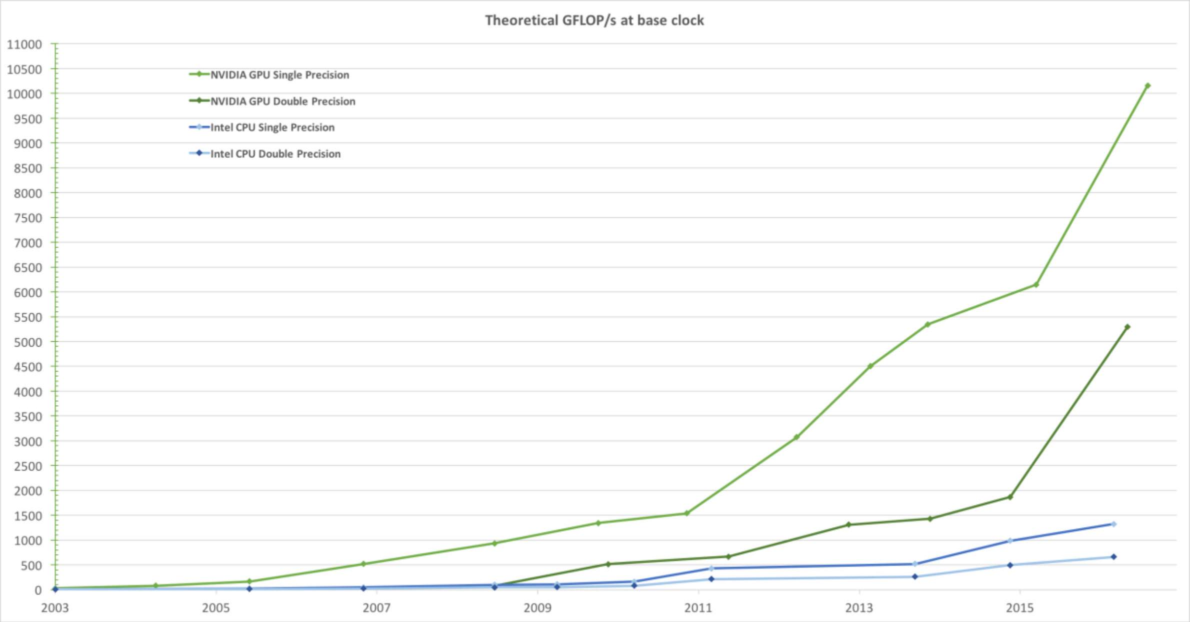
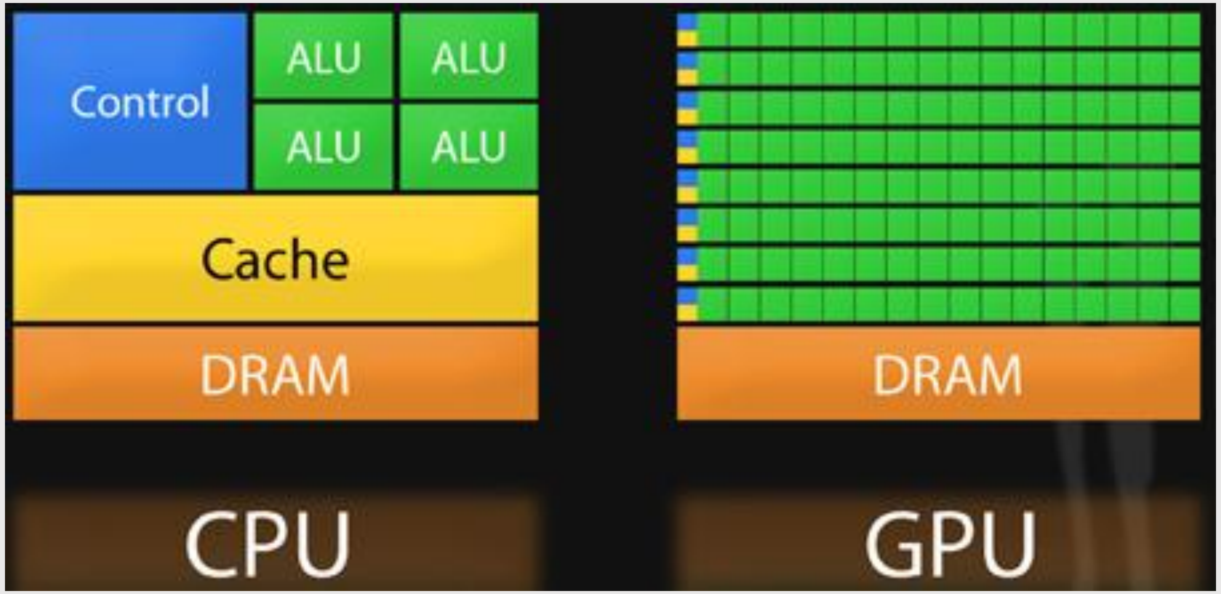
Cuda: NVIDIA GPU programming

