13-01-2020 Sri Ramajayam

CS6023 GPU PROGRAMMING

We think in a sequential manner, inherently. But we would like to learn how to write parallel programs, because sequential programs are slow in practice. ↳ to gain performance.

1976 → 1993 → 1997 Architectural improvements, clock
Automatic improvement in performance.

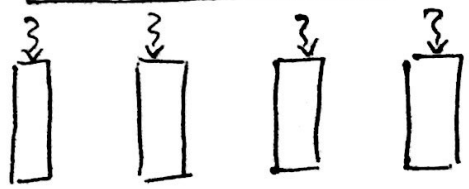
Trend could not be sustained. - cannot further ↑ clock frequency.
heat problems due to extreme power consumption of core.

what to do? - Multi cores.

No. of transistors on a chip ↑✓ clock speed, frequency, power X.

Parallel Platforms. [cat /proc/cpuinfo → no. of cores].

1. Shared memory systems (multicore) ~10s
threads - parallelly run on diff cores
RAM - shared memory among cores.



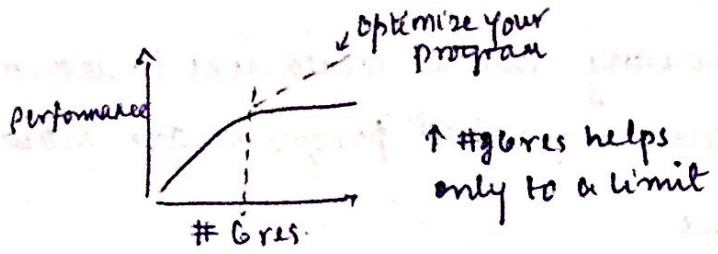
L1, L2 cache - local - to individual cores, not shared

RAM/memory - shared among all cores.
(L3) cache - shared across cores.

Why these many levels of cache?
→ Size v/s cost
→ spatial, temporal locality of reference.
→ Speed matching. CPU is much much faster than RAM.
Bridging the speed gap b/w RAM, CPU.

2. Distributed systems (cluster). eg. cloud computing.
many diff. computers connected by a network.
(nodes)
sequential parallel

3. Graphics Processing Unit (many cores) ~1000s.



4. Field Programmable Gate Arrays (FPGAs) - Configurable after manufacturing
   → Specific to a particular application → create a chip

5. Application Specific Integrated Circuits (ASICs)

GPU performance - much better than CPU performance.

EVALUATION

- 5 assignments (7 + 7 + 13 + 13 + 20)
- Mid sem (20)
- End sem (20)

} Theoretical.

Amazon cloud.
Google colab. } GPU - get.

Cuda Compiler.

nvcc hello.cu

dKernel <<<1,1>>>(); Kernel Launch.

runs as single thread. on GPU

function kernel

CPU and GPU run asynchronously.

cudaThreadSynchronize();

HW

nvcc
cuda version
device query

Ref TB. Hwu, Kirk - book is slightly complicated initially.
Slides are available

14-1-2020.

- where is nvcc
- nvcc --version
- deviceQuery → CUDA SDK.
  ↓ properties of GPU device on our machine.

* Kernel "launch" in CUDA. asynchronous

```
cudaDeviceSynchronize();
dkernel<<< 1, 32 >>>();
```

product = 32 => prints 32 times.

Parallel Programming Concepts.

1. Process: a.out, notepad, chrome - program in execution. ps, top.
2. Thread: light weight process; a process or a part of it gets divided into multiple threads
3. Operating System: Windows, Linux, Android.
   OS is a software, but it manages the hardware
4. Hardware: cache, memory, cores
5. Cores: Threads run on cores; a thread may jump from one core to another.
   1 core - 1 thread at a time

Each thread has its own stack. (LIFO)

~~RAM~~ Global memory, Heap => shared across threads.

We can launch more threads than # of cores.
But it may not be a good idea to launch too many threads on CPU
→ Context switch overhead.
On GPU, it's okay to launch large no. of threads.
Ideal to launch no. of threads = no. of cores to execute parallel programs

Q) Why would a thread jump from one core to another?

```
x = a[i]   ≡ ld add of a, R1
≡          add ..., R1
           ld [R1], R2
           st R2, x
```

dependence

load => wait => wasting cycles. "sleep" => This thread
Pick up some other core's thread which is "ready"

- Thread migration improves utilization of resources.
- Programmatically, we can restrict the threads from switching b/w cores. This is called "thread pinning."

* Context of thousands of threads is stored in a dedicated set of registers in GPUs.

