Arunima Singh Thakur, 18090S218, Sec C, Roll no. 31, Branch CSE, PCAP In-Sem,

Arus

1) a) Application point of view (Diag on next page)

The computers are experiencing a trend of 4 ascending levels of sophistication.

- Data processing — Data objects include numbers, characters, images, audio, video, multidimensional measures, etc.
- Information processing - Information item is a collection of data objects that are related by some syntactic structure or relation.
- Knowledge processing — Knowledge consists of information items plus some semantic meanings.
- Intelligence Processing - Intelligence is derived from a collection of knowledge items

b) Operating system point of view

The computer systems have improved chronologically in 4 phases :-

- Batch Processing
- multiprogramming
- Time sharing
- Multiprocessing

↓ Parallel processing of information increases sharply by exploiting concurrent events

Concurrency : Implies parallelism, simultaneity, & pipelining

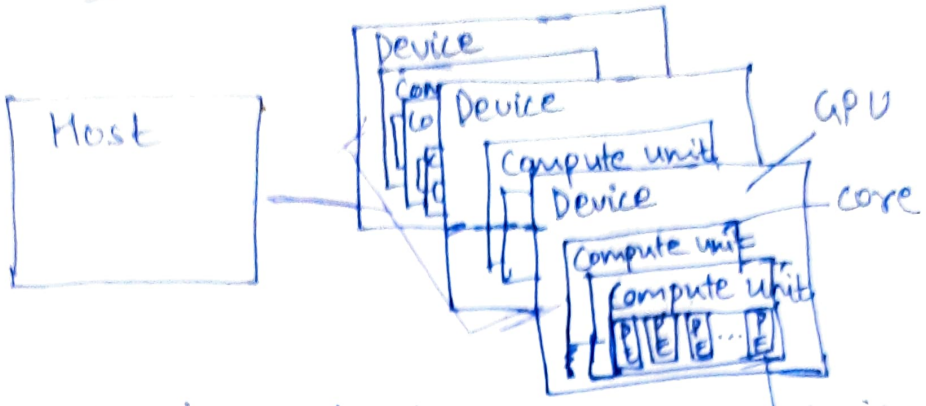
Parallelism : Parallel events may occur in multiple resources during the same time interval

Simultaneity : Simultaneous events may occur at the same time instant

Pipelining : Pipelined events may occur in

overlapped time spans

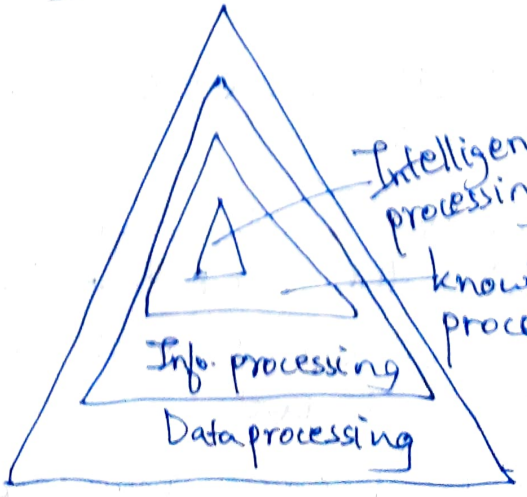
5) The platform model defines a device as an array of computer units, with each compute unit functionally independent from the rest. Compute units are further divided into processing elements.



