**Department of Information Technology**
National Institute of Technology Karnataka, Surathkal

# IT301: INTRODUCTION TO CUDA

**By,**
**Ms. Thanmayee**
**Adhoc Faculty,**
**Department of IT,**
**NITK, Surathkal**

# OUTLINE

- Introduction to GPU
- Evolution of GPU microarchitectures
- General Purpose GPU
- Introduction to CUDA
- CUDA Execution Model
- CUDA Memory Model
- Steps in GPU Execution
- Hello World Program
- CUDA Device Variables
- CUDA Programming examples

# Let's learn about GPU..

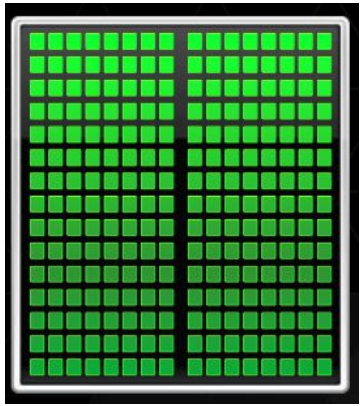
- A little history:
  - The first GPUs were designed as graphics accelerators
    - supported only specific fixed-function pipelines.
  - In the late 1990s, the hardware became increasingly programmable
    - Culminating in NVIDIA's first GPU in 1999.


START


02:37.68
record
00:17.29
time
Swift 18
Nitro 12
Hakkinen
Deuce -132
Burnout -168
Gearhead -180
position 3/6
0%
100%
damage
gear 3

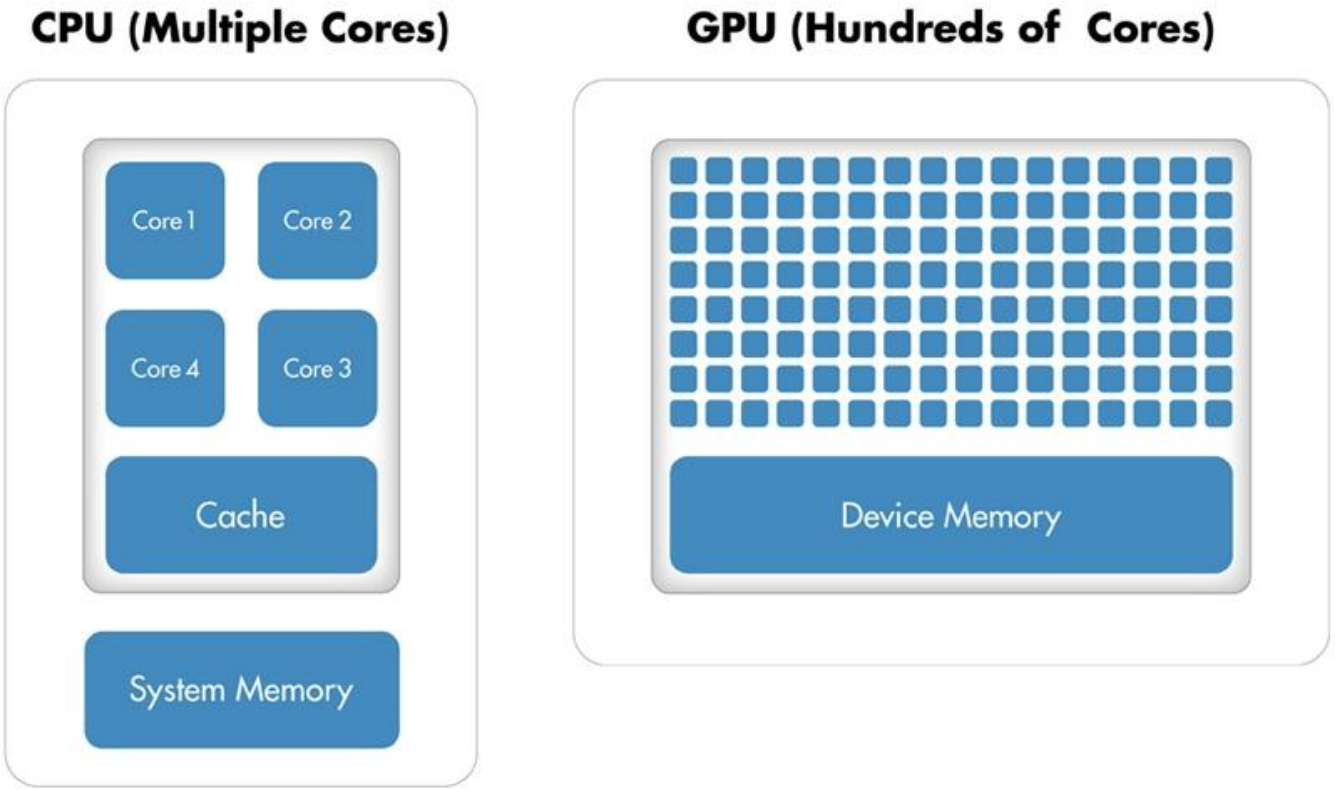

0% PROGRESS
POSITION 8/12
CHECKPOINT
258 M
JusticeCovert
Wolves (feat. Youngblood Hawke) [RAC Remix]
Digitalism
053

# Graphics Processing Unit

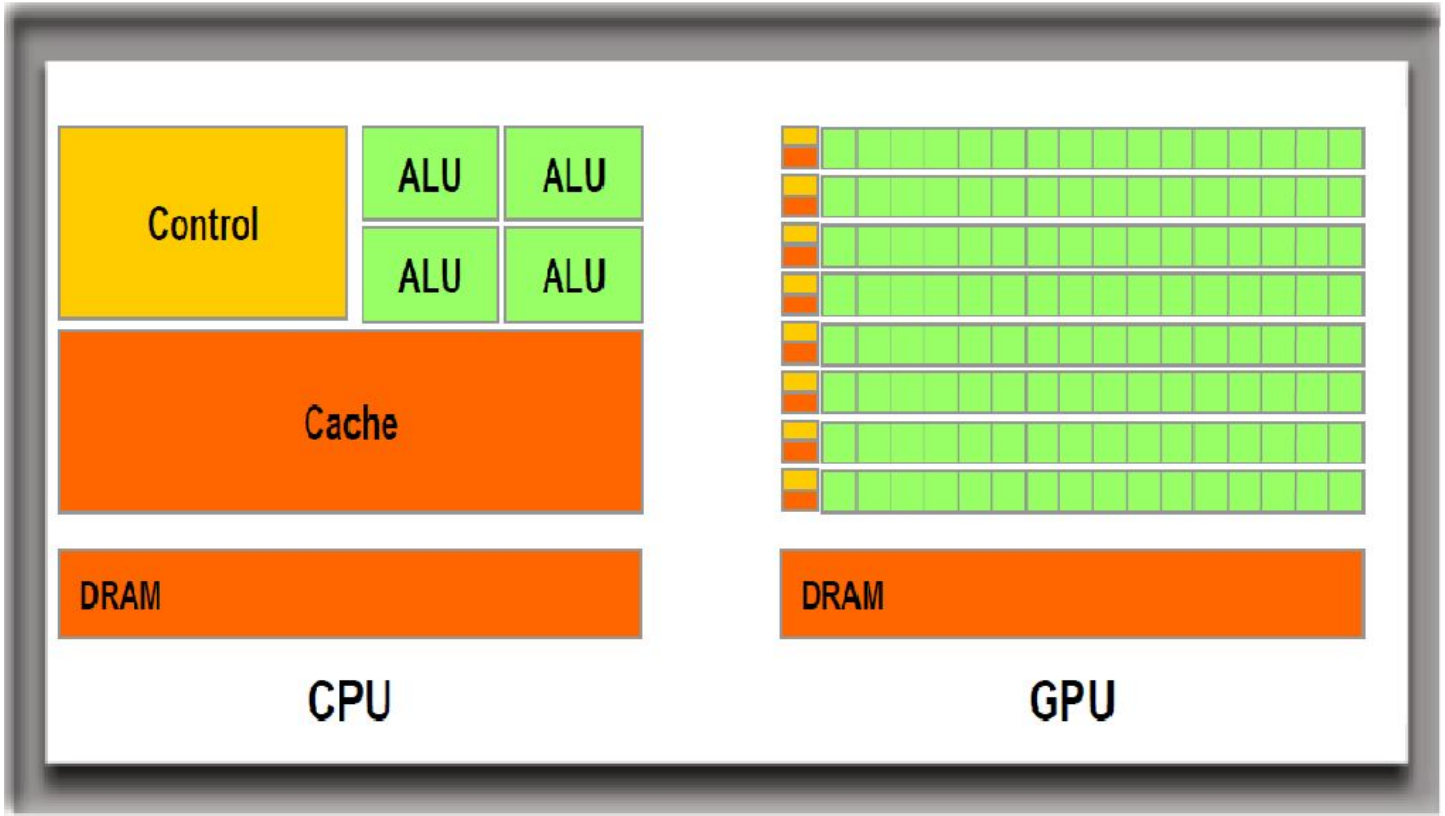


- Has thousands of cores and ALUs.
- They can handle billions of repetitive low level tasks.
- GPU is specialized for compute-intensive, highly parallel computation.
- They are devoted to data processing rather than data caching and flow control.
- Follows SPMD processing model.

# CPU versus GPU



# More closer look at GPU:

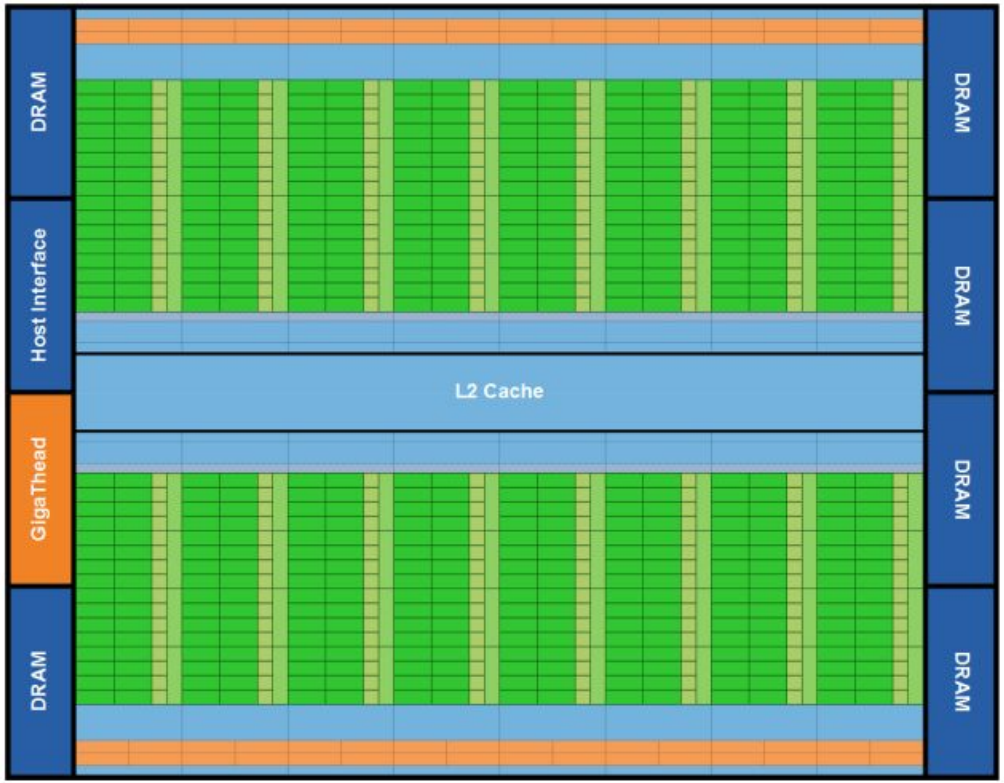


# GPU Terms:

- **Stream processing** -- Term used to denote processing of a stream of instructions operating in a data parallel fashion.
- **Stream Processors (SPs)** – the execution cores that will execute the stream. Each stream processor has compute resources such as register file, instruction scheduler
- **Streaming multiprocessors (SMs)** -- groups of streaming processors that shares control logic and cache.

# GPU Microarchitectures

## Fermi Architecture (2010)

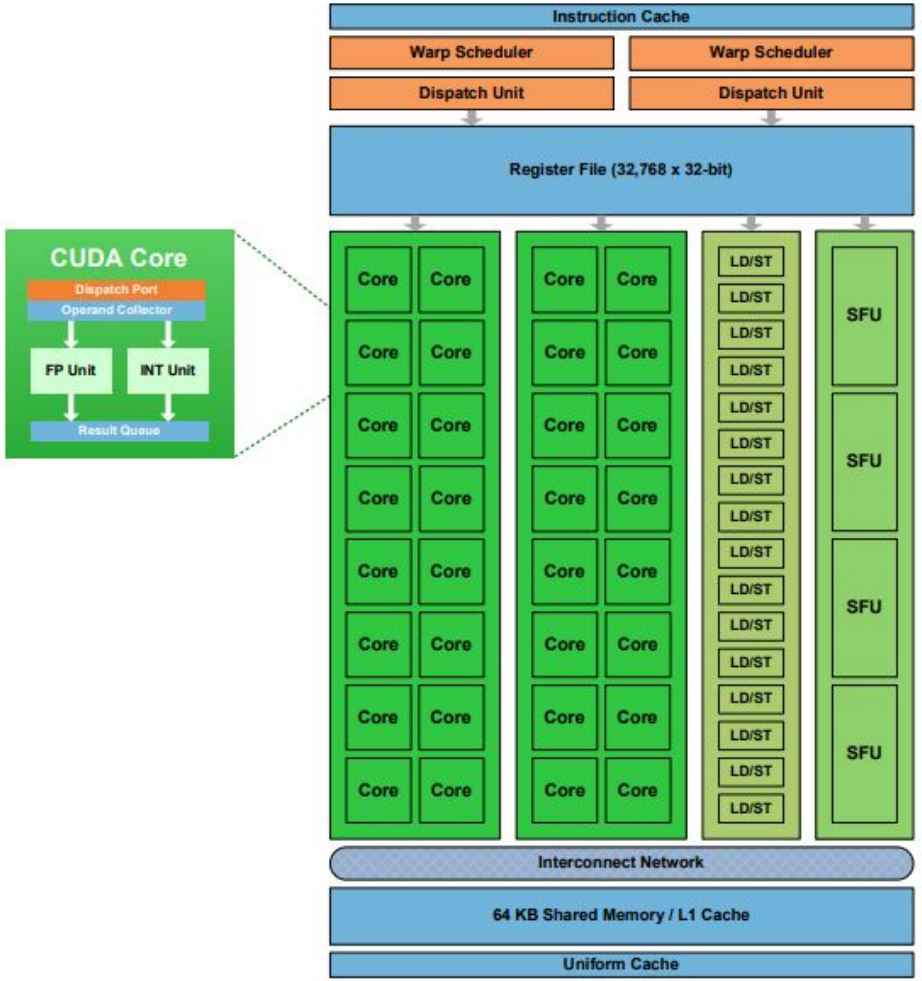



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

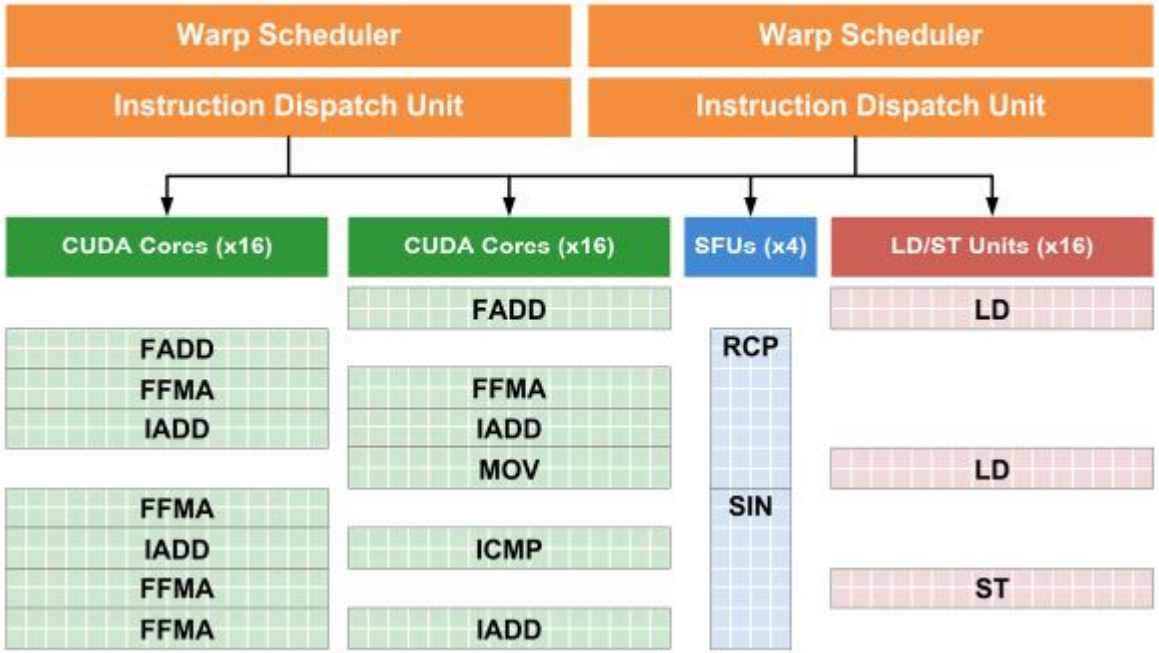
https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

