High Performance Scientific Computing

SOMNATH ROY
MECHANICAL ENGINEERING DEPARTMENT, IIT KHARAGPUR

Module : GPU Computing

Lecture : Introduction to GPGPU and CUDA

Coprocessors and accelerators

The idea is to improve the performance of a computer CPU by augmenting the processing units

Coprocessors are the auxiliary processing units which supplement the processing power of a microprocessor. A co-processor is an extension of the processor hardware, It is connected to the internals of the host processor, which then passes it instructions to execute.

Intel Xeon-Phi (max 72 cores) has been used as coprocessor in scientific computing (2010-2020)

Accelerators provide similar functionality, however, they are not extensions of computer architecture. Rather, accelerators are independent I/O device, connected through programmable interface to the main CPU motherboard with memory mapping provisions.

Graphics processing Unit (GPU)-s (NVIDIA, AMD etc.) are accelerators.

Accelerated computing gives high speed-up

Grid Size	A single-core CPU	Xeon Phi 31SP	K20c GPU
960*240	1.00	10.40	10.83
1920*240	1.00	11.45	12.24
3840*960	1.00	12.27	12.35

~10 times speed up using Xeon Phi or Kepler GPU for WENO based CFD problem

Deng et al.(2015). Kepler GPU vs. Xeon Phi: Performance case study with a high-order CFD application. In 2015 IEEE international conference on computer and communications (ICCC) (pp. 87-94). IEEE

Platform	Time (s)	Speedup
CPU (Baseline)	282.447	-
K20 w/ ECC	8.599	~ 32 x
K20 w/o ECC	6.826	~ 41 x
K40 w/ ECC	7.062	~ 40 x
K40 w/o ECC	5.8	~ 49 x
P100	2.4923	~ 113 x

~100 times speed up for large mesh using Pascal GPUs for simulation of flow over turbine blades

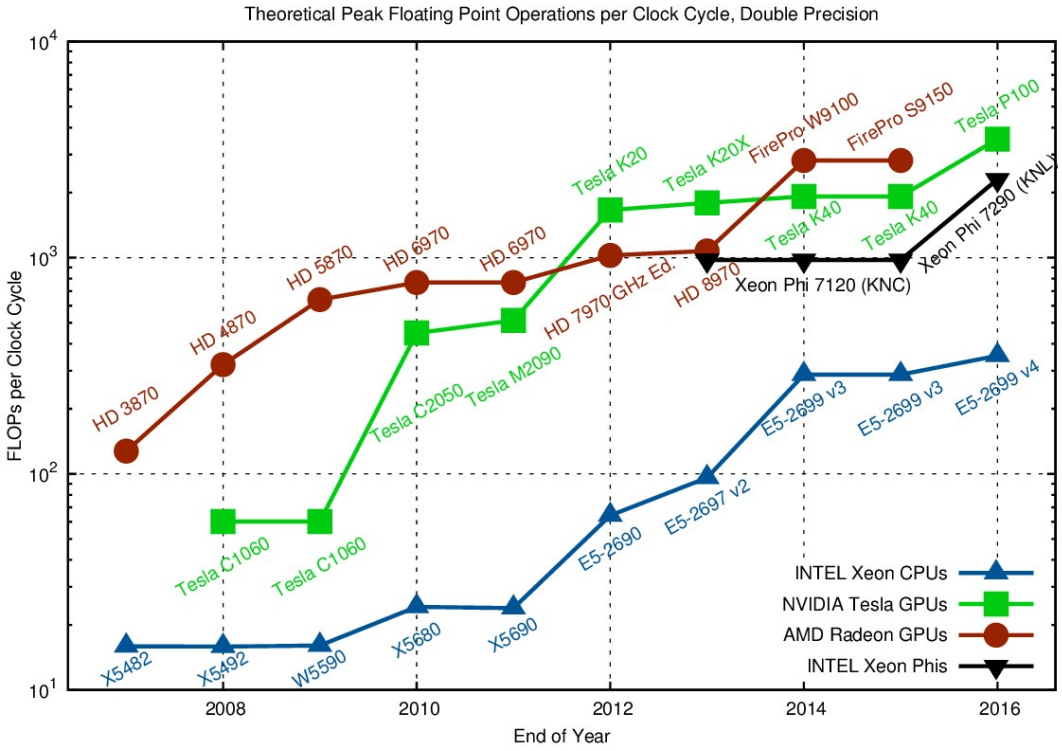
Maruthi, N. H., Ranjan, R., Narasimha, R., Deshpande, S. M., Sharma, B., & Nanditale, S. R. T. (2017). GPU acceleration of a DNS code for gas turbine blade simulations. In Computational science symposium, IISc, Bangalore.

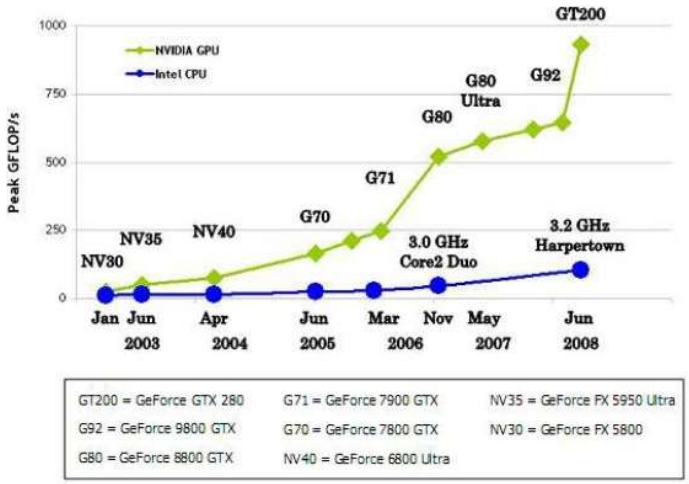
Newer versions of GPU-s are providing better speed-ups

Graphics Processing Unit – Computing Power

Graphical Processing Unit (GPU), initially designed for game development
Around 2000, general purpose GPU-s or GPGPU-s developed

Efficient in matrix calculation



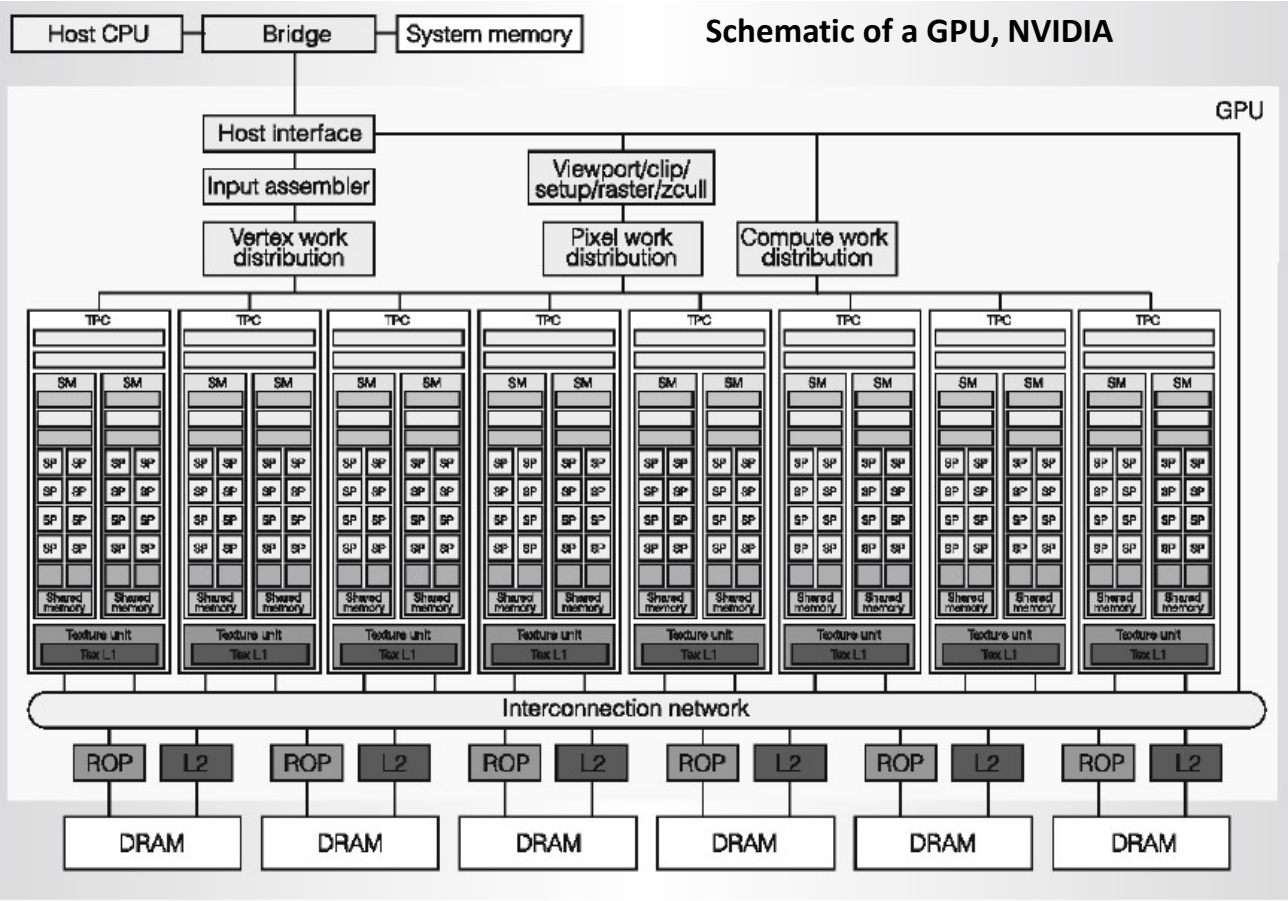


Hardware	Memory Bandwidth (GB/s)	Single Precision TFLOPS	Double Precision TFLOPS
K40	288	4.29	1.4
Titan X	480	11	0.34
P100	720	10.6	5.3
V100	900	15.7	7.8