- The pointer is populated with the available number of platforms. The programmer can then allocate space to hold the platform information
- for a second call, a cl_platform_id pointer is passed with enough space allocated for num_entries platform.
- After platforms have been discovered, the clGetPlatformInfo() call can be used to determine which implementation (vendor) the platform was defined by

1.) (continued)

Inc. vol. of raw material to be processed



Inc. complexity & sophistication in processing

2) Array Computers

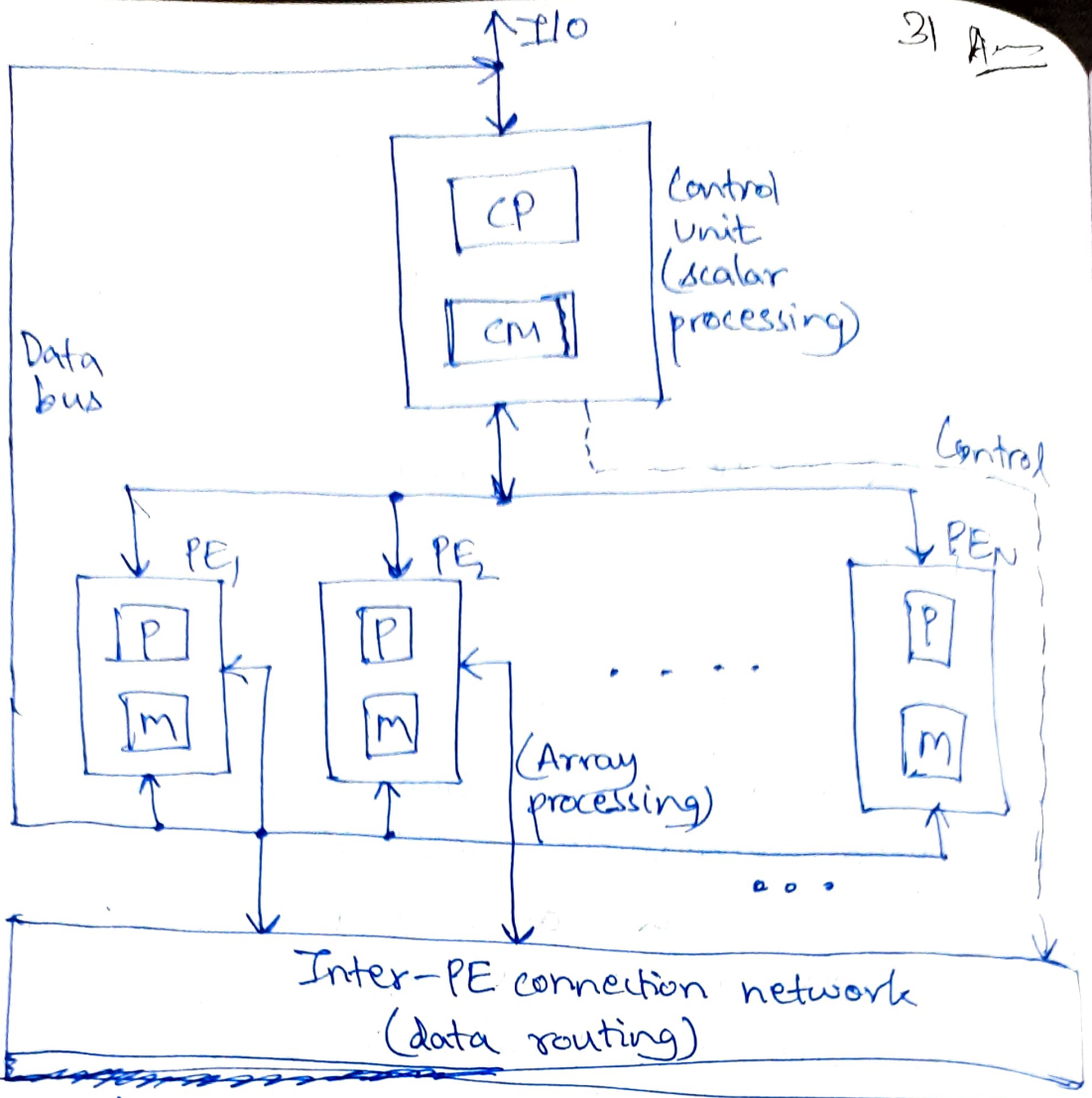
- By replication of ALUs, we can achieve the spatial parallelism
- The PEs are synchronized to perform the same function at the same time
- Scalar & control-type instructions are directly executed in the control unit (CU)
- Each PE consists of an ALU with registers & a local memory