# What each SMs have..

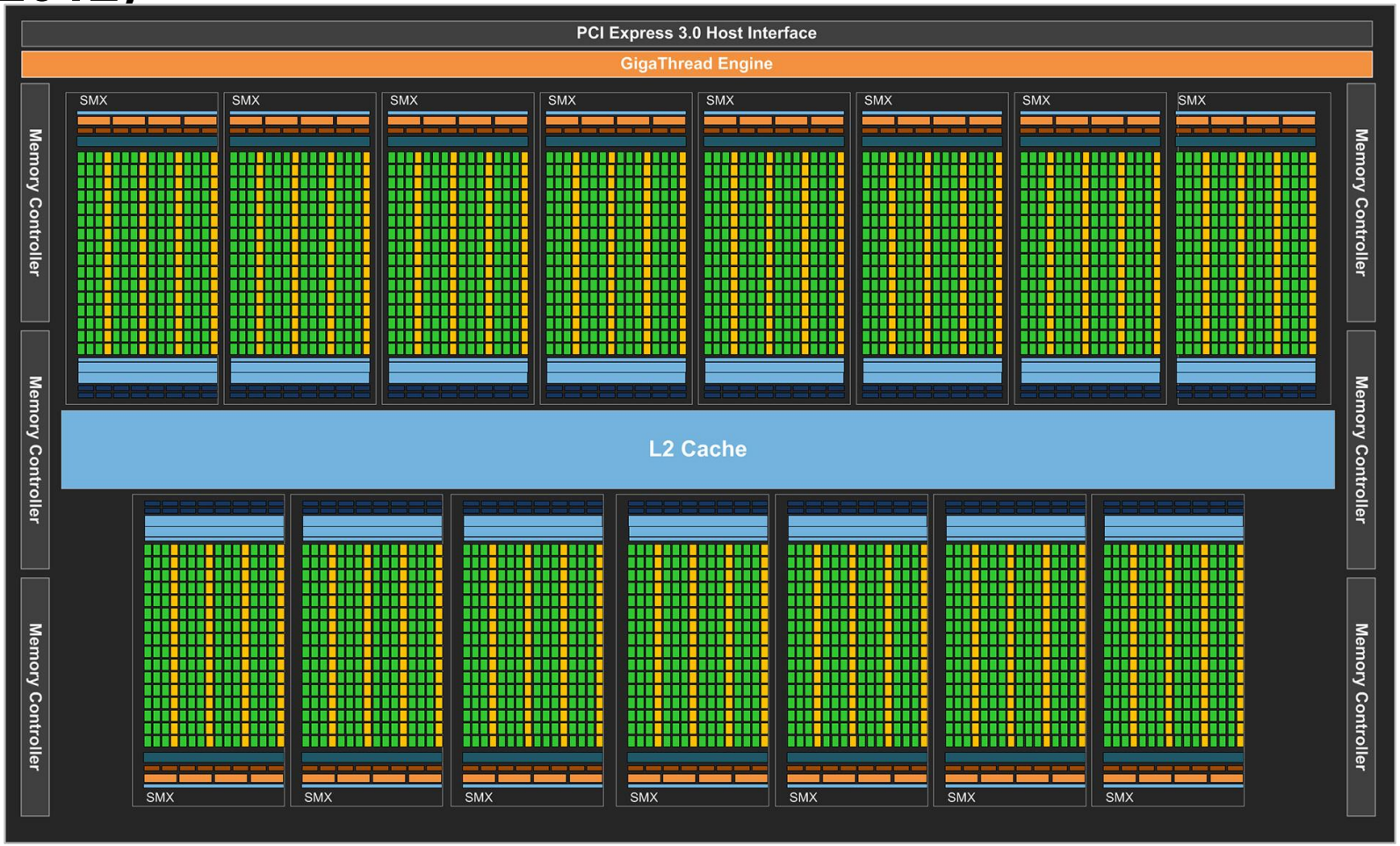
- Memory operations are handled by a set of 16 load-store units ir each SM.
- A set of four Special Function Units (SFUs) is also available to handle transcendental and other special operations such as sin, cos, exp, and rcp (reciprocal)
- Along with the group of 16 load-store units and the four SFUs, there are four execution blocks per SM.
    - 16 + 16 Cores (2 blocks)
    - LD/ST
    - SFU



**A total of 32 instructions from one or two warps can be dispatched in each cycle to any two of the four execution blocks within a Fermi SM**



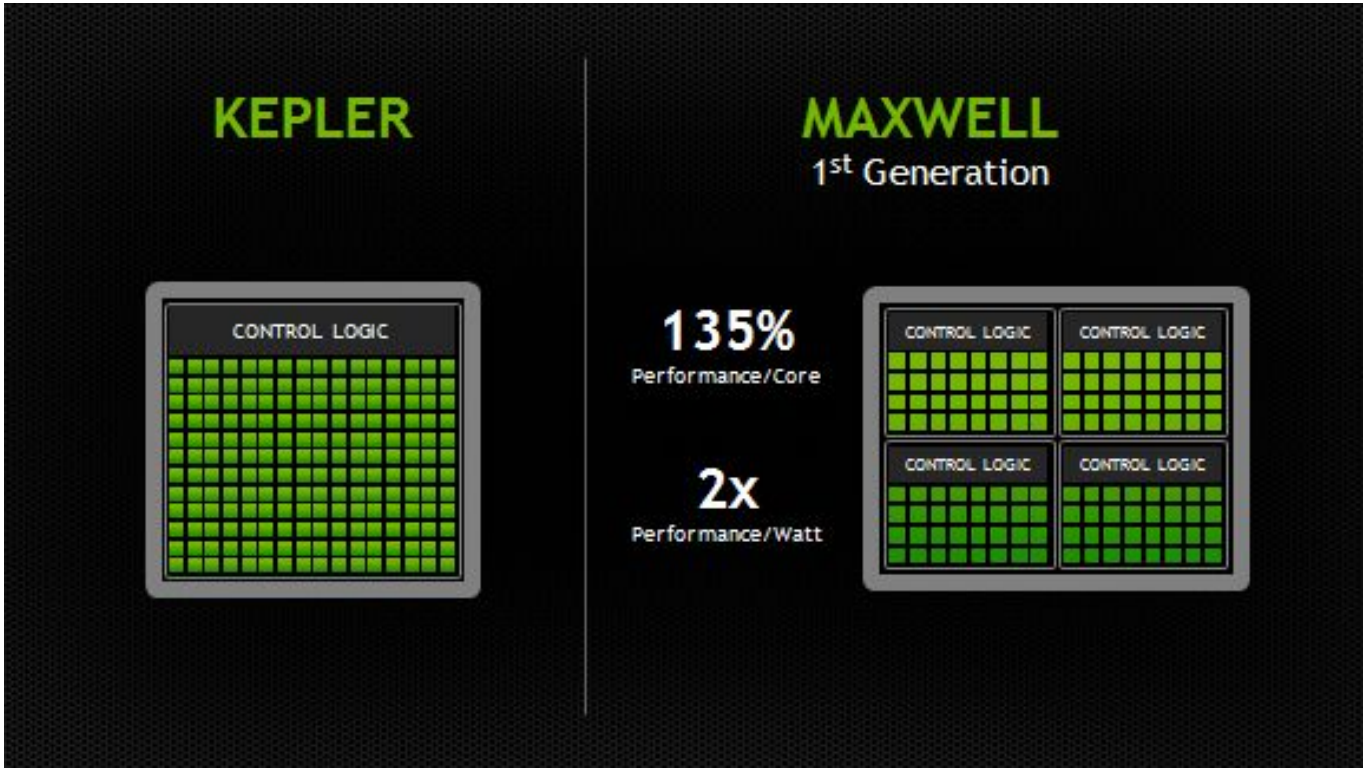
# Kepler (2012)

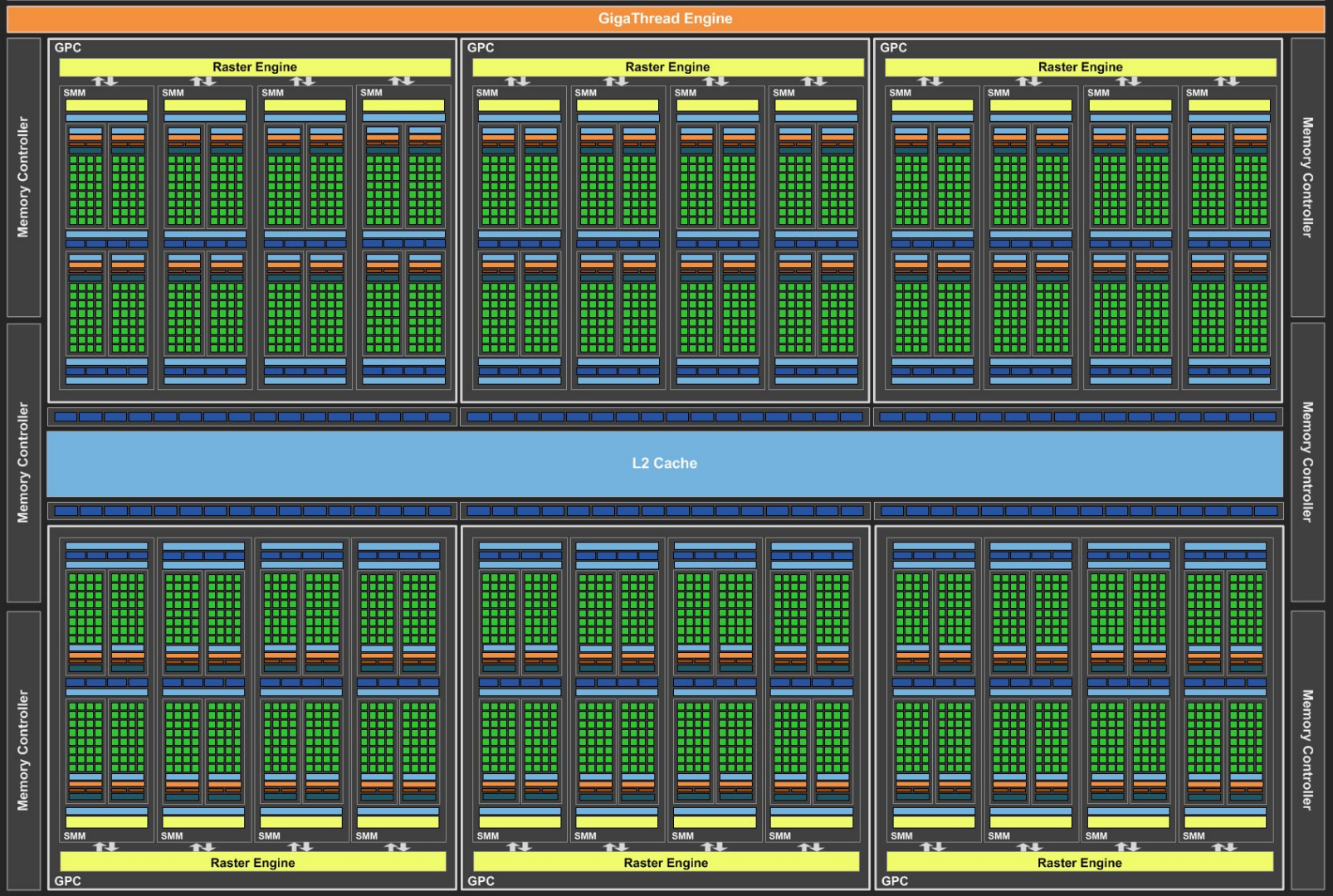

PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
SMX
SMX
SMX
SMX
SMX
SMX
SMX
SMX
L2 Cache
SMX
SMX
SMX
SMX
SMX
SMX
SMX

# Fermi Versus Kepler

| Feature | Fermi (GTX 580) | Kepler (GTX 680) |
|---|---|---|
| Fabrication | 40 nm | 28 nm |
| Power Consumption | 244 Watts | 195 Watts |
| CUDA Cores | 512 | 1536 |
| Compute Capability | 2.1 | 2.1 |
| Streaming Multiprocessors (SM) | 16 | 8 |
| Cores per SM | 32 | 192 |
| Texture Units | 64 | 128 |
| Warp Schedulers per SM | 2 | 4 |
| L1 Cache per SM | 64 KB | 64 KB |
| L2 Cache | 768 KB | 512 KB |
| Memory Interface | 384 bits | 256 bits |
| Bus Type | PCIe 2.0 x16 | PCIe 3.0 x16 |
| Memory Type | GDDR5 | GDDR5 |
| Memory Bandiwdth | 192.2 GB/s | 192.2 GB/s |
| Core Speed | 772 MHz | 1002 MHz |

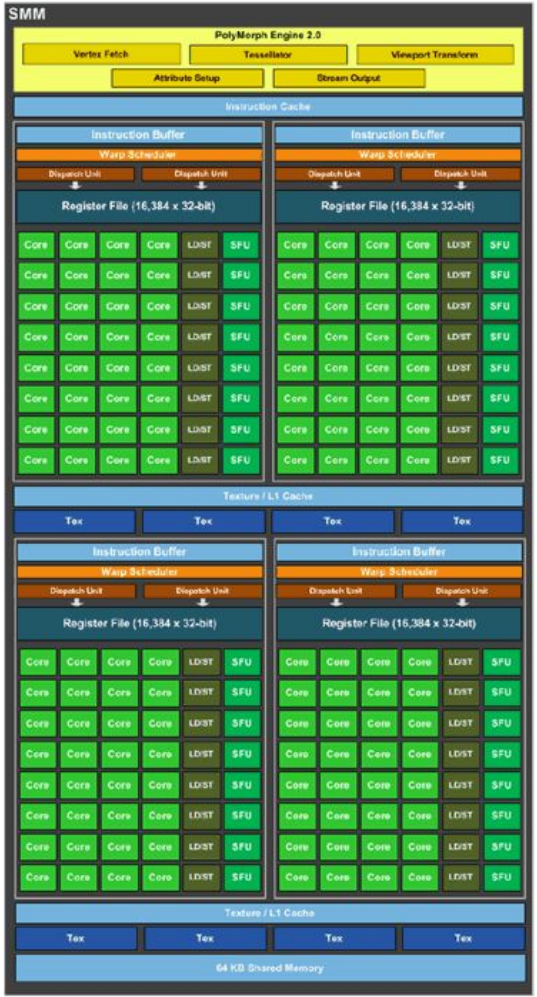
# Maxwell (2014)




GigaThread Engine
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
GPC
Raster Engine
SMM
L2 Cache
GPC
Raster Engine
GPC
Raster Engine
GPC
Raster Engine
GPC
Raster Engine
GPC
Raster Engine

