CUDA Programming

Week 4. Shared memory and register

Outline

- Shared memory and bank confliction
- Memory padding
- Register allocation
- Example of matrix-matrix multiplication
- Homework

SHARED MEMORY AND BANK CONFLICTION

Memory bank

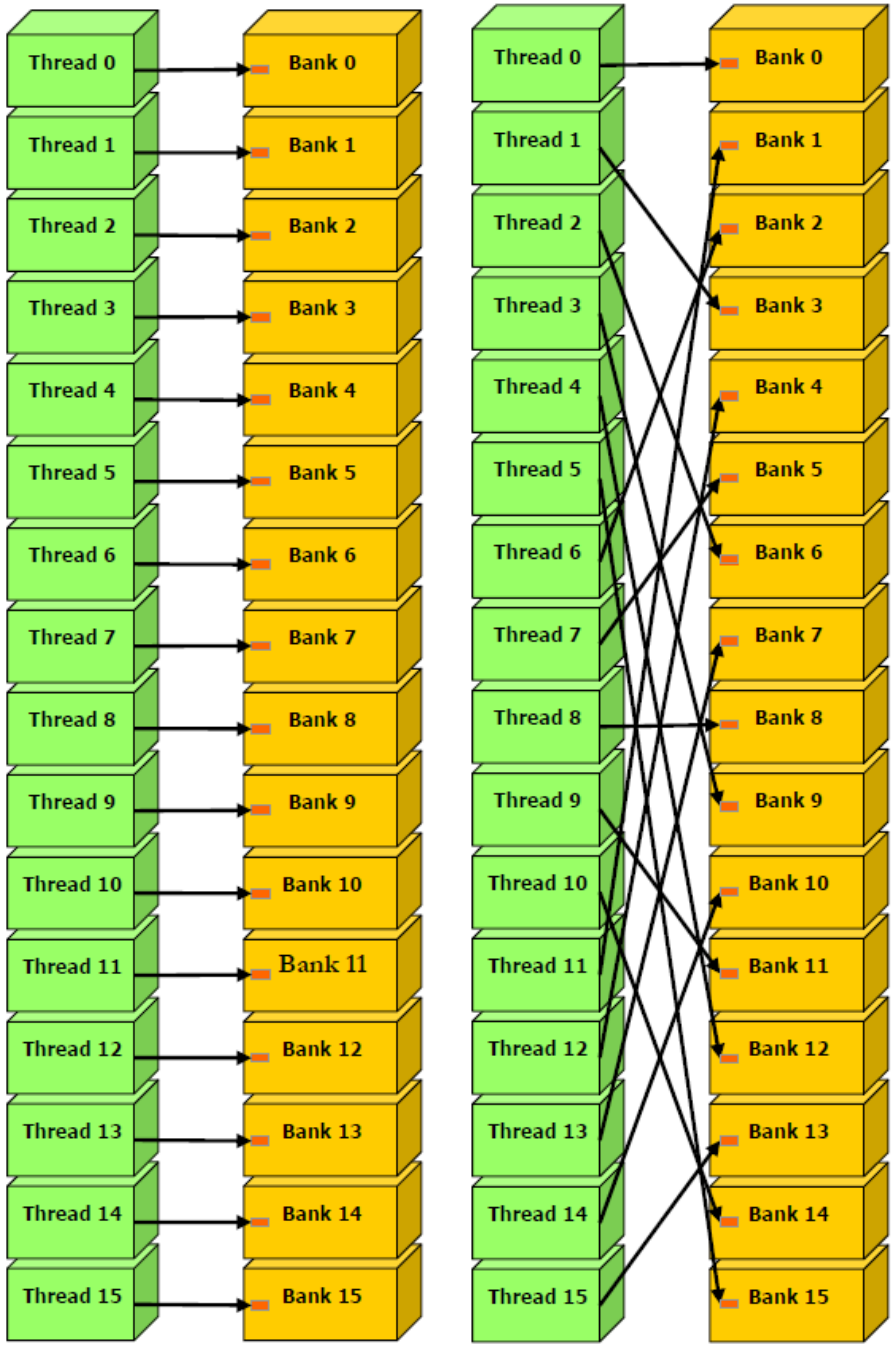
- To achieve high memory bandwidth, shared memory is divided into equally-sized memory modules, called banks
- Memory read/write made of **n** addresses in **n** distinct banks can be serviced simultaneously
- There are 16 banks, which are organized such that successive 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per two clock cycles.

