Parallel Execution of CFD Applications

Somnath Roy
Associate Professor
Mechanical Engineering Department
Jointly with Centre for Computational and Data Sciences
IIT Kharagpur

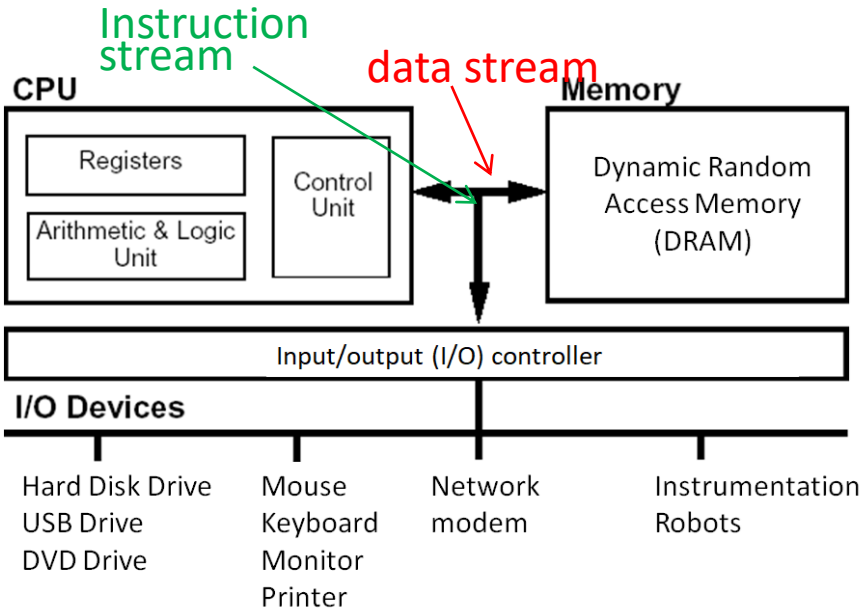
HPC-CFD NSM Workshop, Jadavpur University 2023

Outline of the Talk

GPU vs CPU
Elements of CUDA programing
CUDA programming for matrix operations
OpenACC parallelization of incompressible CFD code
MultiGPU acceleration
Performance studies

Sequential Computer Architecture-- processor & memory (RAM)

Architecture:



Computer Performance Metrics

Processor Speed:

-ex: 1 GHz Processor, one cycle takes 10^{-9} sec. Assuming 4 floating point operations in one cycle, its ideal speed is 4 GigaFLOPS!

RAM bandwidth:

Rate at which data is fetched from RAM (bytes or words per cycle)

Latency:

Time taken by the memory to get instruction from processor and return data to it

Latency is someway related to RAM speed

Sequential Computer -- Processor & Memory & Cache

✓**RAM latency hinders the performance**

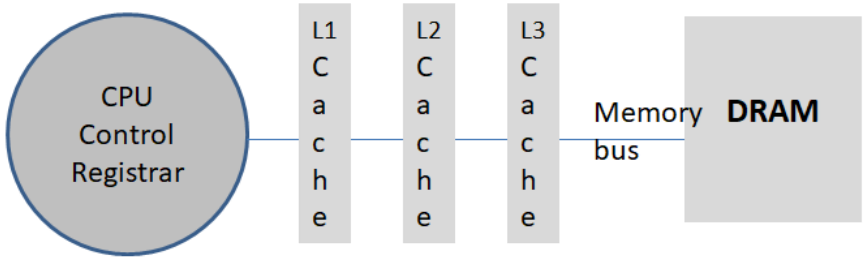
Example Problem: 1 GHZ processor with 4 operations in a cycle
DRAM has latency of 100 ns.

Ideally the processor should give 4 GigaFLOPS speed.
However, every operation needs to request for data from the DRAM and every data fetching waits for 100 ns.
So, the processor waits for 100 ns for next operation.
Therefore frequency of each opeartion is 1/(100 ns)=10 MegaFLOPS

To improve performance Cache is used – Caches are low latency high bandwidth memory units.
They are placed between processor and main DRAM

✓**Improved bandwidth and low latency in Cache augments performance**

Sequential Computer -- Processor & Memory & Cache



Size:	1 kB	64 kB	256 kB	2-4 MB	in GB-s
Speed (latency/ Bus speed)	300 ps	1 ns	~10 ns	~20 ns	~100 ns

When any data is used its subsequent items are pulled into Cache and Cache has latency typically of the order of processor cycle time

There can be multiple levels of Cache

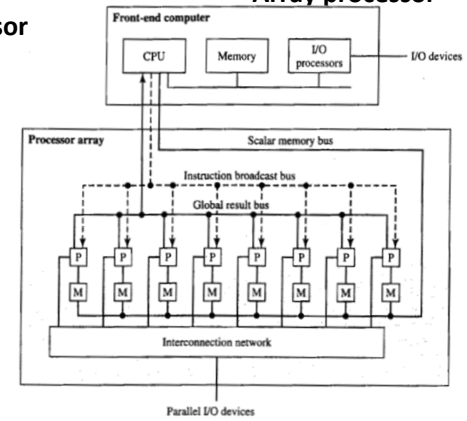
The amount of data pulled in Cache for one DRAM access is known as Cache hit ratio and depends on the algorithm

High Cache hit ratio gives better performance

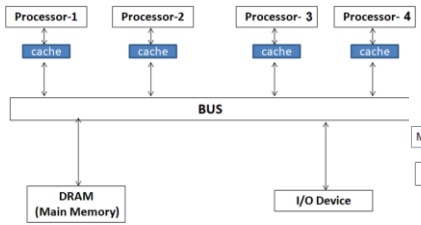
Flynn's Taxonomy-- Processor Architecture

Processor Architecture

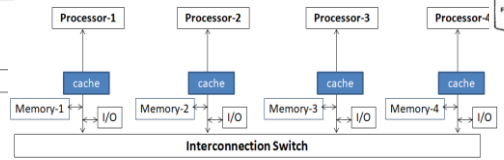
- **Single Instruction Single Data Stream (SISD)**
  - Uniprocessor
- **Single Instruction Multiple Data Stream (SIMD)**
  - Vector processor
  - Array processor



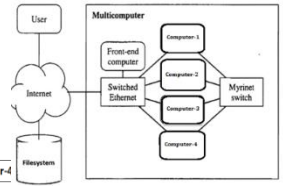
- **Multiple Instruction Single Data Stream (MISD)**
- **Multiple Instruction Multiple Data Stream (MIMD)**
  - Shared Memory (same address space)
    - Symmetric Multiprocessor (SMP) -UMA



    - Distributed Multiprocessor NUMA



  - Distributed Memory (decoupled address space)
    - Clusters



Issues in Solution Methods of Incompressible NS Equations

- Handling non-linearity
- Pressure coupling

  Semi Implicit Pressure Linked Equation (SIMPLE and family: SIMPLEC, SIMPLER)

  Marker and Cell algorithm (MAC)

  Stream function-vorticity approach

- Time-iterations

  Stability issues: Marching in time domain, time spacing has to be controlled

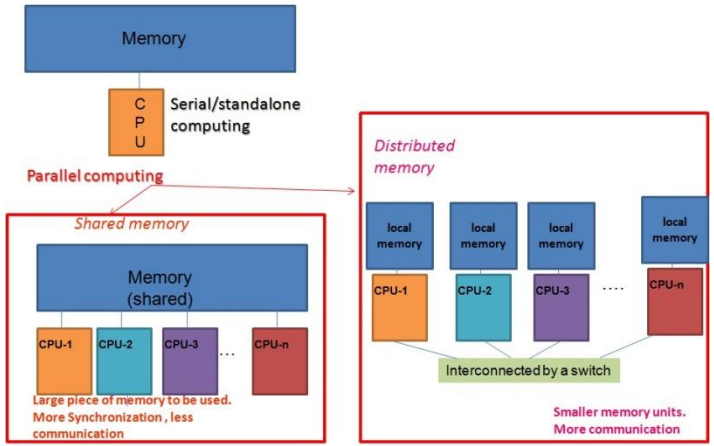
Parallel computing may help!

Incompressible CFD calculations end up in AT=b matrix equations

where A may be a very large depending on the mesh size – 10-100 million rows

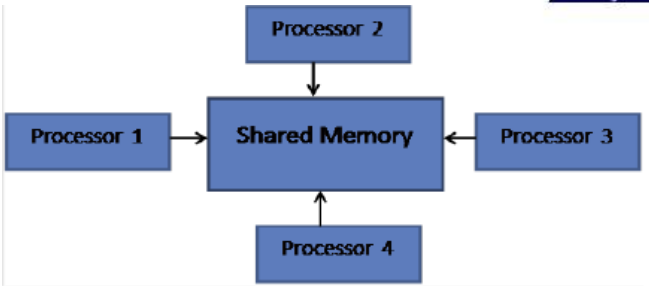
If the matrix has *N* rows, direct solvers (Gauss elimination, LU etc.) require ~(N^3) operations.

Also, the numerical errors (round-off) are high in these solvers and further, a large number of time iterations may be needed



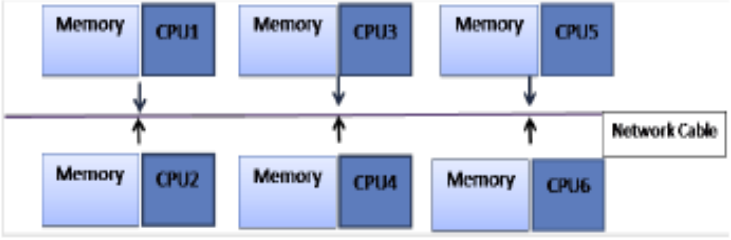
Parallel Programming Models

- **Shared memory programming model**
  - Where all processors can see whole of the memory that is available
  - On a multi-core system a single shared address space



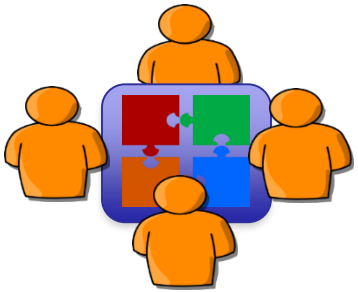
Shared Memory : Processor 1 ,2 3 4 can see whole memory

- **Distributed Programming Model**
  - Where processor can see limited memory.
  - Assume that every CPU has access to its own memory space, and can alter its own local variables.

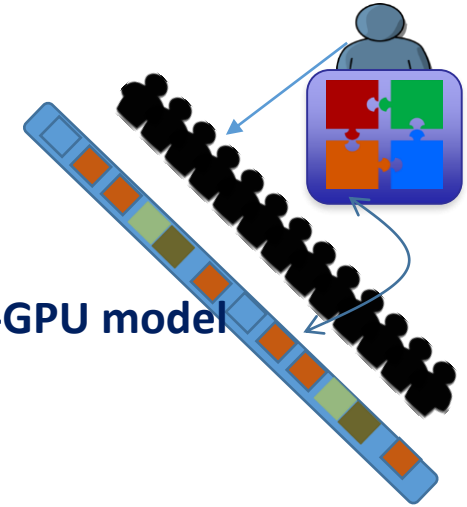


Distributed Memory System: CPU can see only limited memory of their own

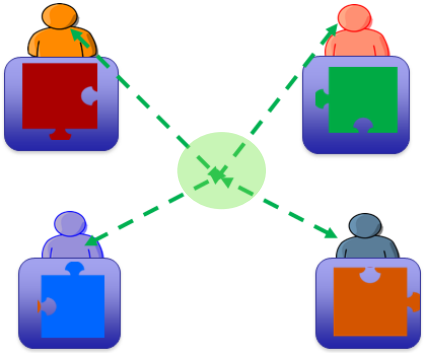
Sharing Data in Parallel Programs



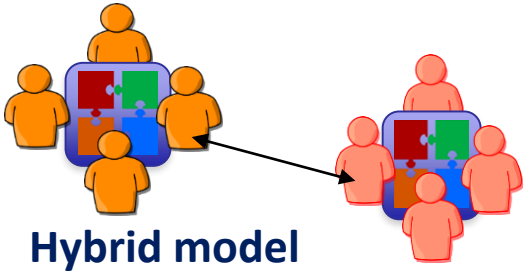
Shared Memory



CPU-GPU model



Distributed Memory



Hybrid model

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 indiepnednt 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

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

~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

~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.

