Use registers and multiple outputs per thread on GPU

Vasily Volkov
UC Berkeley

June 30, 2010

Occupancy is overrated

- It is widely recommended to optimize for higher occupancy

- Indeed, you *can* use higher occupancy to hide arithmetic and memory latencies better
  - *But don't have to!*
- **You can hide latencies keeping occupancy low**
  - Low occupancy has performance advantages

Hiding arithmetic pipeline latency

- Latency of arithmetic instructions is ≈24 cycles
  - Time between collecting operands and when result is available
- But throughput is 4 cycles per (SIMD) instruction
  - 8 scalar instructions complete each cycle on each SM
  - (here we are talking about “streaming processors” only)
- Thus, 24/4 = 6 SIMD instructions must be in the flight, per SM
- E.g. they may come from 6 warps (=192 threads)

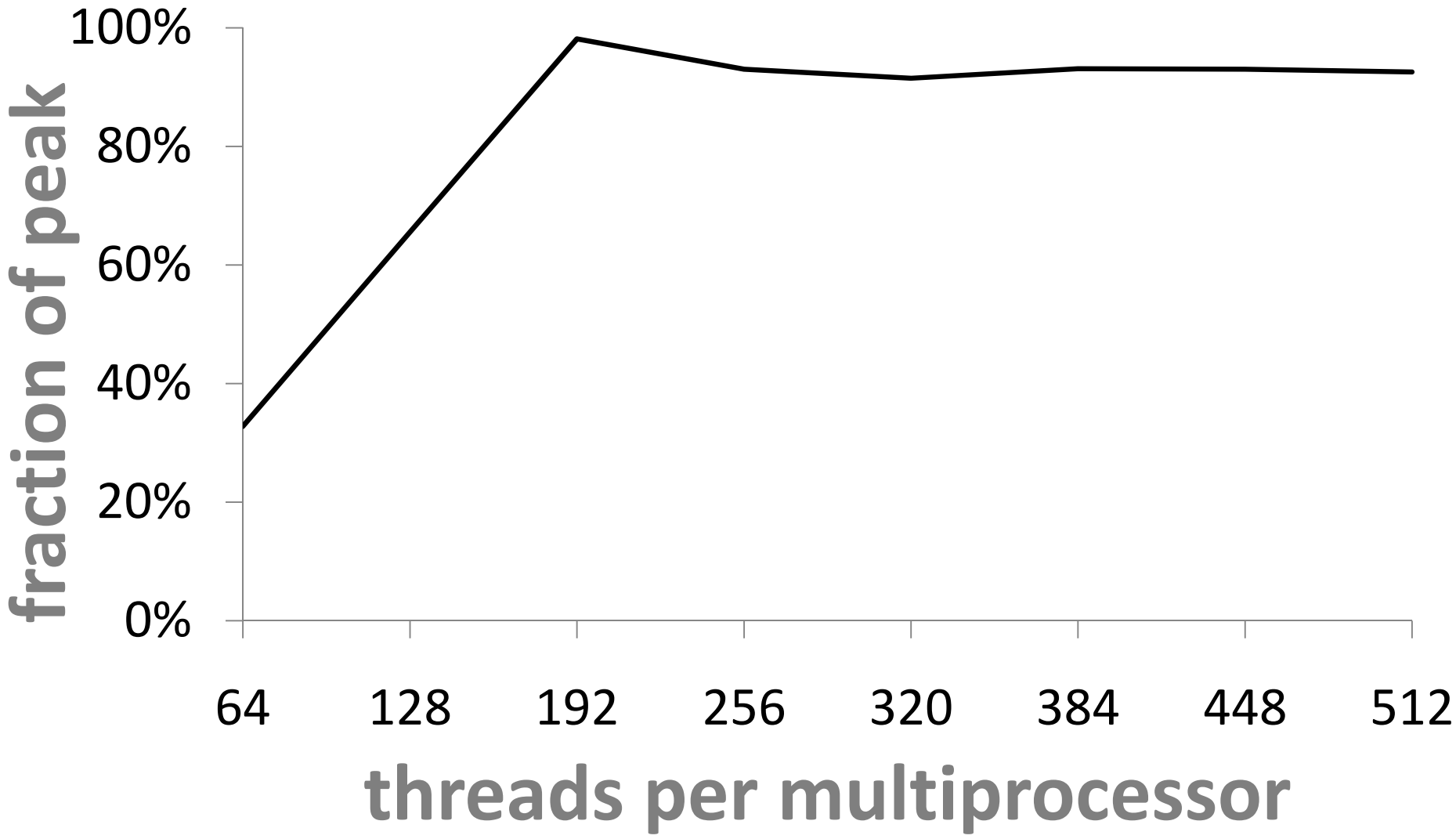
Experimental setup

- Let's check our hypotheses with experiments
- 1024 dependent instructions in a loop:

```
for( int i = 0; i < 1024*1024; i += 1024 )
{
#pragma unroll
    for( int j = 0; j < 1024; j++ )
    {
        a = a * b + c;
    }
}
```

- How its performance varies under occupancy?

