Parallel Programming Notes

Ayush Raina

February 5, 2025

Lecture 1: Architectures 1

Classification of Architectures - Flynns Classification

- SISD: Single Instruction Single Data, for example, serial computers.
- SIMD: Single Instruction Multiple Data, for example, vector processors and processor arrays.
- MISD: Multiple Instruction Single Data, for example, trying different ways to decrypt a message.
- MIMD: Multiple Instruction Multiple Data, for example, multi-core processors.

Classification based on Memory

- **Shared Memory:** All processors share a common memory. Communication is done using this shared address space. This is further classified into UMA (Uniform Memory Access) and NUMA (Non-Uniform Memory Access).
  - UMA: All processors have equal access time to all memory locations. Memory access is slow and has limited bandwidth. UMA has single memory controller.
  - NUMA: Processors have variable access to memory locations. Memory access is faster and has higher bandwidth than UMA. NUMA has multiple memory controllers.
- **Distributed Memory:** Each processor has its own memory.

Shared memory could itself be distributed among processor nodes. Each processor might have some portion of the memory that is physically close to it and therefore accessible in less time.

Cache Coherence Problem

If each processor in a shared memory multiprocessor machine has a data cache, then the problem of cache coherence arises. Our objective is that processes should not read **stale** data.

Cache Coherence Protocols

- **Write Invalidate:** When a processor writes to its cache, it sends invalidate signal to all other caches that have a copy of that memory location. This means all other cache locations must discard their copy of the memory location and fetch it again from the main memory if needed.
- **Write Update:** When a processor writes to its cache, it sends the updated data to other caches that have a copy of that memory location. This keeps all caches up to date.

False Sharing

If two processors access different variables that are located within same cache line but are not related to each other. In this case any write to one variable will cause cache line to be invalidated and other processor might have to reload the data from main memory. This is called false sharing.

In next lecture we will discuss about interconnecting networks.

Lecture 2: Architectures 2

Interconnection Networks

They are used in both shared memory to connect processors to memory and in distributed memory to connect different processors to each other. Components for interconenction networks are interfaces such as Peripheral Component Interconnect (PCI) or PCI - Express used for connecting processors to network and a network link which connected to communication network.

Communication Network

It consists of switching elements to which processors are connected through ports. **Switching elements** receive data from one point and send it to another point. **Switch** refers to collection of these switching elements. **Network topology** is specific pattern of connections in which these switching elements are connected.
In shared memory systems processors as well as memory units are connected to communication network.

Different Kinds of Network Topologies

1. **Bus:** All processors are connected to a single bus. It is simple and cheap but has limited bandwidth.
2. **Crossbar Switch:** It consists of 2D grid of switching elements, where each switching element consists of two input and two output ports. Input Ports are connected to output ports through a switching logic.

In previous case of Crossbar Switch, we require nm switching elements where n is number of processors and m is number of memory units in case of shared memory architecture or m is the number of processors in distributed memory architectures. To reduce switching complexity, we can use **Multistage Network - Omega Network**.

3. **Omega Network:** It consists of $logP$ stages each consisting of $P/2$ switching elements.

Consider Distributed Memory Architecture. In Crossbar switch, there is dedicated path for any processor to communicate with any other processor without contention but in Omega Network, there is contention if 2 processors wants to communicate to 2 different processors, they might have to take same path through some stage of the network.

If P_i and P_j wants to communicate in Omega Network, first convert $ID(P_i)$ and $ID(P_j)$ to binary and then keep comparing most significant bits and follow **cut through routing** or **pass through routing** to reach destination.

Some commonly used network topologies used in distributed memory architectures are Mesh, Torus, hypercubes and Fat tree.

1. **Mesh:** Grid of switching elements where each switching element is connected to 4 directional neighbours.
2. **Torus:** Mesh with wrap around connections. In this $T(i, 1)$ is connected to $T(i, n)\forall i$. Similarly $T(1, j)$ is connected to $T(m, j)\forall j$.
3. **Hypercube:** Keep Joining binary n cubes to get hypercube. In hypercubes, distance between any two processors is bit difference between their binary representation.
4. **Fat Tree:** It is a tree like structure where each node is a switch and each switch has multiple ports. Leaves are processors. As we go up the tree, number of ports in switch increases. This is Non Blocking Network means no contention because there is unique path between any two processors. Fat Tree has a special property which makes all this happen and that is **Number of Links from node to a children = Number of Links from node to parent**.

bandwidth: maximum amount of data that can be transferred over the network in given time, usually measured in bits per second. (bps)

Evaluating Interconnection Topologies

1. **Diameter:** Maximum distance between any two processing nodes.Smaller diameter means lower latencies.
2. **Connectivity:** Number of Paths between two nodes. It can also be defined as minimum number of links that need to be removed to disconnect the network. Higher connectivity improves fault tolerance.
3. **Fault Tolerance:** Ability of network to operate correctly even if some links or switches fail.
4. **Bisection Width:** Minimum number of links that need to be removed to divide the network into two equal halves.
5. **Channel Width:** Number of bits that can be simultaneously communicated over a link i.e number of physical wires between two nodes. Higher Channel width increases data transfer capacity.
6. **Channel Rate:** Performance of single physical wire i.e The speed at which single physical wire transmits the data and is generally measured in bits per second(bps).
7. **Channel bandwidth:** The total data transfer capacity of a link. It is calculated as Channel Width * Channel Rate. Higher Channel bandwidth allows faster communication.
8. **Bisection bandwidth:** This is defined as maximum volume of communication between two halves of network or in other words maximum data transfer capacity between two halves of network and is given by bisection width * channel bandwidth. Higher bisection bandwidth means better performance under heavy traffic.