Bank conflict

- If two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be **serialized**.

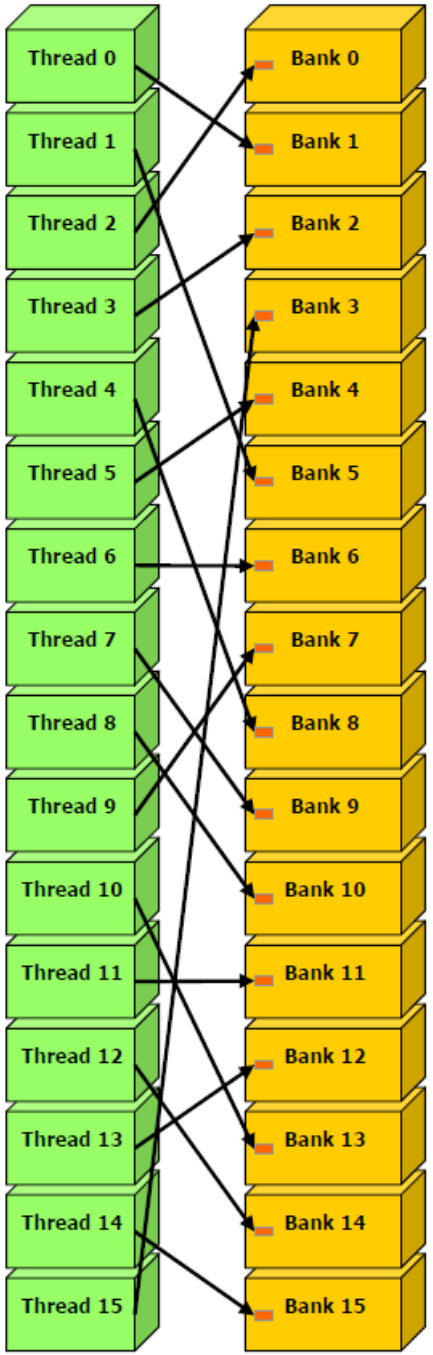
No bank conflict

```
__shared__ float
    shared[32];
float data =
shared[BaseIndex + tid];
Or
float data =
shared[BaseIndex +
    3*tid];
```



Another example

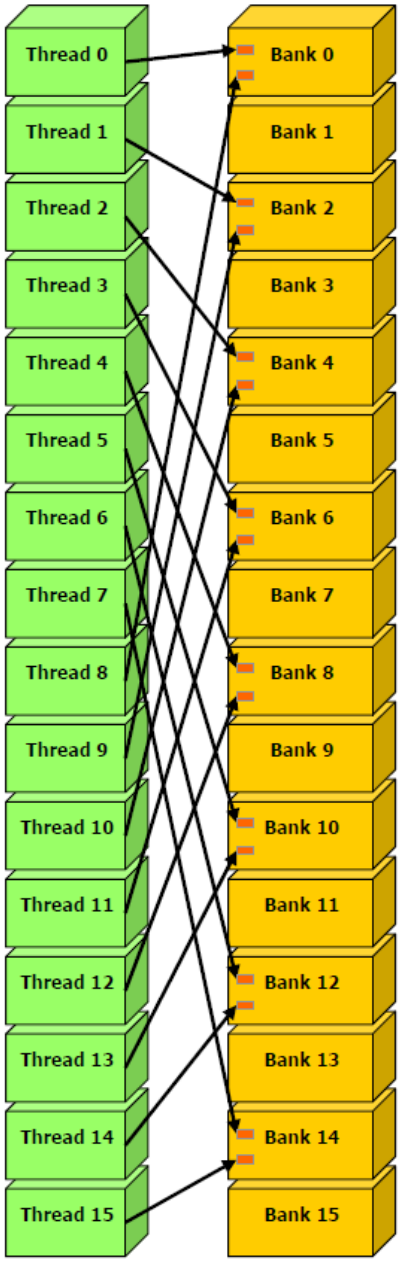
- Random access



2 way bank conflict

- 2-way

```
__shared__ double
    shared[32];
double data =
    shared[BaseIndex +
    tid];
```