Performance vs. Occupancy



Experimental validation: 192 threads is enough

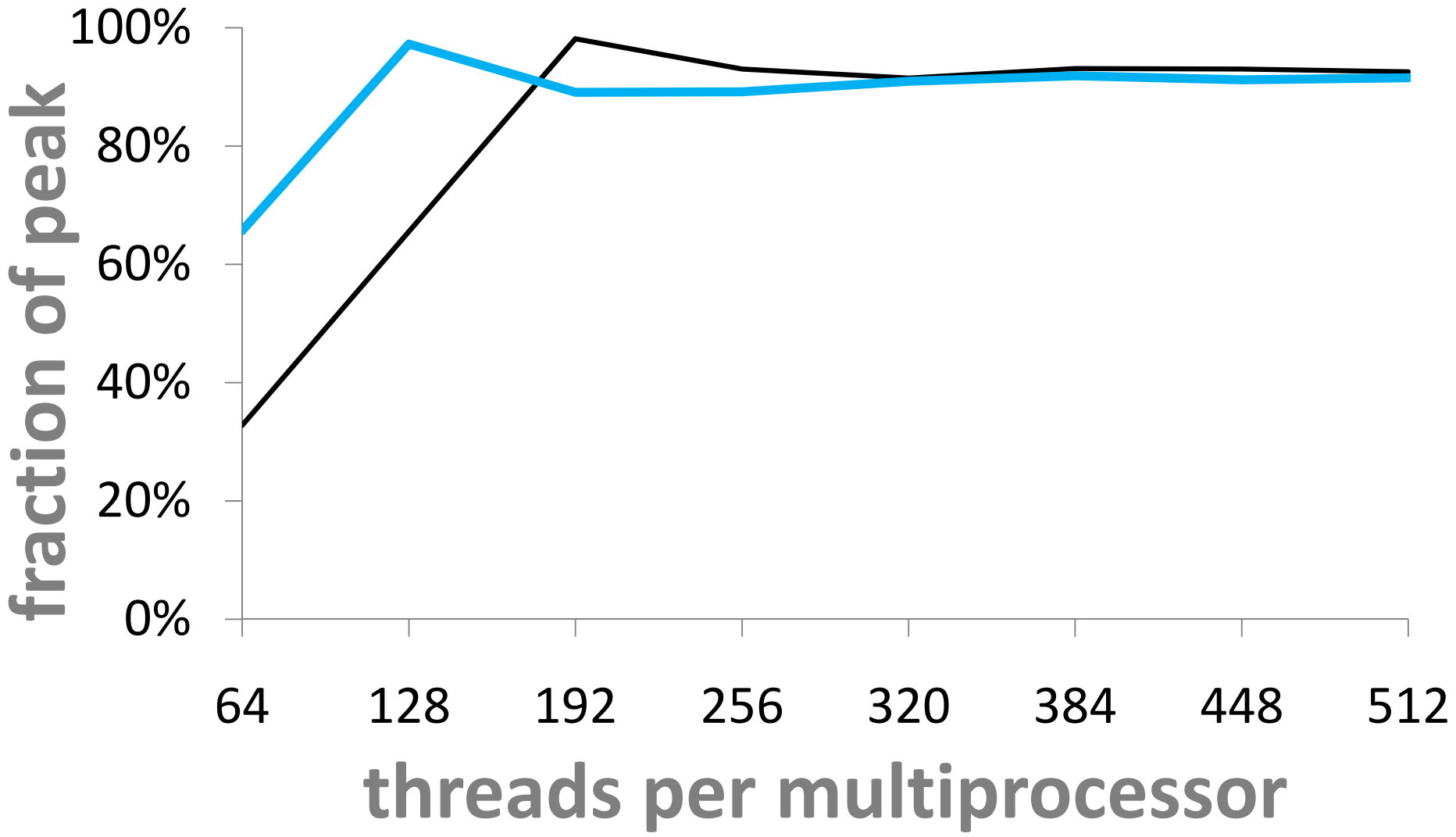
Use instruction level parallelism (ILP)

- What if we supply independent instructions from same thread?

```
for( int i = 0; i < 1024*1024; i += 128 )
{
#pragma unroll
    for( int j = 0; j < 128; j++ )
    {
        a = a * b + c;
        d = d * b + c;
    }
}
```

- Shouldn't this require fewer threads to fill the pipeline?

More ILP needs less warps



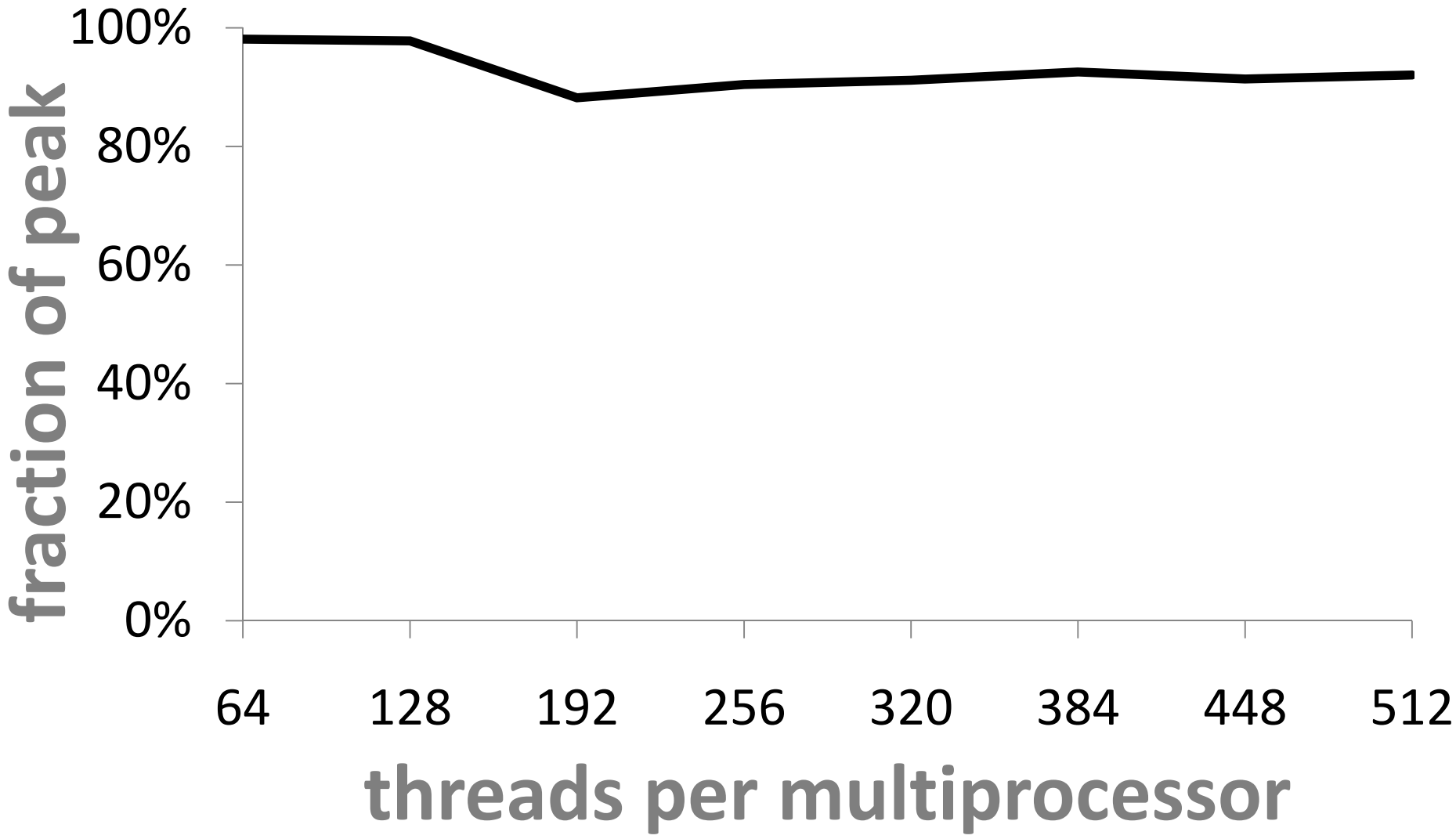
Now 128 threads suffice

Pushing it further

- Can we hide all latency using only 64 threads?
  - (Can't run fewer threads due to other bottlenecks)

```
for( int i = 0; i < 1024*1024; i += 128 )
{
#pragma unroll
    for( int j = 0; j < 128; j++ )
    {
        a = a * b + c;
        d = d * b + c;
        e = e * b + c;
    }
}
```

64 threads is enough



We hid all latency using only 6% occupancy

Does ILP happen in practice?