whereas Multiprocessor systems

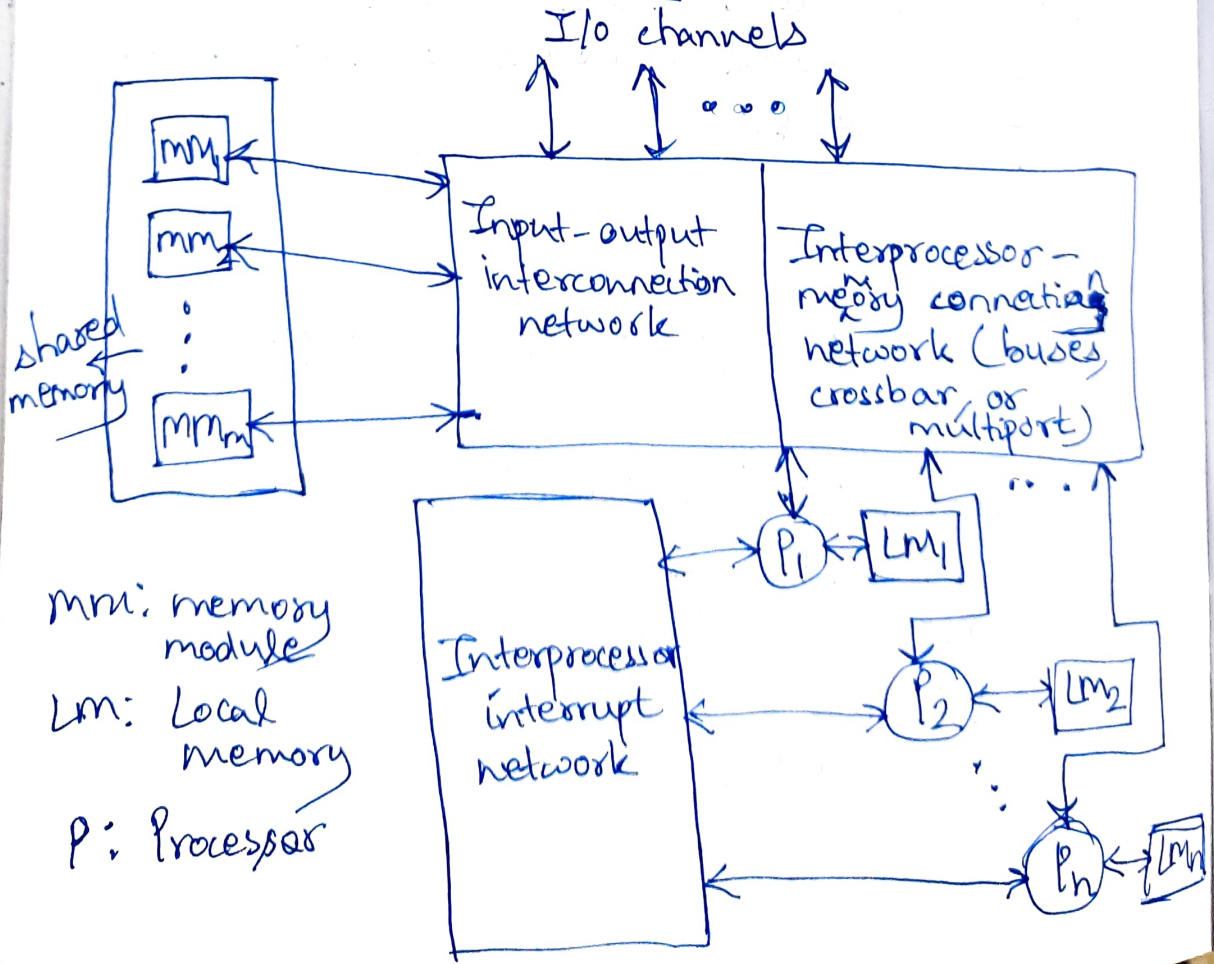
- The system contains two or more processors of approx. comparable capabilities
- All processes share access to common sets of memory modules, I/O channels, & peripheral devices
- Besides the shared memories & I/O devices, each processor has its own memory
- Interprocessor communications can be done through the shared memories or through an interrupt network.