4 way bank conflict

__shared__ char shared[32];

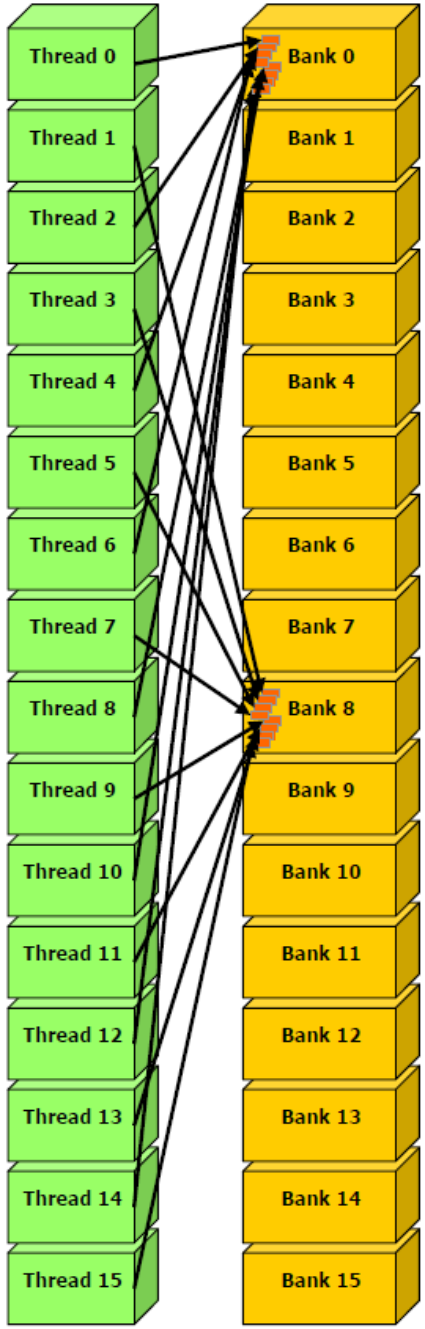
char data = shared[BaseIndex + tid];

Solution:

char data = shared[BaseIndex + 4 * tid];

