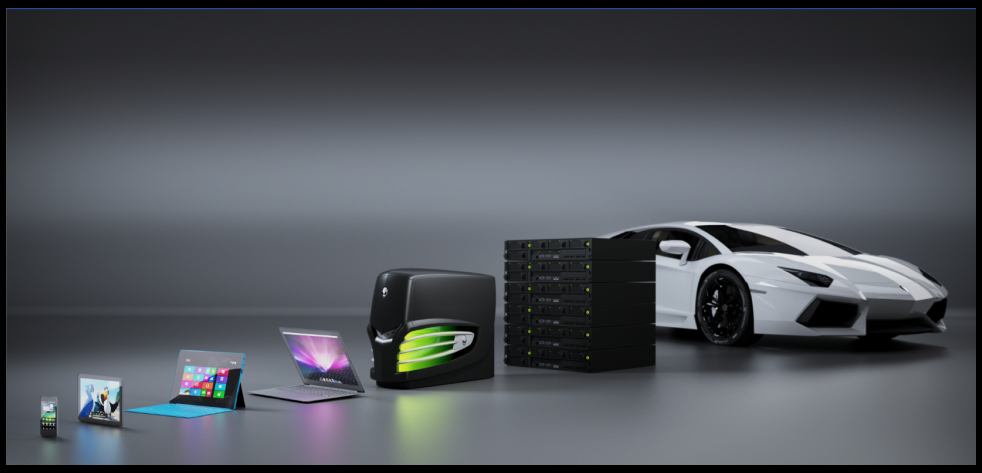
GPU Computing is everywhere!

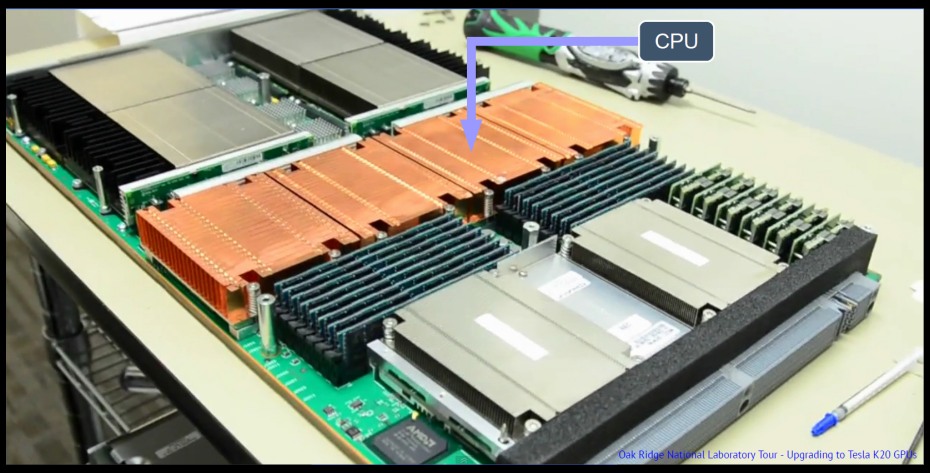


http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1

GPU in a HPC system

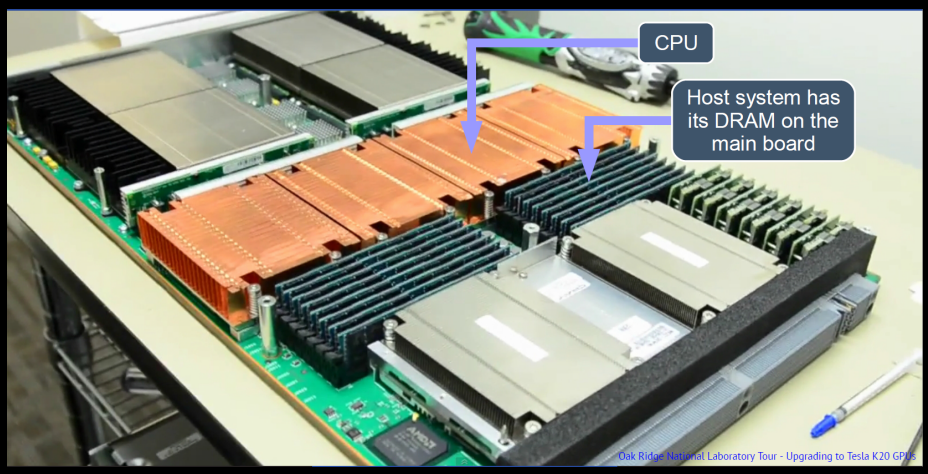
http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1