Paulius Milmantas

Introduction - Purpose of GPU computing

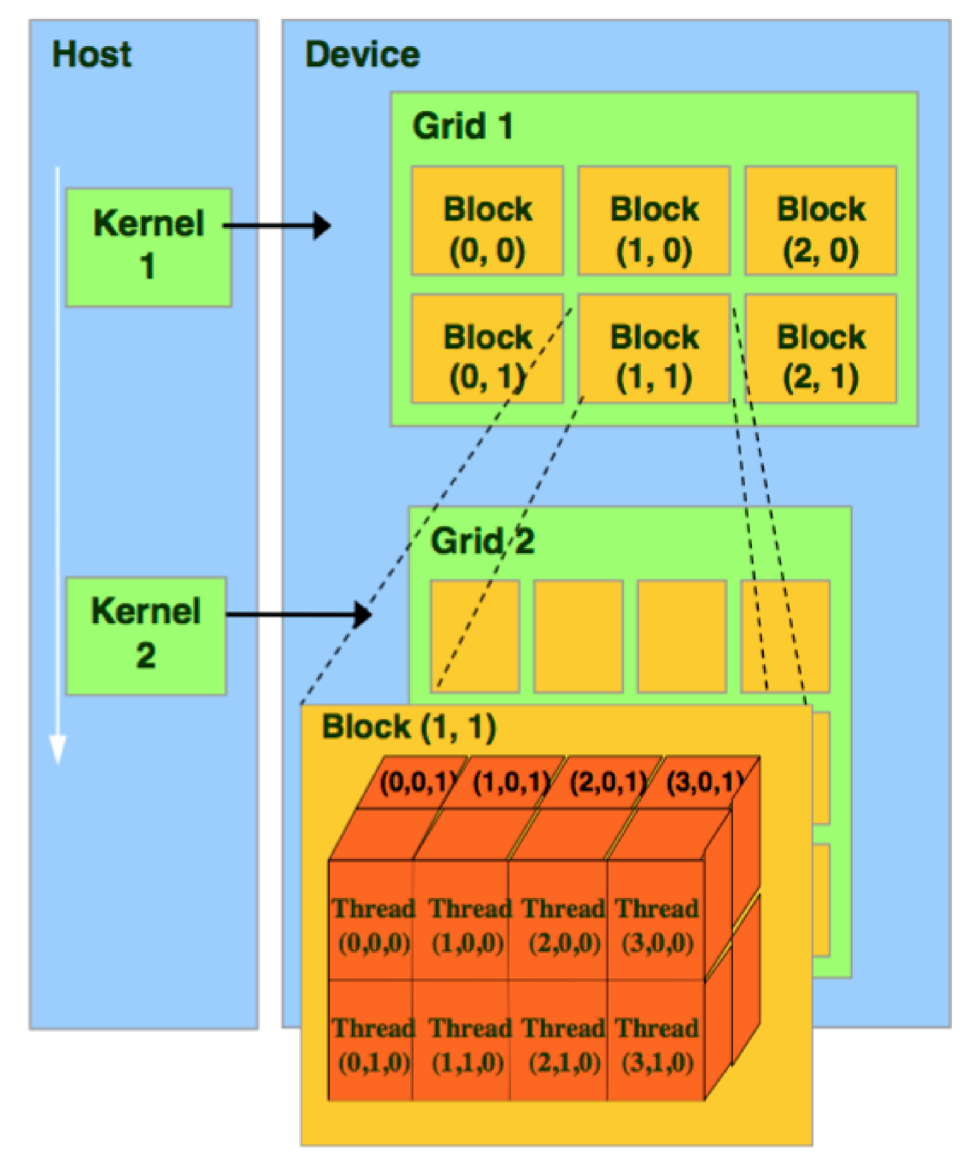
- **CPU** is designed to handle **complex tasks**
- **GPU** is designed to handle repetitive **low-level tasks**.





