Assignment 3

Hans Henrik Brandenborg Sørensen
DTU Computing Center
DTU Compute
<hhbs@dtu.dk>

Introduction to Assignment 3

- GPU Matrix Multiplication
- GPU Poisson Problem

+ Compare with your best CPU versions

Introduction to Assignment 3

- GPU Matrix Multiplication
- GPU Poisson Problem

+ Compare with your best CPU versions

- Programming in CUDA – stepwise improvement:
  1. Sequential version (global memory only, 1 thread).
  2. Naive version (global memory only, simplest design).
  3. If concurrency bound ➔ introduce more threads.
  4. Register version (blocking for registers).
  5. Shared memory version / blocking for L1 cache.
  6. If compute bound ➔ optimize instructions/unroll etc.

Debugging CUDA code

- Recommendations
  - Simple strategies for debugging code
    - Systematically use

```
cudaCheckErrors( ... );
```

    - Remember to wait for kernels to finish

```
cudaCheckErrors( cudaDeviceSynchronize() );
```

    - Compare output with debugged CPU code:

Step 1: Copy array from device to host   Step 2: Print using CPU

    - Last resort: `printf(“tid=%d, val=%f\n”,tid,val);`

CUDA code debugging can be very difficult – nondeterministic

Debugging CUDA code

- `cuda-memcheck` – Detects/tracks memory errors
  - ❑ Out of bounds accesses
  - ❑ Misaligned accesses
  - ❑ Compile with debug `-g` flag gives more information
- VecAdd example

```
// don't process values after N
~~if (tid < N)~~
   C[tid] = A[tid] + B[tid];
```

```
hhs@gpu-lab-01:~/Exercises/Lab02_VecAdd$ cuda-memcheck ./VecAdd
========= CUDA-MEMCHECK
Cuda error in file 'VecAdd.cu' in line 48 : unspecified launch failure
========= Invalid read of size 4
=========     at 0x00000028 in VecAdd_kernel
=========     by thread (16,0,0) in block (39,0)
========= Address 0x00109c40 is out of bounds
=========
========= ERROR SUMMARY: 1 error
```