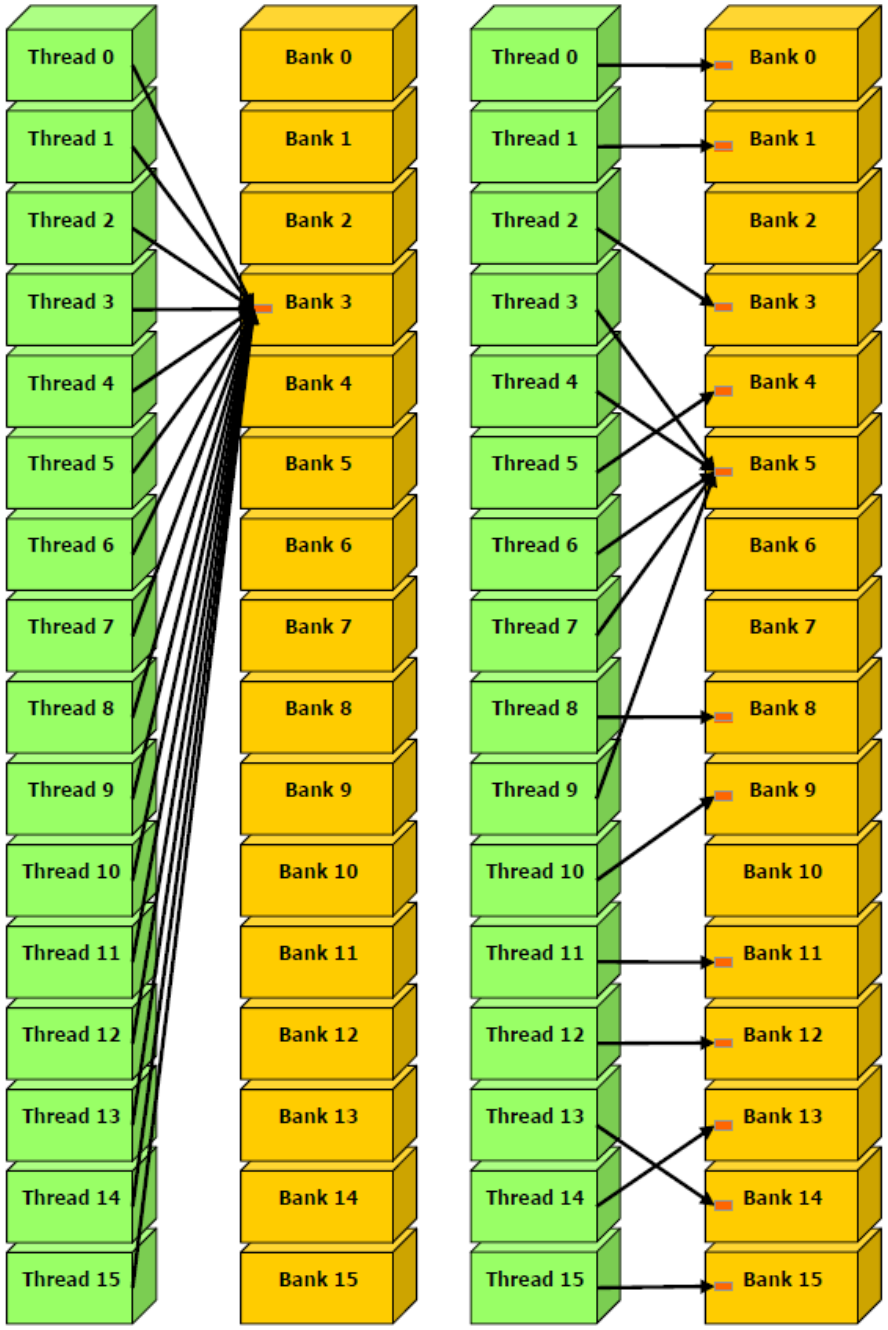
8 way bank conflict

- For example, a structure of 4 floats.



Broadcast

- No bank conflict for broadcast
- Right: This access causes either no bank conflicts if the word from bank 5 is the broadcast during the first step or 2-way bank conflicts.



How to avoid bank conflict?

- The old fashion method: (don't use it)

```
__shared__ int shared_lo[32];
__shared__ int shared_hi[32];
double dataIn;
shared_lo[BaseIndex+tid]= __double2loint(dataIn);
shared_hi[BaseIndex+tid]= __double2hiint(dataIn);
double dataOut =__hiloint2double(shared_hi[BaseIndex+tid],
                  shared_lo[BaseIndex+tid]);
```

- For array of structures, bank conflict can be reduced by changing it to structure of array.
- Memory padding

MEMORY PADDING

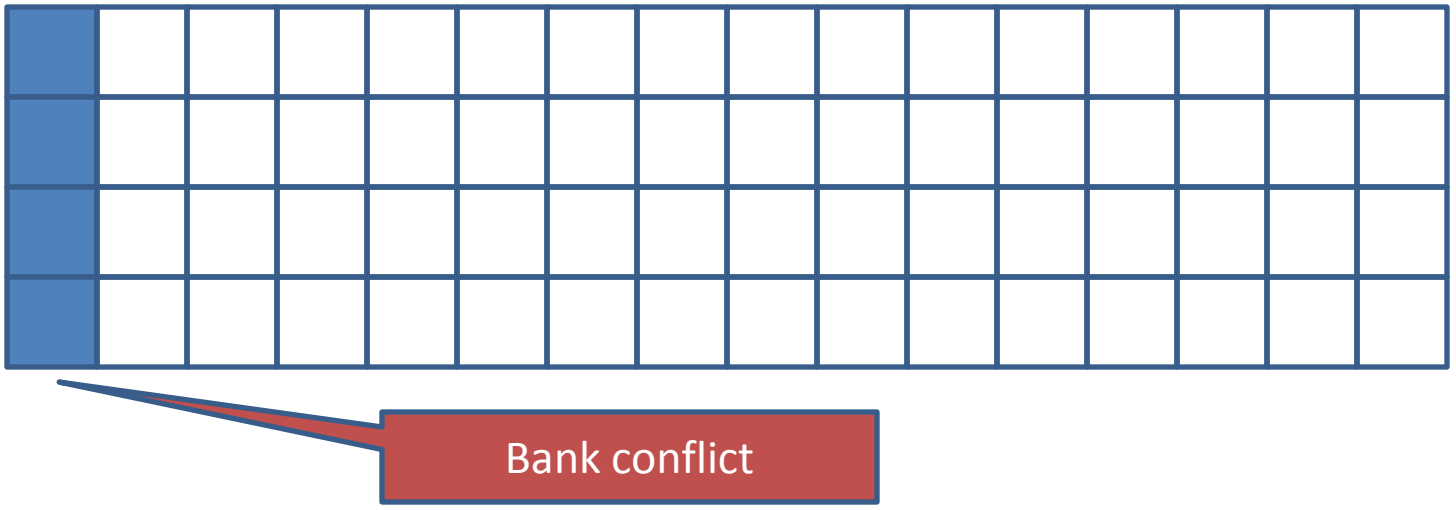
Example: C=A*B

```
__global__ void matMult(const float* a, size_t lda,
   const float* b, size_t ldb, float* c, size_t ldc, int n){
   __shared__ float matA[BLOCK_SIZE][BLOCK_SIZE];
   __shared__ float matB[BLOCK_SIZE][BLOCK_SIZE];
   const int tidc = threadIdx.x;
   const int tidr = threadIdx.y;
   const int bidc = blockIdx.x * BLOCK_SIZE;
   const int bidr = blockIdx.y * BLOCK_SIZE;
   ...
   for(j = 0; j < n; j += BLOCK_SIZE) {
      matA[tidr][tidc] = a[(tidr+bidr)*lda+tidc+j];
      matB[tidr][tidc] = b[(tidr+j)*ldb+tidc+bidc];
      __syncthreads();
      for(i = 0; i < BLOCK_SIZE; i++)
          result += matA[tidr][i] * matB[i][tidc];
      ...
```