## NVIDIA Kepler SM


## NVIDIA Maxwell SM

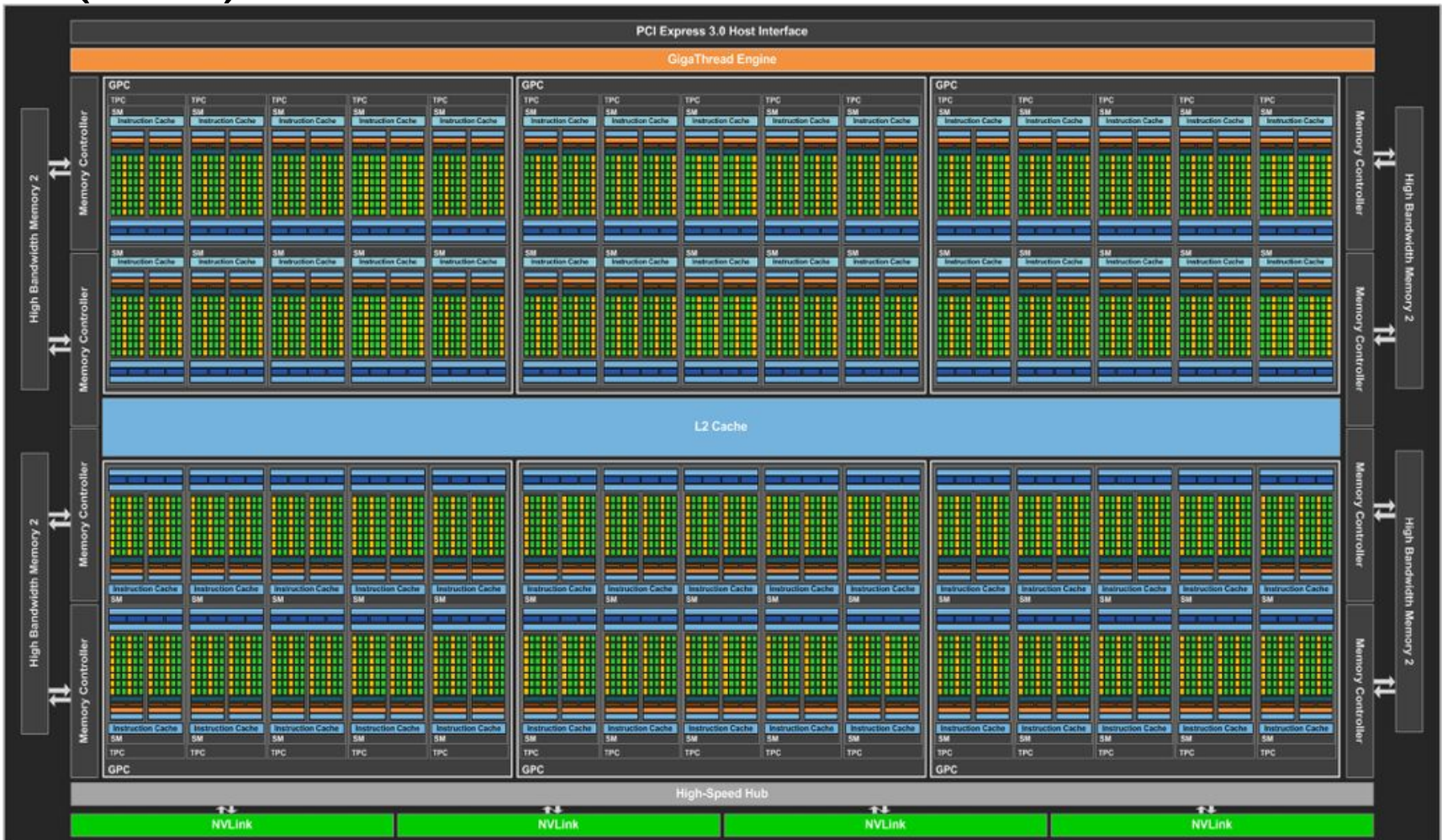


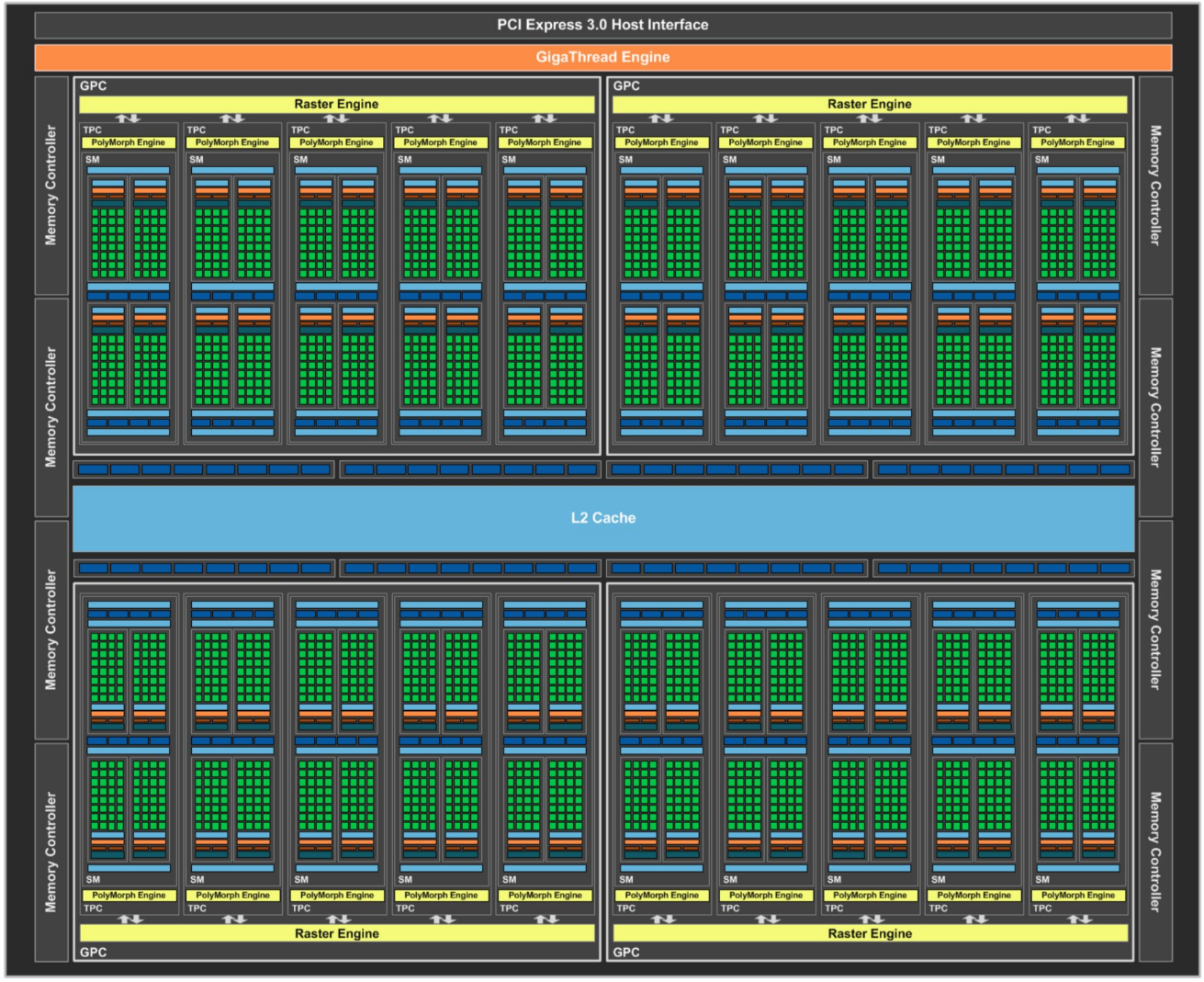
# MAXWELL "GM204"

## Top Level

- 5.2 Billion Transistors
- 2x performance vs GK104
- 16 SMM
- 2048 CUDA Cores
- 16 Geometry Units
- 128 Texture Units
- 64 ROP Units
- 256-bit GDDR5

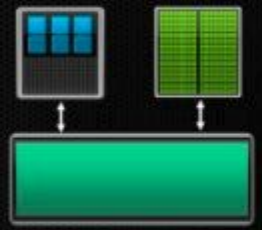
# Pascal (2016)


PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub
NVLink


PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
GPC
Raster Engine
TPC
PolyMorph Engine
SM
L2 Cache
GPC
Raster Engine
GPC
Raster Engine
GPC
Raster Engine

# Pascal

## Unified Memory



Dramatically Lower Developer Effort

## Stacked Memory


4x Higher Bandwidth (~1 TB/s)
3x Larger Capacity
4x More Energy Efficient per bit

## NVLink

GPU high speed interconnect
80-200 GB/s
Supported in POWER & ARM64 CPUs

**Fermi** : GTX480
GTX580

**Kepler** : GTX 680
GTX 780
GTX 780 Ti

**Maxwell**: GTX 980
GTX 980 Ti

**Pascal :** GTX 1080

| GeForce | GTX 480 | GTX 580 | GTX 680 | GTX 780 | GTX 780 Ti | GTX 980 | GTX 980 Ti | GTX 1080 |
|---|---|---|---|---|---|---|---|---|
| Codename | GF100 | GF100 | GK104 | GK110 | GK110 | GM204 | GM200 | GP104 |
| Fab (nm) | 40 | 40 | 28 | 28 | 28 | 28 | 28 | 16 |
| Transistors (Billion) | 3 | 3 | 3.54 | 7.08 | 7.08 | 5.2 | 8.0 | 7.2 |
| Die size (mm2) | 529 | 520 | 294 | 561 | 561 | 398 | 601 | 314 |
| CUDA Cores | 480 | 512 | 1536 | 2304 | 2880 | 2048 | 2816 | 2560 |
| TAU | 60 | 64 | 128 | 192 | 240 | 128 | 176 | 160 |
| ROP | 48 | 48 | 32 | 48 | 48 | 64 | 96 | 64 |
| Memory (MB) | 1536 | 1536 | 2048 | 3072 | 3072 | 4096 | 6144 | 8192 |
| Bus width (bit) | 384 | 384 | 256 | 384 | 384 | 256 | 384 | 256 |
| Bandwidth (GB/s) | 177.4 | 192.3 | 192.2 | 288.4 | 336.4 | 224 | 336 | 320 |
| Release date | Mar-10 | Nov-10 | Mar-12 | May-13 | Nov-13 | Oct-14 | Jun-15 | Jun-16 |
| Price at release | $500 | $500 | $500 | $650 | $700 | $550 | $650 | $600 |

Source : https://www.techspot.com/article/1191-nvidia-geforce-six-generations-tested/

# Volta micro architecture (2017)


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory

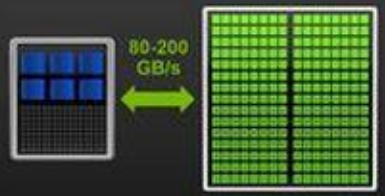
# NVIDIA Tesla GPUs

| Tesla Product | Tesla K40 | Tesla M40 | Tesla P100 | Tesla V100 |
|---|---|---|---|---|
| GPU | GK180 (Kepler) | GM200 (Maxwell) | GP100 (Pascal) | GV100 (Volta) |
| SMs | 15 | 24 | 56 | 80 |
| TPCs | 15 | 24 | 28 | 40 |
| FP32 Cores / SM | 192 | 128 | 64 | 64 |
| FP32 Cores / GPU | 2880 | 3072 | 3584 | 5120 |
| FP64 Cores / SM | 64 | 4 | 32 | 32 |
| FP64 Cores / GPU | 960 | 96 | 1792 | 2560 |
| Tensor Cores / SM | NA | NA | NA | 8 |
| Tensor Cores / GPU | NA | NA | NA | 640 |
| GPU Boost Clock | 810/875 MHz | 1114 MHz | 1480 MHz | 1530 MHz |
| Peak FP32 TFLOPS[1] | 5 | 6.8 | 10.6 | 15.7 |
| Peak FP64 TFLOPS[1] | 1.7 | .21 | 5.3 | 7.8 |
| Peak Tensor TFLOPS[1] | NA | NA | NA | 125 |
| Texture Units | 240 | 192 | 224 | 320 |
| Memory Interface | 384-bit GDDR5 | 384-bit GDDR5 | 4096-bit HBM2 | 4096-bit HBM2 |
| Memory Size | Up to 12 GB | Up to 24 GB | 16 GB | 16 GB |
| L2 Cache Size | 1536 KB | 3072 KB | 4096 KB | 6144 KB |
| Shared Memory Size / SM | 16 KB/32 KB/48 KB | 96 KB | 64 KB | Configurable up to 96 KB |
| Register File Size / SM | 256 KB | 256 KB | 256 KB | 256KB |
| Register File Size / GPU | 3840 KB | 6144 KB | 14336 KB | 20480 KB |
| TDP | 235 Watts | 250 Watts | 300 Watts | 300 Watts |
| Transistors | 7.1 billion | 8 billion | 15.3 billion | 21.1 billion |
| GPU Die Size | 551 $mm^2$ | 601 $mm^2$ | 610 $mm^2$ | 815 $mm^2$ |
| Manufacturing Process | 28 nm | 28 nm | 16 nm FinFET+ | 12 nm FFN |

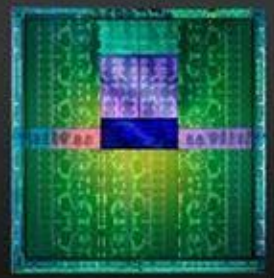
[1] Peak TFLOPS rates are based on GPU Boost Clock

# Accelerated Computing
# 5x Higher Energy Efficiency

**IBM POWER CPU**
Most Powerful Serial Processor

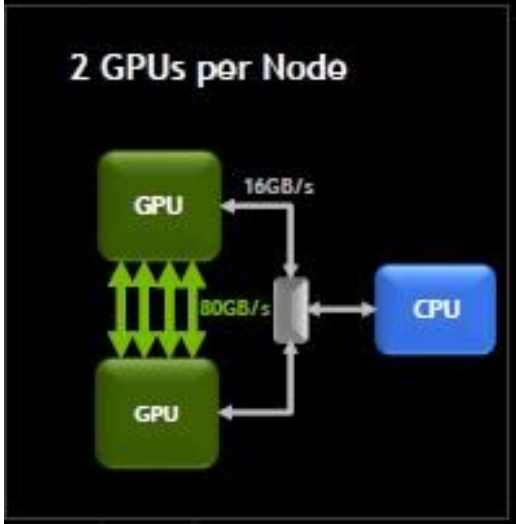


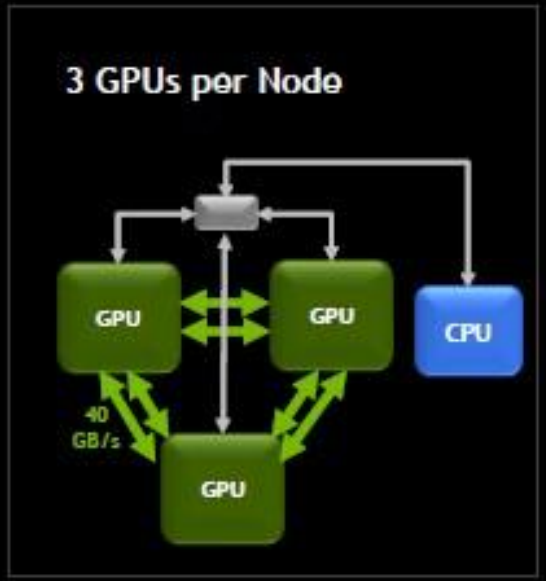

**NVIDIA NVLink**
Fastest CPU-GPU Interconnect

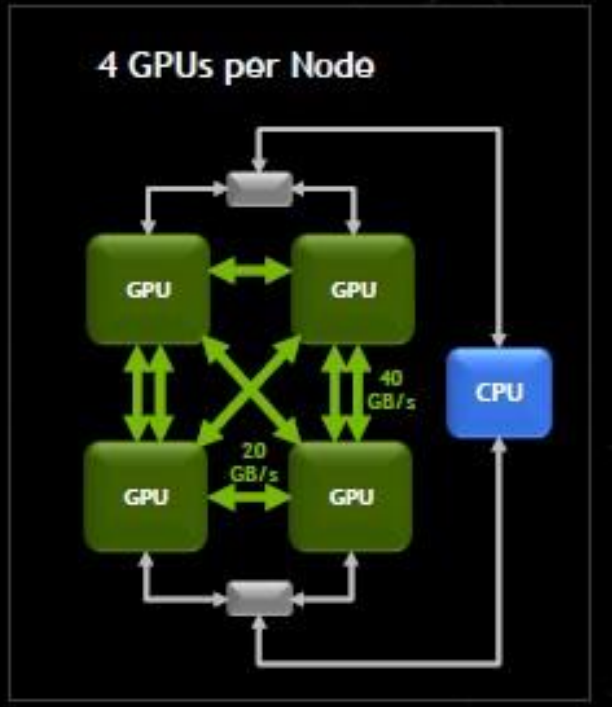


**NVIDIA Volta GPU**
Most Powerful Parallel Processor

# Rich Design Options for Next-Gen Servers




# THANK YOU

**Department of Information Technology**
National Institute of Technology Karnataka, Surathkal

# IT301: INTRODUCTION TO CUDA

By,
Ms. Thanmayee
Adhoc Faculty,
Department of IT,
NITK, Surathkal

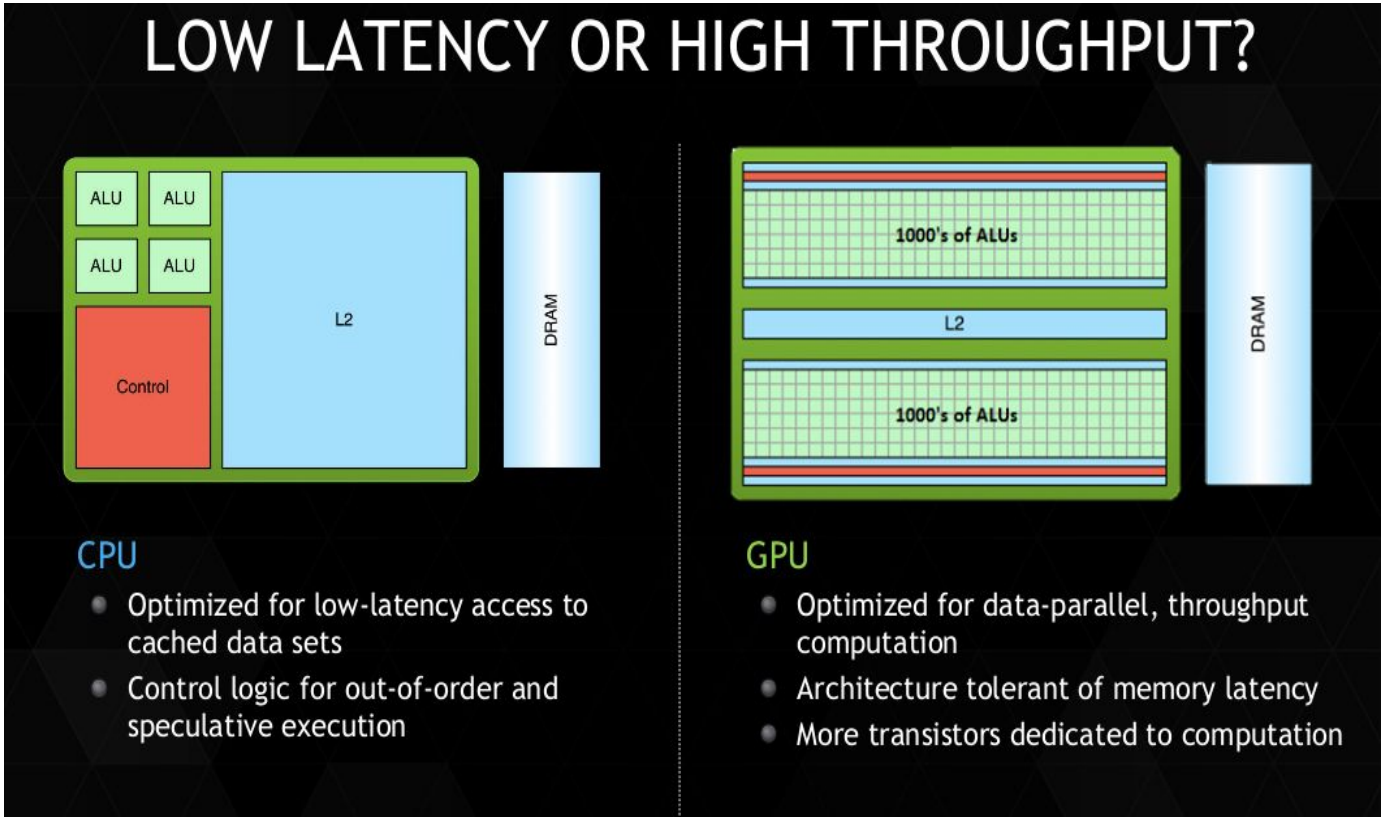
# OUTLINE

- Introduction to GPU
- Evolution of GPU microarchitectures
- General Purpose GPU
- Introduction to CUDA
- CUDA Execution Model
- CUDA Memory Model
- Steps in GPU Execution
- Hello World Program
- CUDA Device Variables
- CUDA Programming examples

