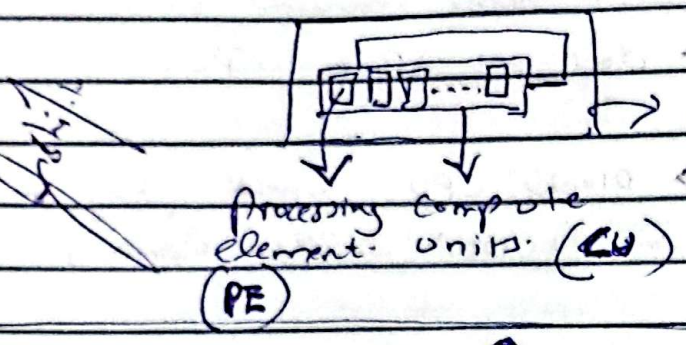
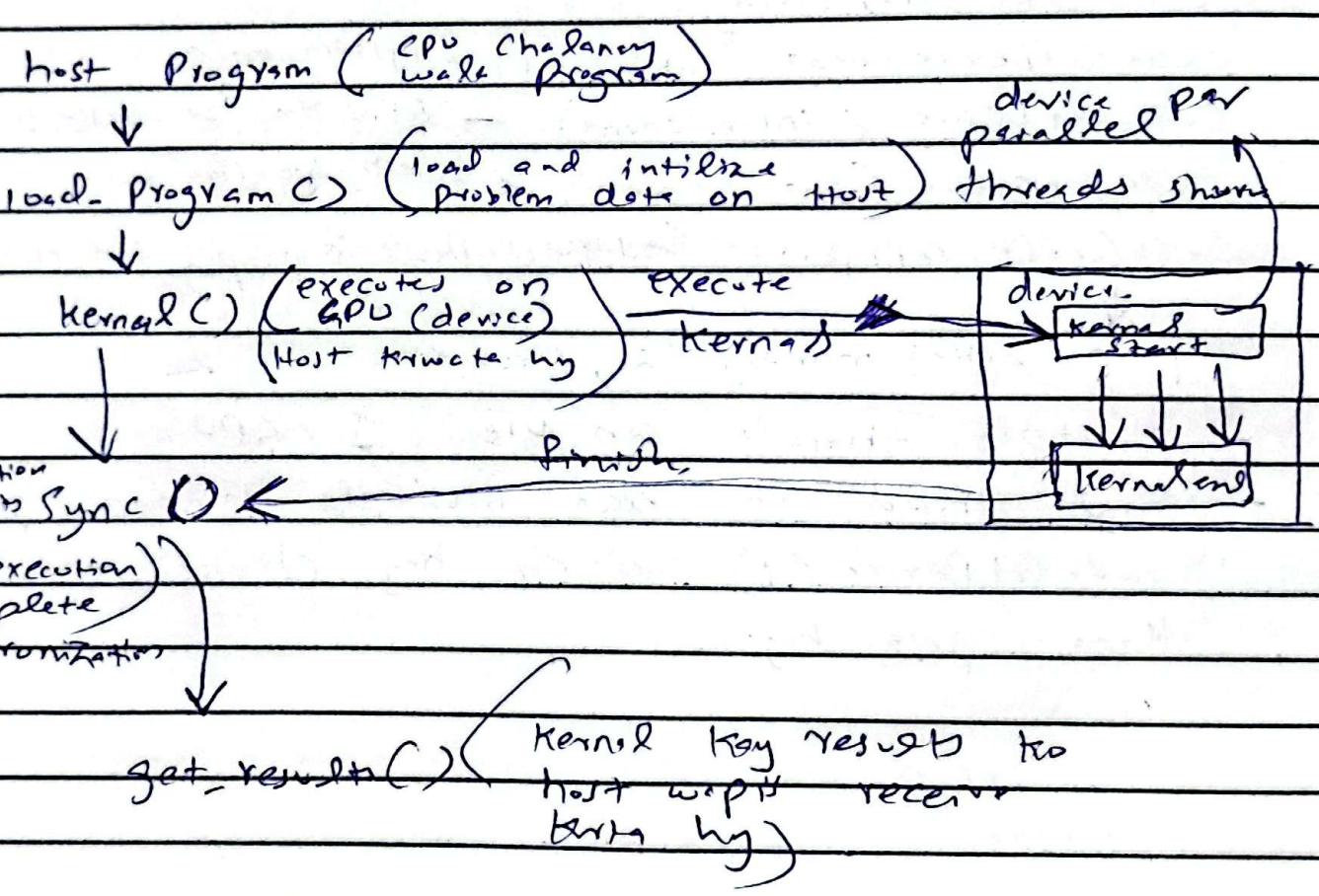
OPENCL (Open Compute language)

- One Host & one or more OpenCL devices
  - Each OpenCL Device is composed of one or more compute units.
  - Each Compute unit is divided into one or more Processing elements.