Memory access pattern

- Suppose block size = 16
- The memory access of `matA[tidr][i]`

- The memory access of `matB[i][tidc]`

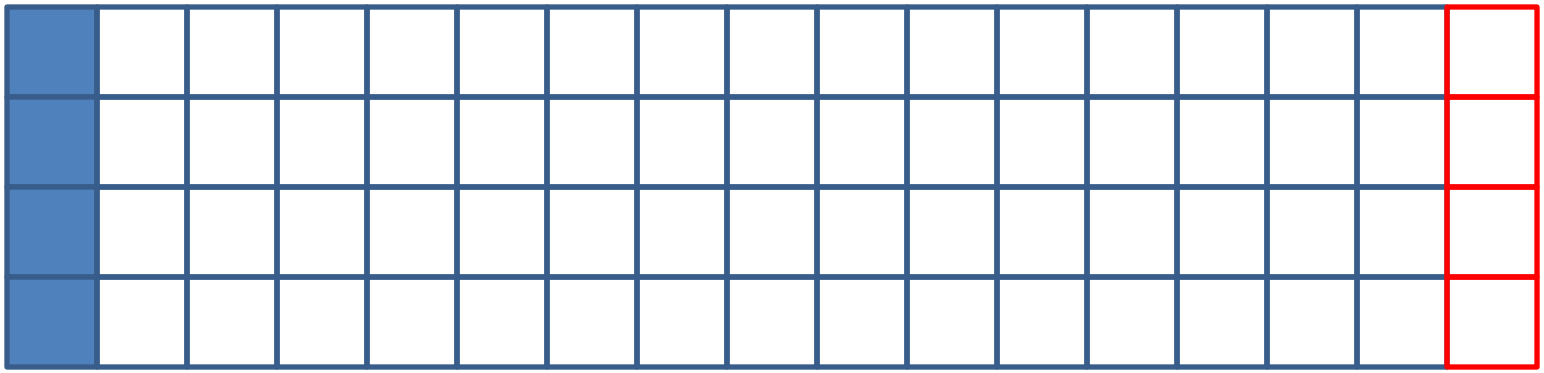


Memory padding

- When declaring the shared memory, pad extra space

```
matB[BLOCK_SIZE][BLOCK_SIZE+1];
```

