Better Performance at Lower Occupancy

Vasily Volkov
UC Berkeley

September 22, 2010

Prologue

It is common to recommend:

- running more threads per multiprocessor
- running more threads per thread block

Motivation: this is the only way to hide latencies

- But…

Faster codes run at lower occupancy:

Multiplication of two large matrices, single precision (SGEMM):

	CUBLAS 1.1	CUBLAS 2.0	
Threads per block	512	64	8x smaller thread blocks
Occupancy (G80)	67%	33%	2x lower occupancy
Performance (G80)	128 Gflop/s	204 Gflop/s	1.6x higher performance

Batch of 1024-point complex-to-complex FFTs, single precision:

	CUFFT 2.2	CUFFT 2.3	
Threads per block	256	64	4x smaller thread blocks
Occupancy (G80)	33%	17%	2x lower occupancy
Performance (G80)	45 Gflop/s	93 Gflop/s	2x higher performance

Maximizing occupancy, you may lose performance

Two common fallacies:

– multithreading is the only way to hide latency on GPU

– shared memory is as fast as registers

This talk

I. Hide arithmetic latency using fewer threads
II. Hide memory latency using fewer threads
III. Run faster by using fewer threads
IV. Case study: matrix multiply
V. Case study: FFT

Part I:

Hide arithmetic latency using fewer threads

Arithmetic latency

Latency: time required to perform an operation

- ≈20 cycles for arithmetic; 400+ cycles for memory
- Can't start a *dependent* operation for this time
- Can hide it by overlapping with other operations

```
x = a + b;// takes ≈20 cycles to execute
y = a + c;// independent, can start anytime
(stall)
z = x + d;// dependent, must wait for completion
```

Arithmetic throughput

Latency is often confused with throughput

- E.g. “arithmetic is 100x faster than memory – costs 4 cycles per warp (G80), whence memory operation costs 400 cycles”
  - One is rate, another is time

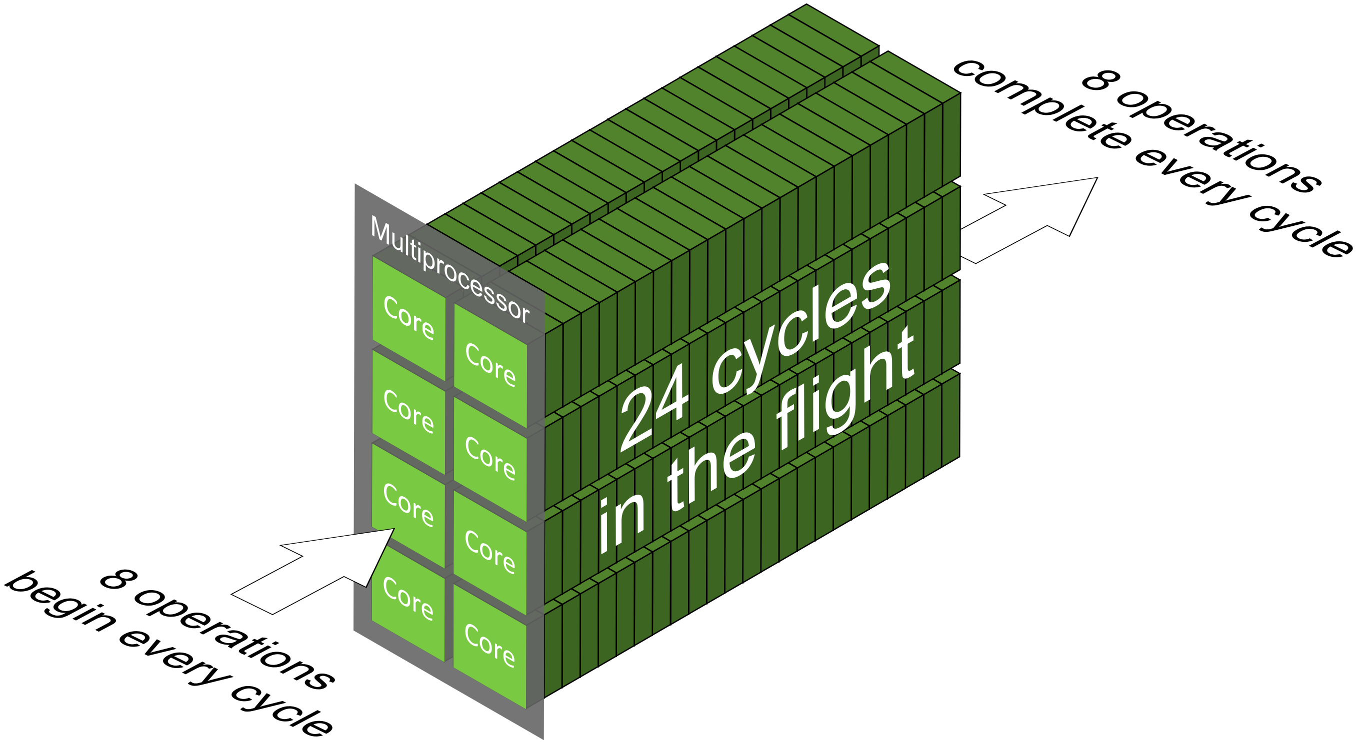
Throughput: how many operations complete per cycle

- Arithmetic: 1.3 Tflop/s = 480 ops/cycle (op=multiply-add)
- Memory: 177 GB/s ≈ 32 ops/cycle (op=32-bit load)

Hide latency = do other operations when waiting for latency

- Will run faster
- But not faster than the peak
- How to get the peak?

Use Little's law



Needed parallelism = **Latency** x **Throughput**

Arithmetic parallelism in numbers

GPU model	Latency (cycles)	Throughput (cores/SM)	Parallelism (operations/SM)
G80-GT200	≈24	8	≈192
GF100	≈18	32	≈576
GF104	≈18	48	≈864

(latency varies between different types of ops)

Can’t get 100% throughput with less parallelism

– Not enough operations in the flight = idle cycles

Thread-level parallelism (TLP)

It is usually recommended to use threads to supply the needed parallelism, e.g. 192 threads per SM on G80:

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

4 independent operations

Instruction-level parallelism (ILP)

But you can also use parallelism among instructions in a single thread:

thread

instructions

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

You can use both ILP and TLP on GPU

This applies to all CUDA-capable GPUs. E.g. on G80:

- Get ≈100% peak with 25% occupancy if no ILP
- Or with 8% occupancy, if 3 operations from each thread can be concurrently processed

On GF104 you *must* use ILP to get >66% of peak!

