CUDA MATMULT REG / SHARED

WIM BOHM, SANJAY RAJOPADHYE
CS675 (2010) CUDA PERFORMANCE STUDY GROUP

SHARED/SHARED OPERANDS

In-product in Matmult PA1:

```
for(int e=0; e<BLOCK_SIZE; ++e)
   Cvalue += shared_A[thread_row][e] * shared_B[e][thread_col];
```

Cuda allows both operands of operations (e.g. mulltiply add) to be in shared memory.

But, is that efficient?

MICRO BENCHMARK

Do a micro benchmark.

Like we studied coalescing in vector add micro benchmark.

Micro benchmark: comparison of 2 (or more) ways of performing a computation in a simplest possible code, that only changes ONE aspect of that code, with the goal of understanding that one aspect.

In our case here: what is the cost, if any, of having both operands in shared memory vs the cost of 1 operand in a register and 1 in shared memory.

Once we understand this behavior we can exploit it in a larger application, e.g. matmult

SHARED/SHARED VS. REG/SHARED

```
// sha sha
for(i = 0; i < NUMREPS; i++)
   for(j = 0; j < VALSPERTHREAD; j++)
     local_C += shared_A[j] * shared_B[j];

// reg sha
for(i = 0; i < NUMREPS; i++)
   for(j = 0; j < VALSPERTHREAD; j++)
     local_C += shared_A[j] * local_B[j];

//  And that is the ONLY difference in the whole code!!
```

CLEANED UP
SHARED SHARED VS REG SHARED ASM

```
label1:
mov.b32 $r0, s[0x0060]
mad.rn.f32 $r1, s[0x0020], $r0, $r1
mov.b32 $r0, s[0x0064]
mad.rn.f32 $r1, s[0x0024], $r0, $r1
...
mov.b32 $r0, s[0x009c]
mad.rn.f32 $r1, s[0x005c], $r0, $r1
add.b32 $r31, $r31, 0x00000001
set.ne.s32 $p0|$o127, $r31,
    c1[0x0000]
@$p0.ne bra.label label1
```

```
label1:
mad.rn.f32 $r3, s[0x0020], $r2, $r3
mad.rn.f32 $r3, s[0x0024], $r4, $r3

…

mad.rn.f32 $r3, s[0x005c], $r32, $r3
add.b32 $r0, $r0, 0x00000001
set.ne.s32 $p0|$o127, $r0,
    c1[0x0000]
@$p0.ne bra.label label1
```