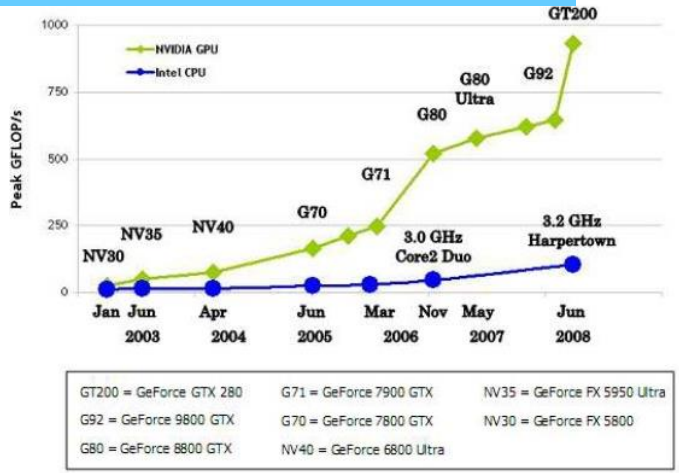
Newer versions of GPU-s are providing better speed-ups

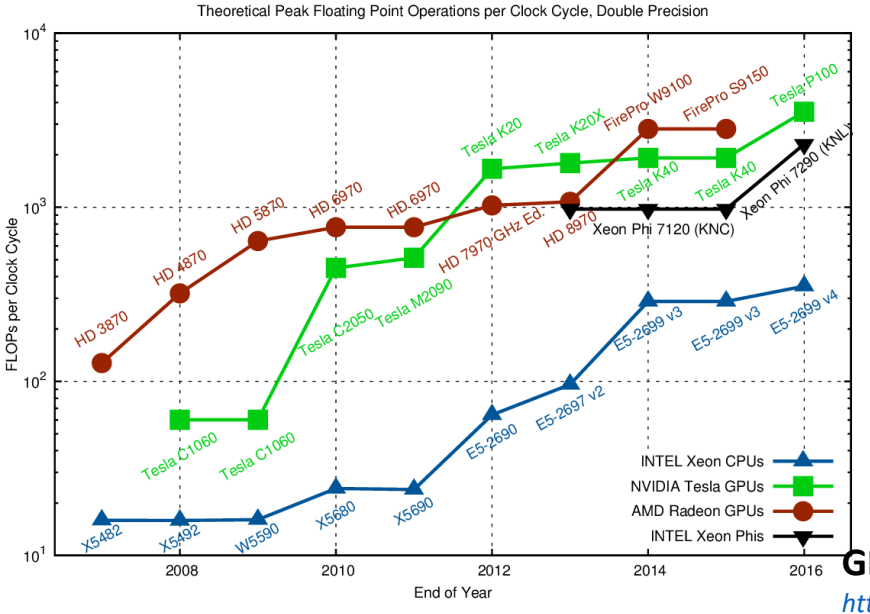
Graphics Processing Unit – Computing Power

Graphical Processing Unit (GPU), initially designed for game development

Around 2000, general purpose GPU-s or GPGPU-s developed

Efficient in matrix calculation



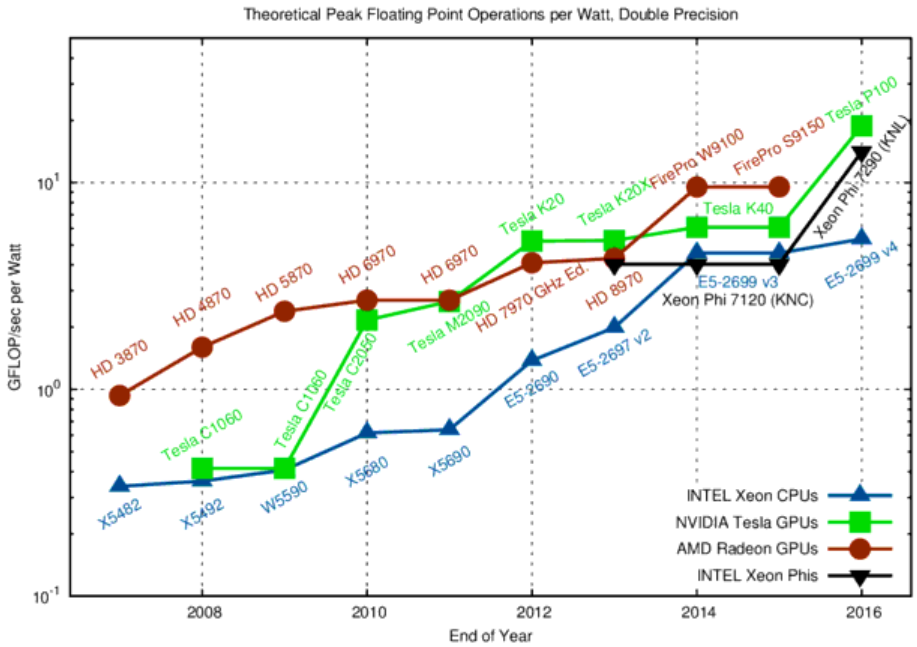


Hardware	Memory Bandwidth (GB/s)	Single Precision TFLOPS	Double Precision TFLOPS
K40	288	4.29	1.4
Titan X	480	11	0.34
P100	720	10.6	5.3
V100	900	15.7	7.8

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

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

GPUs – Energy Efficient Computing

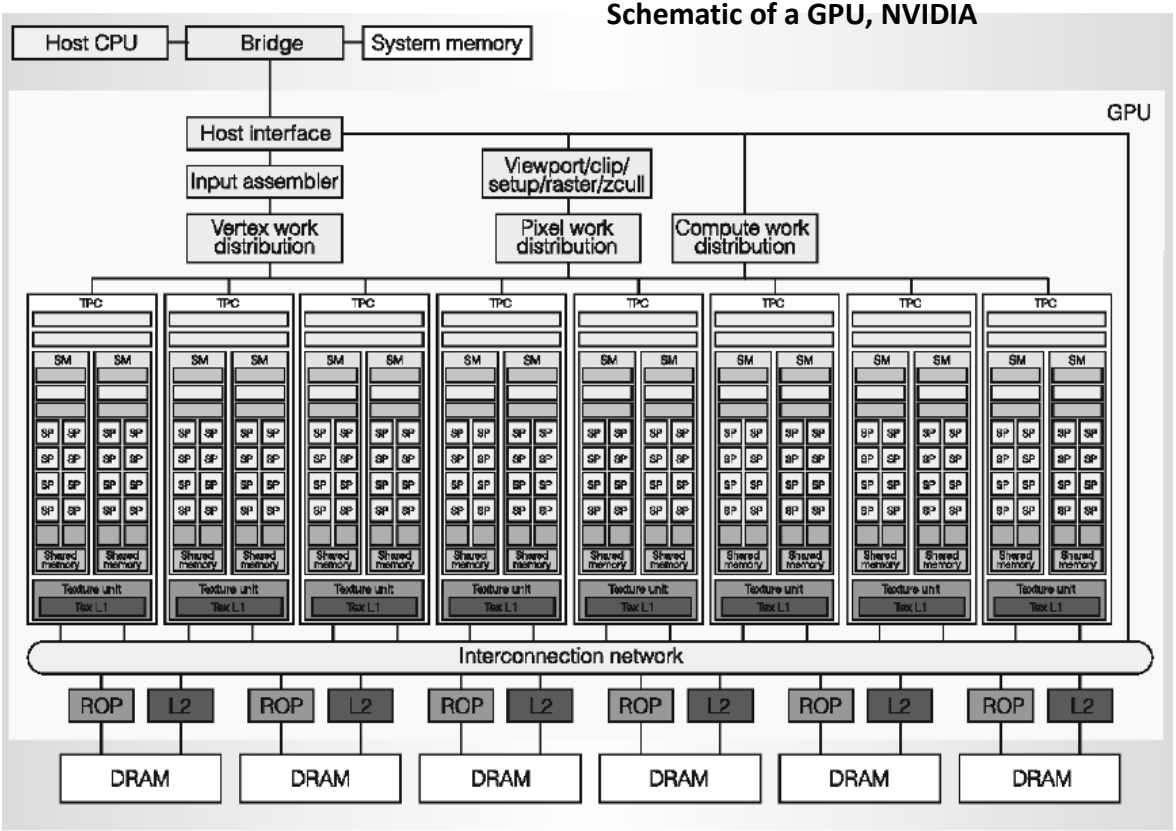


GPU-s itself cost heavy power, but due to less space and less number of components, GPU-s consume much less power than CPU-s of equivalent computing power

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

GPU 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

GPU vs CPU

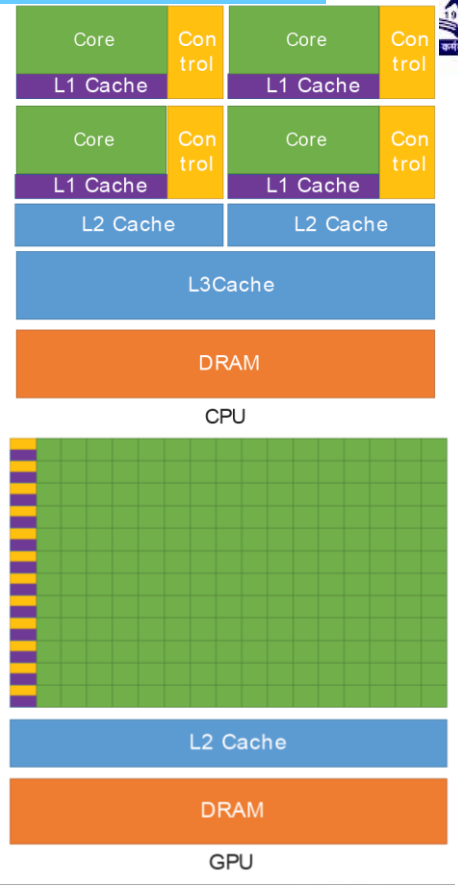
GPUs contain thousands of arithmetic units. Their power can be used to accelerate the program

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 L2 cache (Geforce 2000 seris). However Pascal or Volta has cache upto last levels



Source:
NVIDA CUDA C Programming guide

Some Terminologies

- ✓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

Single Instruction Multiple Thread (SIMT) Model for GPU-s

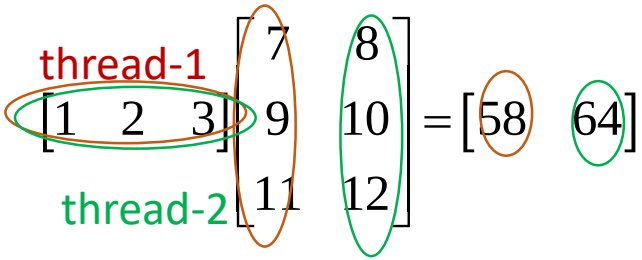
Shared memory parallel computers and GPU-s provide system support for the execution of multiple independent instruction streams.

These instruction streams which use data stream from a common address space are known as *threads*.

Threads run as a part of main program using SIMD architcture, or, even MIMD (SPMD) model.

Although the threads may run independently in different processors, they are often needed to perform synchronization using barrier calls or cache coherency protocols.

