GPGPU and CUDA

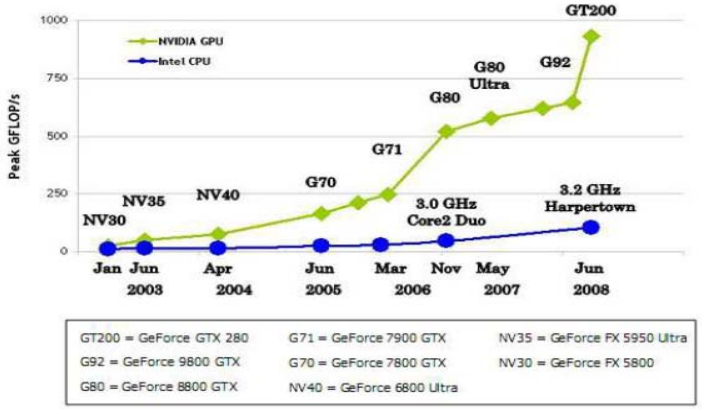
An Introduction

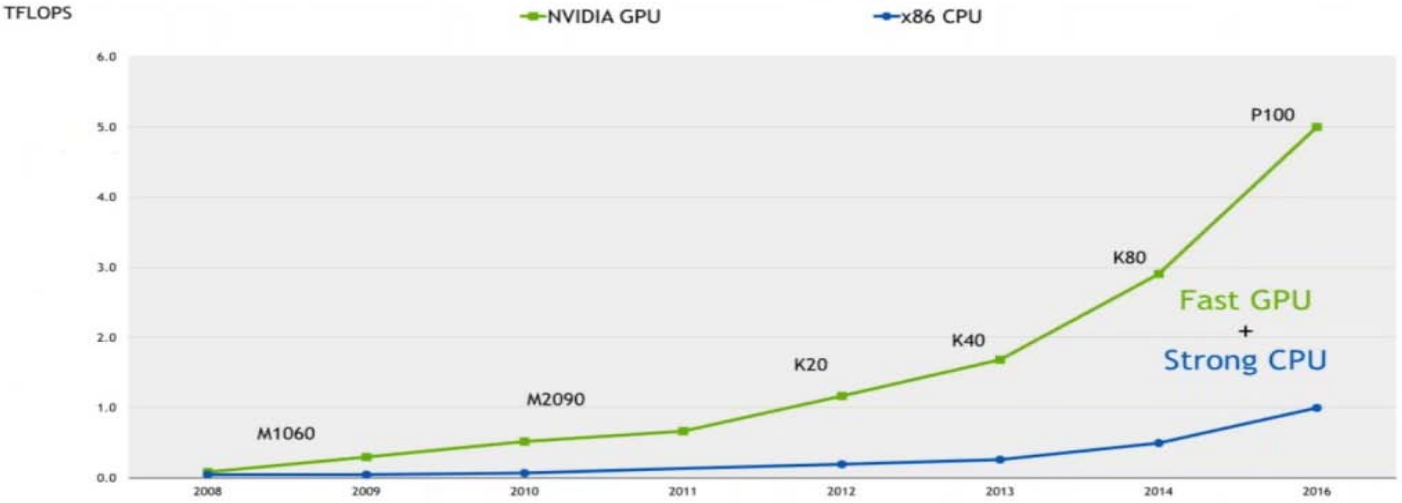
GPU Computing

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

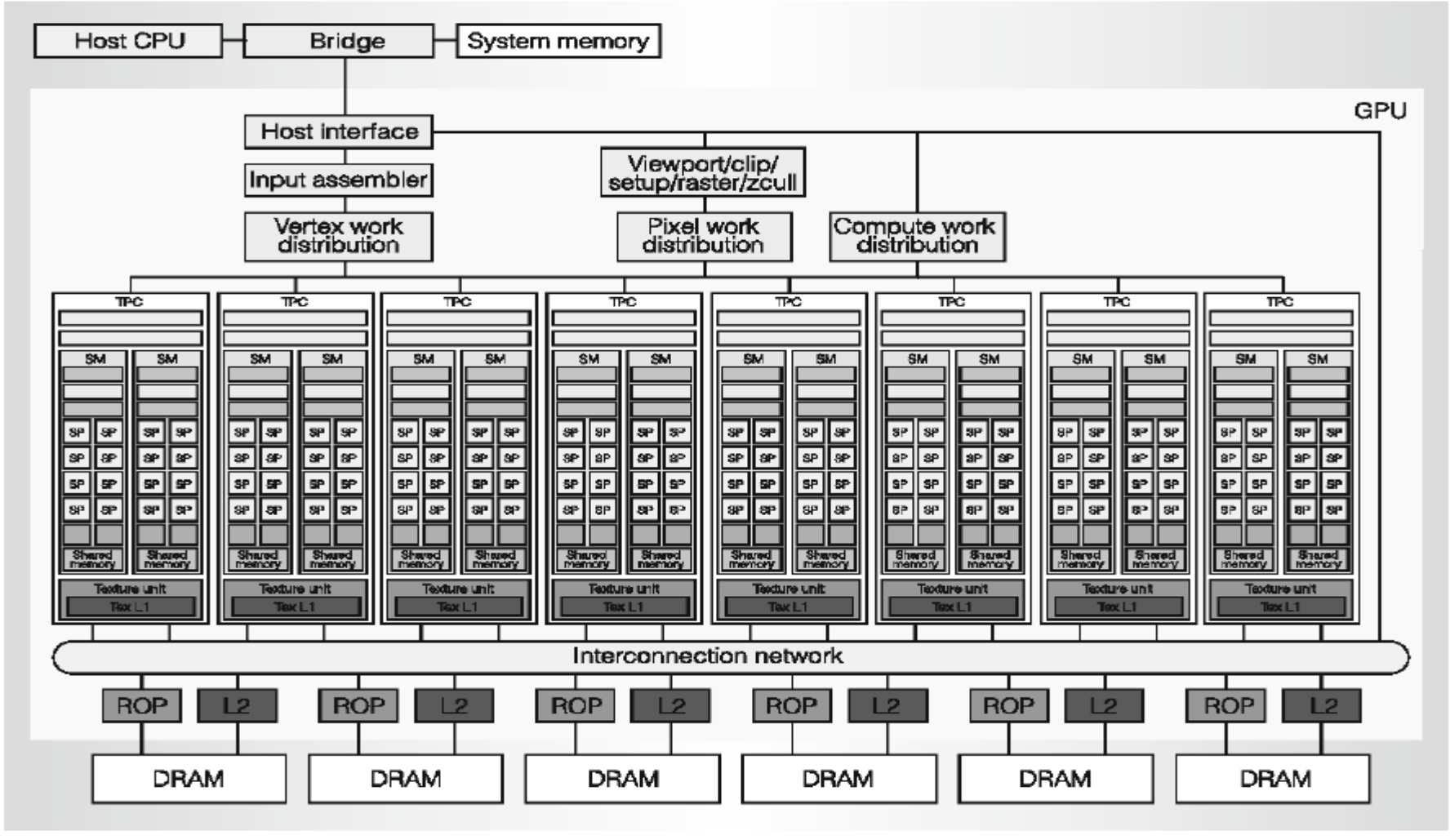
Around 2000, general purpose GPGPU-s developed

Efficient in matrix calculation





GPU -vs- CPU performance, NVIDIA

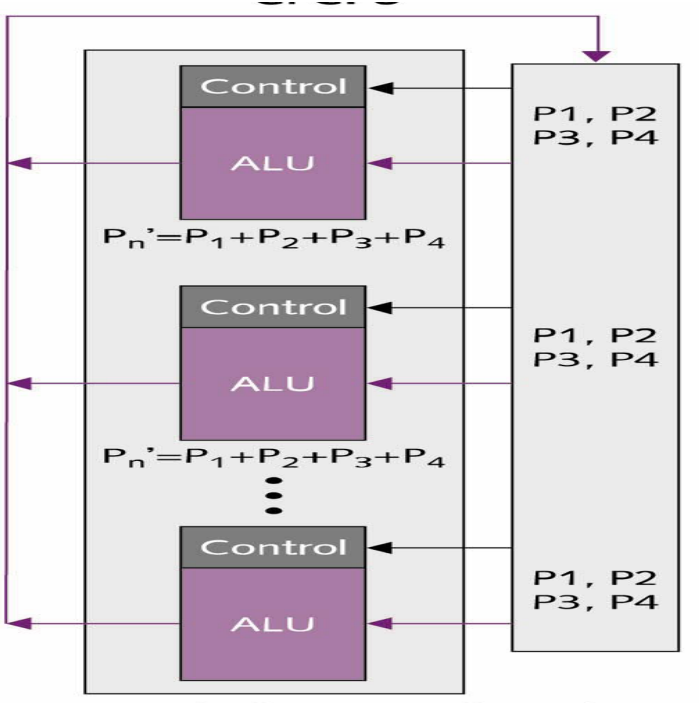

Host CPU
Bridge
System memory
GPU
Host interface
Input assembler
Vertex work distribution
Viewport/clip/ setup/raster/zcull
Pixel work distribution
Compute work distribution
TPC
SM
SP
Shared memory
Texture unit
Tex L1
Interconnection network
ROP
L2
DRAM