shared shared twice as slow as reg shared

REGISTER / SHARED MATMULT

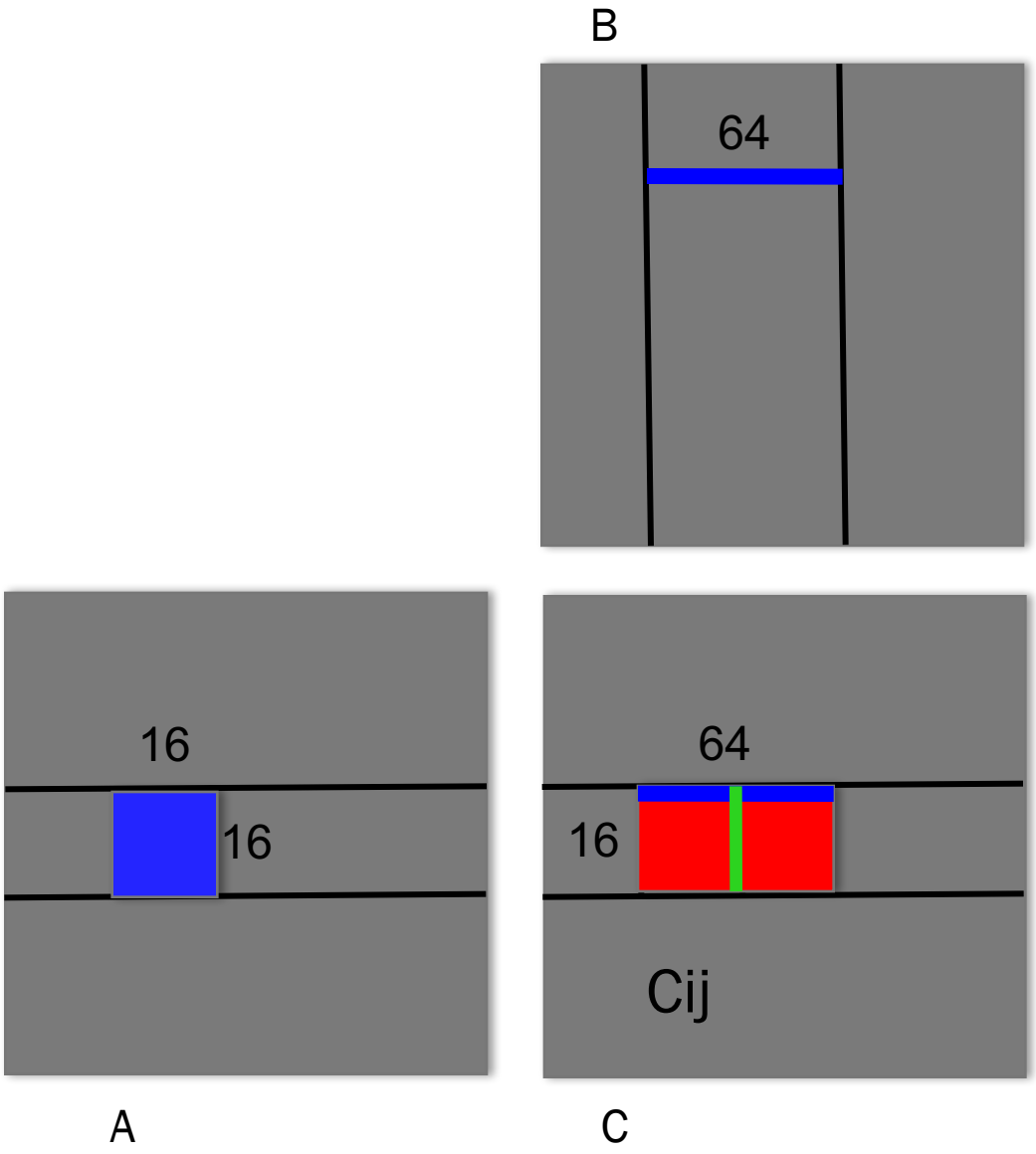
Thread block: 64x1, C footprint: 64x16

- each thread has a column of 16 C values in registers
- dim3 dimBlock(64,1)
- dim3 dimGrid(C.width/64,C.height/16)