- When access the memory, the addresses become 0, 17(=1 mod 16), 34(=2 mod 16) …

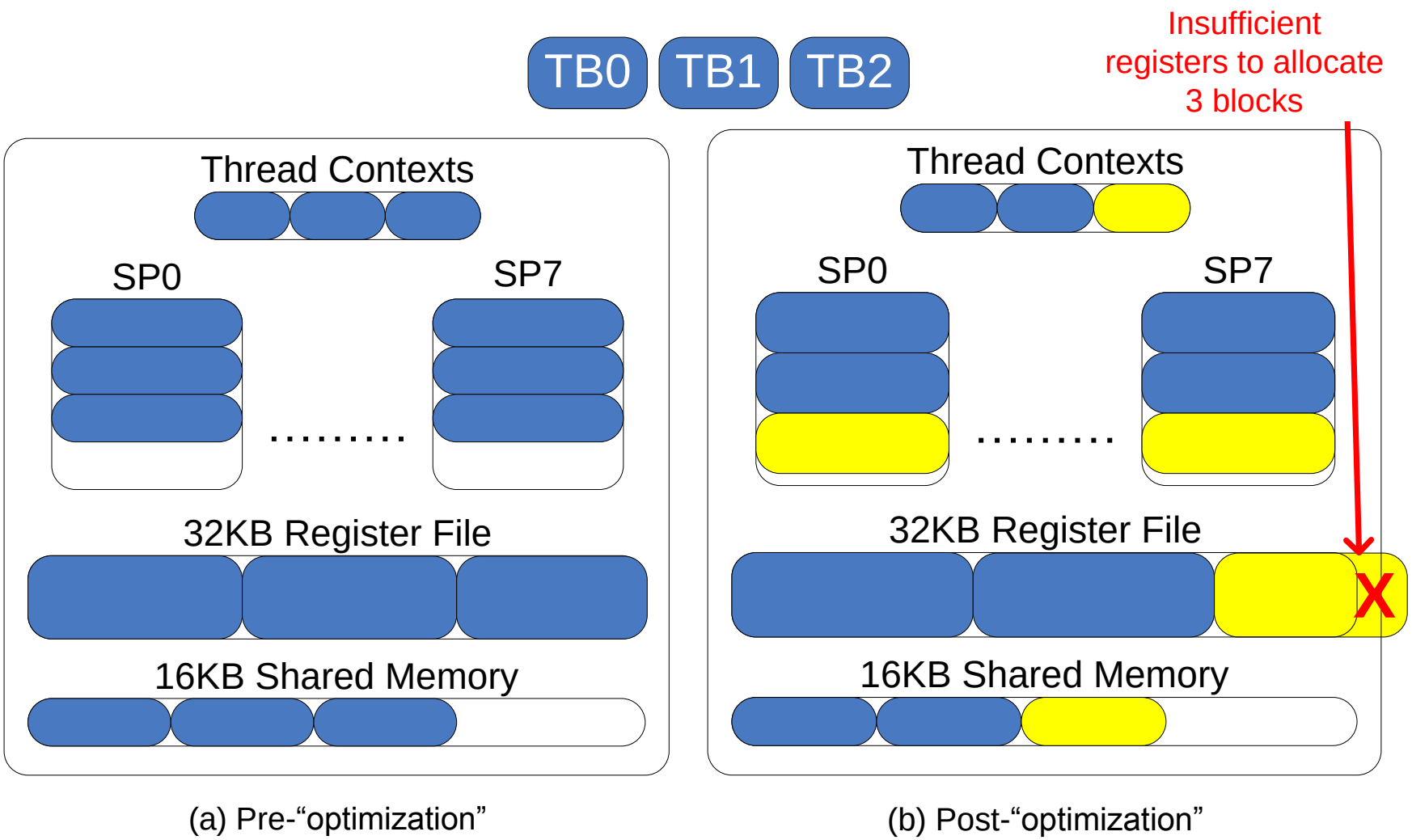


REGISTER ALLOCATION

Register partition

- In G80, 8,192 registers in each SM.
  - Each register is 4byte long. → 32KB
- The automatic variables declared in a CUDA kernel are placed into registers
- Register file is the fastest and the largest on-chip memory
  - Keep as much data as possible in registers

Register allocation example



(a) Pre-“optimization”

(b) Post-“optimization”

Example: Configuration 1

- Assume that each block has 256 thread, and each thread uses 10 registers.
  - 3 Blocks can run on each SM
  - Requires 256*3*10=7,680 registers (<8,192)
  - 768/32=24 warps.
- Suppose for every 4 instructions there is a memory load. Each takes 200 cycles.
  - For normal instruction, a warp takes 4 cycle
  - 4x4x24=384 cycles > 200 cycles

Example: Configuration 2

- If a compiler can 11 registers to change the dependence pattern so that 8 independent instructions exist for each global memory load
  - Each block needs 256*11=2,816 registers
  - Only two blocks can run on each SM
- However, one only needs 200/(8*4) = 7 Warps to tolerate the memory latency
- Two Blocks have 16 Warps. The performance can be actually higher!