GPU in a HPC system



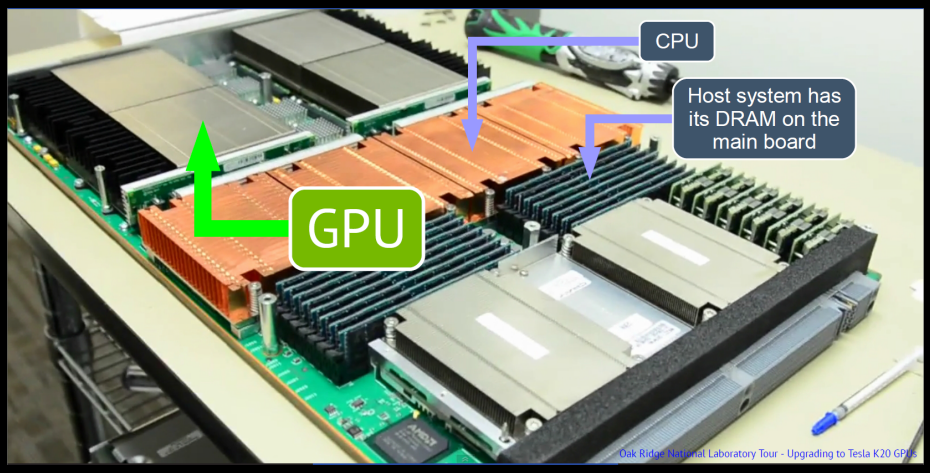
http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1

GPU in a HPC system



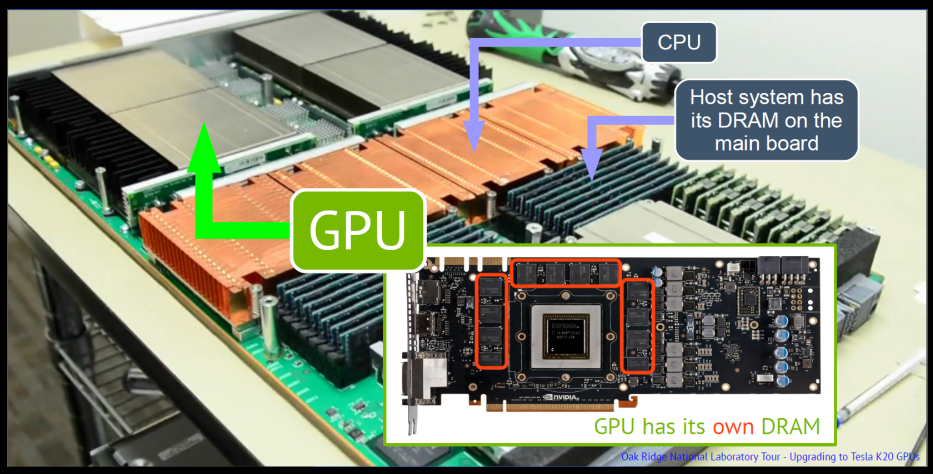
http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1

GPU in a HPC system



http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1

GPU in a HPC system

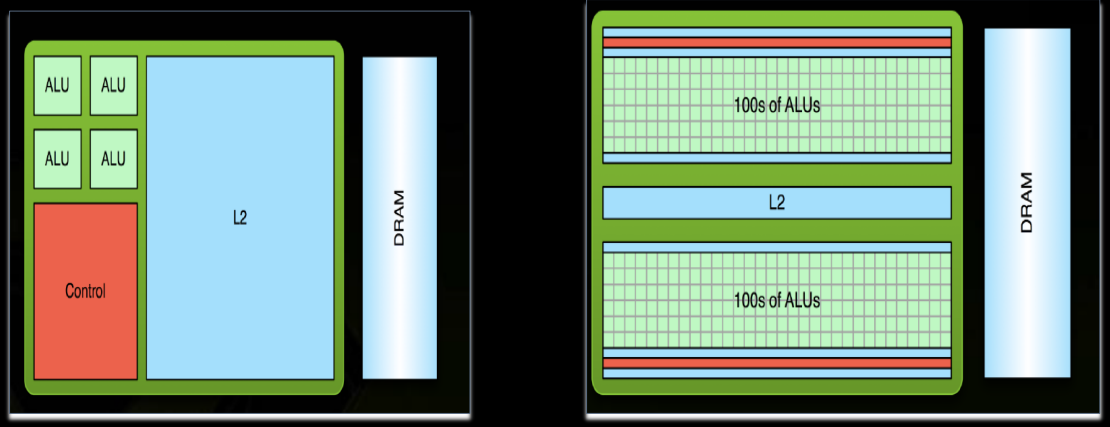


http://www.hpcadvisorycouncil.com/events/2014/swiss-workshop/presos/Day_1

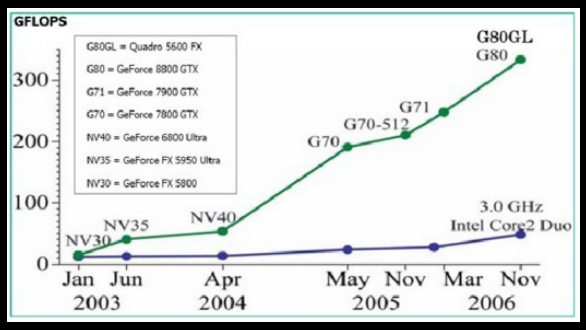
Why do we need a GPU????