https://nyu-cds.github.io/python-gpu/01-introduction/

Memory structure

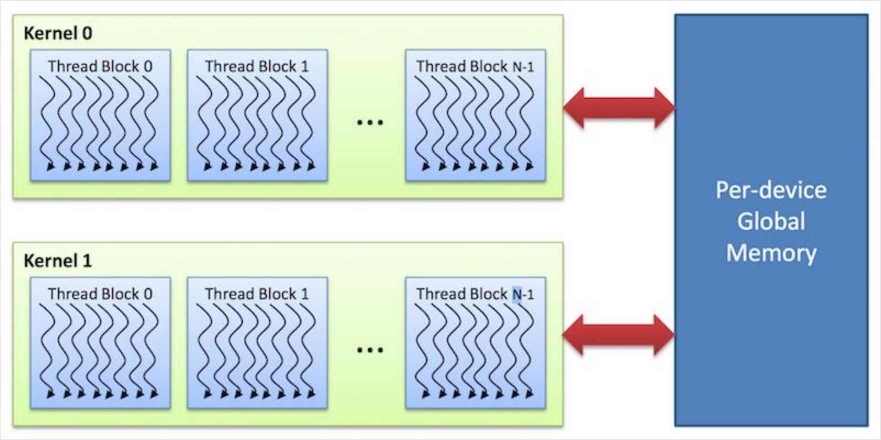


https://nyu-cds.github.io/python-gpu/02-cuda/

Kernel – Shared function executed by GPU
Kernel is executed many times

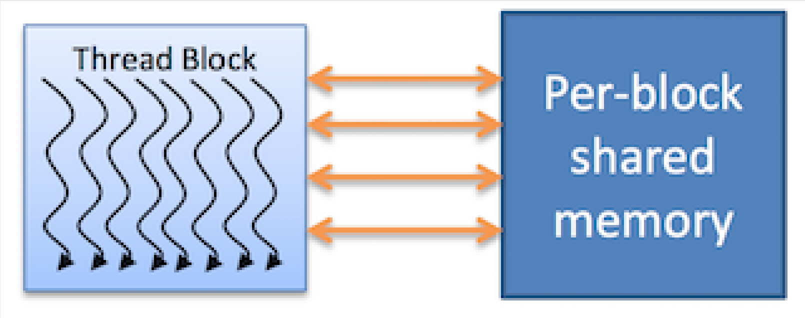
- **Thread** executes **kernel** code
- **Threads** are grouped into **threads blocks**
- **Blocks** are grouped into **grids**

