CME213/ME339
Lecture 9

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

Occupancy calculator

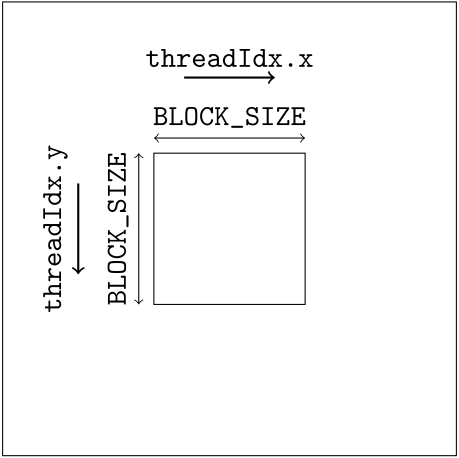
- Google "`occupancy calculator`"
- Input required:
  1. Compute capability: 2.0
  2. Threads per block
  3. Shared memory
- Get these numbers by compiling your code with the option: `--ptxas-options=-v`

Matrix-matrix product

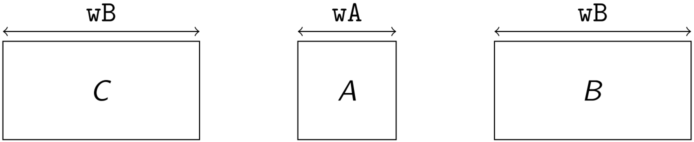
Let's illustrate these concepts using a matrix-matrix product:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Layout



Matrix sizes



```
__global__ void
matrixMul_slow( float* C, float* A, float* B, int wA, int wB)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * blockDim.y * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = blockDim.x * bx;
...
```

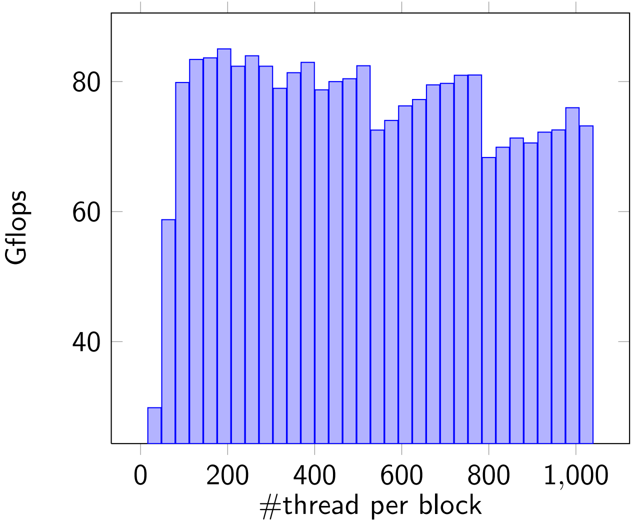
```
...
    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0.0f;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin; a <= aEnd;
         ++a, b += wB) {

      // Multiply the two matrices together;
      // each thread computes one element
      // of the block sub-matrix
      Csub += A[a + wA * ty] * B[b + tx];
    }

    // Write the block sub-matrix to device memory
    int c = wB * blockDim.y * by + blockDim.x * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Output from compiler

```
ptxas info    : Compiling entry function
                '_Z12matrixMul_v2PfS_S_ii' for 'sm_20'
ptxas info    : Function properties for _Z12matrixMul_v2PfS_S_ii
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 18 registers, 64 bytes cmem[0]
```

18 registers, 64 bytes constant memory, no shared memory.

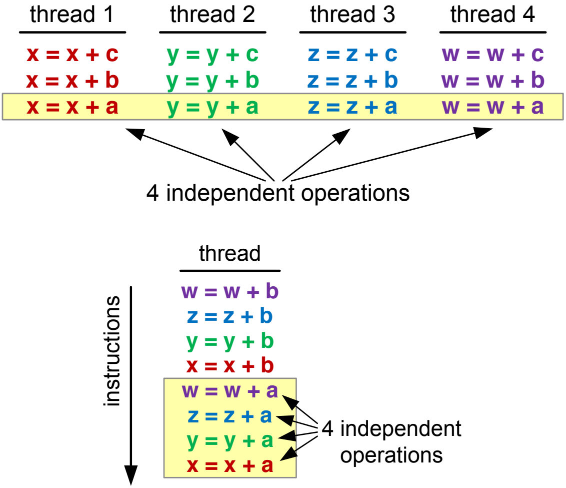
Threads per block



Performance follows the occupancy.

Generating concurrency

- In some cases running enough threads is not possible to reach peak performance.
- For example, if the latency is 24 cycles, we need as many as 24 warps to hide this latency, which is 50% occupancy.
- Parallelism can also be generated at the level of a single thread. That is the hardware will take advantage of multiple instructions if there are no dependencies between them.


thread 1
x = x + c
x = x + b
x = x + a
thread 2
y = y + c
y = y + b
y = y + a
thread 3
z = z + c
z = z + b
z = z + a
thread 4
w = w + c
w = w + b
w = w + a
4 independent operations
instructions
thread
w = w + b
z = z + b
y = y + b
x = x + b
w = w + a
z = z + a
y = y + a
x = x + a
4 independent operations

Number of grid blocks and number of threads per block — heuristics

- We need approx. 18–27 warps = 576–864 threads per SM for good performance (less if you can issue independent instructions).
- The number of blocks must be $>$ than the number of SMs, so each SM has at least one block.
- Typically, # of blocks / # of multiprocessors > 2:
  - Multiple blocks can run concurrently in a multiprocessor.
  - Blocks that aren't waiting at a `__syncthreads()` keep the hardware busy.
  - Subject to resource availability – registers, shared memory.

Performance of matrix-matrix product

- The number of Gflops we achieved with the matrix-matrix product is disappointing. It peaks at about 80 Gflops.
- Consider the memory access pattern:

```
int aBegin = wA * blockDim.y * by;
int bBegin = blockDim.x * bx;
float Csub = 0.f;

for (int a = aBegin, b = bBegin; a <= aEnd;
     ++a, b += wB)
    Csub += A[a + wA * ty] * B[b + tx];

int c = wB * blockDim.y * by + blockDim.x * bx;
C[c + wB * ty + tx] = Csub;
```

Memory access

- `B[b + tx]`: excellent access if `blockDim.x == 32`
- `A[a + wA * ty]`: since we have `++a` between iterations, we can use the cache but chances are data get evicted before we finish the block.
- This suggests using shared memory.
- Load a block of A and a block of B in shared memory and then perform multiplication.

Block multiplication

```
    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin; a <= aEnd;
             a += aStep, b += bStep) {

        // Declaration of the shared memory arrays used to
        // store the sub-matrices for A and B
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from device memory
        // to shared memory
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        __syncthreads();
...
```

```
...
        // Multiply the two blocks together
#pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }
```

Memory access pattern

- `As[ty][tx] = A[a + wA * ty + tx]`: excellent; coalesced; 128 bytes if `BLOCK_SIZE == 32`
- `Bs[ty][tx] = B[b + wB * ty + tx]`: excellent.
- `Bs[k][tx]`: conflict free access.
- `As[ty][k]`: broadcast access; conflict free.
- Bandwidth from memory: `GeForce GTX 480`, bandwidth = 177.4 GB/s; peak flops = 1344.96 GFlops.
- To hide memory access we need at least ~ 30 flops per word read from memory.

Flop count

- Number of words read by a thread: 2
- Number of flops: 2 `BLOCK_SIZE`
- If `BLOCK_SIZE == 32`, the ratio is 32. Enough to hide memory access!
- So we should get peak performance:

```
CUBLAS           784.5752 GFlop/s, Time = 0.00100 s
CUDA matrixMul   235.8730 GFlop/s, Time = 0.00333 s
```

CUBLAS: library for linear algebra that is part of the CUDA SDK.

Shared memory access

- We getting closer to CUBLAS but there is still room for improvement.
- We are currently slowed down by the access to the shared memory. It takes 4 cycles to read `Bs[k][tx]` and `As[ty][k]` but only one cycle to perform the operation.

Instruction throughput

Guidelines to improve instruction throughput:

- Use single precision whenever possible.
- Use bitwise operations to calculate division or modulo by powers of 2

```
i/2^k  --> i >> k
i%2^k  --> i & (2^k-1)
```

Reciprocal square roots

The reciprocal square root should always be invoked explicitly as `rsqrtf()` for single precision and `rsqrt()` for double precision.

This is a common operation in graphics and therefore the hardware is highly optimized for this operation.

Math functions and constants

- When doing floating point arithmetic use single-precision floating-point constants, defined with an `f` suffix such as `3.141592653589793f`, `1.0f`, `0.5f`.
- Use the appropriate math function

`rsqrtf(x)`, `sqrtf(x)`	`rsqrt(x)`, `sqrt(x)`
`expf(x)`, `logf(x)`	`exp(x)`, `log(x)`
`sinf(x)`, `cosf(x)`, `tanf(x)`	`sin(x)`, `cos(x)`, `tan(x)`

Fast math

Some functions have a very fast implementation in hardware but with lower accuracy (?). Their names are prepended by underscores.

`__fdividef(x,y)` [x/y],
`__sinf(x,y)`, `__cosf(x,y)`, `__tanf(x,y)`,
`__expf(x,y)`, `__logf(x,y)`

See the programming guides for a full list.