* warp - warp based execution.
  ↓ warp scheduling. - need not be executed sequentially.
set of threads. Need to write programs so that warp scheduling does not affect final result.

warp size - fixed for the time being (32) - NVIDIA GPUs

Q) Convert sequential C code to a parallel CUDA code.

```
#include <stdio.h>
#define N 100
int main() {
  int i;
  for(i=0; i<N; ++i)
    printf("%d\n", i*i);
  return 0;
}
```

```
#include <stdio.h>
#include <cuda.h>
#define N 100
__global__ void dkernel(){
  printf("%d\n", threadIdx.x *
                 threadIdx.x);
}
int main(){
  dkernel<<<1, N>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

output ⇒ order would be jumbled up

Variables declared inside a kernel are local to that thread.

Thread ID ⇒ ?? 0 ... N-1 so on so forth etc.

threadIdx.x

dkernel<<<1,10>>>(); thread IDs ⇒ 0...9

dkernel<<<1,20>>>(); thread IDs ⇒ 0...19

a) Convert to CUDA

```
#include <stdio.h>
#define N 100
int main() {
    int a[N], i;
    for(i = 0; i < N; ++i)
        a[i] = i * i;
    return 0;
}
```

```
#include <stdio.h>
#include <cuda.h>
#define N 100
int a[N]; ← CPU memory; cannot be accessed on GPU
__global__ void dkernel() {
    a[threadIdx.x] = threadIdx.x *
                     threadIdx.x;
}
int main() {
    dkernel<<<1, N>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

X

```
#include <stdio.h>
#include <cuda.h>
#define N 100.
__global__ void fun(int *a) {
    a[threadIdx.x] = threadIdx.x * threadIdx.x;
}
int main() {
    int a[N], *da;
    int i;
    cudaMalloc(&da, N * sizeof(int));  → allocates mem on GPU
    fun<<<1, N>>>(da);
    cudaDeviceSynchronize(); ← not essential, but good practice.
    cudaMemcpy(a, da, N * sizeof(int), cudaMemcpyDeviceToHost);
                                        Similarly Host to Device.
    for(i = 0; i < N; i++) {
        printf("%d\n", a[i]);
    }
    return 0;
}
```

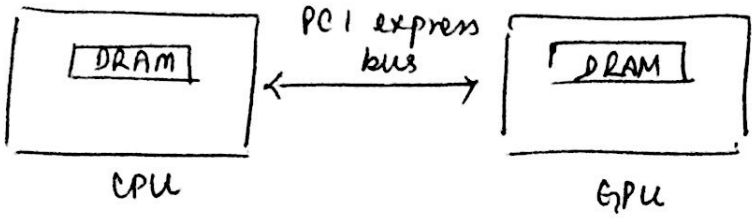
17-1-2020

GPU Hello World with a Global

define – works on CPU, GPU

global var – only CPU. `--device-- Keyword`.

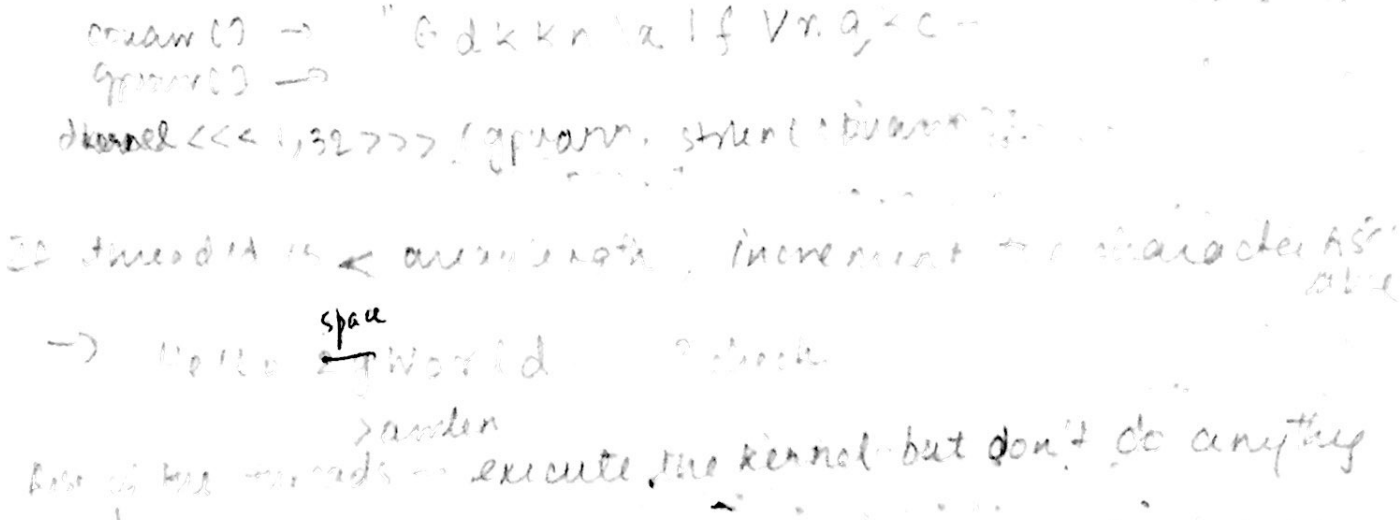
Separate memories.

