Programming Massively Parallel Processes

__/__/____

GPU

Parallel Platform

- Logical View
  - Programming view
    - Figure out Parallel tasks
    - Communication b/w tasks.
- Physical View
  - Architecture - NVDIA
  - Quantification.

→ Parallel Decomposition of Work: Division of tasks among the cores.

→ Resource Requirement: Cores required.

→ Mapping between resources (cores) & tasks.

→ Communication b/w the cores for Synchronization due to dependency of a core's task on another core's task.

→ Latency - Time required for computation
- Measured in seconds.
- ~~Bytes/sec~~.
- FLOPS (Floating-point Operations) (Little Latency)
- Data Access (Majority of Laten
  - → Bandwidth - Bytes/s (Memory Throughput)

Compute

-> Throughput - FLOPs/second

→ Focus on Latency

-> Latency devices (CPU) - good in doing a single operation quick
1 job - 1 min

→ Focus on throughput
10 jobs - 10 min

Throughput devices (GPU) - good in doing multiple jobs at a time but not good for single op.
1 job - 5 min
10 jobs - 5 min.

→ Heterogenous Computing: ~~Used to~~ Use CPU + GPU

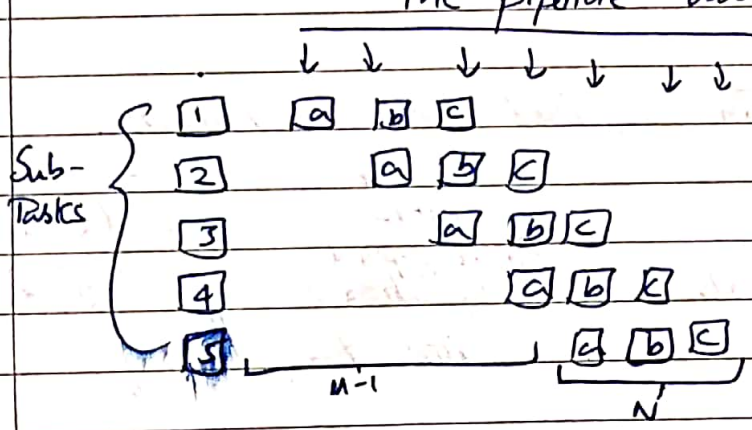
→ The control unit in CPU's are very fast
The ALU in GPU's are very fast.

→ Three levels of Parallelism:-
(i) ILP → Pipeline.
(ii) Thread → GPU → CUDA
(iii) Node → ~~(CPU + GPU)~~ Multiple GPUs

→ Multi-Core → CPU - 6 ~~or 8~~ cores.
Many-Core → GPU - 100s of cores

→ ILP:
↳ Instruction Level Parallelism:

The pipline divides a task into sub-tasks.



Sub-tasks are distributed accross functional units specialising in the sub-task & carry it in a single cycle.

If ~~task~~ task was carried out in serialized manner:

$T_{serial} = M * N$ cycles

(M = No. of sub-tasks)

In ILP, $T_{ILP} = (M-1) + N$ cycles.

Since $(m-1) \ll N$, $T_{ILP} \approx N$

So speed up $= \frac{mN}{N} \approx m$.

The ~~higher~~ more the depth the better the speed up.

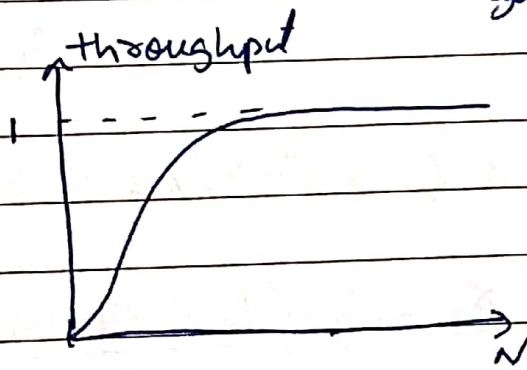
→ Each sub-task is a thread.

→ Wind up time = m-1 → No output for that time.

→ All threads can also have the same task & there is no need for unique sub-tasks.

→ ILP is not helpful when m is ~~small~~ comparable with N. Ex: tight loops → 5 iterations.

→ Instruction throughput (Instructions/cycle) $= \frac{N}{N+m-1} = \frac{1}{1+\frac{m-1}{N}}$



The ideal situation is one instruction per cycle.

So max. throughput is 1 ist/cycle and this is the ILP-wall.

→ Multiplicity is used improve computing i.e compute time & Data access

↓ Multiple cores

↓ Multiple Memories & Multiple data paths

CPU - 2, 6, 8.. 64 GPUs - 1000

__/__/____

→ In CPU memory allocation, location cannot be specified
GPU can be specified.

Hierarchy of Searching for data

→ Processing Unit
↓
Registers
↓
L1 Cache → Present on each cores
↓
L2 Cache → Shared by the cores
↓
L3 Cache
↓
RAM

→ If a cache is K times faster than the RAM
Re-use ratio is r → temporal locality of data.
T_m - time req. to get data from RAM

$$T_c = \frac{T_m}{K}$$

↓
Cache Latency

Avg time to get data = $r T_c + (1-r) T_m$

$$\text{Speed-up} = \frac{T_m}{(r T_c + (1-r) T_m)} = \frac{1}{\left(\frac{T_c (r + (1-r)K)}{K T_c}\right)}$$

$$= \frac{K T_c}{T_c (r + (1-r)K)} = \frac{K}{r + (1-r)K}$$

If $r = 1$, Speed up = K.