Thread based parallelization is an alternative of message passing based programs, which are deployed for distributed memory systems



GPU vs CPU: Programming Aspects

Limited number of complex tasks can go to CPU

GPU-s run a large number of simple tasks

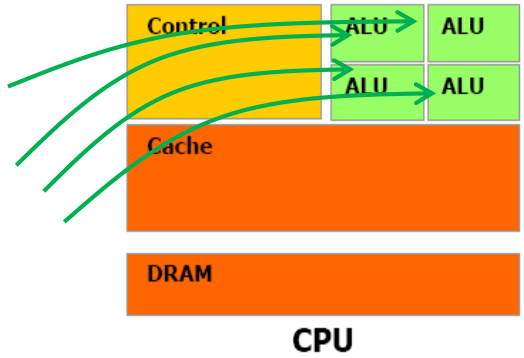
Programmer cannot directly run a job in a GPU, jobs are launched from the host CPU in the device GPU

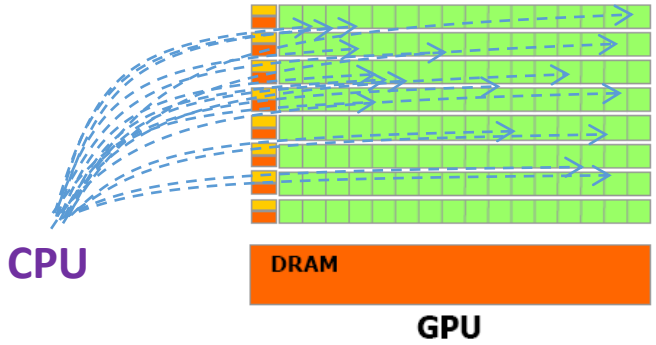
GPU jobs require more granularity and less task dependency

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 memory management 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.





Introducing CUDA

As the GPU-s provide 1000-s of cores, there has been a to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores.

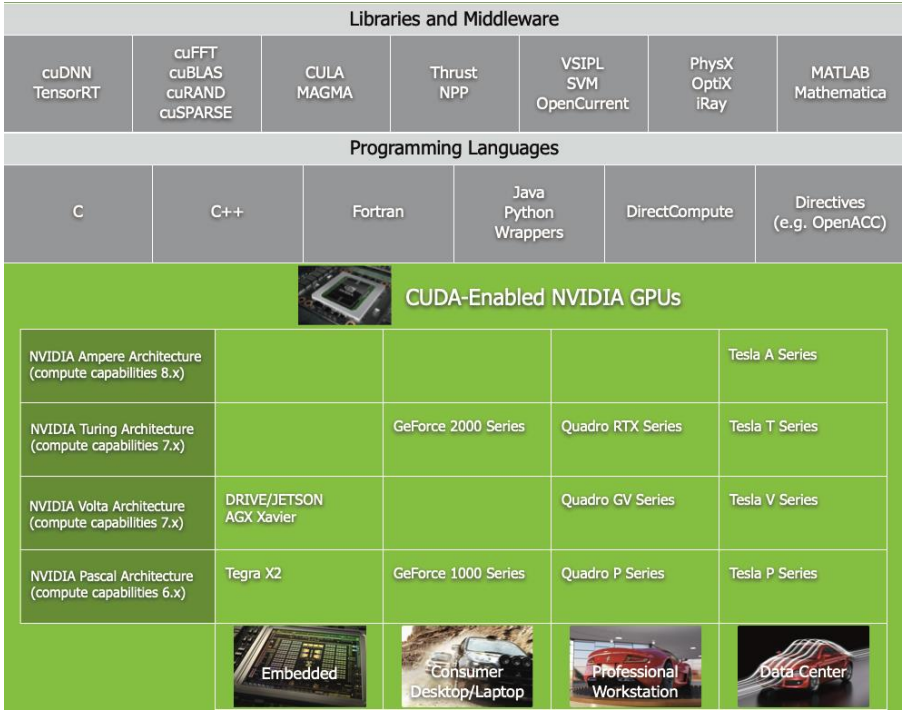
In November 2006, NVIDIA® introduced CUDA®, a general purpose parallel computing platform, programming model and API that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems with the desired scalability up to 1000-s of threads

It was initially named as Compute Unified Device Architecture

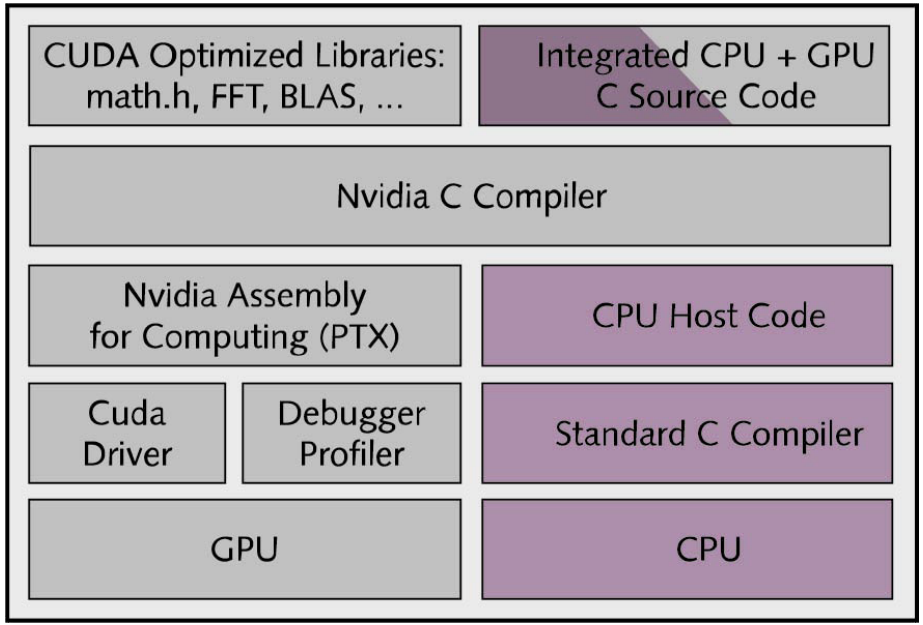
The CUDA parallel programming model is designed to overcome the challenge of scalability while maintaining a low learning curve for programmers familiar with standard programming languages.

At its core, there are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization - that are simply exposed to the programmer as a minimal set of language extensions.

CUDA for GPU Computing Applications



Source:
NVIDA CUDA C Programming guide



Source: PARALLEL PROCESSING WITH CUDA- Nvidia's High-Performance Computing Platform Uses Massive Multithreading by Tom R.Halfhill

CUDA- Programming Model

Integrated host+device app C program

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

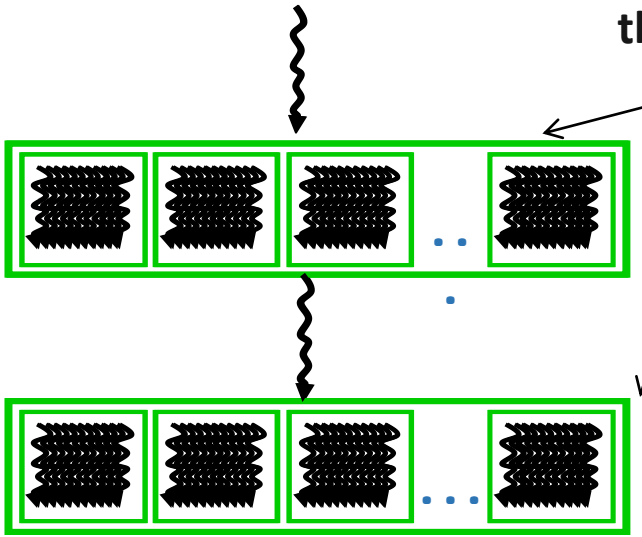
- Kernels are functions called from the CPU code and executed in the GPU
- Kernels return void

Serial Code (host/CPU)

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

Serial Code (host/CPU)

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



CUDA launches threads in blocks and grids

Threads and Blocks

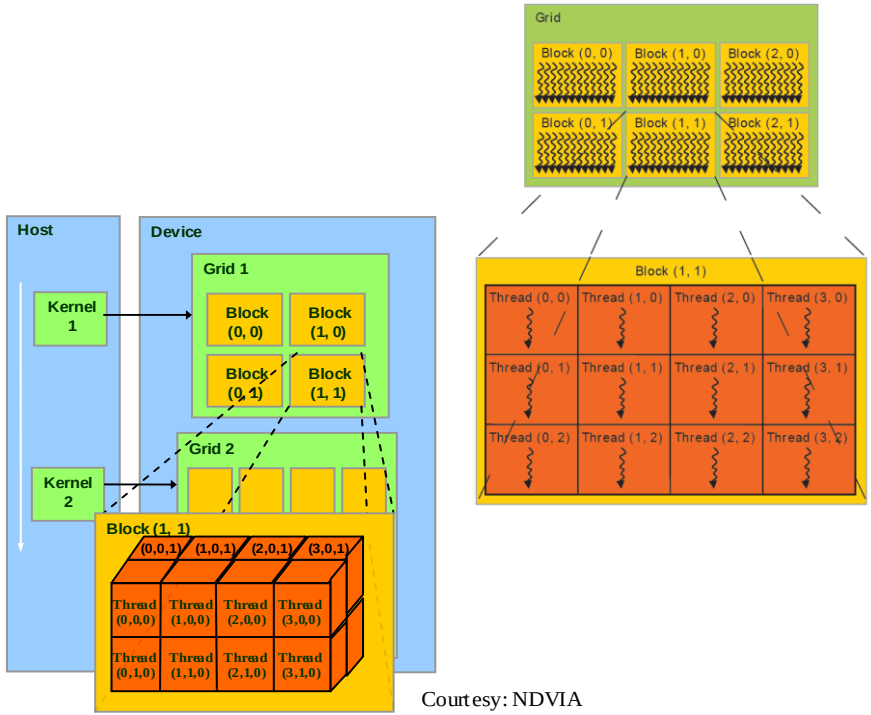
Each thread id determines what data to be worked on.

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

Device Memory Management in CUDA

CUDA provides functions for allocating and managing device (GPU) memory from the host CPU.

Two important functions are: cudaMalloc() and cudaFree().

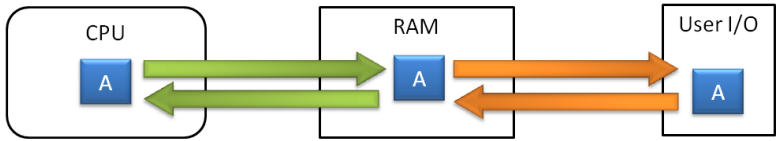
Syntax:
cudaMalloc(void **devPtr, size_t count);
cudaMalloc() allocates memory of size count in the device memory and updates the device pointer devPtr to the allocated memory.
cudaFree(void *devPtr);
cudaFree() deallocates a region of the device memory where the device pointer devPtr points to.

They are comparable to malloc() and free() in C, respectively.

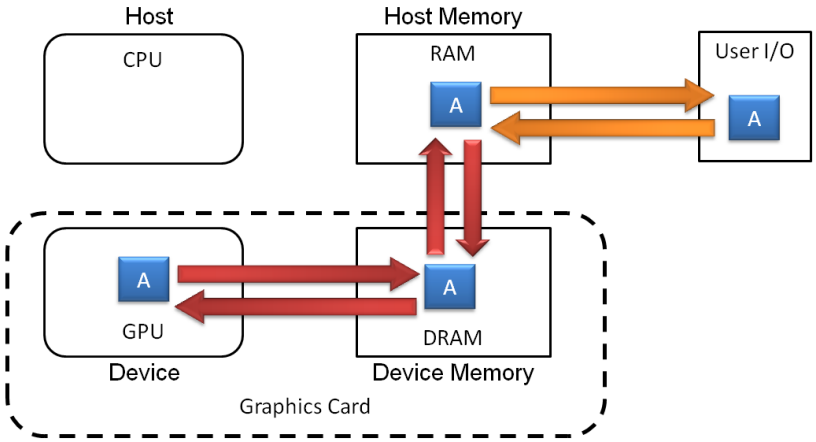
Data Transfer in a CUDA Program

CUDA program requires data transfer from host to device memory and associated workflow