GPU Hardware

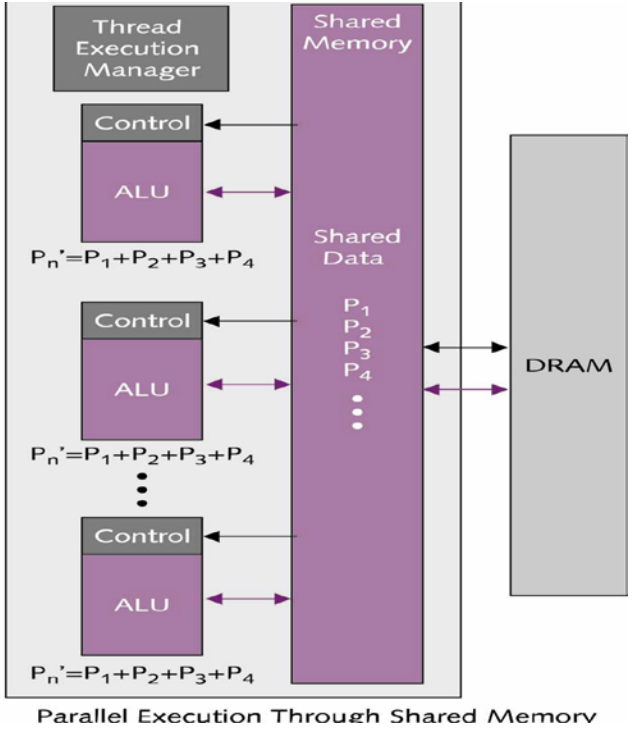
Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	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

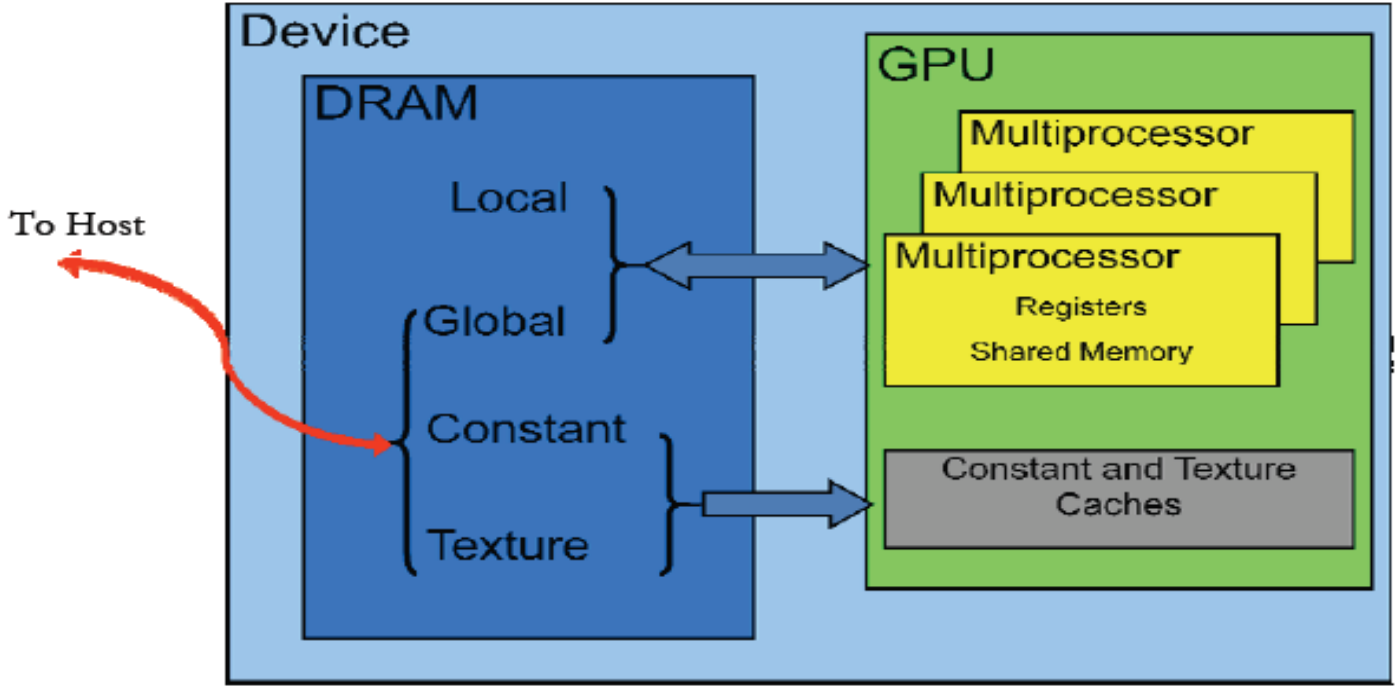
Traditional GPGPU



GPGPU Using CUDA



MEMORIES


Device
DRAM
Local
Global
Constant
Texture
GPU
Multiprocessor
Multiprocessor
Multiprocessor
Registers
Shared Memory
Constant and Texture Caches
To Host

Features of CUDA memories

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	No	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	No	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

Terminology

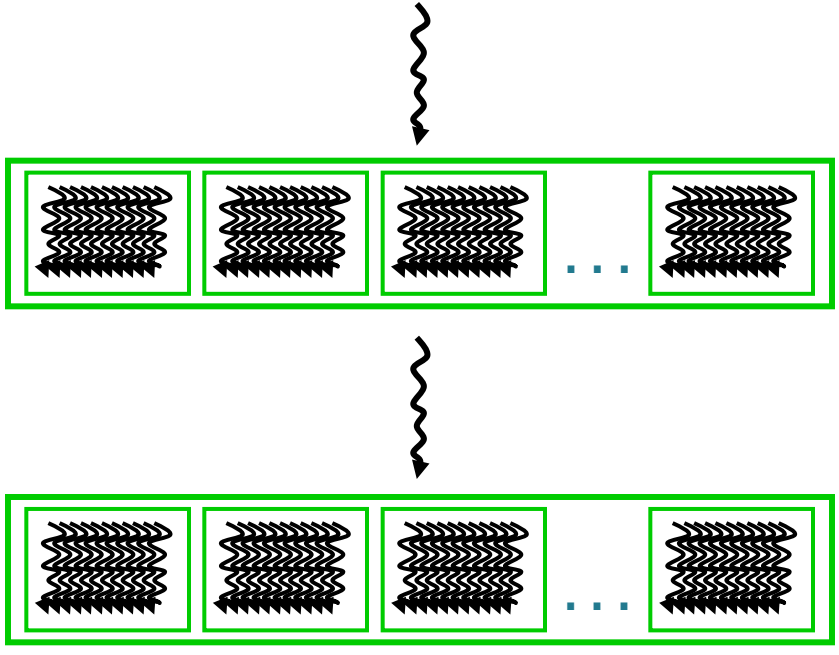
- ✓Device - GPU
- ✓Host – Hosting CPU
- ✓Kernel – Part of the code that runs in GPU
- ✓Global memory - data available to all threads and can be copied directly to host
- ✓DRAM- Device or GPU RAM
- ✓SRAM – on chip shared RAM
- ✓Memory bandwidth

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

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);



Serial Code (host)

