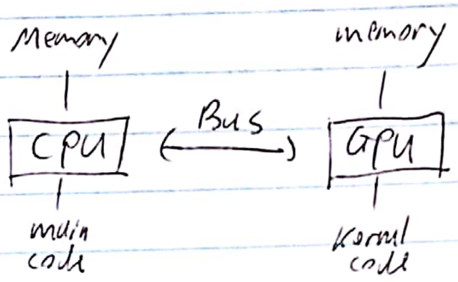
Learn CUDA in an Afternoon

Introduction

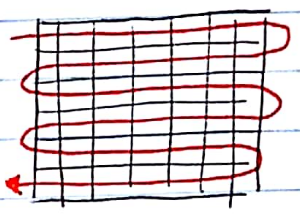
- Why faster?
  many more cores, different kind of memory.
- But, GPUs cannot be used on their own. Not good ~~gor~~ for an OS, for example.
- GPUs accelerate computationally demanding sections of ~~cod~~ code (which we call kernels). Kernels are decomposed to run in parallel on multiple cores.
- Separate memory spaces



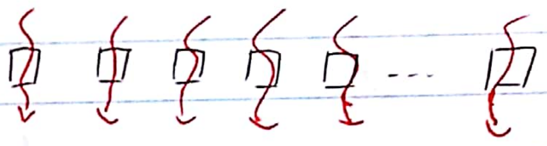
- CUDA is an extension to C/C++ that allows programming GPUs.

Stream computing

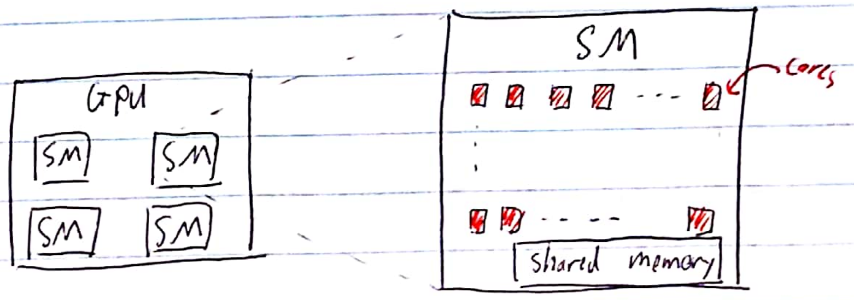
serial or sequential



Parallel



Hardware



SMs - streaming multiprocessors. The number of SMs and cores per SM varries across generation.

- ⟹ We need an abstraction that will perform well across different generations of GPUs.
- This is abstracted as a grid of thread blocks.
- Each block in a grid contains multiple threads, mapping onto the cores in an SM.

So, the mapping is

$$\begin{tikzcd} \text{block} \arrow[r] & \text{SM} \\ \text{thread} \arrow[r] & \text{core in an SM} \end{tikzcd}$$

- Key thing is that we don't need to know the details of the hardware.
- Instead, we oversubscribe, and the system will perform the scheduling automatically. Use more blocks than SMs, and more threads than cores. This way everything is kept as busy as possible, giving the best possible performance.

CUDA

- dim3 type:

```
dim3 my_xyz_values (Xvalue, yvalue, zvalue);
```

- Access with .x (as usual).

Hotel Example

- Serial solution

```
for( i=0; i<N; i++) {
    result [i] = 2*i;
}
```

- parallel ①

```
__global__ void mykernel (int *result)
{
    int i = threadIdx.x; ②
    result [i] = 2*i;
}
```

① specify that this is a kernel

② internal var unique to each thread in a block. It's a dim3 type but since our problem is 1D, we don't use .y and .z.

- Launching the kernel from the CPU:

```
dim3 blocksPerGrid(1, 1, 1);     // 1 block
dim3 threadsPerBlock(N, 1, 1);   // N threads

myKernel <<<< blocksPerGrid, threadsPerBlock >>>> (result);
```

- The above example won't be fast, since ~~it~~ it only uses 1 block (which maps to 1 SM). To use multiple blocks:

```
__global__ void myKernel(int *result)
{
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        result[i] = 2*x;
}
...
dim3 blocksPerGrid(N/256, 1, 1); // assuming N%256 == 0
dim3 threadsPerBlock(256, 1, 1);

myKernel <<<< blocksPerGrid, threadsPerBlock >>>> (result);
...
```

- We have chosen to use 256 threads per block, which is typically a good number (why?)