- 48 cores/SM, one instruction is broadcast across 16 cores
- So, must issue 3 instructions per cycle
- But have only 2 warp schedulers
- Instead, it can issue 2 instructions per warp in the same cycle

Let’s check it experimentally

Do many arithmetic instructions with no ILP:

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

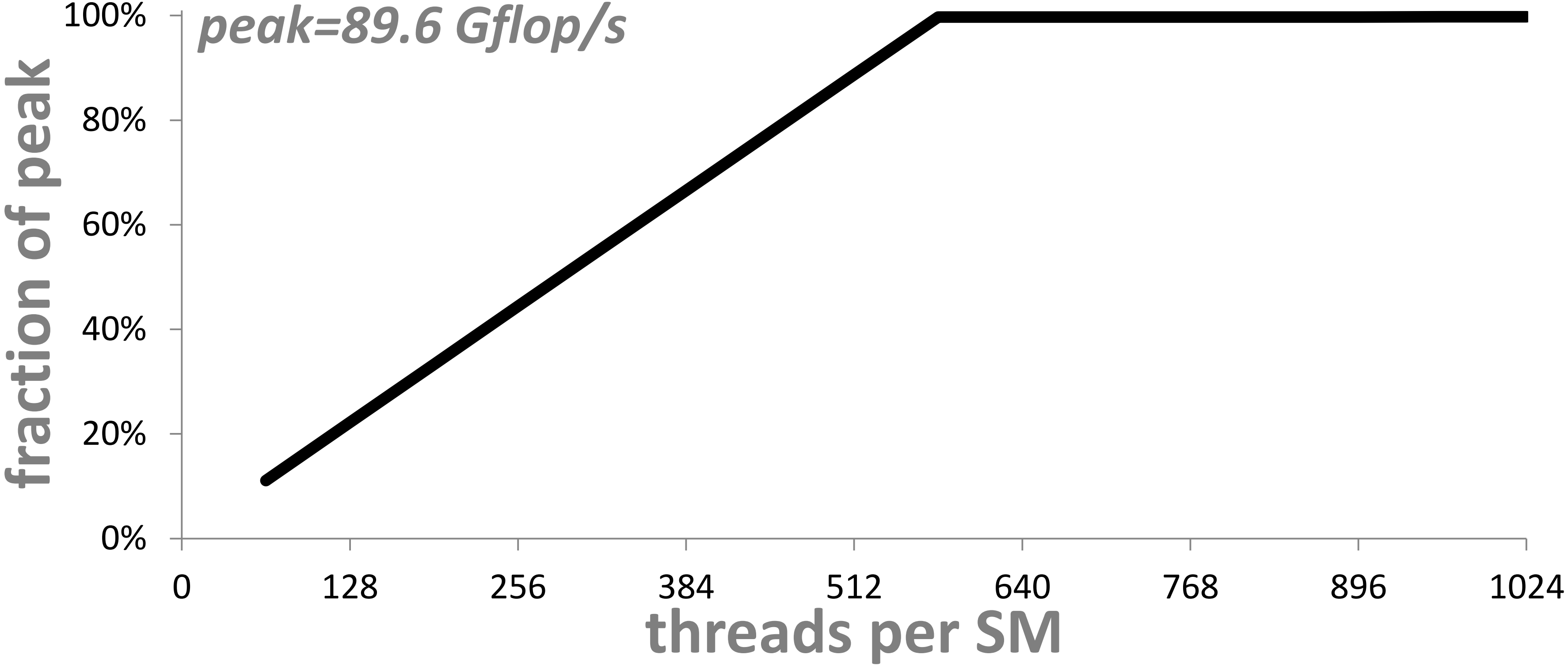
Choose large **`N_ITERATIONS`** and suitable **`UNROLL`**

Ensure **`a`**, **`b`** and **`c`** are in registers and **`a`** is used later

Run 1 block (use 1 SM), vary block size

– See what fraction of peak (1.3TFLOPS/15) we get

Experimental result (GTX480)



No ILP: need 576 threads to get 100% utilization

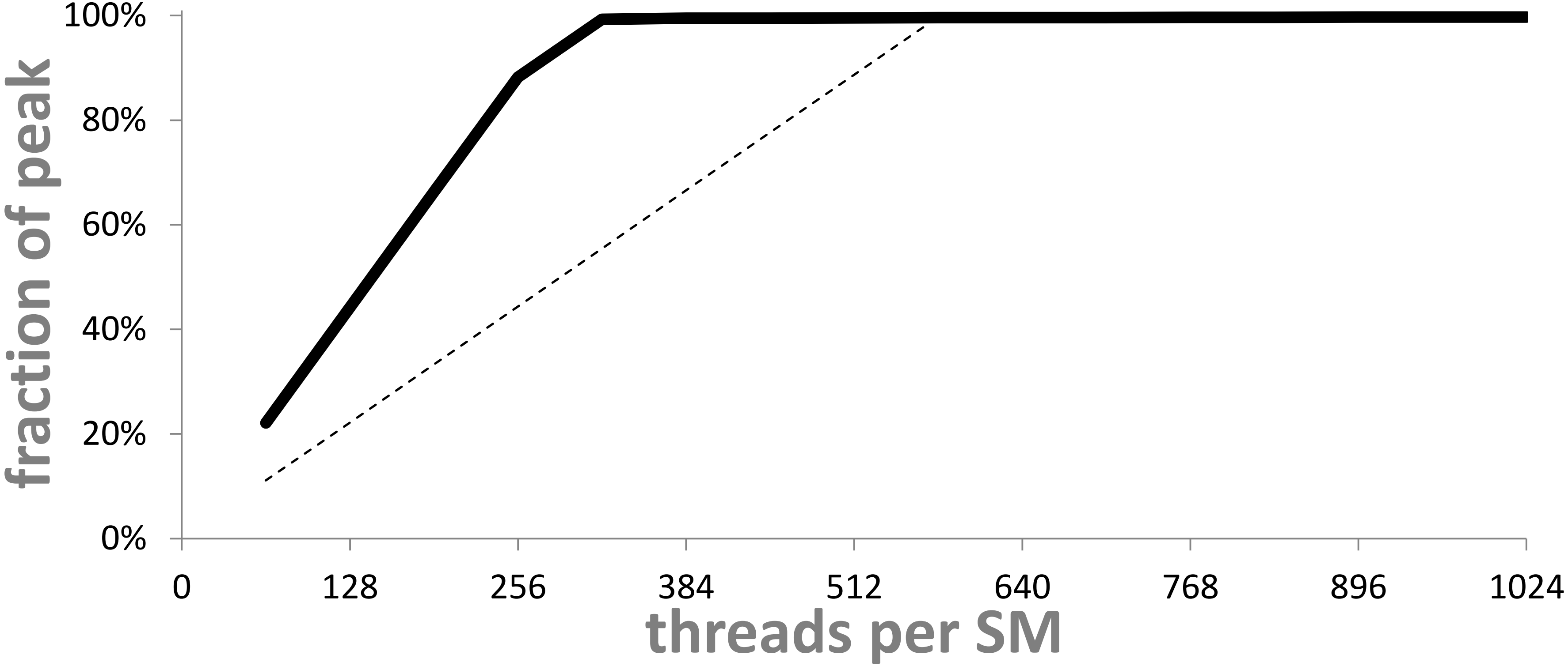
Introduce instruction-level parallelism

Try ILP=2: two independent instruction per thread

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

If multithreading is the only way to hide latency on GPU, we’ve got to get the same performance