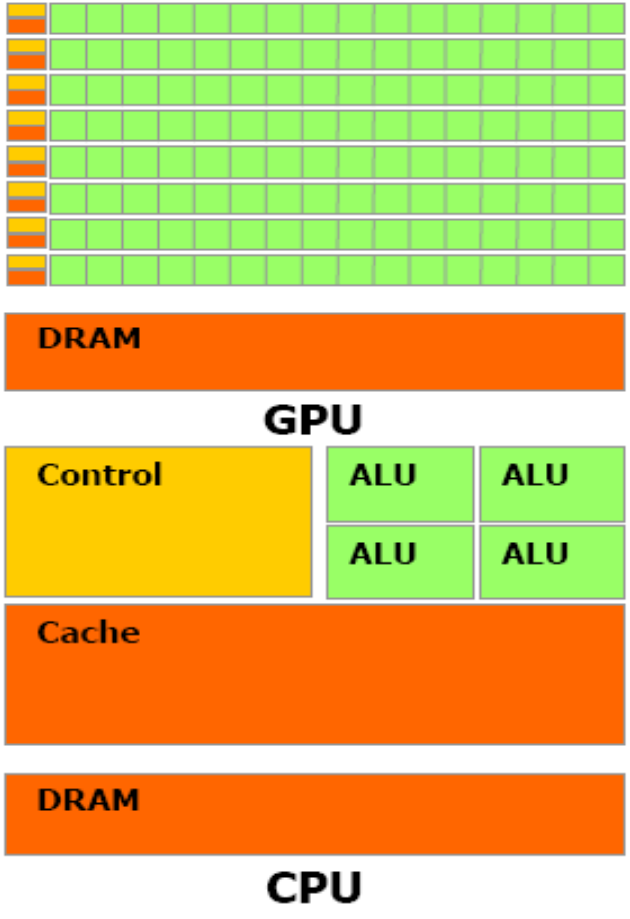
Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);

GPU Versus CPU

✓Modern GPUs contain hundreds of arithmetic units, and their power can be used to accelerate a lot of compute-intensive applications.

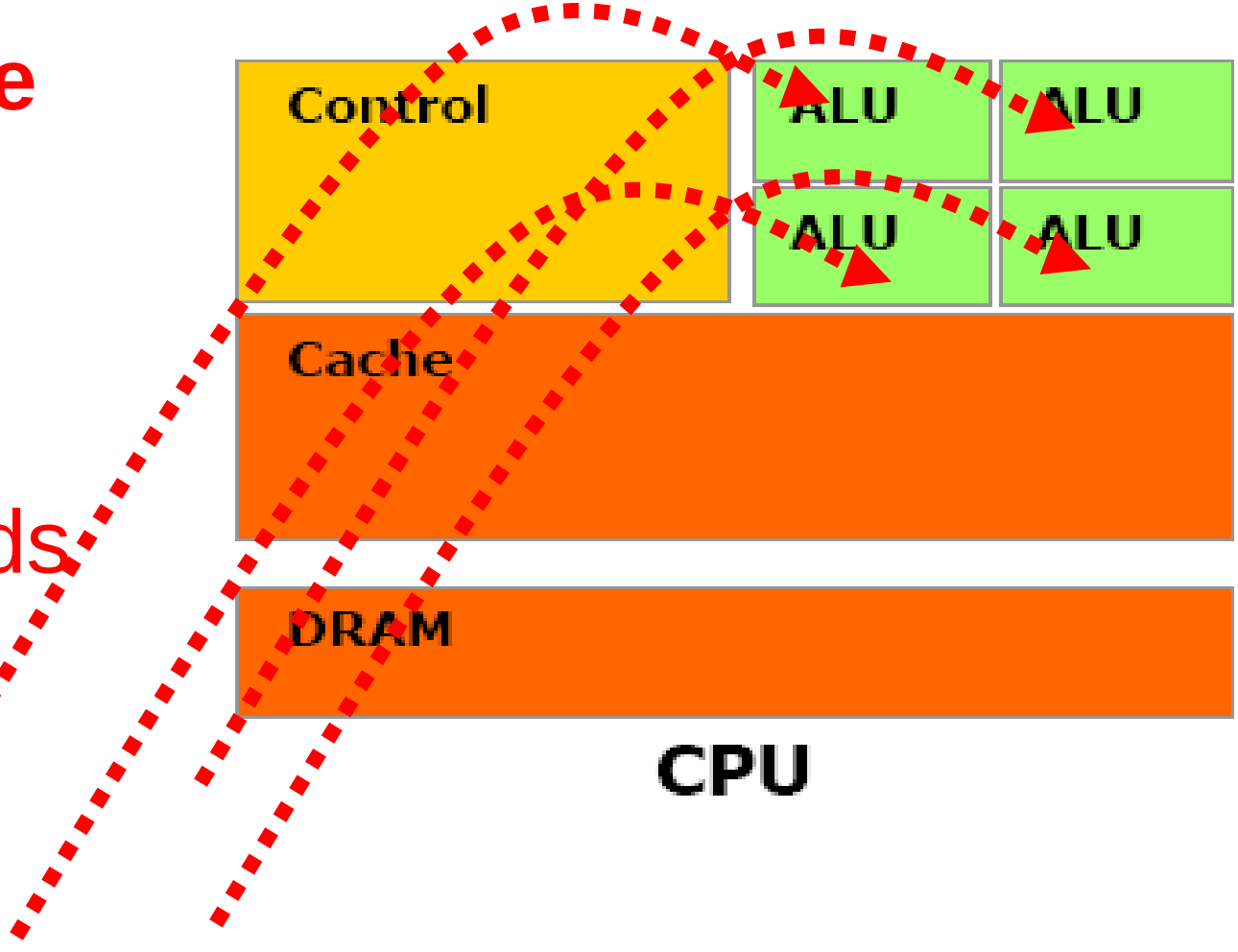
✓The existing generation of GPUs have a flexible architecture.

