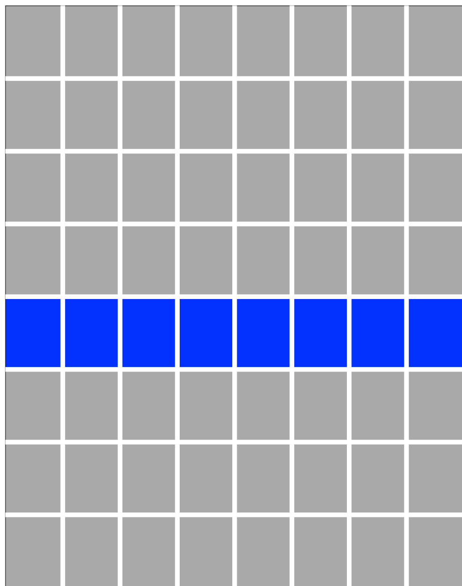
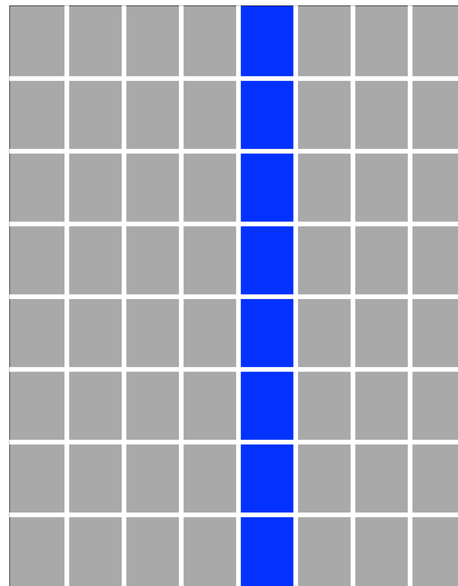
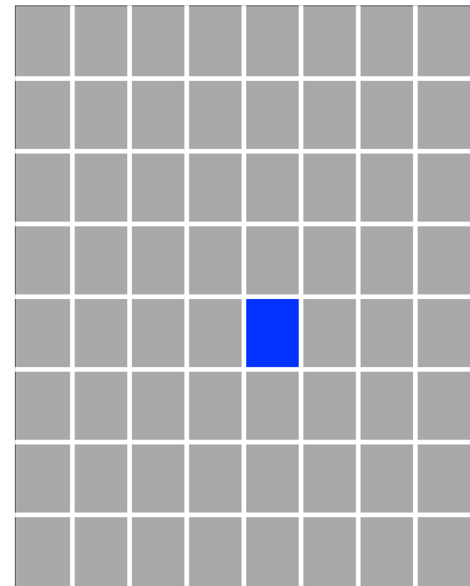
Lecture 8

Matrix multiplication

$$A \times B = C$$


x

=


Naive matrix multiplication - CPU

```
void simpleMultiply_cpu(float *a, float *b, float *c ,int N)
{
float sum;
for (int row=0; row < N ; row++ ){
  for (int col=0; col < N ; col++ ){
    sum = 0.0f;
    for (int i=0; i < N; i++) {
       sum+= a[row*N+i]*b[i*N+col];
    }
    c[row*N+col]=sum;
  }
}
}
```

Naive matrix multiplication - CUDA

```
__global__ void simpleMultiply_gpu(int N)
{
 int row=blockIdx.y*blockDim.y + threadIdx.y;
 int col=blockIdx.x*blockDim.x + threadIdx.x;
 float sum = 0.0f;
 for (int i=0; i < N; i++) {
     sum+= a_dev[row][i]*b_dev[i][col];
 }
 c_dev[row][col]=sum;
}
```

Compare performance of naive vs CUBLAS

- try 1024 x 1024 matrix

  naive (matmulti_CUDA_static.cu) - 4.94 ms

  CUBLAS (matmulti_CUBLAS.cu) - 0.47 ms

  This is a factor of 10 difference. Generally performance of naive becomes worse as matrix grows larger.

  Lesson: always use libraries if possible!

Why is naive code so bad?

- Cache misses

  Consider 1024 x 1024 matrix multiplication with CUDA. Assume blocks 32 x 32. Each block will use 32x32x32 elements of input matrices A and B.

  Float type has 4 bytes, so for each matrix need 32x32x32*4=131072 bytes = 128 KB

  L1 cache on P100 is 24 KB, not enough to fit everything in at once

Solution: work with 32x32 tiles

- Re-order the work

  Load 32x32 tiles of A and B one after another, work with one at a time.

  Copy each 32x32 tile of A and B to shared memory before working with it.