Questions / Doubts

1. How fat tree network has constant bisection bandwidth?

Lecture 3: Parallelization Principles

We have seen several advantages of parallelism, but there are also some overheads which are not seen in sequential programs like **communication delay**, **synchronization** and **idling**.

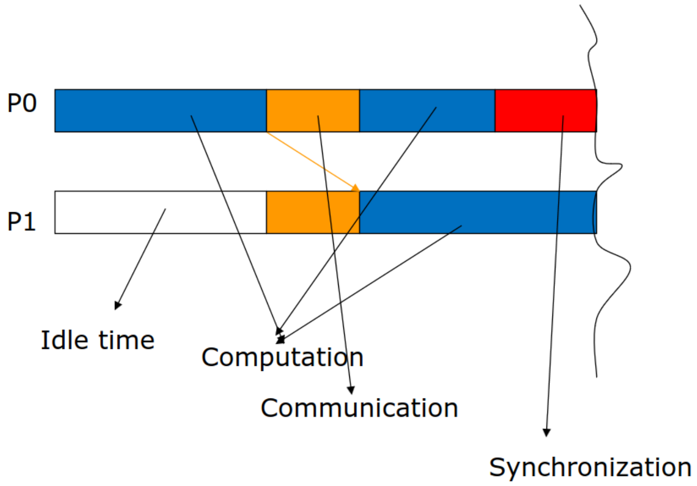

Figure 1: Overheads

Evaluation of Parallel Program

Let us denote execution time by T_p for parallel program using p processors.

1. **Speedup:** $S(p,n) = T(1,n)/T(p,n)$ where $T(1,n)$ is execution time of sequential program. Usually $S(p,n) \leq p$ as we expect program to get p times faster in idead case but sometimes $S(p,n) > p$ which is called **superlinear speedup**. Ideally we want $S(p,n) = p$.

2. **Efficiency:** $E(p,n) = S(p,n)/p$. Efficiency is a measure of how well the parallel program is utilizing the resources. Generally $E(p,n) \leq 1$, but in case of superlinear speedup, $E(p,n) > 1$. Ideally we want $E(p,n) = 1$.

3. **Cost:** $C(p,n) = T(p,n) * p$. It is similar to Efficiency but relates the runtime to number of utilized processors.

4. **Scalability:** We want to measure the efficieny of parallel program for variable number of processors. This is called scalability analysis. There are two types of scalability - **Strong Scalability** and **Weak Scalability**. In strong scalability, we measure the efficiencies for varying number of processors while keeping indut data size fixed. In weak scalability, we measure efficiency of parallel code by varying both the number of processors and input data size.

We will now derive an upper bound on achievable speedup when parallelizing a given sequential program. Let T_{seq} and T_{par} denote the parts whihc cannot benefit from parallelization and can benefit from parallelization respectively. Further assume we can get ideal linear speed up and super linear speedup is also possible. Then we have $T(p,n) \geq T_{seq} + T_{par}/p$.

The Speedup is given by

$$\boxed{S(p,n) = \frac{T(1,n)}{T(p,n)} \leq \frac{T_{seq} + T_{par}}{T_{seq} + T_{par}/p}}$$

Instead of using the actual runtimes, now use fraction of the total runtime. Let f be such that $T_{seq} = f * T(1,n)$ and $T_{par} = (1-f) * T(1,n)$. Then we have

$$\boxed{S(p,n) \leq \frac{1}{f + \frac{1-f}{p}}}$$

The above equation is known as Amdahl's Law. Now we can see some examples:

1. Suppose 95% of a rogram's execution time occurs inside a loop that we want to parallelize. What is the maximum speedup we can expect from a parallel version of our program executed on six processors?

$$\boxed{S(6, n) \leq \frac{1}{0.05 + \frac{0.95}{6}} = 4.8}$$

2. 10% of the program's execution time is spent within sequential code. What is limit to speedup achievable by parallel version ?

Since we are not given how many processors are used, we assume ideal case of unbounded processors.

$$\boxed{S(\infty, n) \leq \lim_{p \to \infty} \frac{1}{0.1 + \frac{0.9}{p}} = 10}$$

We can see the limitation to Amdahl's Law is that it only applies in situation where problem size is fixed. We will now derive a more general upper bound on speedup. Let α be the scaling function for the sequential part of program and β be the scaling function for the parallel part of program. Both α and β are w.r.t to complexity of problem size. This means T_{seq} grows as $\alpha * T(1, n)$ as problem size grows and T_{par} grows as $\beta * T(1, n)$ as problem size grows.

$$T_{\alpha\beta}(1, n) = \alpha * T_{seq} + \beta * T_{par} = \alpha * f * T(1, n) + \beta * (1 - f) * T(1, n)$$
$$T_{\alpha\beta}(p, n) = \alpha * T_{seq} + \frac{\beta * T_{par}}{p} = \alpha * f * T(1, n) + \frac{\beta * (1 - f) * T(1, n)}{p}$$

Then we have

$$\boxed{S_{\alpha\beta}(p, n) \leq \frac{\alpha * f + \beta * (1 - f)}{\alpha * f + \frac{\beta * (1 - f)}{p}}}$$

Let $\gamma = \frac{\beta}{\alpha}$, then we have

$$\boxed{S_{\gamma}(p, n) \leq \frac{f + \gamma * (1 - f)}{f + \frac{\gamma * (1 - f)}{p}}}$$

Now depending on the value of γ, we can have different upper bounds on speedup.