GPUs can hide latency using ILP



ILP=2: need 320 threads to get 100% utilization

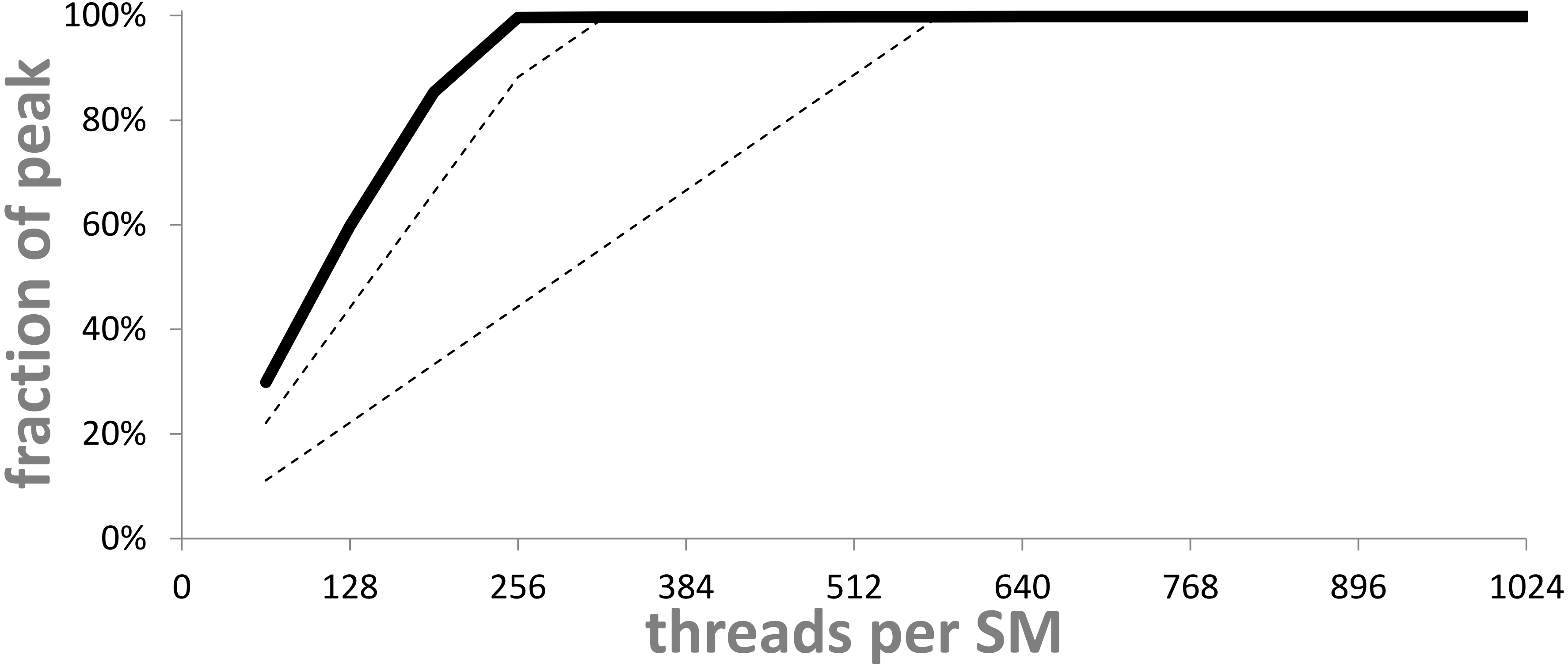
Add more instruction-level parallelism

ILP=3: triples of independent instructions

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

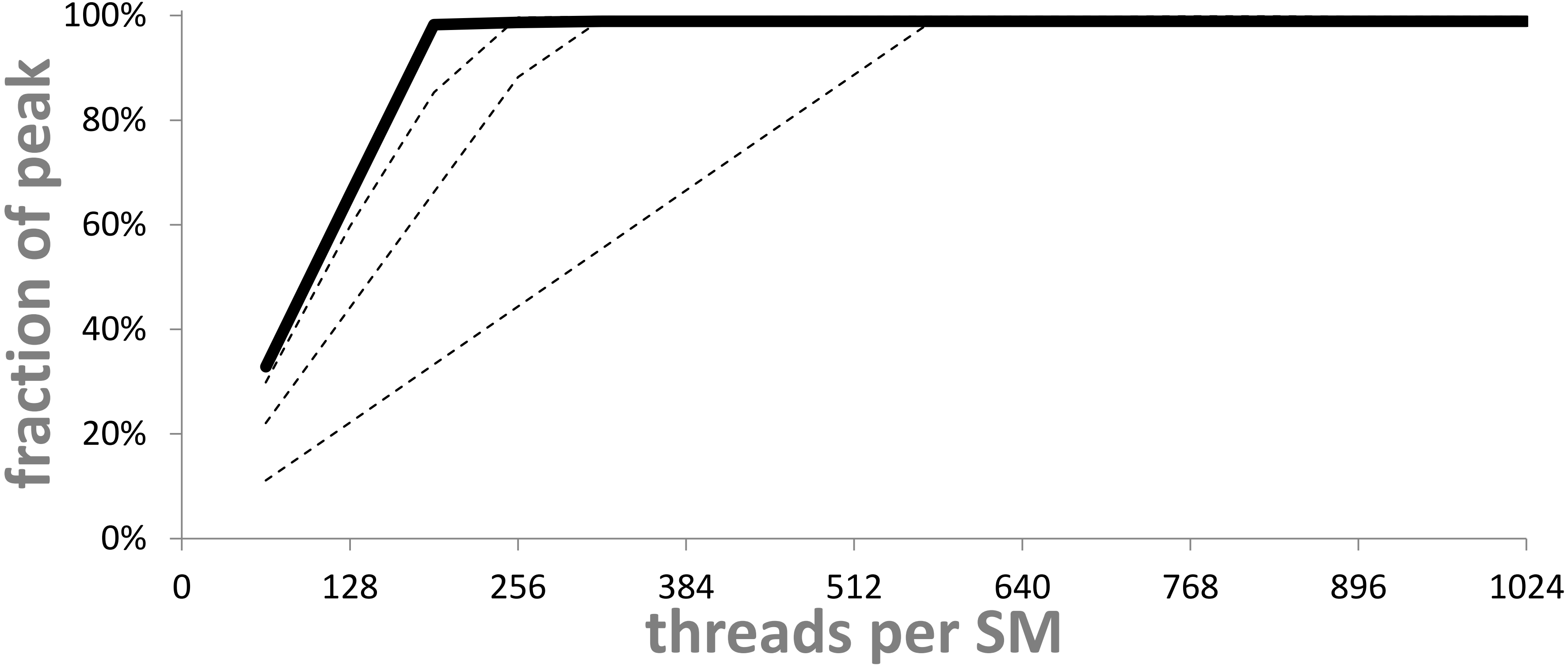
How far can we push it?