✓Most of the transistors in CPU are dedicated for data caching and controlling

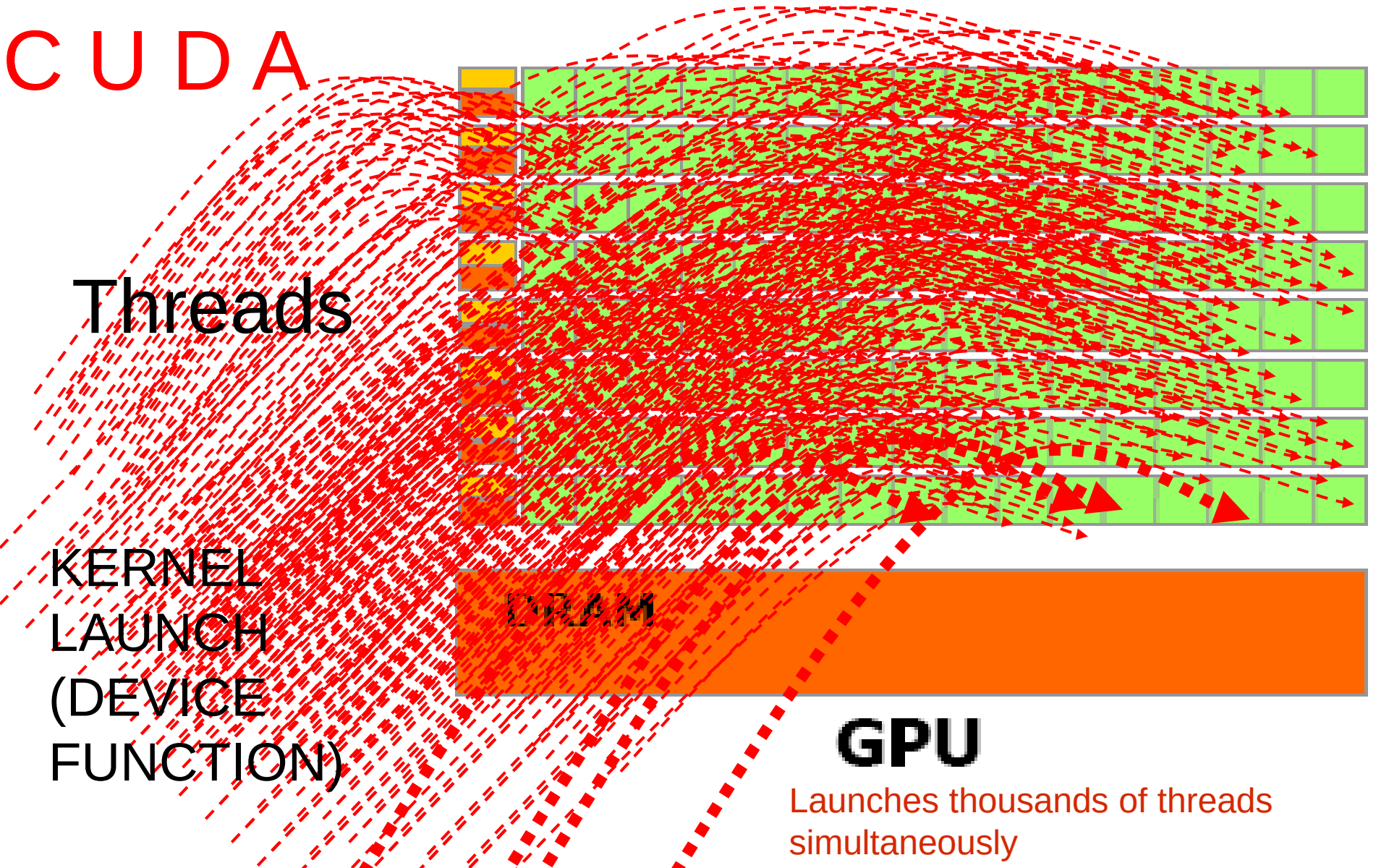


Writing Code on Multi-core Processor

Threads



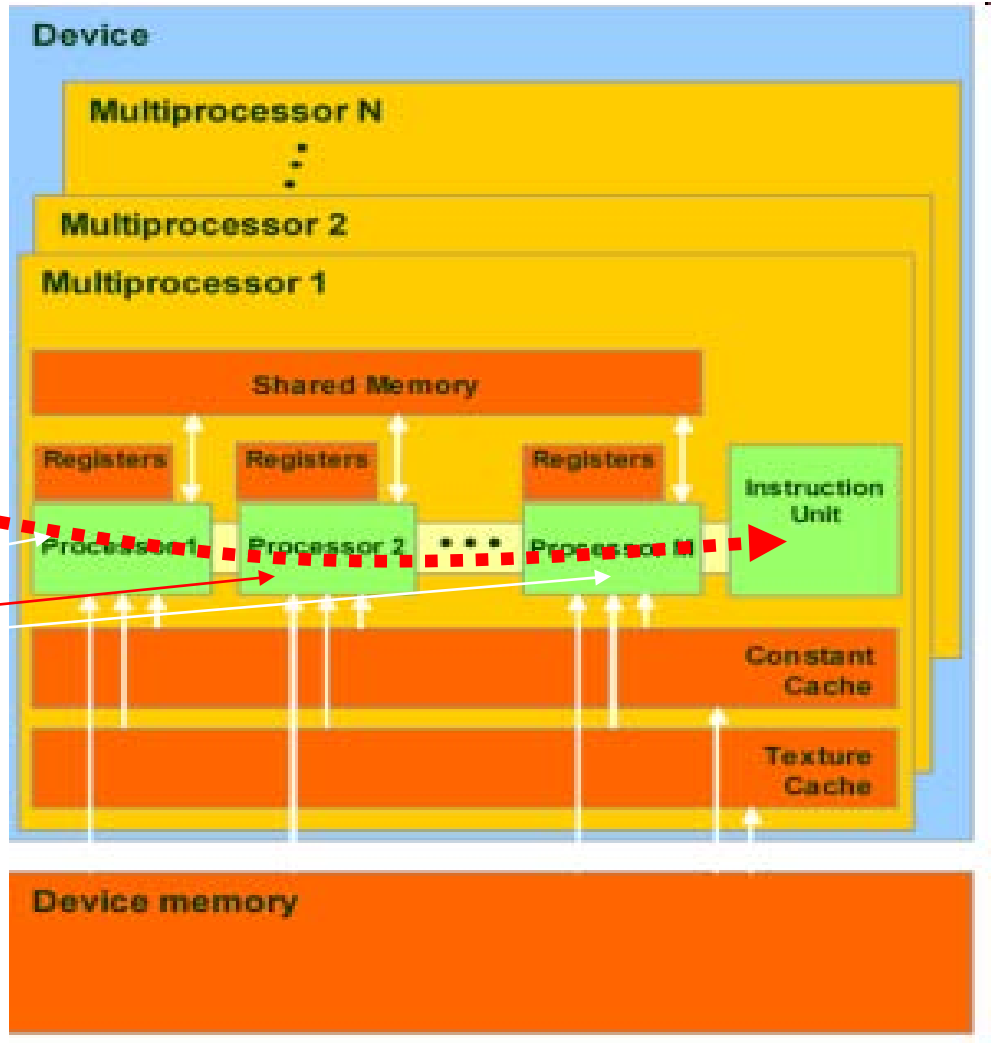
CUDA


Threads
KERNEL LAUNCH (DEVICE FUNCTION)
GPU
Launches thousands of threads simultaneously

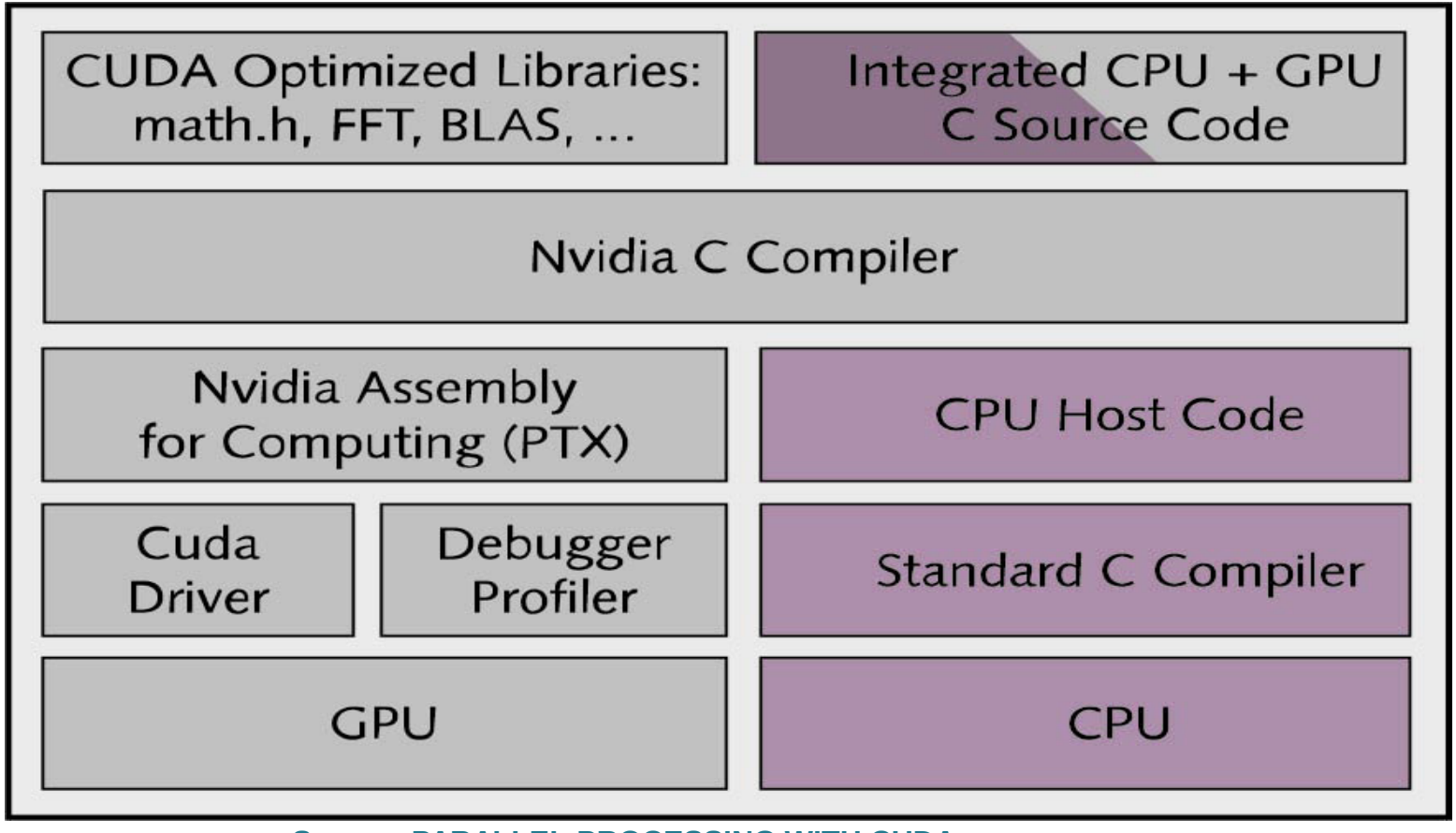
S I M D!