Source:
https://www.stoneridgetechnology.com/company/blog/charged-up-on-volta

GPU -vs- CPU performance,
https://www.karlrupp.net/

GPU Hardware



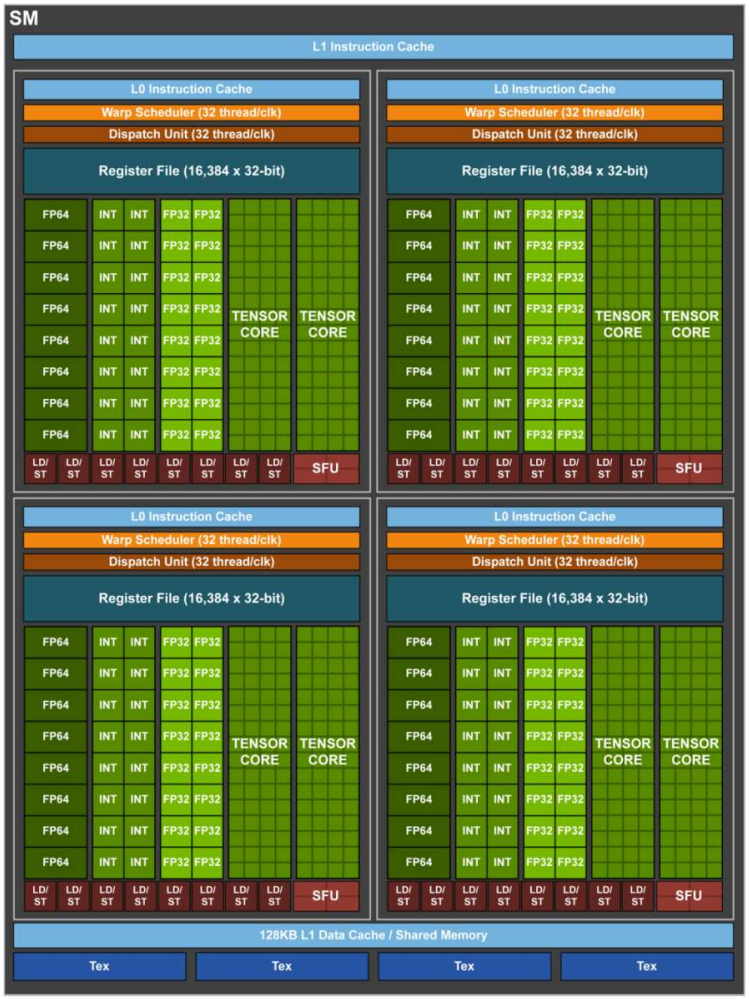
Schematic of a GPU, NVIDIA

TPC- Texture Processing Clusters
SM- Streaming Multiprocessor
SP- Streaming Processor (GPU core)
DRAM- Device RAM (GPU RAM)
ROP- Render Output unit

GPU Hardware (specs)

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



Source:
V100 GPU white paper, NVIDIA

GPGPU

Ordinarily GPU-s are designed for graphics rendering. General purpose graphics processing units (GPGPU-s) are modified to perform the applications traditionally handled by CPU-s (like floating point calculations).

Essentially, all moderns GPU-s are GPGPU-s

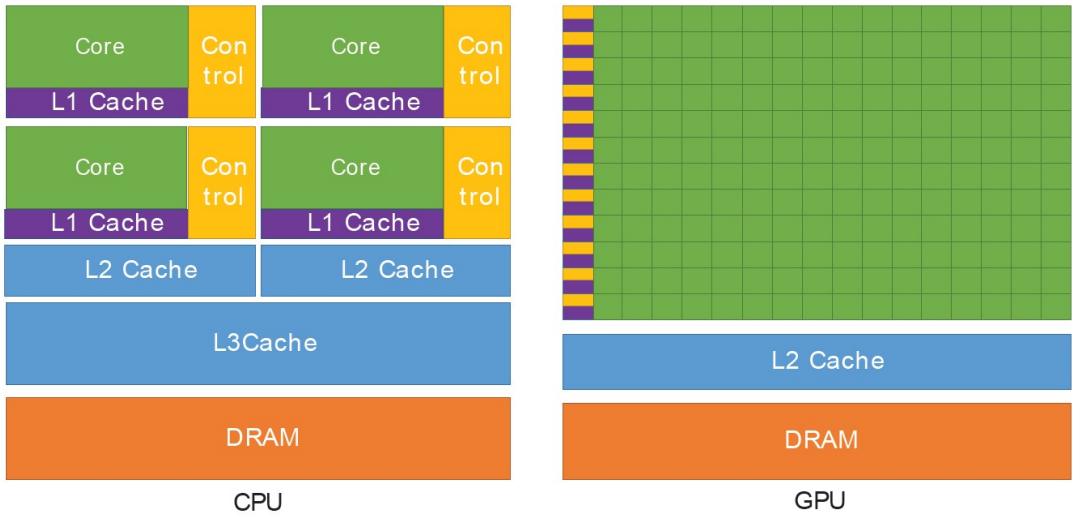
GPGPUs can be programmed to deploy the processing power toward addressing scientific computing needs as well.

GPGPU-s are connected to a CPU, which offloads compute-heavy jobs in terms of concurrent instruction streams to the GPU-s

GPU-s are in market for nearly 50 years but around 2001, floating point support and programmable shaders are made available for graphics cards and GPGPU-s became practical and popular

Matrix multiplication routines made it extendable for HPC-s

GPGPU – architecture and memory



GPU-vs-CPU: Memories

Source:

NVIDIA CUDA C Programming Guide

GPU-s more transistors for data processing than control or caches
L1 cache is small (absent in earlier versions)
A more detailed discussion on GPU memories will be presented while discussing CUDA

Compute capability

Compute capability is a feature set of NVIDIA GPU device. Advanced versions of GPU-s give higher compute capability (compute x.y)

The compute capability of a device is represented by a version number, also sometimes called its "SM version". This version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.

Compute 1.0 was the oldest version for 8000 series cards

Compute 1.1 was for 9000 series card, which allowed data transfer and kernel execution overlap

Compute 2.0 was for Fermi series. It introduced 16 K-48 K L1 cache, shared L2 cache for all SM-s etc

Higher versions of compute capability gives more flexibility for programmers and higher efficiency in implementation

Compute capability- advanced features

Feature Support	Compute Capability				
(Unlisted features are supported for all compute capabilities)	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.x	8.0
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	Yes				
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	Yes				
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)	Yes				
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	Yes				
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())	Yes				

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	255[1]
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/ 48 KB	96 KB	64 KB	Configurable up to 96 KB

[1] The per-thread program counter (PC) that forms part of the improved SIMT model typically requires two of the register slots per thread.

Feature Support	Compute Capability				
(Unlisted features are supported for all compute capabilities)	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.x	8.0
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No		Yes		
Warp vote functions (Warp Vote Functions)	Yes				
Memory fence functions (Memory Fence Functions)					
Synchronization functions (Synchronization Functions)					
Surface functions (Surface Functions)					
Unified Memory Programming (Unified Memory Programming)					
Dynamic Parallelism (CUDA Dynamic Parallelism)					
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No	Yes			
Tensor Cores	No			Yes	
Mixed Precision Warp-Matrix Functions (Warp matrix functions)	No			Yes	
Hardware-accelerated async-copy (Asynchronously Copy Data from Global to Shared Memory)	No				Yes
Hardware-accelerated Split Arrive/Wait Barrier (Split Arrive/Wait Barrier)	No				Yes
L2 Cache Residency Management (Device Memory L2 Access Management)	No				Yes

GPU-s are evolving in terms of hardware and programming flexibility

Source:
V100 GPU white paper, NVIDIA CUDA programming guide

GPU vs CPU

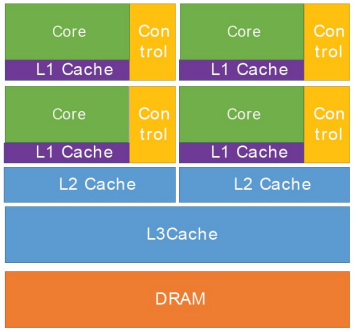
GPUs contain thousands of arithmetic units. Their power can be used to accelerate the program

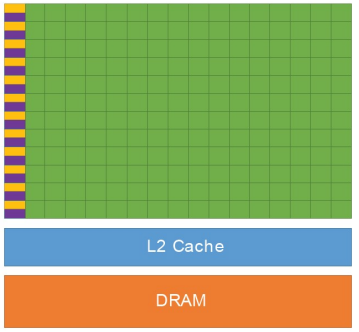
Most of the transistors in CPU are dedicated for data caching and controlling. However, in GPU these are mostly dedicated for calculations

Typically GPU-s have small cache but large number of computing cores. Each SM-s schedule threads on these cores

Large number of registers are available in GPU-s for context switching

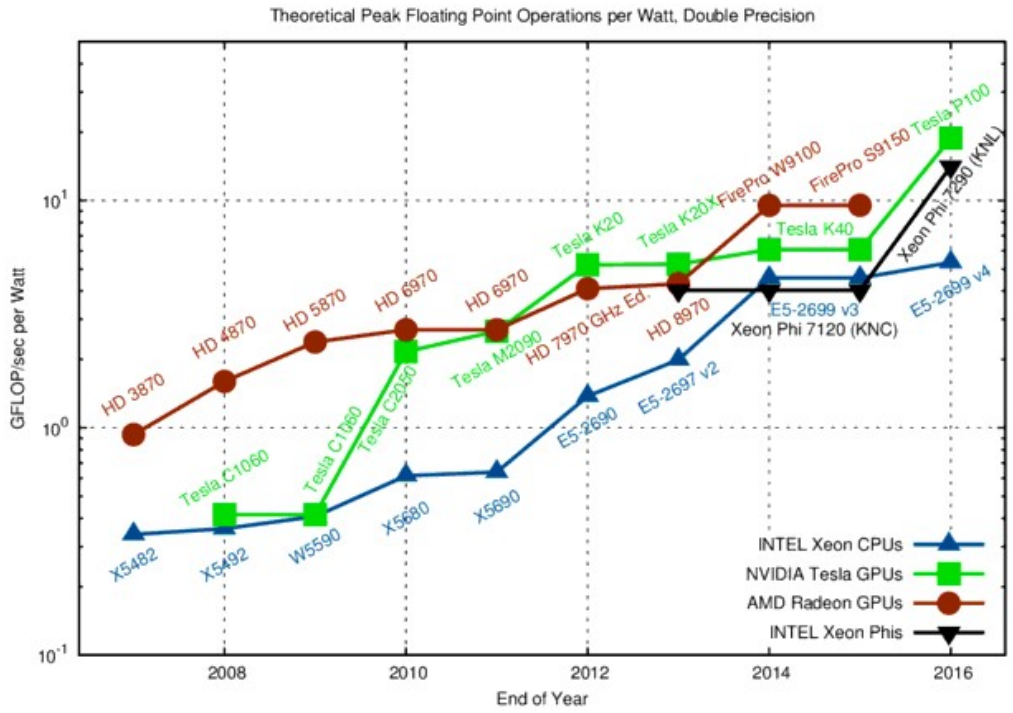
Newer GPU versions show more flexibility in terms of memory usage than older one. Earlier versions of GPU-s did not have an L1 cache (Geforce 2000 seris). However Pascal or Volta has cache upto last levels





Source:
NVIDA CUDA C Programming guide

GPUs give more efficient computing in terms of energy



GPU-s itself cost heavy power, but due to less space and less number of components, GPU-s consume much less power than CPU-s of equivalent computing power