Vasily Volkov's matrix-matrix multiplication

CASE STUDY

Some analysis

- Counter i in “for(int i = 0; i < n; i++)” consumes 2KB in registers for block size = 512
  - Similarly, i++ translates into 512 increments
- Use smaller block size
  - If block size=64, 64 increments and 256 bytes
  - But we need that many threads to hide communication

Smaller block size

- Strip-mine longer vectors into shorter at the program level if necessary
  - E.g. instead of using “float a;” and BS=512 use “float a[8];” and BS=64
  - Instead of “a += b;”, BS=512 use “a[0] += b[0]; …; a[7] += b[7];”, BS=64
  - Use more registers for a, but less space for i, as well as less computation for increasing i.

Matrix-Matrix Multiply: C=C+A*B

- Keep A's and C's blocks in registers
- Keep B's block in a shared storage
- No other sharing is needed if C's height = BS.
  - BS=64 is the best result from experiments
- Choose large enough width of C's block
  - 16 is enough as 2/(1/64+1/16) = 26-way reuse
- Choose a convenient thickness for A's and B's blocks

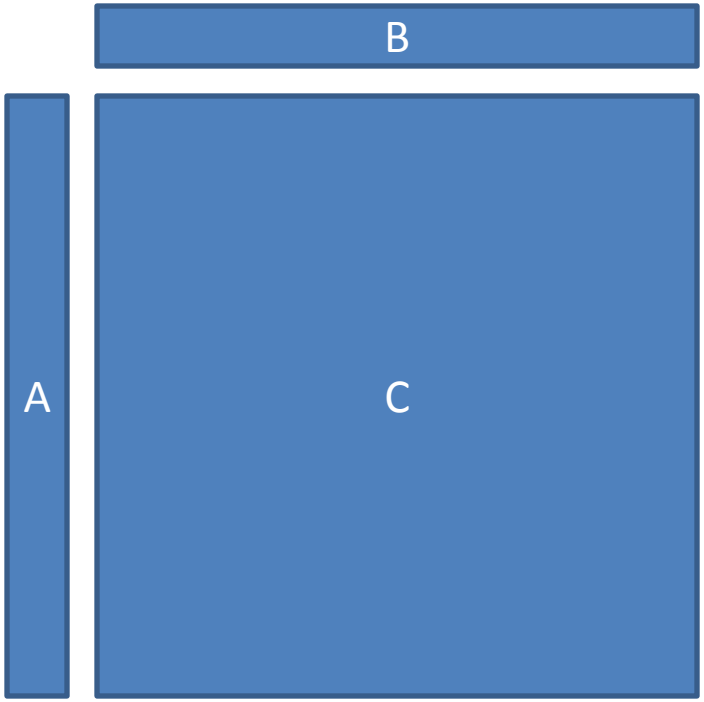
Code

```
__global__ void sgemmNN( const float *A, int lda,
const float *B, int ldb, float* C, int ldc, int k,
float alpha, float beta ){
   // Compute pointers to the data
   A += blockIdx.x*64+threadIdx.x +threadIdx.y*16;
   B += threadIdx.x+(blockIdx.y*16+threadIdx.y)*ldb;
   C += blockIdx.x*64+threadIdx.x+(threadIdx.y+
        blockIdx.y * ldc ) * 16;

   // declare the shared memory
   __shared__ float bs[16][17];
   Declare on chip float c[16] =
                {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
```

The framework

```
const float *Blast = B + k;
do{
#pragma unroll
    for( int i = 0; i < 16; i += 4 )
        bs[threadIdx.x][threadIdx.y+i] = B[i*ldb];
    // Read next B's block
    B += 16;
    __syncthreads();
    ... // the computation part: next slide
    __syncthreads();
__} while( B < Blast );

//Store C's block to memory
for( int i = 0; i < 16; i++, C += ldc )
    C[0] = alpha*c[i] + beta*C[0];
}
```

```
#pragma unroll
// The bottleneck: Read A's columns
for( int i = 0; i < 16; i++, A += lda ){
    c[0]  += A[0]*bs[i][0];
    c[1]  += A[0]*bs[i][1];
    c[2]  += A[0]*bs[i][2];
    c[3]  += A[0]*bs[i][3];
    c[4]  += A[0]*bs[i][4];
    c[5]  += A[0]*bs[i][5];
    c[6]  += A[0]*bs[i][6];
    c[7]  += A[0]*bs[i][7];
    c[8]  += A[0]*bs[i][8];
    c[9]  += A[0]*bs[i][9];
    c[10] += A[0]*bs[i][10];
    c[11] += A[0]*bs[i][11];
    c[12] += A[0]*bs[i][12];
    c[13] += A[0]*bs[i][13];
    c[14] += A[0]*bs[i][14];
    c[15] += A[0]*bs[i][15];
}
```