- Yes, e.g. if using register blocking
- Or if you compute multiple outputs per thread

Can we hide memory latency in a similar manner?

- It is hundreds of cycles…

Memcpy benchmark

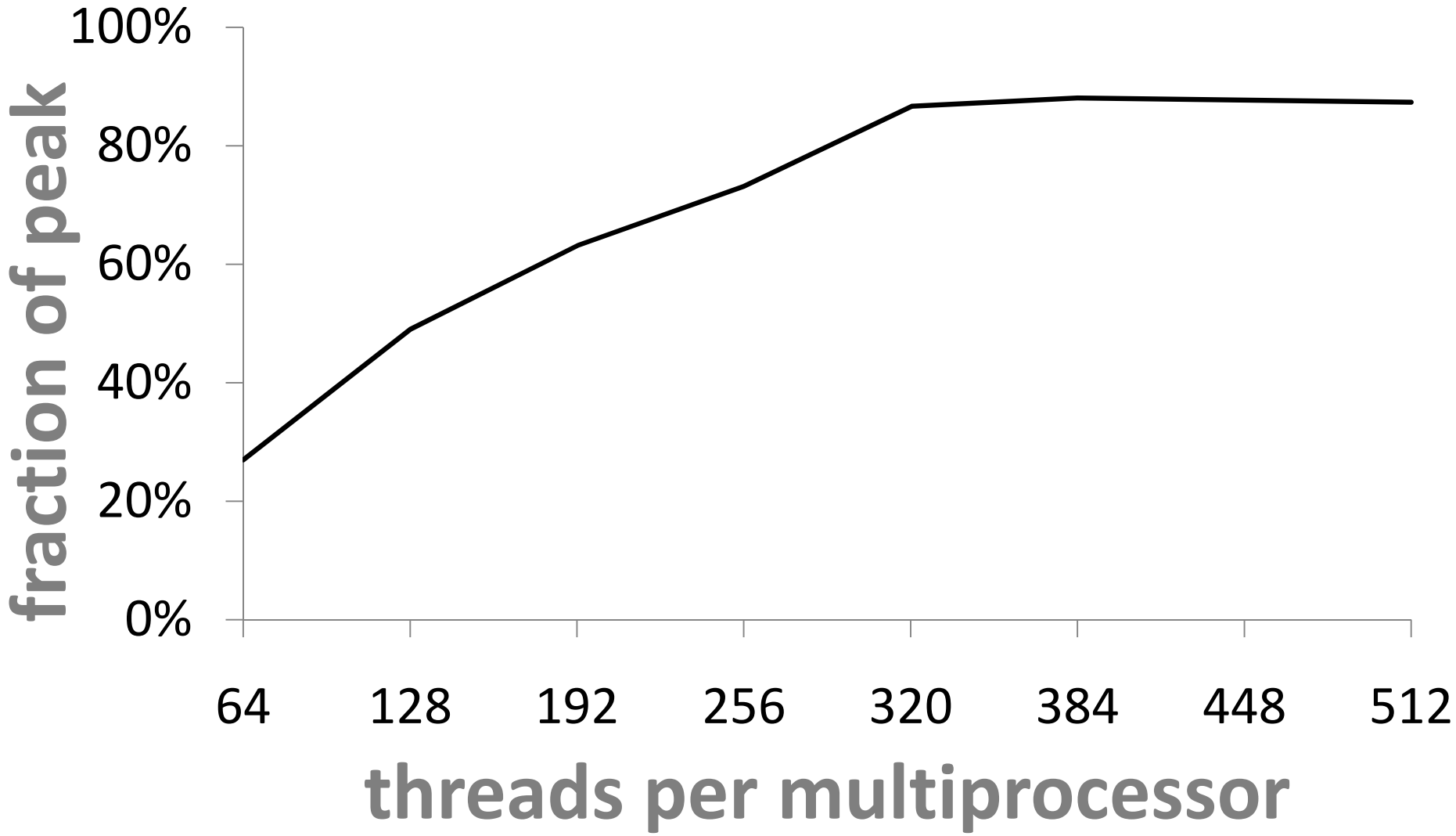
- Copy one 64-bit word per thread:

```
__global__ void memcpy( float2 *dst, float2 *src )
{
    int iblock = blockIdx.x
                + __mul24( blockIdx.y, gridDim.x );
    int index = threadIdx.x
                + __mul24( iblock, blockDim.x );

    float2 a0 = src[index];
    dst[index] = a0;
}
```