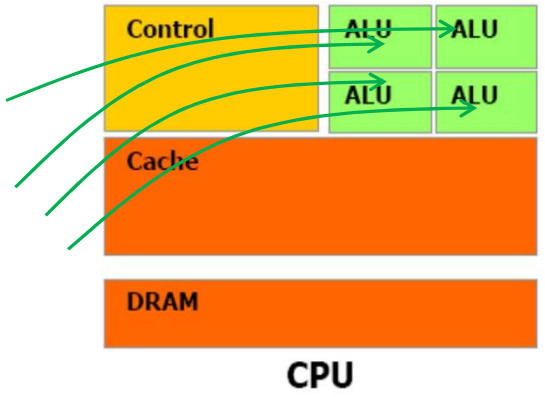
Source:
https://www.karlrupp.net/

GPU vs CPU: programming aspects

Limited number of complex tasks can go to CPU

GPU-s run a large number of simple tasks

Programmer cannot directly run a job in a GPU, jobs are launched from the host CPU in the device GPU

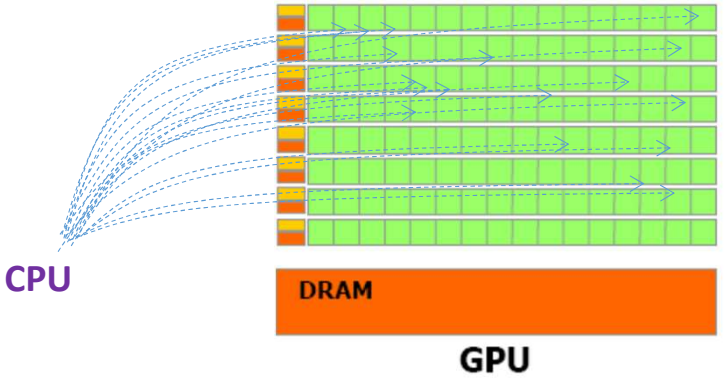


GPU jobs require more granularity and less task dependency

Memory access in GPU-s is different than CPU-s due to small (or absent) cache and small off-chip memory

Programmer needs to give special attention to on-chip memories while GPU programming

GPU threads are scheduled at the SM-s, while large register size is used for latency hiding. So, a huge number of threads can be launched.

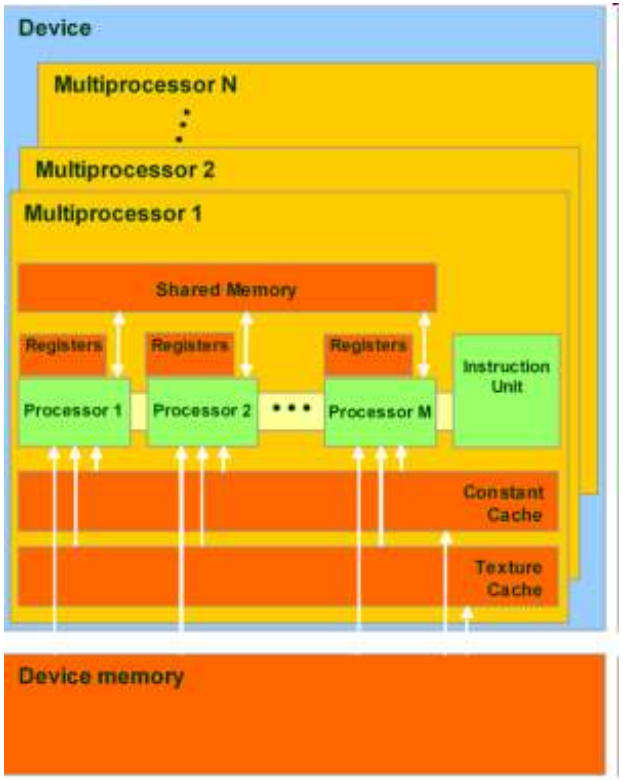


GPU programming

Single Instruction Multiple Data (SIMD)

A number of programming languages are available for GPU programming:

OpneGL- designed for graphics rendering
OpneCL
DirectX- Microsoft
CUDA- NVIDIA
Firestream- AMD
OpenACC
OpenMP 4.0+



CUDA provides optimized performance for NVIDIA GPU-s and also easily extendable to Fortran and Python

Unified architecture in GPU-s

Early shader abstractions (such as Shader Model 1.x) used very different instruction sets for vertex and pixel shaders, with vertex shaders having much more flexible instruction set.

Unified shader models were developed by early 2005-06 (Shader Model 4.0+). This model provided a hardware design by which all shader processing units of a piece of graphics hardware are capable of handling any type of shading tasks.

Most often Unified Shading Architecture hardware is composed of an array of computing units and some form of dynamic scheduling/load balancing system that ensures that all of the computational units are kept working as often as possible.

Unified shader architecture allows more flexible use of the graphics rendering hardware.

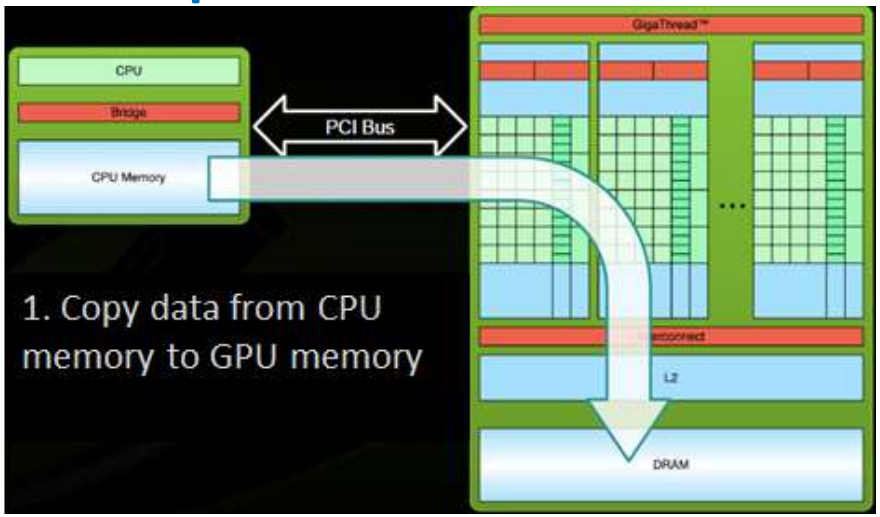
This unified architecture helps GPGPU-s to handle number crunching jobs.
For details: http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf

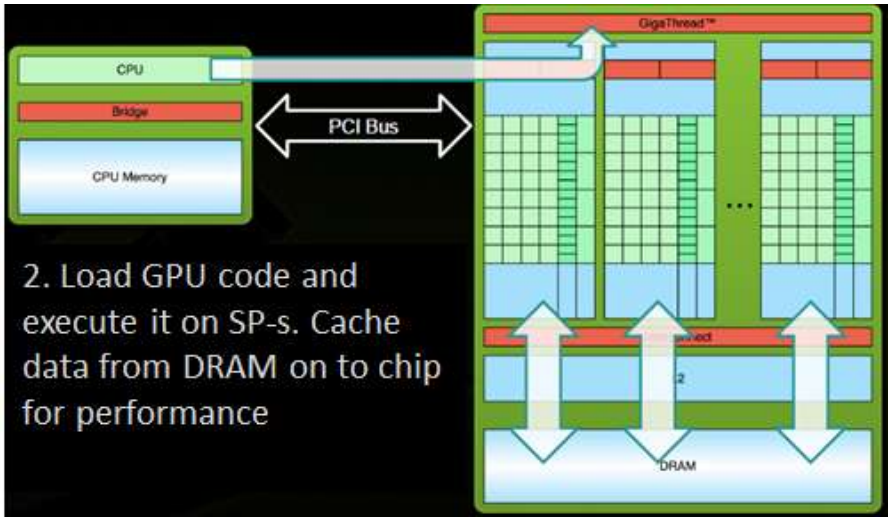
CUDA was initially named as Compute Unified Device Architecture

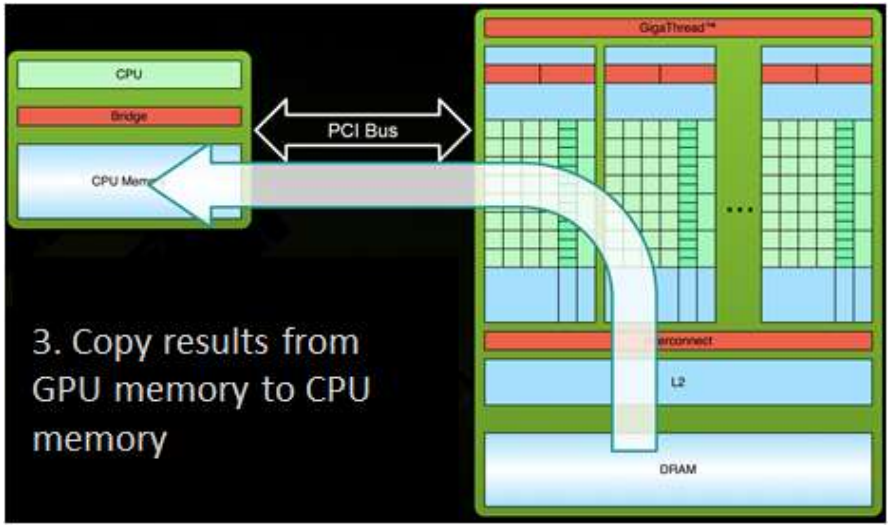
Some terminology

- ✓Device - GPU
- ✓Host – Hosting CPU
- ✓Kernel – Part of the code that runs in GPU
- ✓Global memory - data available to all threads and can be copied directly to host
- ✓DRAM- Device or GPU RAM
- ✓SRAM – on chip shared RAM

CUDA- process flow







Source:
CUDA C/C++ basics, Zeller, NVIDIA

CUDA- programming model

Integrated host+device app C program

- Serial or modestly parallel parts in host C code
- Highly parallel parts in device SIMD kernel C code
- Supported for other languages like Fortran, Python, etc.