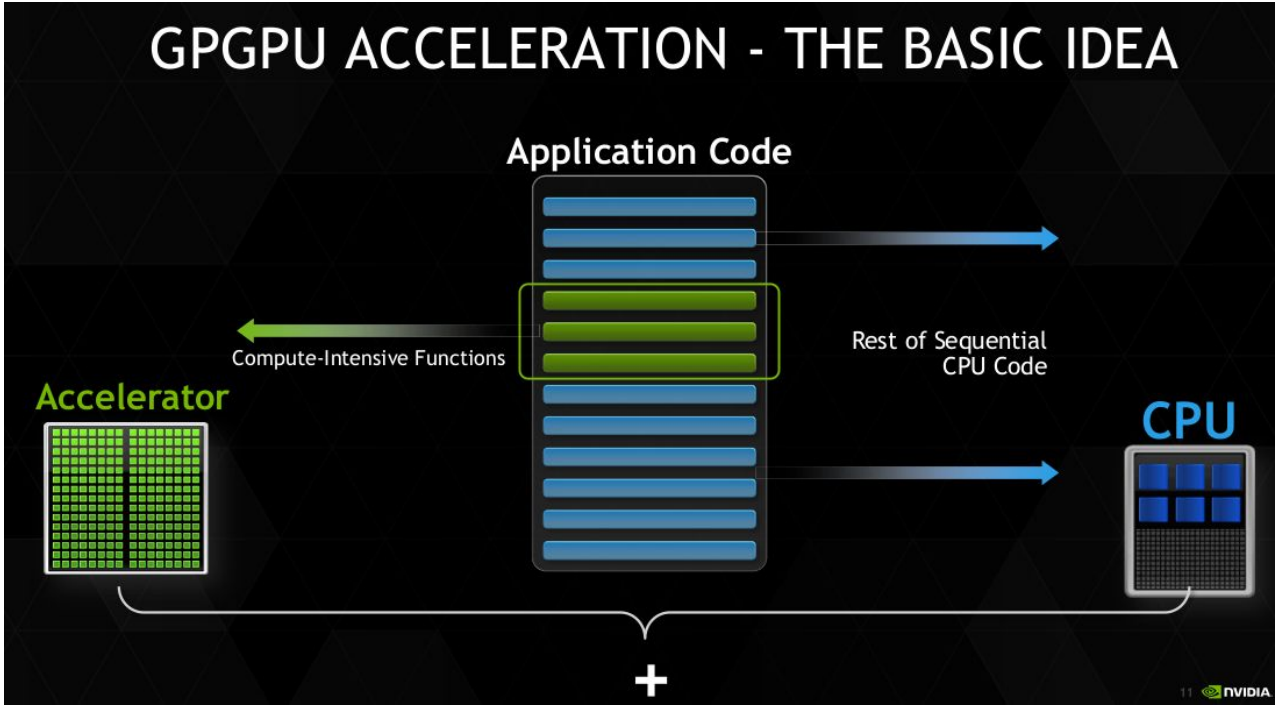
# CPU vs GPU

- Need to understand how CPUs and GPUs differ
  - Simpler calculation versus complex calculation
  - Basic graphics versus 3D rendering, animations.
  - Few higher capacity cores versus many low capacity cores
  - Latency Intolerance versus Latency Tolerance
  - Task Parallelism versus Data Parallelism
  - 10s of Threads versus 10,000s of Threads

# Latency Hiding in GPU



# General Purpose GPU : GPGPU

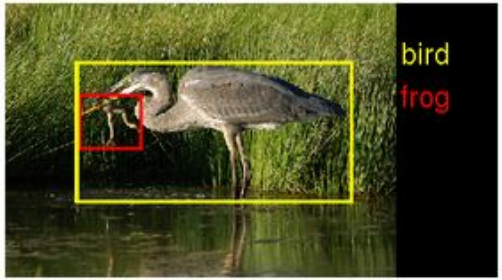


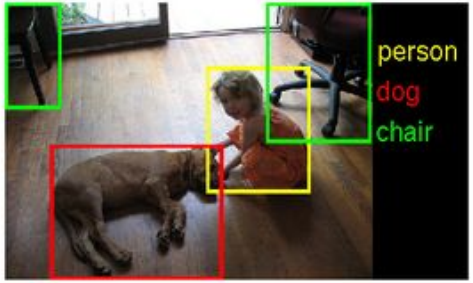
# The dawn of GPGPU

General Purpose Computing on GPU was far from easy back then

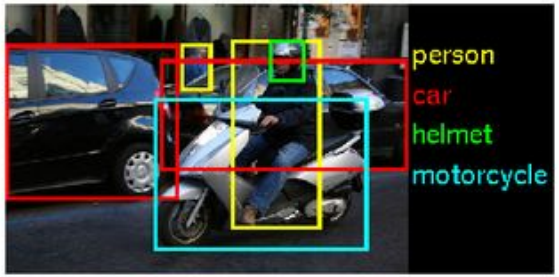
- Even for those who knew graphics programming languages such as OpenGL!
- Developers had to map scientific calculations onto problems that could be represented by triangles and polygons.

# Applications



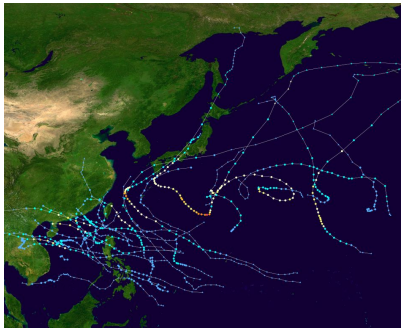



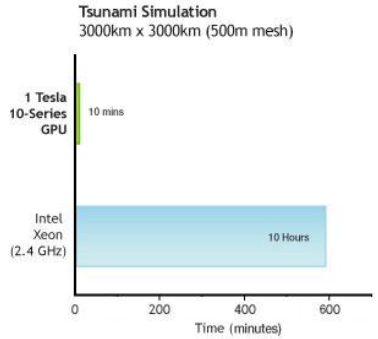


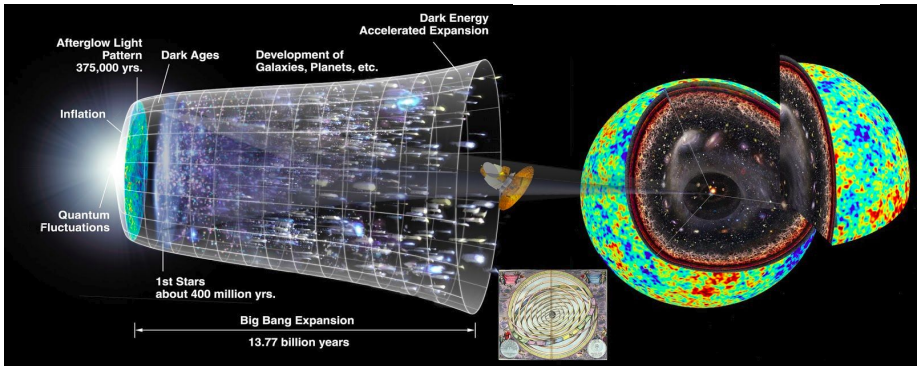


# Applications

- Machine Learning – self driving cars, Watson AI Supercomputer.
- Scientific Applications such as Genome sequencing, molecular simulations.
- Medical Image processing.
- Image tagging in Facebook.
- Numeric weather predictions.
- Oil exploration.
- Movie making.
- Atmospheric simulation.
- Sequencing the novel coronavirus and the genomes of people afflicted with COVID-19.





# CUDA – Compute Unified Device Architecture

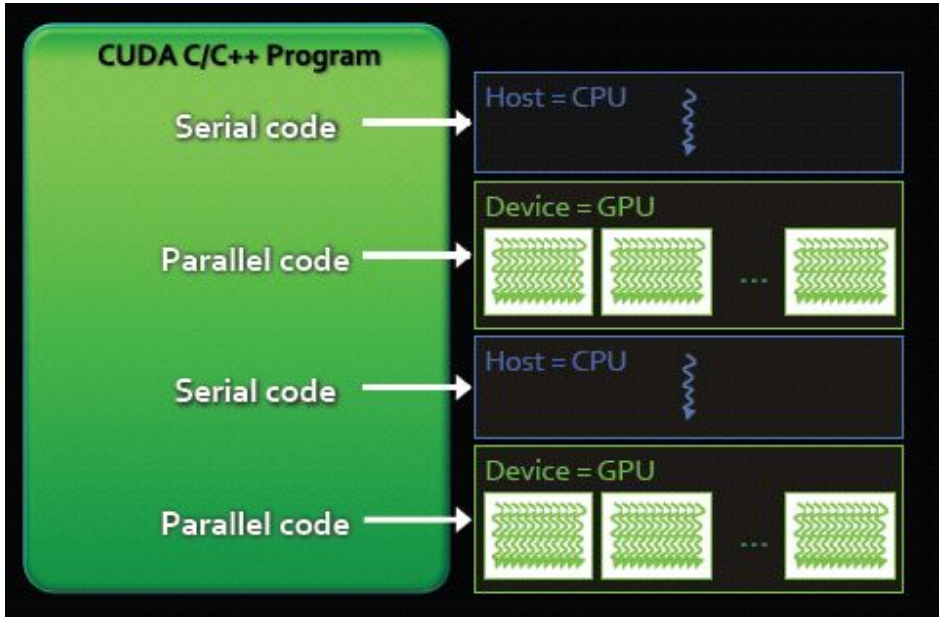
- In 2003, a team of researchers led by Ian Buck unveiled **Brook**, the first widely adopted programming model to extend C with data-parallel constructs.
- Exposed the GPU as a general - purpose processor in a high-level language
  - Most importantly, Brook programs were
    - Easier to write than hand-tuned GPU code
    - Seven times faster than similar existing code

# CUDA – Compute Unified Device Architecture

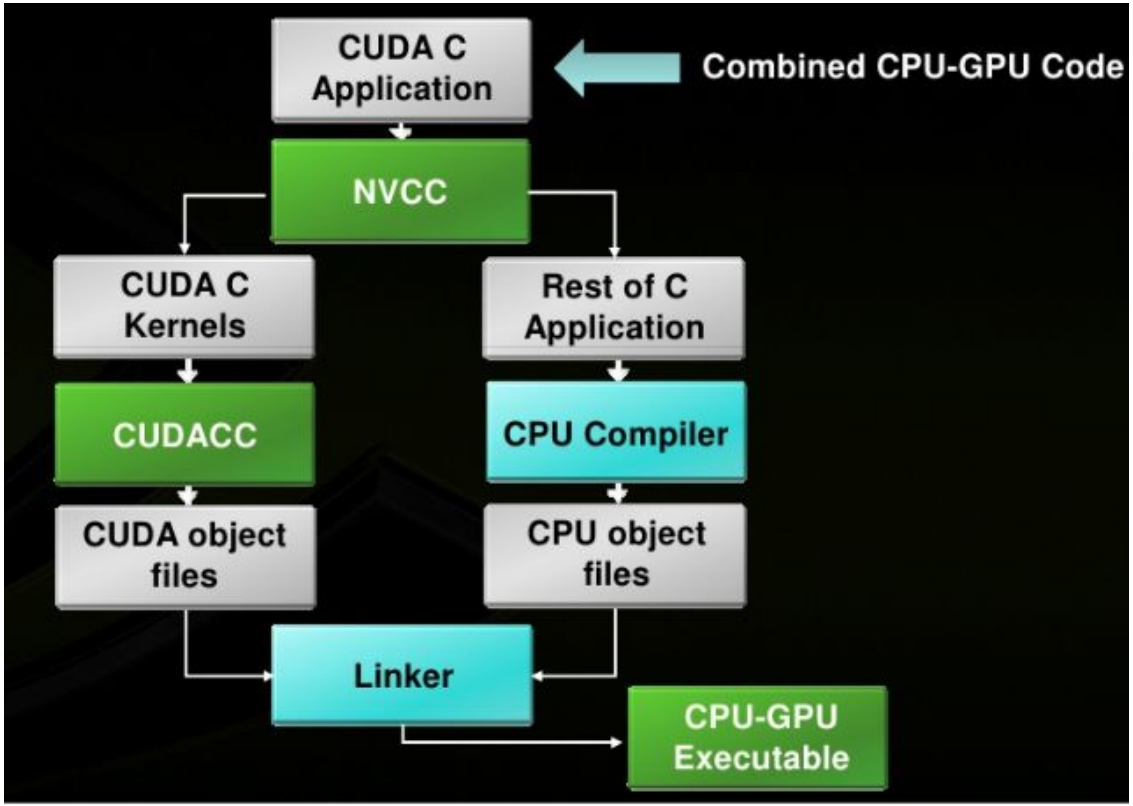
- NVIDIA invited Ian Buck to join the company.
  - Started evolving a solution to seamlessly run C on the GPU.
  - Putting the software and hardware together, NVIDIA unveiled CUDA in 2006
- 
  - CUDA was launched in 2007.
  - The world's first solution for general-computing on GPUs
  - CUDA:
    - is a parallel computing architecture and programming model.
    - Includes C/C++ compiler and also support for OpenCL, DirectCompute.

# General Structure of the GPU Program in CUDA

- Host Program – Executed by the CPU.
  - This is a serial code.
  - Sets up the parameters for GPU (kernel) execution.
- Kernel Program – Executed in Parallel by the SIMD cores (Streaming Processors) in the GPU.

