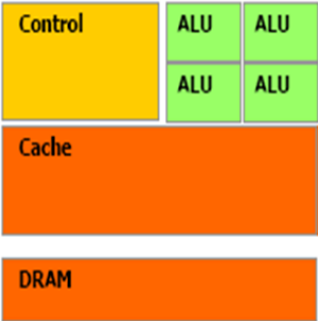
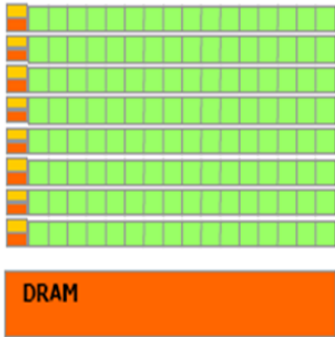


Assignment: Introduction to CUDA

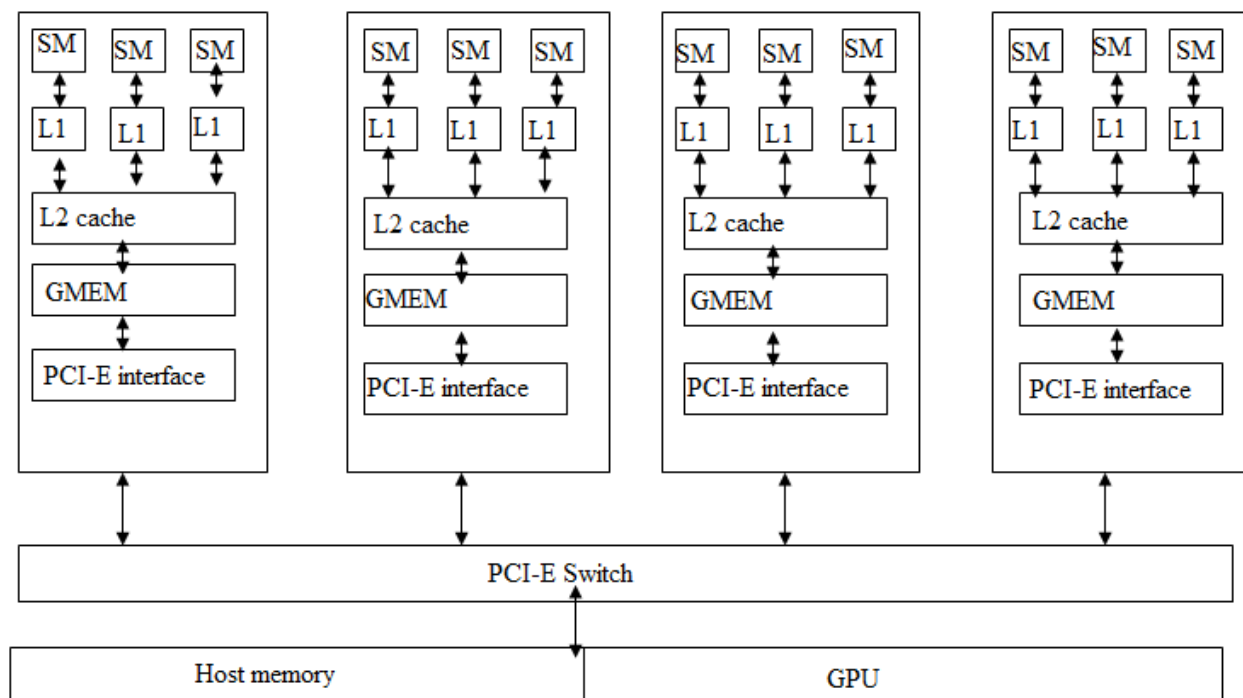
1. CUDA [Compute Unified Device Architecture]:

- CUDA is the hardware and software architecture that enables NVIDIA GPU's to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages.
- A CUDA program calls parallel kernels. A kernel executes in parallel across a set of parallel threads.
- The programmer or compiler organizes these threads in thread blocks and grids of thread blocks.
- The GPU instantiates a kernel program on a grid of parallel thread blocks.
- Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results.
- Heterogeneous serial-parallel computing, Scalable programming model, C for CUDA – extension to C

CPU	GPU
Central Processing Unit.	Graphics Processing Unit.
CPU has multiple cores.	GPU has thousands of cores.
Less number of ALU's	More number of ALU's
Less efficient	More efficient than CPU as processing of large blocks of data is done in parallel
 <p>The diagram illustrates the CPU architecture. It features a yellow 'Control' block at the top left, followed by four green 'ALU' blocks arranged in a 2x2 grid. Below these is a large orange 'Cache' block, and at the bottom is an orange 'DRAM' block. The entire structure is labeled 'CPU' at the bottom.</p>	 <p>The diagram illustrates the GPU architecture. It shows a grid of many small green squares, each representing a processing unit, arranged in 8 rows and 16 columns. To the left of each row is a small orange square, likely representing a control or register unit. Below the grid is a large orange 'DRAM' block. The entire structure is labeled 'GPU' at the bottom.</p>
Optimized for low latency access to cached data set	Optimized for data parallel throughput computation.

