Introduction to Assignment 3

- GPU Matrix Multiplication
- GPU Poisson Problem

+ Compare with your best CPU versions

- Background:
  - Re-read the two previous assignments
- Profiler: `nvvp`
  - Brief demonstration at the end of this introduction

Week 1: BLAS

GPU Matrix multiplication

GPU Matrix multiplication

- The `matmult_f.nvcc` driver is provided

```
matmult_f.nvcc type m n k [bs]

where m, n, k are the parameters defining the matrix sizes, bs is the
optional blocksize for the block version, and type can be one of:

nat     - the native/naive version
lib     - the library version (note that this now calls a multithreaded
library)
gpu1    - the first gpu version
gpu2    - the second gpu version
gpu3    - the third gpu version
gpu4    - the fourth gpu version
gpu5    - the fifth gpu version
gpu6    - the sixth gpu version
gpulib  - the CUBLAS library version

as well as blk, mnk, nmk, ... (the permutations).
```

- See README for more (also week 1 README)

GPU Matrix multiplication

- Reference version: BLAS (e.g., cblas)

```
void DGEMM(char *transa, char *transb,
           int *m, int *n, int *k,
           double *alpha,
           double *A, int *lda,
           double *B, int *ldb,
           double *beta,
           double *C, int *ldc);
```

- You need to use `extern "C" {}` when including header files for C libraries in `.cu` files

```
extern "C" { #include <cblas.h> }
```

GPU Matrix multiplication

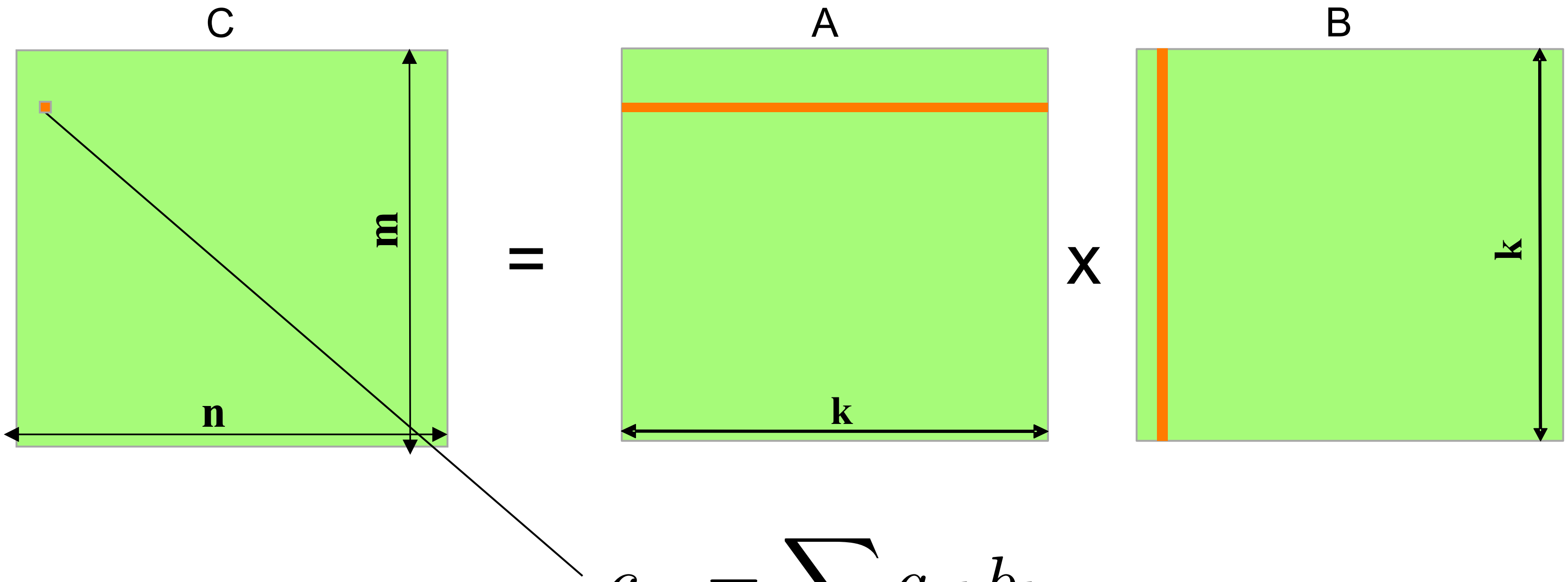
- You also need to use `extern "C" {}` for the functions in your shared library (driver is by `gcc`)

```
extern "C" {
   matmult_lib(...)
   {
       ...
   }
...etc.
}
```

- C code in separate `.c` files may be compiled by `gcc` or `nvcc` but always linked in by `nvcc`