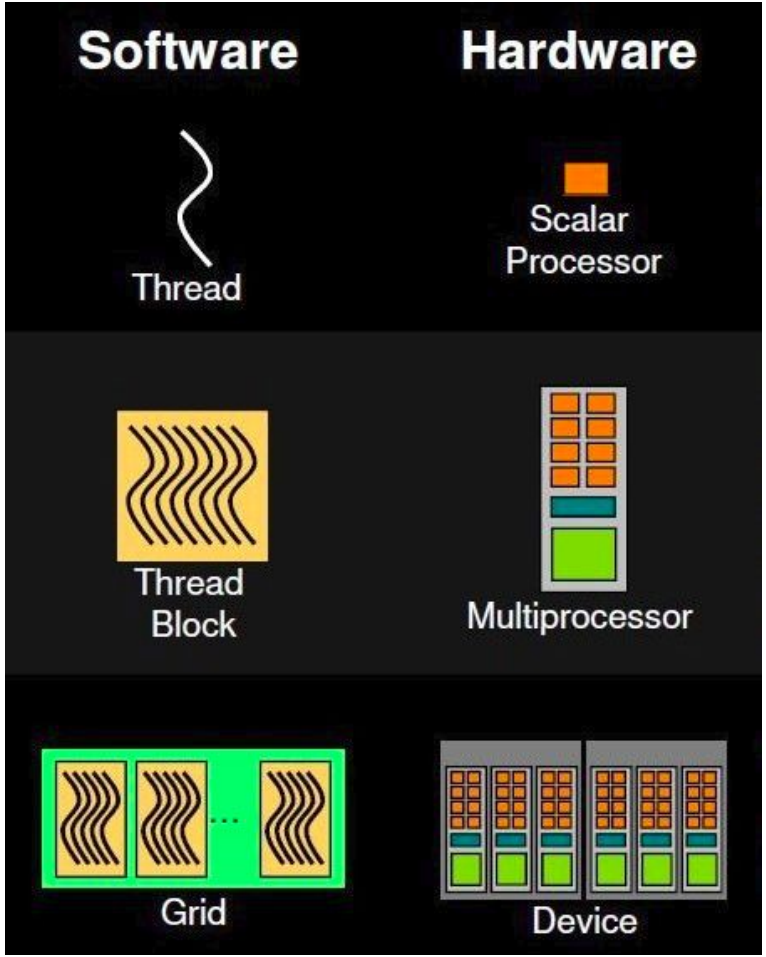


# Compiling CUDA Program:

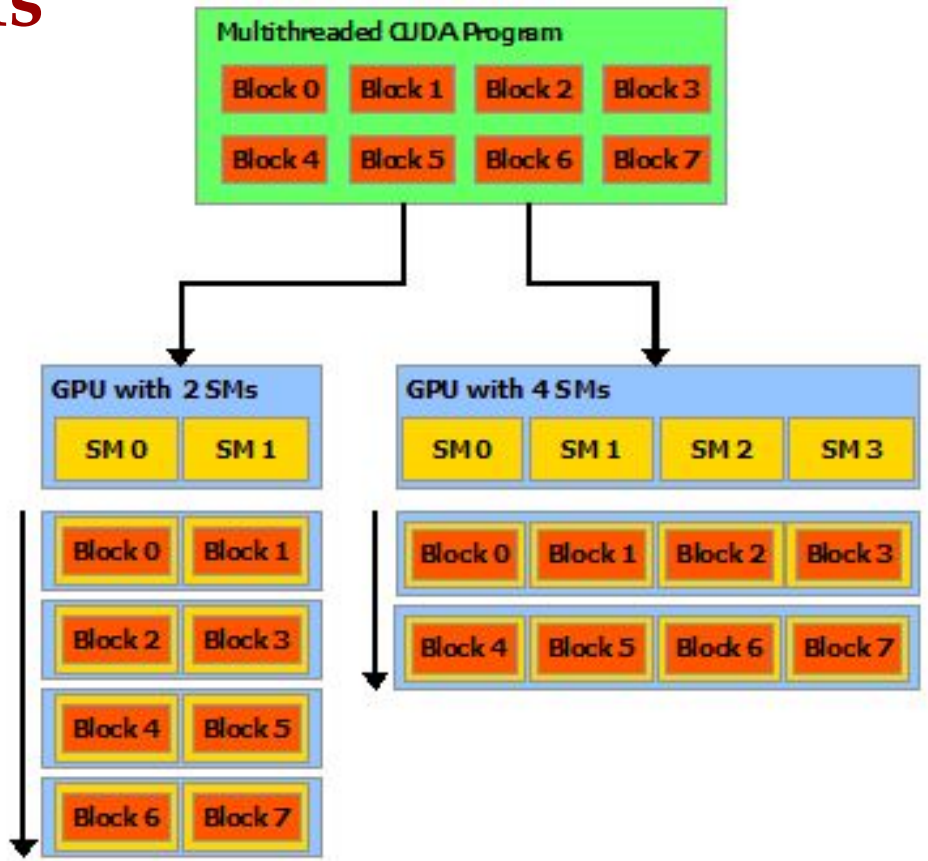


# CUDA Execution Model

- Threads :
  - perform computations. They run on Scalar Processor (Streaming Processors) in GPU.
  - Thousands are needed to get full efficiency.
- Blocks :
  - Group of Threads. Max. Number of Threads vary from 1 to 1024.
  - They are alloted to Streaming Multiprocessors (SMs) in GPU.
  - Multiple blocks can reside in one SM.
- Grid :
  - Group of Blocks.
  - Holds the complete computation task. They represent the Kernel.



# Blocks in SMs



# THANK YOU

**Department of Information Technology**
National Institute of Technology Karnataka, Surathkal

# IT301: INTRODUCTION TO CUDA

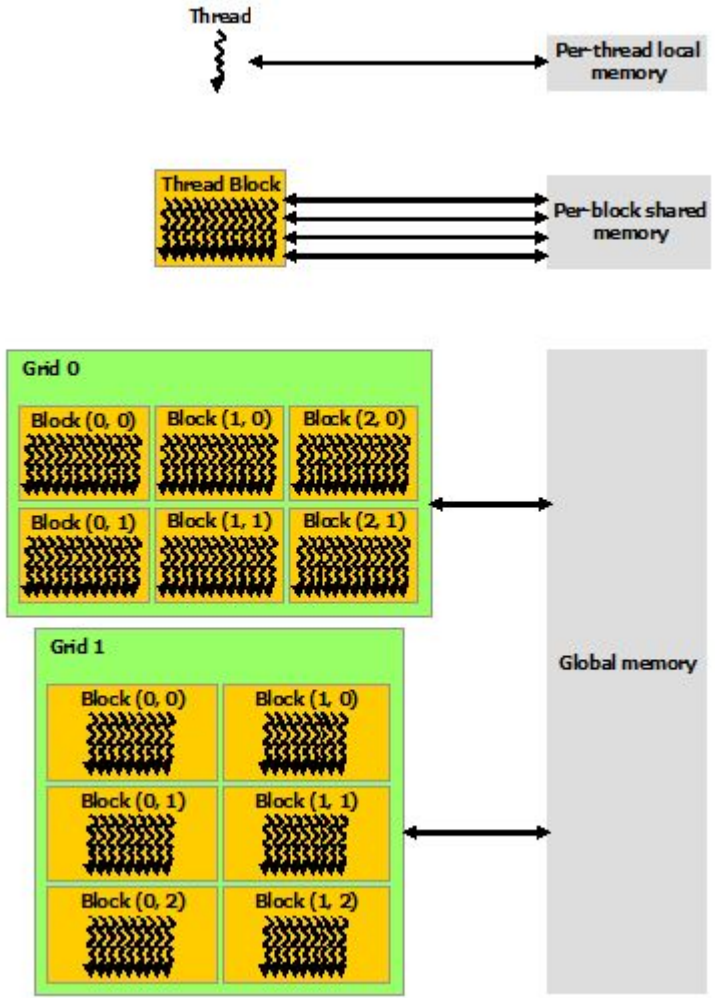
By,
Ms. Thanmayee
Adhoc Faculty,
Department of IT,
NITK, Surathkal

# OUTLINE

- Introduction to GPU
- Evolution of GPU microarchitectures
- General Purpose GPU
- Introduction to CUDA
- CUDA Execution Model
- **CUDA Memory Model**
- **Steps in GPU Execution**
- **Hello World Program**
- **CUDA Device Variables**
- **CUDA Programming examples**

# CUDA Memory Model

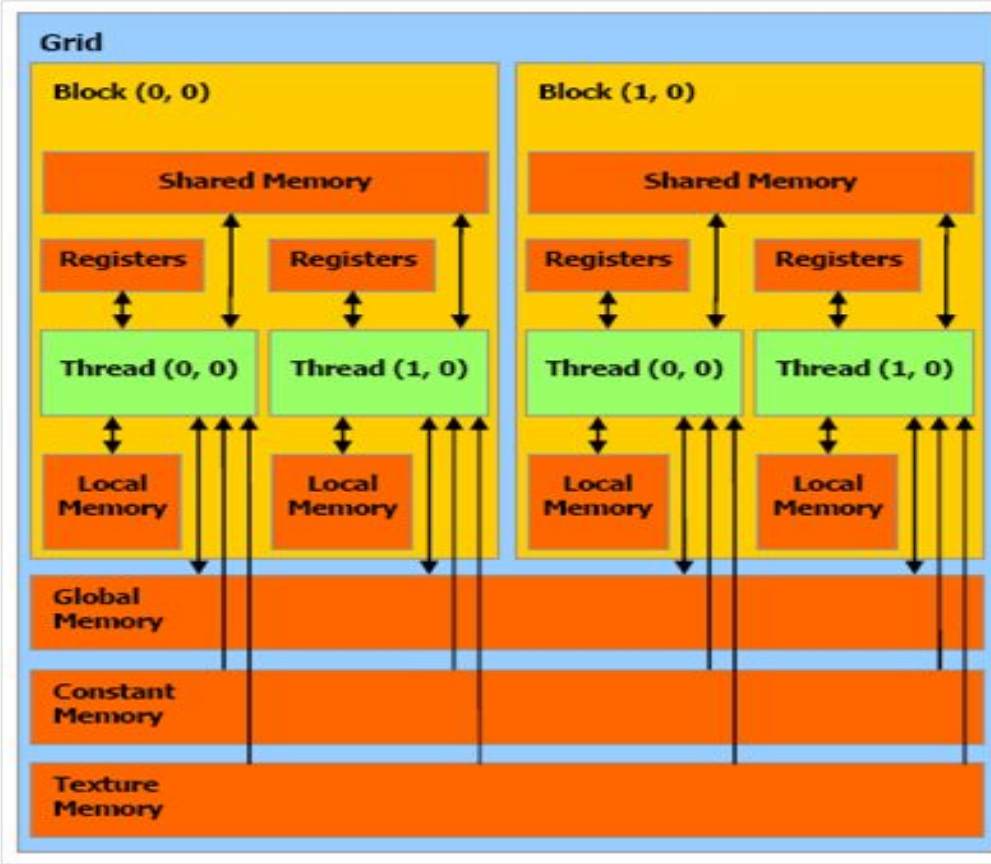
- Local Memory – Accessible only by a single thread in a thread block.
- Shared Memory – All threads in a single thread block have access to the shared memory.
- Global Memory – All thread blocks and Grids (of different Kernels) have access to global memory. Can be read or write.



# CUDA Memory Model

- **Constant Memory** – All thread blocks have access to Constant memory. It is **read only memory**.

- **Texture Memory** – It is a type of Global memory which is **read only cache memory**. Accessible by all the thread blocks. High speed memory.

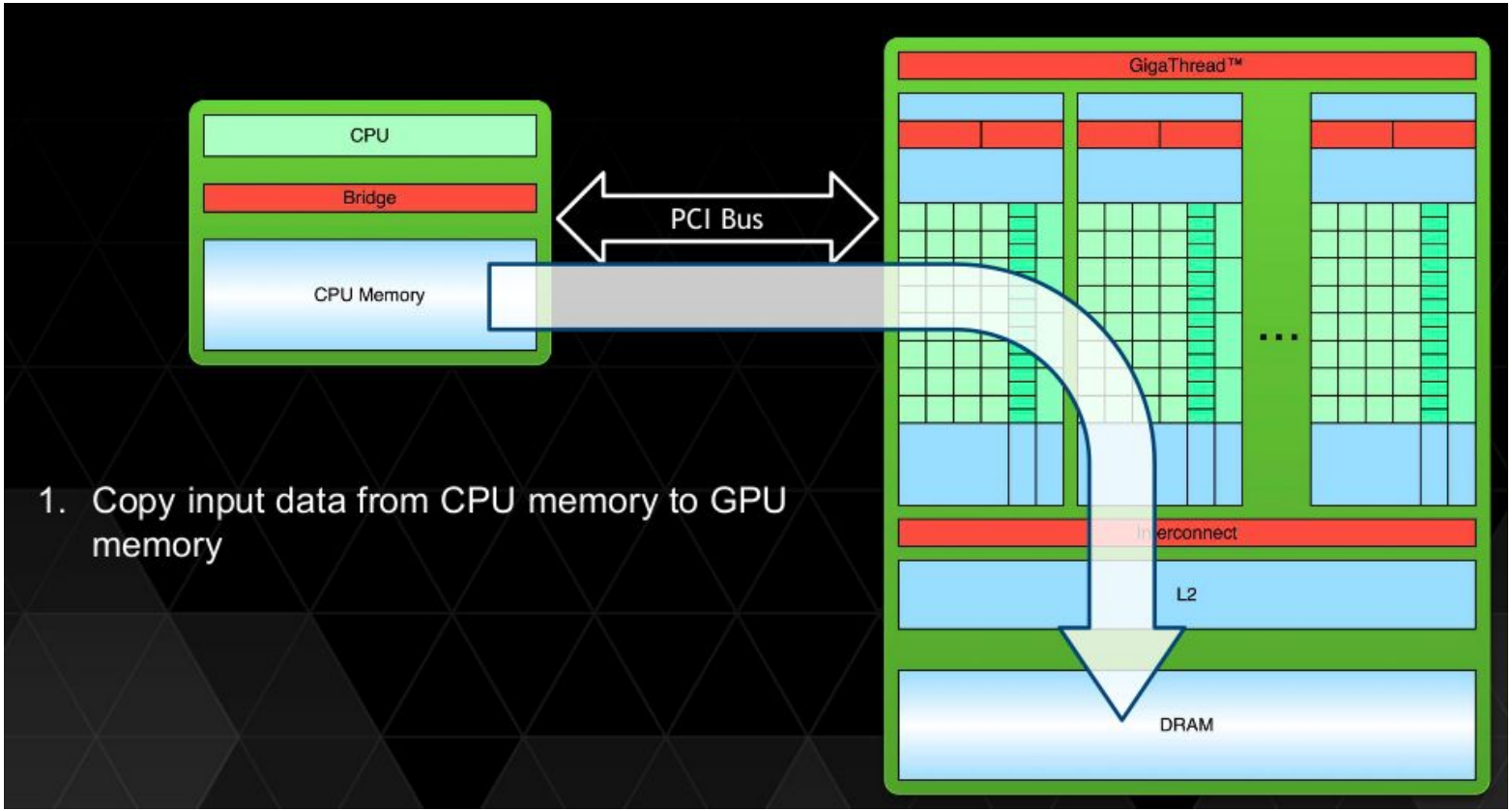


# Execution Flow of CUDA Program

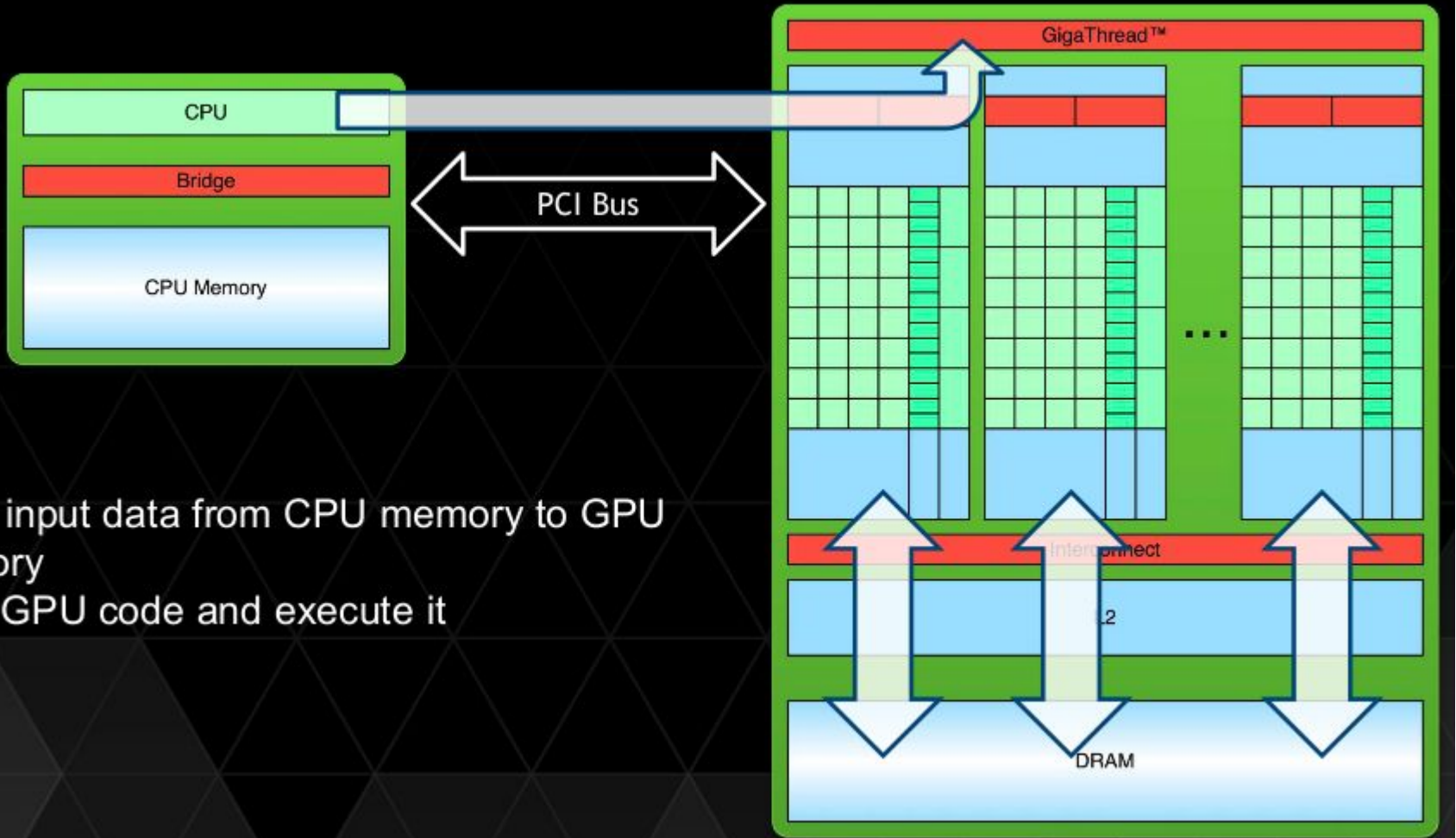
**Step 1: Copy input data from CPU memory to GPU memory**