- Vector addition example:

```
__global__ addVectors(float *a, float *b, float *c) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        c[i] = a[i] + b[i];
}
...
dim3 bpg(N/256, 1, 1);
dim3 tpb(256, 1, 1);
addVectors <<<< bpg, tpb >>>> (a, b, c);
```

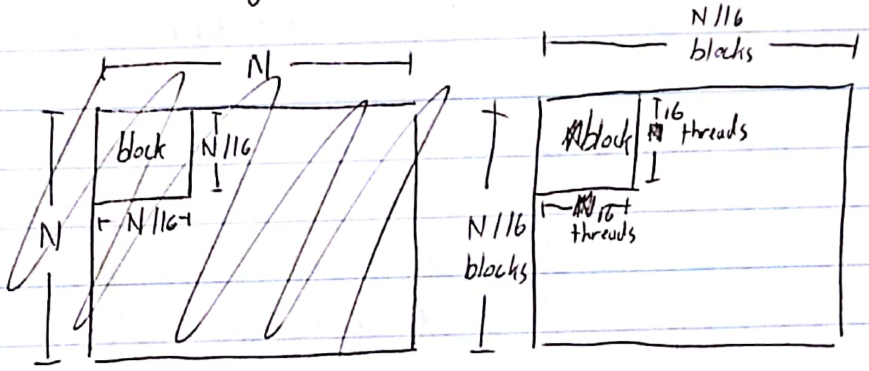
2D Example

```
__global__ void matrixAdd( float a[N][N], float b[N][N],
                           float c[N][N] ) {

        int J = blockIdx.x * blockDim.x + threadIdx.x;
        int i = blockIdx.y * blockDim.y + threadIdx.y;

        c[i][J] = a[i][J] + b[i][J];
}

int main() {
    dim3 bpg (N/16, N/16, 1)
    dim3 tpb (16, 16, 1);
    matrixAdd <<<<bpg, tpb >>>> (a, b, c);
}
```



Memory Management

- stuff in the kernel has to point at GPU memory.
- alloc and free memory on the GPU with:

```
float *a;
cudaMalloc(&a, N* sizeof(float));
...
cudaFree(a);
```

- Copying memory

```
                  destination    source
cudaMemcpy(array_device, array_host, N*sizeof(float),
           cudaMemcpyHostToDevice);
cudaMemcpy(array_device, array_host, N*sizeof(float),
           cudaMemcpyDeviceToHost);
```

* transfers are relatively slow!

(device - GPU
host - CPU)

Sync Between Host and Device

- kernel calls are non-blocking (i.e., host program continues after it calls the kernel).
- Use cudaThreadSynchronize() to wait for kernel to finish
- standard cudaMemcpy calls are blocking (non-blocking variants exist)
- It's good practice to just put the cudaThreadSynchronize().

Sync Between CUDA Threads

- To sync between threads in a block, use syncthreads()
- Example: Communicate a value between threads. (Assume x is local and array is shared.)

```
if (threadIdx.x == 0) array[0] = x;
syncthreads();
if (threadIdx.x == 1) x = array[0];
```

- It is not possible to communicate between different blocks. Must instead exit kernel and start a new one.

GPU Performance Inhibitors

- Copying data to/from device
- Device under-utilisation / GPU memory latency
- GPU memory bandwidth
- Code branching

Host - Device Data Copy

- Copying from host to device is expensive. We want to minimize these copies.
- Keep data resident on device. May require importing routines to device, even if they are not computationally expensive.
- Might be quicker to calculate something from scratch on device instead of copying.

Data Copy optimization Example

```
Loop over timesteps
    inexpensive_routine_on_host (data_on_host)
    copy data from host to device
    expensive_routine_on_device (data_on_device)
    copy data from device to host
end
```

- ~~Ple~~ slow!
  port inexpensive routine to device and move data copies outside of loop.

```
Copy data from host to device
Loop over timesteps
    inexpensive_routine_on_device (data_on_device)
    expensive_routine_on_device (data_on_device)
end
copy data from device to host
```

Exposing parallelism

- GPU performance relies on the usage of many threads
- If a lot of code remains serial, effectiveness of GPU will be limited (Amdahl's law)

Occupancy and Memory Latency Hiding

- Decompose loops to threads
- For best performance, # threads >> # cores
- Accesses to GPU memory have several hundred cycles latency. When a thread stalls waiting for data, if another thread can switch in this latency can be hidden.
- NVIDIA GPUs have very fast thread switching.

Example

original code

- Loop over i from 1 to 512
  - Loop over J from 1 to 512
    - independent iteration

1D decomposition

- Calc i from thread/block ID
  - Loop over J from 1 to 512
    - independent iteration

X 512 threads

2D decomposition

- Calc i & J from thread / block ID
  - independent iteration

✓ 262,144 threads

Memory coalescing

- GPUs have high peak memory bandwith. But this is achieved only when data is accessed for multiple threads in a single transaction : memory coalescing.
- To achieve this, ensure that consecutive threads access consecutive memory location !
- otherwise, memory accesses are serialised, significantly degrading performance.

Example

- Consecutive threads are those with consecutive threadIdx. X values.
- Do consecutive threads access consecutive memory locations?

✓

```
index = blockIdx.X * blockDim.X + threadIdx.X;
output[index] = 2 * input[index];
```

Example

- In C, outermost index runs fastest: J here

X

not coalesced. consecutive threadIdx.x corresponds to consecutive i values!

```
i = blockIdx.x * blockDim.x + threadIdx.x;
for ( J=0 ; J<N ; J++ )
        output [i][J] = 2 * input [i][J] ;
```

```
J = blockIdx.x * blockDim.x + threadIdx.x;
for (i=0 ; i<N ; i++ )
        output [i][J] = 2 * input [i][J] ;
```