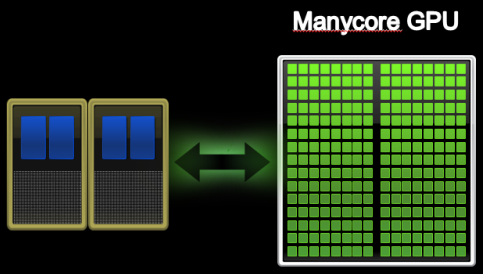
Difference between a CPU and GPU

- Both cater to different needs → **Low Latency or High Throughput?**



Difference between Multicore CPU and Many-core GPU





http://image.slidesharecdn.com/2009-07-22cuda-techtalk-090723122825-phpapp01/95/tech-talk-nvidia-cuda-7-728.jpg

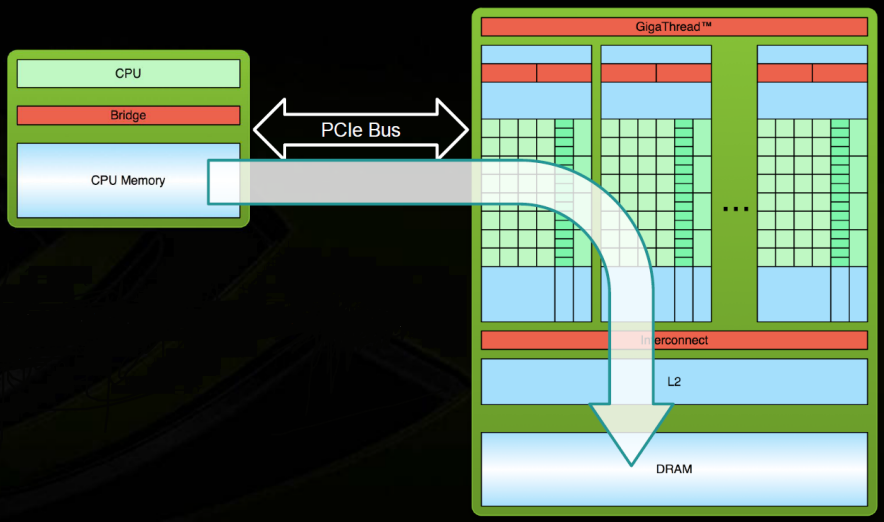
HPC Machines in CS Labs

Name	GPUs
bentley	**GF119 [NVS 315] and GM204 [GeForce GTX 980]**
bugatti	**GF119 [NVS 315] and GM204 [GeForce GTX 980]**
ferrari	**GF119 [NVS 315] and GM204 [GeForce GTX 980]**
jaguar	**GF119 [NVS 315] and GM204 [GeForce GTX 980]**
lamborghini	**GF119 [NVS 315] and GM200 [GeForce GTX TITAN X]**
lotus	**GF119 [NVS 315] and GM200 [GeForce GTX TITAN X]**
maserati	**GF119 [NVS 315] and GM200 [GeForce GTX TITAN X]**
porsche	**GF119 [NVS 315] and GM200 [GeForce GTX TITAN X]**
raspberries	**G96GL [Quadro FX 580]**

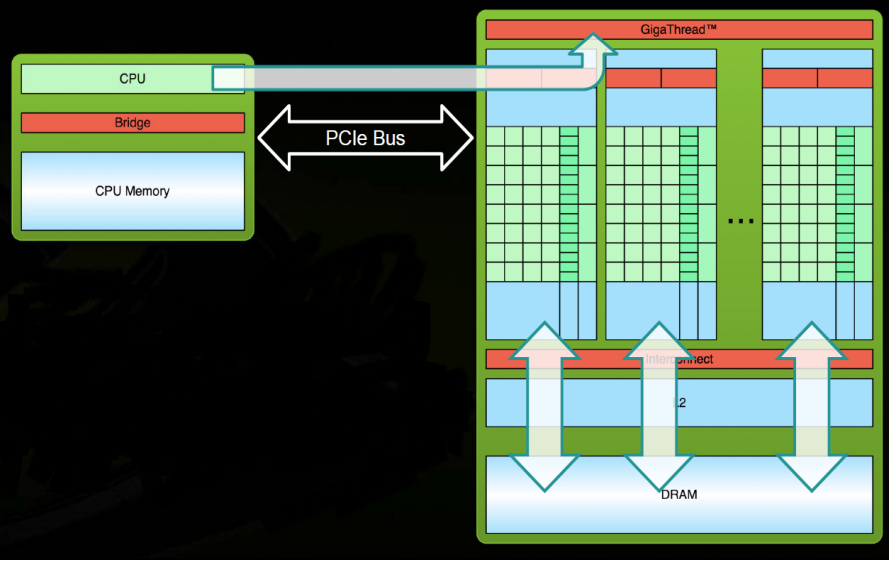
Features of the Graphics card **GeForce GTX 980**

GPU Architecture	Maxwel
GPU Name	GM204
CUDA Cores	2048
Clock Speed	1126 MHz
VRAM	4 GB GDDR5
Memory Bus	256-bit
Memory Clock	7.0 GHz
Memory Bandwidth	224.0 GB/s
Power Connectors	Two 6-Pin