Have more ILP – need fewer threads



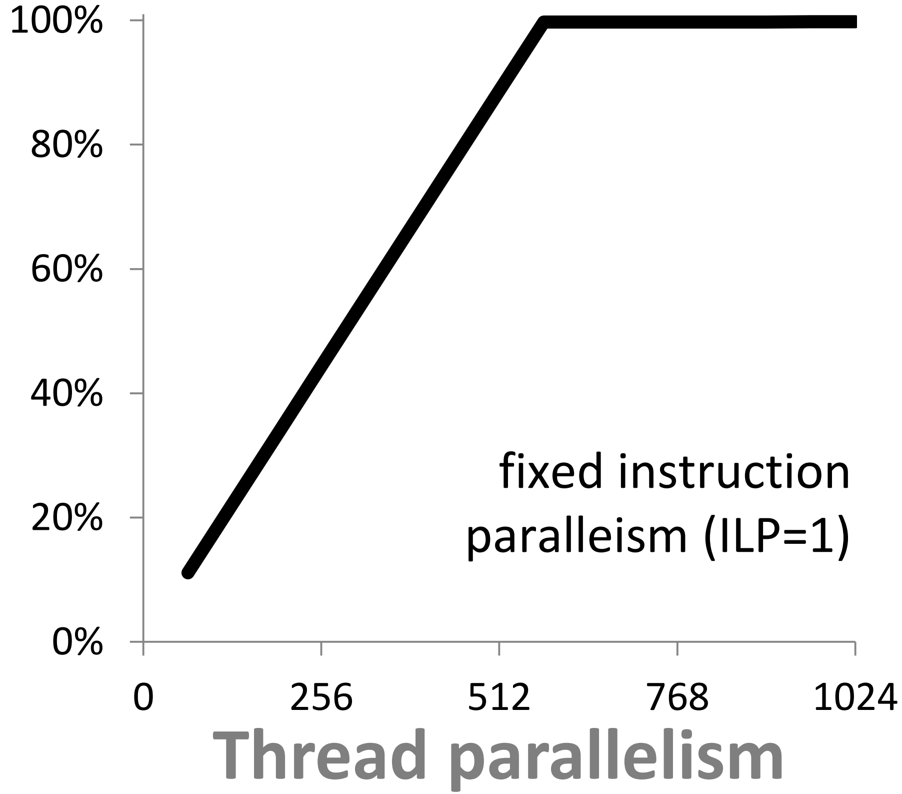
ILP=3: need 256 threads to get 100% utilization