Array Computers diag. (next page)

PE: processing elements
CP: Control processor
CM: Control memory
P: Processor
M: memory



Multiprocessor system



mm: memory module

LM: Local memory

P: Processor

3) A modern GPU is organized into an array of highly threaded streaming multiprocessors (SMs)

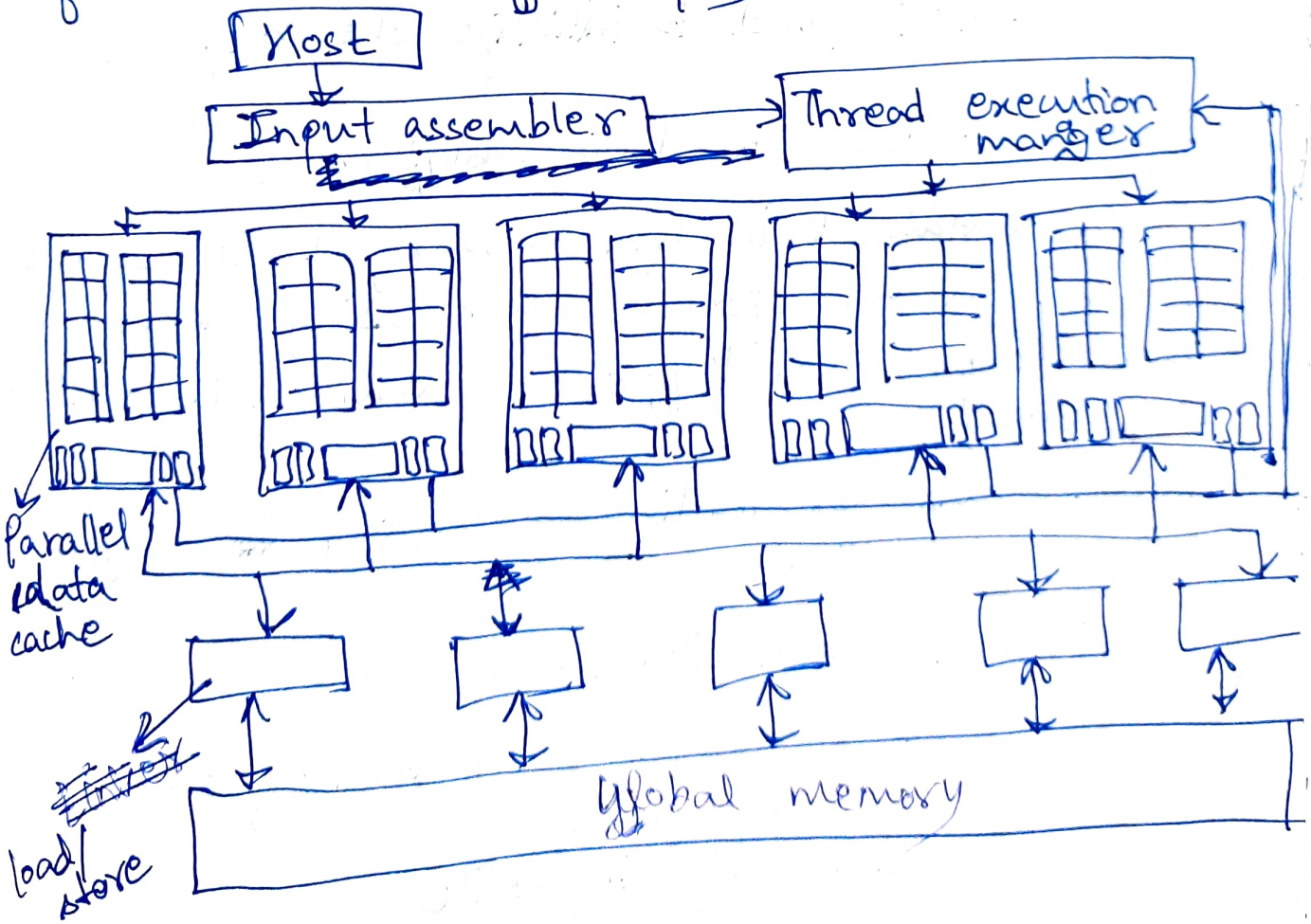
GPUs are high latency high throughput processors.