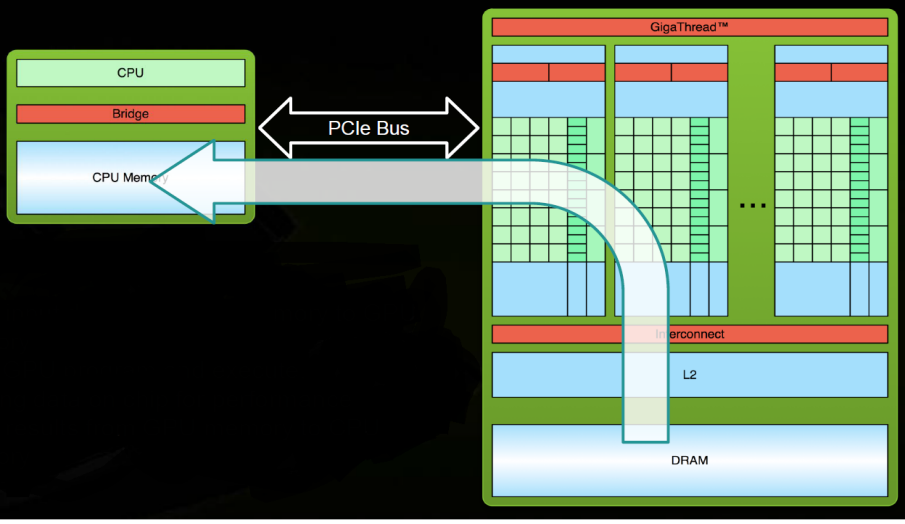
Processing Flow:



Processing Flow(contd.):



Processing Flow(contd.):

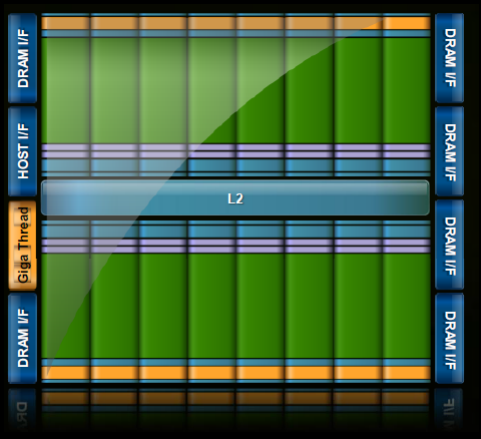


GPU Architecture

GPU Architecture : Two Main Components

1. Global Memory:
-Accessible by both GPU and CPU
-Analogous to RAM in a CPU server

2. **Streaming Multiprocessors (SMs):**
-Perform the actual computations
Each SM has its own:
Control units, registers, execution pipelines, caches



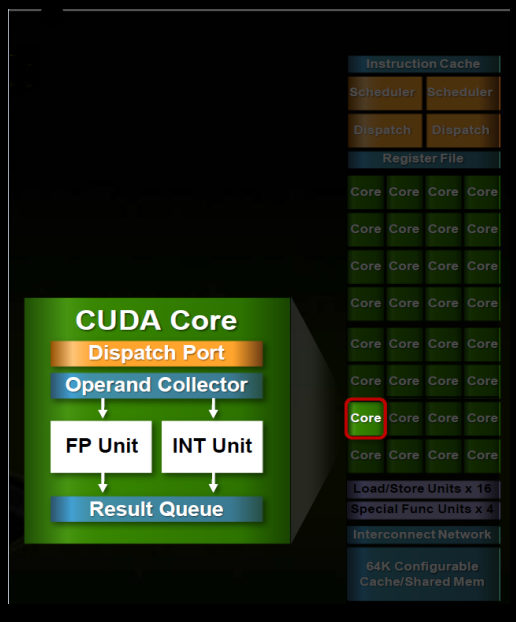
GPU Architecture –Fermi: Streaming Multiprocessor

- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem+ L1 cache
- 32K 32-bit registers
- Register File

GPU Architecture –Fermi : CUDA Core Floating

Core

- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit



Memory System-- Architecture

Shared memory(L1)

- User-managed scratch-pad
- Hardware will not evict until threads overwrite
- 16 or 48KB / SM (64KB total is split between Shared and L1)
- Aggregate bandwidth per GPU: 1.03 TB/s

ECC Protection can be enabled.