- Allocate shared memory dynamically to control occupancy

Memcpy performance



- Need 320 threads to hide memory latency

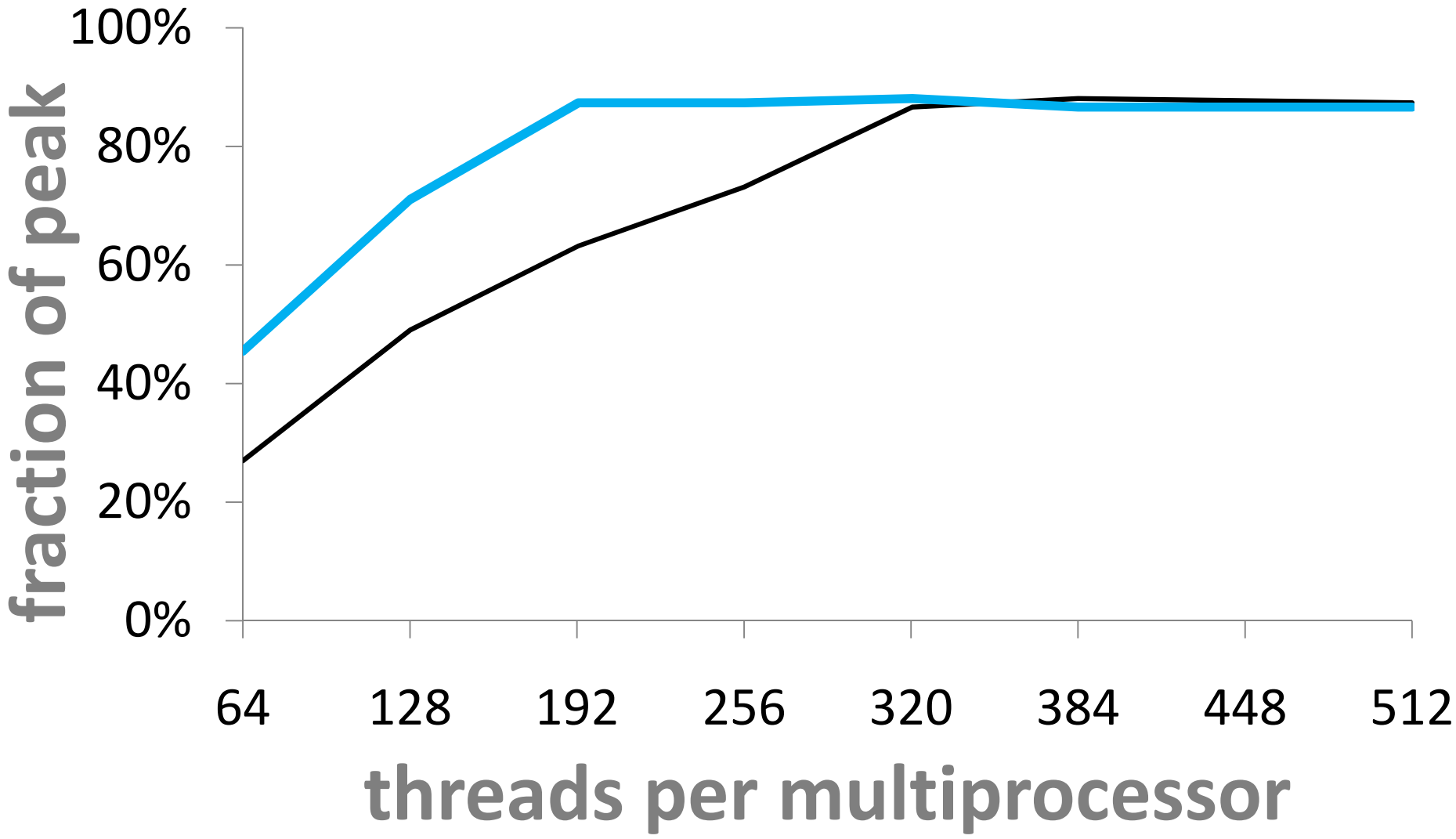
Copy two words per thread

```
__global__ void memcpy( float2 *dst, float2 *src )
{
    int iblock = blockIdx.x
                + __mul24( blockIdx.y, gridDim.x );
    int index = threadIdx.x
               + __mul24( iblock, blockDim.x * 2 );

    float2 a0 = src[index];
    float2 a1 = src[index+blockDim.x];
    dst[index] = a0;
    dst[index+blockDim.x] = a1;
}
```

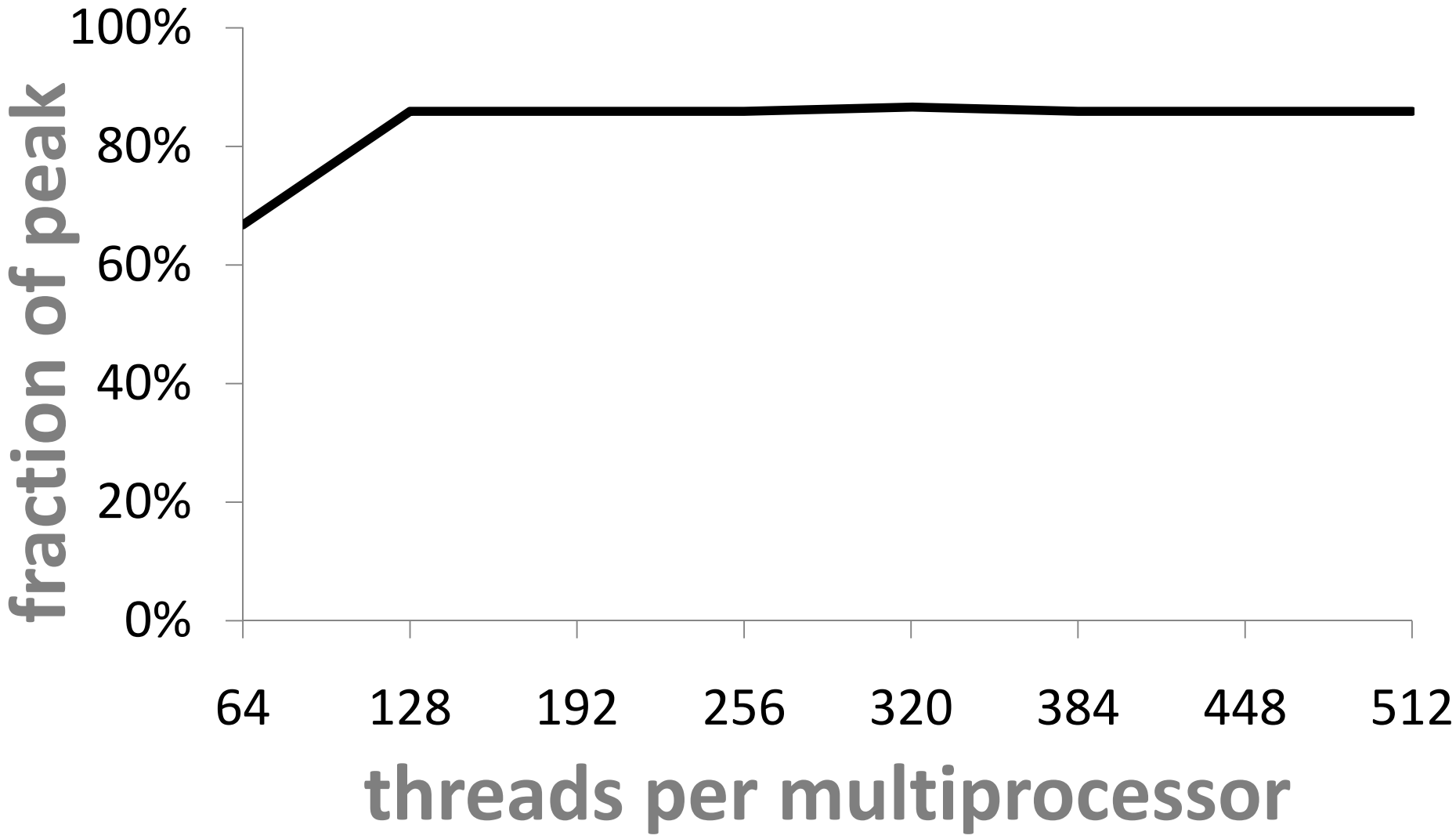
- Load two words but wait for latency once