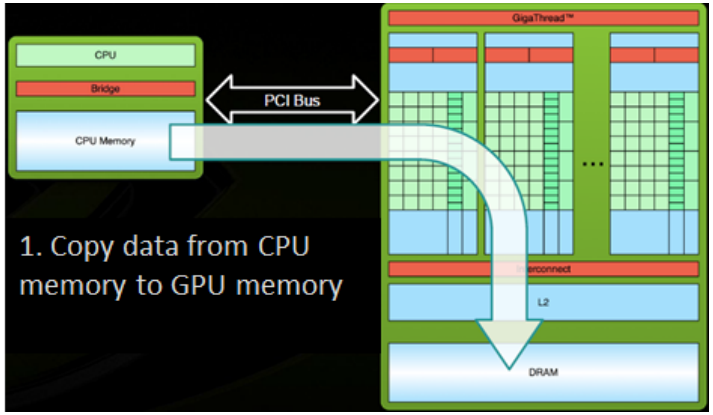
C programming model

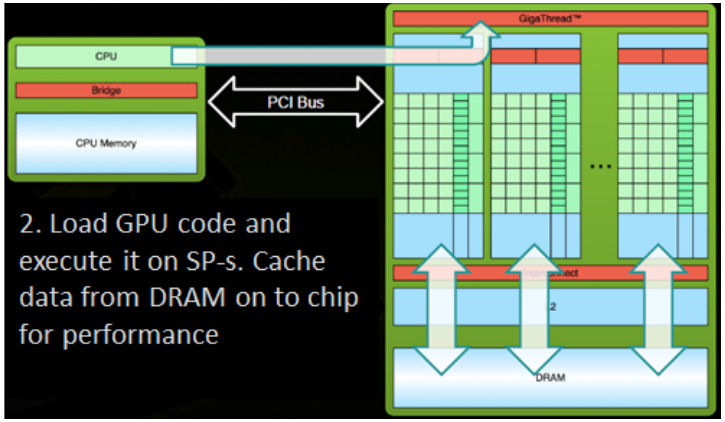


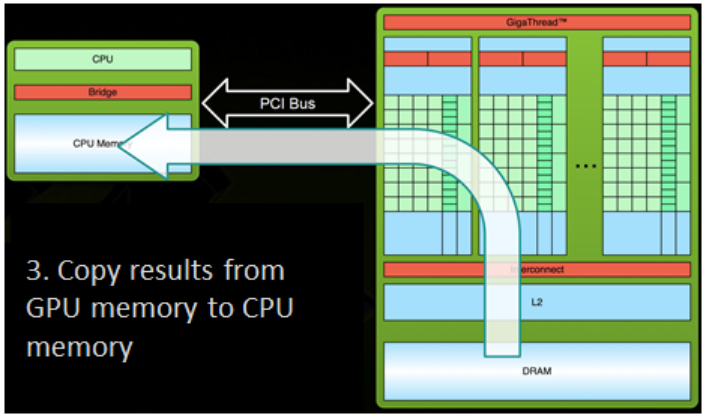
CUDA programming model



CUDA- Process Flow

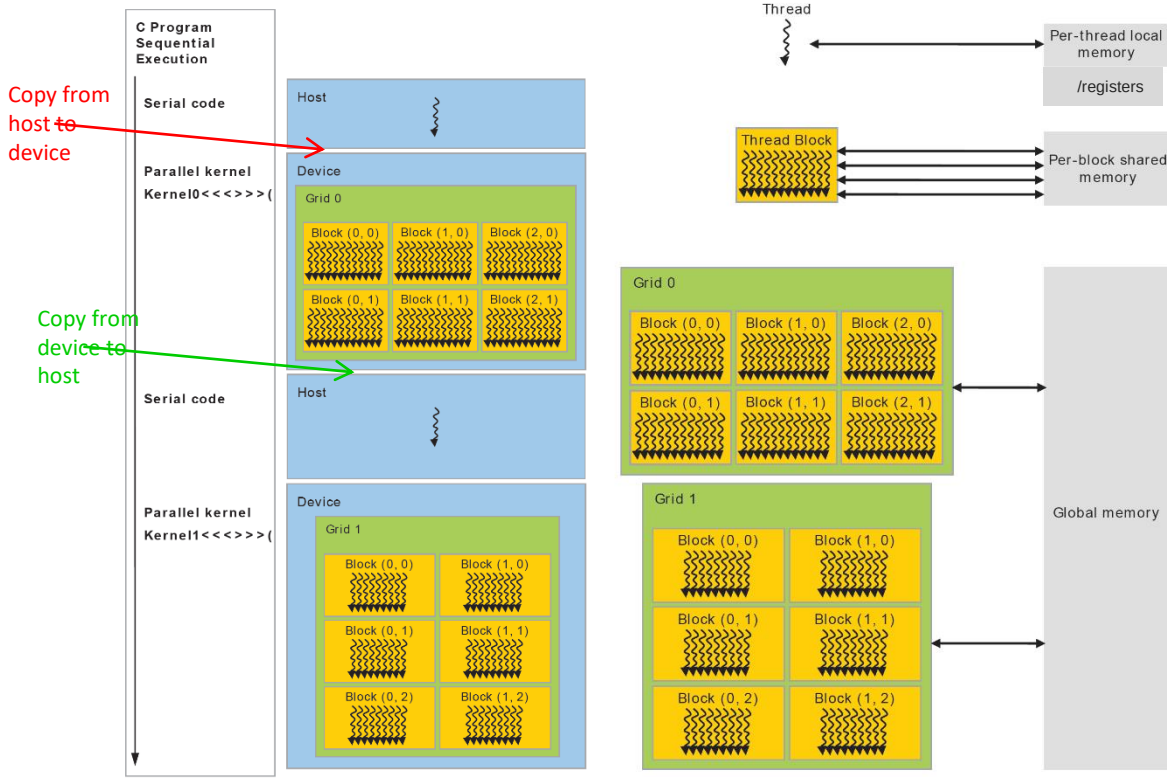






Source:
CUDA C/C++ basics, Zeller, NVIDIA

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

CUDA C Program – Vector Addition

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// CUDA kernel. Each thread takes care of one element of vector c
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
    // Get our global thread ID- one-d bloc and grid
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Add for n number of elements
    if (id < n)
        c[id] = a[id] + b[id];
}
```

Operates on the memory locations d_A, d_b & d_C

Executed by GPU

```
int main()
{
    int N = 100000;
    float *h_A, *h_B,*h_C; // CPU (host) vectors
    float *d_A, *d_B, *d_C; //GPU (device vectors)
    size_t size = N*sizeof(float);//size of each vector

    h_A = (float*)malloc(size);//Allocate CPU vectors
    h_B = (float*)malloc(size);
    h_C = (float*)malloc(size);

    cudaMalloc(&d_A, size);// Allocate device vectors
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);
```

Executed by CPU

CPU allocated memory in device via cudaMalloc

```
int i;
for( i = 0; i < N; i++ ) {
    h_A[i] = 0.75; //initialization of
    h_B[i] = 0.25; // host vectors
}
// Copy host vectors to device vector
cudaMemcpy( d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy( d_B, h_B, size, cudaMemcpyHostToDevice);

int blockSize, gridSize;
blockSize = 1024; // One-dimensional block- number of threads
gridSize = (int)ceil((float)N/blockSize); //One-d grid
                                          // number of grids
// Execute the kernel
vecAdd<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);

// Copy array back output vector from device to host
cudaMemcpy( h_C, d_C, size, cudaMemcpyDeviceToHost );

// Sum up vector c and print result divided by N: 1- no error
float sum = 0;
for(i=0; i<N; i++)
    sum += h_C[i];
printf("final result: %f\n", sum/N);
cudaFree(d_A);// Release device memory
cudaFree(d_B);
cudaFree(d_C);
free(h_A);// Release host memory
free(h_B);
free(h_C);
 return 0;
```

A function called by CPU, executed in GPU

Kernel in a CUDA program

A kernel is defined using the __global__ declaration as execution space specifier. The number of CUDA threads that execute that kernel is specified using execution configuration syntax <<<...>>> with the kernel call.

```
// Execute the kernel
vecAdd<<<gridSize, blockSize>>>(d_A, d_B, d_C, N);
```

Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through built-in variables.

```
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
    // Get our global thread ID- one-d bloc and grid
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Add for n number of elements
    if (id < n)
        c[id] = a[id] + b[id];
}
```

id gives unique number for each thread

The kernel function is executed gridsize*blocksize times through all active threads.

N threads compute for vector addition

Ref:
NVIDIA CUDA programming guide

Grid and Block Size Specification– dim3

This type is an integer vector type that is used to specify dimensions (of grid, blocks). This is a CUDA-defined structure of unsigned integers.

When defining a variable of type dim3, any component left unspecified is initialized to 1.

```
Example:
dim3 gridsize(8, 8, 2); // Grid -- 8 x 8 x 2 blocks
dim3 blocksize(32, 16, 4); // Block -- 32 x 16 x 4 threads per block
myKernel<<<gridsize, blocksize>>>(...);
```

Total number of threads in a grid= gridsize x blocksize

```
dim3 gridsize(16, 16); // Grid -- 16 x 16 ( x 1) blocks
dim3 blocksize(32, 32); // Block -- 32 x 32 ( x 1) threads per block
myKernel<<<gridsize, blocksize>>>(...);
```

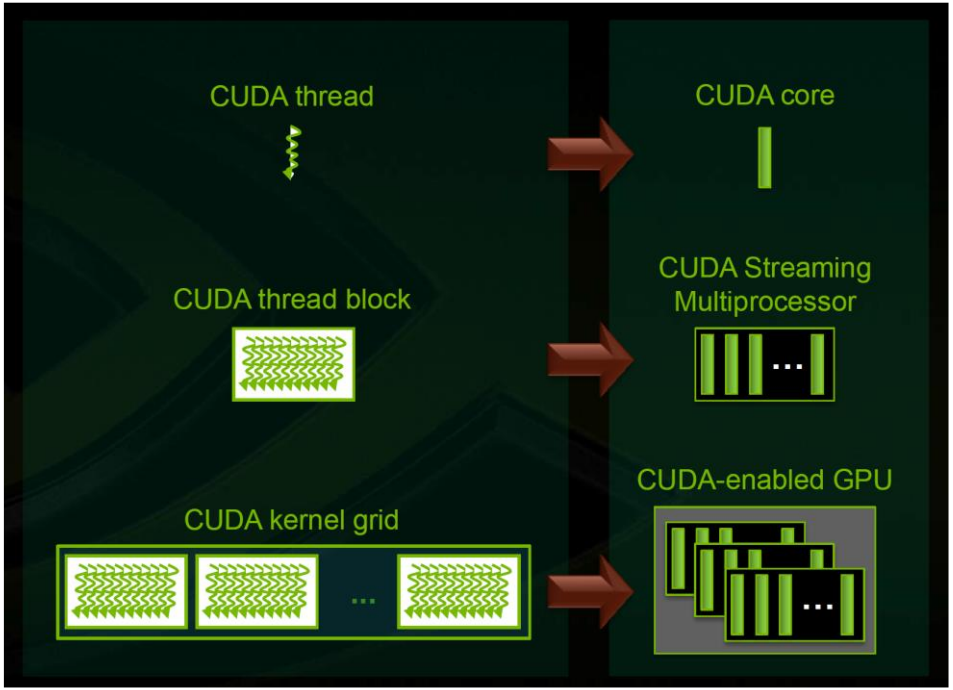
2D grid & blocks