GPU Hardware:

- Both CPU and GPU consist of clusters of processors. Let us consider a simple example of CPU with 4 cores.
- Every node is associated with DRAM and local storage. This is considered as L1 cache. There is a network interface per node that connects the local storage and DRAM with network switch of the system.
- This network interface is L2 cache. The network switch is L3 cache which is intern connected to network storage which is again DRAM.
- The architecture of GPU is similar to CPU. There are number of streaming multiprocessors which can be considered as CPU cores.
- The SMs are connected to shared memory (L1 cache). L1 cache connected to L2 cache which acts as intern SM switch.
- Data is held in global memory storage. It is then extracted and either used by host or sent via PCI-E switch directly to the memory or another GPU. The PCI-E switch is faster than any other networks interconnect.
- The PCI interconnects CPU and GPU.
- This switch is required because not all instructions are executed on either GPU or CPU in any task.
- The control must switch over between the two. In general the instruction that are to be executed serially are executed on CPU, and those which can be executed in parallel, are on GPU.
- All the above combination forms a node. This node may be replicated many times ,which forms clu-ster.



Streaming Processor:

- The CUDA architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity.
- The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.
- Each SM contains 8 CUDA cores, and at any one time they're executing a single warp of 32 threads - so it takes 4 clock cycles to issue a single instruction for the whole warp.

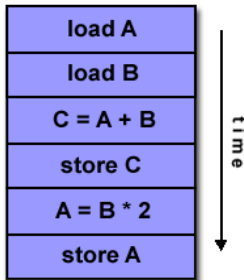
Flynn's Classification Model :

SISD Single Instruction, Single Data	SIMD Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

Single Instruction, Single Data (SISD):

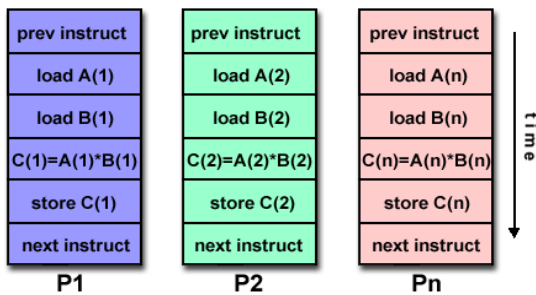
- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle

- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU workstations and mainframes



Single Instruction, Multiple Data (SIMD):

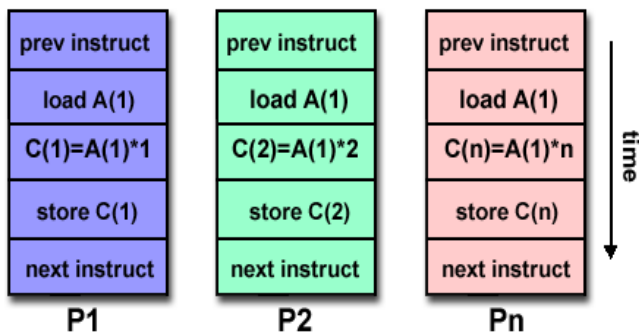
- A type of parallel computer
- Single instruction: All processing units execute the same instruction at any given clock cycle
- Multiple data: Each processing unit can operate on a different data element
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines



Multiple Instruction, Single Data (MISD):

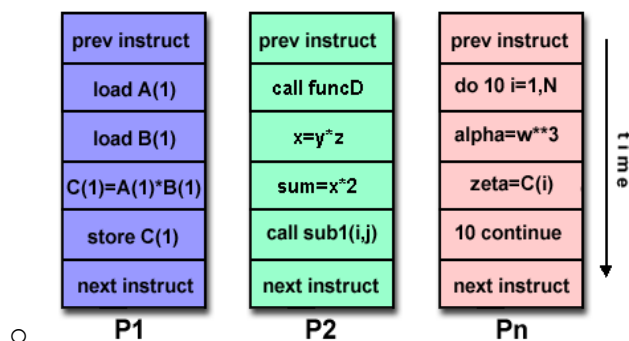
- A single data stream is fed into multiple processing units.