- Kernels are functions called from the CPU code and executed in the GPU
- Kernels return void

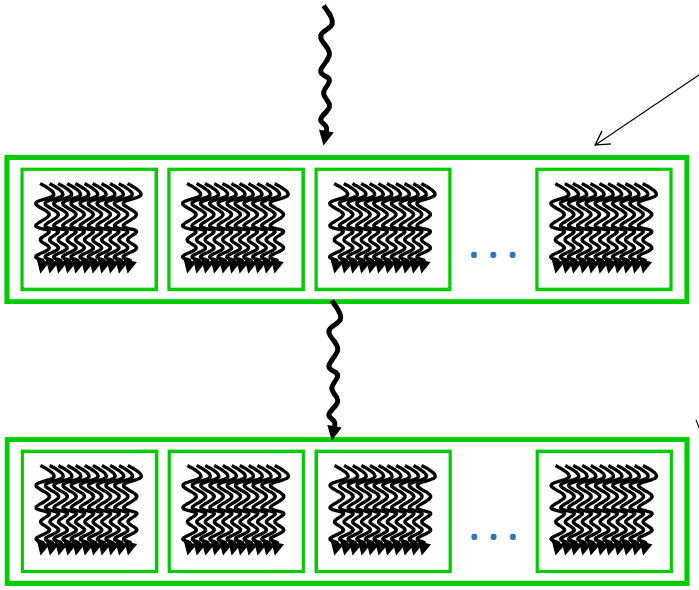
CUDA launches threads in blocks and grids

Serial Code (host/CPU)

CUDA Kernel (device/GPU)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host/CPU)

Parallel Kernel (device/GPU)
KernelB<<< nBlk, nTid >>>(args);



Threads and blocks

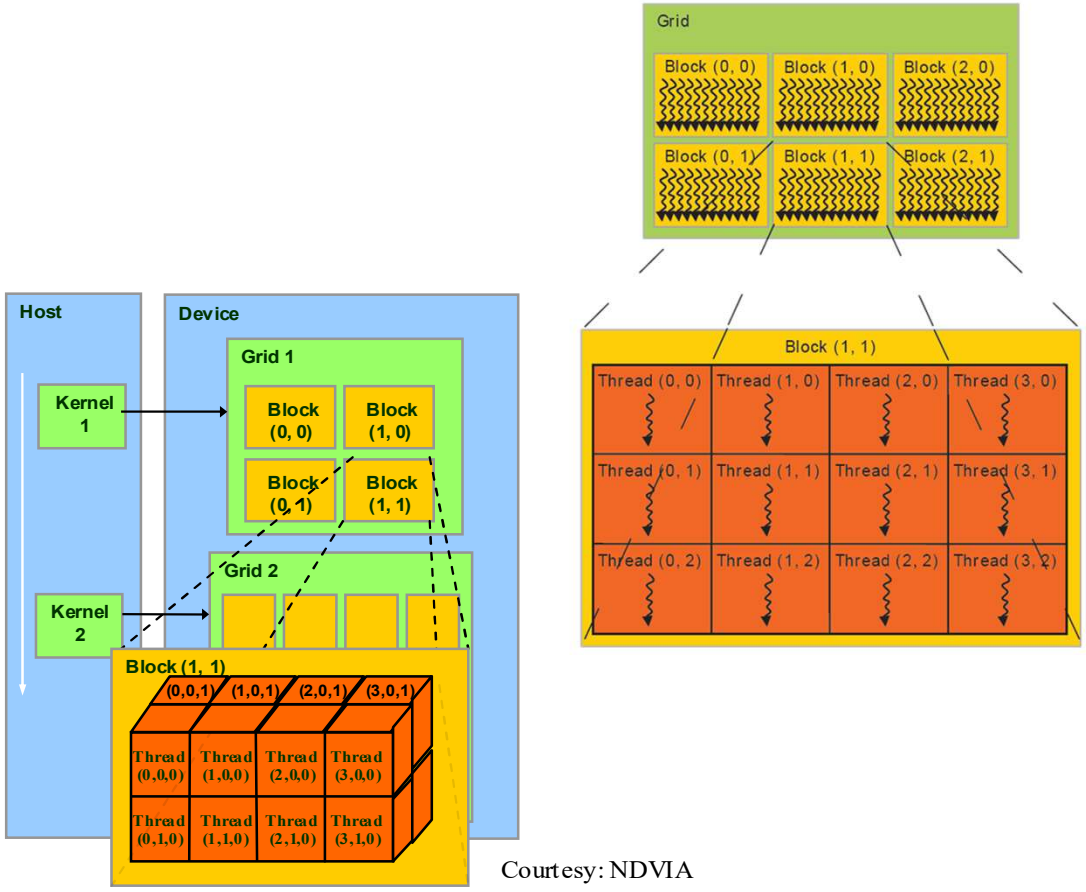
Each thread id determines what data to be worked on.

Threads belong to blocks. Kernel is launched as a grid, which are collection of blocks.

Blocks and grids follow Cartesian structures

This definition of threads facilitate image processing and matrix computing (array handling)

The thread id inside a particular block is mapped to the matrix datastructure in CUDA program– thread algebra



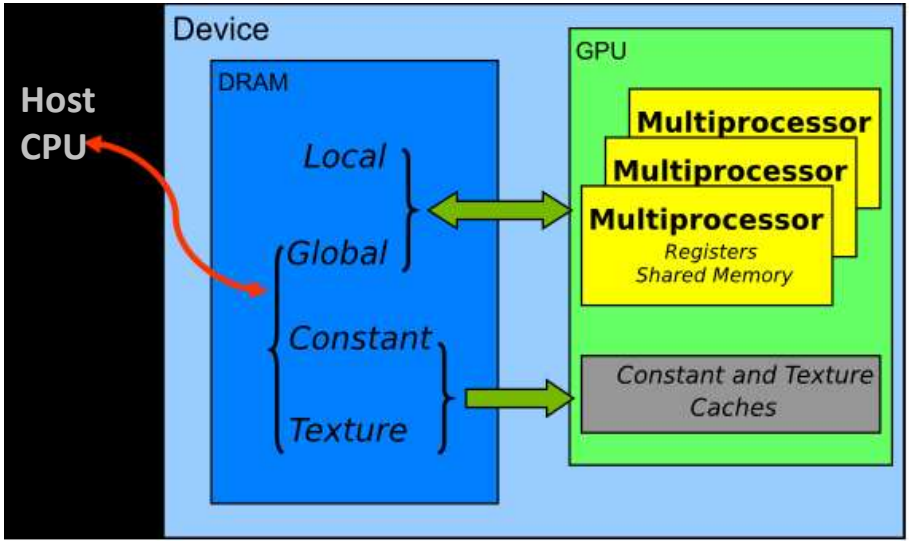
Courtesy: NDVIA

CUDA- memory architectue

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

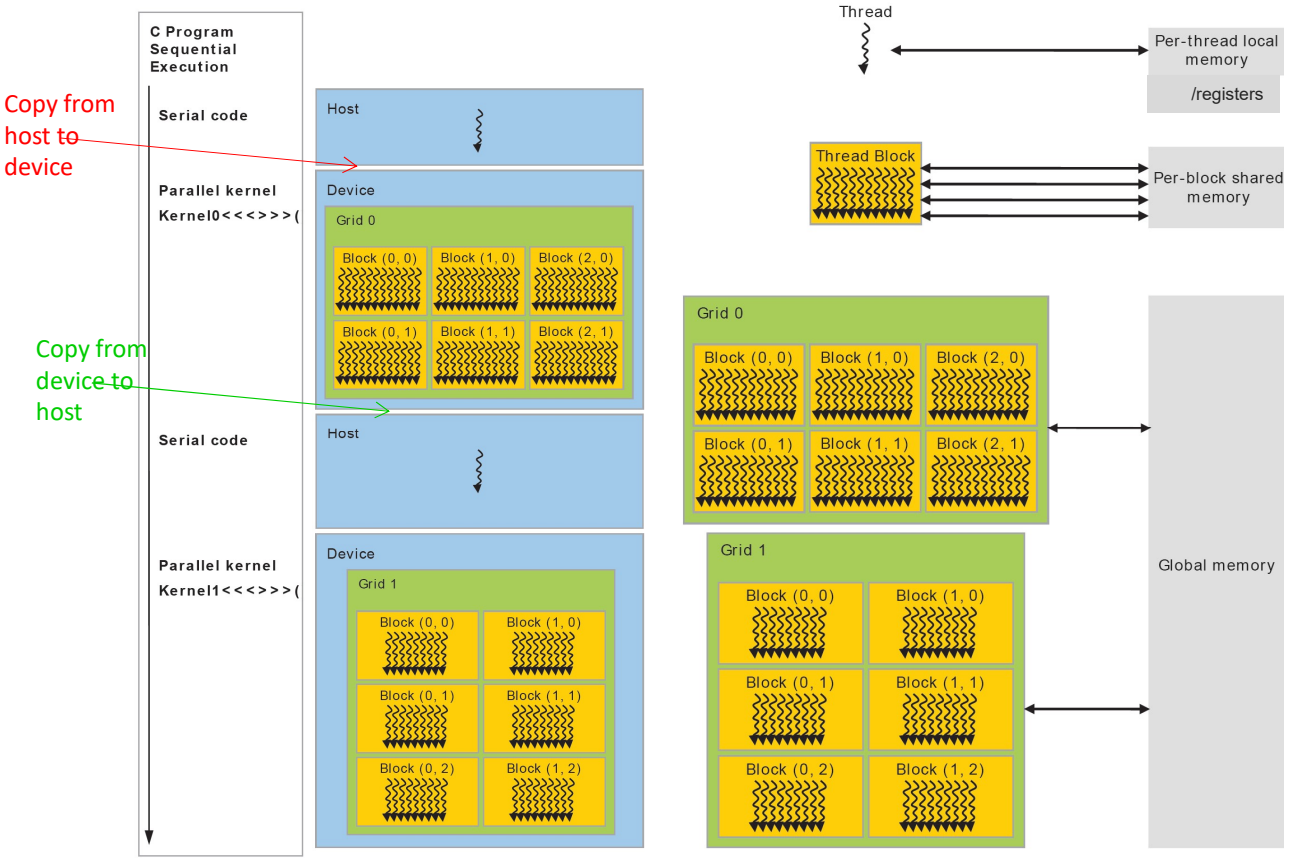
†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.



Performance of a CUDA/GPU program depends on efficient use of memory architecture

Source: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

Thread and memory hierarchy in a heterogeneous CUDA program

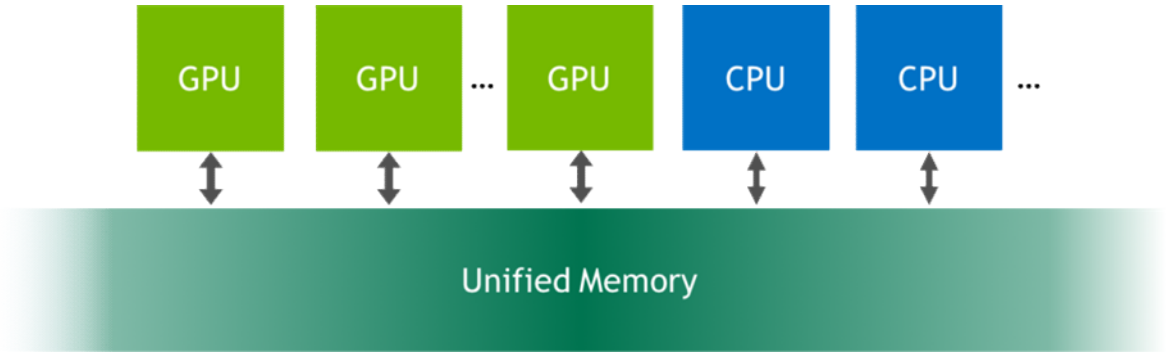


In heterogeneous program, serial part of the code run as a simple C instruction in CPU and massively parallel part are executed as CUDA kernels in GPU

CPU and GPU has their own memory, with CUDA enabling copying of memory from host to device or vice-versa

Source:
NVIDA CUDA C Programming guide

Unified memory programming in GPU



Source:
https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

GPU-s of class Kepler and above with CUDA version 6.0+ (SM Architecture 3.0+) facilitate unified memory programming

Here, GPU programming is simplified by unifying memory spaces coherently across all GPUs and CPUs in the system and by providing tighter and more straightforward language integration for CUDA programmers.