2 words per thread: performance



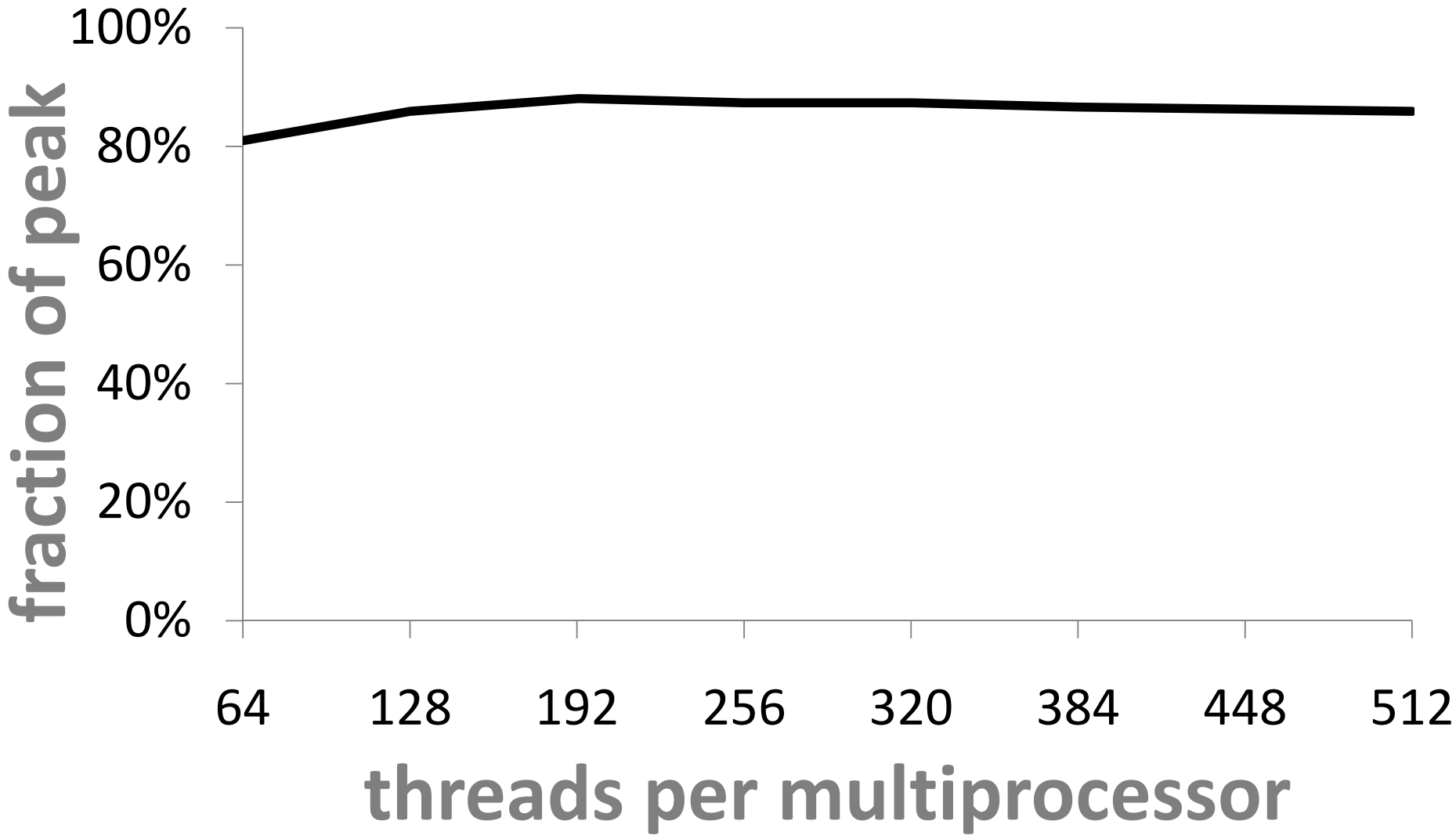
Get same performance at lower occupancy

4 words per thread: performance



Get same performance at even lower occupancy

8 words per thread: performance



Get 80% of memory peak at 6% occupancy

Conclusion so far

Can hide both memory and arithmetic latency using 64 threads

Who cares?

- Low occupancy = many registers per thread
- So, can keep large working set in registers
  - To reduce traffic to other memories
  - E.g. to access shared memory less

Can shared memory be a bottleneck?

	G80/GT200	Fermi
flops/cycle, a*b+c, single precision	16 flops	64 flops
words/cycle, 32-bit, shared memory	8 words	16 words
ratio	2 flops/word	4 flops/word

- Naïve matrix multiply has 1 flop/word
  - Bound by shared memory bandwidth

Shared memory

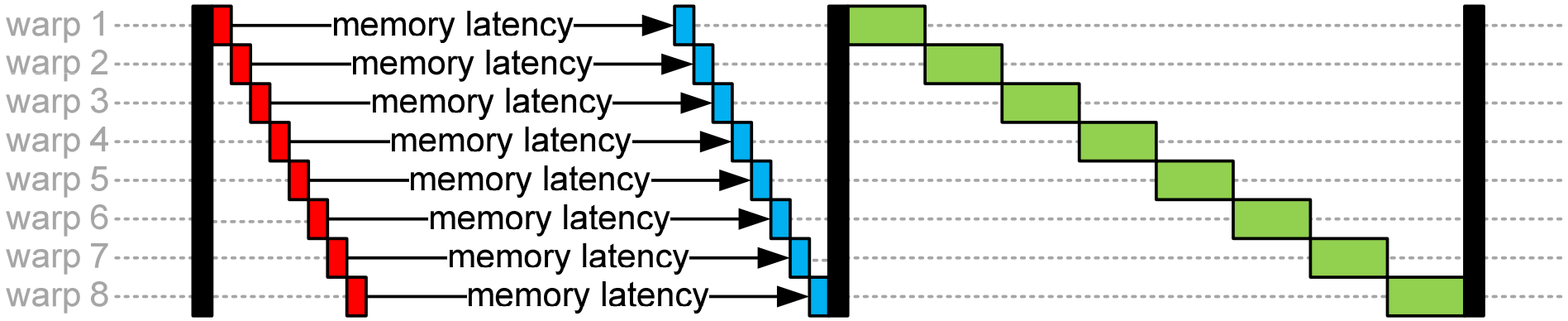
Common computational pattern when using shared memory:

- **Read from global memory**
- **Store to shared memory**
- **Synchronize threads**
- **Compute using shared memory**

Is occupancy important in this case?