If $r = 0$, Speed up = 1 → no speedup.

% operator is much slower than + or - since the pipeline is much longer

__/__/__

→ Spatial Locality:-

Cache Lines: If a[1] is requested, a[1], a[2] --- a[n-1] are fetched

→ FLOPS = (No. of Cycles per Second / Clock rate) × No. of FLOP per cycle × No. of cores per processing Unit

For a CPU, FLOPS = $2.5 \times 10^9 \times 4 \times 1$ (2.5 GHz)

= 10 Giga Flops.

For a GPU, FLOPS = 10^4 giga Flops = 10 TFlops.

But for a GPU → Clock rate is ≈ 1 GHz

$= \frac{10}{2.5} = 4$ TFlops.

Data rate is in 0 GigaBytes per second & Flops is in Tera Flops So parallelism is required to match them.

→ Missed Class - 1/8/19

→ Vector Triad → Low level Benchmarking → Small standard codes to test architectural aspects

total is to ensure that no. of iterations are constant

Properties of Vector triad:

Loads → 3

Store → 1

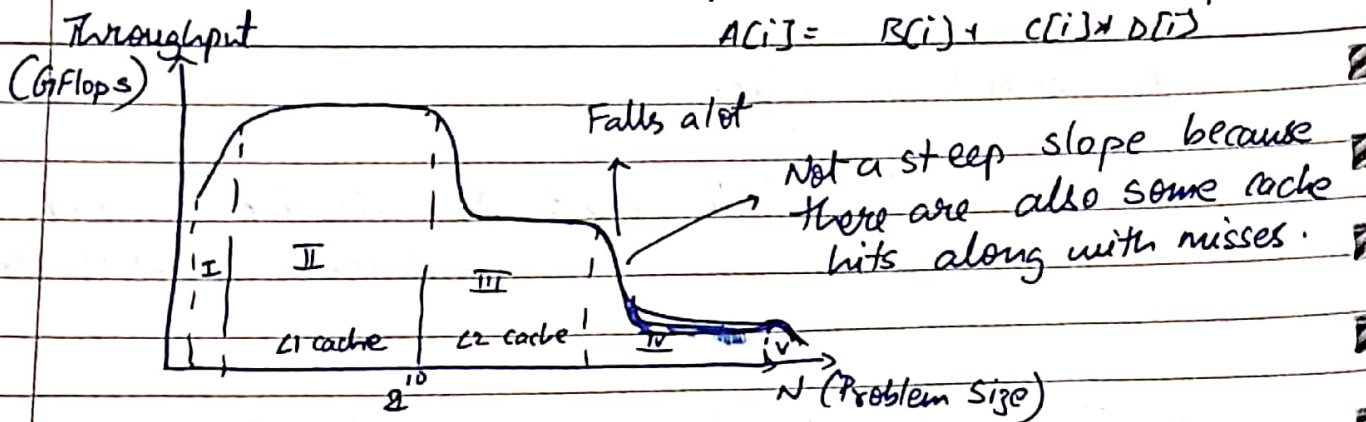
Operations → 2

Spatial Locality & Temporal Locality

```
for i=0:R
  for j=0:N
    a[j] = b[j] + c[j]*d[j]
```

Double matrices

A[i] = B[i] + C[i] * D[i]



→ Initially pipeline does not offer any benefit because no. of operations are small.

→ Bash Command: lscpu → Memory size

→ II ends at 2^{10} operations — $2^{10} \times 8 \times 4 = 2^{15}$ bytes → L1 cache size.
(8: double; 4: A, B, C, D values)

→ L1 cache can be used by just one process because it is present on each core.

→ Bash Command: top : Processes Running
ps-a :

→ III to IV drops a lot → maybe because L3 is shared. Not entirely

But the drop would remain the same even if there is a single process & it is allowed to access unlimited memory

→ L1 & L2 → Different Latencies but same bandwidth (Memory Throughput)
L3 → Different Latencies & bandwidth w.r.t L1 & L2

__/__/__

Computational Intensity → Describes an algorithm.

→ ~~Code Balance~~ → Compute to Memory Access Ratio (CMA Ratio). (Flops/Bytes)

For Vector triad:

(i) If store is considered ⇒ $\frac{2}{4} = \frac{1}{2}$

(ii) " " " ignored ⇒ $\frac{2}{3}$.

Can be ignored if RHS has a lot more data access

GPU is suitable for higher CMA.

$$\text{Computation Intensity} = \frac{1}{\text{Code Balance}}$$ → Bytes/Flops

→ Machine Balance = $\frac{\text{Theoretical Max. Memory Bandwidth}}{\text{Theoretical Max. Compute Performance}}$ (Bytes/Flops)

↳ Describes an Architecture → so fixed for a computer.