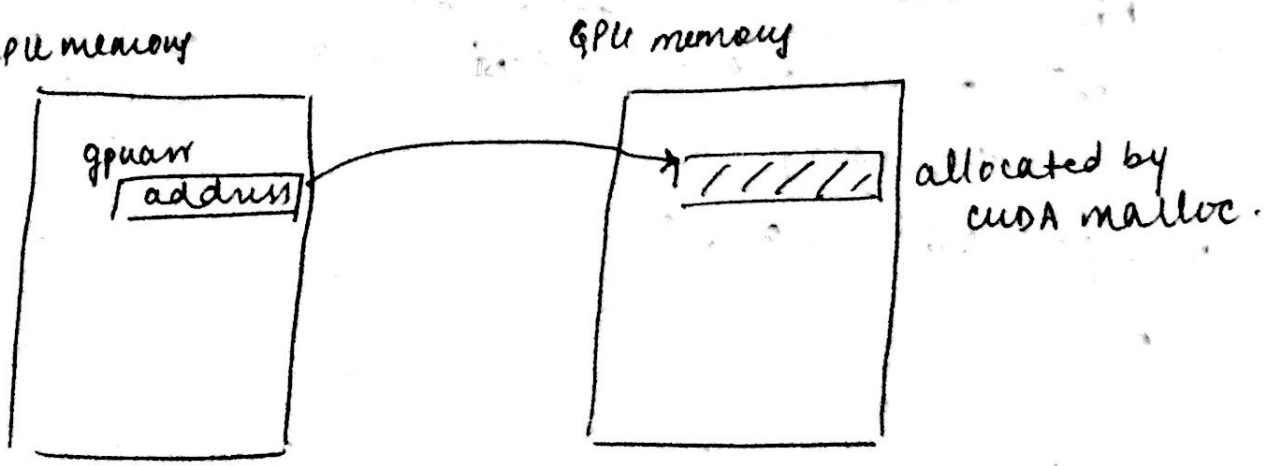


A variable in CPU memory cannot be accessed directly in a GPU kernel.

Programmer needs to maintain copies of variables.

Typical CUDA program flow →





* gpuarr – should NOT be accessed on CPU

Same address could be for CPU or GPU.

Q) What does cudaDeviceSynchronize() do? Why is it not reqd? ⇒

* GPU stream - queue.

↳ No analysis of whether GPU mem is being used

cuda Malloc	cudaMemcpy host to device	kernel launch	cuda memcpy device to host

cuda memcpy device to host → blocks.

Q) Passing strlen(cpuarr) to ~~dddd~~ dkernel?

⇒ pass by value ✓ Okay.

Problem is only when we pass by reference.

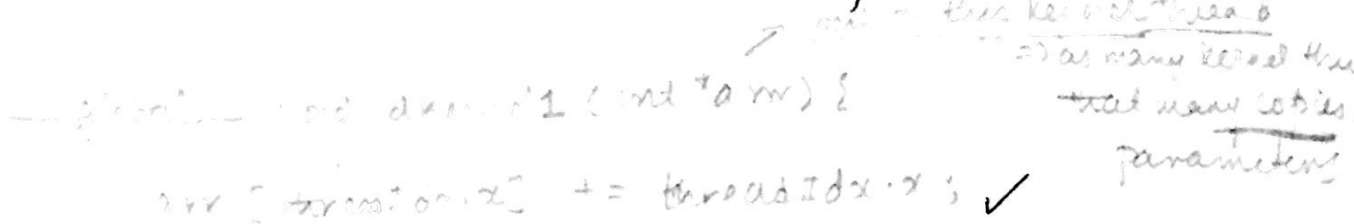
Classwork.

1. CUDA program - initialize an array of size 32 to all zeros in parallel.

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(int *arr){
    arr[threadIdx.x] = 0;
}
int main(){
    int cpuarr[32];
    int *gpuarr; // no need of CPU variable
    cudaMalloc(&gpuarr, 32*sizeof(int));
    //cudaMemcpy(gpuarr, cpuarr, 32*sizeof(int), cudaMemcpyHostToDevice);
    dkernel<<<1,32>>>(gpuarr);
    //cudaMemcpy(cpuarr, gpuarr, 32*sizeof(int), cudaMemcpyDeviceToHost);
    return 0;
}
```

2. change array size to 1024 ✓

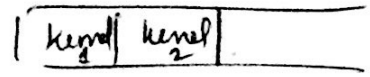
3. another kernel that adds i to array[i]



Will size of ptr vary b/w CPU & GPU? - In theory, it is possible. But cuda checks this compatibility & ensures that both sizes are same

→ add in main() also ...

4. change array size to 8000 ... looks okay.



Stream - queue

so init happens, then addition..

but... dkernel <<<1, 8000>>>

↓
not allowed
max limit ~ 1024 or so.

dkernel <<< 8, 1000 >>>

Something like that is okay.

But code doesn't work correctly. threadIdx.x → only refers to 2nd parameter, not 1st parameter of <<<, >>>

Note: Kernel call semantics - same as C function call semantics in terms of nested -- etc.

Thread Organization.

Kernel launch : grid of threads.

Grid: 3D array of thread blocks.

(gridDim.x, gridDim.y, gridDim.z)

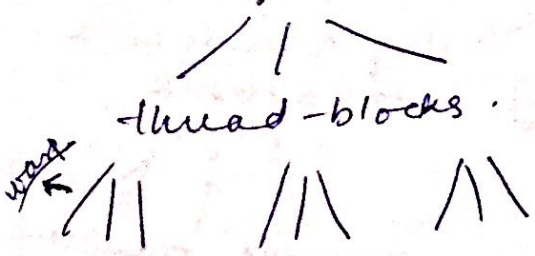
Each block → blockIdx.x, .y, .z

A thread block is a 3D array of threads

(blockDim.x, .y, .z)

Each thread → threadIdx.x, .y, .z

grid <=> kernel.



<<< 8, 1000 >>>.

↓ # thread blocks.

↓ # threads in each block.

<<< 1, 1024 >>>

→ large parallelism

v/s.

<<< 2, 512 >>>.

↑

if it is a streaming multiprocessor, thread blocks get assigned to separate hardware -

→

20/1/2020.

One kernel – one grid. (1-1 mapping)

No performance difference for 1 dim v/s 2 dim v/s 3 dim
↳ only simplifies programming ←

- 1-5 thread blocks get assigned to a streaming multiprocessor.
- Few 100s of threads assigned to a thread block.
- Total few 1000 to 100 000 threads.

dim3 : structure provided by CUDA → grid Dims