Single Instruction
(Issued by Instruction Unit)

Multiple Data
(e.g. fetched from
Shared Memory)



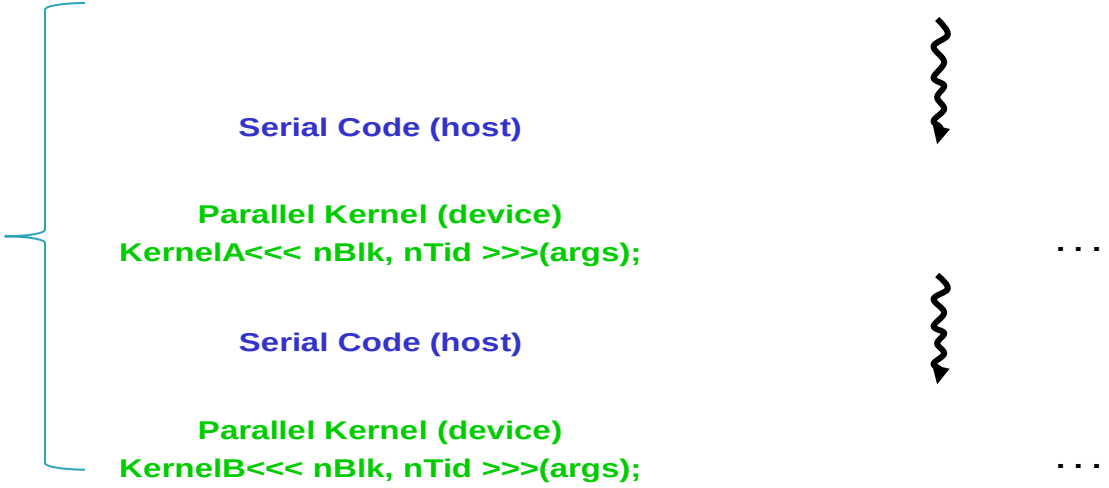
CUDA platform for parallel processing on Nvidia GPUs.



Source: PARALLEL PROCESSING WITH CUDA
Nvidia's High-Performance Computing Platform Uses Massive Multithreading By Tom R.Halfhill {01/28/08-01}

- Integrated host+device app C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code

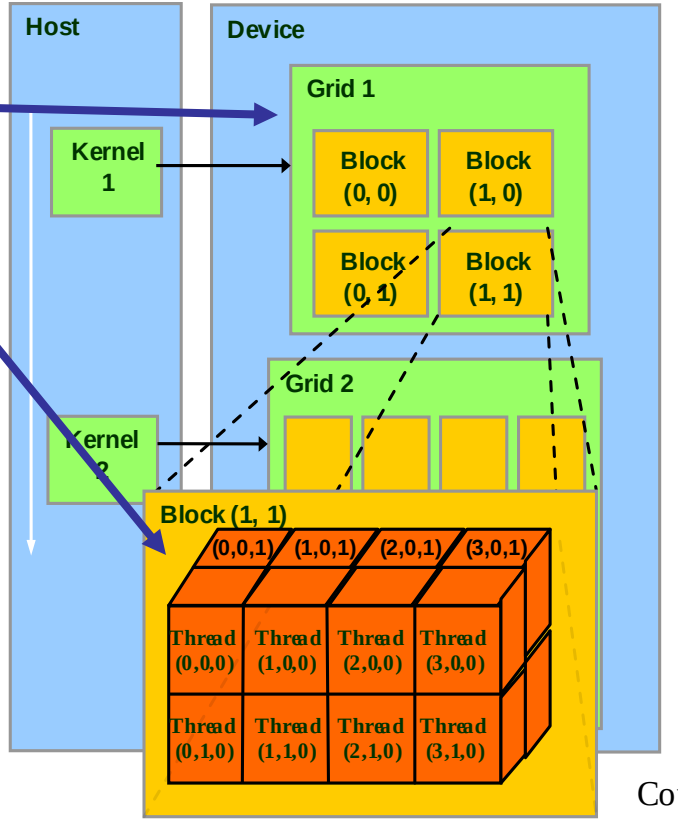
CUDA launches threads in blocks and grids



Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - Matrix, 2D and 3D arrays



Courtesy: NDVIA

Thread Block Algebra

- *Thread*: concurrent code and associated state executed on the CUDA device (in parallel with other threads)
  - The unit of parallelism in CUDA

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

Hardware limits

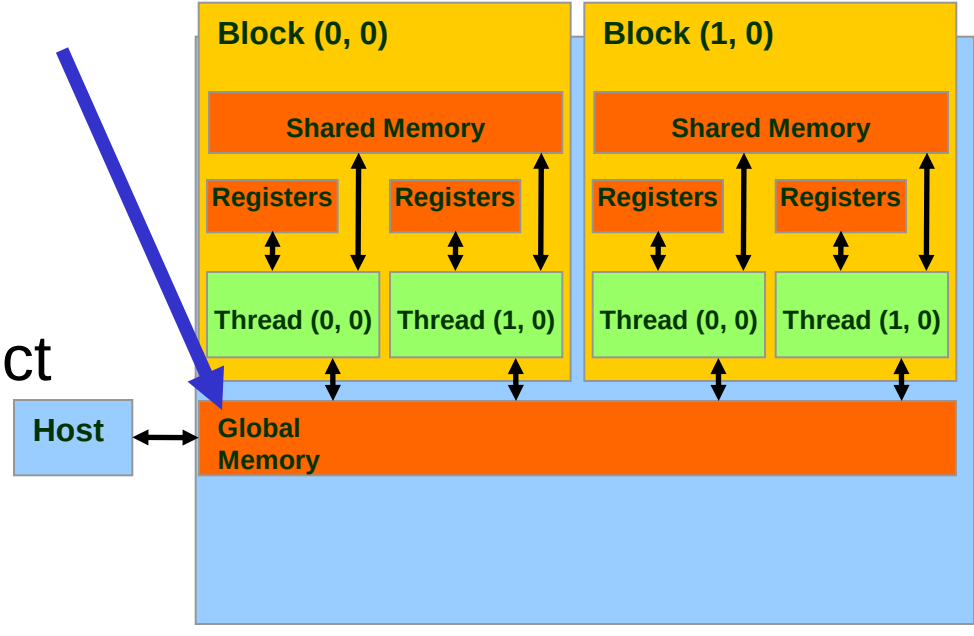
- *Block*: a group of threads that are executed together and form the unit of resource assignment –

Programming specification

- *Grid*: a group of thread blocks that must all complete before the next kernel call of the program can take effect

Essentials-- CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



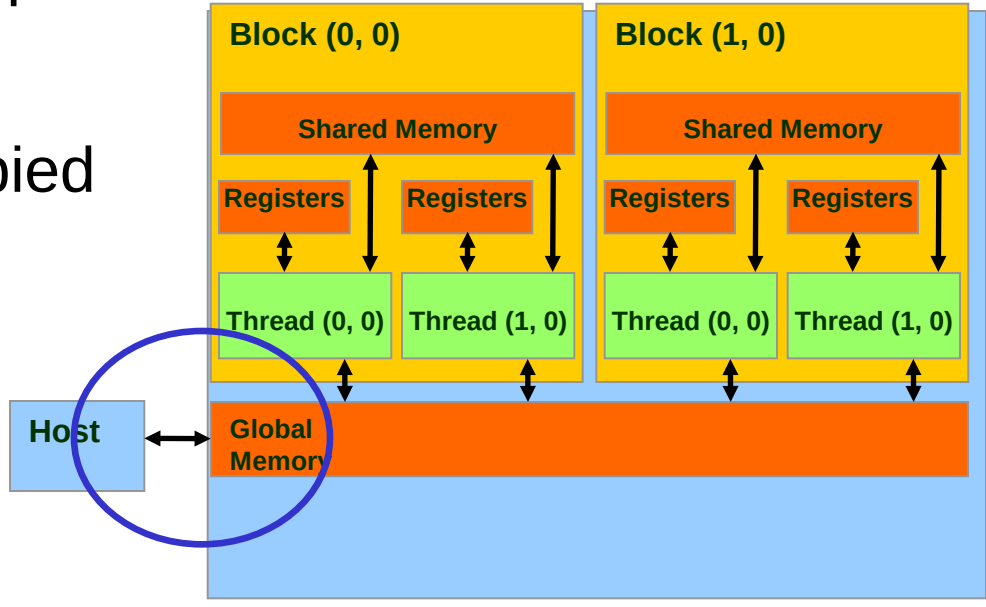
CUDA Device Memory Allocation (2)