1. Amdahl's Law: $\gamma = 1$ which means $\alpha = \beta$
2. Gustafson's Law: $\gamma = p$ which means $\alpha = 1$ and $\beta = p$. This means parallel part of program grows linearly in p as problem size grows. In this case the formula becomes

$$\boxed{S(p, n) \leq f + (1 - f) * p}$$

This law can also be thought as by knowing f , we can use it to predict the theoretically achievable speedup using multiple processors when parallelizable part scales linearly with the problem size while the serial part remains constant

let us discuss some examples:

1. Suppose we have a parallel program that is 15% serial and 85% linearly parallelizable for a given problem size. Assume that the (absolute) serial time does not grow as the problem size is scaled.

   (i) How much speedup can we achieve if we use 50 processors without scaling the problem - $S_{\gamma=1}(50, n)$ (ii) Suppose we scale up the problem size by a factor of 100. How much speedup could we achieve with 50 processors - $S_{\gamma=100}(50, n)$

Roofline Performance Model

This gives a bound on the performance of an application on a particular architecture and it also helps to categorize the code's performance as memory bound or performance bound. It depends on three parameters - Peak Performance of the machine P_{peak} $(FLOP/s)$, Memory Bandwidth B_{peak} $(Bytes/s)$ and Computational Intensity I_{op} $(FLOP/Byte)$.

Questions / Doubts

1. Discuss this Roofline Model again with Professor and work out some examples.

Lecture 4.1: Parallel Programming Models

Some parallel programming models include Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD). We will focus on SPMD model.

Programming Paradigms

1. Shared Memory Model: Threads, Open MP, CUDA.

2. Distributed Memory Model: MPI - Message Passing Interface.

Data Parallelism and Domain Decomposition

Given data is divided into processing entities. Each process owns and computes a portion of the data.
Multidimensional Data in simulations is divided into subdomains and each subdomain is assigned to a processing entity. This is called **Domain Decomposition**.

Process Grids

Given P processors are arranged in multi dimensions forming a process grid. Once this is done then domain of problem is divided in process grid.

Distribution of Data

There are various ways of distributing data like block distribution, cyclic distribution, block cyclic distribution etc. Check Parallelization Principles pdf in good notes for visualization.

Lecture 4.2: PRAM Model

PRAM Stands for Parallel Random Access Machine. Helps to write precursor (initial version) parallel algorithms without any architecture constraints. Allows algorithm designers to treat processing power as unlimited and it ignores complexity of inter communication.

Benefits of PRAM Model

1. Helps in designing parallel algorithms as PRAM provides a basic structure that can be adapted to real world parallel machines.

2. Base PRAM algorithm can help establish tight lower and upper bounds for practical implementations and assumptions made in PRAM model helps architecture designers for improving their designs.

3. Can be suitable for modern day architectures like GPU's.

PRAM Model Architecture

The PRAM can be viewed as an ideal shared memory architecture that does not consider any overheads. . Thus, when designing PRAM algorithms we can focus on the best possible parallel algorithm design. Asymptotic runtimes of optimal PRAM algorithms can often be taken as lower bounds for implementations on actual machines.

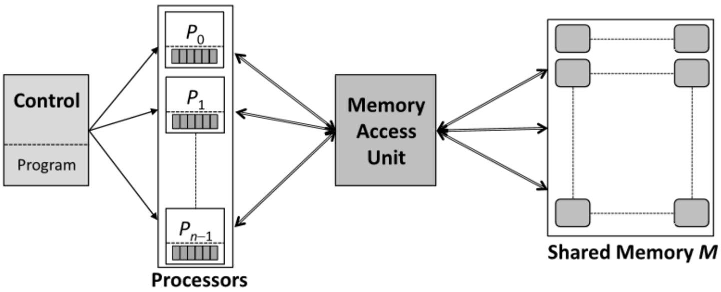

FIGURE 2.1

Important features of a PRAM: n processors $P_0, \ldots, P_{n-1}$ are connected to a global shared memory M, whereby any memory location is uniformly accessible from any processor in constant time. Communication between processors can be implemented by reading and writing to the globally accessible shared memory.

Figure 2: PRAM Model

PRAM consists of n identical processors P_i for $0 \leq i \leq n-1$. In every step each processor executes an instruction cycle in three phases:

1. **Read Phase:** Each processor can simultaneously read a single data item from a (distinct) shared memory cell and store it in a local register.

2. **Compute Phase:** Each processor can perform a fundamental operation on its local data and subsequently stores the result in a register.