- Host & Kernal.



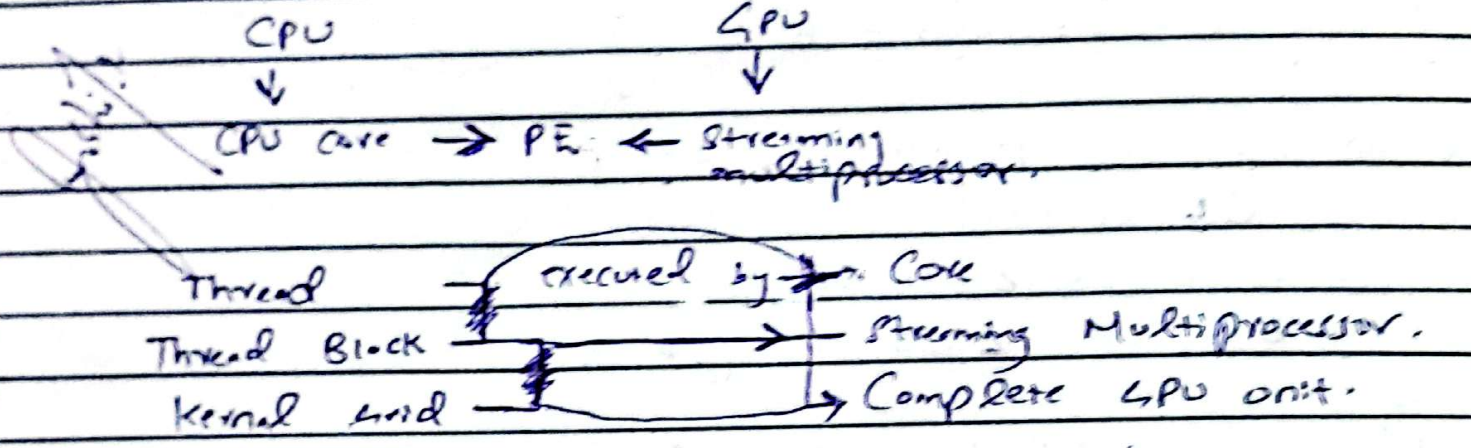
Minimum 2 files

1 host program file (CPU code) .c or .cpp

1 Kernel file (GPU code) .cl

- OpenCL Platform Examples:-

CPU	GPU
- Whole CPU is single openCL device.	- Every GPU is a diff diff openCL device.
- Every CPU core is CU	- GPU may contain streaming multiprocessors ka CU kehte hain.
- Every CU has PE	
- If SIMD lanes used multiple PE's are there.	
- CPU apna host kam krta hai.	- Use CPU or GPU all concurrently.



- Private Memory → Diffent for every work-item (thread)
- Local Memory → • Work-group key andr threads main share hoti hai group key jis threads can access memory. • accessible to all work-groups.
- Global / Constant Memory → • read only
- Host Memory → • CPU memory • Data copy to GPU and then came back.

Memory Management is explicit.

↓

Programmer is responsible for.

moving data from

host → global → local & back.

Kernal → Choti program or function jo accelerator device pr chali jati hy.

- OpenCL main CPU se kernal execute krwatay hain. using clEnqueueNDRangeKernal()

- Vector addition - Host.

Basic host program → • Define Platform = devices + context + queues.
• Create + build Program.
• Setup memory objects
• Define Kernal
• Submit commands.

If you want to point memory to __local or __private.

__kernal void add (__global const float *a, __global b--, c)
{ int id = get_global_id(0);
c[id] = a[id] + b[id]; }.

Kernal code GPU pr chalay ga.

- Set up Host Program in C++ ?

__CL_ENABLE_EXCEPTIONS.
#include <iostream>
#include <CL/cl.hpp>

- 1) Context → • No environment jahan kernels chalay jain.
• jahan memory managment aur syn chronization define hti hy.

- Command queue → • A queue jisko say sari commands device ko bhejay jain.

only for 1 device with a context.

— Kernel (GPU pr chalay gy).

cl:: Context Context (CL_DEVICE_TYPE_DEFAULT);
// // // CPU
// // // GPU
// // // ACCELERATOR.

cl:: Command Queue queue (context);

2) Commands include: ① kernal executions
② Memory object management
③ Synchronization

inorder queues (Jo commands host program jis order mein diyay jatay hain device ko bhejay jatay hain)
out of order queues (device can execute in any order)

cl:: Program program(context, kernal source, true); → written on host.