GPU Matrix multiplication

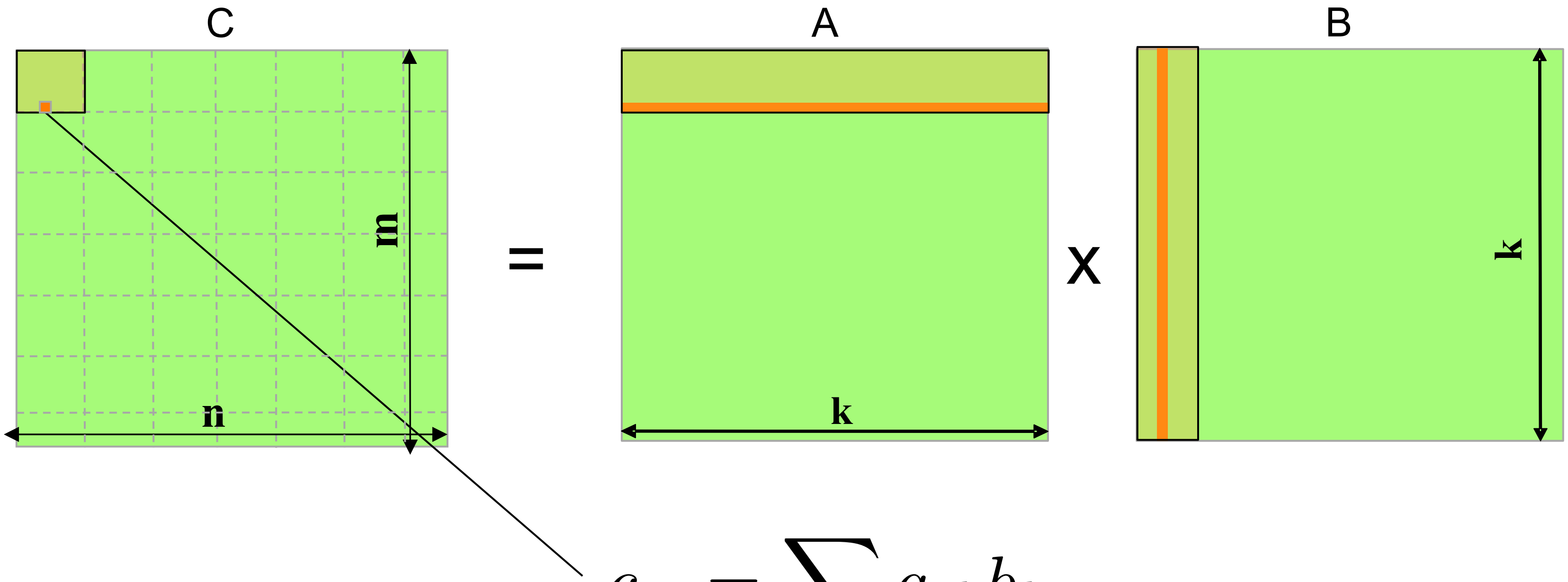
- Sequential version v1: One thread does it all
  - Launch configuration `<<<1,1,>>>`



$$c_{ij} = \sum_{k} a_{ik} b_{kj}$$

GPU Matrix multiplication

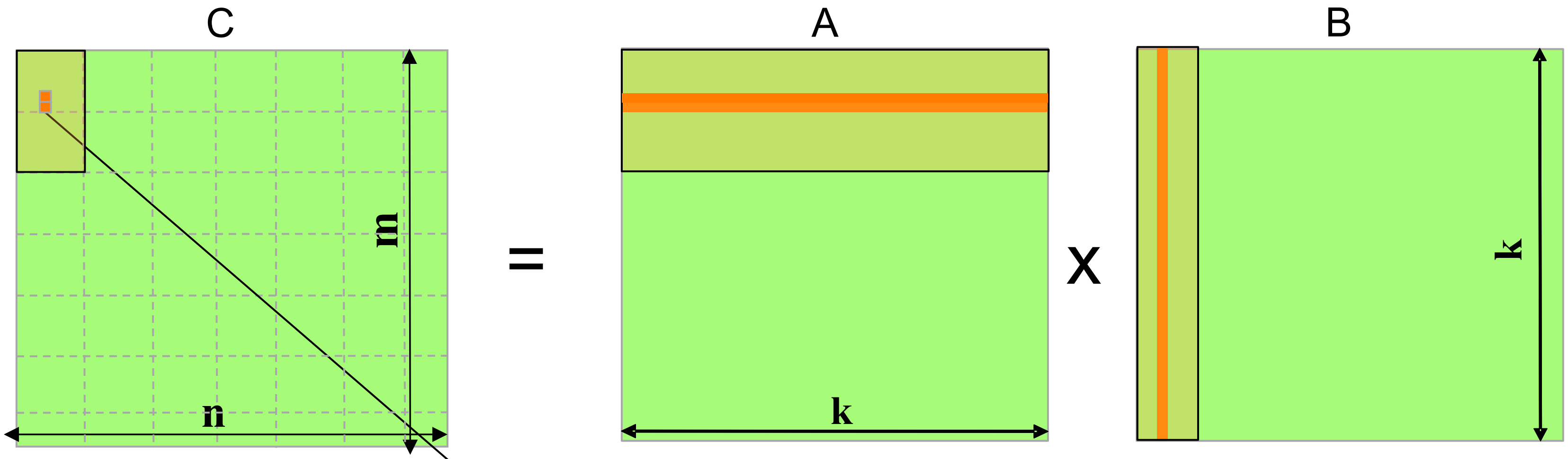
- Naive version v2: One thread per element in C
  - 2D Grid, 2D block (for example 16 x 16 threads)