**Step 2: Launch the kernel on GPU to process the data**

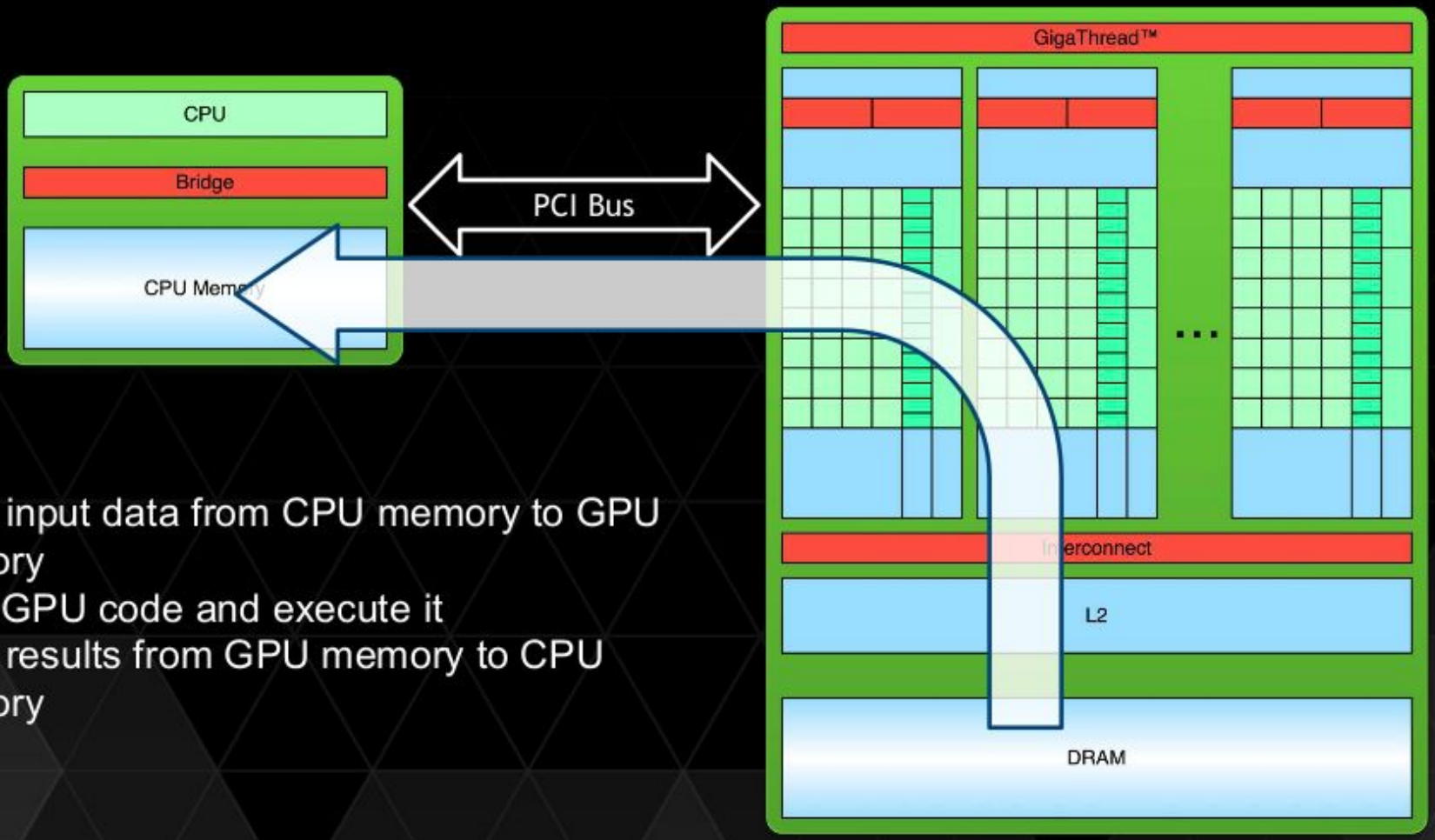
**Step 3: Copy results back to CPU memory from GPU memory**




1. Copy input data from CPU memory to GPU memory




1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it




1. Copy input data from CPU memory to GPU memory
2. Load GPU code and execute it
3. Copy results from GPU memory to CPU memory

# Every programming introduction starts with Hello World!

```
int main(void)
{
printf("Hello World!\n");
return 0;
}
```

# Hello World in CUDA:

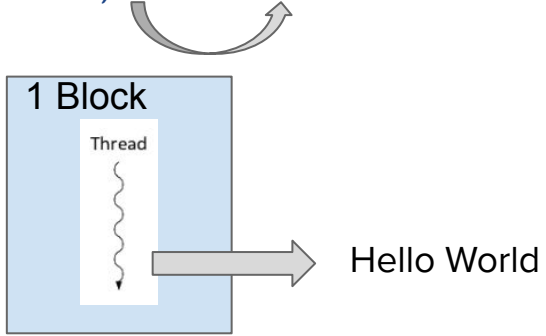
```
__global__ void helloworldKernel(void)
{

}

int main(void)
{
helloworldKernel<<<1,1>>>();
printf("Hello World!\n");
return 0;
}
```

- Keyword __global__ indicates a function that runs on device.It is called from host.
- helloworldKernel() is a device function processed by NVIDIA compiler.
- Host functions are processed by host compiler.
- helloworldKernel()<<<1,1>>>() launches the kernel. Two parameters indicate the number of blocks and number of threads per block.
- What is the output of this code?

# Hello World by Parallel Threads

```
__global__ void helloworldKernel(void)
{
printf("Hello World\n");
}

int main(void)
{
helloworldKernel<<<1,1>>>();
printf("Hello World!\n");
return 0;
}
```

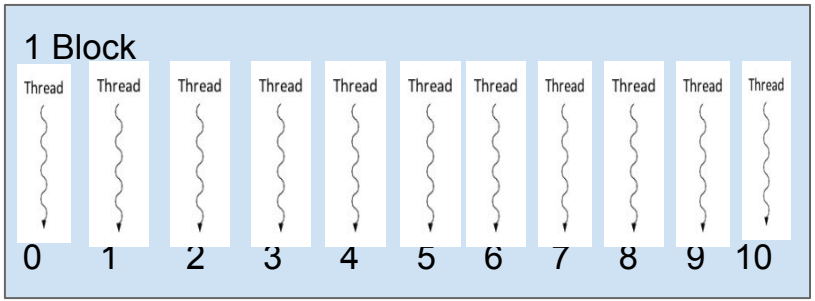
- What do we expect from this code execution?
- How many times the Hello world will be printed?
- <<<1,1>>> number of threads



# Hello World by Parallel Threads

```
__global__ void helloworldKernel(void)
{
printf("Hello World\n");
}

int main(void)
{
helloworldKernel<<<1,10>>>();
printf("Hello World!\n");
return 0;
}
```

- What do we expect from this code execution?
- How many times the Hello world will be printed?
- <<<1,10>>> number of threads



# THANK YOU

**Department of Information Technology**
National Institute of Technology Karnataka, Surathkal

# IT301: INTRODUCTION TO CUDA

By,
Ms. Thanmayee
Adhoc Faculty,
Department of IT,
NITK, Surathkal

# OUTLINE

- Introduction to GPU
- Evolution of GPU microarchitectures
- General Purpose GPU
- Introduction to CUDA
- CUDA Execution Model
- CUDA Memory Model
- Steps in GPU Execution
- Hello World Program
- **CUDA Device Variables**
- **CUDA Programming examples**

# Execution Flow of CUDA Program

Step 1: Copy input data from CPU memory to GPU memory

cudaMemcpy()

This also requires CPU to allocate storage on GPU using cudaMalloc()

Step 2: Launch the kernel on GPU to process the data

kernel<<<blocks,threads>>>

Step 3: Copy results back to CPU memory from GPU memory

cudaMemcpy()

# Simple vector addition in GPU

- Task : c[i]=a[i]+b[i]
- This task needs parallelization.
- which means this task needs to be executed on the GPU cores.
- What are the steps to follow?
  - Allocate memory on GPU
  - Copy input data to GPU
  - Launch the task (Kernel - multiple instances)
  - Copy output data to CPU

# Simple addition in GPU

- **Compute : c[i]=a[i]+b[i]**
- Procedure:
  - // Size of vectors
  - int n = 1000;
  - **// Host input and output vectors**
  - float *h_a, *h_b, *h_c;
  - **// Device input vectors**
  - float *d_a, *d_b, *d_c;
  -

- // Size, in bytes, of each vector
  size_t bytes = n*sizeof(float);

  // Allocate memory for each vector on host
  h_a = (float*)malloc(bytes);
  h_b = (float*)malloc(bytes);
  h_c = (float*)malloc(bytes);

  **// Allocate memory for each vector on GPU**
  cudaMalloc(&d_a, bytes);
  cudaMalloc(&d_b, bytes);
  cudaMalloc(&d_c, bytes);

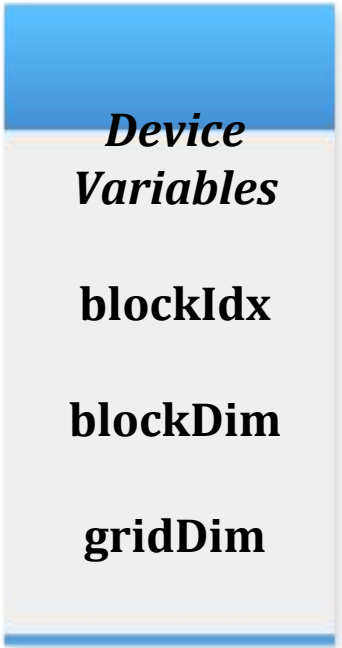
```
int i;
  // Initialize vectors on host
  for( i = 0; i < n; i++ ) {
      h_a[i] = i+1;
      h_b[i] = i+1;
  }

  // Copy host vectors to device
  cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
  cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
```

```
// Execute the kernel. n=1000.
 vecAdd<<<1,n>>>(d_a, d_b, d_c, n);

// Copy array back to host
  cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

   float  sum = 0;
  for(i=0; i<n; i++)
      sum += h_c[i];
  printf("final result: %f\n", sum);
```

```
// Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);
    return 0;
}
```

# Using Only Threads

```
__global__ void vecAdd(float *a, float *b,float *c, int n)
{
    int id = threadIdx.x;
   // Make sure we do not go out of bounds
   if (id < n)
      c[id] = a[id] + b[id];
}
```



# Using multiple blocks

**In terms of multiple blocks:**

```
vecAdd<<<n,1>>>(d_a, d_b, d_c, n);

__global__ void vecAdd(int *a, int *b,int *c, int n)
{
    int id = blockIdx.x;

  // Make sure we do not go out of bounds
  if (id < n)
     c[id] = a[id] + b[id];
}
```

# THANK YOU

**Department of Information Technology**
National Institute of Technology Karnataka, Surathkal

# IT301: INTRODUCTION TO CUDA

By,
Ms. Thanmayee
Adhoc Faculty,
Department of IT,
NITK, Surathkal

# OUTLINE

- Introduction to GPU
- Evolution of GPU microarchitectures
- General Purpose GPU
- Introduction to CUDA
- CUDA Execution Model
- CUDA Memory Model
- Steps in GPU Execution
- Hello World Program
- **CUDA Device Variables**
- **CUDA Programming examples**

# Execution Flow of CUDA Program

Step 1: Copy input data from CPU memory to GPU memory

cudaMemcpy()

This also requires CPU to allocate storage on GPU using cudaMalloc()

Step 2: Launch the kernel on GPU to process the data

kernel<<<blocks,threads>>>

Step 3: Copy results back to CPU memory from GPU memory

cudaMemcpy()

# Simple vector addition in GPU

- Task : c[i]=a[i]+b[i]
- This task needs parallelization.
- which means this task needs to be executed on the GPU cores.
- What are the steps to follow?
  - Allocate memory on GPU
  - Copy input data to GPU
  - Launch the task (Kernel - multiple instances)
  - Copy output data to CPU

# Simple addition in GPU

- **Compute : c[i]=a[i]+b[i]**
- Procedure:
  - // Size of vectors
  - int n = 1000;
  - **// Host input and output vectors**
  - float *h_a, *h_b, *h_c;
  - **// Device input vectors**
  - float *d_a, *d_b, *d_c;
  -

- // Size, in bytes, of each vector
  size_t bytes = n*sizeof(float);

  // Allocate memory for each vector on host
  h_a = (float*)malloc(bytes);
  h_b = (float*)malloc(bytes);
  h_c = (float*)malloc(bytes);

  **// Allocate memory for each vector on GPU**
  cudaMalloc(&d_a, bytes);
  cudaMalloc(&d_b, bytes);
  cudaMalloc(&d_c, bytes);

```
int i;
  // Initialize vectors on host
  for( i = 0; i < n; i++ ) {
      h_a[i] = i+1;
      h_b[i] = i+1;
  }

  // Copy host vectors to device
  cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
  cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
```

```
// Execute the kernel. n=1000.
 vecAdd<<<1,n>>>(d_a, d_b, d_c, n);

// Copy array back to host
  cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

   float  sum = 0;
  for(i=0; i<n; i++)
      sum += h_c[i];
  printf("final result: %f\n", sum);
```

```
// Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);
    return 0;
}
```

# Using Only Threads

```
__global__ void vecAdd(float *a, float *b,float *c, int n)
{
    int id = threadIdx.x;
   // Make sure we do not go out of bounds
   if (id < n)
      c[id] = a[id] + b[id];
}
```

| *Device Variables* |
| --- |
| threadIdx.x |
| blockIdx |
| blockDim |
| gridDim |

# Device Variables

- ***threadIdx:*** *Used to access the index of a thread inside a thread block*
  - threadIdx.x : is Thread index in a block in x dimension.
  - threadIdx.y = Index of a thread inside a block in Y direction
  - threadIdx.z = Index of a thread inside a block in Z direction
- ***blockDim*** = *Number of threads in the block for a specific direction*
  - blockDim.x = Number of threads in the block for X direction
  - blockDim.y = Number of threads in the block for Y direction
  - blockDim.z = Number of threads in the block for Z direction

# DEvice Variables

- ***blockIdx*** = *Used to access an index of a thread block inside a thread grid*
  - blockIdx.x = Index of a tread block in X direction
  - blockIdx.y = Index of a tread block in Y direction
  - blockIdx.z = Index of a tread block in Z direction
- ***gridDim*** = *Number of thread blocks in a specific direction.*
  - gridDim.x = Number of thread blocks in X direction
  - gridDim.y = Number of thread blocks in Y direction
  - gridDim.z = Number of thread blocks in Z direction

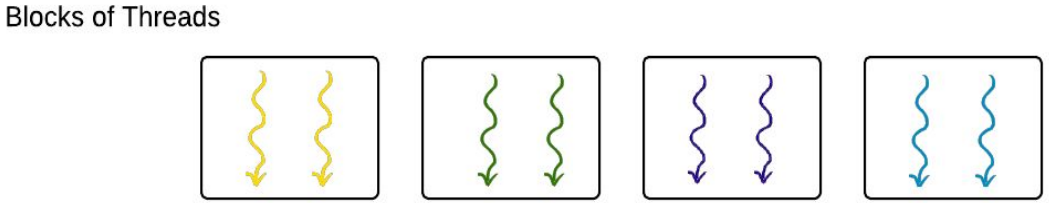
# Using multiple blocks

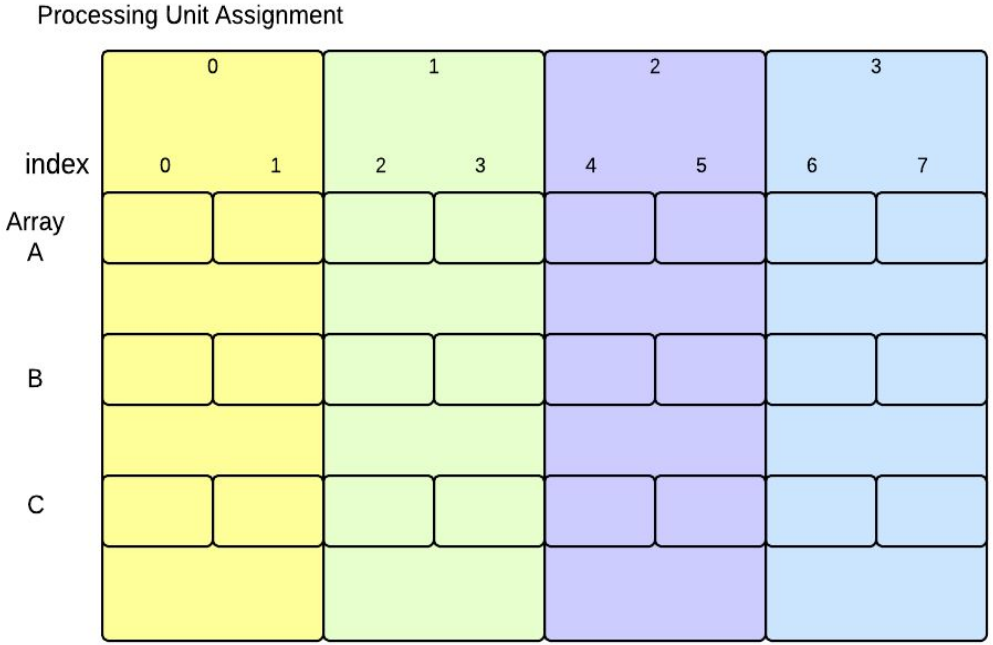
**In terms of multiple blocks:**

```
vecAdd<<<n,1>>>(d_a, d_b, d_c, n);

__global__ void vecAdd(int *a, int *b,int *c, int n)
{
    int id = blockIdx.x;

  // Make sure we do not go out of bounds
  if (id < n)
     c[id] = a[id] + b[id];
}
```