~~Lower Code~~ ~~Right now Code Balance is increasing~~ ~~but~~ Machine Balance is decreasing. (current trend)

→ CPUs can be 10× faster for sequential code

GPUs can be 10× faster for parallel code.

→ Constraints on parallel programming → Time
↳ Resources
↳ Accuracy ∝ No. of Computations.

→ Memory system's consume a lot of power.

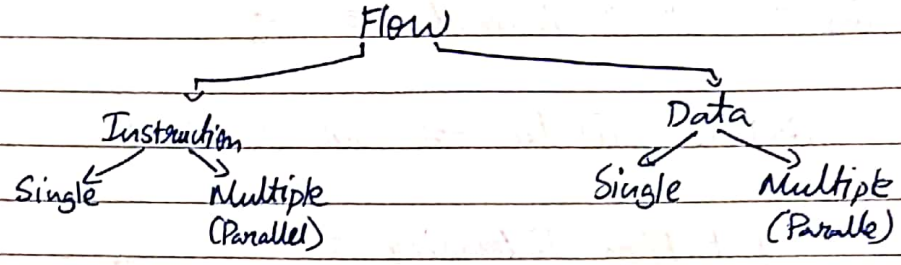
→ Vector processes → A[i] = B[i] + C[i]

Instead of doing it index by index: Entire chunks are computed at a time

__/__/____

→ Von Neumann bottleneck → The increasing gap b/w the data access & processing architectures (Performance-wise)

→

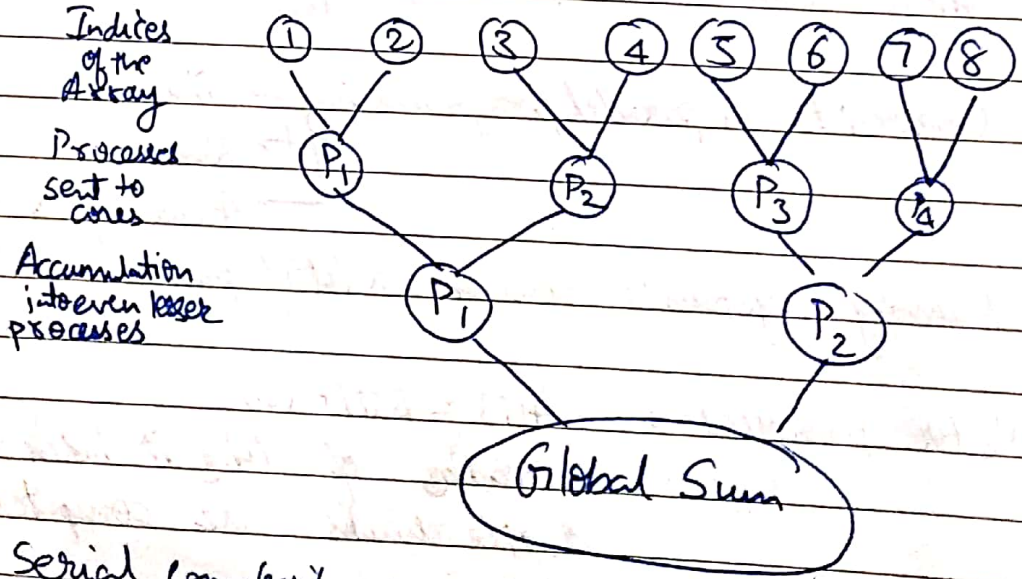


So Flynn's Taxonomy : Single Instruction Single Data
Multiple " " "
Single " Multi "
Multi " " "

GPU programming is usually SI Multi-Thread (Data) (SIMT) / (SIMD)

→ Dependency Analysis :-

Sum of nos. in an array :



Serial Complexity → O(n)
Parallel Complexity → $O(\log_2 n)$

Speedup = $\frac{N}{\log_2 N}$

__/__/____

Such a speedup is possible for any associative operation.

For such a process distribution analysis, we assume:

* Memory Access time is neglected
* No. of available cores is comparable to size of the problem
* Each operation takes same time
* Efficiency of each core is the same.

→ Theoritical Maximum Speedup = $\frac{N}{N/p} = p$.

But memory cache can be used to get a practical speedup > p. → Silver combined core cache >>> Single core or serial cache

→ Efficiency (No Units) = $\frac{\text{Speedup}}{\text{Processes}}$

Sum of nos. in Array (Contd.):-

Serial & Parallel Work Complexity = O(N).

Sometimes parallel work complexity > serial.] → due to dependency
But time complexity may be lower.

→ Amdahl's Law → Theoritical speed up given enough processors

If p=1, Speed up = $\frac{1}{1-1} = \frac{1}{0} \to \infty$.

→ Only when ∞ processors are given

gprof --line a.out gmon.out

gcc -pg file

__/__/____

Speedup = $\frac{1}{\frac{P}{N} + S}$ → fraction of code that can be parallelized

fraction of that can be parallelised → No. of processes.

So here S determines the scope of parallization.

→ Dependency:

```
I   for i = 1:N
II      for j = 1:M
            b[i,j] = b[i,0] + ... + b[i,j-1].
```

Since there is a dependency on j-1 for j only loop I can be parallelised.