$$c_{ij} = \sum_k a_{ik} b_{kj}$$

GPU Matrix multiplication

- Register blocking v3: Each thread does 2 elements
  - ❑ Which position of second element is optimal?
  - ❑ 2D grid is 1/2 in y dimension, 2D block is the same



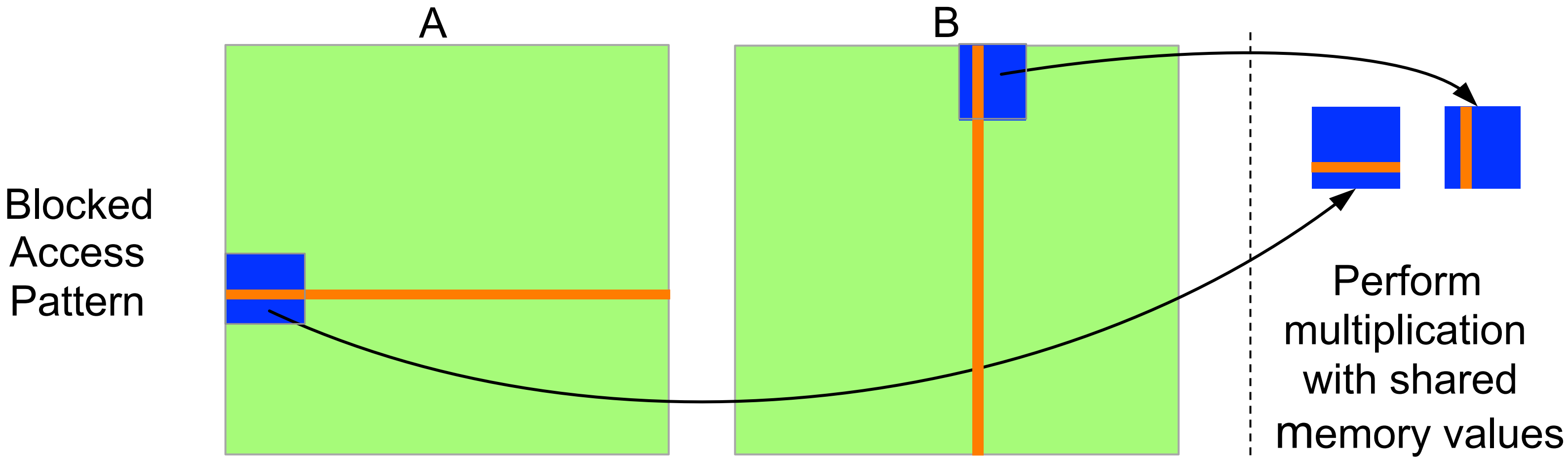
$$c_{ij} = \sum_k a_{ik} b_{kj}$$

GPU Matrix multiplication

- Extended register blocking v4:
  - Each thread does more than two elements
    - How many are optimal? Which positions are optimal?
  - Easiest way to allocate more elements in registers
    - `double C_reg[4] = {0.0, 0.0, 0.0, 0.0};`
    - `double C_reg[2][2] = {0.0, 0.0, 0.0, 0.0};`
  - Write several trial cases and choose the best
  - Or write a generic kernel that can run all choices, and select the best parameters

GPU Matrix multiplication

- Shared memory v5: Read in blocks of A of B
  - E.g. use `dim3(16,16)` blocks and split the ’k’ loop in pieces of 16
  - Allocate shared memory: `A_s[16][16]` and `B_s[16][16]`.



