HP3 Notes

CUDA

1. RISC n CISC: Two different philosophies.
2. Elementary CPU MIPS datapath executes different instructions with variable delays
3. Pipeling has structural, control and data hazards
4. Talking about Memory hierarchy, caches, write policies, mapping
5. CPI = Ideal CPI + Stalls
6. Handle hazards using hardware (RAW) and compiler techniques (WAR, WAW)

7. Data parallel algorithms: Vector processors (vector registers, vector func. units, handles as vectors), SIMD, GPU
8. In GPUs, Scalar Processors (SP) together make a Streaming Multiprocessor (SM, has own shared memory, cache, etc.), which together make a Texture/Processor Cluster (TPC)
9. GPGPU SM has 8 SP cores (scalar MAD), special func. units, multi threaded instruction fetch and issue unit, an I-cache, a c-cache, read write shared units.
10. SIMT, single instruction to multiple lanes (thread)
11. MT unit, create and manages threads in groups of 32 called warps
12. Warp threads are allotted to the SP, and single warp can be executed across many cycles.
13. SMs also has a register file (as a register lane), a load store unit with the same number of SPs.
14. Local memory per thread private, in DRAM. Shared memory & L1 cache in SM for low latency access. L2 is unified for all SMs.

15. Host code (CPU) C compiler, device code (GPU) device JIT compiler
16. cudaMemcpy to transfer between device and host.
17. Kernel launch runs a grid of threads, organised as blocks. C[i] = A[i] + B[i]
    i=blockIdx.x*blockDim.x+threadIdx.x
18. gridDim, blockDim, blockidx, threadidx can be used as identifiers for a thread. Dim3 used for dimensions of blocks and grids

19. 1024 threads maximum in a block
20. 1 Grid - > Block -> threads
21. blockNum = blockIdx.z * (gridDim.x * gridDim.y) + blockIdx.y * gridDim.x + blockIdx.x
    threadNum = threadIdx.z * (blockDim.x * blockDim.y) + threadIdx.y * (blockDIm.x) +
    threadIdx.x
    globalThreadId = blockNum * (blockDim.x * blockDim.y * blockDim.z) + threadNum
22. Each block can execute in any order relative to other blocks. Enables scalability

23. syncthreads() block level synchronisation barrier

```
int main()
{
  int N=1024;
  int size_M=N*N;
  int size_V=N+1;

  cudaMemcpy(d_M,M,size_M*sizeof(float),
  cudaMemcpyHostToDevice);
  cudaMemcpy(d_V, V, size_V*sizeof(float),
  cudaMemcpyHostToDevice);
  dim3 grid(1,1,1);
  dim3 block(N,1,1);
  sumTriangle<<<grid,block>>>(d_M,d_V,N);
  cudaMemcpy(V,d_V,size_V*sizeof(float),
  cudaMemcpyDeviceToHost);

}
```

24.

```
__global__
void sumTriangle(float* M, float* V, int N){

int j=threadIdx.x;
float sum=0.0;
for (int i=0;i<j;i++)
        sum+=M[i*N+j];

V[j]=sum;
__syncthreads();

if(j == N-1)
{       sum = 0.0;
        for(i=0;i<N;i++)
                sum =sum + V[i];
        V[N] = sum;
}
```

Sample Sync prog.

25. Each SM has registers, caches, warp schedulers, SFUs
26. In a resource constrained situation, threads > SPs, execution is serialised.
27. 1024 thread contexts max per SM
28. A collection of threads called warp (unit of thread scheduling) is physically guaranteed to be executed in parallel.
29. Each warp executes the same instruction across the parallel threads
30. Issue one warp which has all operands and other resources like functional units ready per cycle in a round robin fashion.
31. The assembler maintains internal masks, a branch synchronisation stack and special markers to handle divergence inside a warp. Depending on the value of the mask,

corresponding threads execute. Once one block is finished, mask is popped, flipped and pushed back.

32. Reconvergence (the endpoint after the immediate divergence), Target (next after handling current), the stack (for handling the masks) are maintained for every branch.
33. Syncing across different blocks can be achieved by terminating kernel.