```
dim3 grid(2,3,4);
dim3 block(5,6,7);
dkernel <<< grid, block >>>();
```

block Dims

No of thread blocks in grid = 2 * 3 * 4
(blockIdx.x, blockIdx.y, blockIdx.z)

total # threads = 2 * 3 * 4 * 5 * 6 * 7

No of threads in a thread block = 5 * 6 * 7
(threadIdx.x, threadIdx.y, threadIdx.z)

for 2d → dim3 grid(a, b, 1); (not 0)

or dim2 grid(a, b);

and so on.

By default → 1d (single number).

threadIdx → thread ID within a block
blockIdx → Block ID within a grid.

For eg. program → Only 1 line of input.
2 3 4 5 6 7.

Q) If cond is changed to threadIdx.x == 0 ?

2D.

N ← 5, M ← 6

dim3 block (N, M, 1);

Though it is intended to be a matrix, it is passed as a single dimensional array.

:

kernel <<< 1, block >>> (matrix);

↑ 1 block ↑ these many threads in the block.

How to access.

matrix [tidx.x * blockDim.y + tidx.y]

a[i][j]: (i * # cols + j)

Q) Write the kernel to initialize the matrix to unique ids.

```
__global__ void kernel (unsigned *matrix) {

    unsigned uid = (tidx.x * blockDim.y + tidx.y);

    matrix [uid] = uid;

}
```

tidx.x can vary from 0, 1, 2, 3, 4

tidx.y can vary from 0, 1, 2, 3, 4, 5

tidx.x \ tidx.y	0	1	2	3	4	5
0	0	1	2	3	4	5
1	6	7	8	9	10	11
2	12	13	14	15	16	17
3	18	19	20	21	22	23
4	24	25	26	27	28	29

Q) can we access the matrix in a 2D way?

→ matrix [i][j]

Q) make it unsigned ** matrix everywhere ...?

→ we are not passing # columns. So compiler will not know how to find index.. (M) doesn't work ✗

Q) unsigned ** matrix [#columns]. ✗ works in CPU, not GPU

Why unsigned *matrix [M] doesn't work? HW. Try it out.

→ Arrays are passed by reference.

* Consecutive mem locns should hold the addresses of the M rows.
* 1 cuda malloc followed by M cuda mallocs for consecutive individual elements.
  ↳ cudaMalloc(&a...) ↑ &a[0]
* dereferencing on CPU shouldn't be done.

1D.

N ← 5, M ← 6.

dkernel <<<N, M>>>(matrix);

write corresponding kernel.

*)

```
__global__ void dkernel(unsigned *matrix){
    unsigned wid = threadIdx.x + blockDim.x * blockIdx.x;
    matrix[wid] = wid;
}
```

21-1-2020

Takeaway: One can perform computation on a multidimensional data using a 1-d block.

cudaMemcpy: blocking for CPU & GPU.

cudaMemcpyAsync - not blocking for CPU.

Launch configuration for huge data.

Good practice: <<< _ , = >>>

↑ Fix this param as constant, say 1024. (Block size).

ceil (N / BLOCKSIZE). ← could result in more no. of threads than expected.
⇒ add if condition in the kernel, to avoid access out of bounds.

if (id < vectorsize)
vector[id] = id

↓ This itself would result in an integer division
→ need to convert to floating point division.

ceil ((float) N / BlockSize);

$$\frac{N + BS - 1}{BS}$$

8). Read several points as (x,y) coords from input. For each pair of points, compute Euclidean distance in parallel. Print the maximum distance.