→ Flow dependence : Statement i precedes j & (True dependence) i computes a value j uses.

(i) x = 1

(j) y = x + 2

→ Anti dependence : Statement i precedes j & i uses a value j computes.

(i) y = x + 2

(j) x = z - w

→ Output dependence : " " " i computes a value j computes

(i) x = z - w

(j) x = y / z

→ Input dependence : " " " i uses a value j uses

(i) y = x + 2

(ii) z = x * 2

Also

__/__/____

→ Loop Independent Dependence:

```
Loop {
    a[i] = b[i] + c[i]
    d[i] = a[i]
}
```

Dependence distance is 0
No. of iterations over which dependence exists

→ Loop Carried Dependence:

```
Loop{
    a[i] = b[i] + c[i]
    d[i] = a[i-1]
}
```

Dependence distance is 1.

Also have flow dependence

→

```
for (i=0 ; i<N; i++)        → Can be parallelised
    for (j=0; j<N; j++){    → Cannot be parallelized.
        a[i][j+1] = a[i,j]+1
    }
}
```

→

```
for (i=0 :N)
    a[i-1] = b[i]
for (j=0 :N)
    a[j] = a[j]
```

Can ~~Cannot~~ be parallelized

Loop Fusion

```
for (i=0 : N)
    a[i-1] = b[i]
    c[j] = a[i]
```

Cannot be parallelized.
Better for ~~seq~~
Serial running due to locality of reference

Loop Distribution

__/__/____

→ CUDA - Compute Unified Device Arch.

→ GPU Kernels - Functions · Usually SIMT

In a program some part is executed on CPU & GPU
The parts going to GPU - kernel.

→ GPU & CPU have their own RAM & memory.

→ Stack vs. Heap

* Scopes are different. Stak - once out scope, memory is cleared.
* Stacksize is static. Heap is dynamically allocated.
* Size of Heap is much larger than the size of Stacks.
* Bookkeeping mechanisms are different.
* Programmer manages heap but OS manages OS.

→ CPU manages the ~~usual~~ cuda control. & ~~Implementation~~
Since CPU's archi. supports control.

→ Threads → (Taken Care by) → Cores
↓
Blocks → ~~Streaming Symmetric~~ Multiprocessors (SM)
↓
Grids → Device (GPU)

__/__/____

→ A thread is a lightweight process

→ A process has * memory - Stak, Leap, code, data
* Process Control Block - Stores the context
↳ Stores PID
↳ Stores Open files - to allow context switch
↳ Stores Register values

→ A warp is 32 threads currently - can change depending on the capabilities of the GPUs in the market
* A SM runs a warp.

→ cudaMemcpy → 4th arg of direction is required, because ~~an old~~ the same address can exist in both CPU & GPU → so details regarding direction is required.

→ threadIdx is a struct generated and blockId, blockDim
The x, y, z elements show the id in the axis direction.
Can be accessed only inside the kernel.

→ Registers are very valuable in GPU & therefore extra variables must be minimized.

→ The code might look like:

kernel → Run by GPU ~~Waiting for~~ Making the CPU wait while the kernel runs wastes time
=
Serial code → Run by CPU
=
kernel
=
serial code

When operations are asynchronous, other ~~other~~ operations can be run concurrently.

nvcc → .cu

__/__/____

→ Scheduling threads takes 100s - 1000s of clock cycles in CPUs
takes 2-3 clock cycles in GPUs.
because there is no-premption in GPU scheduling

→ __device__ → executed on device, callable from device
__global__ → " " " , " " host
__host__ → " " host , " " host

Can be used together. i.e a function can be both

Quiz: 2×

→ 0 1 2 3 4 5

→ 0,1 2,3 ~~2,5~~ 6,7 8,9 10,11

→ Constraints:
* No. of threads per block
* No. of blocks per SM
⋮

Such constraints can be known with cudaGetDevice function.

Architectures { Fermi, Tesla, Kepler Major ↖ 3.1, 1.6 ↗ Minor → Improvements to arch.

→ Compute Capability → Hardware Ex: 3.1, 1.6
CUDA Toolkit Version → Software Ex: 10.1

127/128
255

Quiz:

→ id = 2x (Blockid.x * Blockdim.x + threadid.x).

→ Quiz 2: id = 2(Blockdim.x * Block ID) + threadid.x

→ Image Processing.
↳ Memory is flat & not in 2-D.

→ Latency hiding Scheduling – Allocate any block whose requested resources are available.

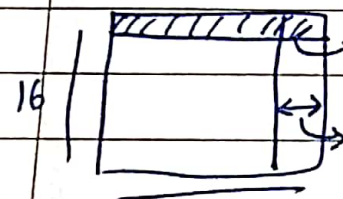
→ Since the Control Unit (CU) is small in the GPU

When all the threads are have the same instructions then all of them (in a block) are executed concurrently – SIMD

if(n!=2) do x
if(n!=50) do y
2 groups

→ ~~Even if~~ If there are n groups of threads in a block which have n different instructions then the n groups will run in different phases serially.
↳ possible
↳ n Control Divergence.
(can be less than n)

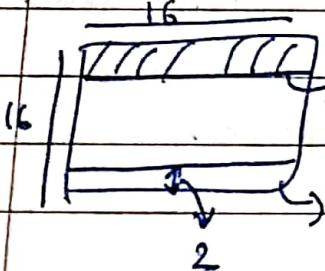
→ A warp is executed at the same time (32 threads).