```
gridsize=32
blocksize=1024
myKernel<<<grid, block>>>(...); // 32 blocks, 1024 threads per block
```

1D grid & blocks

Grids and blocks can be of any dimension between 1-3

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

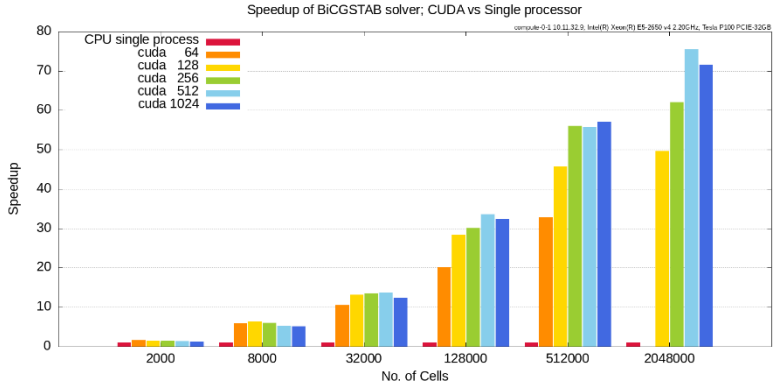
Block size can be specified by the programmer

Groups of threads within a same block are launched as sets of warps in a particular SM
All threads in a warp execute the same instruction.
If the threads of a warp diverge in execution path, each branch is executed serially adding latency

Important 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. Avoid if-else statement within a block

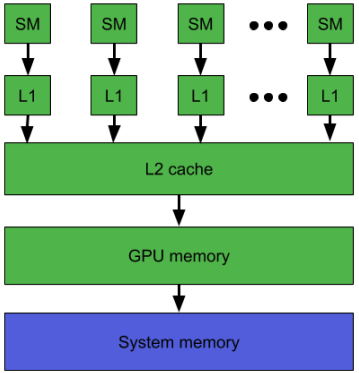
Memory Access by Threads from DRAM

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

Compute to Global Memory Access

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.

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 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

OpenACC - “More Science and Less Programming”

GPU-s can potentially give high scalability

However, thread management and data handling demand serious programming effort

CPU based codes are often to be rewritten for porting them to GPU-s using CUDA or OpenCL

Performance depends on optimizing several factors like block-size, memory access etc.

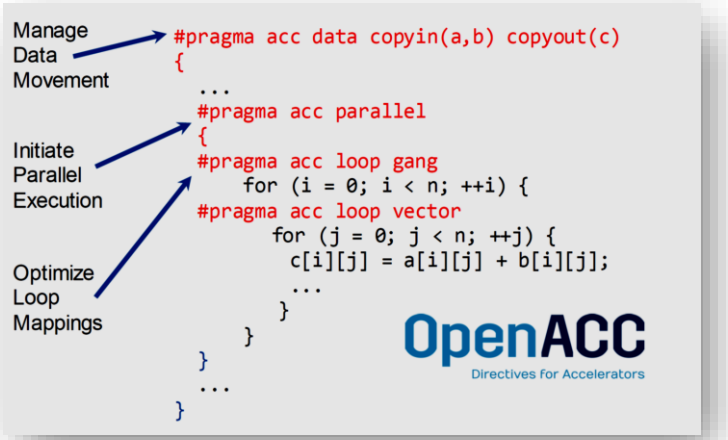
OpenACC (*open accelerators*) is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems.

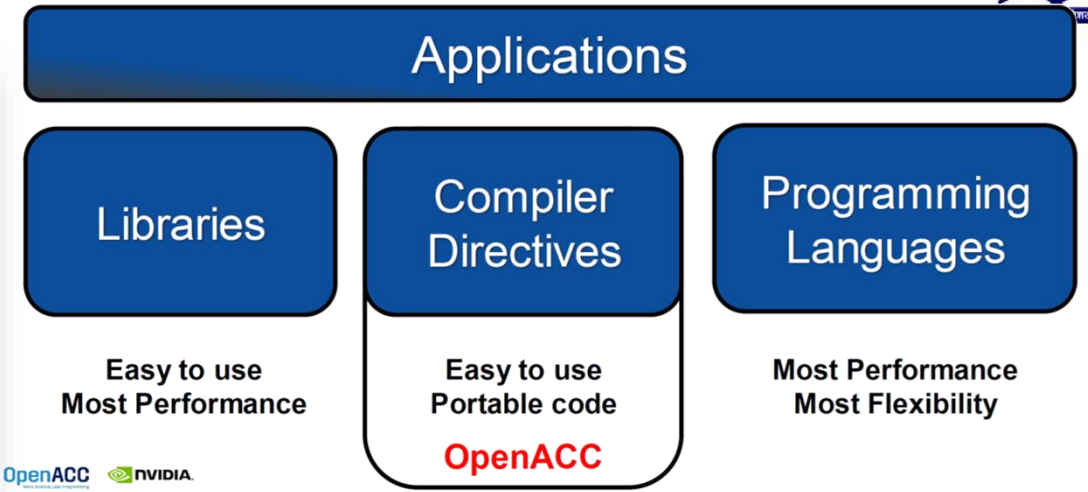
OpenACC is a directives-based parallel programming model designed for performance and portability

OpenACC is supported in both multicore CPU and GPU architecture and can launch parallel part of the computational code on them

Overall, it can make accelerated computing easy for domain experts

OpenACC – Directive Based Parallelism





OpenACC.org is a nonprofit organization founded to help scientists and researchers do more science and less programming by providing a high-level directives-based programming model for high performance computing.

- ✓**Developed by Cray, CAPS, Nvidia and PGI**
- ✓**Works on Nvidia, AMD and Intel accelerators**

OpenACC Parallel Code Development- Incremental Efforts

Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate more of the code

OpenACC NVIDIA

Enhance Sequential Code

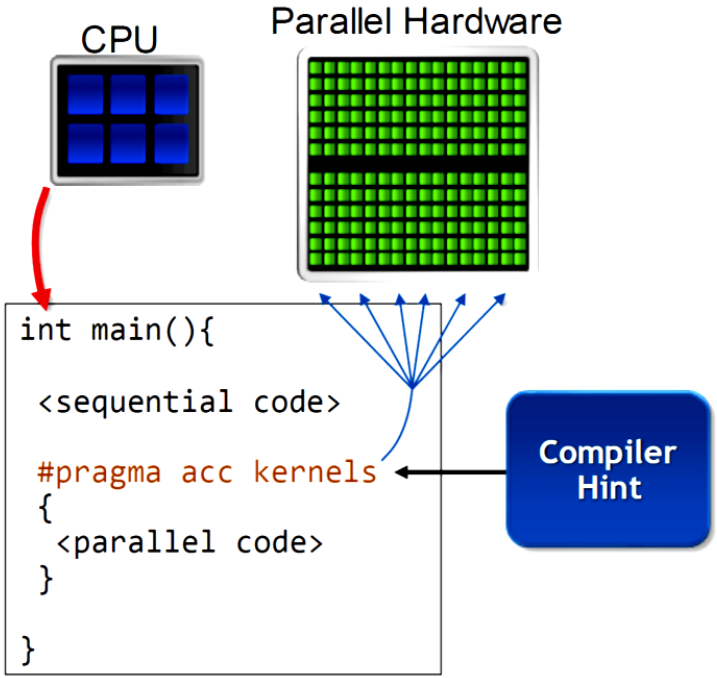
```
#pragma acc parallel loop
for( i = 0; i < N; i++ )
{
    < loop code >
}

#pragma acc parallel loop
for( i = 0; i < N; i++ )
{
    < loop code >
}
```

Begin with a working sequential code

Parallelize it with OpenACC

Verify, annotate

OpenACC – Low Learning Curve for Programmers



The programmer will give hints to the compiler.

The compiler parallelizes the code.

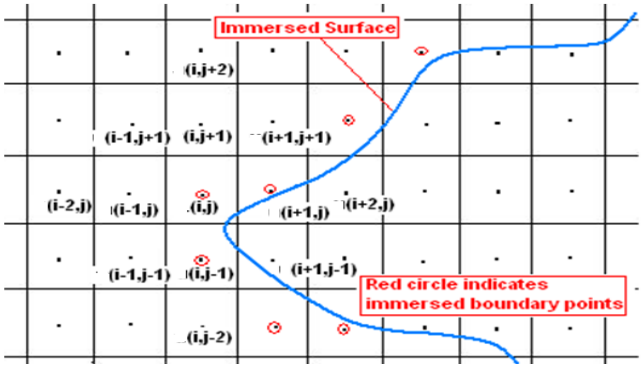
Low Learning Curve

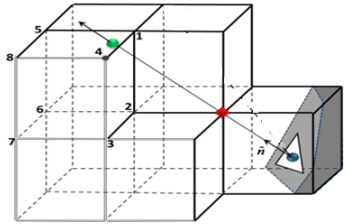
- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No requirement to learn low-level details of the hardware.

Legacy code parallelization

OpenACC Parallelization of a IBM Solver

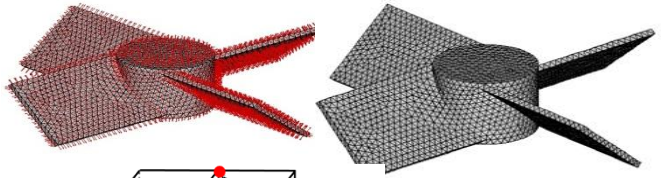
Immerse boundary method (IBM) solves flow over complex/moving boundary using a fixed Cartesian mesh

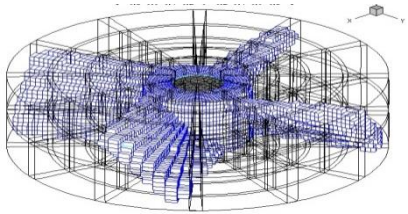




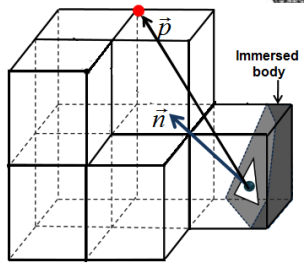
Momentum and mass conservation equations are solved in regular fluid cells
Cells nearer to the immersed surface are identified (search)
Velocity/pressure are interpolated at these points to impose boundary condition (forcing)

IBM Implementation: Search and Interpolation

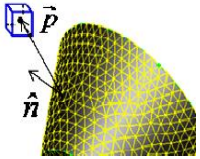




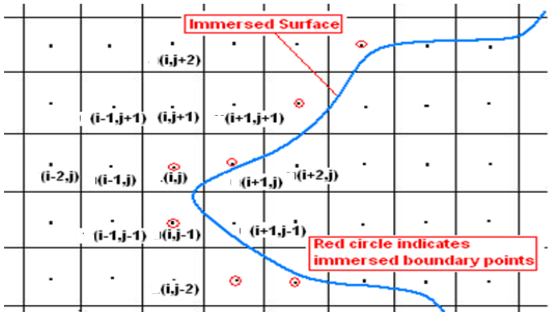
Search for intercepted cells



Check if $\vec{p}.\vec{n} > 0$?

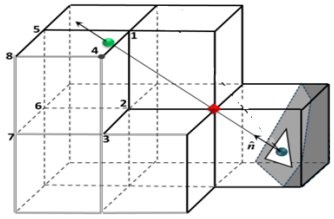


- Immersed points are identified from an stl surface mesh
- Fluid mesh points belong to a fixed structured mesh
- Immersed surface normal vectors are used to identify and tag forced cells at each time-step. Forced cells are shown in blue.



- Governing equation (incompressible NS) is solved at the fluid cells
- Flow variables are interpolated in the intercepted cells (red-circled) through normal direction quadratic function $f(n)=an^2+bn+c$
- Cells inside the solid are explicitly set to the desired motion

Interpolation/ forcing



Computational Challenges

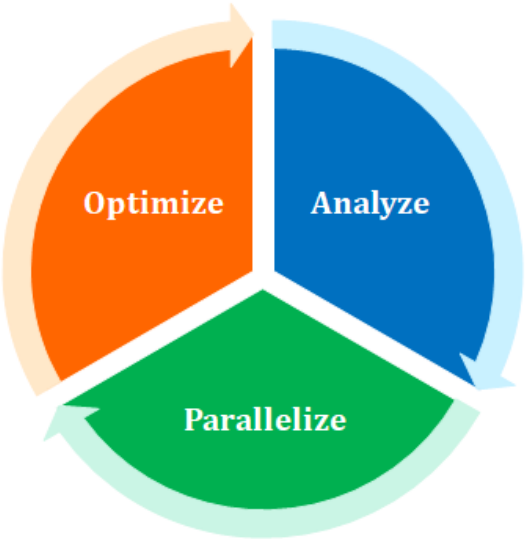
Solution of incompressible NS equation using Marker and Cell -- pressure correction equation (matrix solution) ~ $O(N^3)$

Search for intercepted cells (~10^5 surface mesh points find nearest cells from ~10^6 volume cells)

Interpolation at the forced points during solution of momentum equations

Function	% time
Poisson equation	73.7
Surface tracking	13.2
Momentum equation	5.6

OpenACC Parallelization

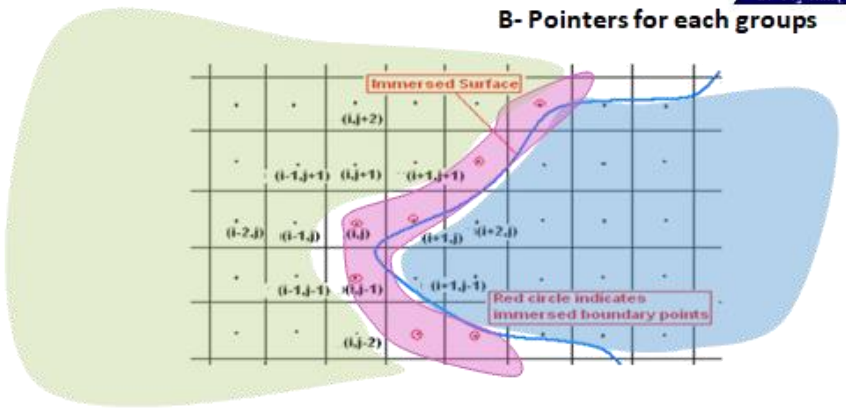


Analysis

1. Poisson equation – Jacobi solver may be well parallelized but slow convergence
2. Surface boundary condition – difficult to parallelize, possible warp divergence
3. Reconstructing parameters – memory overheads for storing fluid points data
4. Extra constraint equation for mass conservation (SOLA) – expensive for large scale CFD problems, may be dropped

GPUs – inefficient in code complexity, conditional looping etc. has small cache and memory too!

Modifications in the Algorithm

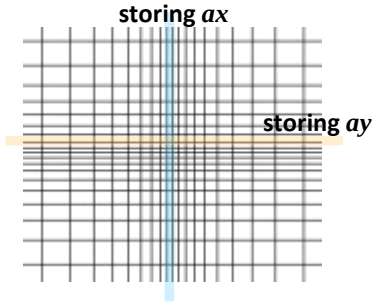


A- Red-black successive over-relaxation for faster convergence, loops for removing warp divergence

B- Grouping of cells as per fluid, intercepted and solid in one-datastructure

C- Compressed diagonal storage of matrix for reduced memory

-- cont.



- Reducing memory requirement from 7*nx*ny*nz to 2*(nx+ny+nz)
- Helps in handling ~100M cells in a single GPU

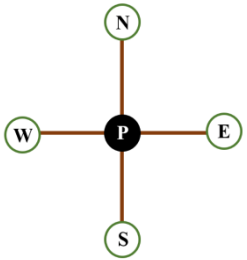
Compressed Diagonal Storage

Algorithm Matrix decomposition method

1: **for** every grid point i **do**
2: store directional coefficients in separate reduced dimensionalized arrays, e.g.,
3: Ax(i,1)=West; Ax(i,2)=Center; Ax(i,3)=East;
4: **end for**
5: **for** every grid point j **do**
6: store directional coefficients in separate reduced dimensionalized arrays, e.g.,
7: Ay(j,1)=South; Ay(j,2)=Center; Ay(j,3)=North;
8: **end for**
9: To reconstruct the coefficients for grid point i, j use,
10: Ay(j,1) + Ax(i,1) + (Ay(j,2)+Ax(i,2)) + Ax(i,3) + Ay(i,3)

$$\text{Memory utilization ratio} = \frac{\text{Banded sparse matrix storage}}{\text{Linearized matrix storage}}$$

	2D	3D
	$\frac{5*nx*ny}{3*(nx+ny)}$	$\frac{7*nx*ny*nz}{3*(nx+ny+nz)}$
	$5*n/6$	$7*n^2/9$



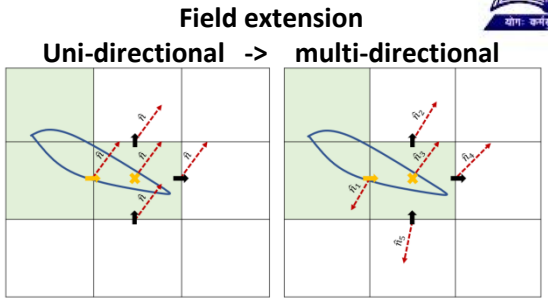
JCP, 2023

Memory utilization comparison for 5M cells

	Usage (GB)
CSR	5.12
Present	0.597

Modifications in the Algorithm (cont.)

D- Strategies for removing extra SOLA optimization loop, controlling pressure oscillation using field extension



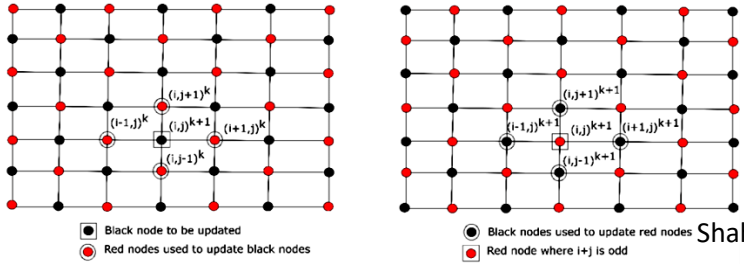
E- Simplified reconstruction scheme that is easily parallelizable, one weight point. Using gradient values for interpolation

F- Using explicit forcing (last time-step's values) to update projected velocity field – avoiding contention among threads!

G- Dynamic allocation; create and destroy arrays at each time-step

GPU Acceleration Steps using OpenACC

Red-black SOR for GPUs



Shah et al., 2019

Speedup - SOR (CPU) vs RB-SOR (GPU) without individual grouping of the cells

Mesh Size	speedup
20480	7x
163840	30x

An additional 10-15% time saving is observed after separate grouping of red and black cells