A is read into shared memory in 16x16 blocks

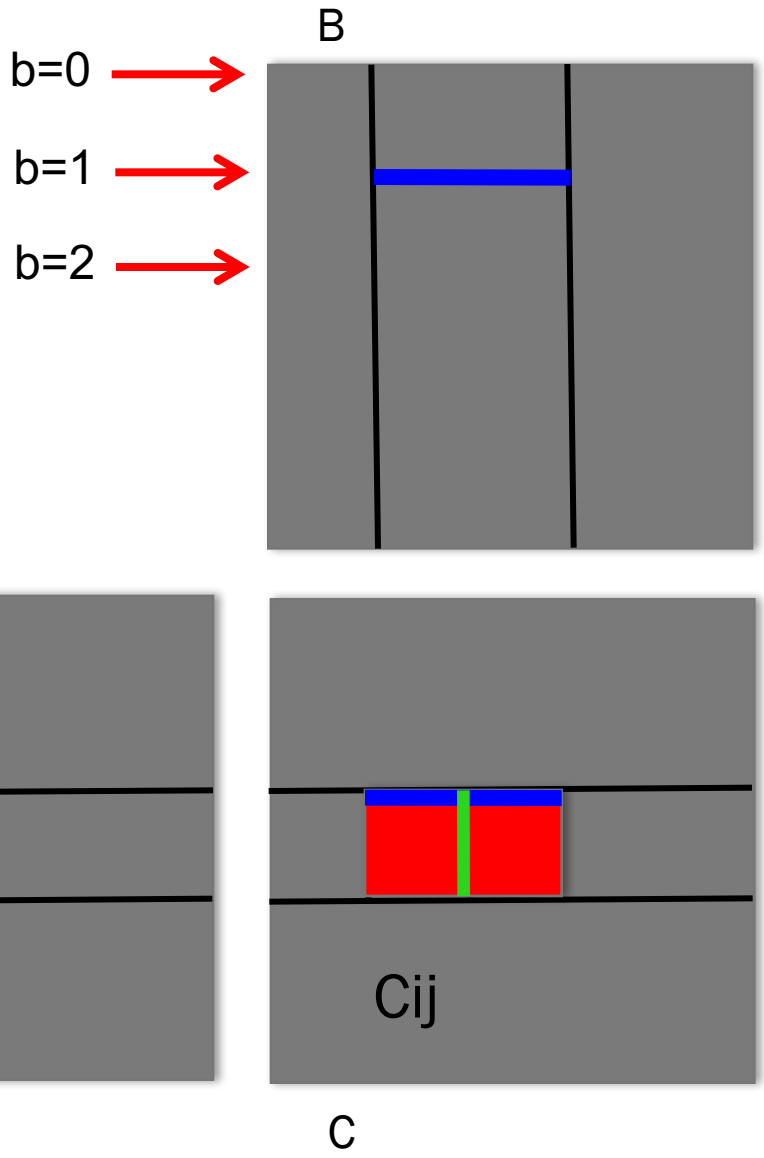
B is read into registers (1 per thread) one row part at the time

better performance: pull loop invariant code out


B
64
16
16
64
16
Cij
A
C

```
for(b=0; b<A.width/16; ++b)
    // locate A block
    // read 4 A elements into shared memory As
    // locate base: starting row part in B
    for(k=0;k<16;++k)
      // read B element Bel   ( B[k,j] )
      for(i=0;i<16;++i)
         C_local[i] += As[i*16+k]*Bel

for(i=0;i<16;++i)
   write C_local[i] to C
```



PERFORMANCE

We compare our code to the matrix multiply code from CudaBLAS for square 2048x2048 matrices on a Tesla1060.

Our original 16x64 C footprint register shared code:

265 GFLOPS (GigaFlops per Second)

CudaBLAS

354 GFLOPS

PERFORMANCE

We compare our code to the matrix multiply code from

CudaBLAS for square 2048x2048 matrices on the Tesla1060.

Our original 16x64 C footprint register shared: 265 GFLOPS

CudaBLAS: 354 GFLOPS

First optimization: loop unrolling, pointer arithmetic (avoiding a*i+b by pointer addition), code hoisting (taking loop invariant code out of loop bodies) 321 GFLOPS

PERFORMANCE

We compare our code to the matrix multiply code from

CudaBLAS for square 2048x2048 matrices on the Tesla1060.

Our original 16x64 C footprint register shared: 265 GFLOPS

CudaBLAS: 354 GFLOPS

Loop unrolling, code hoisting, pointer arith.: 321 GFLOPS

Second optimization

32x64 C footprint (less glob. mem. traffic)

4x16 thread block (simpler mapping of thread space to data space)

Transposing 32x16 A into 16x33 A-shared block

(accesses are consecutive, no bank conflicts) 372 GFLOPS

CAN WE DO BETTER?

- REG/REG:
  Every operand in register?

- We never made it work, but maybe you can (we have bigger GPUs now with more registers….)