→ 1 warp → 2 groups in one warp. So 2 cycles per row.
So 32 cycles.

16

Choosing different block size can help too.



→ 1 warp.
→ Same warp

→ 16 cycles.

→ So the scheduling happens for warps & not threads unlike CPU.

→ Stencils - The data access organisation of a thread.

→ Matrix multiplication of matrixes M & N → P

$$P_{i,j} = M[i,:] \cdot N[:,j]$$

↳ dot product

Each thread is mapped to an i,j in P

→ __syncthreads() will ensure that no threads will not cross the function until all threads have reached the function.

→ Must be executed by all threads of the block and it must be the same function call.
i.e

```
if
    xxxx
    __syncthreads()  - (1)
else
    xxxx
    __syncthreads()  - (2)
```

A thread cannot execute (1) & while another thread in the same block executes - (2)

→ If this happens - waiting forever.

→ 2 times the bandwidth of global memory & also consumes an order of energy lesser

Very fast
→ Local Memory - R/w per thread
Registers - " " " → A variable stored here will not have a copy on global
Shared Memory - R/w per block.
Takes time
Global Memory - R/w per grid & R/w from host
Constant Memory - R per grid & R/w from host

Needs a load instruction but Shared is faster
→ Every variable has a scope & lifetime
scope: if a thread or a block of threads can access it
lifetime: If it is erased after the kernel or after entire process

	Memory	Scope (declare in kernel)	Lifetime (Outside the func.)
→ Automatic Variables (not Arrays)	Register	Thread	Kernel
→ Automatic Arrays	Local	Thread	Kernel
Declared in kernel or device func. → __device__ __shared__ int x (Optional)	Shared	Block	Kernel
→ __device__ int x	Global	Grid	Application
Declared outside any function → __device__ __constant__ int x (Optional)	Constant	Grid	Application

Tiling:-

→ In matrix multiplication, $t_{0,0}$ and $t_{0,1}$ both access the 0th row of M. So if $t_{0,0}, t_{0,1}, t_{1,0}, t_{1,1}$ are in a single block & all corresponding rows are brought to shared memory — no. of accesses to global halves
If size of tile is N×N — " " " " " reduces N times
One tile

→ It is better to have 2 threads that access the same memory at a time to execute simultaneously

Global to Shared is one operation

__/__/__

→ Reduction Operator

Ex: Addition, Max, Min of an array
- Associative
- Commutative.

} - To use this pattern, the algo must ~~have~~ be both.

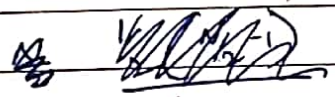
We have a running value that is updated in each iter.

→ 

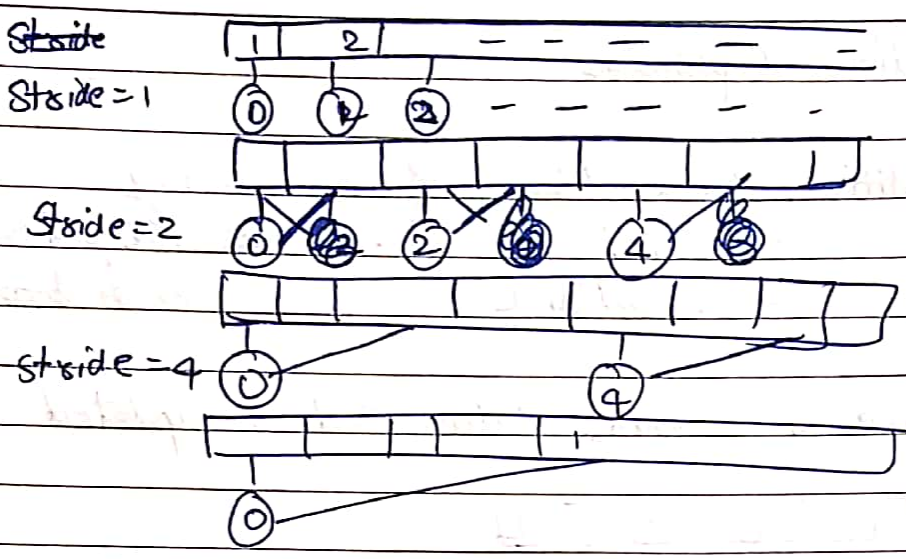
Each thread ~~cost~~ deals with 2 elements & so on
$-\log_2 N$ steps.

In global memory, data access is 2 & computations is 1
So CGMA = ½. (very bad)

So first, transfer data to shared memory - using N threads.
- CGMA = 1
and then work by divide & conquer.

 Total No. of operations is (N-1)
So work complexity O(N)
Step complexity O(logN)
↳ In all parallel algos benefit is in terms of Step complexity

→



→ for (stride = 1 ; stride = blockdim.x ; stride *= 2)

if (tid % stride == 0)

Active

else

deactive .