GPUs are designed for tasks that can tolerate latency.

GPUs can have more ALUs for the same sized chip & therefore run many more threads of computation. (Modern GPU run 10,000s of threads concurrently). Threads are managed and scheduled by the hardware

GPUs use data parallelism; SIMD (Single Instruction multiple Data) streams. Same instruction is run on multiple data streams.

GPUs have higher memory bandwidth, and therefore are suited for display devices.



b) Create device buffers

- The API function clCreateBuffer() allocates the buffer & returns a memory object

```
cl_mem clCreateBuffer(
        cl_context context,
        cl_mem_flags flags,
        size_t size,
        void *host_ptr,
        cl_int *errcode_ret)
```

- Creating a buffer requires supplying the size of the buffer & a context in which the buffer will be allocated; it is visible for all devices associated with the context.
- The host_ptr is used to initialize the buffer.

Write host data to device buffers

- Data contained in host memory is transferred to & from an opencl buffer using the commands clEnqueueWriteBuffer() & clEnqueueReadBuffer() respectively.
- The API calls for reading & writing to buffers are very similar

```
cl_int clEnqueueWriteBuffer(
        cl_command_queue command_queue,
        cl_mem buffer,
        cl_bool blocking_write,
        size_t offset,
        size_t cb;
        const void * ptr,
        cl_uint num_events_in_wait_list,
        const cl_event * event_wait_list,
```

cl_event * event)

31

Parameters

- command_queue - refers to the command-queue in which the write command will be queued
- blocking_write - indicates if the write operation is blocking (CL_TRUE) or non-blocking (CL_FALSE)
- offset in bytes in the buffer object to write to.
- cb is the size in bytes of data being written.

---

8) b) (continued)

a) Deadlock (Recv-Recv)

```
int a, b, c;
int rank;
MPI_Status status;
....
if (rank==0)
{
}
```

To create device buffers the API function clCreateBuffer() is used to allocate the buffer which ret. a memory obj. Suppose we want to create 2 input arrays A & B & 1 output array C

```
buffer A = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
buffer B = clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL, &status);
bufferC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize, NULL, &status);
```

To write an input array to OpenCL buffer we use clEnqueueWriteBuffer() & For reading we

use clEnqueueReadBuffer(). To write array A & B to device buffer,

31 Ans

status = clEnqueueWriteBuffer(cmdQueue, bufferA, CL_FALSE, 0, datasize, A, 0, NULL, NULL);

status = clEnqueueWriteBuffer(cmdQueue, bufferB, CL_FALSE, 0, datasize, B, 0, NULL, NULL);

Size & context is supplied while creating buffer as datasize & context resp. Flag parameter makes it read-only, write only or read-write type.

While writing the buffer, cmdQueue refers to the cmd queue in which it is queued & Blocking-write is set to false.