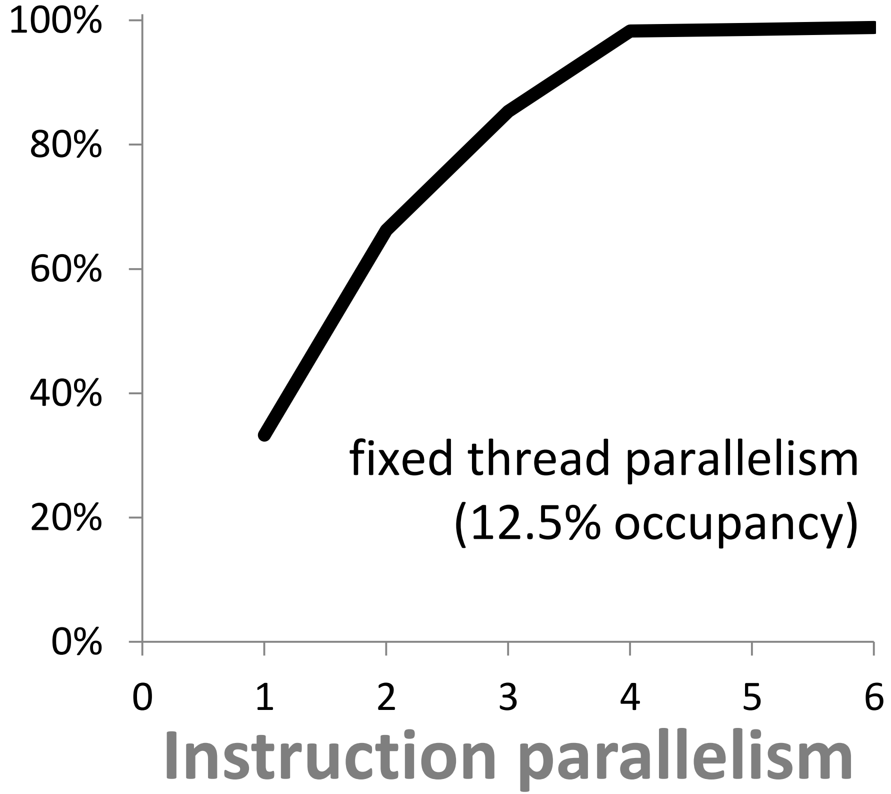
Unfortunately, doesn't scale past ILP=4



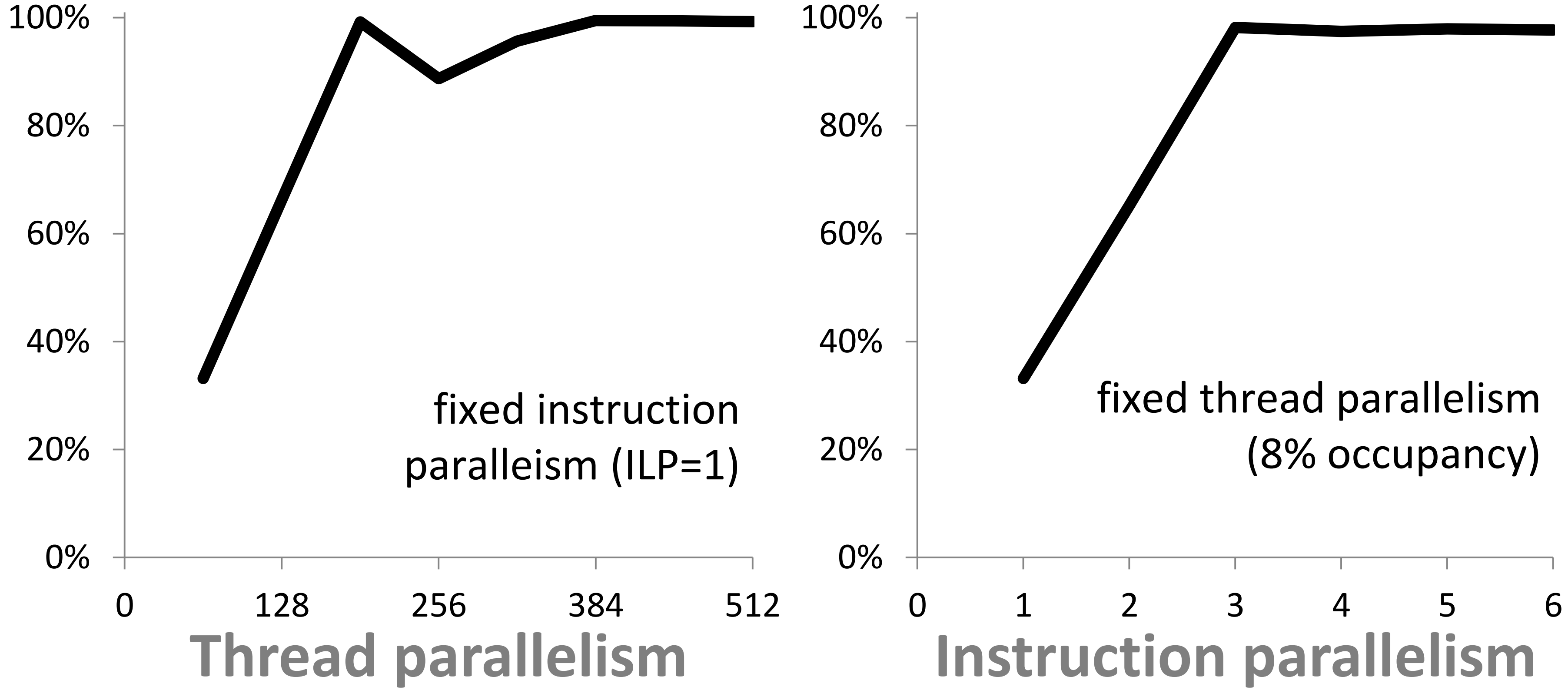
ILP=4: need 192 threads to get 100% utilization

Summary: can hide latency either way





Applies to other GPUs too, e.g. to G80:



Fallacy:

Increasing occupancy is the only way to improve latency hiding

– *No, increasing ILP is another way.*

Fallacy:

Occupancy is a metric of utilization

– *No, it's only one of the contributing factors.*

Fallacy:
"To hide arithmetic latency completely, multiprocessors should be running at least 192 threads on devices of compute capability 1.x (…) or, on devices of compute capability 2.0, as many as 384 threads" (**CUDA Best Practices Guide**)

– *No, it is doable with 64 threads per SM on G80-GT200 and with 192 threads on GF100.*

Part II:

Hide memory latency using fewer threads

Hiding memory latency

Apply same formula but for memory operations:

Needed parallelism = **Latency** x **Throughput**

	Latency	Throughput	Parallelism
Arithmetic	≈18 cycles	32 ops/SM/cycle	576 ops/SM
Memory	< 800 cycles (?)	< 177 GB/s	< 100 KB

So, hide memory latency = keep 100 KB in the flight

– Less if kernel is compute bound (needs fewer GB/s)

How many threads is 100 KB?

Again, there are multiple ways to hide latency

- Use multithreading to get 100KB in the flight
- Use instruction parallelism (more fetches per thread)
- Use bit-level parallelism (use 64/128-bit fetches)

Do more work per thread – need fewer threads

- Fetch 4B/thread – need 25 000 threads
- Fetch 100 B/thread – need 1 000 threads

Empirical validation

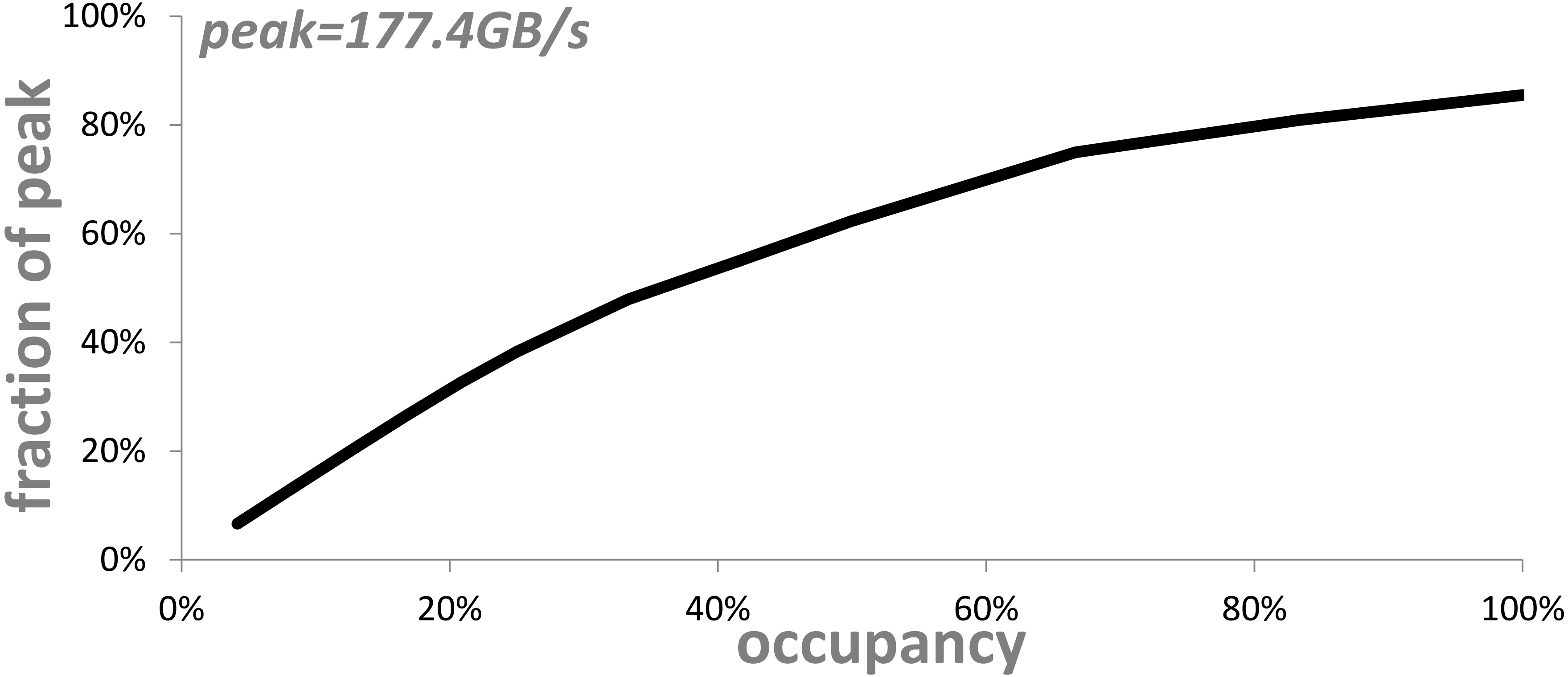
Copy one float per thread:

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

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

Run many blocks, allocate shared memory dynamically to control occupancy

Copying 1 float per thread (GTX480)



Must maximize occupancy to hide latency?

Do more parallel work per thread

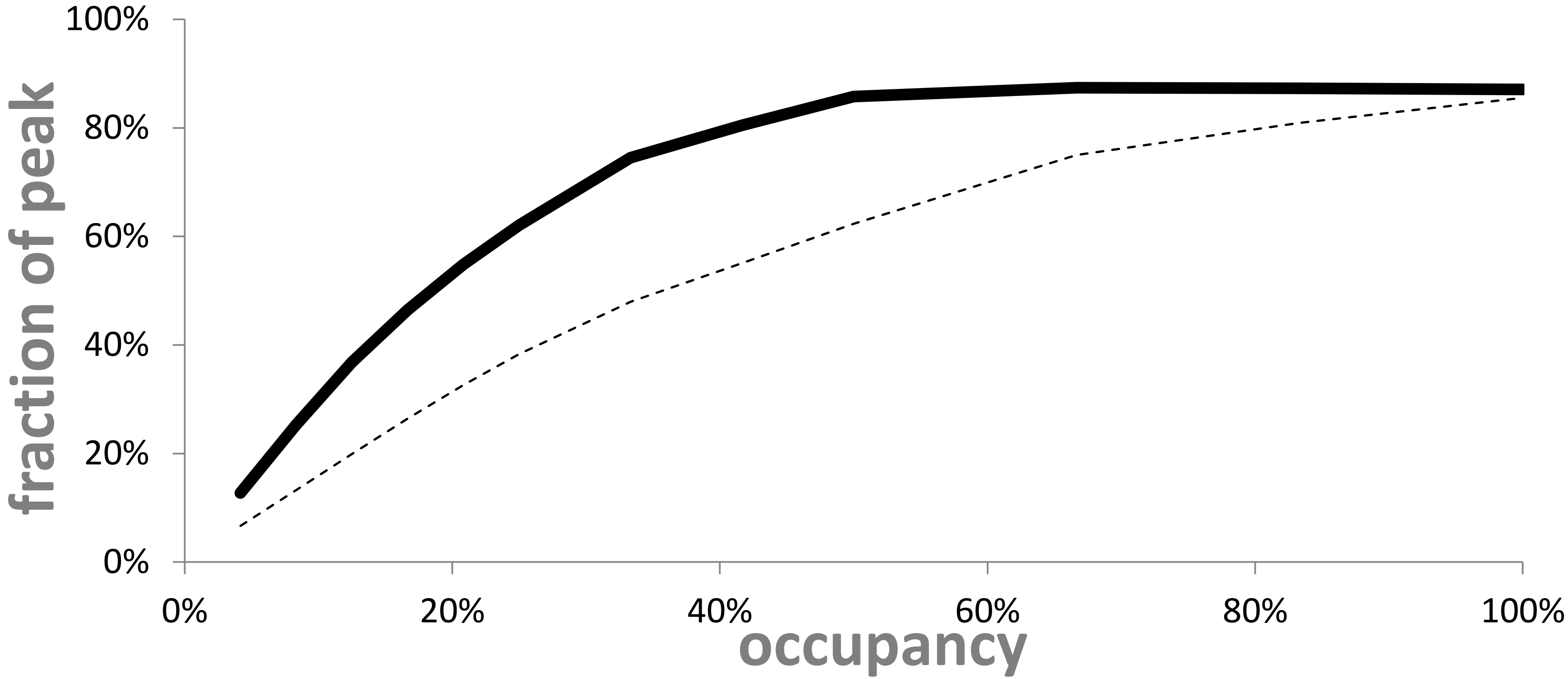
```
__global__ void memcpy( float *dst, float *src )
{
  int iblock= blockIdx.x + blockIdx.y * gridDim.x;
  int index = threadIdx.x + 2 * iblock * blockDim.x;

  float a0 = src[index];
  //no latency stall
  float a1 = src[index+blockDim.x];
  //stall
  dst[index] = a0;
  dst[index+blockDim.x] = a1;
}
```