Data access speed is maximized by transparently migrating data towards the processor using it
Facilitates multi-GPU programming

High Performance Scientific Computing

SOMNATH ROY
MECHANICAL ENGINEERING DEPARTMENT, IIT KHARAGPUR

Module : GPU Computing

Lecture : Matrix multiplications in CUDA

Blocks and threads

Threads are organized in blocks of the grid.

Kernel launches all threads in a grid following an SIMT model.

The kernel function is executed as gridsize x blocksize number of times in that many threads

Blocksize and gridsize information is available in the execution configuration myKernel<<<gridsize, blocksize>>>(...)

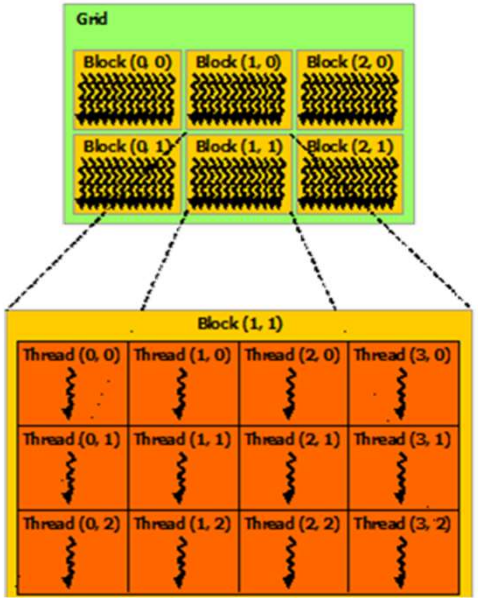
The variables gridsize and blocksize are of type dim3 and can have three-dimenstions

Block dimension (number of threads in each dimension in a box) is given as: blockDimx.x, blockDimx.y, blockDimx.z

id of a given block in the grid is given as: blockIdx.x, blockIdx.y & blockidx.z

Local thread id (within a block) is given as threadIdx.x, threadIdx.y & threadIdx.z

It might be important to find global thread id for some problems



Source:
NVIDIA CUDA programming guide

Blocks and threads (cont.)

In the example problem of matrix addition, 2D thread id-s are used as pointers to matrix elements in the kernel function

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

x id of the thread points to matrix row

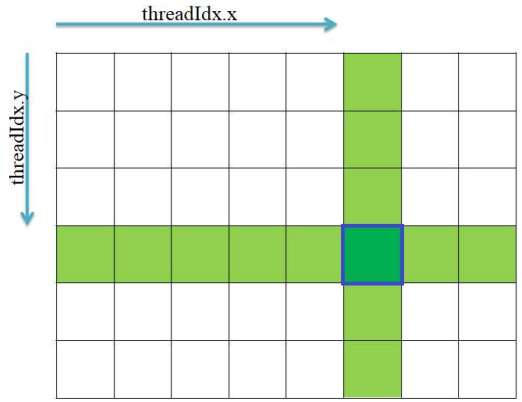
y id of the thread points to matrix column

thread id is used to point to the memory location which will be accessed by the particular thread

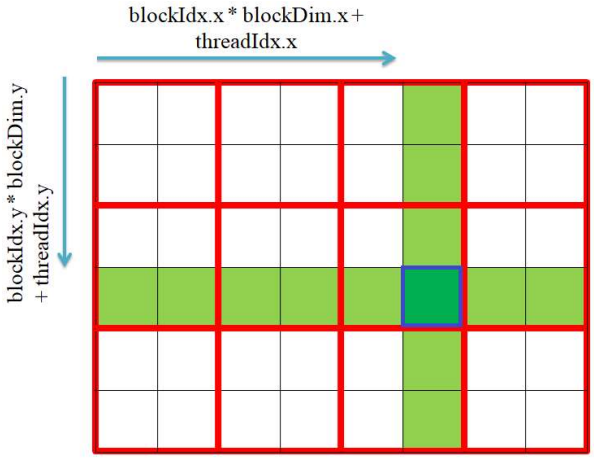
A single block is launched in the grid. So, local thread id is same as the global thread id and points to the location of the matrix column and row

Pointing matrix memory location

If the gridsize is (1,1,1) [single block]



For a general two-dimensional grid:



Global id of the thread is obtained from the local thread id, local block id and blocksize.

Courtesy: HBeonGpgpu
https://pccsgpgpu.webs.com

Matrix-vector multiplication using CUDA

Serial subroutine:

```
void matvec (int *d, int r[n],int k[n], int n)
{
  int i,j;
  for(i=0;i<n;i++)
  for(j=0;j<n;j++)
  k[i]=k[i]+(d[n*i+j]*r[j]);
}
```

Put the matrix in a one-d array for coalesced access

For a 1280X1280 matrix, serial execution in Xeon E5-2620 v4 CPU takes 0.447904 s
P100 GPU (12 GB) takes 0.384960 s

CUDA kernel:

```
__global__ void mvm (int *a, int *x, int *b, int n,int u)
{
  int j=blockIdx.y * blockDim.y + threadIdx.y;
  int i=blockIdx.x * blockDim.x + threadIdx.x;
  int ind= i+gridDim.x*u*j;
  if(ind<n)
  {
    int l;int m=(ind*n);
    *(b+ind)=0;
    for(l=0;l<n;l++)
    *(b+ind)=(*(b+ind))+(*(a+m+l))*(*(x+l));
  }
__syncthreads();
}
```

blocksize = uxu

A coalesced global memory access pattern is followed

Courtesy: Mr. Dilip Subbaian G

Matrix-matrix product

CPU function

```
int matmul(int a[n][n], int b[n][n],int c[n][n], int n)
{ int sum=0;int i,j;int k=0;
  for(int i=0;i<n;++i)
  {sum=0;
    for(int j=0;j<n;++j)
    {
      for(int h=0;h<n;h++)
      {
        sum=sum+a[i][h]*b[h][j];
      }
      c[i][j]=sum;
    }
  }
}
```

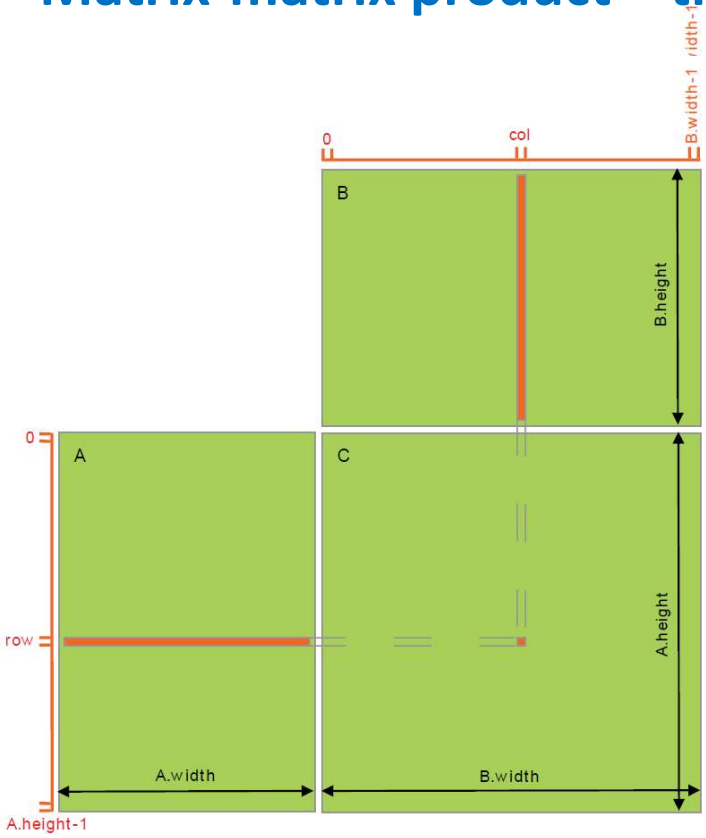
Observations:

1. **Large number of memory access**
2. **Cache unfriendly access of bd**
3. **Shared memory and tiling can help**

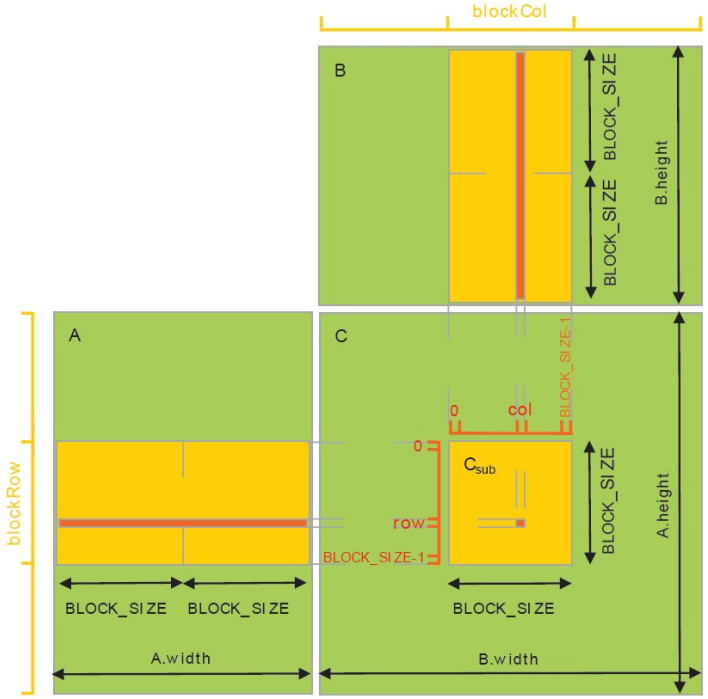
CUDA kernel

```
__global__ void paramul(int *ad, int *bd, int *cd,int n)
{
  int k
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  int idy=blockIdx.y*blockDim.y+threadIdx.y;
  if(idx<n && idy<n)
  {
    for(k=0;k<n;k++)
    (*(cd+n*idx+idy))=(*(cd+n*idx+idy))+(*(ad+n*idx+k))*(*(bd+n*k+idy));
  }
}
```