# Blocks of Threads

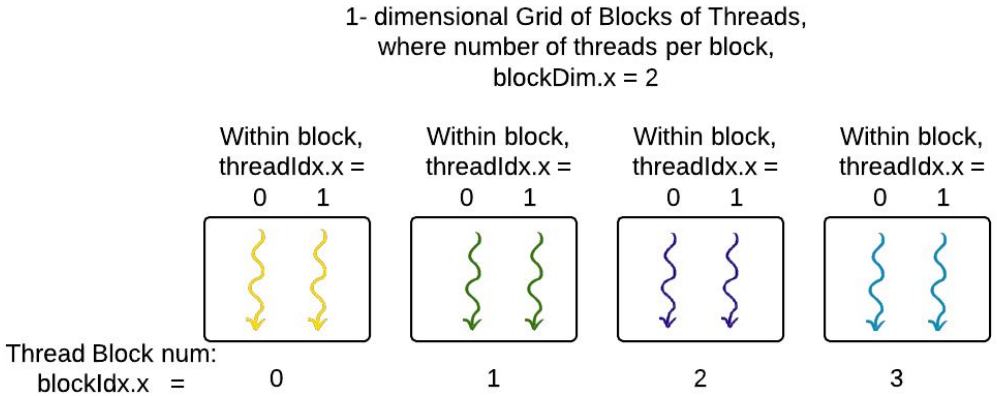
- Maximum number of threads per block is 1024.
- Number of registers per Block (SM) is 65536.
- Max number of blocks in Grid in x direction is 65535 and y direction is 65535.
- Warp is a batch of 32 threads

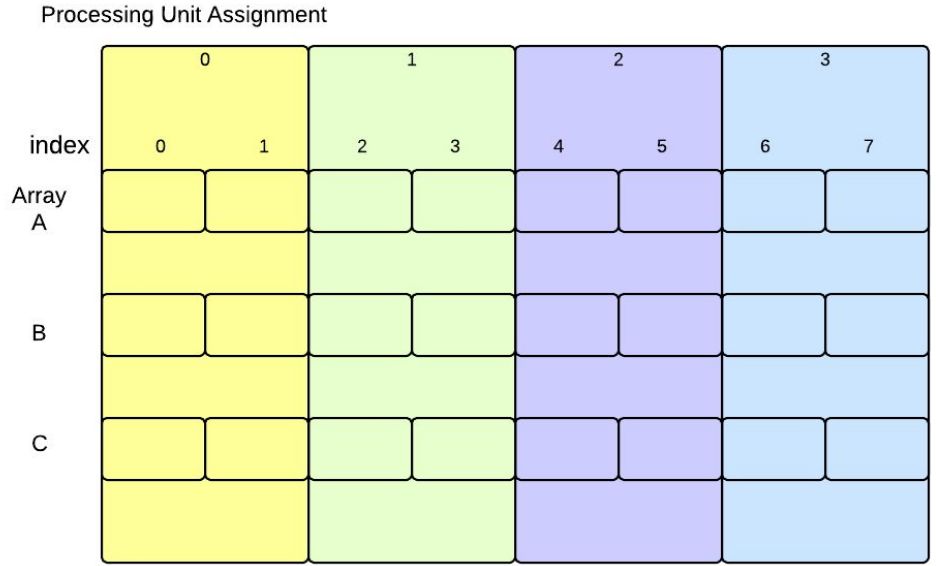


# Blocks of Threads

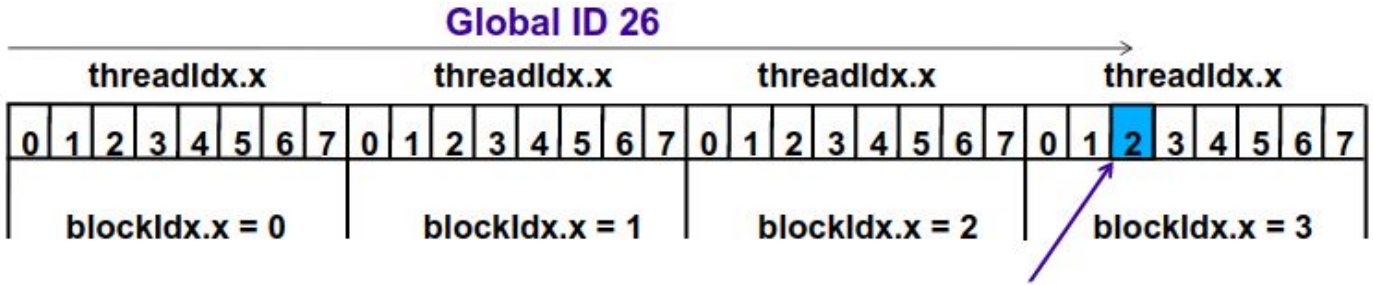
- We have 4 blocks.
- Each block has 2 threads.
- In the example you see the thread IDs within a block. Blocks also have IDs.
- Now we need to calculate the global thread ID.



# Blocks of Threads

In the following example: There are 4 blocks. Each block has 8 threads.




globalThreadId = threadIdx.x+blockIdx.x*blockDim.x;

# Using multiple blocks and threads

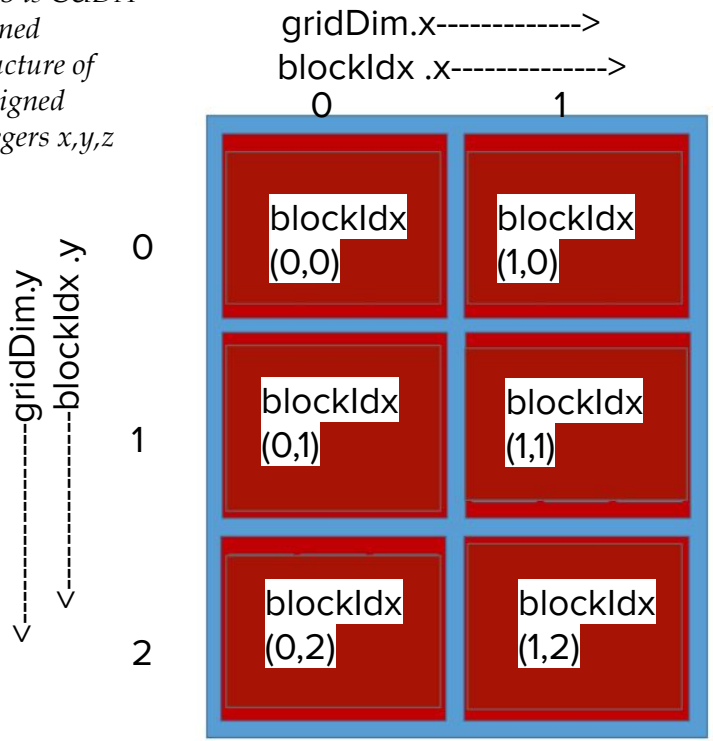
**In terms of multiple blocks and threads:**

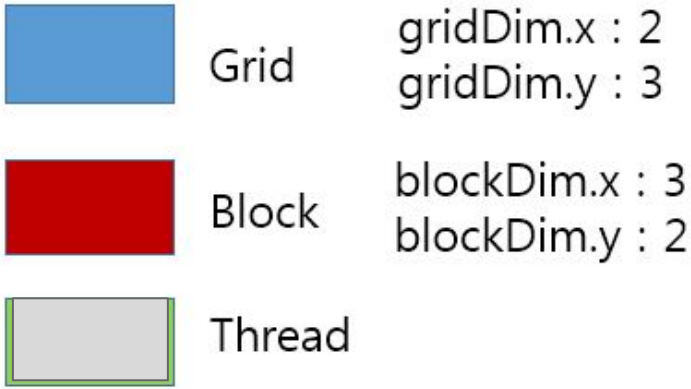
```
vecAdd<<<2,500>>>(d_a, d_b, d_c, n);

__global__ void vecAdd(int *a, int *b,int *c, int n)
{
    int id = threadIdx.x+blockIdx.x*blockDim.x;

  // Make sure we do not go out of bounds
  if (id < n)
     c[id] = a[id] + b[id];
}
```

```
dim3 blocks(2,3);
dim3 threads(3,2)
Kernel<<<blocks, threads>>>
```

*blocks(x,y) → blocks(2,3) means : 2 blocks on x dimension and 3 blocks on y dimension*
*threads(x,y) → threads(3,2) means : 3 threads on x dimension and 2 threads on y dimension)*

*dim3 is CUDA defined structure of unsigned integers x,y,z*

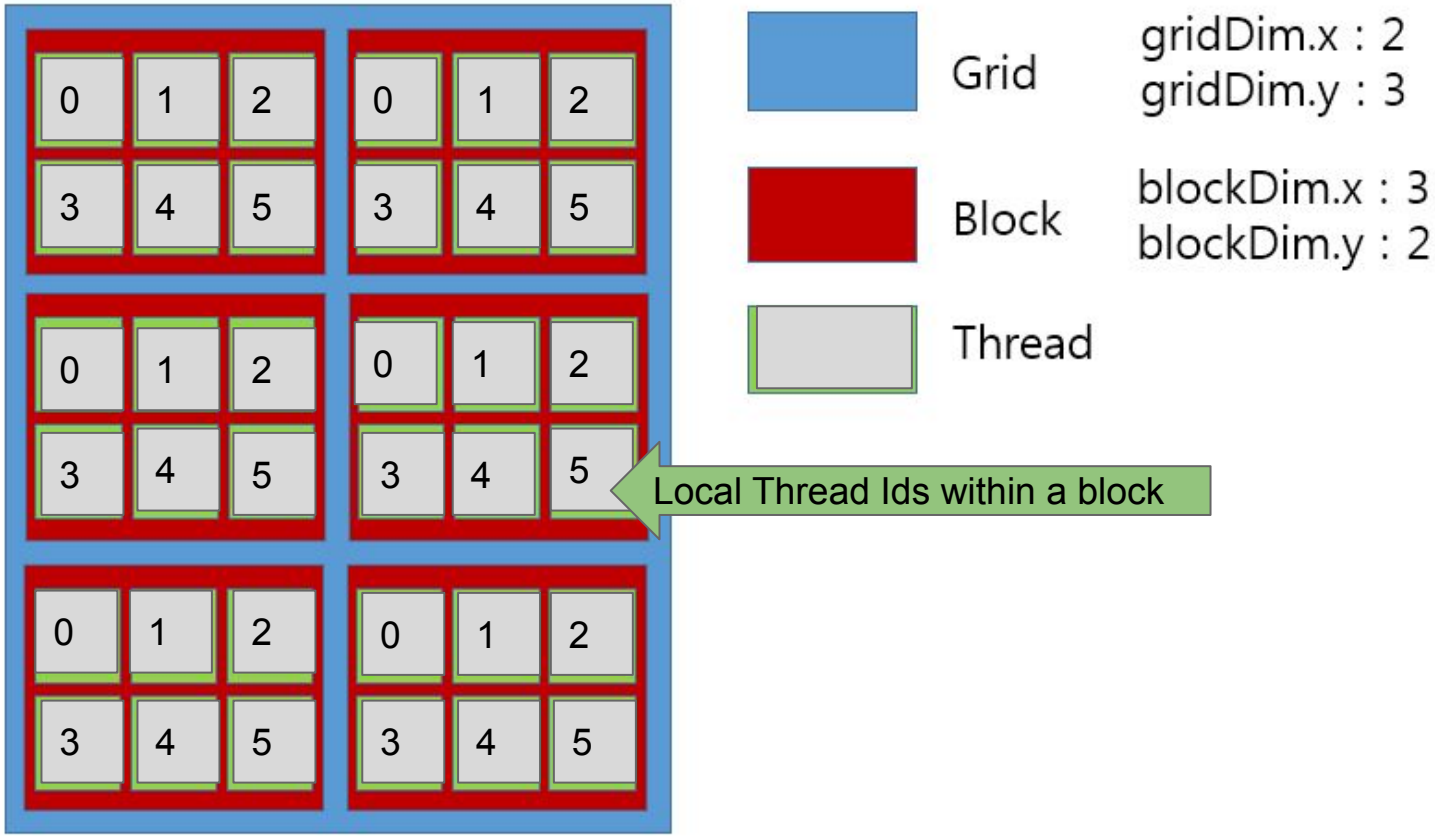


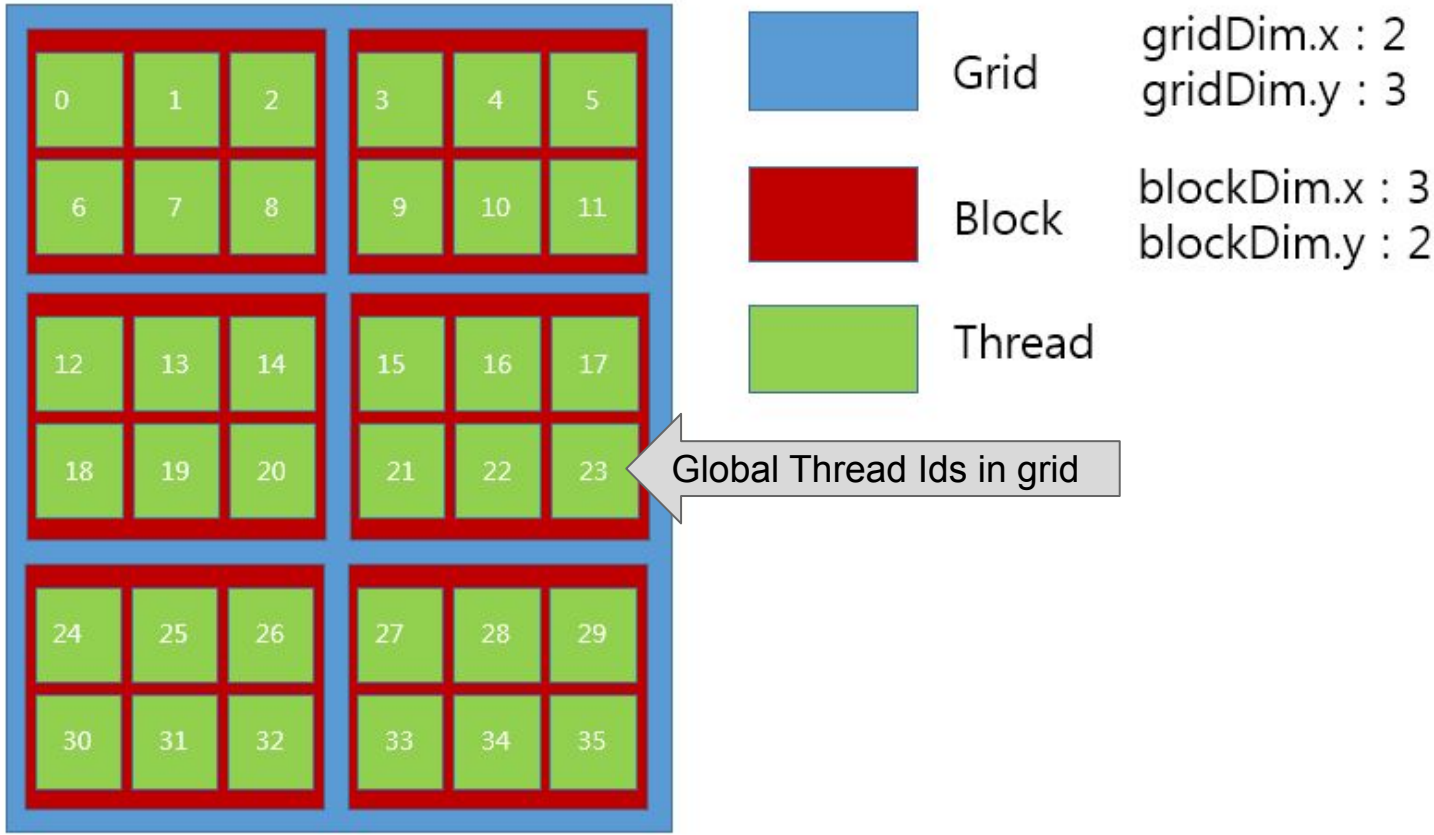

Find the blockIDx values:

blockIdx(0,0) means blockIdx.x=0, blockIdx.y=0
blockIdx(0,1) means blockIdx.x=0, blockIdx.y=1

```
dim3 blocks(2,3);
dim3 threads(3,2)
Kernel<<<blocks, threads>>>
```



```
dim3 blocks(2,3);
dim3 threads(3,2)
Kernel<<<blocks, threads>>>
```