- Each processing unit operates on the data independently via independent instruction streams.
- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be:
 - multiple frequency filters operating on a single signal stream
- multiple cryptography algorithms attempting to crack a single coded message.



Multiple Instruction, Multiple Data (MIMD) :

- Currently, the most common type of parallel computer. Most modern computers fall into this category.
- Multiple Instruction: every processor may be executing a different instruction stream
- Multiple Data: every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.



Alternatives to CUDA:

OpenCL(Open Computing Language):

- OpenCL is an open and royalty free standard supported by NVIDIA, AMD .Its trademark is owned by Apple.
- It is low level API for heterogeneous computing that runs not only on CUDA powered GPUs but other GPUs which are not NVIDIA made.
- OpenCL provide parallel computing using task-based and data based parallelism
- It defines C like language for writing programs called Kernel, that executes on compute devices..
- A compute device consists of many processing elements. A single kernel execution runs on many PEs in parallel.

Direct Compute:

- Direct Compute is Microsoft made tool. It is proprietary product linked to the windows operating system, and in particular, the direct X11 API.
- It aims to accelerate PC applications performance in Microsoft Windows.
- It is basically targeted on following problems:
 - Image Processing.
 - Video Processing.
 - Games and artificial intelligence.

Understanding Parallelism:

OpenMP:

- The OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in [C/C++](#) and FORTRAN on much architecture, including UNIX and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
- OpenMP is a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer.

- An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI).
OpenMP: shared memory
MPI: distributed memory

MPI:

Message Passing Interface is the most well known standard used for coding parallel-computing applications, based on the message-passing programming paradigm.

- MPI-CH and Open-MPI are actual libraries that have implemented MPI functions in C and C++. However, Java programs cannot use these libraries directly, for which reason mpiJava was designed to bridge Java applications to the underlying MPI-CH functions.
- all mpiJava applications must import the following package:
 - `import mpi.*;`
- To start and stop mpiJava, the `main(String[] args)` function must call the following functions at the beginning and the end respectively:
 - `MPI.Init(args);`
 - `MPI.Finalize();`
- Between these function calls, you may use any mpiJava functions such as:
 - `MPI.COMM_WORLD.Bcast();` Broadcast an array to all processes.
 - `MPI.COMM_WORLD.Send();` Send an array to a destination process.
 - `MPI.COMM_WORLD.Recv();` Receive an array sent from a source process.
- In mpiJava, each process engaged in the same computation is identified with a rank (starting from 0.) You may use the rank 0 process as a master and the other processes as a worker. You can find each process's rank information and the number of processes engaged in the computation through:
 - `MPI.COMM_WORLD.Rank();`
 - `MPI.COMM_WORLD.Size();`

2. CUDA Memory Hierarchy

- Thread:
 - Registers
 - Local memory
- Block of threads:
 - Shared memory
- All blocks:
 - Global memory



Grid of Thread Blocks:

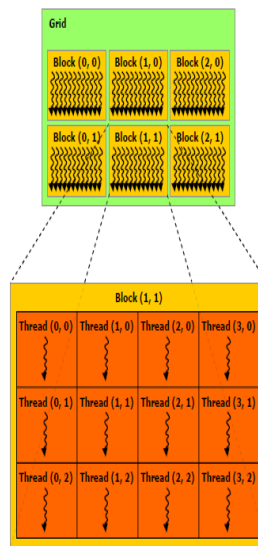


Figure 2-1. Grid of Thread Blocks

Shared Memory:

`__shared__ int a[SIZE];`

- Allocated per thread block, same lifetime as the block
- Accessible by any thread in the block
- Latency: a few cycles
- High aggregate bandwidth:
 $14 * 32 * 4 \text{ B} * 1.15 \text{ GHz} / 2 = 1.03 \text{ TB/s}$

- Several uses:
 - Sharing data among threads in a block
 - User-managed cache (reducing gmem accesses)

Global memory:

- Accessible by all threads of any kernel
- Data lifetime: from allocation to deallocation by host code
 - cudaMalloc(void ** pointer, size_t nbytes)
 - cudaMemset(void * pointer, int value, size_t count)
 - cudaFree(void * pointer)
- Latency: 400-800 cycles
- Bandwidth: 156 GB/s

3. Writing programs with CUDA (Sample Program for CUDA)

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Parallel C Code

Functions must be declared with a qualifier:

__global__ : GPU kernel function launched by CPU, must return void

__device__ : can be called from GPU functions

__host__ : can be called from CPU functions (default)

__host__ and __device__ qualifiers can be combined

"Hello " sample program for CUDA!