Sychronise .

→ Implementation — Stride Update .
- Activate / Deactivate .
- Synchronisation .

→ Evaluation - Shared Memory
— CGMA
— Code Divergence .

) Code Divergence is horrible

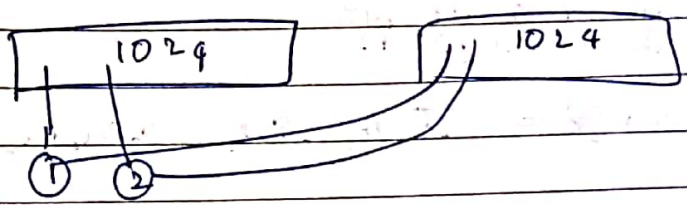
Replace with if (tid < blockdim/stride)

Activate

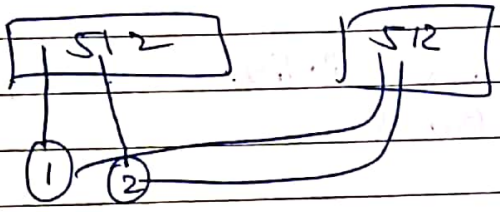
else deactivate .

Then consecutive threads will either be activated or de-activated

If N = 2048 then divid into 2 sections.



Then N = 1024



→ Allocation - Done for a sm. by blocks.

Even if resources are not available for a single thread in a block. ~~block~~ - the block is not allocated to the SM.

Scheduling in terms of warps

→ for (stride = blockdim , stride >=1 , for stride /=2)

if (tid < stride)
active.

→ ? Why load into shared - copy value in shared after first operation.

→ SCAN :-

I/P 1 2 3 4 5

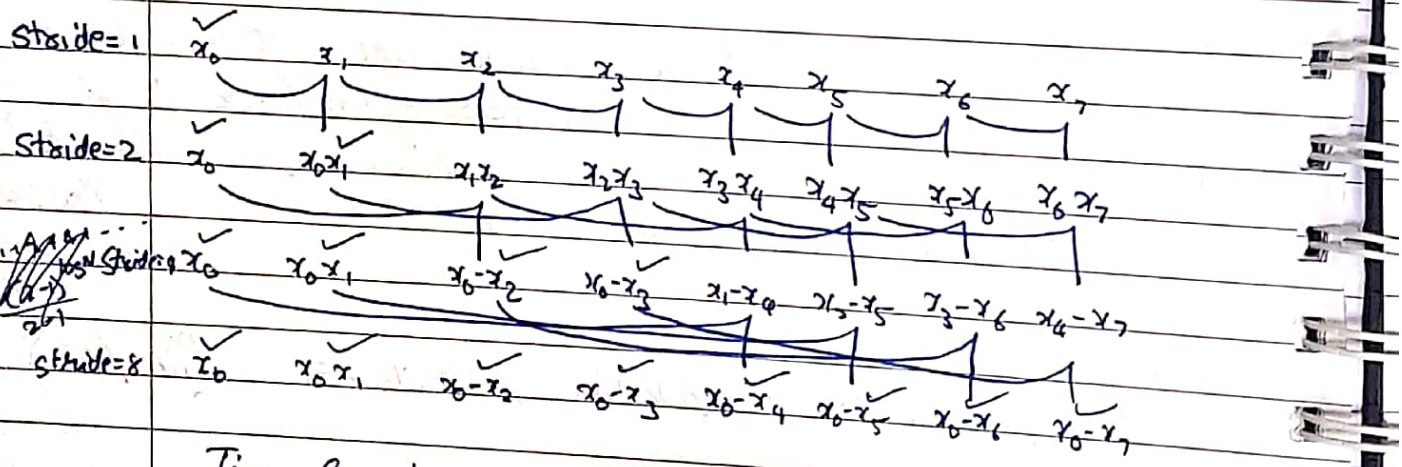
O/P 1 1+2 3+3 5+3 6+4 (12+5)
(3) (5) (8) (12) 17

scan (A) = $[I, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots, a_0 \oplus a_1 \oplus a_2 \cdots a_{n-2}]$

Prefix scan if ⊕ is addition
(Inclusive)

Inclusive begins with first element & last element is the sum of all elements

Exclusive sum begins with 0 & last element is the sum upto the (n-1)th element.