```
struct point { int x; int y; };
int main (int argc, char* argv[]){
  int n = atoi(argv[1]); // # pairs of points.
  struct point *matrix;   ← malloc...
  int i;
  for (i=0; i<n; i++) {
    matrix[i].x = atoi(argv[2*i]);
    matrix[i].y = atoi(argv[2*i+1]);
  }
  int *distances;  ← cudaMalloc
  kernel <<< n, n >>>;
cuda memcpy from device to host
  int max = 0; for loop — update max — printf(max); }
```

```
__global__ void kernel
    (int *dist, struct point*
     matrix){
  int tid = blockIdx.x * blockDim.x +
            blockIdx.x;
  dist[tid] = dist b/w points
    matrix[tid].x,
    matrix[blockIdx.x];
}
```

- We could launch n threads. So, a kernel will take one point and find its distance from all n-1 remaining points from it+1 onwards.

* If we have a for loop inside the kernel, then that for loop executes sequentially.

- n, n threads [element = blockIdx.x * blockIdx.x] — L, NxN threads [→ element-wise] ← check

ele1 = blockIdx.
ele2 = threadIdx.

- Maximum:

If we put `if (d > curr_max) curr_max = d;` inside the kernel,

access to/write to curr_max

race condition can occur, since it is not done in an atomic manner.

Data race.

1. Shared variable / resource
2. Atleast 1 write operation.
3. multiple parties / threads.
4. ?

- Max can be calculated sequentially on the CPU using for loop
- GPU: cuda instruction: atomic maximum

atomicMax(&max, dist) ← each thread (kernel).

- We can use locks also - sequential...
- We can launch another kernel with single thread on GPU. It will calculate max sequentially. But no need to copy the vector back from GPU to CPU
- Divide & conquer mechanism.
  Each thread takes a pair of distances & finds max.
  Keep doing this for a few more steps...
  → "Reduction" technique.

24/1/2020.

Evolution of GPUs.

1995 - 2000 : Fixed function

2001 - 2005 : Programmable shaders. - variety of ways to render graphics
⇒ if it is programmable, you can use for applns other than Graphics

2006 onwards : CUDA

GPGPU : General Purpose Graphics Processing Unit.

Nvidia → Pioneers in this field.

AMD

Other vendors : Intel, Qualcomm, ARM, Broadcom, Samsung - not very mainstream

* Vertices / Pixels
* Triangulation
* Fragments
* Shader → Video memory.

Graphics - very good setting for parallel computations.

Interesting aspect of GPU.

Data Parallelism v/s Instruction Level Parallelism

eg. pixel

Each data item is being operated upon by different threads. — GPU

SCALABLE

↑ CPU

inherent limitation due to sequential nature of code.

GPU languages.

- **CUDA** (compute unified device language).
  - proprietary, NVIDIA-specific
- **OpenCL** (open computing language).
  - universal, works across all computing devices (not only GPU) - CPU, FPGA etc.
- **OpenACC** (open accelerator).
  - universal, works across all accelerators.
    FPGA, GPU - not CPU.

I. `#pragma openacc ...` → gets ignored by GCC - only gpu compiler will look at it.
very useful in CPU — OpenMP.

```
for (- —)   // c or c++
——
```

// implicit parallelism

compiler takes care of parallelizing c code.

II. Programmer explicitly tells what to parallelize

→ CUDA, OpenCL : explicitly parallel.

Interfaces.

Python → CUDA

Javascript → OpenCL

LLVM (CPU) → PTX (GPU).

intermediate representation

Kepler family v/s Pascal family

DVFS: Dynamic Voltage Frequency Scaling.

Compute capability - versioning system
Which features are supported.

GPU RAM - much less than CPU RAM.
mainly cost issue.

top500.org - list of top supercomputers in the world.

- Ranked by performance (FLOPs)
- As of Nov 2019, 1) Summit from USA (>2.4 million cores)
  2) Sierra from USA (>1.5 million cores)
  3) TaihuLight from China (>10 million cores).

India - 57th position - Pratyush (Pune).
(Indian Institute of Tropical Meteorology, Pune, India).
119232 Cores

Matrix Squaring.

```
}
}
}
```

Q) How to parallelize?

→ Each element of result matrix can be calculated in parallel.
(one row, one column).

```
__global__ void squareGPU (unsigned* matrix, unsigned* result, unsigned matrixSize) {
    unsigned ii = blockIdx.x ...   ⟹ Same okay...
    unsigned jj = ...
    for (unsigned kk = 0; ...
```

24-1-2020

Matrix Squaring - Parallelize i loop (outer loop).

GPU - lower clock frequency than CPU's.

So sequential code on GPU is slower than on CPU

→ Slower than CPU

⇒ Parallelization of i loop alone is not able to compensate for the GPU's lower clock frequency.

Parallelize both i and j loops