---

10) Given grid length in X dir = 32

256 block arranged in 2-D

→ ∴ grid length in y dir = y

$32 * y = 256$

$y = \frac{256}{32} = 8$

∴ grid length in y dir = 8

⇒ Grid length in z = 1 (size its 2-D)

→ Thread in a block is arranged in 3D

block length in x dir = 3

block length in y dir = 4

Total no. of threads in a block = 60

∴ block len (in x) * (in y) * (z) = 60

∴ block len in z = $\frac{60}{3 \times 4} = 5$

∴ block len in z = 5

→ Global thread id of a thread (2,1,3) in block (2,8,7)

blockId = blockIdx.x + blockIdy.y gridDim

= 98 + 7 × 32 = 252

ThreadId = blockId * (blockDim.x * blockDim.y * blockDim.z) + (threadIdx.z * (blockDim.x * blockDim.y)) + (threadIdx.y * blockdim.x) + threadIdx.x

= 252 * (3*4 × 5) + (3 × (3+4)) + (1+3) + 2

= 252 × 60 + 36 + 5 = 15161

→ Global block id of (2,2)

block id = (gridDim.x * blockIdx.y) + blockId.z

= 2 + (32 * 3)

= 98

Grid Dim x	Grid Dim y	Grid Dim z	block dim x
32	8	1	3
block dim y	block dim z	Global thread id	Global block
4	5	15161	98

8) a) Deadlock (Recv-Recv)

```
int a,b,c;
int rank;
MPI_Status status;
.....
if (rank == 0)
{
```

3) Ans

```
MPI_Recv(&b, 1, MPI_INT, 1, 0,
    MPI_COMM_WORLD, &status);
MPI_Send(&a, 1, MPI_INT, 0, MPI_COMM_
    WORLD);
c = a+b/2
}
else if (rank == 1)
{
MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_
WORLD, &status);
MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_
    WORLD);
c = a+b/2;
}
```

b) Deadlock (Tag mismatch)

```
int a, b, c;
int rank;
MPI_Status status;
_____;
if (rank == 0)
{
MPI_Send(&a, 1, MPI_INT, 1, 1, MPI_COMM_
    WORLD);
MPI_Recv(&b, 1, MPI_INT, 1, 1, MPI_COMM_
    WORLD, &status);
c = a+b/2;
}
else if (rank == 1)
{
MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_
    WORLD);
MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_
    WORLD, &status);
```

```
    c = a+b/2;
}
```

c) Deadlock (Rank mismatch)

```
int a, b, c;
int rank;
MPI_Status, status;
_____.
if(rank==0)
{
    MPI_Send(&a, 1, MPI_INT, 2, 1, MPI_Comm_
                                    WORLD);
    MPI_Recv(&b, 1, MPI_INT, 2, 1, MPI_Comm_
                WORLD, &status);
    c = a+b/2;
}
else if(rank==1)
{
    MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_Comm_
                                    WORLD);
    MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_Comm_
        WORLD, &status);
    c= a+b/2;
}
```

---

7) MPI_Collective Communication Encryption

```
#include "mpi.h"
#include<stdlib.h>
int factorial(int n){
    if(n>=1)
        return n*factorial(n-1);
    return 1;
}
```

```
int main(int argc, char *argv[]) 31, A
{
    int rank, size;
    int local_sum = 0, fact = 1, substr_len = 0
    int ENC[16];
    const int n = size;
    char *str;
    char substr[16];
    MPI_Init(&argc, &argu);
    MPI_CommRank(MPI_COMM_WORLD, &rank);
    MPI_CommSize(MPI_COMM_WORLD, &size);
  if(rank = 0) {
    printf("Enter String with len %d zero", n);
     gets(str);
     if(strlen(str) % n != 0){
            exit(1);
     }
    substr_len = strlen(str)/n;
  }
  MPI_Bcast(substr_len, 1, MPI_INT, 0, MPI_
                                COMM_WORLD);
 MPI_Scatter(str, substr_len, MPI_CHAR,
  substr, substr_len, MPI_CHAR, 0, MPI_COMM_
                                WORLD);
 printf("Process with rank %d received substring
   %s \n", rank, substr);
    for(int i=0; i< substr_len; i++) {
       if(substr[i] >= '0' && substr[i] <= '9')
            local_sum += substr[i] - '0';
 }
```

```
fact = factorial(rank+1);
int Scan_Sum = 0;
MPI_Scan(&local_sum, &Scan_Sum, 1, MPI_INT,
  MPI_Sum, MPI_COMM_WORLD)
local_sum += Scan_Sum;
local_sum += fact;
MPI_Gather(&local_sum, 1, MPI_INT, ENC, 1,
    MPI_INT, 0, MPI_COMM_WORLD);
for(int i=0; i<n; i++)
{
  printf("%d", ENC[i]);
}
MPI_Finalize();
return 0;
}
```