- Code example:
  - Allocate a 32* 32 single precision float array
  - Attach the allocated storage to Md
  - “d” is often used to indicate a device data structure (“h” is used to indicate host)

```
TILE_WIDTH = 32;
Float* Md
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Device
      - Device to Host
      - Device to Device
      - Host to Host
- Asynchronous transfer



CUDA Host-Device Data Transfer (2)

- Code example:
  - Transfer a 32 * 32 single precision float array
  - Mh is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

```
cudaMemcpy(Md, Mh, size,cudaMemcpyHostToDevice);
cudaMemcpy(Mh, Md, size,cudaMemcpyDeviceToHost);
```

C language extension - Operators in Device and Host

Function type qualifiers (where to call and execute a function): __device__, __global__,__host__

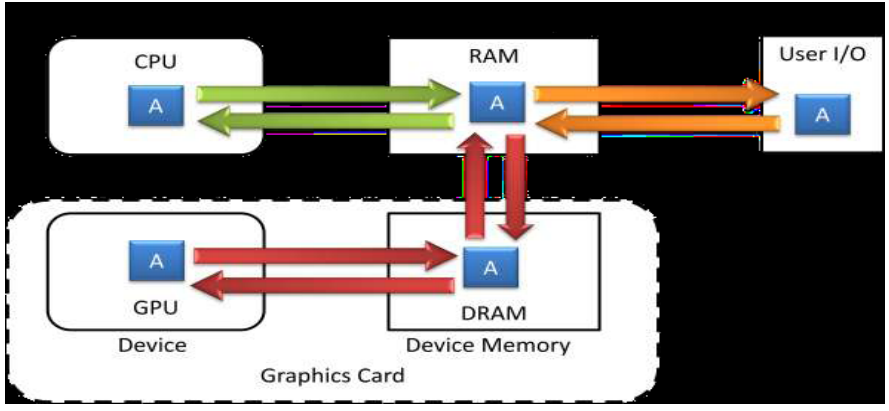
Variable type qualifiers (__device__, __constant__ and __shared__)

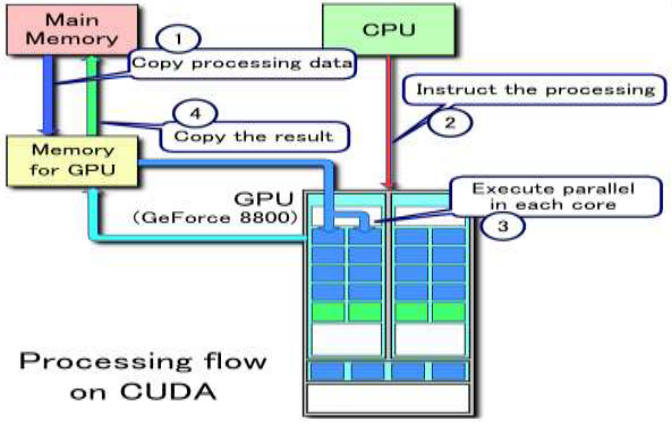
Kernel execution directive (foo<<...>>(...))

Built-in variables for grid/block size and block/thread indices

- `__global__` defines a kernel function
  - Must return `void`

CUDA Programming Model





Compute Unified Device Architecture

Threads are executed to do parts of job

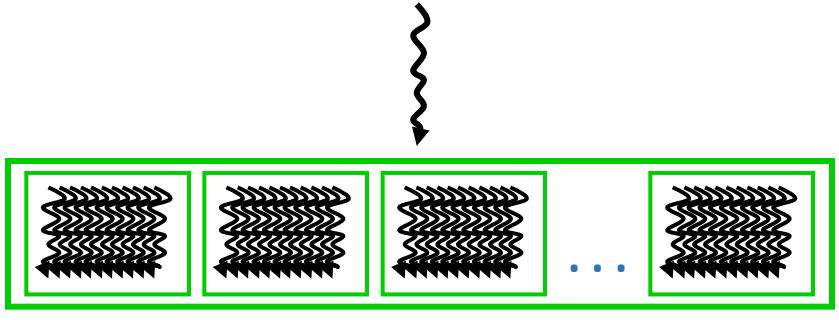
CUDA Programming Model

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)

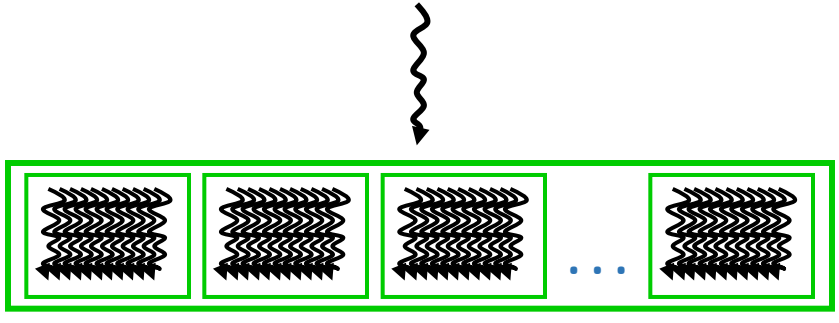
KernelA<<< nBlk, nTid >>>(args);



Serial Code (host)

Parallel Kernel (device)

KernelB<<< nBlk, nTid >>>(args);



Compiling a CUDA Program

Use the following command:

#: nvcc VectorAdd.cu –o VectorAdd

#: ./ VectorAdd

nvcc is the compiler while “./” executes the binary

Any CUDA program broadly consist of the following components:

1) Include header files

2) **Kernel that executes on the CUDA device, e.g:**

*//*__global__ void MatrixMulKernel(float *Md, float *Nd, float *Pd, int Width)

3) **main( ) routine, the CPU must find.**

3.1:- Define pointer to host and device arrays

3.2:- Define other variables used in the program

e.g. arrays etc.

3.3:- Allocate array on the host

/e.g. a_h=(float*)malloc(size)

3.4:- Allocate array on device (DRAM of the GPU)

/e.g. cudaMalloc ((void**) (a_d,size))

3.5:- Copy the data from host array to device array.

// cudaMemcpy(Md_d,Md_h,size,cudaMemcpyHostToDevice);

3.6:- Kernel Call, Execution Configuration // e.g add_array<<<n block,p size>>>(…..)

3.7:- Retrieve result from device to host in the host memory, e.g; cudaMemcpy(Pd_h,Pd_d,size,cudaMemcpyDeviceToHost);

3.8:- Print result // for (i=0,……)

printf("%f ",,a_h[i]) ;

3.9:- Free allocated device and host memories //

e.g free(a_h);
cudaFree(a_d);

Using the above programming steps, the following program calculates and

prints the square of first 1000 integers.

```
// 1) Include header files
#include <stdio.h>
#include <cuda.h>
#include <conio.h>
// 2) Kernel that executes on the CUDA device
__global__ void square_array(float*a,int N)
{
int idx=blockIdx.x*blockDim.x+threadIdx.x;
if(idx<N)a[idx]=a[idx]*a[idx];
}
// 3) main( ) routine, the CPU must find
int main(void)
{
// 3.1:- Define pointer to host and device arrays
float*a_h,*a_d;
// 3.2:- Define other variables used in the program e.g. arrays etc.
const int N=100;
size_t size=N*sizeof(float);
```