- Go to the Nvidia online documentation: [http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html - shared-memory](http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory)
  - Copy-paste code from here and modify to fit the matmult driver

GPU Matrix multiplication

- Compare with cuBLAS version

```
cublasStatus_t cublasDgemm(cublasHandle_t handle,
                           cublasOperation_t transa,
                           cublasOperation_t transb,
                           int m, int n, int k,
                           const double *alpha,
                           const double *A, int lda,
                           const double *B, int ldb,
                           const double *beta,
                           double *C, int ldc)
```

  - Note: cuBLAS library uses column-major storage
  - New and legacy cuBLAS APIs (see here)
  - `#include <cublas_v2.h>`

GPU Matrix multiplication

- The `-G` flag sets debug lines into your code
  - This reduces the performance drastically
  - Remove it for performance tuning!!!
- For large matrices please use
  - `MFLOPS_MAX_IT=1 ./matmult_f.nvcc ...`
- For benchmarks please use
  - `MATMULT_COMPARE=0`
- Benchmark runs should be submitted as batch jobs (see last slide)

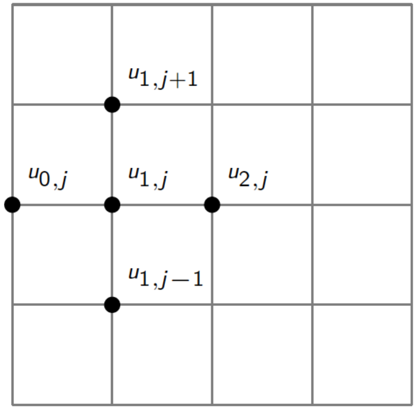
GPU Poisson Problem

GPU Poisson problem

- Reference version: Your best OpenMP version from assignment 2.
  - Also use your code to allocate and initialize the necessary matrices for the square room problem
  - Note that if you used the `cc` sun compiler before there might be slight differences to the `gcc` compiler

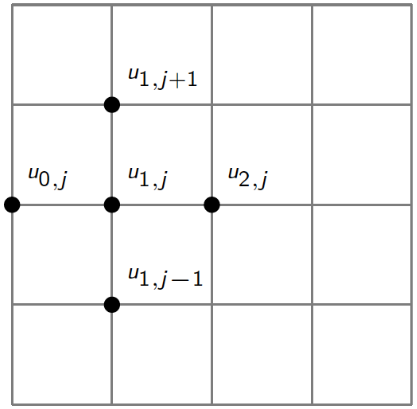
GPU Poisson problem

- Sequential version: One thread does it all
  - Launch configuration `<<<1,1,>>>`
  - Hint: Do only one iteration per kernel launch!
  - Hint: Swap pointers for `u` and `u_old` on the CPU



GPU Poisson problem

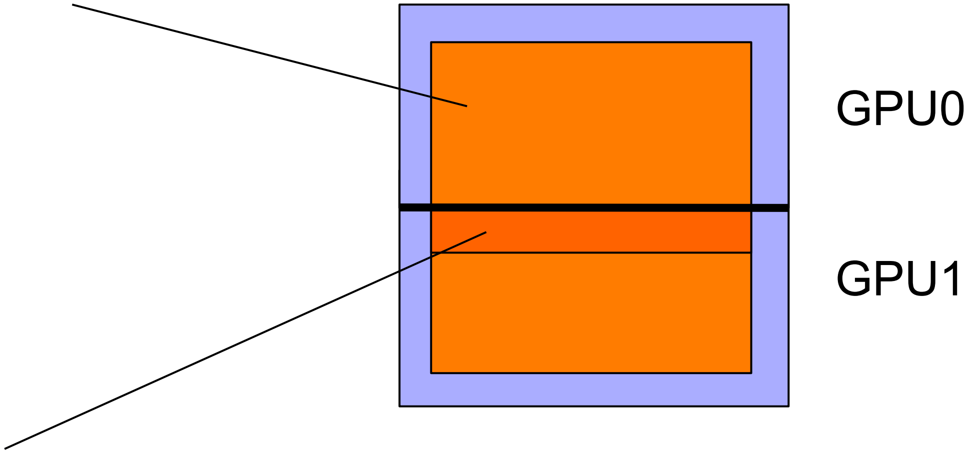
- Naive version: One thread per grid-point
  - 2D grid, 2D block
  - Global memory usage only – rely on caches to help



GPU Poisson problem

- Multi-GPU version
  - ❑ Split task into two – top and bottom
  - ❑ Interior points can be updated from global memory
  - ❑ Border points must read ”peer values” from other GPU

Read from global memory



Available from other GPU

Reports / Analysis

- Assess your performance (Gflops / Bandwidth)
- Speed-up calculations (fair)
- Tuning considerations
- Profiler analysis
- Relevant comments and observations

Submitting GPU batch jobs

- Benchmark runs should always be submitted
  - Select queue and ask for GPUs using these options
    - `-q gpuv100`
    - `-gpu "num=1:mode=exclusive_process:mps=yes”`
  - Put these into your job scripts
- Maximum wall-clock time on jobs 1 hour!
- For jobs using two GPUs use `num=2`
- For CPU-only jobs please do not request GPUs