  32x32x4= 4 KB, will fit in 64 KB of shared memory easily

  Shared memory is as fast as L1 cache

Exercise - improve naive matrix multiply

- `cd matrix_multiply`
- Use matmulti_CUDA_static.cu as starting point, write a new kernel which works with 32x32 tiles of A and B.
- Copy each pair of tiles to shared memory first
- More than one synchronization might be needed
- Think carefully whether bank conflicts are something to worry about in this case

Helpful tools

- CUDA includes Nsight, an Integrated Development Environment (IDE) for Linux/Mac based on Eclipse. IDE incorporates CUDA-aware editor, profiler and debugger in one close-integrated package. Try it out!
- There is a Visual Studio edition of Nsight for Windows
- On SHARCNET the DDT visual debugger has powerful GPU debugging capability

Concurrent execution and streams

- Concurrency (parallel execution) between GPU and CPU is either a default, or easily enabled behaviour
  - Kernel launches are always asynchronous with regards to the host code; one has to use explicit device-host synchronization any time a kernel needs to be synchronized with the host:
    - CudaDeviceSynchronize ()
  - The default behaviour of GPU<->CPU memory copy operations is asynchronous for small transfers (<64kB; only host->device), and synchronous otherwise. But one can enforce any memory copying to be asynchronous by adding Async suffix, e.g.:
    - cudaMemcpyAsync ()
    - cudaMemcpyToSymbolAsync ()
  - For debugging purposes, one can enforce everything to be synchronous by setting the CUDA_LAUNCH_BLOCKING environment variable to 1.

Concurrent execution and streams (cont)

- Concurrency between different device operations (kernels and/or memory copying) is a completely different story
  - On a hardware level, modern GPUs are capable of running multiple kernels and memory transfers both to and from the device concurrently
  - By default, everything on device is done sequentially (no concurrency)
  - To make use of the device concurrency features, one has to explicitly use multiple streams in the CUDA code
  - But even with multiple streams, there are some limitations to concurrency on GPU

Concurrent execution and streams (cont)

- A stream is a sequence of commands (possibly issued by different host threads) that execute in order
- If stream ID is omitted, it is assumed to be "NULL" (default) stream. For non-default streams, the IDs have to be used explicitly.
- For concurrent memory copying on GPU, one has to both add the Async suffix and specify the (non-zero) stream ID.

```
mykernel <<<Nblocks, Nthreads, 0, ID>>> ();

cudaMemcpyAsync (d_A, h_A, size, cudaMemcpyHostToDevice, ID);
```

Concurrent execution and streams (cont)

- Before using, streams have to be created. At the end, they have to be destroyed

```
// Host code
cudaStream_t  ID[2];

// Creating streams:
for (int i = 0; i < 2; ++i)
    cudaStreamCreate (&ID[i]);

// These two commands will run concurrently on GPU:
mykernel <<<Nblocks, Nthreads, 0, ID[0]>>> ();
cudaMemcpyAsync (d_A, h_A, size, cudaMemcpyHostToDevice, ID[1]);

// Destroying streams:
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy (ID[i]);
```

Concurrent execution and streams (cont)

- Limitations:
  - For memory copying operations to run concurrently with any other device operation (kernel or another memory copying operation), the host memory has to be *page-locked* (or *pinned*; allocated with cudaMallocHost instead of malloc; static variables are pinned using cudaHostRegister)
  - Up to 128 kernels can run concurrently (P100)
  - Concurrency on GPU is not guaranteed (e.g., if kernels use too much local resources, they will not run concurrently)

Concurrent execution and streams (cont)

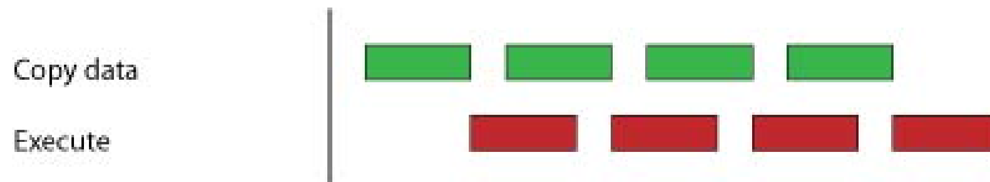
- Other stream-related commands
  - cudaDeviceSynchronize() : global synchronization (across all the streams and the host);
  - cudaStreamSynchronize (ID) : synchronize stream ID with the host;
  - cudaStreamQuery (ID) : tests if the stream ID has finished running.

Optimizing memory copying between GPU and CPU

- Staged concurrent copy and execute

One stream scenario:

You need two streams for this:



Hands on exercise

- Staged: using streams to stage copying and computing