Matrix-matrix product – tiling and shared memory usage



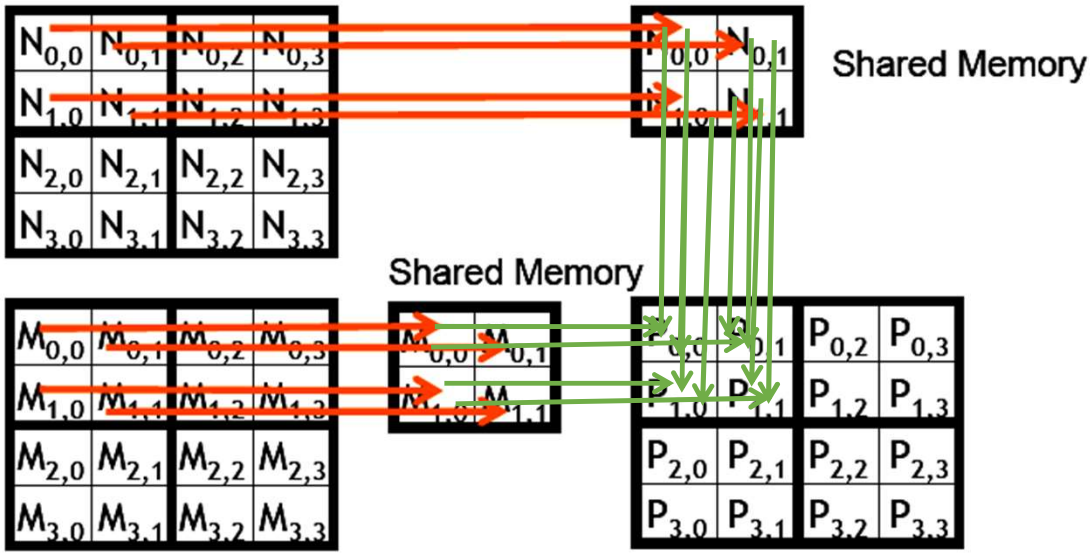
simple matrix multiplication



matrix multiplication with tiling tile size matches with block size and fits shared memory

Source: NVIDA CUDA Programming Guide

Matrix-matrix product – tiling and shared memory (cont)



Source: GPU teaching kit, NVIDIA-UIUC

Matrix-matrix product – tiling and shared memory optimized

```
__global__ void paramul(int *ad, int *bd, int *cd,int n)
{
  int k;int tile=16;int m;
  int idx=blockIdx.x*blockDim.x+threadIdx.x;
  int idy=blockIdx.y*blockDim.y+threadIdx.y;
 int bx=blockIdx.x;int by=blockIdx.y;
  int tx=threadIdx.x; int ty=threadIdx.y;
  int row = by *blockDim.y + ty;
 int col = bx *blockDim.x + tx;
 __shared__ int as[16][16];
 __shared__ int bs[16][16];
  if(idx<n && idy<n)
  {
    for(m=0;m<n/tile;++m)
    {as[ty][tx]=*(ad+(row*n+(m*tile)+tx));
    bs[ty][tx]=*(bd+((m*tile+ty)*n+col));
    __syncthreads();
    for (k=0;k<tile;++k)
    cd[row*n + col]+=as[ty][k]*bs[k][tx];
  }
}
__syncthreads();
}
```

Non coalesced [2d array] used for shared memory as the memory size is small, bandwidth is high

For 1280 x 1280 matrix
Sequential code takes 214.9249 sec
Optimized CUDA code takes 6.3870 sec

Matrix solvers- performance

In Conjugate gradient or Biconjugate gradient matrix solvers, matrix-vector products are expensive and they are parallelized by calling CUDA kernels.
The performance results (using P100) are:

size	CUDA	serial
5000	0.79	19.95
10000	1.96	134.91
20000	7.17	668.8
30000	29.08	1844.44
40000	45.77	4259.42

Conjugate gradient solver for symmetric matrix, size as no. of rows, time in seconds

size	BiCGstab serial	Jacobi serial	BiCGstab CUDA	Jacobi CUDA
144	0.021	0.029	0.019	0.066
3310	27.91	792.74	0.62	18.01
7705	125.4	9482.46	4.25	187.24

Different matrix solvers for non-symmetric unstructured matrix, size as no. of rows, time in seconds

GPU codes give better speed-up for larger matrices

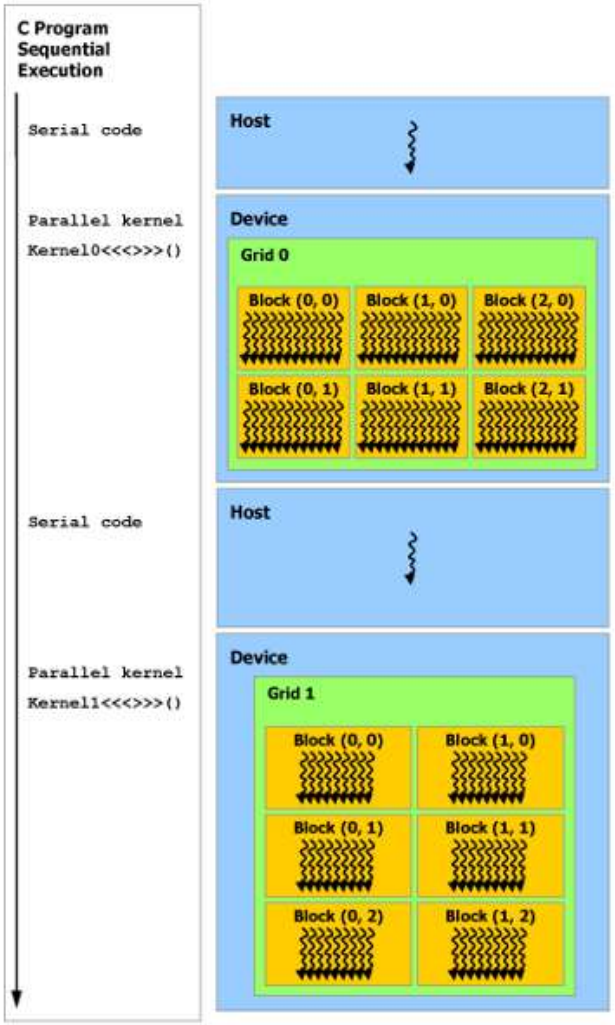
High Performance Scientific Computing

SOMNATH ROY
MECHANICAL ENGINEERING DEPARTMENT, IIT KHARAGPUR

Module : GPU Computing

Lecture : Thread execution in CUDA program - scheduling and memory access

CUDA Kernel – large number of threads



A kernel is a function which is executed on the GPU. It is executed as an array of threads.

A kernel launches a grid which has number of blocks of threads

CUDA can launch a large number of threads.

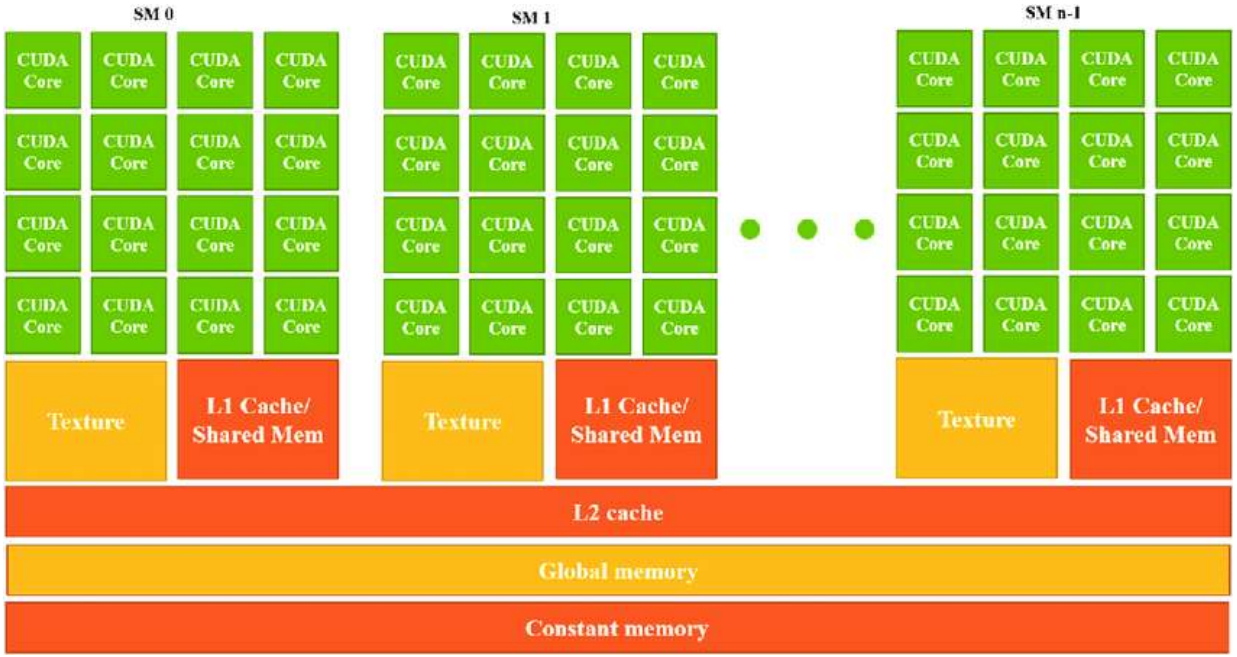
Maximum number of blocks in x-dimension in a grid is 2^{31}-1

Each block can have 1024 threads max.

So, a large number of concurrent threads are launched through a kernel. This number is much higher than available number of cores.