Whole thread block stalls at once

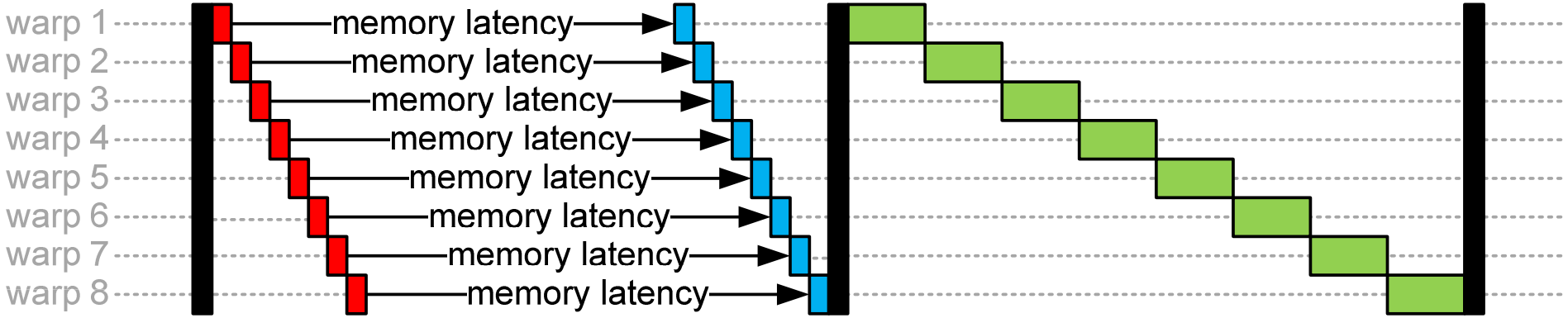
- Due to synchronization, whole thread block stalls at once
  - no matter how many threads in it:



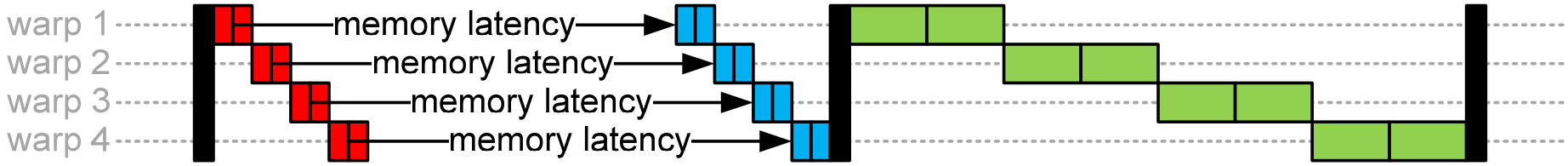
What you need is not many concurrent threads, but many concurrent thread blocks

Smaller blocks hide latency same

In particular, if you do same work:



Using fewer threads, you hide latency sam:e



(This implies doing more work per thread)

- In fact, smaller thread blocks are better!

Small thread blocks are better (I)

- Less threads = more registers per thread

Small thread blocks are better (II)

- There is a limit on total number of threads
- 1024 on GT200
- This is only 2 thread blocks of size 512
  - Enough to hide latency?
- But 8 thread blocks of size 128

Small thread blocks are better (III)

- 2x more work per thread – less than 2x more registers per thread
  - So, less registers per thread block
  - Thus, can run more thread blocks concurrently

- If already enough concurrent thread blocks?
  - Use the extra registers to process larger data blocks

Demo: matrix multiply from SDK

A few simple changes to get 1.4x speedup

The baseline

- Matrix multiply example from SDK 2.3:
- Uses 16x16 matrix blocks
- Computes **one output per thread**
- 16x16 thread blocks
- Well optimized otherwise:
  - All memory accesses are coalesced
  - Data is cached in shared memory

The baseline (CUDA SDK 2.3)

```
float Csub = 0;
for (int a = aBegin, b = bBegin; a <= aEnd;
                     a += aStep, b += bStep) {

    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    __syncthreads();

    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);
    __syncthreads();
}

int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

The original code (comments not included)

The baseline performance

- The baseline runs at 200 Gflop/s
  - For 1008x1008 matrices
  - Measure only GPU time (no PCIe transfers)

- Uses only 14 registers per thread
- Sustains 100% occupancy

- What can be better?

Step I: do 2 outputs per thread

- In the new code I run 16x8 thread blocks
  - Grid size is same
- Half of the threads is eliminated
- Each remaining thread does 2x more work

Two outputs per thread (I)

```
float Csub[2] = {0,0};
for (int a = aBegin, b = bBegin; a <= aEnd;
                      a += aStep, b += bStep) {

    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+8, tx) = A[a + wA * (ty+8) + tx];
    BS(ty+8, tx) = B[b + wB * (ty+8) + tx];
    __syncthreads();
```

Changes are marked in **red**

- Now have 2 outputs (`Csub`)
- Each thread fetches 2 elements of A and B

Two outputs per thread (II)

```
#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub[0] += AS(ty, k) * BS(k, tx);
        Csub[1] += AS(ty+8, k) * BS(k, tx);
    }
    __syncthreads();
}

int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+8) + tx] = Csub[1];
```

- 2x more flops per thread
- Store 2 outputs in the end
- Now compiler needs a hint to unroll the loop

2 outputs/thread: performance

- New performance: 253 Gflop/s
  - 27% speedup!

- Uses only 18 registers per thread
  - 4 more
- Sustains 75% occupancy
  - 25% less

Why the speedup?

Data fetched from shared memory is reused:

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
    Csub[0] += AS(ty, k) * BS(k, tx);
    Csub[1] += AS(ty+8, k) * BS(k, tx);
}
```

Reuse was not possible before

- The data was fetched in different threads
- Can't access registers of another thread

Result: **reduced shared memory traffic**

- Why not apply same technique again?
- Now use 16x4 thread blocks
- 4 outputs per thread

Four outputs per thread (I)

```
float Csub[4] = {0,0,0,0};
for (int a = aBegin, b = bBegin; a <= aEnd;
                    a += aStep, b += bStep) {

    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    AS(ty+4, tx) = A[a + wA * (ty+4) + tx];
    BS(ty+4, tx) = B[b + wB * (ty+4) + tx];
    AS(ty+8, tx) = A[a + wA * (ty+8) + tx];
    BS(ty+8, tx) = B[b + wB * (ty+8) + tx];
    AS(ty+12,tx) = A[a + wA * (ty+12)+ tx];
    BS(ty+12,tx) = B[b + wB * (ty+12)+ tx]
__syncthreads();
```

Same idea…

Four outputs per thread (II)

```
#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub[0] += AS(ty, k) * BS(k, tx);
        Csub[1] += AS(ty+4, k) * BS(k, tx);
        Csub[2] += AS(ty+8, k) * BS(k, tx);
        Csub[3] += AS(ty+12,k) * BS(k, tx);
    }
    __syncthreads();
}

int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub[0];
C[c + wB * (ty+4) + tx] = Csub[1];
C[c + wB * (ty+8) + tx] = Csub[2];
C[c + wB * (ty+12)+ tx] = Csub[3];
```

Get even more reuse now…

Unexpected slowdown

New performance is only 235 Gflop/s

- 8% slowdown

What's the problem?

Use decuda to figure it out

decuda: disassembler of GPU binaries

- second most useful tool after compiler
- many thanks to Wladimir J. van der Laan for developing it!