```
// 3.3:-   Allocate array on the host
          a_h=(float*)malloc(size);
// 3.4:-    Allocate array on device (DRAM of the GPU)
           cudaMalloc((void**)&a_d,size);
       for(int i=0;i<N;i++)a_h[i]=(float)i;
// 3.5:-  Copy the data from host array to device array.
       cudaMemcpy(a_d,a_h,size,cudaMemcpyHostToDevice
// 3.6:-   Kernel Call, Execution Configuration
           int block_size=4;
       int n_blocks=N/block_size+(N%block_size==0);
       square_array<<<n_blocks,block_size>>>(a_d,N);
// 3.7:-    Retrieve result from device to host in the host memory, e.g;
           cudaMemcpy(a_h,a_d,sizeof(float)*N,cudaMemcpyDeviceToHost
// 3.8:-    Print result
            for(int i=0;i<N;i++)
         printf("%d\t%f\n",i,a_h[i]);
// 3.9:-    Free allocated memories on the device and host
        free(a_h);
        cudaFree(a_d);
         getch();
}
```

GPU Computing in Computational Mechanics Problem

CUDA (Compute Unified Device Architecture) coupled over basic compilers (C, Fortran)

Uses a SIMD (single Instruction Multiple Data) model with multiple threads

In-built synchronization – excellent scalability

Hybrid (shared + distributed) memory management

Domain decomposition and parallel computing can be done on top of GPU processing --(two levels of parallelization)

Speed

Iterative solver	GPU	Multi-core CPU	Speed-up
RB-GS	642.1 s	1894.7 s	2.95
RB-SOR	233.9 s	806.16 s	3.45

Comparison with 32 Core CPU and P100 GPU for 1000 times solution of a 144000 x 144000 matrix

Code Profiling and Performance Optimization using OpenACC in GPU

73.7	500	ceqcp
13.2	1	immersedtag2
5.7	89573	pbc
5.6	500	nseqcp

CPU Profiling of the IBM code

	Time (s)			
%time	total	CPU	GPU	func name
43.8	45.7	3.57	41.8	ceqcp
22.4	23.4	-	23.4	memcpy (host↔device)
20.9	21.8	21.8	-	immersedforcing2
3.5	3.6	3.6	-	immersedforcing3

CPU-GPU Profiling of the IBM code, with MAC in GPU and SOLA in CPU

	Time (s)			
%time	total	CPU	GPU	func name
43.8	45.7	3.57	41.8	ceqcp
22.4	23.4	-	23.4	memcpy (host↔device)
20.9	21.8	21.8	-	immersedforcing2
3.5	3.6	3.6	-	immersedforcing3

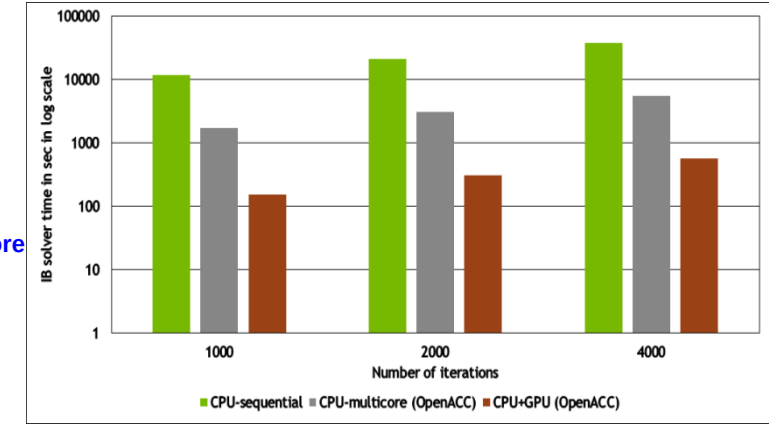
CPU-GPU Profiling of the IBM code after moving most of the subroutines to GPU

```
do 80 i=2,iloop
do 80 j=2,jre
do 80 k=2,kre
   if(cell(i,j,k).eq.0)then
      //compute dudt,dvdt,dwdt
      dact=dmax1(dudt,dvdt,dwdt)
      if(dact.gt.dtmax) then
         idtm=i
         jdtm=j
         kdtm=k
         dtmax=dact
      endif
   endif
80 continue
```