```
dkernel <<< N, N >>> ( matrix, result, N);
__global__ void dkernel ( matrix, result, matrixsize) {
unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
unsigned ii = id / matrixsize;
unsigned jj = id % matrixsize;
for (unsigned kk = 0; kk < matrixsize ; ++kk){
    result [ii*matrixsize + jj] += matrix [ii*matrixsize + kk]
                                 * matrix [kk*matrixsize + jj];
}
```

(HW) What happens when you interchange ii, jj?

Sometimes we deliberately make a thread more sequential to improve performance ~.

When we try to parallelize k loop also - race condition may arise Threads will have to coordinate among each other ⇒ Overhead will result.

ii, jj loops are "embarassingly parallel" - scales very well

kk loop is not.

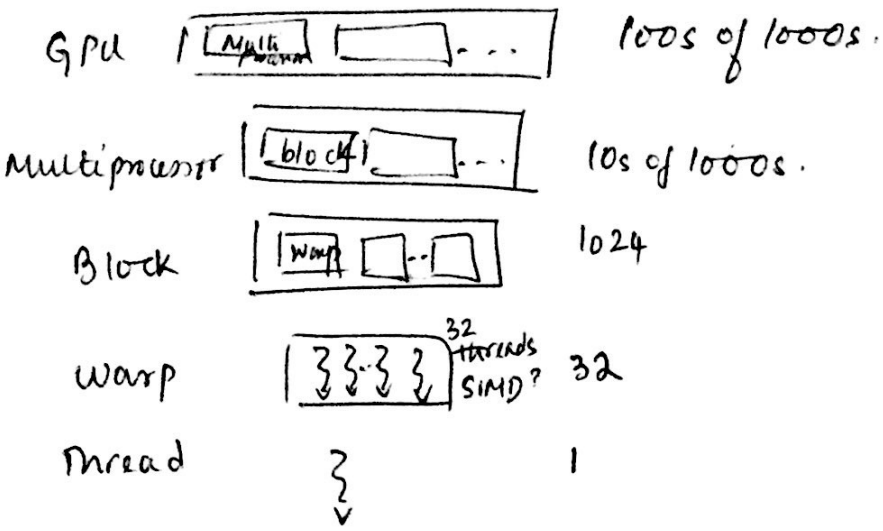
When no. of cores is increased beyond a limit, Then physical architecture / organization comes into picture.

→ Systemic aspects come into picture.

PRAM model - theoretical model.

Von Neumann architecture.

GPU computation Hierarchy.



Each core in a multiprocessor executes one thread.

Multiple grids could be simultaneously executing on a GPU

27/1/2020.

Warp : SIMD. - Single Instruction Multiple Data

* warp threads are fully synchronized.
* Implicit barrier after each step/instruction.

Lock step execution is only true for threads within a warp.

* Hardware takes care of adding the NOPs.

* Thread privatisation
Modify local copies in each thread; after barrier, merge the computations of each thread

When diff warp threads execute different instructions, threads are said to diverge. Hardware executes threads satisfying same condition together, ensuring that other

threads execute no-op. This adds sequentiality to the execution.
⇒ "Thread divergence"

How to avoid?

1) Carefully remove if conditions
2) Reorganize data / computation. - separate chunks.

(*) Simple if conditions are not as much a problem as for loops, with different # iterations for diff warp threads

→ Reduce disparity across amt of work that each warp thread is doing. (Bucket sort, in some sense).
→ reduces thread divergence within a warp.

* Static ✓ bucket sort
* Load time: inspect data, then do quiet part...
* dynamic: can't use any sorting method..

Degree of Divergence.
DoD for a warp is the # of steps reqd to complete 1 instruction for each thread in a warp.

without any thread divergence, DoD = 1
for fully divergent code, DoD = 32.

CW: Write a code to achieve DoD = 4.

```
int *matrix, int N
__global__ void kernel () {
    int uid = blockIdx.x * blockDim.x + threadIdx.x;
    if (uid % 4 == 0) {
        matrix [uid % N] = uid;
    }
    else if (uid % 4 == 1) {
        matrix [uid % N] = uid + 1;
    }
    and so on...
}
```

Q) Calculate degree of divergence of :

case 0

vector[id] = 0

case 1

vector[id] = vector[id]

case 2

vector[id] = vector[id-2]

case 3

vector[id] = vector[id+3]

case 4

vector[id] ≠ 4 + 4 + vector[id].

case 5

vector[id] = 5 - vector[id].

case 6.

vector[id] = vector[6].

case 7

vector[id] = 7 + 7.

case 8

vector[id] = vector[id] + 8

case 9

vector[id] = vector[id] * 9.

default.

Answer - dependson Compiler optimisations

11 10 9

gcc by default -O0

-O1 } for performance
-O2 } optimisation.
-O3 }

nvcc

→ by default -O3

(i)
There can be conditions in the kernel. But there needn't be divergence.

if (vectorsize < N) S1; else S2;

(ii) Conditions evaluating to diff. truth values.
→ Need not lead to divergence

eg. if (id/32) S1; else S2;

∴ no divergence within a warp.
∴ evaluates to same truth value

(iii) Conditions evaluating to diff truth values within a warp

28-01-2020

Memory.

CUDA Memory model overview.

- Within a thread block: shared memory.
- B/w blocks - ~~can~~ share global memory.

Shared memory - similar to L1 cache.
↳ or scratchpad cache.

We can decide, to a certain extent, what goes in this 'cache'. ⇒ on chip

Multiprocessor has a separate register file.
- Registers are also available ~~with~~ for each thread. - split among the threads.

"# registers is good" for a particular thread if we give the "right" no. of threads - else register spills. ↳ goes into global memory ↳ poor performance.

Q) How to check if # registers is good enough?

NVCC gives a warning...

- Global / video memory
  - → main means of communicating data b/w host & device (PCI express bus).
  - → contents visible to all GPU threads.
  - → longer latency. (400 - 800 cycles)
  - → throughput ~200 GBPS.
- Texture Memory. (smaller)
  - → Read only (12 KB)
  - → ~ 800 GBPS
  - → Optimized for 2D spatial locality

- Constant Memory.
  - → Read only (64 KB).

Bucket Sort.

We have marks from 0-100.
Want to divide into buckets 0-10, 10-20 .. 90-100.

1) Sort in O(nlogn), then do a linear pass, O(N).
2) Push each element/mark into a separate vector (10 diff. vectors: 0-10, 10-20 ... 90-100).
   → O(n) time.

* cudaGetLastError() to catch errors on GPU.

SM: Streaming Multiprocessor

- L2 cache.
  - → 768 kB cache
  - → shared among SMs
  - → fast atomic instructions can be supported.

- L1$/ shared memory.
  - → configurable 64 kB per SM.
  - → 16 kB shared + 48 kB L1 or vice versa
  - → low latency (20-30 cycles)
  - → high bandwidth (~1 TBPS).

- Registers. unified,
  - → 32 K in number per SM.
  - → Max ~21 regs per thread.
  - → very high bandwidth (~8 TBPS)

Bandwidth.

- wide data bus rather than fast data bus
- Parallel data transfer.
- Techniques to improve bandwidth:
  - Share/ reuse data
  - Data compression. (decompress locally).
  - Recompute rather than store + fetch

Latency.

- Time reqd for I/O / data transmission.
- Ideally, it should be zero. But not possible. So it should be minimised as much as possible.
  - Procesor should have data available in no time.
  - memory I/O is bottleneck.
- Latency can be reduced using caches.
  - CPU: Regs, L1, L2, L3, L4, RAM
  - GPU: small L1, L2, many threads

CPU – ILP: take another instruction.

GPU – take another thread block.

- Latency hiding on GPUs is done by exploiting massive multithreading, rather than ILP.

Individual latencies can be large. But overall throughput is maximized. The GPU keeps itself busy all the time as far as possible.

hiding latency via pipelining

* May lead to super-linear speedup.

Locality.

- Only a small cache is available per thread.
- In GPU, another form of spatial locality is critical.

⇒ Across threads

All threads in a thread block access their L1 cache.

Pascal GPU

64 KB L1 cache

16 KB + 48 KB / 48 KB + 16 KB.

(L1 & scratchpad)

29-1-2020.

locality.

1. Spatial

If a[i] is accessed, a[i+k] would also be accessed soon.