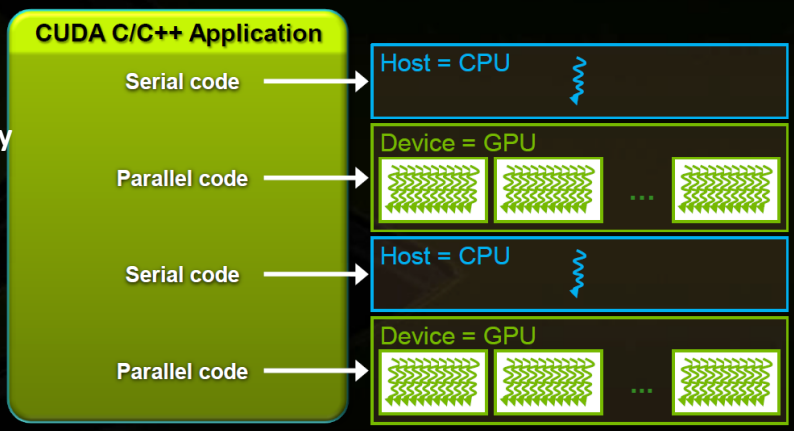
CUDA Programming Abstractions

Shivani Dave

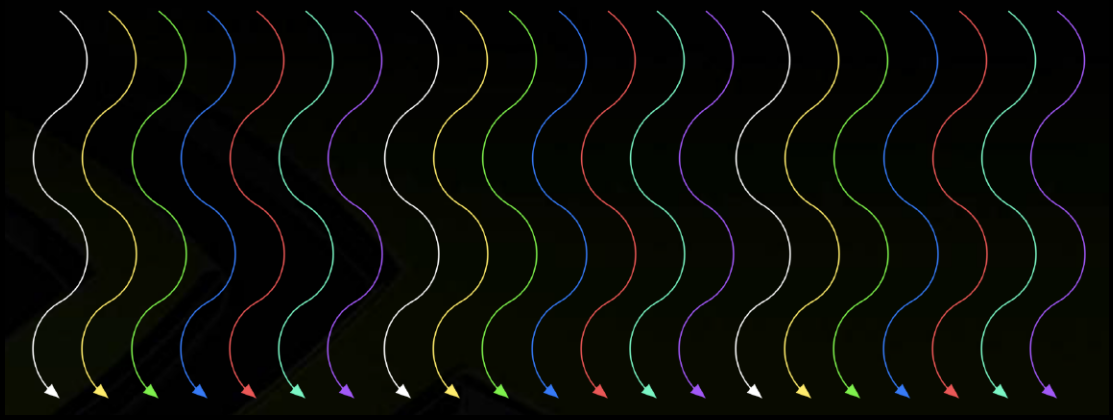
CUDA Application structure:

Serial code executes in a host (CPU) thread

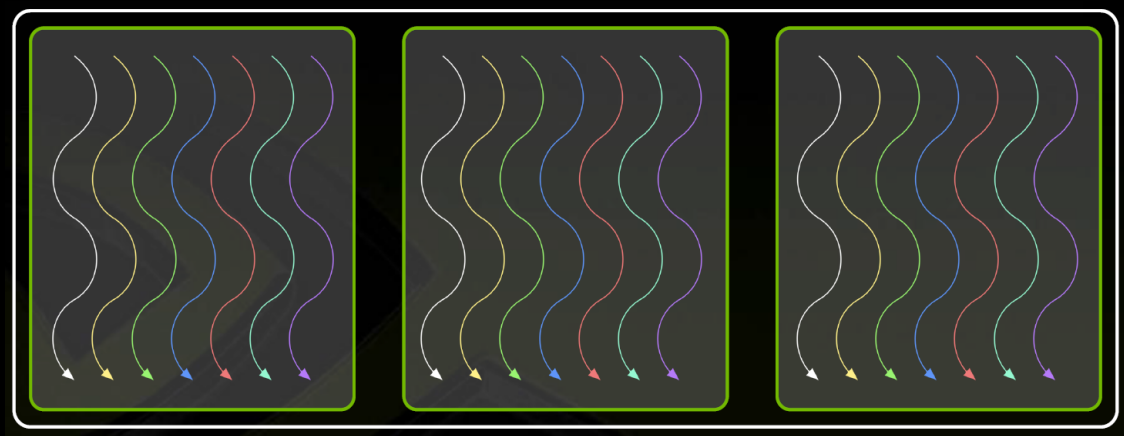
Parallel code executes in many device (GPU) threads across multiple processing elements



Threads

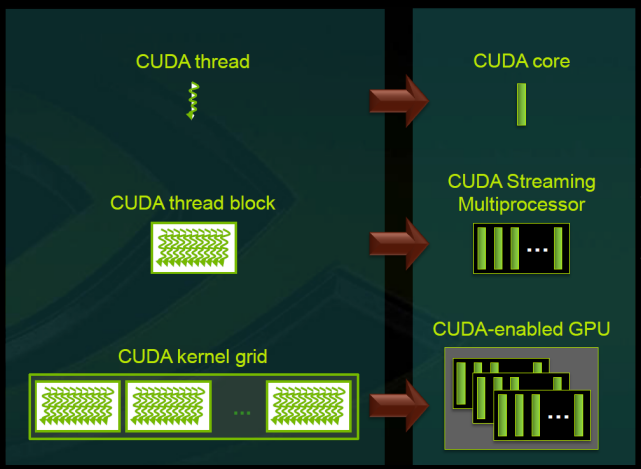


Threads are grouped into blocks



- A kernel is executed as a grid of blocks of threads

Kernel Execution



Each thread is executed by a core CUDA

--Each block is executed by one SM and does not migrate
--Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

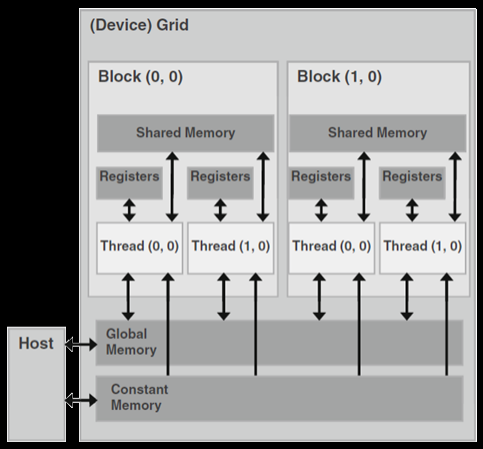
--Each kernel is executed on one device
--Multiple kernels can execute on a device at one time in an asynchronous way

Memory Model

Device Code:
R/W per-thread registers
R/W per-thread local memory
R/W per-block shared memory
R/W per-grid global memory
Read only per-grid constant

Host Code:
Transfer data to/from per-grid
Global and Constant memories

Threads	Registers and Local Memory
Block of Threads	Shared Memory
All the blocks	Global Memory



Memory Model (contd …)

- Global memory is the slowest memory on the GPU
- Coalescing improves memory performance; it occurs when multiple (row major order) consecutive threads (IDs) read / write consecutive data items from / to global memory
- 16 (half a warp) global array elements are accessed at once: coalescing produces vectorized reads / writes that are much faster than element wise reads / writes.
- A warp in CUDA, is a group of 32 threads, which is the minimum size of the data processed in SIMD fashion by a CUDA multiprocessor.