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

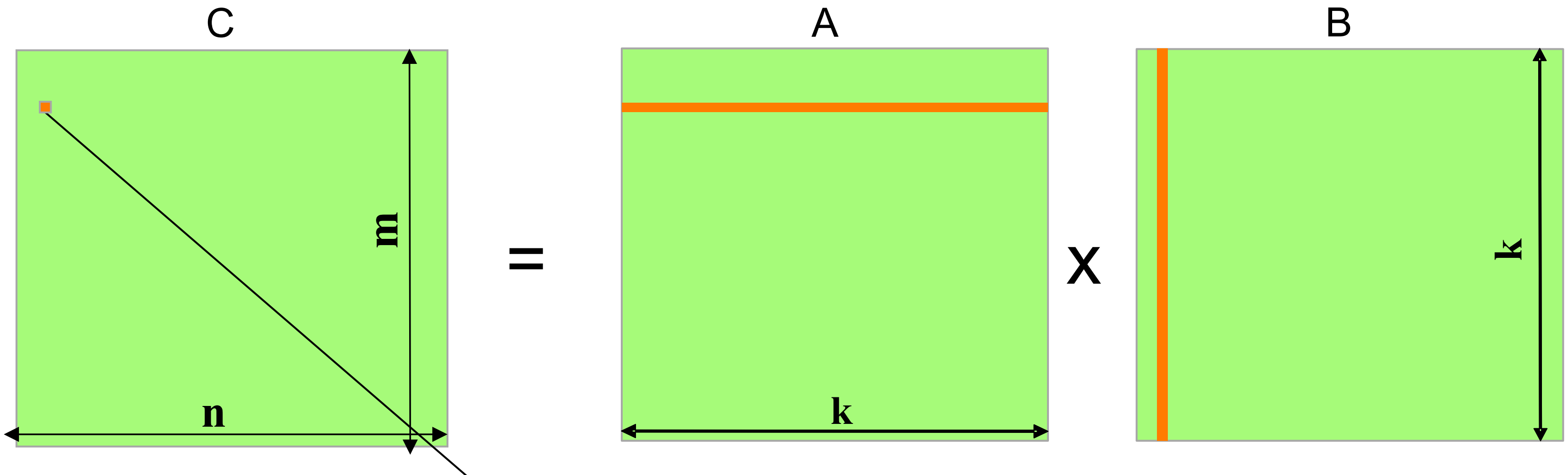
- You also need to use `extern "C" {}` for the functions in your shared library (driver is by `gcc`)

```
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

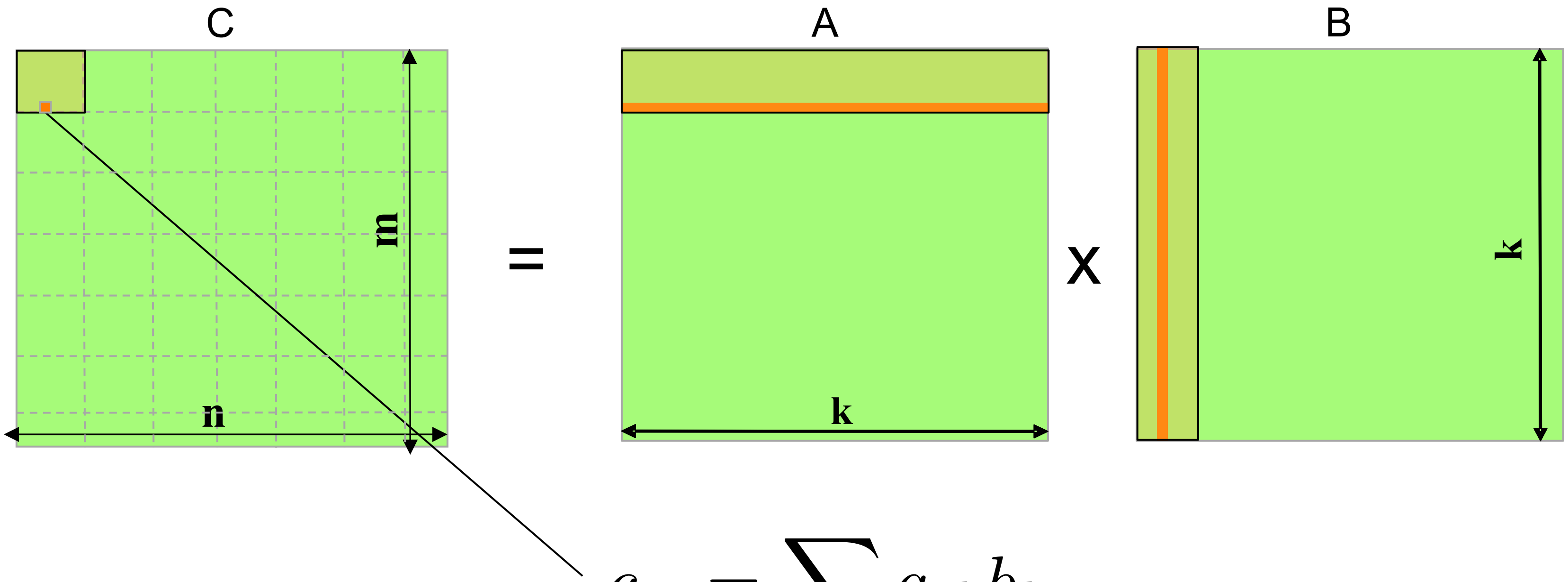
- Sequential version: One thread does it all
  - Launch configuration `<<<1,1,>>>`



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

GPU Matrix multiplication

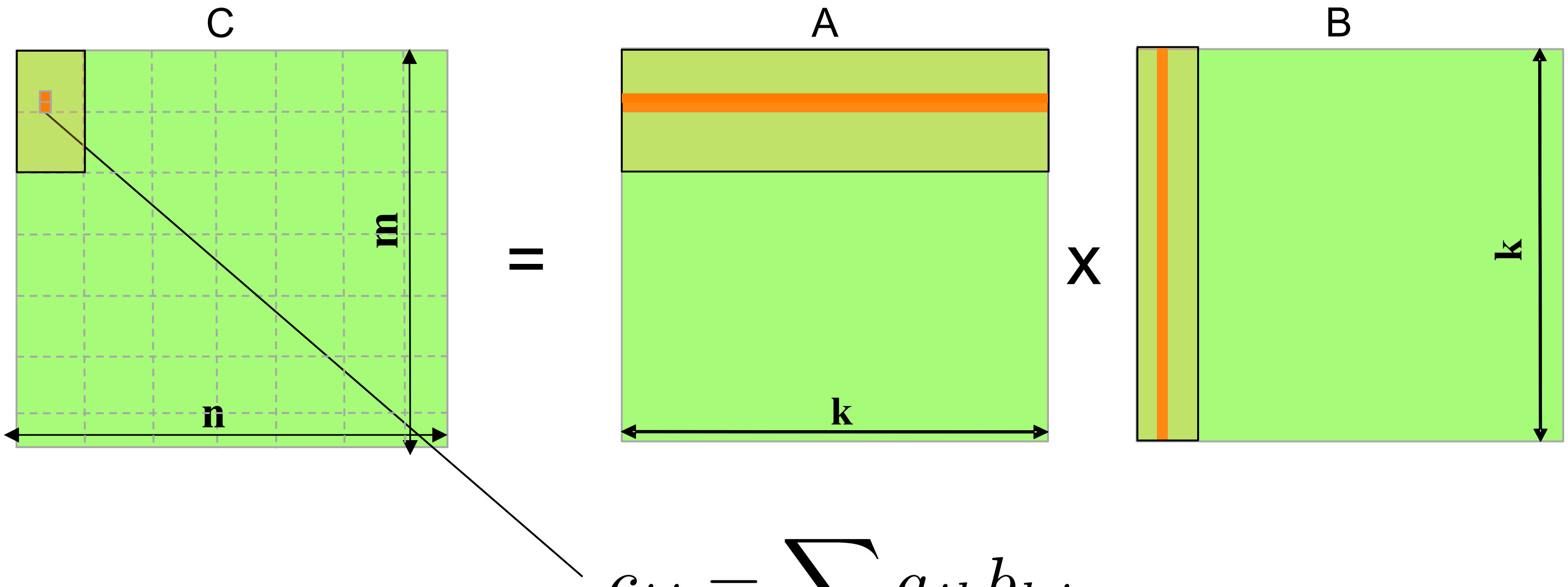
- Naive version: One thread per element in C
  - 2D Grid, 2D block (for example 16 x 16 threads)



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