```
for(i=0, i<N; i++)
    a[i] = 0;
```

2. Temporal.

If a[i] is accessed now, it would be accessed again soon.

```
for(i=0; i<N; i++) {
    a[i] = i;
    a[i] += N;
    b[i] = a[i] * a[i];
}
```

a[i] - temporal locality
b[i] - no temporal locality

These localities are applicable to ~~both~~ CPUs & GPUs, but more so on CPUs.

(cw)

```
for(i=0; i<M; ++i)
I   for(j=0; j<N; ++j)
        for(k=0; k<P; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

```
II
for(i=0; i<M; i++)
    for(k=0; k<P; ++K)
        for(j=0; j<N; ++j)
            C[i][j] += A[i][k] * B[k][j];
```

On CPU, II takes half as much time as I.

$i \longrightarrow k \longrightarrow j$ ✓

DAG - Topological order of access patterns.

Bunch of dependencies : construct DAG ; use topo order for efficient cache accesses.

Q) What about GPU?

Memory coalescing.

- If consecutive threads access words from the same block of 32 words, their memory requests are clubbed into one. → to avoid warp stalling (recall: barrier...)

eg. If a[id] is accessed by thread 'id', then a[id+1] is also brought into the cache of thread 'id+1'.

"Memory requests are coalesced"

→ "stride" a[i] → a[20*i + 1] Strided access pattern.

as opposed to a[i+1]

- Memory coalescing can be effectively achieved for regular programs (such as dense matrix operations).

  a[id] ✓ a[id+5] ✓ (check) (no need to be at boundary of 32)

- CUDA can be associated with C as well as ~~Fortran~~
  ↑ row major access ↑ column major access.

- Coalesced – doesn't matter which order the memory is accessed

- Not only read accesses, but also write accesses are coalesced.

- Vectorization on Intel processors → somewhat similar to this
  * Large instruction / Vector Instruction
    → Same instruction, multiple data
    → sequential program
    → handled by hardware.

Intel compiler: icc → automatically vectorizes "good" access patterns.

- Degree of coalescing.
  – DoC is the inverse of (32 –) The no. of memory txns reqd for a warp to execute an instruction.

  eg. a[threadIdx.x] has DoC of ~~32 (31?)~~ 31

  a[rand()] likely has a DoC of 0.

CW: Write a kernel to vary the degree of coalescing from 0 to 32 based on an input argument.

```
__global__ void kernel (int doc) {
    1) access a[threadIdx.x * doc]
    2) declare a[doc];
}
```

3

31-1-2020 Tutorial Session

03-02-2020.

Memory Coalescing.

- Coalesced - deterministic sequential access
- Strided access - sequential access but in chunks.
- random access - < a[rand()]
  a[input[id]]: input-dependent pattern.
  ↳ eg. bucket sort -
  graphs representation.
  edges/ neighbours of a vertex.

① CPU - → spatial locality
- Each thread should access consecutive elements of a chunk (strided).
- Array of Structures (AOS) has better locality.