String or for function jo kernal code file say load hwee.

Mean OpenCL ko bolo ke program ko compile aur link kr de.

• - Buffers (wo memory object jo host (CPU) se declare htay hain ~~cl:: Buffer~~)
- Host has arrays which has actual data.
- Create device-side buffer which holds data in GPU or device memory.

cl:: Buffer d_a (context, h_a.begin(), h_a.end(), true);

host array ka data kaha say kaha tk

Read only buffer Only device can read.

kernal operates on this device buffer.

3) Setup Memory Objects

- Create input vectors on host.

```
std::vector<float> h_a(LENGTH), h_b(LENGTH), h_c(LENGTH);
for (i=0; i<LENGTH; i++)
{   h_a[i] = rand();
    h_b[i] = rand();
}
```

```
cl::Buffer d_a(context, CL_MEM_READ_ONLY, a.size() x sizeof(float));
cl::Buffer d_b(context, CL_MEM_READ_ONLY, b.size() x sizeof(float));
cl::Buffer d_c(context, CL_MEM_READ_ONLY, c.size() x sizeof(float));
```

- Device memory mein 2 tarah ke memory objects hotay hain.

a) Buffer Object.
- Its linear.
- Simple arrays, matrices jo continous line main hota hy wo buffer mein store hotay hain.

b) Image Object.
- 2D or 3D region of memory.
- Image data can only be accessed through read or write functions.

- Device buffer se host memory mein data kesay copy hotay hy ?

```
cl::copy(queue, d_c, h_c.begin(), h_c.end());
```
↓
device buffer.

- Host memory say device buffer.

```
cl::copy(queue, h_c.begin(), h_c.end(), d_c);
```

4) Define Kernal. (Create a Kernal function for Kernal we want to call)

cl::Kernal kernel (program, "vecadd");

↓

__kernel void vecadd(

kernel.setArg(0, bufA);
kernel.setArg(1, bufB);
kernel.setArg(2, bufC);

You can call the kernal as a function in your host code

5) Enqueue Commands.

For kernal launches, specify global & local dimensions.

cl::NDRange global(1024)
cl::NDRange local(64) // If we don't specify local dimension it is assumed as cl::NullRange.

- Kernal Enqueue. (non-blocking)

queue.enqueueNDRangeKernal(kernal, cl::NullRange, cl::NDRange(a.size()), cl::NullRange);

- Red Back Result (blocking)

queue.enqueueReadBuffer(bufC, CL_TRUE, 0, c.size()*sizeof(float), c.data());

Do not use this. Just for practice.

Vector addition by OpenCL

```
#define __CL_ENABLE_EXCEPTIONS
#include <CL/cl.hpp>
#include <iostream>
#include <vector>
using namespace std;
int main() {
```

C++

1) `vector<float> A = {1,2,3,4};`
`vector<float> B = {5,6,7,8};`
`vector<float> C(4);`

2) // Set up opencl context and queue.
`cl::Context context(CL_DEVICE_TYPE_DEFAULT);`
`cl::CommandQueue queue(context);`

3) // Write kernel source as string.

~~`const char* KernelSource = R"CLC( kernel void VecAdd(`~~
~~`__global const float* A, __global const float* B,`~~
~~`__global float* C)`~~

```
const char* KernelSource = R"CLC(
    kernel void VecAdd(
        __global const float* A,
        __global const float* B,
        __global float* C)
    {
        int i = get_global_id(0);
        C[i] = A[i] + B[i];
    }
)CLC";
```

4) // Build the Program

```
cl::Program::Sources sources;
sources.push_back({kernelSource, strlen(kernelSource)});
cl::Program program(context, sources);
program.build();
```

5) // Create device buffers, copy A & B from host → device

```
cl::Buffer bufA(context, CL_MEM_READ_ONLY |
        CL_MEM_COPY_HOST_PTR, sizeof(float)*A.size(),
        A.data());
cl::Buffer bufB(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
        sizeof(float)*B.size(), B.data());

cl::Buffer bufC(context, CL_MEM_WRITE_ONLY, sizeof(float)*
        c.size());
```

6) // Setup Kernal & its arguments

```
cl::Kernel kernel(program, "vecadd");
kernel.setArg(0, bufA);
kernel.setArg(1, bufB);
kernel.setArg(2, bufC);
```

7) // Enqueue kernal for execution

```
queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange
        (A.size()), cl::NullRange);
```