Time Complexity - $\log_2 N$

Work Complexity - $(N-1) + (N-2) + (N-4) + \ldots$ $\log_2 N$ times

$= N \log_2 N - (N-1)$

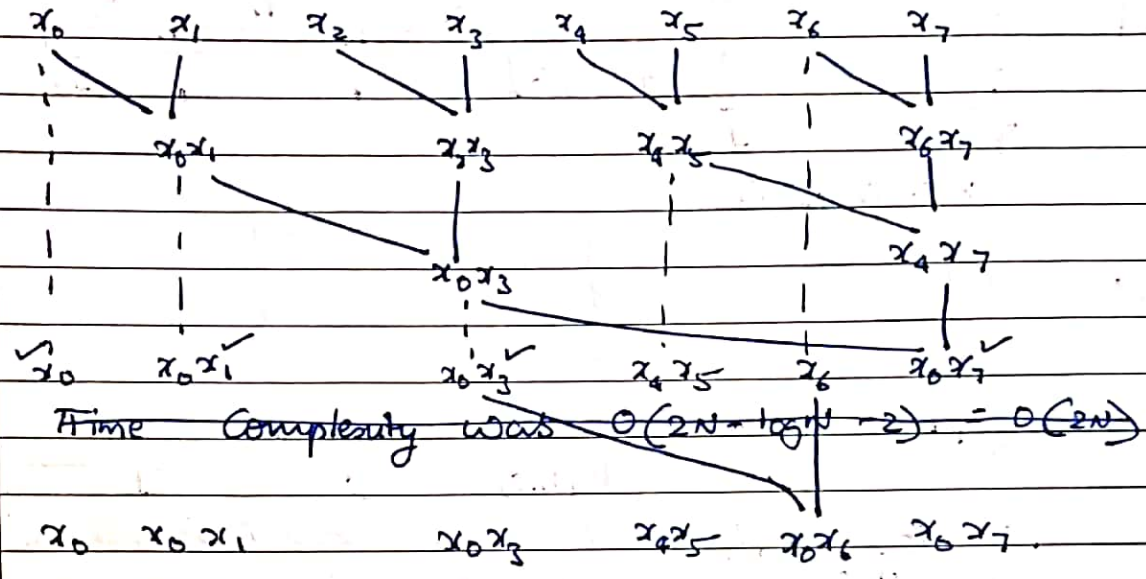
$= N(\log_2 N) - 1$

So we have our work complexity being $\log_2 N$ times that of the serial complexity

N-1 + N-1 - logN
2N - logN - 2

__/__/____

So we need atleast $\log N$ threads to acheive the speed-up equal to serial code.



~~Time Complexity was $O(2N - \log N - 2) = O(2N)$~~

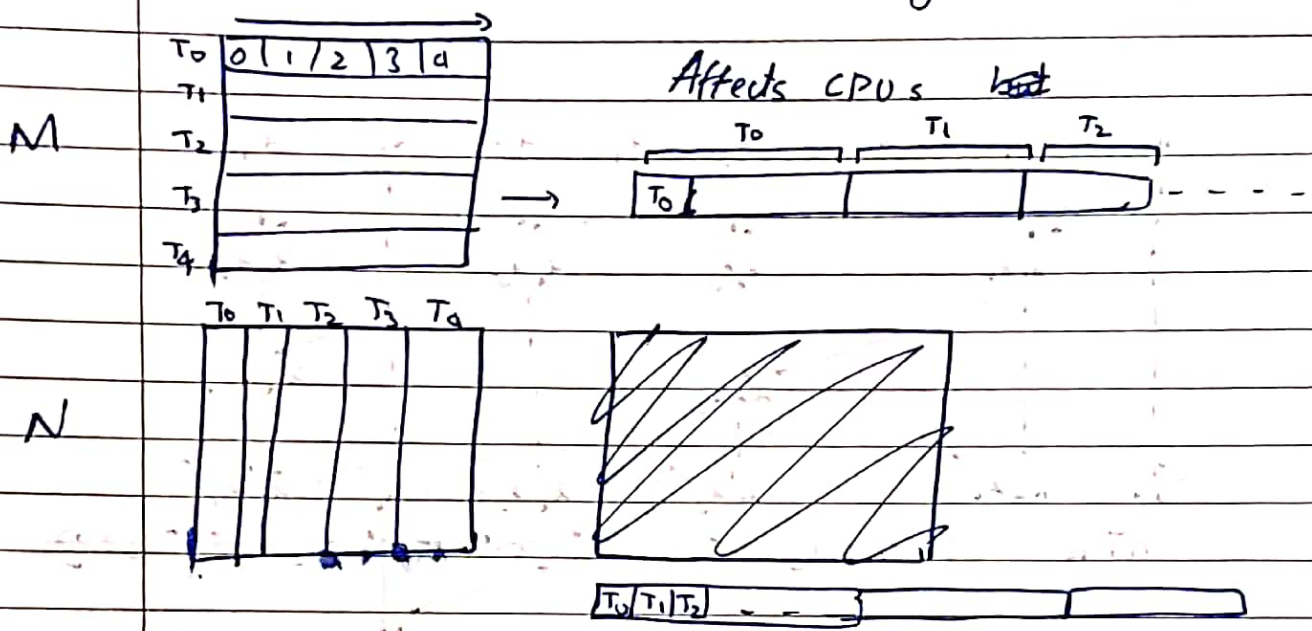
Time Complexity is $O(2N - \log N - 2) = O(2N)$

We divide them into blocks $\Rightarrow$ block 1 depends on block 0.
block 2 " " block 0 & 1

So we do a scan of the last values in the blocks.
& then add these to all values

Coalse → Coalesced

→ Coalsing - the direction we move in is the direction in which memory is arranged



At to, $T_0, T_1 \dots T_n$ all access their memories so it is easier for the warp to pack the memories in the N for to than for M for to.