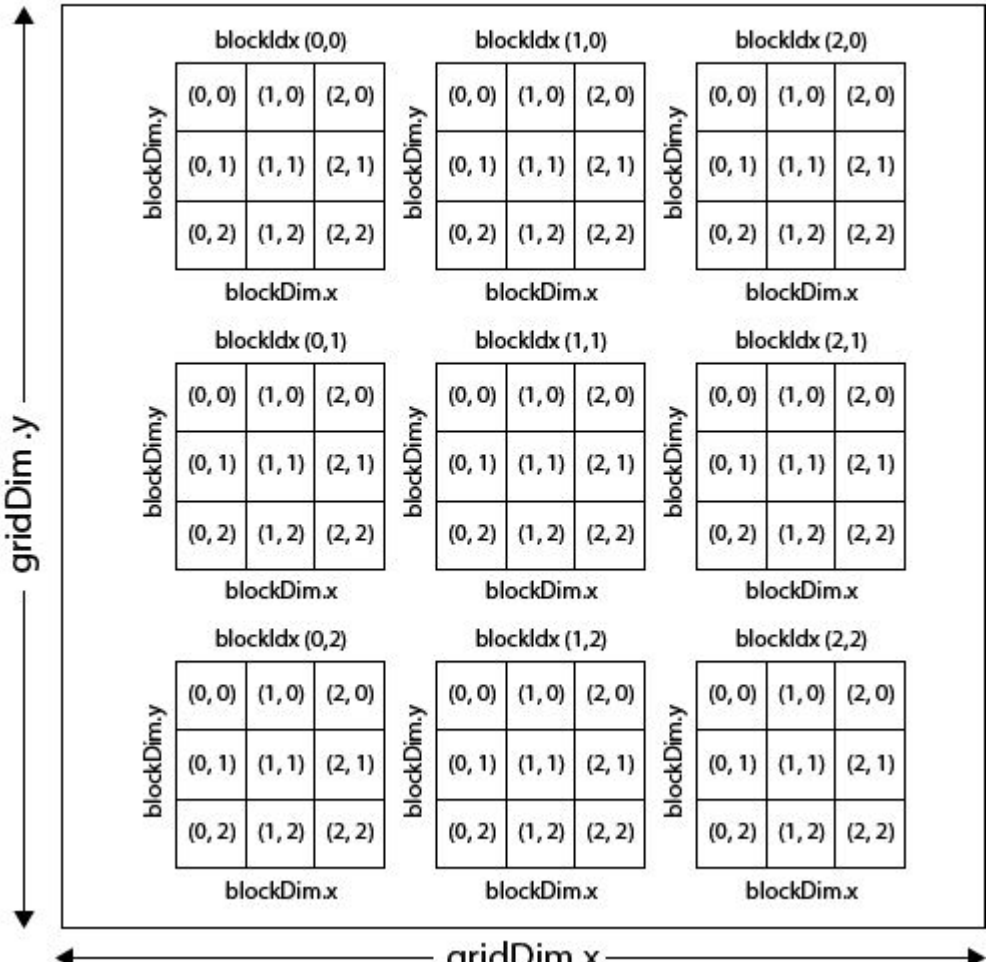


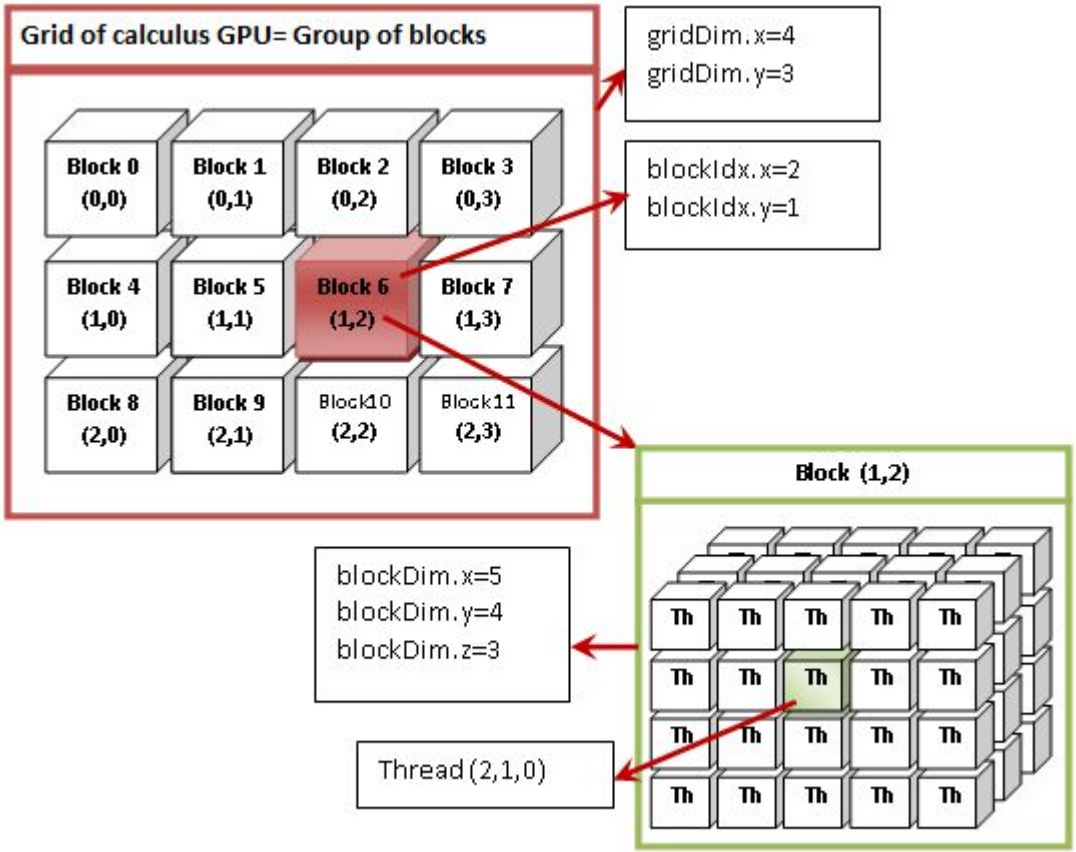
# Calculating Global Thread ID with 2D

Calculate global tid:

- **x=threadIdx.x+blockIdx.x*blockDim.x**
- **y=threadIdx.y+blockIdx.y*blockDim.y**
- **tid=x+y*blockDim.x*gridDim.x**

# CUDA Grid


gridDim .y
gridDim .x
blockIdx (0,0)
blockIdx (1,0)
blockIdx (2,0)
blockIdx (0,1)
blockIdx (1,1)
blockIdx (2,1)
blockIdx (0,2)
blockIdx (1,2)
blockIdx (2,2)
blockDim.y
blockDim.x
(0, 0) (1, 0) (2, 0)
(0, 1) (1, 1) (2, 1)
(0, 2) (1, 2) (2, 2)


Grid of calculus GPU= Group of blocks
gridDim.x=4
gridDim.y=3
Block 0 (0,0)
Block 1 (0,1)
Block 2 (0,2)
Block 3 (0,3)
Block 4 (1,0)
Block 5 (1,1)
Block 6 (1,2)
Block 7 (1,3)
Block 8 (2,0)
Block 9 (2,1)
Block10 (2,2)
Block11 (2,3)
blockIdx.x=2
blockIdx.y=1
Block (1,2)
blockDim.x=5
blockDim.y=4
blockDim.z=3
Th
Thread (2,1,0)

# Answer this

- Consider the input size is 10,00,000. The computation on this data must be performed on GPU. What will be the values of kernel launch parameters?

## Answer this

- The number of registers available in SM is 65536. How many registers will be available for the threads in a block of size 1024?

# Answer this

- If each thread requires 32 registers.Say there are 1024 threads in one block. How many blocks can reside in one SM for execution?

# Answer this

- Calculate the global thread ID for the 3rd Thread in the second Block in a 1D Grid.

# THANK YOU

**Department of Information Technology**
National Institute of Technology Karnataka, Surathkal

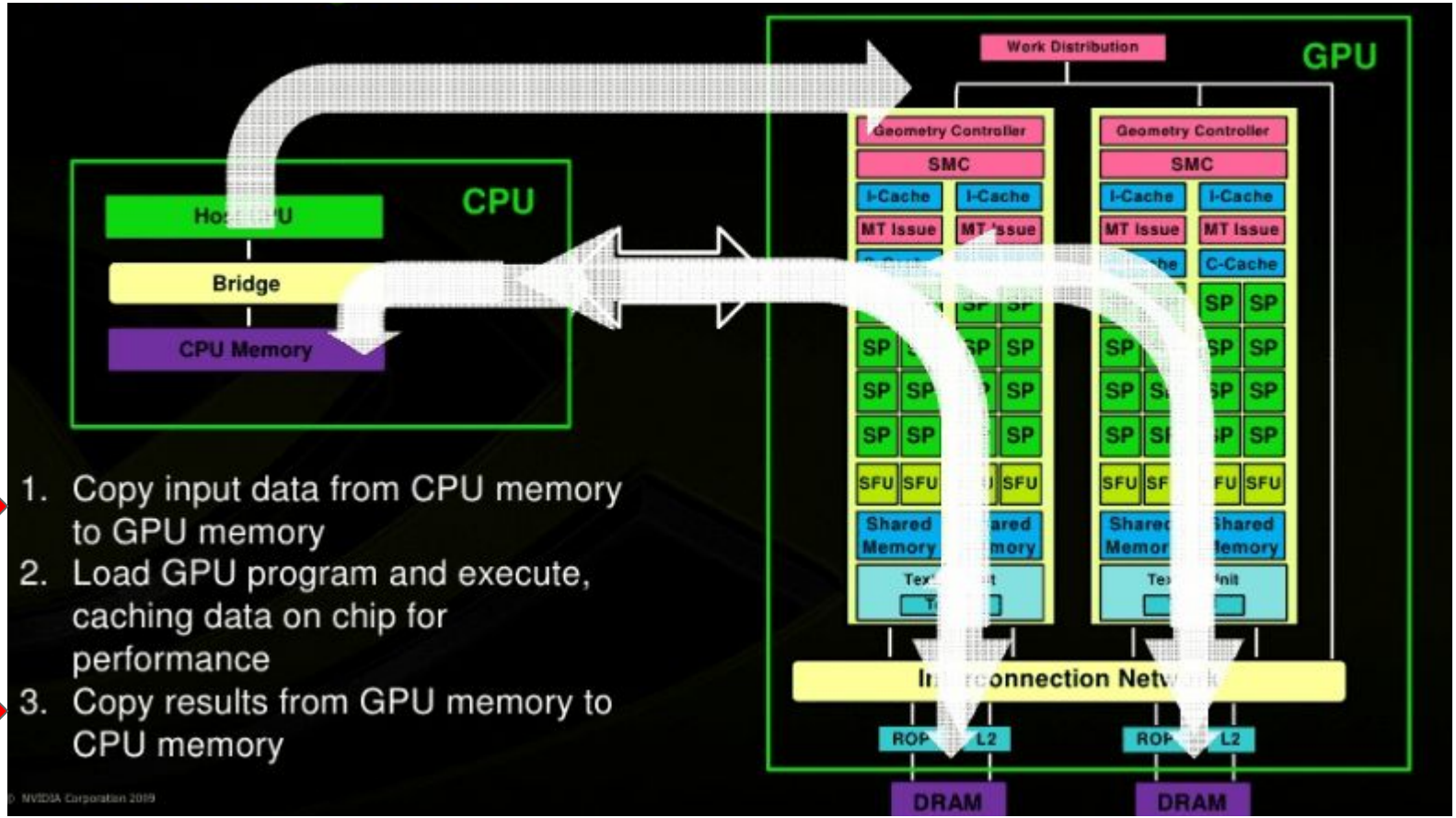
# IT301: INTRODUCTION TO CUDA

By,
Ms. Thanmayee
Adhoc Faculty,
Department of IT,
NITK, Surathkal

# Concepts Covered

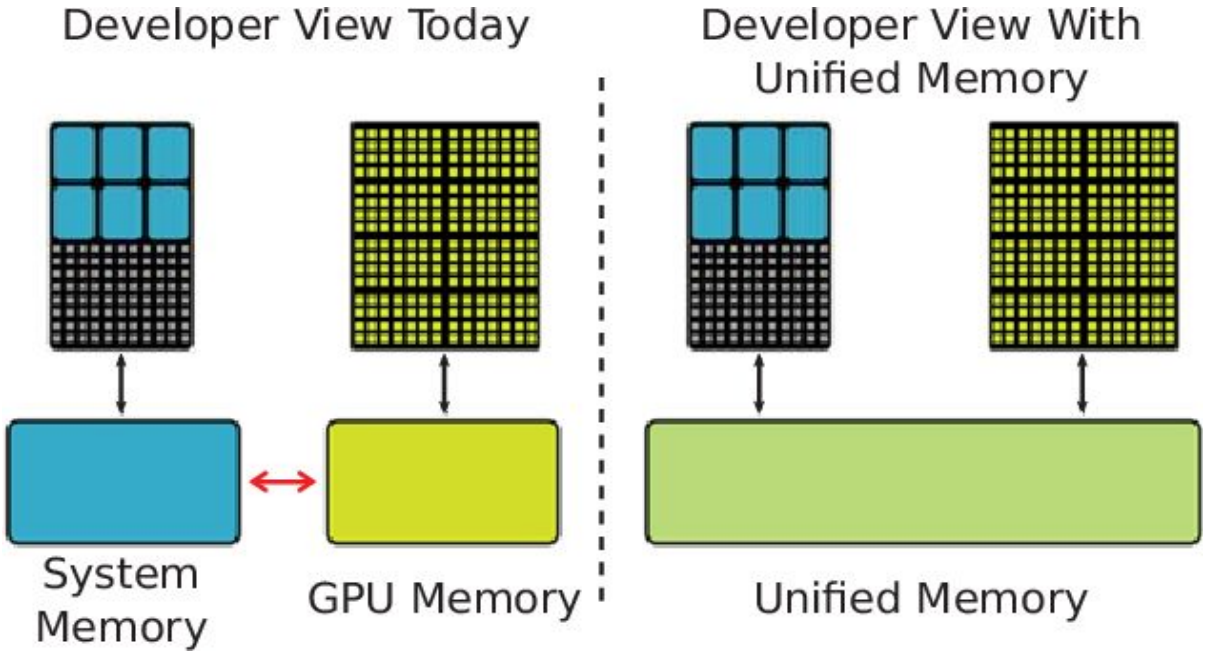
- Introduction to GPU
- Evolution of GPU microarchitectures
- General Purpose GPU
- Introduction to CUDA
- CUDA Execution Model
- CUDA Memory Model
- Steps in GPU Execution
- Hello World Program
- CUDA Device Variables
- CUDA Programming examples

# RECALL CUDA Processing Flow



# CUDA Unified Memory

Simplifies programming by enabling applications to access CPU and GPU memory without the need to manually copy data from one to the other.



# CUDA Unified Memory

- Traditionally programmer had to allocate memory space in both GPU and CPU and explicitly move data between CPU and GPU.
- With unified memory, we allocate a pool of managed memory that can be accessible from any processor.
- The CUDA runtime manages migrating data between CPU and GPU memory.
- Benefits:
  - Simplified Code
  - Simplified Code Porting
  - Global Coherency
  - Simpler Memory Model by eliminating deep copies.
  - Prefetching can be used for higher optimizations.

# The Simplified Memory Management:

## CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

## Ordinary CUDA Code

```
void sortfile(FILE *fp, int N) {
  char *data, *d_data;
  data = (char *)malloc(N);
  cudaMalloc(&d_data, N);
  fread(data, 1, N, fp);
  cudaMemcpy(d_data, data, N, …); // 1
  qsort<<<...>>>(data,N,1,compare); // 2
  cudaMemcpy(data, d_data, N, …); // 3

  use_data(data);
  cudaFree(d_data);
  free(data);
}
```

# The Simplified Memory Management:

## CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

## CUDA Code with Unified Memory

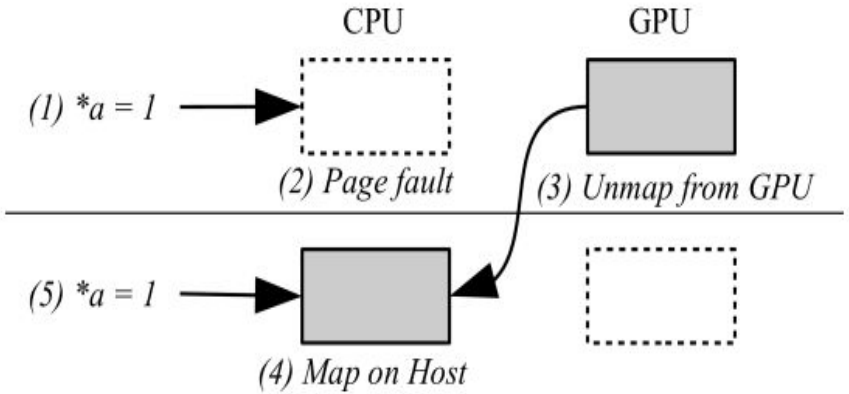
```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

# The Simplified Memory Management:

- cudaMallocManaged() is a allocator in CUDA6.
- It allocates data on a managed memory pool in unified memory that is migratable between GPU and CPU.
- With this, we can do data initialization in CPU and computation on GPU. Thus we migrate data from CPU to GPU.
- So there is single pointer to data accessible anywhere.
- Pages in virtual address space in an application process may be mapped to physical pages either on GPU or CPU memory.

# Page Faults and Data migration

- When a device accesses a virtual page that is not mapped to a physical page on the device memory, a page fault occurs.
- The CUDA runtime solves this be remapping the page to a physical page on the device memory and copying the data.
- This process is called on-demand paging.



# Optimize Data Migration : Using Prefetch

- Prefetch results in asynchronous data migration to GPU or CPU before the data is accessed.
- It reduces the page faults and overhead in handling page faults.
- cudaMemPrefetchAsync() : is a asynchronous page prefetching mechanism in order to perform data migration.

# Prefetch for eliminating page faults

```
cudaMalloc(&d_data, N);
cpu_func1(data, N);
cudaMemcpy(d_data, data, N, ...)
gpu_func2<<<...>>>(d_data, N);
cudaMemcpy(data, d_data, N, ...)
cudaFree(d_data);
cpu_func3(data, N);

free(data);
```

```
cudaMallocManaged(&data, N);
cpu_func1(data, N);
cudaMemPrefetchAsync(data, N, GPU)
gpu_func2<<<...>>>(data, N);
cudaMemPrefetchAsync(data, N, CPU)
cudaDeviceSynchronize();
cpu_func3(data, N);

free(data);
```

# Single Instruction Multiple Thread: SIMT

**Consider a kernel Code:**

```
if(a[id] % 2==0)

a[id] = a[id]+1

else

a[id] = a[id] + 2
```

**What is the problem here?**

# THANK YOU