---

4) Kernel code

```
--kernel
void question4(--global char *S, --global
    char *Res, --global int N) {
  int idx = get_global_id(0);
  for(int i=0; i<N-idx; i++)
  { Res[idx*N-idx+i] = S[N*idx+i]
  }
```

Host codes

a)
```
size_t globalWorkSize[1];
globalWorkSize[0] = N;
```

b)
```
cl_mem buffers, bufferRes, N;
buffers = clCreateBuffer(context, CL_MEM
```

```
READ_ONLY, sizeof(char)*(N+1)*N),
        NULL, &status);

bufferRes = clCreateBuffer(context, CL_MEM_WRITE_
        ONLY, sizeof(char)*(N)*(N+1)/2, NULL,
                &status);
N = clCreateBuffer(context, CL_MEM_READ_ONLY,
    sizeof(int), NULL, &status);
status = clEnqueueWriteBuffer(cmdQueue, buffers,
    CL_FALSE, 0, sizeof(char)*N*(N+1), S, 0, NULL,
                NULL);
status = clEnqueueWriteBuffer(cmdQueue, buffer-
    Res, CL_FALSE, 0, sizeof(char)*(N)*(N+1)/2,
        Res, 0, NULL, NULL);
status = clEnqueueWriteBuffer(cmdQueue, bufferN,
    CL_FALSE, 0, sizeof(int), N, 0, NULL, NULL);
```

4) `cl_program = clCreateProgramWithSource(context, 1, (const char **)&programSrc, NULL, &status);`

```
status = clBuildProgram(program, numDevices,
        devices, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "gk",
                &status);
status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                &buffers);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
                &bufferRes);
status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
                &bufferN);
status = clEnqueueNDRangeKernel(cmdQueue,
                kernel,
                1,
                NULL,
```

```
    &blockAndWorkSize,
    NULL,
    0,
    NULL,
    NULL);
```

---

9)

```
int main()
{
    int N;
    scanf("%d", &N);
    int mat[100][100];
    int result[4];
    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++)
            scanf("%d", &mar[i][j]);
    int *d_m, *d_r;
    cudaMalloc((void **)&d_m, sizeof(int)*N*N);
    cudaMalloc((void**)&dr, sizeof(int)*4);
    cudaMemcpy(d_m, mat, sizeof(int)*N*N,
               cudaMemcpyHostToDevice);
    cudaMemcpy(d_r, result, sizeof(int)*4, cuda-
               MemcpyHostToDevice);
    dim3 matrixGrid(2,2,1);
    DiagonalSum<<<matrixGrid, 1>>>(d_m,d_r,N);
    cudaMemory(result, d_r, sizeof(int)*4, cudaMem-
               cpyDeviceToHost);
    printf("Diagonal sums from threads of different
            blocks:\n");
    for (int i=0; i<4; i++)
        printf("From block %d diagonal sum=%d\n",
               &i, &result[i]);
```

```
    cudaFree(d_m);
    cudaFree(d_r);
    return 0;
}
__global__ void DiagonalSum(int *m, int *r,
                            int N)
{
    int S = (N/2 - 1);
    int Bid = blockIdx.x + blockIdx.y +
              gridDim.x;
    for (int i=0; i<=S; i++)
        r[Bid] += m[(S + i * blockIdx.y) +
                    (Bid + i)];
}
```