Scalability and scheduling are important

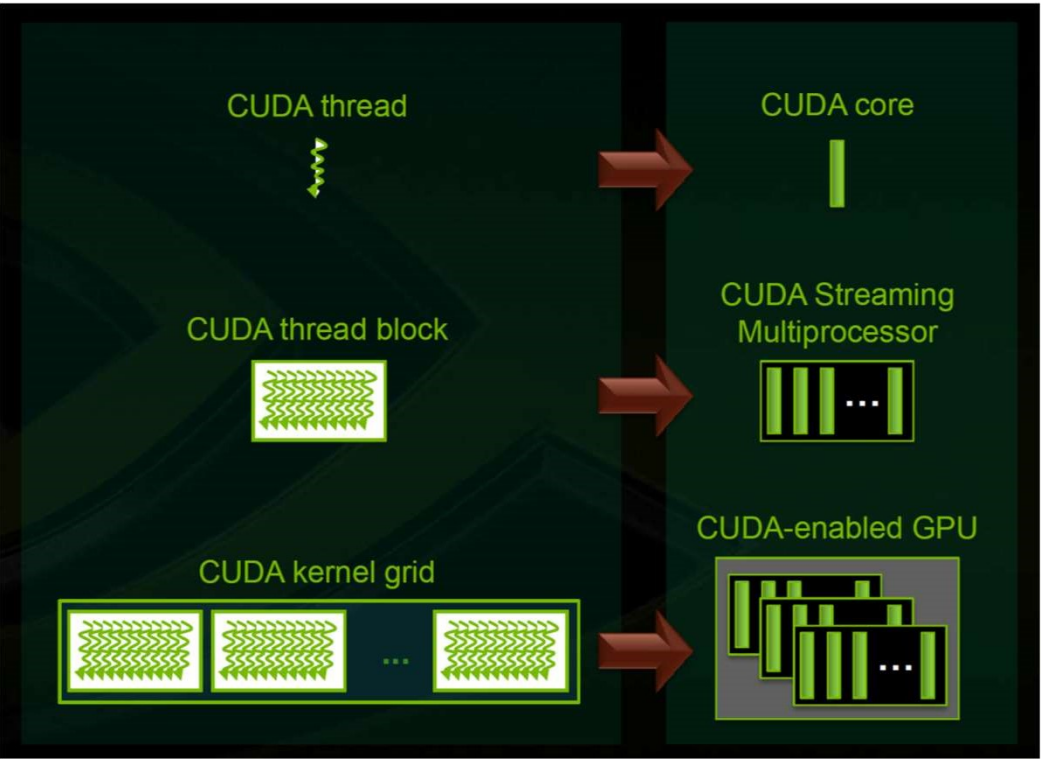
Revisiting GPU architecture



Source: GPU Acceleration of Optical Measurement, Wang and Kimao

GPU card has a number of streaming multiprocessors or SM-s (80 for V100) and each SM has a number of processors or computing cores (32 dual precision core in a SM for V-100s)

Threads in a GPU architecture



Source: NVIDIA

A GPU thread runs in a CUDA core

A thread block runs within one SM

- **There can be more threads in a block than the number of cores in an SM**
- **There can be more than one block assigned to one SM**
- **Scheduling of threads in an SM is important- latency hiding and scalability**
- **Memory requirement of the block and memory of the SM are important too**

Grid is launched for the whole GPU by the kernel

Concurrent kernels may run in one GPU

Executing the threads within a block- Warps

Warps are the basic units of execution on the GPU. In an SM a group of threads are executed together and this group is known as Warp. Number of threads in an Warp is fixed as per GPU architecture

- ***Warp***: **a group of threads executed *physically* in parallel in any GPU, basic unit of scheduling hardware-**

Hardware limits

The Warp size is fixed as 32 in modern GPU-s

- ***Block***: **a group of threads that are executed together and form the unit of resource assignment –**

Programming specification

Block size can be specified by the programmer

Groups of threads within a same block are launched as sets of warps in a particular SM

Executing the threads within a block- Warps (cont.)

The SM creates, manages, schedules and executes threads at warp-level granularity.

- ✓ **Each warp consists of 32 threads of contiguous thread Ids.**
- ✓ **All threads in a warp execute the same instruction.**
- ✓ **At this level essentially a vector-processor level architecture works.**
- ✓ **If the threads of a warp diverge in execution path, each branch is executed serially adding latency.**
- ✓ **When a warp executes an instruction that accesses global memory, in ideal case, it coalesces the memory accesses of the threads within the warp into as few transactions as possible**

Scheduling the warps within a block

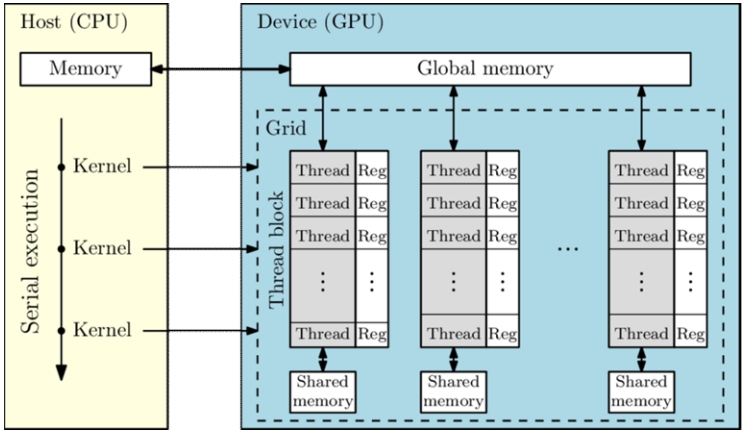
Once a thread block is launched on a multiprocessor (SM), all of its warps are resident until their execution finishes.

Within the block, warps are scheduled for best performance.

Consider a warp of 32 threads executing an instruction. If some of the operands are not ready and fetching memory from DRAM, the SM schedules next warp for execution. A process called 'context switching' takes place which transfers control to another warp.
When switching away from a particular warp, all the data of that warp remains in the register file so that it can be quickly resumed when its operands become ready.

Through this process GPU-s hide the thread level latency and show improved scalability

GPU-s have large number of register files for context switching and thus scalable scheduling.



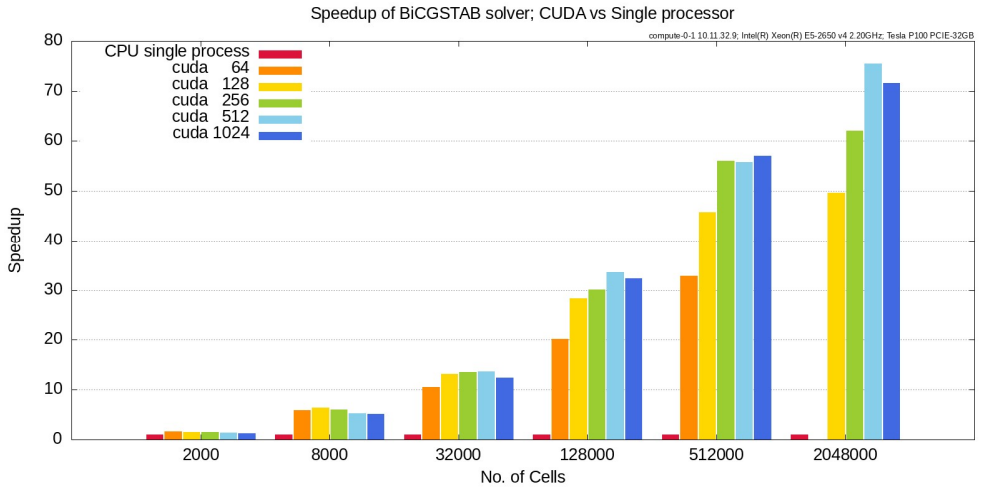
Source:
Implementation of a Fully-Parallel Turbo Decoder on a General-Purpose Graphics Processing Unit , Li et al, 2016

Wikipedia

Important programming aspects on threads and blocks

1. Number of threads in a block (blocksize) should be multiple of the warp size for right scalability

Optimum blocksize depends on the memory usage of the program



2. Thread divergence to be avoided

Threads within a block must not take different execution path.

Thread divergence

Threads in a particular warp must follow same instruction following an SIMD model. In case the instruction sets are different (through if-else statements), they are executed serially. This adds heavy latency and should be avoided

Example with divergence:

```
{
  i = int threadIdx.x;
  if(i%2 == 0){
     x[i] = 5;
  }else{
     x[i]= 10;
  }
}
```

- **This creates two different control paths for threads in a block**
- **Threads 0 and 1 follow different path than the rest of the threads in the warp, but all the threads will reach the two instructions. So, a sequentiality arises**
- **Thread divergence is performance killer**

To avoid divergence, use:

```
{
  i = int threadIdx.x;
     x[i] = (1-i%2)*5 + (i%2)*10;
}
```

Here, all threads in a warp essentially executes same instruction

Thread divergence can cause deadlocks

Consider this example

```
if (threadidx.x <16)
{
    myFunc_then();
    __syncthread();
}else if (threadidx >=16)
{
    myFunc_else();
    __syncthread();
}
```

Source: Virtual Wrokshop, Cornell Univ

- **Half of the threads in the warp will execute first instruction and then wait for other threads to finish till *syncthreads***
- **However, due to sequentiality, the next 16 threads cannot start the second task as the second execution path is not initiated as first 16 threads are not free**

- **Code stalls!**

Avoiding thread divergence in CUDA program

Sometimes it is essential that threads execute instruction following a logical operation (if-else)

Following can be the solution while avoiding thread divergence

1. **Diverge at warp level- different warps may take different execution path. This is not difficult as warp size is always 32. ---** *If (threadIdx.x / WARP_SIZE >= 2) { }*
2. **Diverge at block level, different instruction sets are created for different blocks. Use logical statement over block id**
3. **Launch different kernels with different execution instruction**

Thread synchronization

When a kernel function calls *syncthreads()*, all threads in a block are held at the calling location until all other threads in that block reaches the location. This is a barrier synchronization function.

Ensures that all threads in a block have completed a phase of their execution of the kernel before they all move on to the next phase.

This adds waiting time (and latency) but are important in many cases, especially when the subsequent executions have data dependency with the present execution.

The waiting time can be reduced by assigning execution resources of all threads in a block as a unit. Proximity of threads reduce the latency due to synchronization

Operations like cudaMemcpy, cudaMalloc do implicit synchronization

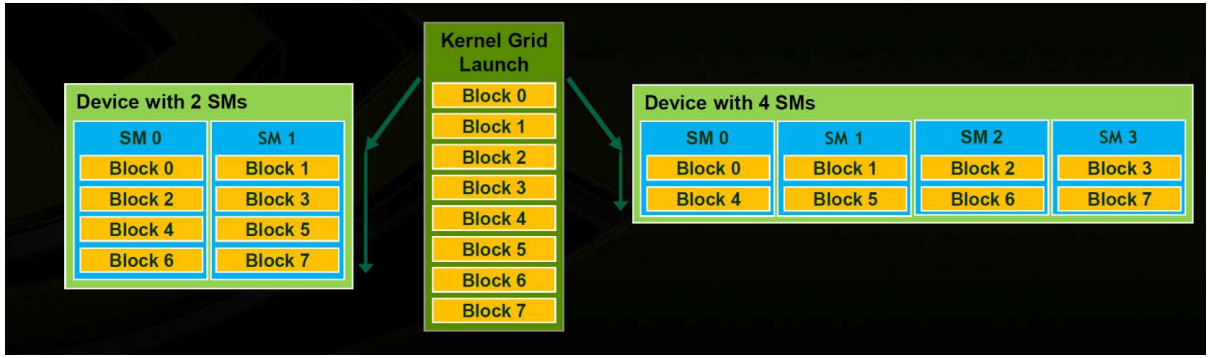
Synchronization works for threads in a particular block only. No coordination possible among different blocks; they all run independently