3. **Write Phase:** Each processor can simultaneously write a data item to a shared memory cell,whereby the exclusive write PRAM variant allows writing only distinct cells while concurrent write PRAM variant also allows processors to write to the same location which can lead to race conditions.

Three phase PRAM instructions are executed synchronously. Communications between the processors in the PRAM needs to be implemented in terms of reading and writing to shared memory. This type of memory can be accessed in a uniform way; i.e., each processor has access to any memory location in unit (constant) time. This makes it more powerful than real shared memory machines in which accesses to (a large) shared memory is usually non-uniform and much slower compared to performing computation on registers. The PRAM can therefore be regarded as an idealized model of a shared memory machine; e.g. we cannot expect that a solution for a real parallel machine is more efficient than an optimal PRAM algorithm.

PRAM Variants

Several types of PRAM variants have been defined which differ in how data in shared memory can be read/written: only exclusively or concurrently. So we have 4 possible combinations **ER** - Exclusive Read, **CR** - Concurrent Read, **EW** - Exclusive Write, **CW** - Concurrent Write. These are also shown in Figure 2.2. We now describe three most popular variants the EREW, CREW, and CRCW PRAMs.

EREW PRAM

EREW stands for Exclusive Read Exclusive Write. No two processors are allowed to read or write to the same memory location during same cycle.

CREW PRAM

CREW stands for Concurrent Read Exclusive Write. Several processors may read data from the same shared memory cell simultaneously. Still, different processors are not allowed to write to the same shared memory cell.

CRCW PRAM

CRCW stands for Concurrent Read Concurrent Write. Both simultaneous reads and writes to the same shared memory cell are allowed in this variant. In case of a simultaneous write (analogous to a race condition) we need to further specify

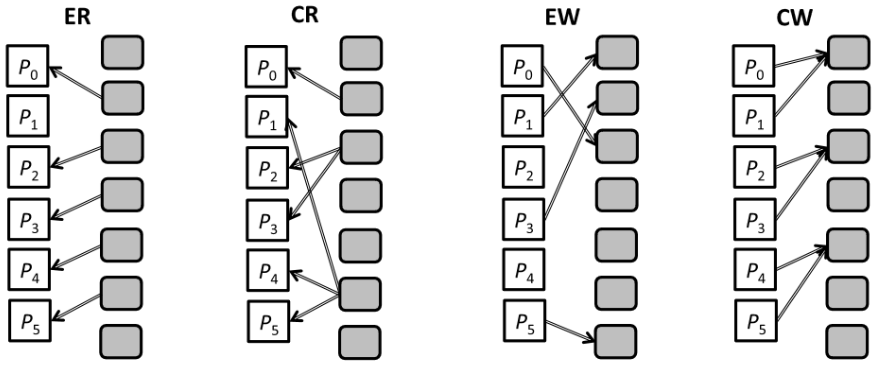

FIGURE 2.2

The four different variants of reading/writing from/to shared memory on a PRAM.

Figure 3: PRAM Variants

which value will actually be stored.

There are 4 common approaches to deal with the situation where two or more processors attempt to write to same memory location during the same clock cycle are:

1. Priority Concurrent Write (CW) - Processors have been assigned distinct priorities and the processor with the highest priority succeeds in writing its value to the shared memory.
2. Arbitary Concurrent Write (CW) - A randomly choosen processor succeeds in writing its value to the shared memory.
3. Common Concurrent Write (CW) - All processors writing to same global memory must write the same value.
4. Combining Concurrent Write (CW) - All values to be written are combined into a single value by means of an associative binary operations such as sum, product, minimum, or logical AND.

Parallel Prefix Sum Computation

1. First method involves spawnning N threads and each thread computes one element of the prefix sum array. Here is simple CUDA kernel code for this one assuming that threads are distributed in such a way global thread ID $\leq N-1$.

Prefix Sum CUDA Kernel Implementation $\mathcal{O}(n^2)$

```
__global__ void NaivePrefixSum(int* A, int* C) {
    int index = threadIdx.x;

    int sum = 0;
    for(int i = 0;i <= index; i++) {
        sum += A[i];
    }

    C[index] = sum;
}
```