```
Do i = 1 to fluidcellcount    i = ...
   j = ...
   If (mod(i+j),2).EQ.0 then
     ....do red cell computation and update
   Endif

  If (mod(i+j),2).NE.0 then
    ...do black cell computation and update
  End if
End do
```

Warp divergence

```
!$acc parallel loop default(present)
DO n=1,redCellCount
     i=redIndexPtr(n,1)
     j= redIndexPtr(n,2)
     redCell pressure Correction calculation
END DO
```

```
!$acc parallel loop default(present)
DO n=1,blackCellCount
     i=blackIndexPtr(n,1)
     j= blackIndexPtr(n,2)
     blackCell pressure Correction calculation
END DO
```

Without warp divergence

Performance of the Parallel Solver

Profiling using GPROF and NSIGHT SYSTEMS

Function	CPU time (%)	CPU time (s)	GPU time (%)	GPU time (s)
Poisson equation (Fluid cells)	71.35	10827.8	67.6	50.1
Surface tracking	26.99	4095.58	31	22.8
Momentum equation	0.34	51.59	0	0.9
Surface boundary condition	1.25	189.31	1.4	.93
Memcpy (host ↔ device)				1.63

Speedup - CPU vs GPU

Mesh points (millions)	CPU time (s)	GPU time (s)	speedup
≈8	4172	37.2	112x
≈12	9267	72.3	128x
≈26	19021	136.9	139x

5 iterations on an Intel Xeon SKL 6148 CPU and NVIDIA Tesla V100 GPU

≈11.7 million mesh points	**V100**		**A100**	
Function	**time (%)**	**time (s)**	**time (%)**	**time (s)**
Poisson equation	85.9	145	80.3	83.4
Surface tracking	**13**	**22.5**	**18.1**	**18.8**
Momentum eqn	0.2	0.36	0.2	0.24
Interpolation	0.9	1.49	1.3	1.32
Memcpy		**3.12**		**1.82**

Optimization of the Parallel Solver

1. Surface tracking algorithm

2. Using DECLARE directive with CREATE cause instead of allocating in each time-step – memory reduction for moving body problems

3. Better arithmetic for memory indexing, checking convergence after 50 sub-iterations, reducing reduction steps

4. Z-directional domain specification, packing overlap data for multi-GPU framework

Nearest neighbor	Bounding box	Speedup
13.40	4.18	3.21x

PERFORMANCE OF CURRENT RBSOR IMPLEMENTATION

Mesh (M)	Previous Implementation	(s)	Present (s)	Speedup
2.5		0.34	0.18	1.864
5.0		0.66	0.35	1.879
10.0		1.24	0.67	1.853
20.0		2.39	1.30	1.830
40.0		4.88	2.67	1.829
80.0		11.00	5.49	2.003

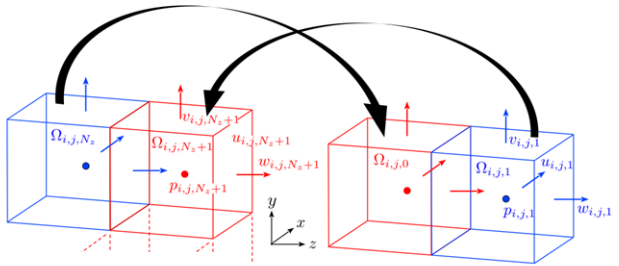
Overall performance improvement by another 1.5x factor

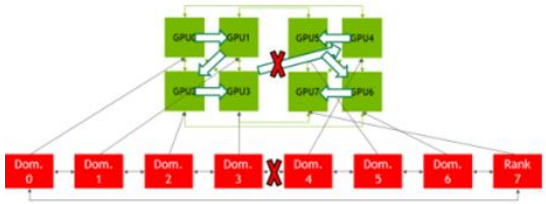
Multi-GPU Optimization

1. Domain decomposition is performed for optimal load balancing.
2. Dual level parallelism- MPI, OpenACC.
3. Layer of buffer cells are introduced for implementation of sub-domain boundary conditions.
4. Asynchronous point-to-point communication routines are used for overlapping multiple data transfers

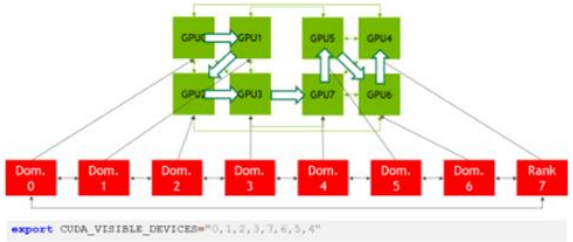
Asynchronous Point-to-point calls

Asynchronous	Synchronous	Speedup
90.71	118.99	1.24x





DGX V100 layout and usual GPU assignment



Optimized GPU assignment based on ring topology

Multi-GPU Performance

Poisson Equation

GPUs / Points	1	2		4	
	Time (s)	Time (s)	Speedup	Time (s)	Speedup
2.56 M	5.3	3.4	1.55x	2.7	1.93x
20.48 M	36.8	19.7	1.87x	11.1	3.31x
69.12 M	123.8	62.0	2.00x	33.3	3.72x
109.76 M	213.7	99.9	2.14x	52.9	4.04x

IB method (Pipe)

GPUs / Points	2			4		
	Total	Poisson	Surf. Track.	Total	Poisson	Surf. Track.
5.76 M	1.8x	1.5x	2.0x	3.0x	1.7x	3.9x
19.44 M	2.0x	1.7x	2.0x	3.7x	2.6x	4.0x