Outer product

Some statistics

- The table from Vasily's talk

	CUBLAS 1.1	Our code
Registers per thread	15	30
Shared memory per thread	8.3 KB	1.1 KB
Vector length	512	64
Occupancy (8800 GTX)	67%	33%
Performance (8800 GTX)	128 Gflop/s	205 Gflop/s

- This record keeps a while until Feb 2010

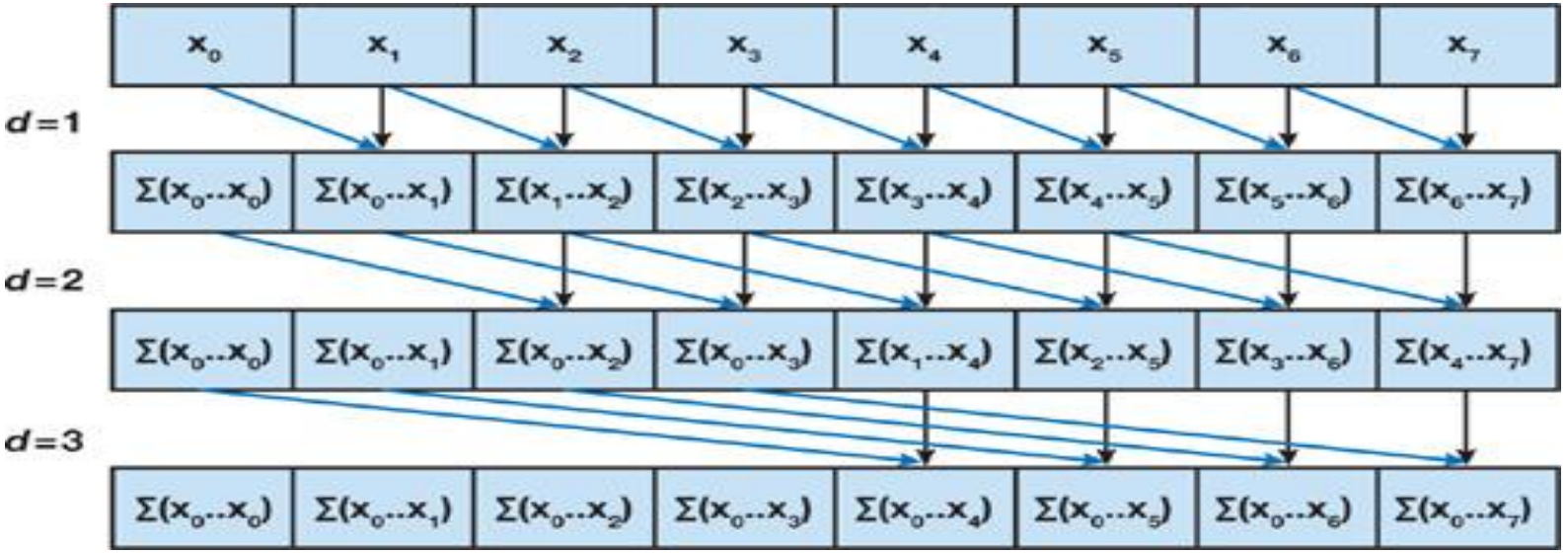
HOMEWORK

Prefix Sum

- EX: a = [3 1 7 0 4 1 6 3]. The prefix sum of a is [0 3 4 11 11 15 16 22].
- One of the fundamental tool in many algorithms
  - Radix sort, compression, etc
- The sequential code

```
ps[0] = 0;
For(int j=1, j<n; j++)
    ps[j] = ps[j-1] + a[j-1];
```

Parallel prefix sum



for d = 1 to $\log_2 n$ **do**
  for all k in parallel **do**
    if $k >= 2^{d}$ **then**
      $x[k] = x[k - 2^{d-1}] + x[k]$

Homework

- Play with Vasily's code SGEMM
  - In the repository of the google group
- Read parallel prefix sum implementation on GPU
  - http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html
  - Implement your own