34. GPU coalesces global memory requests by a warp of threads, typically 32 4 byte words.
35. When aligns as a 32 (or similar) bytes, can finish in 1 transaction, else requires multiple, in case of access with offsets (in most cases). In single transaction, 32 4 byte words are usually accessed in the global memory.
36. For matrix multiplication, divide the matrix into tiles that fit into shared memory and load them. Then use them to multiply with the help of sync threads.
37. Bank conflicts occur when multiple memory access requests to the same bank of the memory are made from multiple threads. Padding can be done to make them move away from the same partition or bank. Diagonal block reordering can help
38. Lot of profiling things in this chapter.

39. Decompose reductions into multiple kernel launches
40. 1: Add 2 consecutive elements. Half the threads are inactive
41. 2: Reduce divergence by making consecutive threads active in a strided manner, final portion of threads inactive.
42. 3: Replace strided index
43. 4: Inactive threads are also made active by using twice the block size of data
44. 5: Unroll loops
45. Bitonic Sort: Convert arbitrary sequence to bitonic sequence. Convert bitonic to sorted.
46. BS(1, n) = BS(1, n/2, 0), BS(n/2, n, 1), BM(1, n) | BM (1, n) = for loop [i, i + n/2], BM(1, n/2), BM(n/2, n)
47. Prefix Sum: Hills Scan (for d = 1 to logn do; forall k >= 2 power (d-1) do; x[k] = x[k-2p(d-1)] + x[k])
48. Blelloch Scan: Build a balanced binary search and sweep it to and from the root

49. Loop fusion, classic compiler optimisation. Reduces kernel call.
50. Inner thread fusion merges two different kernel into a single one where they operate on the same thread. Not done when not same number of threads or unbalanced workload
51. Inner block fusion merges two different kernel into a single one where they operate on the same corresponding block, but different threads. Not done when sync is involved.
52. Inter block fusion. Same idea, but work shared between blocks (of the same grid).
53. Thread coarsening is merging multiple threads into one, that is, works on multiple data as opposed to the previous one. Allows reuse of result. Less number of threads needed. Increases utilization.
54. Coarsening factor is the number of threads merged into one.

55. Thread level coarsening combines two or more threads from the same block. Each block performs the same amount of work but fewer threads.
56. Stride length is the offset between the threads to be combined. Greater than warp length to ensure coalescing
57. Block level coarsening combines work of several thread blocks into one. Thread count remains same in a block, but handles increased workload.
58. Various resources inside the SM are the limiting factors for coarsening.
59. Cache pressure will also increase due to coarsening.

OPENMP

60. Multi core processors have all processors in same chip and follow MIMD instructions.
61. Simultaneous multi threading allows multiple independent threads to operate on the same core
62. Each thread has an affinity mask telling what cores the thread can run on

63. Openmp is a compiler directive parallelisation library
64. HPC is only possible via parallelism
65. G++ -fopenmp. #pragma omp parallel OMP_NUM_THREADS > omp_set_num_threads > num_threads clause
66. private -> variable type in directive. Private to each thread.
67. #pragma omp for - parallelize for loops

68. #pragma omp single - serial part of the parallel region
69. #pragma omp master - serial part executed by master thread
70. Static, dynamic, guided, run time scheduling for #pragma omp for
71. #pragma omp for collapse - running perfectly nested loops
72. Firstprivate initalises variable with value before parallel region, Lastprivate retains the last value for the variable after the parallel region
73. #pragma omp critical, atomic - critical section
74. #pragma omp barrier - threads wait till all of them reach that point
75. #pragma omp for reduction - reduces variable according to function

76. Care about deadlocks. Don’t use omp barrier in sections, critical, atomic, single, etc.

MPI

77. MPI: Message passing and shared memory. MPI is used to communicate between processes that have different address space.
78. Send, receive, synchronisation. Sender cooperates with destination.

79. MPI_Init, MPI_Finalize present in all programs. Number of processes is given in command line arguments.
80. MPI_Comm_size (total processes), MPI_Comm_rank(process no.) are identifiers

81. Communicators are groups of processes. MPI_COMM_WORLD includes all processes.
82. MPI_Abort terminate all processes
83. Communication takes place within a communicator only. Source and destination identified by rank.
84. MPI_Barrier synchronises all processes within a barrier
85. MPI_Send, MPI_Recv (Blocking), can accept wildcards in Recv.
86. MPI_Broadcast
87. MPI has its own defined types and derived ones.
88. MPI_Scatter: Root sends a message and the i the segment is passed to the ith process.
89. MPI_Gather: Inverse of scatter
90. MPI_Reduce, Allreduce
91. MPI_Isend, Irecv, non blocking
92. MPI_Alltoall, matrix transposition