46.08 M	2.0x	1.8x	2.0x	3.9x	2.9x	4.0x
90.00 M	2.0x	1.9x	2.0x	3.9x	2.8x	4.0x

Multi-GPU Scalability of flow through a stenosed artery (IBM)- Solver time (~Poisson+Search)

GPUs / Routine	16	32	
	sec	sec	speedup
Main	891.0	483.8	1.85x
Search	489.3	244.5	3.98x
Poisson	398.8	237.8	1.68x

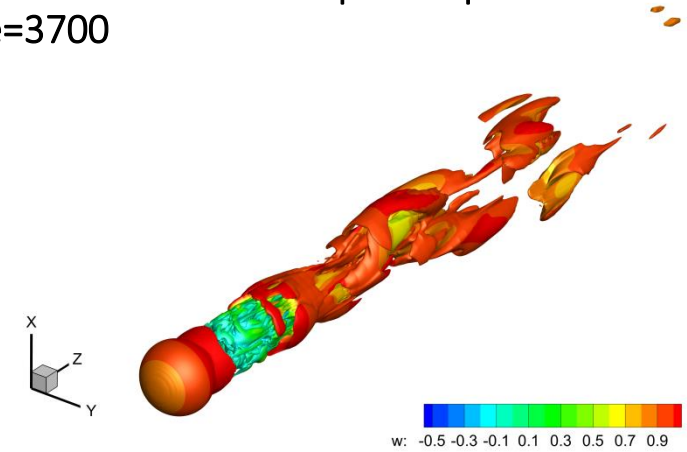
Parameters: ita=100; pcItaMax=1000; no. of cells=500M

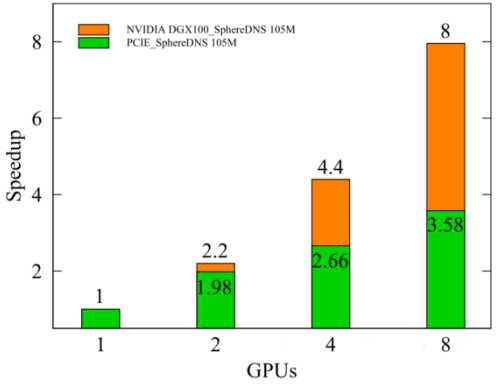
VIII. A100 vs V100 performance

Platform	GPUs	Speedup	Relative Performance A100 / V100
DGX A100 NVSwitch	1	-	1.82
	2	1.91x	1.67
	4	3.66x	1.59
	8	6.37x	1.48

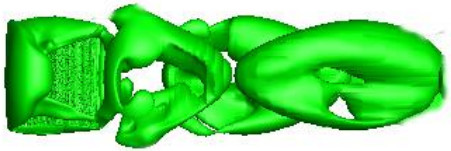
External Flow Cases

Vortex structure for flow past a sphere at Re=3700

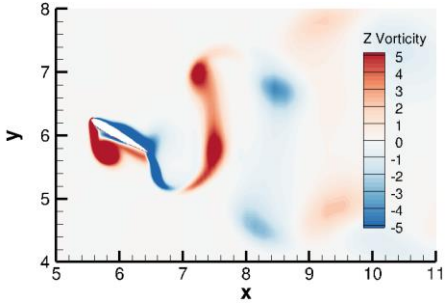


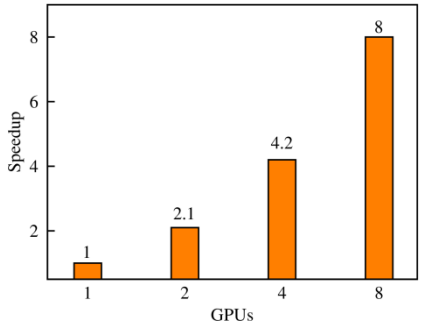


Flow over wings



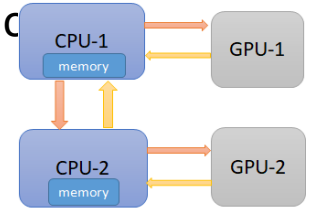
58 M cells



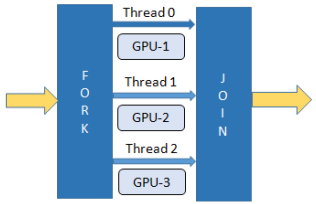


OpenMP-OpenAcc Multigpu Implementation -DNS

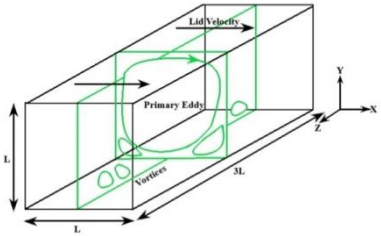
For small mesh size, OpenMP parallelization can help in reducing communication c



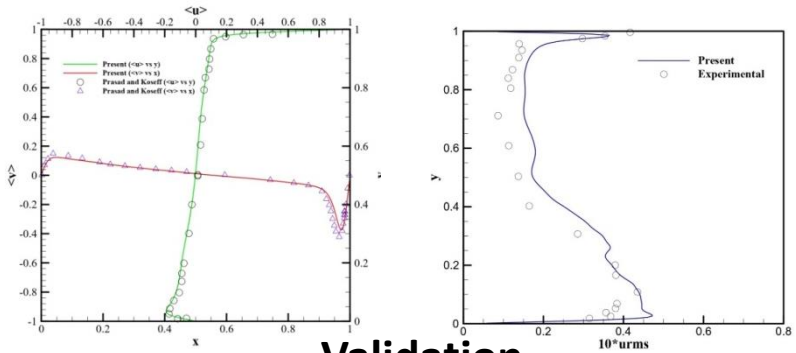
MPI + CUDA/OpenACC



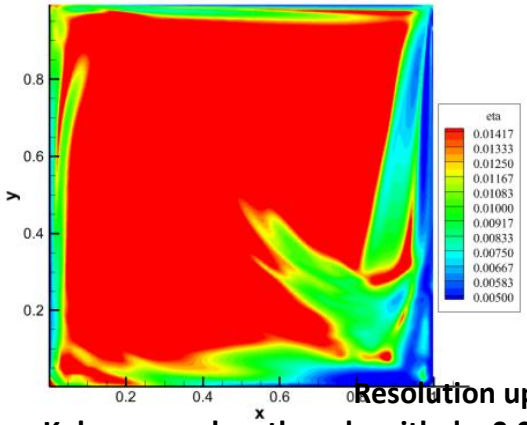
OpenMP + OpenACC



Re = 10000



Validation



Resolution upto Kolmogorov length scale with dx=0.004

Size	CPU		GPU	
	Single Thread	Two Threads	Single GPU	Multi GPU
0.375M	7.598	3.802	2.149	2.424
1.305M	27.549	14.127	2.375	2.615
3.000M	61.841	31.508	3.232	2.784
7.875M	167.625	84.401	4.463	2.925
10.125M	213.350	109.035	5.052	3.545

Performance of Multi-GPU Codes

Multi-GPU programs are well scalable

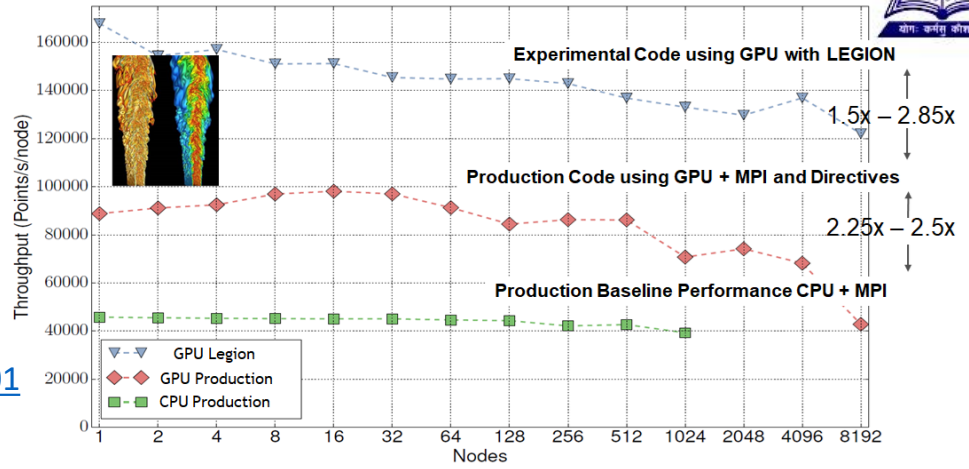
Example: DNS combustion code FUN3D

https://www.olcf.ornl.gov/

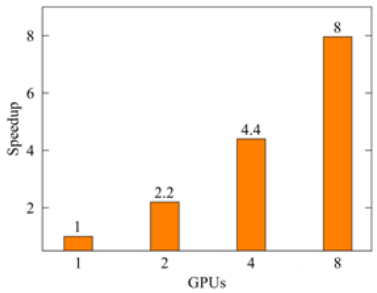
Titan generation GPU-s are used

Legion programming gives best optimization

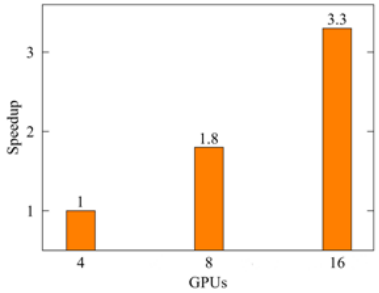
http://developer.download.nvidia.com/GTC/PDF/GTC2012/Posters/P0535_Legion__GTC__2012.pdf



In-house code performance for flow over moving bodies



Mesh: 105 million.
Infra: 1x DGX-A100 40 GB.
Nvidia's cluster.



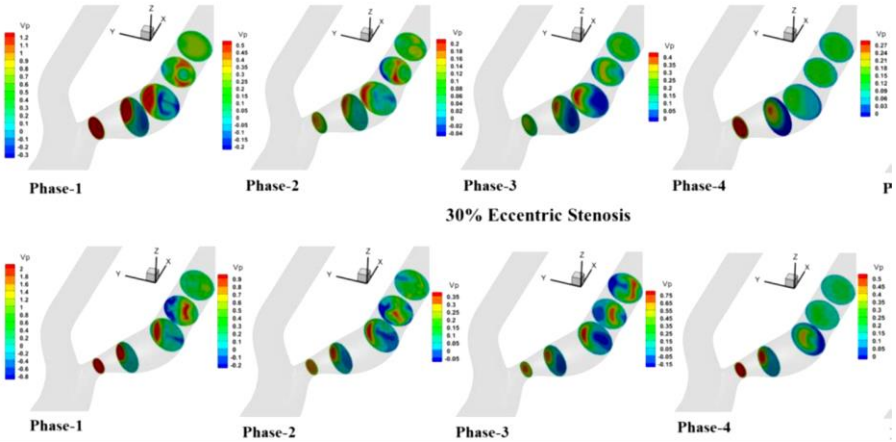
Mesh: 80 million-1.28 billion
Infra: 2-8x nodes (2x V100 16GB PCIE each)
PARAM Shakti.

Multi-GPU Solver for Arterial Flows

Multi-GPU Scalability of flow through a tube (IBM)- Solver time (~Poisson+Search)

GPUs / Cells	2			4		
	Main	Poisson	Search	Main	Poisson	Search
5.76	1.8x	1.5x	2.0x	3.0x	1.7x	3.9x
19.44	2.0x	1.7x	2.0x	3.7x	2.6x	4.0x
46.08	2.0x	1.8x	2.0x	3.9x	2.9x	4.0x
90.00	2.0x	1.9x	2.0x	3.9x	2.8x	4.0x

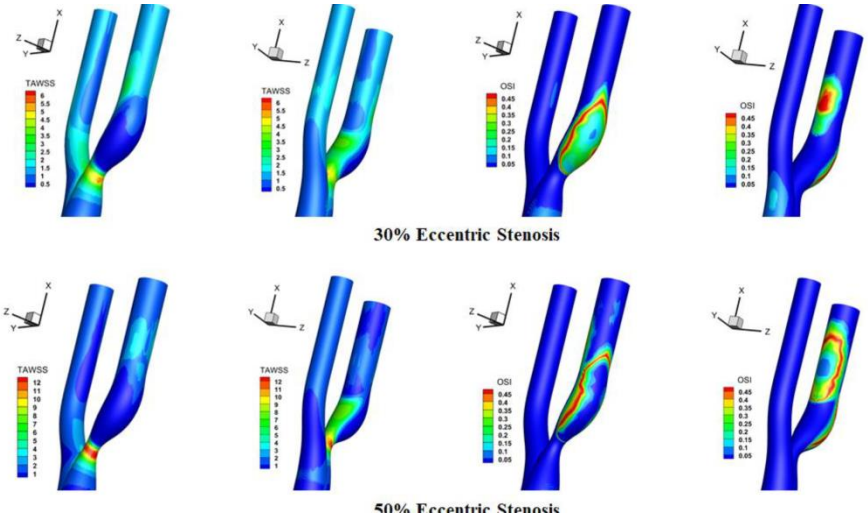
Parameters: ita=100; pcItaMax=1000; cells in Millions



Multi-GPU Scalability of flow through a stenosed artery (IBM)- Solver time (~Poisson+Search)

GPUs / Routine	16	32	
	sec	sec	speedup
Main	891.0	483.8	1.85x
Search	489.3	244.5	3.98x
Poisson	398.8	237.8	1.68x

Parameters: ita=100; pcItaMax=1000; no. of cells=500M



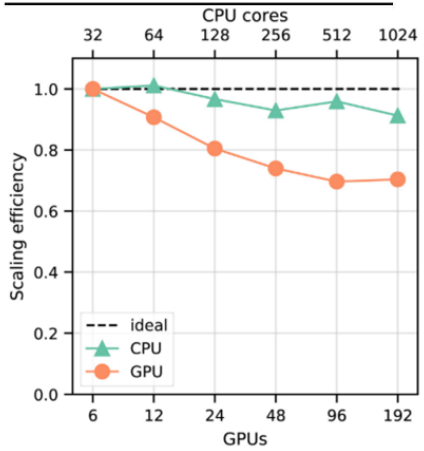
Khan et al., 2023

Multi-GPU Hemodynamics Simulations – Jeff et al., 2020

- A complete GPU-accelerated FSI simulation applicable for cell-resolved hemodynamics.
- Scaling shown in two different architectures . One having 6 GPUS per node(Summit) and another having a single GPU (DCC) per node.
