1) • ! Nvidia - smi (System Management Interface)

• cudaMallocManaged() ... cudaFree()

• cudaDeviceSynchronize()

code in CPU → host code

Code in GPU → Device Code

A.cu → CUDA - accelerated Program

HellGPU<<< 1, 1 >>>();

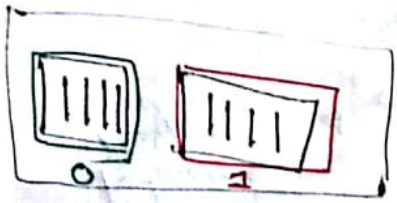
collection of thread is called block

1 block of thread

each block contains 1 thread

Memory allocated in this way is called UM (unified Memory) which is automatically migrated between CPU & GPU

2) .·. nvcc = nvidia cuda Compiler.



→ A collection of blocks associated with a given kernel launch is a grid

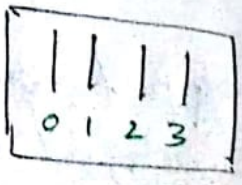
• HellGPU<<< 1, 1, >>>();

→ GPU functions are called kernel.

3) gridDim.x → 2

blockDim.x → 4

blockIdx.x → 0, 1



threadIdx.x → 0, 1, 2, 3

4) Each thread has access to the size of its block via ⟹ blockDim.x

also ⟹ blockIdx.x

threadIdx.x + blockIdx.* blockDim.x

These are CUDA variable

threadIdx.x → range (0 – 1023)

0
1
2
3
4
5
6
7

5) int N = 2 << 20;

size_t size = N * sizeof(int);

int *a;

cudaMallocManaged(&a, size); ← which can be used by both CPU & GPU.

cudaFree(a);

8) cudaError_t arrayError; ← asyncError

arrayError = cudaGetLastError();

= cudaDeviceSynchronize();

if (arrayError != cudaSuccess)

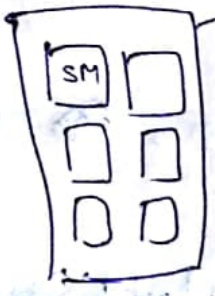
printf("Error: %s\n", cudaGetErrorString(arrayError));

- Managing Accelerated Application Memory with CUDA Unified Memory and nsys

CudaDe	— 1	— 2376366692	831362457
cudaMa	— 3	— 202862158	203838270
cudaFre	— 3	— 18467746	18060721
cudaLa	— 1	— 67525	48899

cudaDeviceSynchronize	5407285599
cudaMallocManaged	205270300
cudaFree	18339185
cudaLaunchKernel	50679

- Streaming Multiprocessor (SM)



→ SM is functional unit of NVIDIA GPU

No of SM = No of Multiprocessor

- int deviceId;
- cudaGetDevice(&deviceId);

cudaDeviceProp prop; prop.

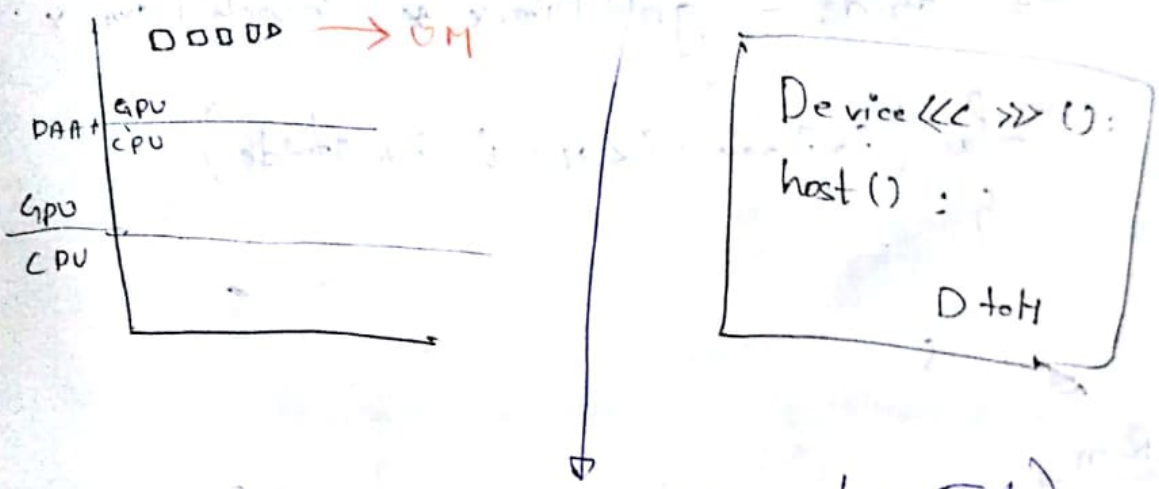
cudaGetDeviceProperties(&prop, deviceId)

