processing - 245x to 415x speedup http://bic-test.beckman.uiuc.edu
  - Atmospheric Cloud Simulation - 50x speedup http://www.cs.clemson.edu/~jesteel/clouds.html.