- The maximum fluid grid used had 17M RBCs.

Nodes	Fluid Grid	RBCs
1	500x500x500	0.53M
2	629x629x629	1.05M
...	...	...
32	1587x1587x1587	17M

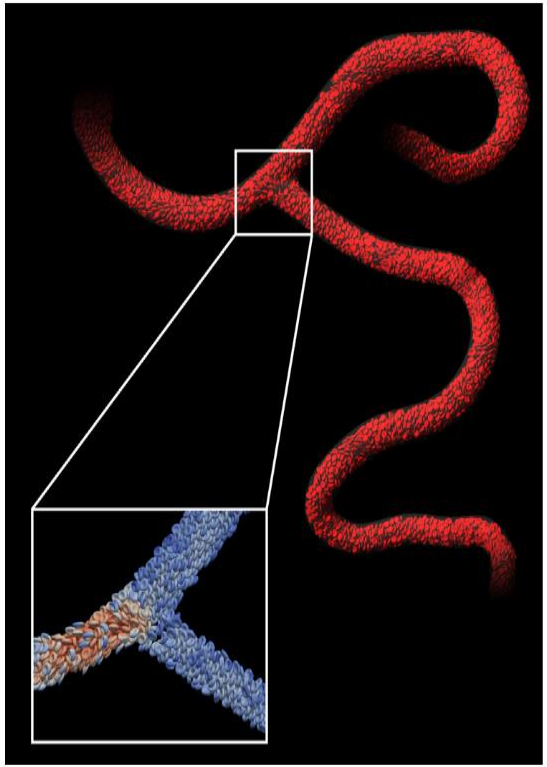
Nodes	Fluid Grid	RBCs
1	300x300x300	0.133M
2	377x377x377	0.224M
4	376x376x376	0.457M
8	600x600x600	0.918M



Top table shows fluid grids and corresponding RBCs used in Summit, bottom table shows the same for DCC.

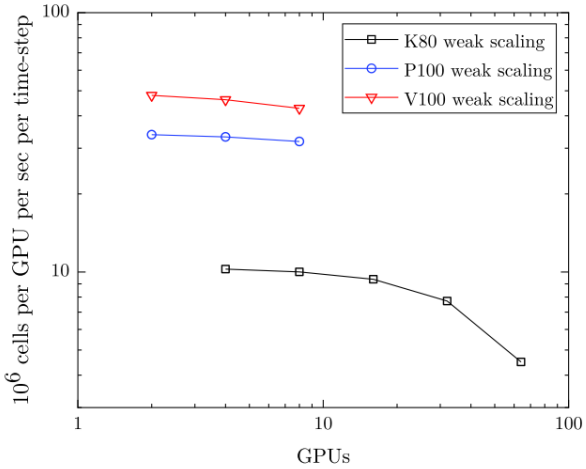
The figure on the right shows red blood cells in a human cerebral vascular geometry simulated with HARVEY. Cells in the highlighted window are colored by vertex velocity

Below graphs show weak scaling for Summit(left) and DCC(right).

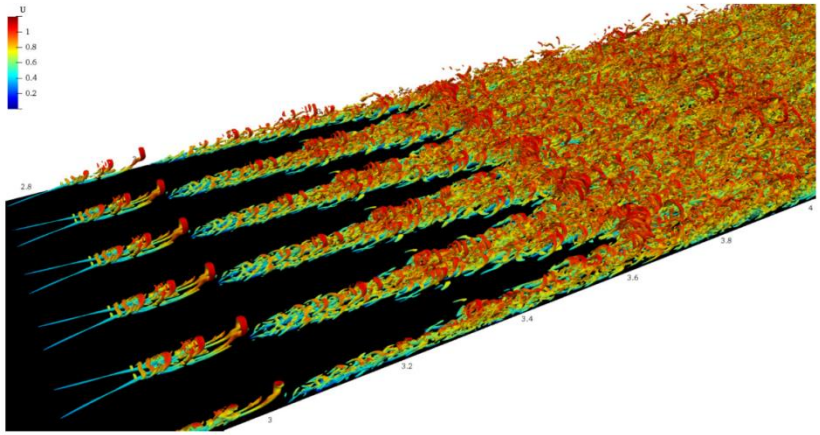


Multi-GPU Direct Numerical Simulations – Sanghyun et al., 2021

- GPU-accelerated fractional-step integration of incompressible Navier-Stokes equations.
- DNS of canonical zero-pressure gradient turbulent boundary layer(1.1B) and DNS of K-type boundary layer transition(1.4B) were simulated.



Weak scaling of the flow solver on different GPUs. A straight horizontal line corresponds to an ideal scaling. The number of grid cells per GPU is fixed to 6.71 × 107



Iso-surfaces of the second invariant of the velocity gradient $Q = 3.07 \times 10^{-2} \left(\frac{\partial \bar{u}}{\partial y}\right)^2 |_w$ at $Re_x = 2.8 \times 10^5 - 4.4 \times 10^5$ in the transitional boundary layer. Surfaces are colored by instantaneous streamwise velocity.

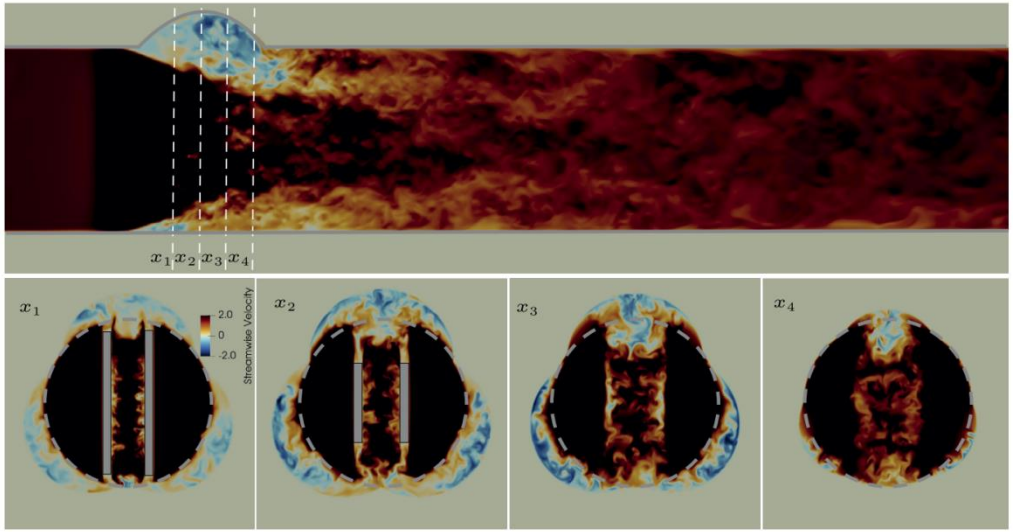
1.4B

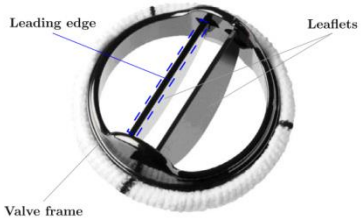
GPU name	# of GPUs	Grid cell dimension	Total grid cells
K80	2 x 8	3072 x 512 x1024	1.6B
P100	8	3600 x 512 x 768	1.4B
V100	8	3840 x 512 x 1296	2.5B

Maximum grid size supported by the code using 8 GPUs. The number of GPUs for K80 is shown as a multiple of 2, since each card contains two visible devices.

Multi-GPU Heart Valve Simulations – Zolfaghari and Obrist, 2021

- Multi-GPU code with successive over-relaxation Jacobi method for solving pressure Poisson equation.
- DNS of systolic flow past a bi-leaflet mechanical heart valve(BMHV).

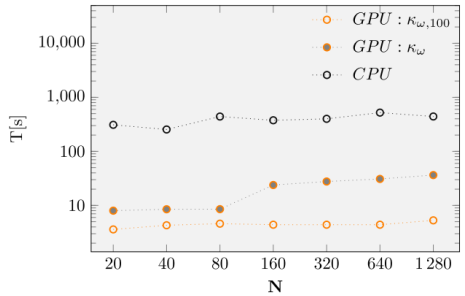




Leading edge view of Regent mechanical valve.(Above)

Fully resolved turbulent flow past an open BMHV. (Left)

337M

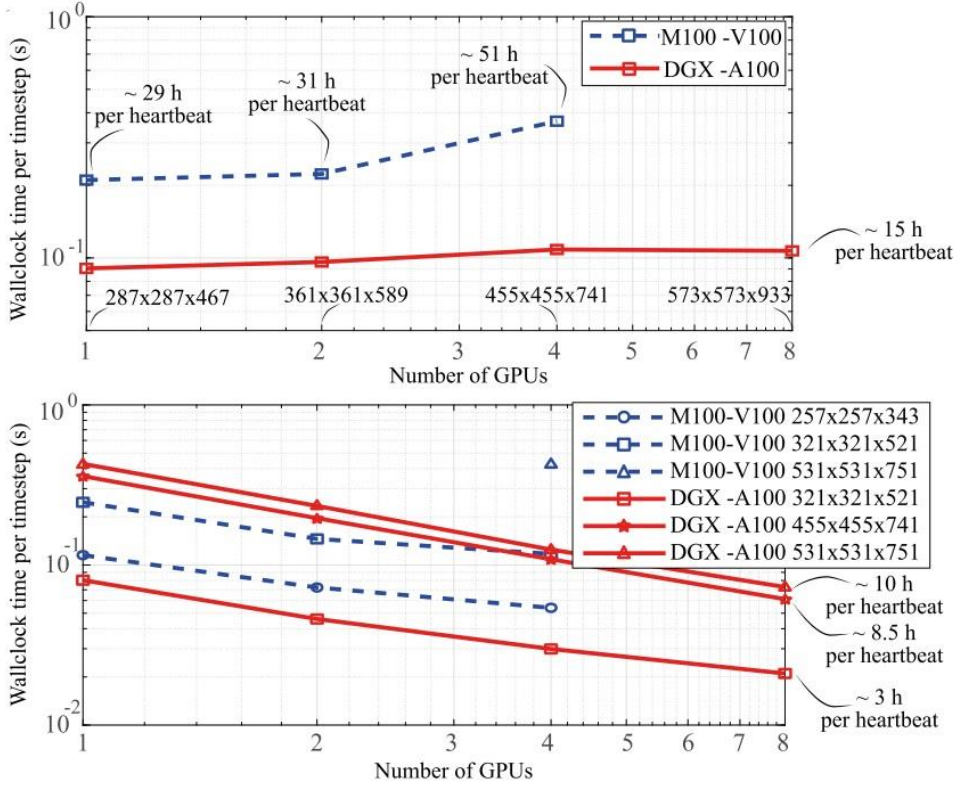


Weak scalability when the termination criterion is checked every 100 iterations (κ_{ω}, 100), compared against the scalability when norm checked in every iteration κ_{ω}.(Left)

Electrophysiology Interaction in the Left Heart (FSI)- , Francesco, 2022

- Acceleration of FSEI code using CUDA on heterogenous clusters, for faster results using relatively small computing facility with few GPU cards
- Simulation performed on both A100 and V100 cards.

223M



Weak (Top) and strong (Bottom) scaling performance of the GPU accelerated code running on one node of Marconi100 (blue dashed curves) having V100 cards and on a DGX machine (red solid curves) with A100 cards. Various combinations of grid points are tested, and the amount of time for a whole heartbeat is also recorded for each configuration.

Conclusions

Application cases of GPUs in large scale engineering/scientific problems
Hybrid parallelization and exascale computing applications