② GPU - → Coalescing
- A chunk should be accessed by consecutive threads (coalesced).
- Structure of Arrays (SoA) has a better performance.

AoS v/s SoA.

AoS.

```
struct node {
    int a;
    double b;
    char c;
};
struct node allnodes[N];
```

Expectation: When a thread accesses an attribute of a node, it also accesses other attributes of the same node.

LOCALITY
BETTER ^ ON CPU

SoA.

```
struct node {
    int alla[N];
    double allb[N];
    char allc[N];
};
```

Expectation: When a thread accesses an attribute of a node, its neighbouring thread accesses the same attribute of the next node.

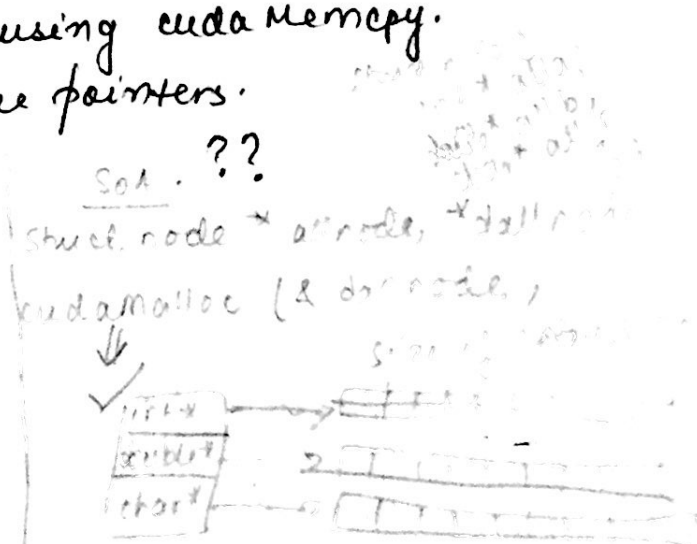
COALESCING
BETTER ^ ON GPU.

Q) Write codes for the two types using cudaMemcpy.
Note that all arrays would be pointers.

AoS.

```
struct node {
    int a;
    double b;
    char c;
};
struct node* allnodes;
struct node* dallnodes;
cudaMalloc(&dallnodes,
           N*(sizeof(
               struct node)));
cudaMemcpy(dallnodes, allnodes,
           N*(sizeof(struct node)),
           cudaMemcpyHostToDevice);
```

SoA. ??



(I)

(II)

(III)

```
cudaMalloc(
                                    // a/b
                                    // a/b
                                    ) //malloc
```

* malloc() is available on GPU.

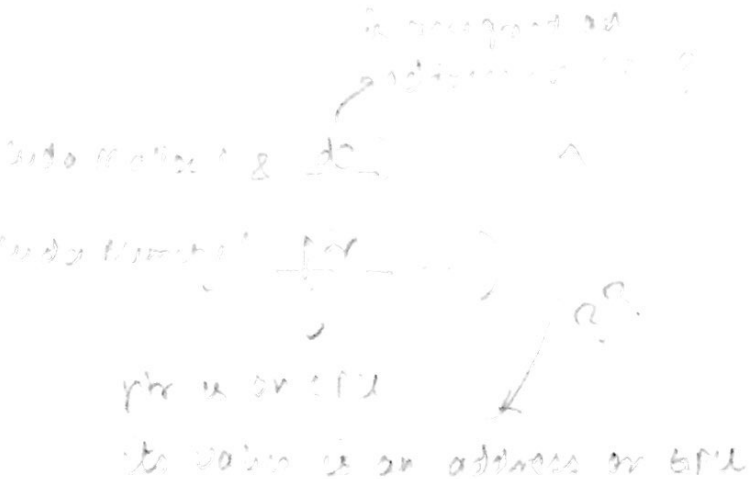
SOA takes almost 1/3rd the time as AOS.

04/02/2020.

Classwork.

Copy a linked list from CPU to GPU

- Each node contains roll no, name, facad.



Struct on CPU,

ptr on CPU which points to GPU memory

cudaMalloc & c.o.

CW.


```
struct node {
    char* rollnumber;
    char* name,
    char* facad,
    struct node *next;
};

struct node *headptr;
struct node * gpuhead,*gptr;
```

cudaMalloc(&gptr, sizeof(struct node));

cudaMalloc(&gpu ptr → roll no)

→ name

→ facad

→ next

cudaMemcpy(gptr →

→ name

→ facad

Recursive code :

returns pointer to linked list in GPU

final head of

```
node * getLinked (node *root) {

    if (root == NULL) return NULL;

    node *ptr;   // ptr will point to a node in GPU

    cudaMalloc (&ptr, ...);

    node *rptr = getLinked (root -> next);

    struct node *copy = root;

    copy -> next = rptr;

    cudaMemcpy (ptr, copy, ...);   // copy data in root along with ptrs.

    return ptr;

}
```

HW. Stolen parallel implementation. — without array out of bounds exception

Idea: Break up the given string into some number of small chunks. Each chunk's length is found sequentially by a thread. If '\0' is found we know that the entire string is traversed. Else retrace