Use decuda to figure it out

Many operations on pointers to shared memory:

```
movsh.b32 $ofs4, $r29, 0x00000000
mad.rn.f32 $r17, s[$ofs4+0x000c], $r4, $r17
mad.rn.f32 $r10, s[$ofs2+0x000c], $r4, $r10
mad.rn.f32 $r4, s[$ofs3+0x000c], $r4, $r18
movsh.b32 $ofs4, $r9, 0x00000002
add.b32 $ofs4, $ofs4, 0x000002a4
mov.b32 $r18, $ofs4
mad.rn.f32 $r16, s[$ofs1+0x0010], $r3, $r16
movsh.b32 $ofs4, $r29, 0x00000000
mad.rn.f32 $r17, s[$ofs4+0x0010], $r3, $r17
mad.rn.f32 $r10, s[$ofs2+0x0010], $r3, $r10
mad.rn.f32 $r30, s[$ofs3+0x0010], $r3, $r4
movsh.b32 $ofs4, $r18, 0x00000000
```

Workaround: transpose blocks

- The problem is poor locality in sequential access to shared memory
  - Need to reload pointers too often
- Solution:
  - Use transposed layout in shared memory
    - Change all AS(**yy**,**xx**) to AS(**xx**,**yy**), same with BS
  - Pad the arrays
    - Define as As[BLOCK_SIZE][BLOCK_SIZE**+1**]

New 4 outputs/thread: performance

- New performance: 284 Gflop/s

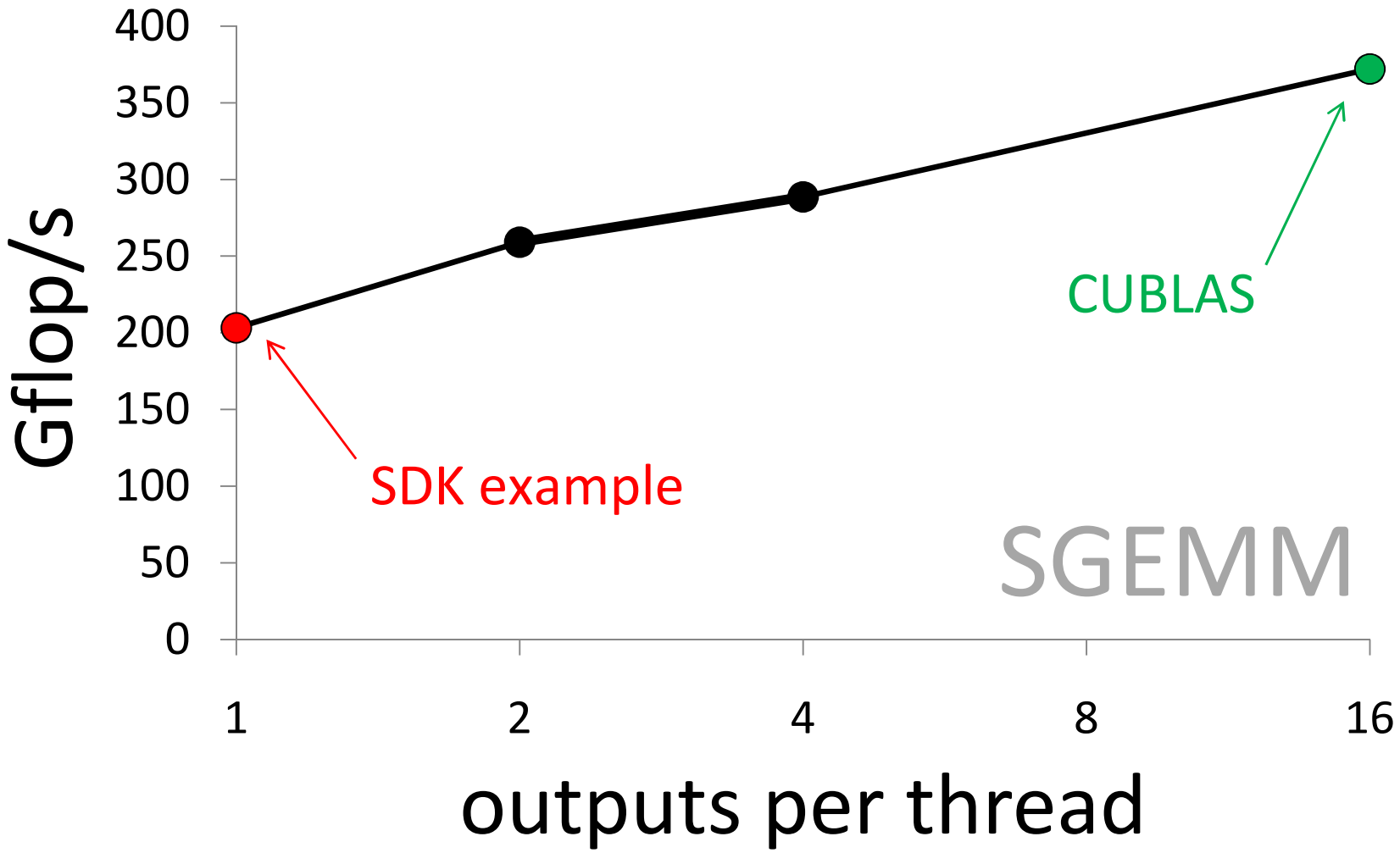
- Uses only 29 registers per thread
  - 11 more
- Sustains 37.5% occupancy
  - 2x lower