Sequential C code -> Vector Addition

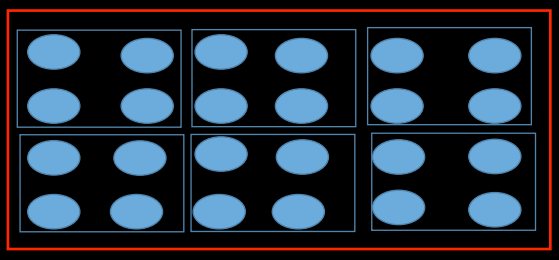
```
#include<stdio.h>
void Addition_Vector(int *a, int *b, int *c)
{
        int i=0;
        for(i=0;i<25;i++)
        c[i]=a[i]+b[i];
}

int main()
{
        int a[25],b[25],c[25],i;
        for(i=0;i<25;i++)
        {
                a[i]=i;
                b[i]=i+1;
        }

        Addition_Vector(a,b,c);
        for(i=0;i<25;i++)
        printf("\n%d",c[i]);
}
```

Parallelism and Threads

- To process operations in parallel, the operations must be independent of each other i.e., no data dependencies
- A thread is mapped to a single processor which executes in parallel with the remaining threads.
- A CUDA kernel is executed by a grid (array) of threads with each thread with a unique index id in a specific block.
- A grid is organized as a 2D array of blocks (gridDim.x and gridDim.y)
- Each block is organized as 3D array of threads (blockDim.x, blockDim.y, and blockDim.z)



gridDim = (3,2)
blockDim = (2,2,1)

Thread Allocation

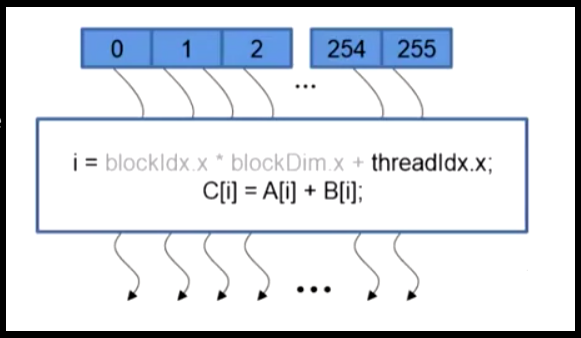
- A thread block can be allocated on any stream multiprocessor and thread blocks must be independent of each other, i.e., cannot communicate with each other at all.
  - pro: now the computation can run on any number of SMs
  - con: this makes programming a GPU harder
- Multiple thread blocks can be scheduled on one multiprocessor, if resources allow it. They still are independent of each other.

Thread Synchronization

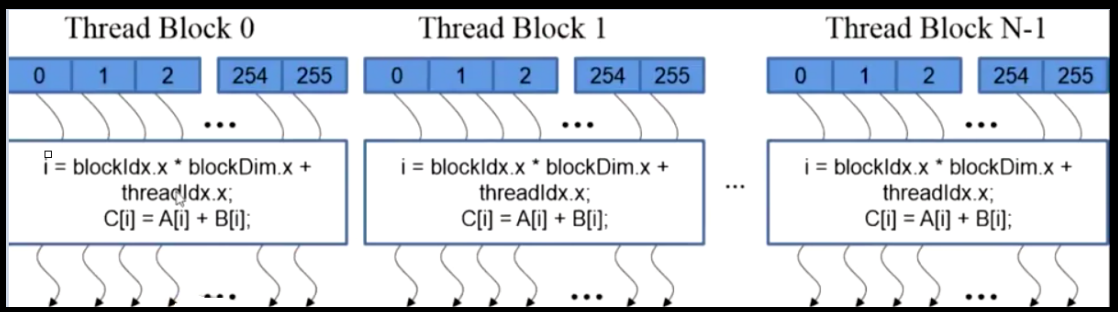
- Threads inside one thread block can synchronize – _syncthreads() command
- host can synchronize kernel calls – either explicitly through cudaThreadSynchronize() – or implicitly through memcpy()-s

Parameterizing the thread code

- The programmer's job is to write the code so that when a collection of threadblocks, each with a collection of threads executes in concert, the collective work done solves the problem.
- But there is only one piece of code (what it does is a function of the grid and thread "coordinates."
  - GridIdx.x, GridIdx.y
  - threadIdx.x, threadIdx.y, threadIdx.z
- In our first example, there is just one block which has 256 threads in it: all but threadIdx.x are zero, and unused.
- So, we will have all values from 0 to 255
- Analogy with OpenMP/MPI:
  - Dim is line num_threads(), and MPI_Size
  - Ids is like thread_num() and MPI_rank