8) // Read back results (blocking read)

```
queue.enqueueReadBuffer(bufC, CL_TRUE, 0, sizeof(float)*c.size(),
        c.data());
```

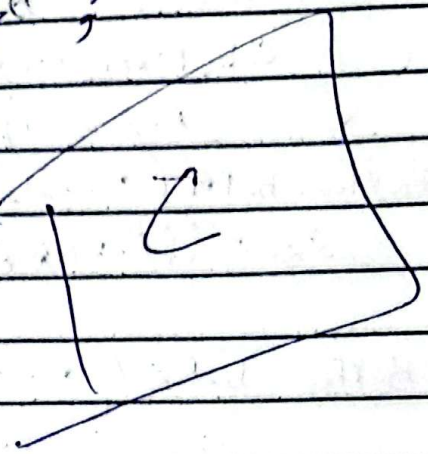
Use this.

```
#include <CL/cl.h>
#include <iostream>
#include <vector>

const char * KernelSource =
"__kernel void vecadd(__global const float *a, b ... , c)
{   int i = get_global_id(0);
    c[i] = A[i] + B[i]; " "}";

int main()
{   float h_a[N] = {1,2,3,4};
    float h_b[N] = {5,6,7,8};
    float h_c[N];

    cl_platform_id   platform;
    cl_device_id     device;
    cl_context       context;
    cl_command_queue queue;
    cl_program       program;
    cl_kernel        kernel;
    cl_mem bufA, bufB, bufC;
    cl_int status;
```



① // Select platform & device.

```
status = clGetPlatformIDs(1, &platform, NULL);
if (status != CL_SUCCESS) { cout << "Error" << endl; }
status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT,
                        1, &device, NULL);
```

clCreateProgramWithSource.

(2) // Context Create Karlay hain.

Context = clCreateContext(NULL, 1, &device, NULL, NULL, &Status);

(3) // Command queue.

queue = clCreateCommandQueue(context, device, 0, &Status);

(4) // Program + Kernel Source load.

Program = clCreateProgramWithSource(context, 1, &KernelSource, NULL, &Status);

Status = clBuildProgram(program, 1, &device, NULL, NULL, NULL);

Kernel = clCreateKernel(Program, "vecadd", &Status);

(5) // Buffers Create Kro.

bufA = clCreateBuffer(context, CL_MEM_READ_ONLY, a.size() * sizeof(float), H_a, &Status)

bufB = clCreateBuffer(context, CL_MEM_READ_ONLY, b.size() * sizeof(float), H_b, Status)

bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY, c.size() * sizeof(float), NULL, &Status)

(6) // Set Kernel Arguments

Status = clSetKernelArg(Kernel, 0, sizeof(cl_mem), &bufA);

Status = clSetKernelArg(Kernel, 1, sizeof(cl_mem), &bufB);

Status = clSetKernelArg(Kernel, 2, sizeof(cl_mem), &bufC);

(7) // Enqueue kernel

```
size_t global_work_size = N;
Status = clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
        &global_work_size, NULL, 0, NULL, NULL);
```

(8) // Results wapis host memory mein read kro.

```
status = clEnqueueReadBuffer(queue, bufC, CL_TRUE,
        0, sizeof(float)*N, h_c, 0, NULL, NULL);
```

(9)

```
clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
return 0;
}
```

→ OpenCL kernal

- Derived from ISO C99. → • no recursions
  • function pointers
- OpenCL has a lot of threads which work parallel.
- These thread groups (work group) mein divided hoty hain. aur har group hardware ke compute unit pr chalta hy.
- Har thread apni unique ID se apna data process krta hy
- Main purpose → loops ki jagah parallel executions karna hy jis se performance ↑

- small and fast memories like private or local should be used more. Global & host memories mein use krna bcz slow.

- NDRange means pora grid jahan kernels execute hoty hain.
- Har grid chote chote parts mein divided hota hy jise work groups kehte hain (local size).
- Work groups mein work items hotay hain.

- Synchronization functions.
  - Barriers → 1 workgroup kay sab work items ek dosray kay sath synchronize hotay hain.
  - Memory fences → memory read/write sahi order mein perform hon.

- Restrictions in OpenCL
  - Kernel kay andr functions kay pointers use nai kr sktay.
  - Arrays whose size vary on run-time, we cannot use.
  - Recursion not supported

- Matrix Multiplication cost depends on FLOPs and memory movement.
- Flops calculations $2 \times n^3 = O(n^3)$
  $3 \times n^2 = O(\frac{2}{n})$
- Whenever memory access hoy max Flops perform krien.