Optimization summary

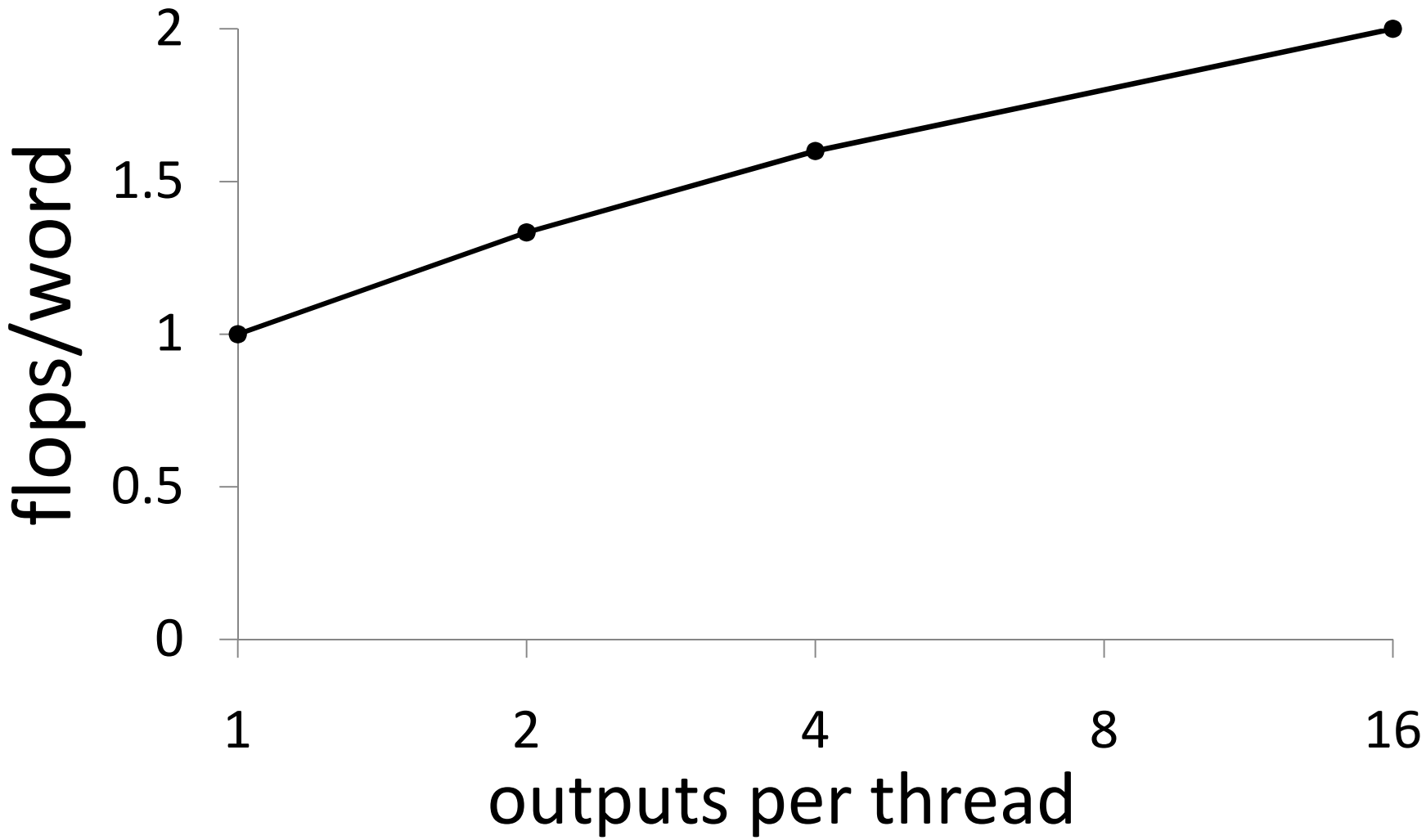
Outputs/thread	1	2	4
Registers/thread	14	18	29
Occupancy	**100%**	**75%**	**37.5%**
Registers/block	3584	2304	1856
Blocks/SM	4	6	6
Gflop/s	**200**	**253**	**284**

Optimize further?

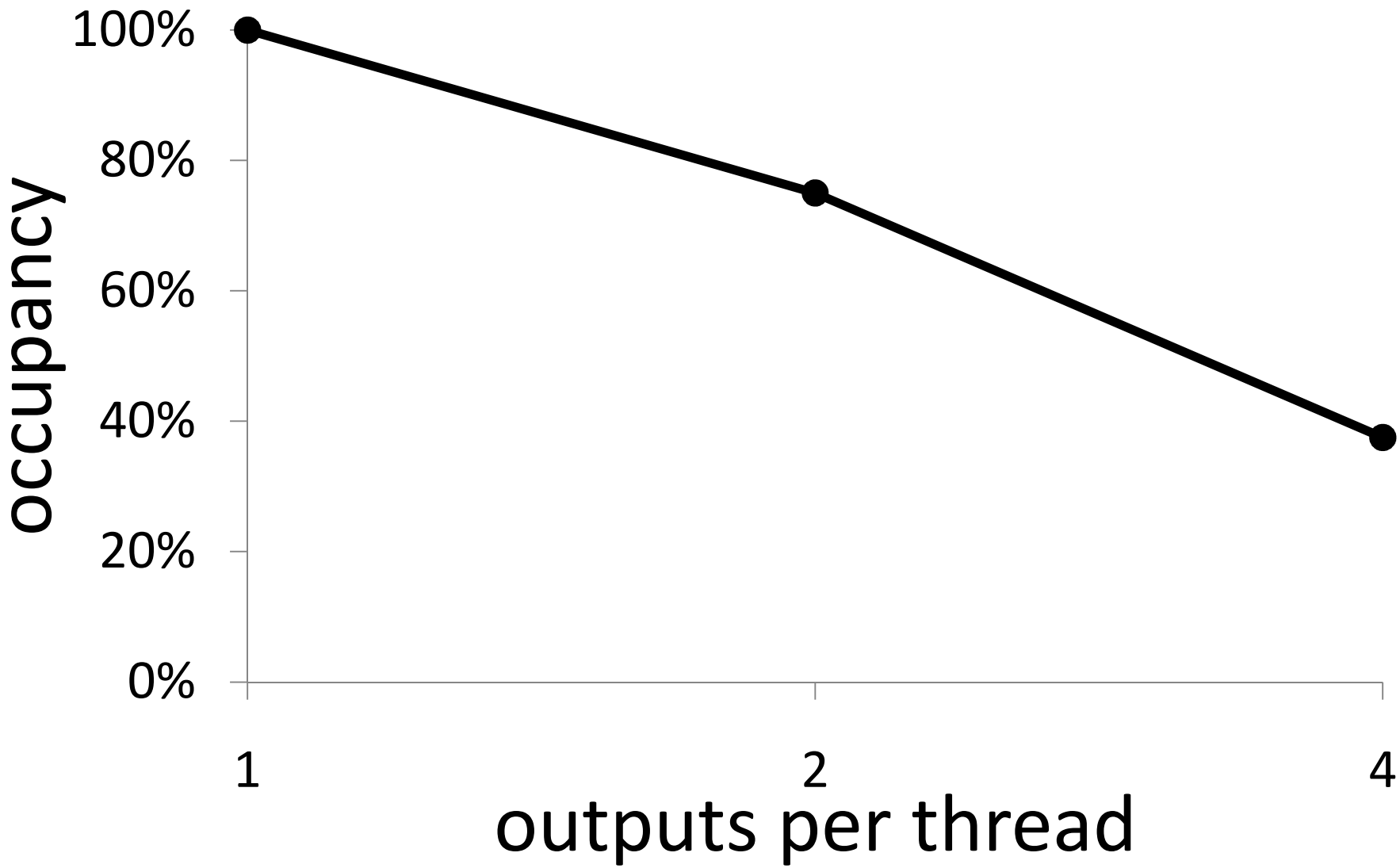
At this rate we'll get to CUBLAS soon:



Speedup is due to less shared memory traffic



Run faster at lower occupancy



Conclusion

- If you optimize for perfect occupancy, you may lose performance opportunities
- Consider hiding latency by computing multiple outputs per thread
- Use registers instead of shared memory whenever possible