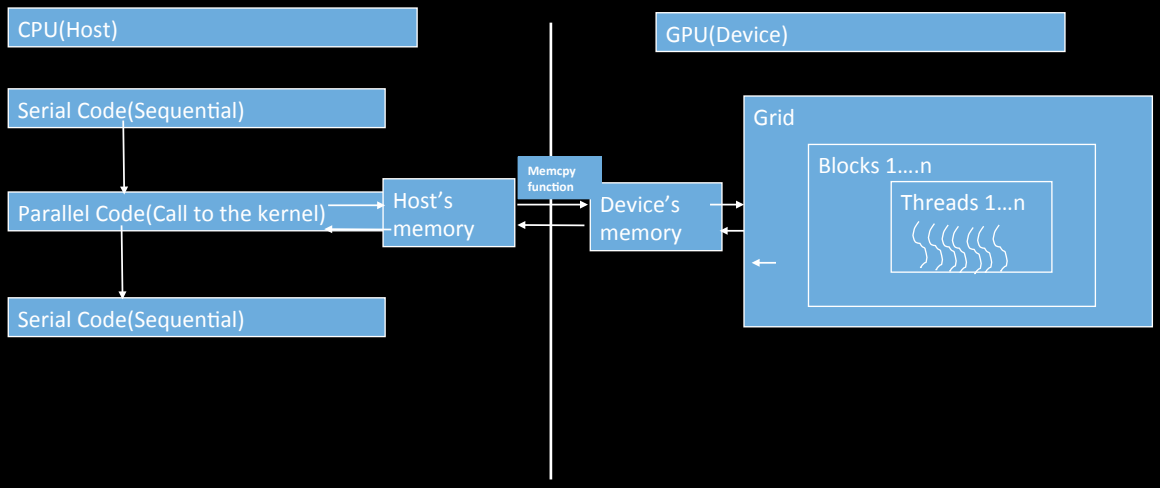
Multiple Blocks



Multiple block in the Grid.
Each block will have an equal number of threads as defined in the program.
In the above example: blockIdx.x keeps on increasing from 0 to 255 similar to the threadId.x . The variable blockDim.x will be 256 which is equal to the number of blocks.

©https://class.coursera.org/hetero-004/lecture/105

Program Flow



Vector Addition Map

1:1 Mapping

Iteration Space

Range of variable I
0 to 255
0……..255

Data Space

Vectors A, B and C
Function used f(i)
…255

Thread Space

i will be mapped as a
f(ThreadId)
…..255

There exists a 1:1 mapping between the iterator variable and the ThrThere exists only one block each being mapped to a set of 255 threadseadId.
.

Parallel Code→Vector Addition

```
#include<cuda.h>
void main(){
int n=256; int b[n]; int a[n],c[n]; //initialize the arrays too
int *dev_b; int *dev_a, int *dev_c;
cudaMalloc((void**)&dev_a, n*sizeof(int));
cudaMalloc((void**)&dev_b, n*sizeof(int));
cudaMalloc((void**)&dev_c, n*sizeof(int));
cudaMemcpy(dev_a, &a, n*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, &b, n*sizeof(int), cudaMemcpyHostToDevice);
dim3 gridDim = 1;//Number of blocks in the grid
dim3 blockDim =n;//Number of threads in each block
Addition_Vector<<<gridDim,blockDim>>>(dev_a,dev_b,dev_c);// Call to the device
cudaMemcpy(&c, dev_c, n*sizeof(int), cudaMemcpyDeviceToHost);
}
```

Kernel definition

```
__global__ void AddIntegers(int *a, int *b,int *c)
{

  c[threadIdx.x]=a[threadIdx.x]+b[threadIdx.x];
}
```

__global__ lets the compiler know that it is a kernel function and accordingly appropriate actions will be taken

Call to the kernel from the host is made using the below format:
<<<number of blocks, number of threads>>>

Built-in Variables

- In the kernel a set of built-in variables specifies the grid and block dimensions (Dim) and indices (Idx).
- These can be used to determine the thread ID
  - gridDim contains .x and .y grid dimensions (sizes)
  - blockIdx contains block indices .x and .y in the grid
  - blockDim contains the thread block .x, .y, .z dimensions (sizes) – threadIdx contains .x, .y and .z thread block indices
- 1D thread block: ID = threadIdx.x
- 2D thread block: ID = threadIdx.x + threadIdx.y*blockDim.x
- 3D thread block: ID = threadIdx.x + threadIdx.y*blockDim.x + threadIdx.z*blockDim.x*blockDim.y

API functions in CUDA

- Device Memory Allocation function
- Host-Device Data /transfer

Device Memory Allocation function

cudaMalloc()

- Similar to Malloc function in C
- It will allocate memory in the GPU's global memory.
- Two parameters

1. Address of a pointer to the allocated object
2. Size of allocated object in terms of bytes

cudaFree()

- Frees object from device global memory
- Parameter - Pointer to freed object

Host-Device Data /transfer

cudaMemcpy()

--Used for memory data transfer

Parameters:

1. Pointer to destination
2. Pointer to source
3. Number of bytes copied
4. Type of transfer

There are 4 types of data transfer:

-Host to Host

-Host to Device

-Device to Host

-Device to Device

All these data transfers are asynchronous.

Programming Examples

2D Grid Block

- Cuda allows us to create 1-D,2-D,3-D grid blocks
- 2-D Kernel Launch:
- Image processing tasks typically impose a regular 2D raster (an image) over the problem domain. Computational Fluid dynamics tasks might be most naturally expressed by partitioning a volume over a 3D grid.