2. Second method is slightly better approach than first one and is easy to understand from followig figure

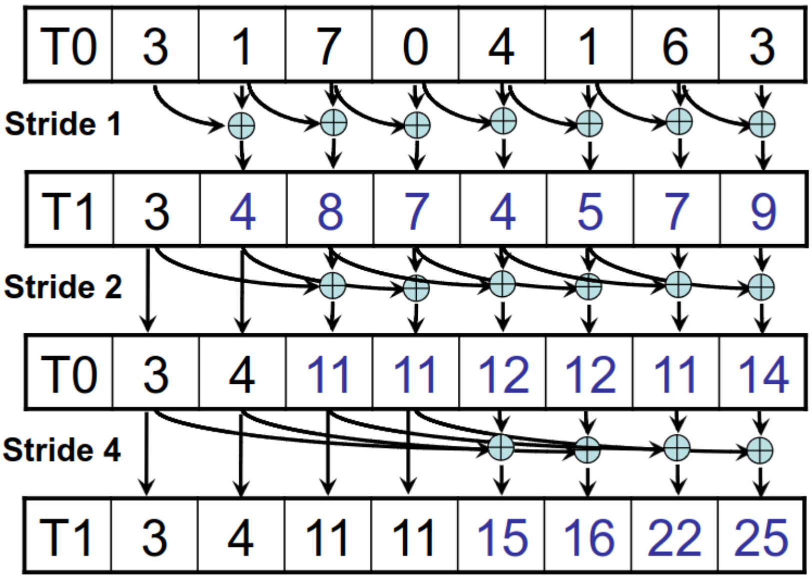

Figure 4: Prefix Sum Approach 2 $\mathcal{O}(nlogn)$

Prefix Sum Approach 2 $\mathcal{O}(nlogn)$

```
__global__ void PrefixSum2(int* A, int* C, int N) {
    int index = threadIdx.x;
    for(int stride = 1; stride < N; stride *= 2) {

        if(index + stride < N) {
            C[index + stride] += C[index];
        }

    }
}
```

Parallel Merge of 2 Sorted Lists

Normally merge of two sorted lists takes $\mathcal{O}(m+n)$ time and $\mathcal{O}(m+n)$ space. Where as if we have to do this in constant space then it will take $\mathcal{O}(mlogm + nlogn)$ time, Here in our parallel algorithm we will have n processors for n elements in first list and m processors for m elements in second list.

- The processor knows index of element in own list.
- It then finds the index of element in other list using binary search.
- sum of these two indices gives the index of element in merged list.
- Write the element to this index in merged list.

Clearly we can see that the complexity is $\mathcal{O}(nlogm + mlogn)$. Here is the CUDA kernel code for this algorithm. This algorithm will only work if all elements in both lists are **distinct**.

Device Function for Binary Search

```
__device__ int binarySearch(int* nums, int target, int N) {
    int lo = 0;
    int hi = N-1;
    int mid;
    while(lo <= hi) {
        mid = lo + (hi-lo)/2;
        if(nums[mid] == target) return mid;
        else if(nums[mid] < target) lo = mid+1;
        else hi = mid-1;

    }
    return lo;
}
```

Merge of 2 Sorted Lists CUDA Kernel

```
__global__ void merge2SortedLists1(int* A, int* B, int* C, int N, int M) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if(idx >= N+M) return;

    if(idx < N) {
        int otherIdx = binarySearch(B,A[idx],M);
        C[idx+otherIdx] = A[idx];
        printf("Idx␣=␣%d,␣otherIdx␣=␣%d␣for␣element␣%d␣\n", idx, otherIdx, A[idx]);
    }
    else {
        int otherIdx = binarySearch(A,B[idx-N],N);
        C[idx-N+otherIdx] = B[idx-N];
        printf("Idx␣=␣%d,␣otherIdx␣=␣%d␣for␣element␣%d␣\n", idx-N, otherIdx, B[idx-N]);
    }
}
```

Enumeration Sort

This sorting algorithm takes $\mathcal{O}(n^2)$ comparisons, so we are going to spawn n^2 threads for each comparison and hence we can sort in $\mathcal{O}(1)$ time. Here is the algorithm for Enumeration Sort.

- Each thread compares $a[i]$ and $a[j]$. If $a[i] > a[j]$ then $pos[i] = pos[i] + 1$.
- Threads will do concurrent write to $pos[i]$ and we used atomic add operations to put the sum of all increments in $pos[i]$.
- Finally we will have position of each element in sorted array.

CUDA Kernel Implementation for Enumeration Sort

```
__global__ void EnumerationSort(int* A, int* POS) {
    int threadID = threadIdx.x;
    int blockID = blockIdx.x;

    if(A[threadID] > A[blockID]) {
        atomicAdd(&POS[threadID],1); // Ensures correct parallel addition
    }
}
```

Post Processing Step

This Post processing step is not needed if all the elements are distinct. If there are duplicates then we need to do this step.

```
    // Initializing answer array to 0.
    int* answer = (int*)malloc(size);
    for(int i = 0;i < N; i++) answer[i] = 0;

    // Fill Elements at correct place
    for(int i = 0;i < N; i++) {
        int idx = pos[i];

        if(answer[idx] != 0) {  // This is case when duplicate elements are present
            int k = 1;
            bool flag = true;

            while(flag) {
                if(answer[idx+k] == 0) {
                    answer[idx+k] = answer[idx];
                    flag = false;
                }
                else k++;
            }
        }
        else answer[idx] = A[i];
    }
```

Lecture 5.1: Many Core Architectures

Introduction

A single CPU can consists of maximum 128 cores whereas GPU consists of large number of light weighted cores. Less computational part of the program runs on CPU and highly parallel part of the program runs on GPU.
NVIDIA's GPU Architecture is called **CUDA - Compute Unified Device Architecture**.

NVIDIA A100 GPU Architecture

1. CUDA has hierarchical architecture. It consists of 7 Graphical Processing Clusters (GPCs).
2. It has 7 or 8 Texture Processing Clusters (TPCs) per GPC. If we assume 8 then we have 56 TPCs in total.
3. It has 2 Streaming Multiprocessors (SMs) per TPC. Hence upto 16 SMs/GPC.
4. It has 108 SMs in total.