GPU Matrix multiplication

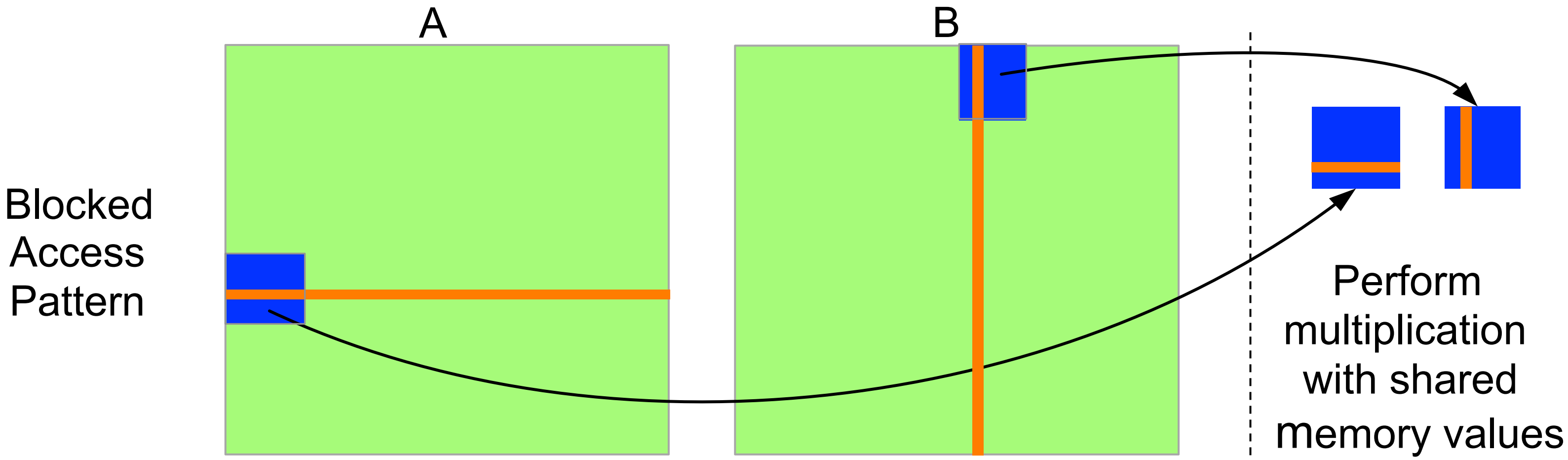
- Register versions: Each thread does 2 elements.
  - `double C_r[2]={0.0, 0.0};`
  - 2D grid is 1/2 in y dimension, 2D block is the same



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

GPU Matrix multiplication

- Shared memory version: 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]`.



  - Start from the naive version and modify it one small step at a time! This is a difficult version to make.

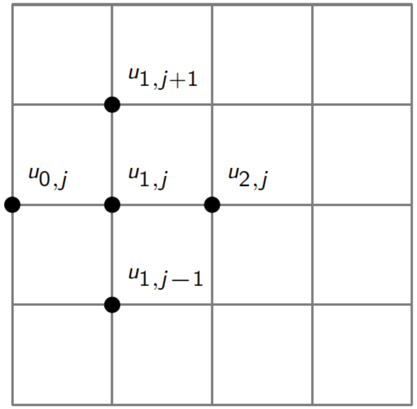
GPU Poison Problem

Possion 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.

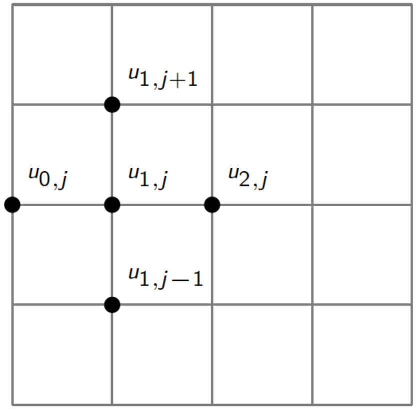