```
!$acc parallel loop collapse(3) reduction(max:dtmax)...
...
      dact=dmax1(dudt,dvdt,dwdt)
      if(dact.gt.dtmax) then
         dtmax = dact
      endif
   ...
!$acc end parallel
!$acc parallel loop collapse(3) reduction(min:idx)...
   ...
      dact=dmax1(dudt,dvdt,dwdt)
      if(dact.eq.dtmax) then
        index = k*jre*iloop + j*iloop + i
        if(index.lt.idx) then
           idx = index
        endif
      endif
...
!$acc end parallel
      kdtm=dtmaxindex/(jre*iloop)
      jdtm=(dtmaxindex - (kdtm*jre*iloop))/iloop
      idtm=dtmaxindex - (kdtm*jre*iloop) - (jdtm*iloop)
```

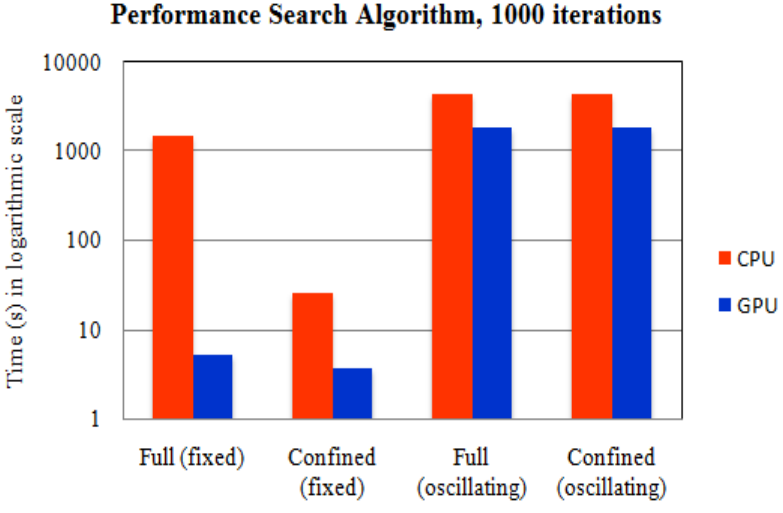
Speed-up:

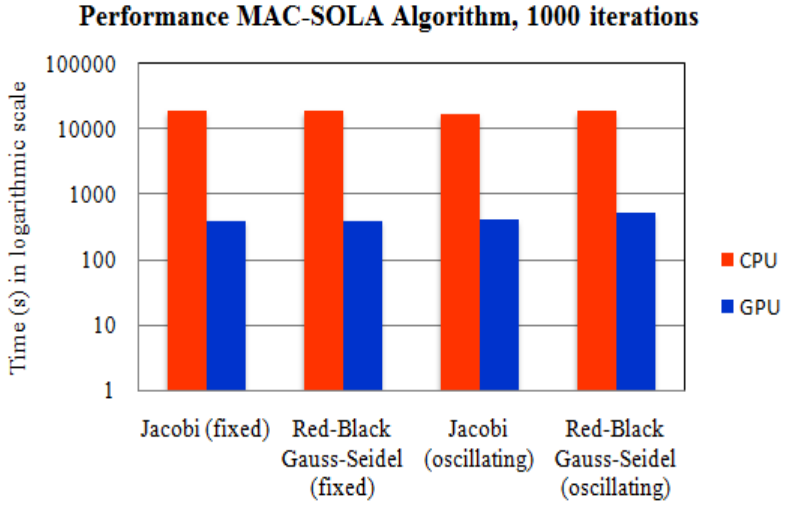
70x with single processor
10 x with multi(16) core



Flow over fixed cylinder

Performance Enhancement





Moving boundary case- 3 million cells
10x speed-up in search

40x speed-up in solver

Performance results for large structured matrices (septadiagonal)

size	1000	5000	10000	20000
serial	1.48	59.29	236.7	979.88
parallel	0.41	1.61	6.07	30.23
shared	0.49	1.24	4.11	20.8

Jacobi solver for different memory optimization

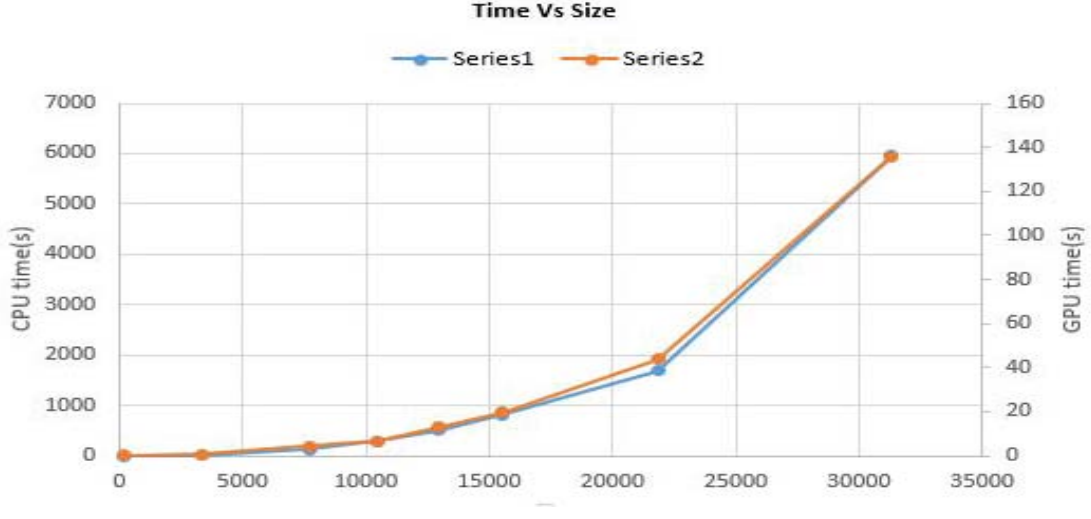
Size	2000	10000	100000	1E+06	2E+06	5E+06
Serial	0.72	0.84	1.45	160.4	473.5	1367.8
Parallel	0.23	0.25	0.43	22.32	90.45	176.61
Speedup	3.13	3.36	3.37	7.18	5.23	7.74

Performance of BiCGSTAB solver

Performance results for unstructured matrices

size	BiCGstab serial	Jacobi serial	BiCGstab parallel	Jacobi parallel
144	0.021	0.029	0.019	0.066
3310	27.91	792.74	0.62	18.01
7705	125.4	9482.46	3.25	187.24

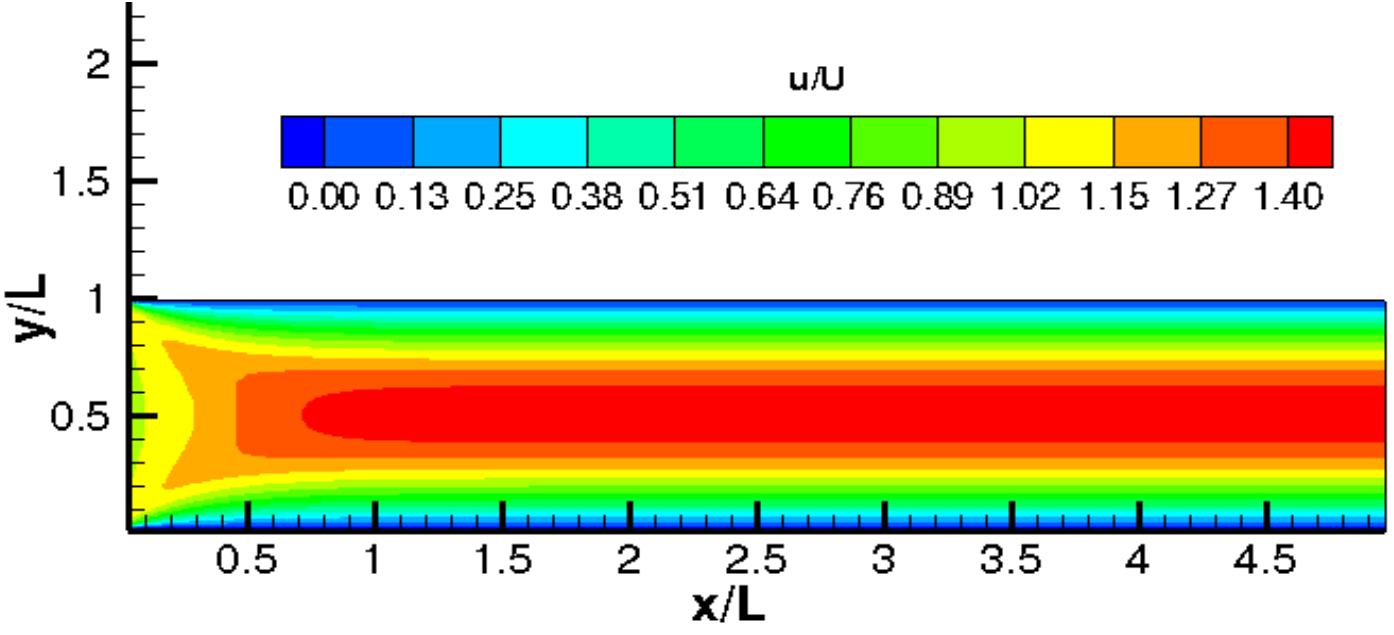
Performance of different solvers



Convergence for different matrix sizes

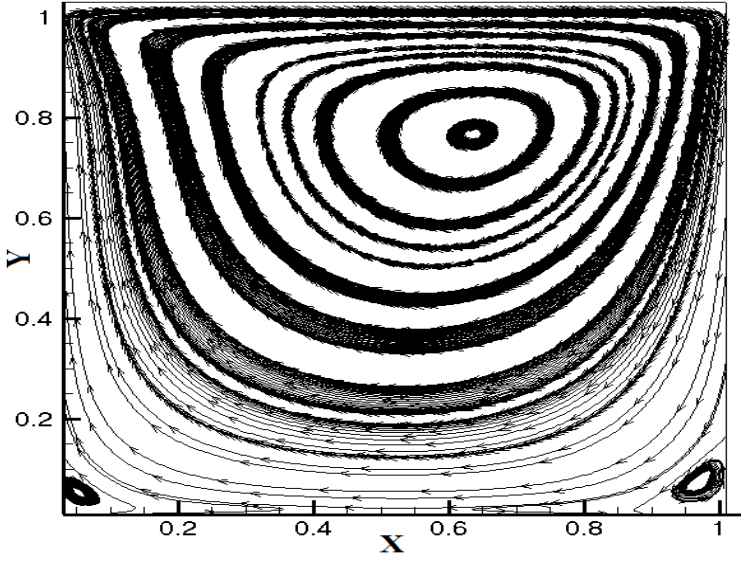
GPU Computing for Channel Flow

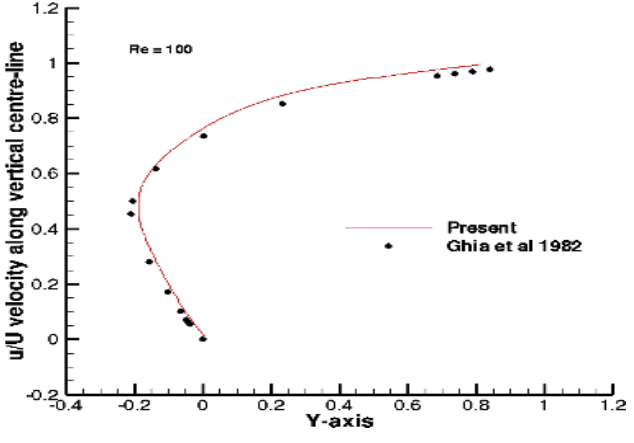
Case	Step Size	Length	Width	Re	Mesh Size	Error	Time Elapsed (s)
Serial	0.02	5	1	25	250 X 50	10^{-4}	**1140.56**
Parallel	0.02	5	1	25	250 X 50	10^{-4}	**285.54**



Lid Driven Cavity

Case	Step Size	Length	Width	Re	Mesh Size	Error	Time Elapsed (s)
Serial	0.02	1	1	100	50 X 50	10^{-4}	**855.05**
Parallel	0.02	1	1	100	50 X 50	10^{-4}	**114.42**





Agarwal, S., Kumar, M., and Roy, S, Demonstration of GPGPU Accelerated Computational Fluid Dynamics Calculations, Intelligent Computing and Applications, Springer India, 2015

References

1- NVIDIA CUDA Programming Guide

2- Programming Massively Parallel Processors, By David Kirk, NVIDIA Fellow, Wen-mei Hwu, Professor, University of Illinois

3- Course EC498, University of Illinois

http://courses.engr.illinois.edu/ece498/al/syllabus.html

4- David A. Patterson, The Landscape of Parallel Computing Research: A View from Berkeley

5- PARALLEL PROCESSING WITH CUDA, Nvidia's High-Performance Computing Platform Uses Massive Multithreading By Tom R.Halfhill Published in Microprocessor Report, {01/28/08-01}
http://www.nvidia.com/docs/IO/47906/220401_Reprint.pdf

6-Paweł Macioł, Krzysztof Bana´s, Testing Tesla Architecture for Scientific Computing: the Performance of Matrix-Vector Product. Proceedings of the International Multiconference on Computer Science and Information Technology pp. 285–291