Streaming Multiprocessors (SMs)

1. 64 FP32 Cuda Cores per SM, Hence $64 * 108 = 6912$ FP32 Cuda Cores per GPU.
2. 4 third Generation Tensor Cores per SM. Hence $4 * 108 = 432$ Tensor Cores per GPU. These can together deliver 1024 FP16 / FP32 operations per clock.
3. 192 KB of combined shared memory and L1 cache per SM which is 1.5x larger than V100.

Tensor Cores

1. Tensor Cores are specialized high-performance compute cores for matrix math operations that provide groundbreaking performance for AI and HPC applications.
2. Tensor Cores perform matrix multiply and accumulate (MMA) calculations. Hundreds of Tensor Cores operating in parallel in one NVIDIA GPU enable massive increases in throughput and efficiency

Memory Hierarchy

1. **Global or Device Memory (DRAM):** This is largest, slowest and most abundant memory on a GPU and this can be accessed by all threads executing in all the SMs. It has high latency. It can also be accessed by CPU Host. In A100 GPU we have 80 GB of Device Memory.
2. **Shared Memory (On-Chip Memory):** This is available in each SM. This is a small but fast memory that is shared by threads within the same block. It is much faster than global memory because it's physically closer to the cores. In A100 it is around 192 KB per SM. Lower latency as compared to global memory.
3. **Registers:** Each thread has its own registers which is used for storing local data of threads. In A100 there are 64K registers per SM.
4. **Constant and Texture Memory:** Used to improve performance of reads that exhibit spatial locality among threads making it efficient for 2D or 3D image data. It can be accessed by all threads.

Latency of Data Accesses

1. Device Memory: 200-400 clock cycles about 300ns.
2. Shared Memory: 20-30 clock cycles about 5ns.

Difference with CPU Threads

1. Context switching in GPUs is faster because thread states are kept in registers and shared memory until execution is complete, minimizing delays.
2. Cache management in GPUs is explicit, meaning the programmer needs to handle moving frequently used data into shared memory for faster access, whereas in CPUs, the hardware automatically manages the cache.

Questions

1. Is constant and texture memory accessible to threads among any SM ?
2. Algebra of 108 SMs in A100 GPU.
3. What is GA100 and meaning of line A100 GPU implementation of GA100.

Lecture 5.2: CUDA Programming

Hierarchical Parallelism

1. Parallel Computations are arranged in grids, one grid executes after another.
2. Grid Consists of blocks which are assigned to SMs. Multiple blocks can be assigned to single SM.
3. Maximum thread blocks executed concurrently per SM = 16. Max threads per thread block = 1024.
4. A thread is executed on GPU Core.

Question - Check about concurrent block execution in SM vs warps.

Thread Block

An array of concurrent threads that execute the same code and can cooperate to compute the result. It can be 1D, 2D or 3D. Threads of a thread block share memory.

CUDA Programming Model

1. A kernel is executed by a grid of thread blocks.
2. Threads in a thread block can cooperate with each other by sharing their data through shared memory, synchronizing their execution.
3. Threads from different blocks cannot cooperate.

Example1: Add Two Integers using CUDA Kernel

```
__global__ void addIntegers(int* a, int* b, int* c) {
    *c = *a + *b;
}
```

Here is the implementation of main function.

```
int main() {

    int a, b, c;
    int *dA, *dB, *dC;

    size_t size = sizeof(int);

    // Allocate space on GPU
    cudaMalloc(&dA,size); cudaMalloc(&dB,size); cudaMalloc(&dC,size);

    // Initialize;
    a = 4; b = 4;

    // Copy
    cudaMemcpy(dA,&a,size,cudaMemcpyHostToDevice); cudaMemcpy(dB,&b,size,cudaMemcpyHostToDevice);

    // Launch kernel
    addIntegers<<<1,1>>> (dA,dB,dC);

    // Copy back to CPU
    cudaMemcpy(&c,dC,size,cudaMemcpyDeviceToHost);

    // Free memory
    cudaFree(dA); cudaFree(dB); cudaFree(dC);

    cout << "Answer␣is␣->␣" << c << endl;
    return 0;
}
```

Example 2: Vector Addition 1

```
__global__ void VectorAdditionUsingParallelBlocks(int* A, int* B, int* C) {
    /* N Threads Blocks with 1 Thread Per Block */
    int i = blockIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    int threadsPerBlock = 1;
    int numBlocks = N;

    // Invoke the kernel
    VectorAdditionUsingParallelBlocks<<<numBlocks, threadsPerBlock>>> (dA,dB,dC);
}
```

Example 3: Vector Addition 2

```
__global__ void VectorAdditionUsingParallelThreads(int* A, int* B, int* C) {
    /* 1 Thread Block and N Threads in that Block */
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    int threadsPerBlock = N;
    int numBlocks = 1;

    VectorAdditionUsingParallelThreads<<<numBlocks,threadsPerBlock>>> (dA,dB,dC);
}
```

Example 4: Vector Addition 3

```
__global__ void VectorAdditionUsingBlocksAndThreads(float* A, float* B, float* C, int N) {
    // Using both Blocks and Threads
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    // This is because every block has same number of threads and it is possible that some threads need not to do anything
    if(i < N) {
        C[i] = A[i] + B[i];
    }

    return;
}

int main() {
    // Kernel Parameters
    int threadsPerBlock = 512;
    int blocksPerGrid = (int)ceil((float)(N/(threadsPerBlock*1.0))); // This can also be achieved using (N + M - 1)/M;

    // Invoke kernel, it has to void return type
    VectorAdditionUsingBlocksAndThreads<<<blocksPerGrid,threadsPerBlock>>> (dA,dB,dC,N);
    cudaDeviceSynchronize();
}
```