2-D Representation Program

- 2-D mapping

```
__global__ void kernel(int *array)
{
  int index_x = blockIdx.x * blockDim.x + threadIdx.x;
  int index_y = blockIdx.y * blockDim.y + threadIdx.y;

  // map the two 2D indices to a single linear, 1D index
  int grid_width = gridDim.x * blockDim.x;
  int index = index_x * grid_width + index_y;

  // map the two 2D block indices to a single linear, 1D block index
  int result = blockIdx.x * gridDim.x + blockIdx.y;

  // write out the result
  array[index] = result;
}
```

Vector Addition Map

2:1 Mapping

Iteration Space

Range of variable i
Range of variable j
0….255
0….255

Data Space

Vectors A(2-D array), C
Function used f(I,j)

Thread Space

i and j will be mapped as a f(,BlockIdThreadId, BlockId)

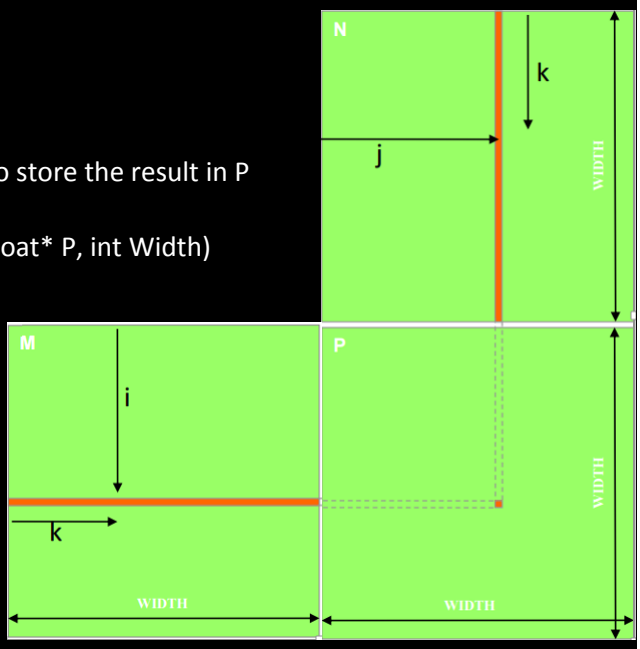

…..16


…16

There exists a 2:1 mapping between the iterator variable i, j and the ThreadId for every BlockId
There exists 16 blocks each being mapped to a set of 2 values in x and y direction

Matrix Multiplication

- Consider two Matrices M and N. We need to store the result in P
- Executing on the host:
- void MatrixMulOnHost(float* M, float* N, float* P, int Width)
- {
- for (int i = 0; i < Width; ++i)
- for (int j = 0; j < Width; ++j) {
  double sum = 0;
  for (int k = 0; k < Width; ++k) {
  double a = M[i * Width + k];
  double b = N[k * Width + j];
  sum += a * b;}
  P[i * Width + j] = sum;}
- }



Cuda Code:

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;

1. // Transfer M and N to device memory
  cudaMalloc((void**) &Md, size);
  cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
  cudaMalloc((void**) &Nd, size);
  cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

  // Allocate P on the device
  cudaMalloc((void**) &Pd, size);

2. // Kernel invocation code - to be shown later
  ...
3. // Transfer P from device to host
  cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
  // Free device matrices
  cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
  }
```

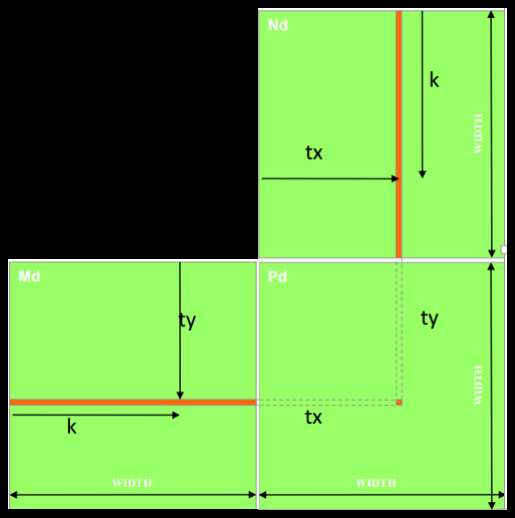
Kernel Code

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int W
{
  // 2D Thread ID
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  // Pvalue stores the Pd element that is computed by the thread
  float Pvalue = 0;

  for (int k = 0; k < Width; ++k)
  {
     float Mdelement = Md[ty * Width + k];
     float Ndelement = Nd[k * Width + tx];
     Pvalue += Mdelement * Ndelement;
  }

  // Write the matrix to device memory each thread writes one element
  Pd[ty * Width + tx] = Pvalue;
}
```



Programming Example-Jacobi1D.cu

References:

- http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/02-cuda-overview.pdf
- https://class.coursera.org
- Programming Massively Parallel Processors - David B. Kirk and Wen-mei W. Hwu
- http://docs.nvidia.com/cuda/parallel-thread-execution/#axzz3p0rtlrXV
- https://code.google.com/p/stanford-cs193g-sp2010/wiki/TutorialMultidimensionalKernelLaunch