It takes the string "Hello ", prints it, then passes it to CUDA with an array of offsets. Then the offsets are added in parallel to produce the string "World!"

```

#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void hello(char *a, int *b)
{
    a[threadIdx.x] += b[threadIdx.x];
}

int main()
{
    char a[N] = "Hello \0\0\0\0\0\0";
    int b[N] = { 15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

    char *ad;
    int *bd;
    const int csize = N*sizeof(char);
    const int isize = N*sizeof(int);

    printf("%s", a);

    cudaMalloc( (void**)&ad, csize );
    cudaMalloc( (void**)&bd, isize );
    cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
    cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );

    dim3 dimBlock( blocksize, 1 );
    dim3 dimGrid( 1, 1 );
    hello<<<dimGrid, dimBlock>>>(ad, bd);
    cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );
    cudaFree( ad );
    cudaFree( bd );

    printf("%s\n", a);
    return EXIT_SUCCESS;
}

```

FAQs:

1. How to analyze the performance of the parallel algorithm?

An algorithm is analyzed based on its execution time (Time Complexity) and the amount of space (Space Complexity) it requires.

Parallel algorithms are designed to improve the computation speed of a computer. For analyzing a Parallel Algorithm, we normally consider the following parameters –

- Time complexity (Execution Time)

The main reason behind developing parallel algorithms was to reduce the computation time of an algorithm. Thus, evaluating the execution time of an algorithm is extremely important in analyzing its efficiency.

Execution time is measured on the basis of the time taken by the algorithm to solve a problem. The total execution time is calculated from the moment when the algorithm starts executing to the moment it stops. If all the processors do not start or end execution at the same time, then the total execution time of the algorithm is the moment when the first processor started its execution to the moment when the last processor stops its execution.

Time complexity of an algorithm can be classified into three categories–

- Worst-case complexity – When the amount of time required by an algorithm for a given input is maximum.
 - Average-case complexity – When the amount of time required by an algorithm for a given input is average.
 - Best-case complexity – When the amount of time required by an algorithm for a given input is minimum.
- Total number of processors used

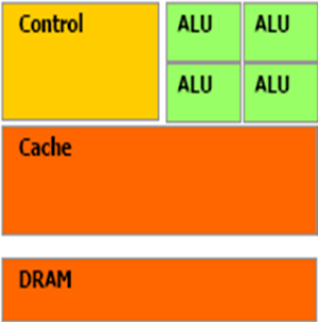
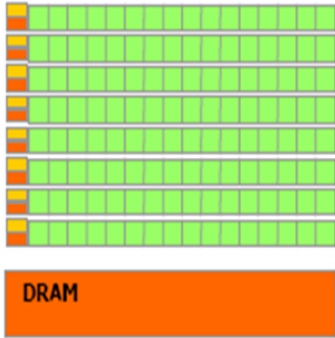
The number of processors used is an important factor in analyzing the efficiency of a parallel algorithm. The cost to buy, maintain, and run the computers are calculated. Larger the number of processors used by an algorithm to solve a problem, more costly becomes the obtained result.

- Total cost

Total cost of a parallel algorithm is the product of time complexity and the number of processors used in that particular algorithm.

$$\text{Total Cost} = \text{Time complexity} \times \text{Number of processors used}$$

2. Differentiate between GPU and CPU.

CPU	GPU
Central Processing Unit.	Graphics Processing Unit.
CPU has multiple cores.	GPU has thousands of cores.
Less number of ALU's	More number of ALU's
Less efficient	More efficient than CPU as processing of large blocks of data is done in parallel
 <p>The diagram shows a CPU architecture with a single yellow 'Control' block, four green 'ALU' blocks arranged in a 2x2 grid, an orange 'Cache' block, and an orange 'DRAM' block.</p> <p style="text-align: center;">CPU</p>	 <p>The diagram shows a GPU architecture with multiple rows of small green blocks representing SMs, each with a small yellow 'Control' block, and a single large orange 'DRAM' block at the bottom.</p> <p style="text-align: center;">GPU</p>
Optimized for low latency access to cached data set	Optimized for data parallel throughput computation.

3. What is streaming multiprocessor(SM) ?

- 1 SM contains 8 scalar cores
- Up to 8 cores can run simultaneously
- Each core executes identical instruction set, or sleeps
- SM schedules instructions across cores with 0 overhead
- Up to 32 threads may be scheduled at a time, called a warp, but max 24 warps active in 1 SM
- Thread-level memory-sharing supported via Shared Memory
- Register memory is local to thread, and divided amongst all blocks on SM