Example 5: Vector Addition 4

```
__global__ void VectorAdditionUsing3DBlocks(int* A, int* B, int* Answer, int nX, int nY, int nZ) {

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int z = blockIdx.z * blockDim.z + threadIdx.z;

    if(x < nX and y < nY and z < nZ) {
        int index = x + y*nX + z*nX*nY;
        if(index < nX * nY * nZ) {
            Answer[index] = A[index] + B[index];
        }
    }

}
```

Here is main function for this kernel.

```
int main() {
    int N = 1000000; // 10ˆ6
    size_t size = N * sizeof(int);

    int *A = (int*) malloc(size), *B = (int*) malloc(size), *C = (int*) malloc(size);

    for(int i = 0;i < N; i++) {
        A[i] = rand() % 15; B[i] = rand() % 20;
    }

    int *dA, *dB, *dC;
    cudaMalloc(&dA,size); cudaMalloc(&dB, size); cudaMalloc(&dC, size);

    cudaMemcpy(dA,A,size,cudaMemcpyHostToDevice); cudaMemcpy(dB,B,size,cudaMemcpyHostToDevice);

    dim3 blockSize(16,8,8);

    int numBlocksInX = (100 + 16 - 1) / 16, numBlocksinY = (100 + 8 - 1) / 8, numBlocksInZ = (100 + 8 - 1) / 8;
    dim3 numBlocks(numBlocksInX,numBlocksinY,numBlocksInZ);

    // Kernel with 3D Blocks
    auto s = getTime();
    VectorAdditionUsing3DBlocks<<<numBlocks,blockSize>>> (dA,dB,dC,100,100,100);
    cudaDeviceSynchronize();
    auto e = getTime();

    nanoseconds duration = duration_cast<nanoseconds> (e - s);
    cout << "Time␣taken␣to␣do␣Vector␣Addition␣using␣3D␣block␣is:␣" << duration.count() << endl;

    // Kernel with 1D Blocks
    s = getTime();
    VectorAdditionUsing1DBlock<<<(N + 128 - 1) / 128,128>>>(dA,dB,dC,N);
    cudaDeviceSynchronize();
    e = getTime();

    duration = duration_cast<nanoseconds>(e - s);
    cout << "Time␣Taken␣to␣do␣Vector␣Addition␣using␣1D␣Blocks␣is:␣" << duration.count() << endl;

    cudaMemcpy(C,dC,size,cudaMemcpyDeviceToHost);
    cudaFree(dA); cudaFree(dB); cudaFree(dC);

    free(A); free(B); free(C);
    return 0;

}
```

Example 6: Matrix Vector Multiplication

```
    // GPU Kernel
__global__ void matrixVectorMult(float* A, float* V,float* Answer, int M, int N) {

    int curr_row, curr_col;
    float sum = 0;
    curr_row = blockIdx.x * blockDim.x + threadIdx.x;

    if(curr_row < M) {

        for(curr_col = 0; curr_col < N; curr_col++) {
            sum += A[N*curr_row + curr_col]*V[curr_col];
        }
        Answer[curr_row] = sum;

    }
}
```

main() function is discussed in next page.

```
int main() {

    // Change these values to 1024, 1024 to see time difference in Cuda vs CPU
    int m = 10;
    int n = 20;

    size_t size_matrix = m*n*sizeof(float), size_vector = n*sizeof(float), size_answer = m*sizeof(float);

    // Allocating Space on CPU
    float* A = (float*) malloc(size_matrix), *V = (float*) malloc(size_vector), *Answer = (float*) malloc(
        size_answer);

    // Allocating Space on GPU
    float* dA; cudaMalloc((void**) &dA, size_matrix);
    float* dV; cudaMalloc((void**) &dV,size_vector);
    float* dAnswer; cudaMalloc((void**) &dAnswer,size_answer);

    // Initialization of Matrix
    for(int i = 0;i < m; i++) {
        for(int j = 0;j < n; j++) {
            A[i*m + j] = rand() % 10;
        }
    }

    // Initialization of Vector
    for(int i = 0;i < n; i++) V[i] = rand() % 10;

    // Copy
    cudaMemcpy(dA,A,size_matrix,cudaMemcpyHostToDevice); cudaMemcpy(dV,V,size_vector,cudaMemcpyHostToDevice);

    int threadsPerBlock = 128;
    int numBlocks = (m + threadsPerBlock - 1) / threadsPerBlock;

    // Invoking kernel;
    auto start = getTime();
    matrixVectorMult<<<numBlocks, threadsPerBlock>>> (dA,dV,dAnswer,m,n);
    cudaDeviceSynchronize(); // Wait until CUDA kernel finishes completely.
    auto end = getTime();

    // Time Taken By CUDA
    nanoseconds duration = duration_cast<nanoseconds>(end-start);
    cout << "Time␣Taken␣by␣GPU␣Kernel:␣" << duration.count() << endl;

    // Copy Result Back
    cudaMemcpy(Answer,dAnswer,size_answer,cudaMemcpyDeviceToHost);

    // Free
    cudaFree(dA); cudaFree(dV); cudaFree(dAnswer);

    // Check Output
    displayVector(Answer,m);

    // Invoke CPU Call
    start = getTime();
    matrixVectorMultiplicationCPU(A,V,Answer,m,n);
    end = getTime();

    // Time Taken by CPU
    duration = duration_cast<nanoseconds> (end - start);
    cout << "Time␣Taken␣by␣CPU␣Call:␣" << duration.count() << endl;

    displayVector(Answer,m);

    free(A); free(V); free(Answer);

    return 0;
}
```