Note, **threads don't stall on memory access**

- Only on data dependency

Copying 2 float values per thread



Can get away with lower occupancy now

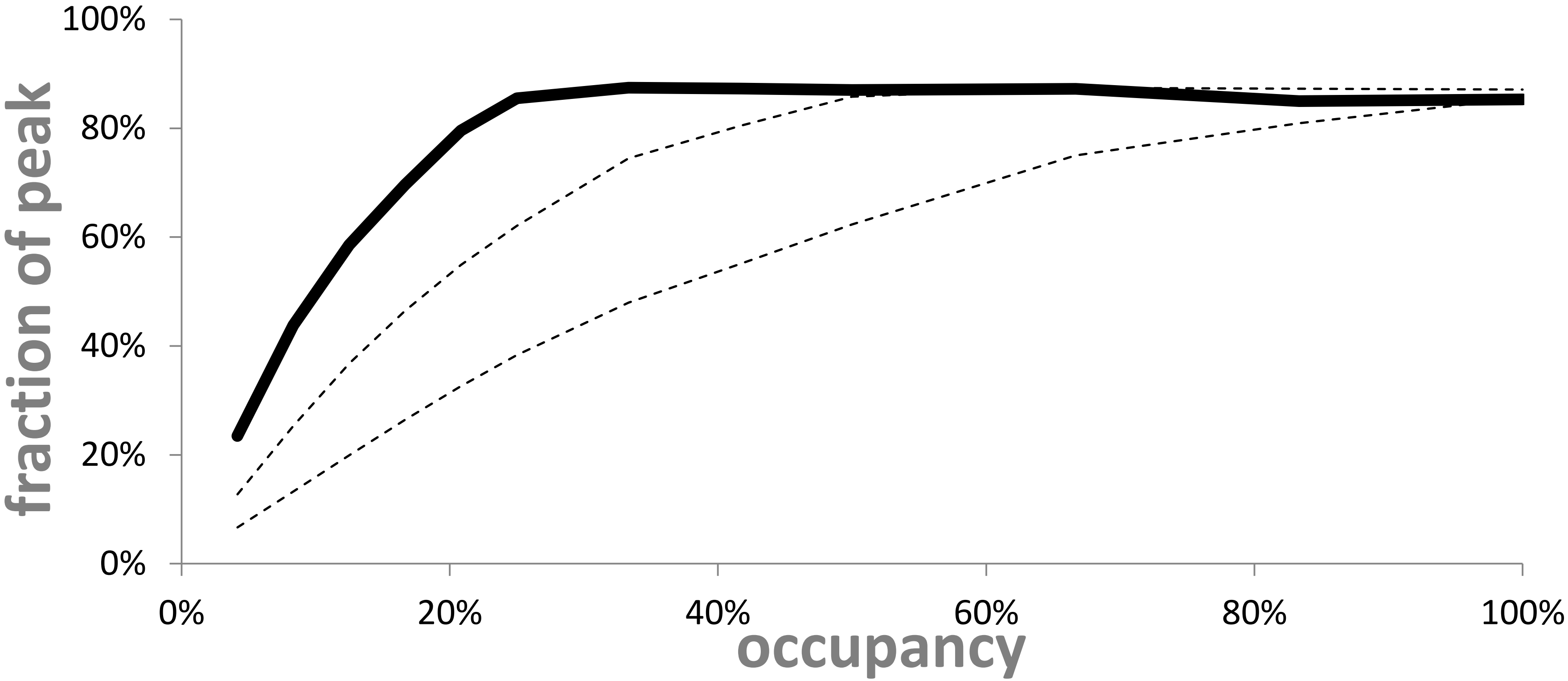
Do more parallel work per thread

```
__global__ void memcpy( float *dst, float *src )
{
  int iblock = blockIdx.x + blockIdx.y * gridDim.x;
  int index  = threadIdx.x + 4 * iblock * blockDim.x;

  float a[4];//allocated in registers
  for(int i=0;i<4;i++) a[i]=src[index+i*blockDim.x];
  for(int i=0;i<4;i++) dst[index+i*blockDim.x]=a[i];
}
```

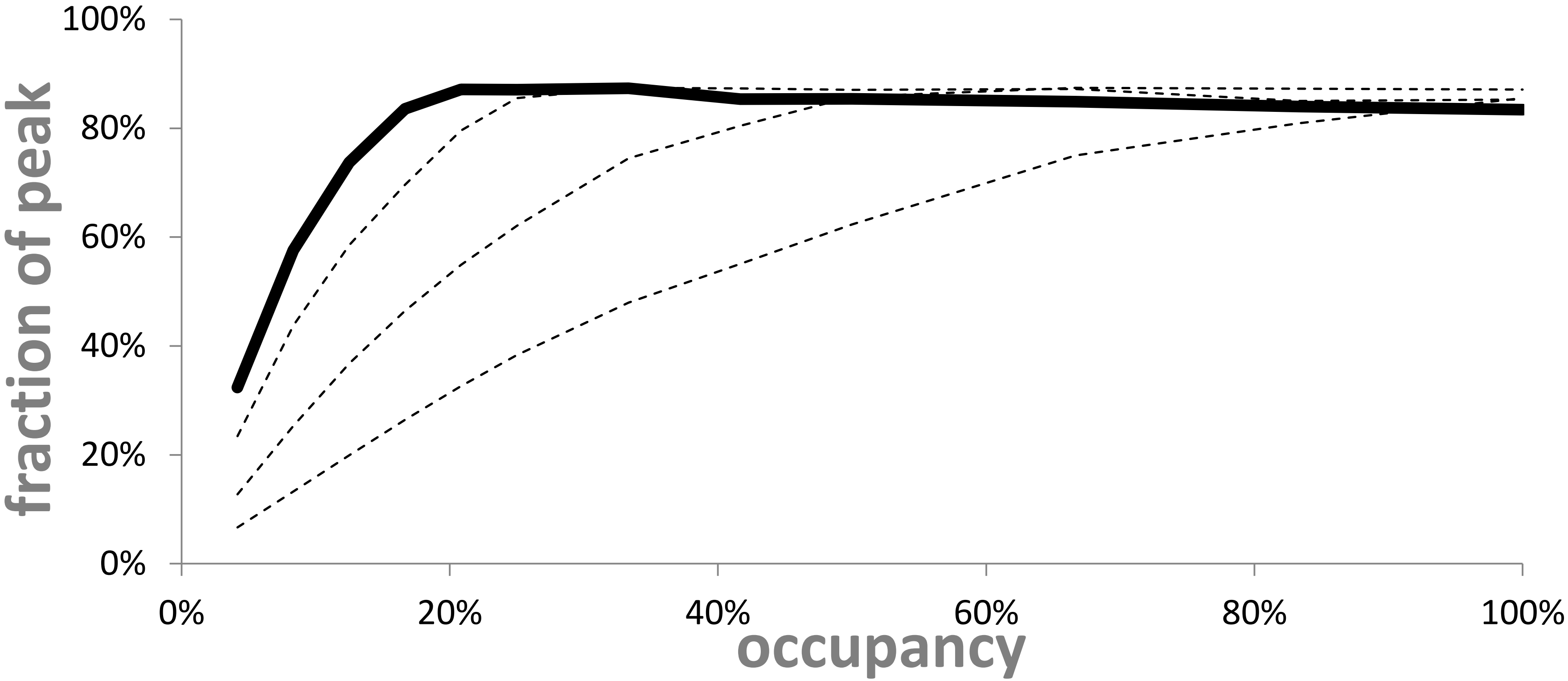
Note, **local arrays are allocated in registers if possible**