Global memory



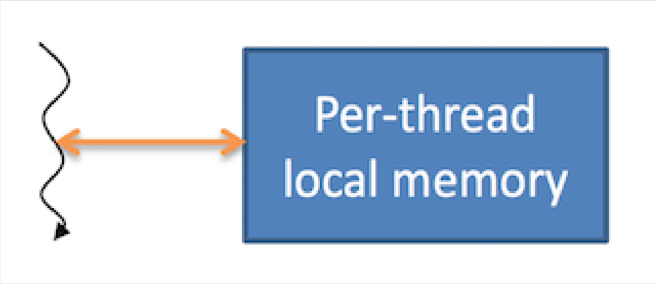
- Memory allocated by the host
- Use by all GPU processes

Shared memory



- Accessible only by the threads with a block
- Memory not shared across different blocks
- Much faster than local/global memory
- Only exists for the lifetime of the block

Local memory



- Only exists for the lifetime of the thread
- Handled automatically by the compiler

https://nyu-cds.github.io/python-gpu/02-cuda/

Typical execution process

1. Copy data from host memory to GPU memory
2. Execute program
3. Copy results from GPU memory to CPU memory.

Example #1 – Kernel

```
__global__ void Add(float *a, float *b, float *c)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

int main()
{
    Add<<<1, N>>>(a, b, c);
}
```

For a 1-dimensional grid:

```
tx = cuda.threadIdx.x
bx = cuda.blockIdx.x
bw = cuda.blockDim.x
i = tx + bx * bw
array[i] = compute(i)
```

For a 2-dimensional grid:

```
tx = cuda.threadIdx.x
ty = cuda.threadIdx.y
bx = cuda.blockIdx.x
by = cuda.blockIdx.y
bw = cuda.blockDim.x
bh = cuda.blockDim.y
x = tx + bx * bw
y = ty + by * bh
array[x, y] = compute(x, y)
```

- Declared using:

Keyword	Executed on	Call on
__device__	Device	Device
__global__	Device	Host
__host__	Host	Host

- Executed with <<< X, Y >>>
  - X – Number of blocks per grid
  - Y – Threads per block

Example #2 – 2D kernel

```
__global__ void Add(float* a[N][N], float* b[N][N], float* c[N][N])
{
    int x = threadIdx.x;
    int y = threadIdx.y;
    c[x][y] = a[x][y] + b[x][y];
}

int main()
{
    dim3 threadsPerBlock(N, N);
    Add<<<1, threadsPerBlock>>>(a, b, c);
}
```

Example #3 – Memory allocation & data transfer

```
int main()
{
    const unsigned int X=1048576; //1 Megabyte
    const unsigned int bytes = X*sizeof(int);
    int *hostArray= (int*)malloc(bytes);
    int *deviceArray;
    cudaMalloc((int**)&deviceArracy,bytes);
    memset(hostArray,0,bytes);
    cudaMemcpy(deviceArray,hostArray,bytes,cudaMemcpyHostToDevice);
    cudaMemcpy(hostArray,deviceArray,bytes,cudaMemcpyDeviceToHost);

    cudaFree(deviceArray);

}
```

Allocate memory on the device
cudaMalloc (void** devPtr, size_t size)

Copy data between host and device
cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)

kind = 0 => Host -> Host
kind = 1 => Host -> Device
kind = 2 => Device -> Host
kind = 3 => Device -> Device
kind = 4 => Direction of the transfer is inferred from the pointer values. Requires unified virtual addressing

Free memory on the device
cudaFree (void* devPtr)

https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html

Example #4 – Reading GPU properties

```
int main()
{
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, device);

    printf("GPU clock rate: %d\n", prop.clockRate);
    printf("Device can concurrently copy memory and execute a kernel: %d\n", prop.deviceOverlap);
}
```

Return information about the compute-device

cudaGetDeviceProperties (cudaDeviceProp* prop, int device)

Conclusions

- Simple low-level operations (e.g., addition, multiplication…) can be more effectively parallelized on GPU than on CPU.
- GPU is not meant for complex operations such us slicing array or complex data comparisons.
- GPU has more complex data management structure than CPU does.