Example 7: Matrix Matrix Multiplication

```
    // naive version of matrix multiplication on CUDA, we will implement optimized version soon
__global__ void MatrixMult(int* A, int* B, int* C, int N, int K, int M) {
    int curr_row = blockIdx.y * blockDim.y + threadIdx.y;
    int curr_col = blockIdx.x * blockDim.x + threadIdx.x;

    int sum = 0;
    if(curr_row < N and curr_col < M) {
        for(int i = 0;i < K; i++) {
            sum += A[curr_row * K + i] * B[i * M + curr_col];
        }
        C[curr_row * M + curr_col] = sum;
    }
}

int main() {
    int N = 1024;
    int K = 1024;
    int M = 1024;

    size_t sizeA = N*K*sizeof(int);
    size_t sizeB = K*M*sizeof(int);
    size_t sizeC = N*M*sizeof(int);

    int *A = (int*)malloc(sizeA), *B = (int*)malloc(sizeB), *C = (int*)malloc(sizeC);

    fill(A,N*K);
    fill(B,K*M);

    int *dA, *dB, *dC;
    CUDA_CHECK_ERROR(cudaMalloc((void**)&dA,sizeA)); CUDA_CHECK_ERROR(cudaMalloc((void**)&dB,sizeB)); CUDA_CHECK_ERROR(cudaMalloc((void**)&dC,sizeC));
    CUDA_CHECK_ERROR(cudaMemcpy(dA,A,sizeA,cudaMemcpyHostToDevice)); CUDA_CHECK_ERROR(cudaMemcpy(dB,B,sizeB,cudaMemcpyHostToDevice));

    // Invoking the kernel
    dim3 threadsPerBlock(2,2,1);

    int blocksInX = (M + threadsPerBlock.x - 1) / threadsPerBlock.x;
    int blocksInY = (N + threadsPerBlock.y - 1) / threadsPerBlock.y;
    dim3 gridSize(blocksInX,blocksInY,1);

    auto s = getTime();
    MatrixMult<<<gridSize,threadsPerBlock>>> (dA,dB,dC,N,K,M); CUDA_CHECK_ERROR(cudaDeviceSynchronize());
    auto e = getTime();


    nanoseconds durationGPU = duration_cast<nanoseconds> (e - s);
    cout << "Time␣taken␣for␣CUDA␣Kernel:␣" << durationGPU.count() << endl;

    CUDA_CHECK_ERROR(cudaMemcpy(C,dC,sizeC,cudaMemcpyDeviceToHost));
    CUDA_CHECK_ERROR(cudaFree(dA)); CUDA_CHECK_ERROR(cudaFree(dB)); CUDA_CHECK_ERROR(cudaFree(dC));

    int* C_CPU = (int*)malloc(sizeC);

    s = getTime();
    MatrixMultOnCPU(A,B,C_CPU,N,K,M);
    e = getTime();

    nanoseconds durationCPU = duration_cast<nanoseconds> (e - s);
    cout << "Time␣taken␣for␣CPU:␣" << durationCPU.count() << endl;

    // cout << "Matrix A: " << endl; display(A,N,K);
    // cout << "Matrix B: " << endl; display(B,K,M);

    // cout << "Matrix C from GPU: " << endl; display(C,N,M);
    // cout << "Matrix C from CPU: " << endl; display(C_CPU,N,M);

    cout << "SpeedUP:␣" << (float) (durationCPU.count()) / (durationGPU.count() * 1.0) << endl; // Around 47 times faster

    free(A); free(B); free(C);
    return 0;
}
```

Do checkout other functions and macros in code available no github.

Now we will discuss a optimised version of matrix vector multiplication using shared memory.

Example 8: Matrix Vector Multiplication Using Shared Memory

```
__global__ void MatrixVectorMultUsingSharedMemory(int* A, int* V, int* Answer, int N, int M) {

    __shared__ int sharedMemory[512]; // This needs to be same as block size = cols in matrix
    int tid = threadIdx.x;

    sharedMemory[tid] = V[tid]; // Load vector V into shared memory
    __syncthreads();

    int curr_row = blockDim.x * blockIdx.x + tid;
    int sum = 0;

    for(int curr_col = 0; curr_col < M; curr_col++) {
        sum += A[curr_row * M + curr_col] * sharedMemory[curr_col];
    }

    Answer[curr_row] = sum;

}
```

To see time difference we choosed rows = 10240 and cols = 512, then time taken by code dicussed in Example 8 = 0.438 ms and time taken by code discussed in Example 6 = 0.472 ms.