Convolution:

- Called embarrasingly parallel algo
- → Bigger the control divergence if mask size ↑
- ~~Cannot Read without shared memory is +~~
- → Constant Memory is the cache.

Quiz:

0 1 2 0 4 5 0 0 8 9

0 1 1 0 1 1 0 0 1 1

1 2 2 3 4 4 4 5 6

Scan

→ ~~Protot~~ Mask / filter can be stored in constant memory

→ Each thread makes (2n+1) global memory accesses (Fiter size). & (2n+1) multiplications & (2n+1) additions.

So CGMA = $\frac{4n+1}{2n+1} \approx \frac{4n}{2n} = 2$ (not good)

→ To increase CGMA, we use shared memory. but in block if tid get tid index's element then we need n elements to the left & right of the block.

↳ Control divergence

CGMA = $\frac{\text{block dim} \times 4n}{\text{block dim} + 2n} \approx \frac{\text{block dim} \times 4n}{\text{block dim}} = 4n$

∝ Size of Mask

Matrix Compression:

→ Compressed Sparse Row:

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Data = {1, 2, 2, 3, 1, 2}

Col = {(0,3), (0,1), (2), (3)}

Row = {0, 2, 2, 4, 5, 6}

2 0 2 1 1 → No. of ele. in the rows.

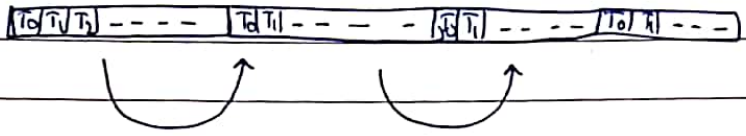
No. of elements

Starting index of the row

__/__/____

→ Histogram :-

~~Date~~ Due to data coalescency (I) is better than (II)

I) 

II) 

In the case of matrix mult. M * N → M is coalesced memory access
& N is partitioned memory access!

→ If all threads are doing the same thing - maybe with different memory locations then there is no control divergence

→ When we have $T_0 - T_4$, all having 'p' as their characters → all access same bin
"Race Condition"

__/__/____

→ Bank Conflict:

Occurs when:

* A warp is sharing/accessing data in a given pattern

Word - 4 bytes

~~* Word = 4 by~~

Each threads has a bank of memory:

	B_0	B_1	B_2	B_3	- - - - - - -	B_{31}
Words in shared memory	W_0	W_1	W_2	- - - - - - -		
	W_{32}	W_{33}	- - - - - - -			
	W_{64}	W_{65}	- - - - - - -			

If shared memory is 8000 ~~Bytes~~ - 2000 words.

=> The memory is W_0 W_1 W_2 - - - - - - - W_{1999}

per clock cycle

Bank bandwidth - 32 bits / bank (fixed)

So if two threads access a bank at the same time, we will have latency.

Individual banks for each thread.

Broadcast

Multicast (Slower than Broadcast)

So if A[threadIdx x 0] → Bank conflict at B_0

A[threadIdx x 1] → No bank conflict

A[threadIdx x 2] → Bank conflicts at every even bank

=> T0 & T16 access W_0 & W_{32} → B_0 by 2 threads

↓

Two-way bank conflict

A[threadIdx x 3] since we only have 32 threads there is no bank conflict.

If multiple threads use the same memory (not bank)

Ex: A[TID x 0] → Broadcast

If divided into 2 or 3 memories

Ex: ~~A[0] if TID~~ A[TID % 2] → Multicast

Broadcast is faster than Multicast.

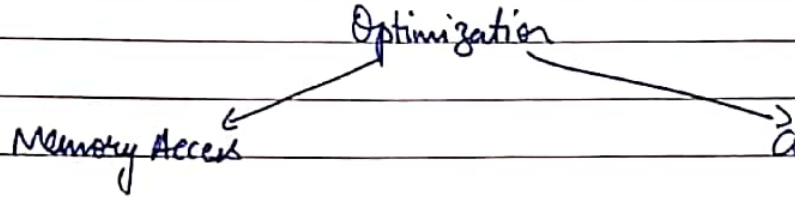
Optimizing Parallel Reduction in CUDA

__/__/____

-> Even if only 1 byte is required, 1 full word is read.

-> If 8 bytes (double) array, then even A[TID] has bank conflict.

-> It is not necessary that all the words in 8 bytes or 12 bytes DS are being accessed at the same time.



Algorithms can be

Memory Bound — Compute Bound.

Reduction - CGMA= $\frac{1}{1}$ poor ratio — Memory bound.

For memory optimization, parameters - memory bandwidth (GB/s)

compute optimization - Compute throughput (GFlops/s)

-> __syncthreads() synchronises in a block but not among blocks — kernel exit & launch must be used for sync b/w blocks.

-> Kernel launch has negligible overhead unlike thread launch in CPU.

#1

Interleaved Addressing - ~~Bcon~~ Bank conflict,
- Too many deactivated threads
- %. is a very slow operation
- Code divergence

#2

→ Solution by multiplying tid with 2*s → code divergence
& %. problem solved.
Bank conflict increased.