- Unified Memory Management → Initially it is resident neither on CPU nor in GPU.

↳ UM

↳ If it is known that memory will be accessed somewhere. it is not resident, asynchronous prefetching can be used.

cudaMemPrefetchAsync (cpu).



prefetch (pointerToSomeUMData, size, deviceId)

prefetch (" , cudaCpuDeviceId);

- cudaDeviceGetAttribute(&numberOfSM, cudaDevAttrMultiProcessorCount, deviceId)

thread = 256

Block = 32 * SM;

- GPU functions are called kernels. (kernels are launched with an execution configuration)
  - A CUDA function that executed on a GPU
- Unit of execution for CUDA kernels. [thread] (add vectors)

- Grid stride loop

↳ Multi GPU prog
↳ cuda Programming

```
int index = threadIdx.x + blockIdx.x *
                          blockDim.x;
int stride = gridDim.x * blockDim.x;
for (i = index; i < N; i = i + stride)
{

}
```

Run

» ! nvcc -o output 1st.cu -run

» ! nsys profile --stats=true ./output

↳ nsight system command Line

- There is a concept of Fallow SM or Wild SM

Matrix Multiply

size_t size = N * N * sizeof(int);
(64 64)

```
for (int row=0; row < N; row++)
{
    for (int col=0; col < N; col++)
    {
        a[row * N + col] = row;
        b[row * N + col] = col+2;
        c[row * N + col] = 0;
    }
}
```

- ```
  int row = blockId.x * blockDim.x + threadId.x;
  int col = blockId.y * blockDim.y + threadId.y;
  if (row < N and col < N)
  {
      for (int k=0; k < N; k++)
      {
          val += a[row*N + k] * b[k*N + col]
      }
      c[row * N + col] = val;
  }
  ```

- `dim3 thread_per_block (16, 16, 1);` // A 16*16 block threads.

`dim3 number_of_blocks((N/thread_per_block.x) + 1, (N/thread_per_block.y) + 1, 1);`

(4+1, 4+1, 1)
  ```

matrixMulGPU <<< no_of_blocks, thread_per_block >>>
(a, b, c_gpu);

---

for (int istep = 0; istep <= nsteps; istep++)
{

Kernal

cudaError_t

Scanp

}

- %f = 93000000.0
- %g = 9.3e+07
- ("%.3g", 3.14159) → 3.14
- ("%.03f", 3.14159) → 3.141
- void foo (dt) → velocity
{
int i = threadId.x + blockId.x * blockDim.x;
int stride = gridDim.x * blockDim.x;
while (i < n)
{
p[i].x += b[i].vx * dt;
y vy
z vz
i += stride;
}

```
void foo2(dt) ⟶ body force
{
    while (i<n){
        float Fx=0.0f, Fy=0.0f, Fz=0.0f;
        for (int j=0; j<n; j++){
            float dx = P[j].x - P[i].x;
                   dy =      y
                   dz =      z
            float distSqr = dx*dx + dy*dy + dz*dz +
                                        SOFTENING;
            float invDist = rsqrtf(distSqr);
            float invDist3 = invDist * invDist * invDist;
            Fx += dx * invDist3;
              y     y
              z     z
        }
        P[i].vx += dt * Fx;
                y       y
                z       z
        i += stride;
    }
}
```

cudaMemcpy

- Allocate memory on the device
- Copy data from host to device.
- Perform some calculations
- Copy Data from device to Host.
- Free Allocated device memory

cudaMemcpy (deviceArray, HostArray, bytes, cudaMemcpyHostTo Device);

cudaMemcpy (HostArray, deviceArray, bytes, cudaMemcpyDevice To Host);