GPU Poisson problem

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



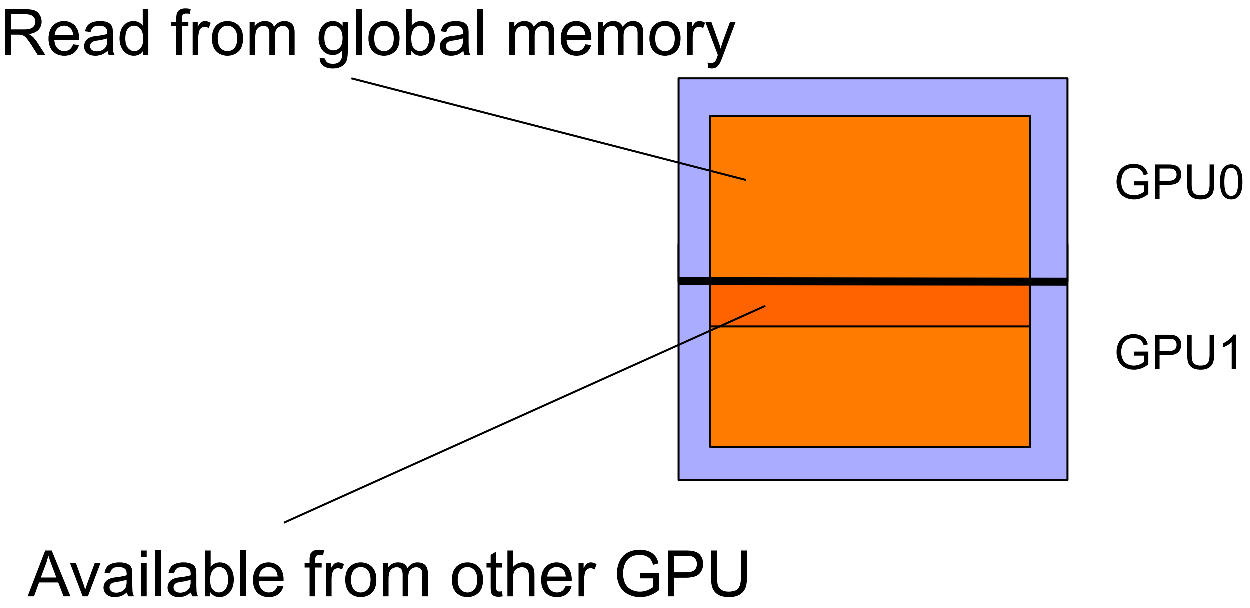
GPU Poisson problem

- 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 ”ghost values” from other GPU



General advice

- Use profiler `nvvp` to help with analysis (ask TAs)
- Assessing speed-up – see the slides (yesterday)
- Manage your time – this is not a quick assignment.
  - Do not get stuck in a question for too long, rather ask the TA or continue to do the simpler questions in the Possion problem.
  - Maybe divide tasks among the members of the group
- Remember to reserve time write good reports
- Ask the TA from 9-17.