Block scheduling

Bocks run as independent units of threads

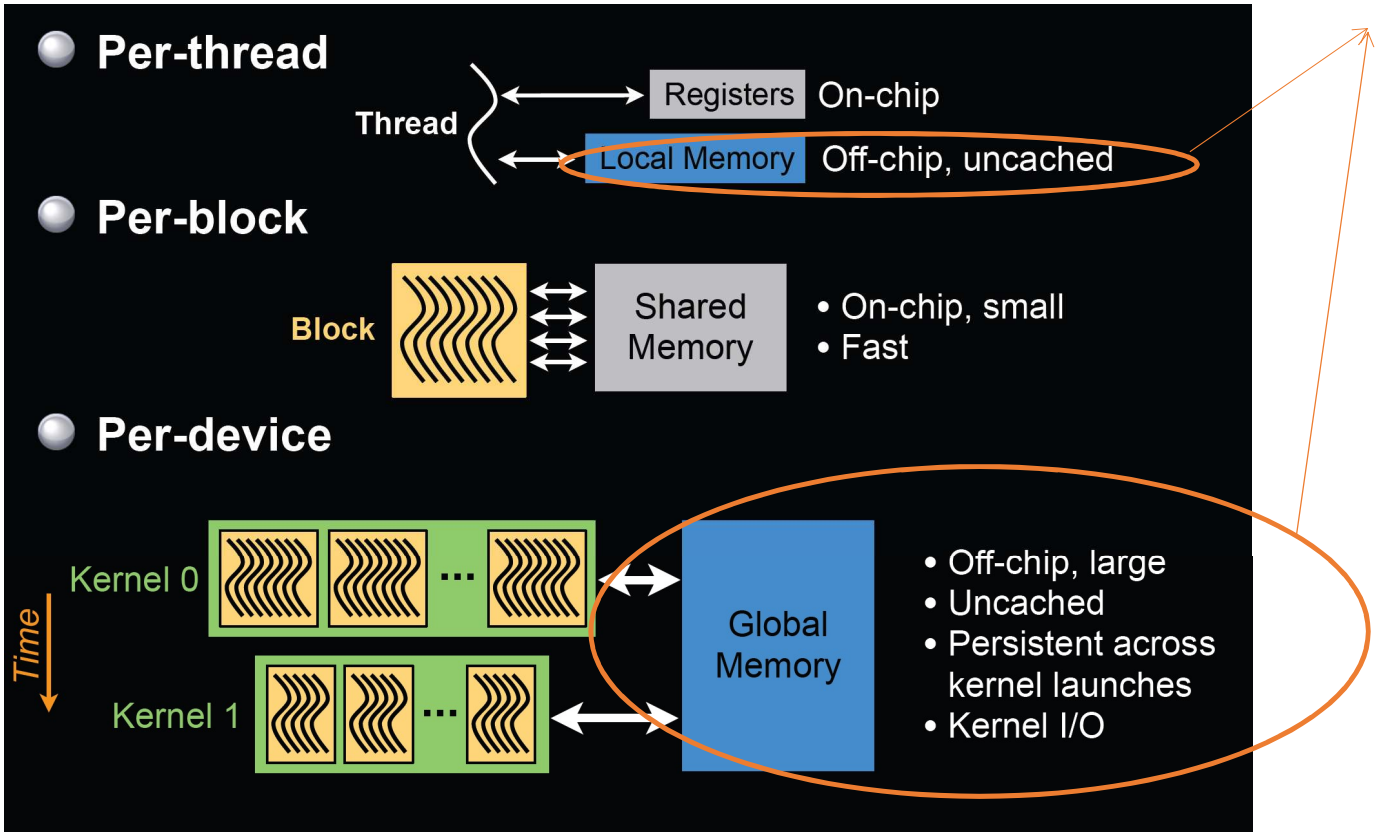
Blocks are scheduled on the SM-s based on the SM availability and on-chip (shared memory) availability

Block to SM allocation may vary during different execution of the same code. Their order of execution, thus, may vary. This may lead to different solutions of the same problem due to round-off errors in GPU codes. However, all solutions are correct to the order of machine precision.



Source: NVIDIA

GPU memory access during kernel execution



From DRAM

Source: NVIDIA

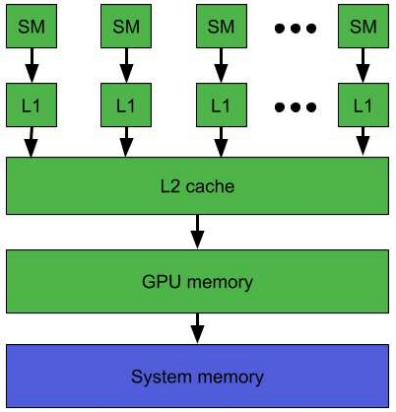
Memory access by threads from DRAM

When a CUDA kernel accesses a data region in the global memory repeatedly, such data accesses can be considered to be *persisting*. On the other hand, if the data is only accessed once, such data accesses can be considered to be *streaming*.

DRAM (global memory) is connected with the SM-s through an interconnect and hence memory transfer from the DRAM has smaller bandwidth and higher latency. So, persisting memory access is costly.

Different levels of cache is used in GPU-s. However, the cache size for each SM is in kB-s

Starting with CUDA 11.0, devices of compute capability 8.0 and above have the capability to influence persistence of data in the L2 cache, potentially providing higher bandwidth and lower latency accesses to global memory.



Source: https://medium.com/@ashanpriyadarshana/cuda-gpu-memory-architecture-8c3ac644bd64

CGMA

The *Compute to Global Memory Access (CGMA) ratio* is the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program.

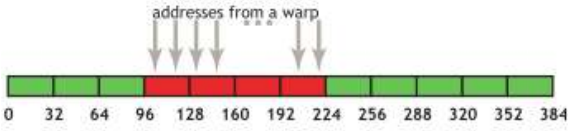
If the CGMA ratio is 1.0, then the memory clock rate determines the upper limit for the performance.

Memory transfer rate is usually one order less than processor speed. So, for CGMA=1.0 performance will be at least one order less than theoretical peak performance.

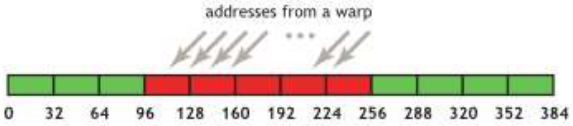
Therefore, for better performance, more calculations should be done compared to global memory access. Cache friendly codes are better performing.

Global memory access by a warp- recommendation

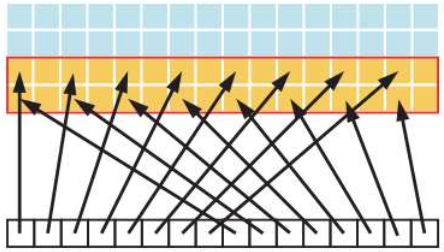
A very important performance consideration in programming for CUDA-capable GPU architectures is the coalescing of global memory accesses. Global memory loads and stores by threads of a warp are coalesced by the device into as few as possible transactions.



Simple coalesced access pattern



Sequential but misaligned access pattern



Strided access pattern
Adjacent threads accessing memory with a stride=2

First case gives highest bandwidth

Bandwidth in the second case is maintained due to L2 cache (above compute capability 6.0)

In third case, effective bandwidth reduces with increase in stride size drastically

Source: NVIDA CUDA Best practice guide

Features of device memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	Yes††	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation
† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.					
†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.					

Source: NVIDIA CUDA Best Practice Guide

Constant , Texture and local memory

Constant memory (R): There is a total of 64 KB constant memory on a device. The constant memory space is cached. The constant cache is best when threads in the same warp accesses only a few distinct locations. If all threads of a warp access the same location, then constant memory can be as fast as a register access.

Texture memory (R): It is read-only cached memory. In certain addressing situations, reading device memory through texture fetching can be an advantageous alternative to reading device memory from global or constant memory.

Local memory (R/W): Local memory is so named because its scope is local to the thread, not because of its physical location. In fact, local memory is off-chip. Hence, access to local memory is as expensive as access to global memory.

Local memory is used only to hold automatic variables. This is done by the nvcc compiler when it determines that there is insufficient register space to hold the variable. Automatic variables that are likely to be placed in local memory are large structures or arrays.

Shared memory (R/W)

Shared memory is small high througput on-chip Read-Write memory. Typically its size is 64-96 kB in modern GPUs.

Because it is on-chip, shared memory has much higher bandwidth and lower latency than local and global memory .

Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory.

Threads can access data in shared memory loaded from global memory by other threads within the same thread block.

However, as different threads can try to access same memory location in shared memory, race condition may arise leading to errors in calculation. So, thread synchronization is often required while using shared memory.

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

Performance of Jacobi solver

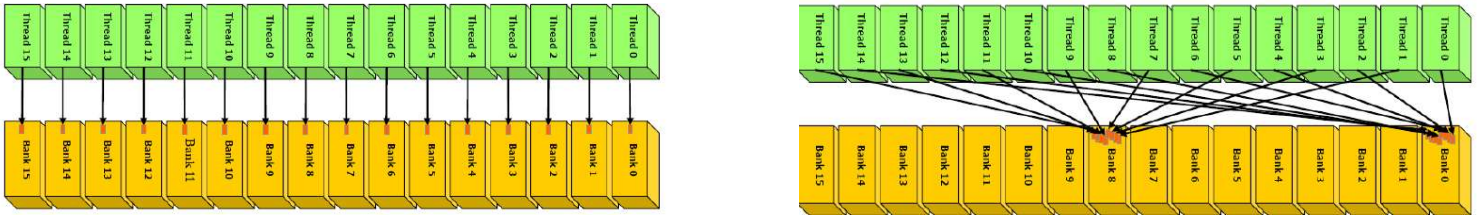
size	1000	5000	10000	20000
serial	1.48	59.29	236.7	969.88
parallel	0.41	1.61	6.07	31.23
shared	0.49	1.34	4.21	26.8

Bank conflict

To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously.

Any memory read or write request made of *n* addresses that fall in *n* distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is *n* times as high as the bandwidth of a single module.

However, if multiple addresses of a memory request map to the same memory bank, the accesses are serialized. The hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary, decreasing the effective bandwidth by a factor equal to the number of separate memory requests.



Source: http://cuda-programming.blogspot.com/

Atomic operations for avoiding race conditions

An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

For example, *atomicAdd()* reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete.

```
__global__ void mykernel(int *addr) {
  atomicAdd_system(addr, 10);       // only available on devices with compute capability > 6.x
}

void foo() {
  int *addr;
  cudaMallocManaged(&addr, 4);
  *addr = 0;

   mykernel<<<...>>>(addr);
   __sync_fetch_and_add(addr, 10);  // CPU atomic operation
}
```

The other available atomic options are: *atomicSub(), atomicMin(), atomicMax(), atomicAnd()* etc.

However, the operation is essentially serialized here and hence can degrade parallel performance for large block sizes.