Copying 4 float values per thread

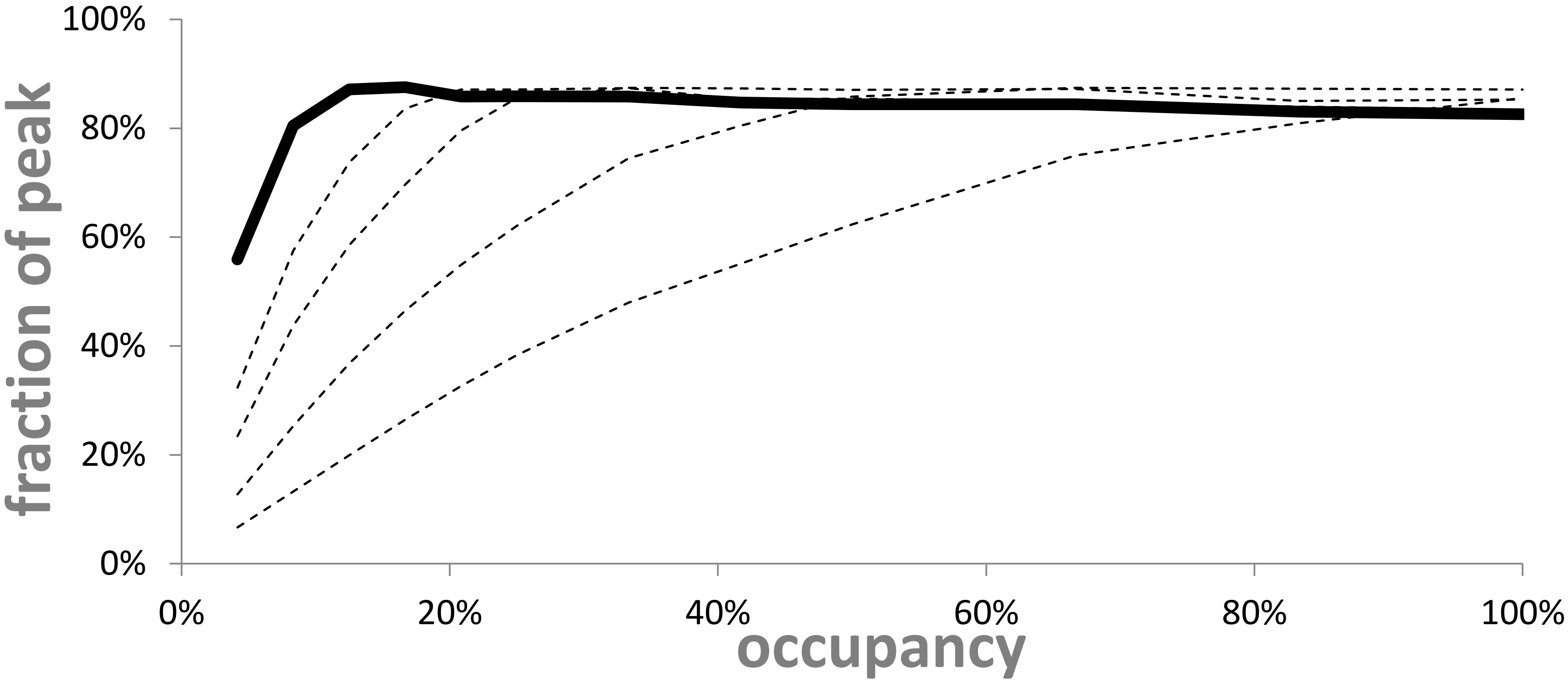


Mere 25% occupancy is sufficient. How far we can go?

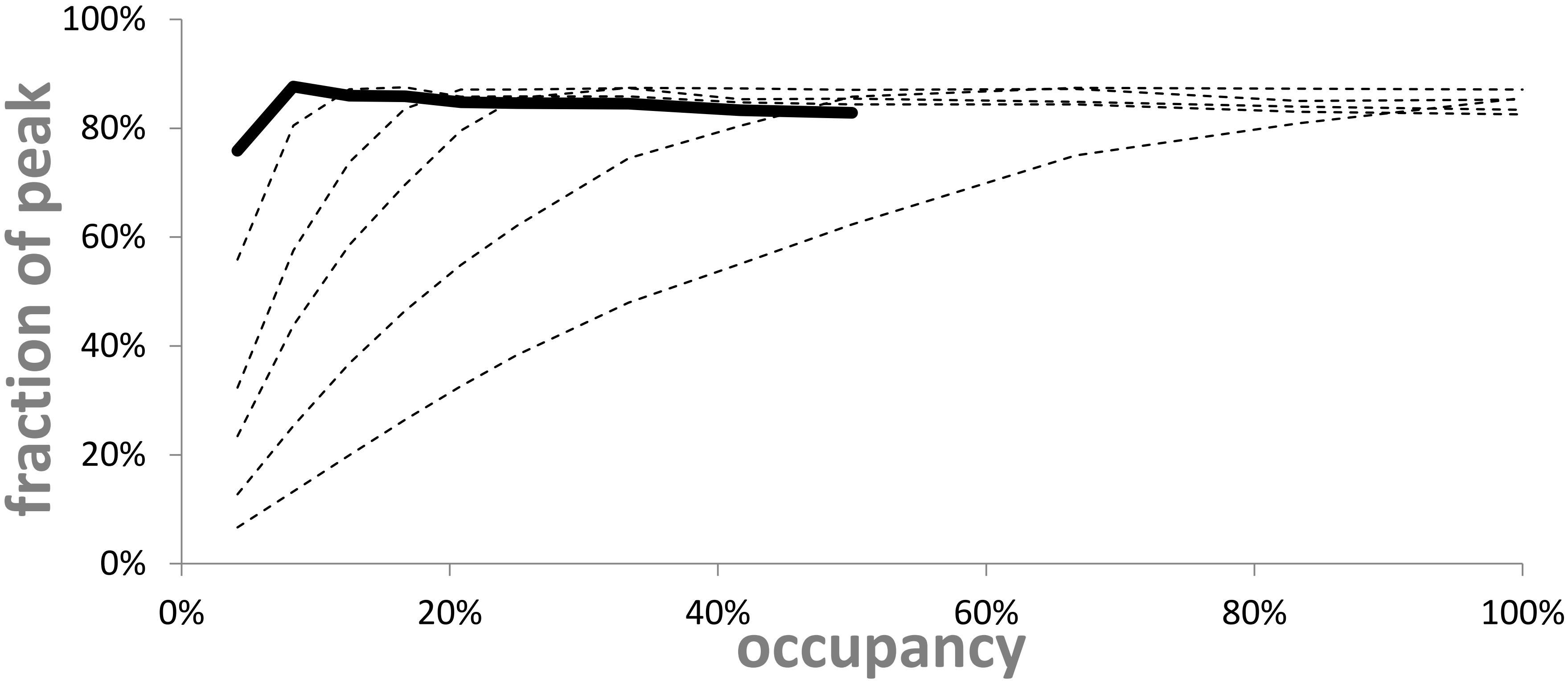
Copying 8 float values per thread



Copying 8 float2 values per thread

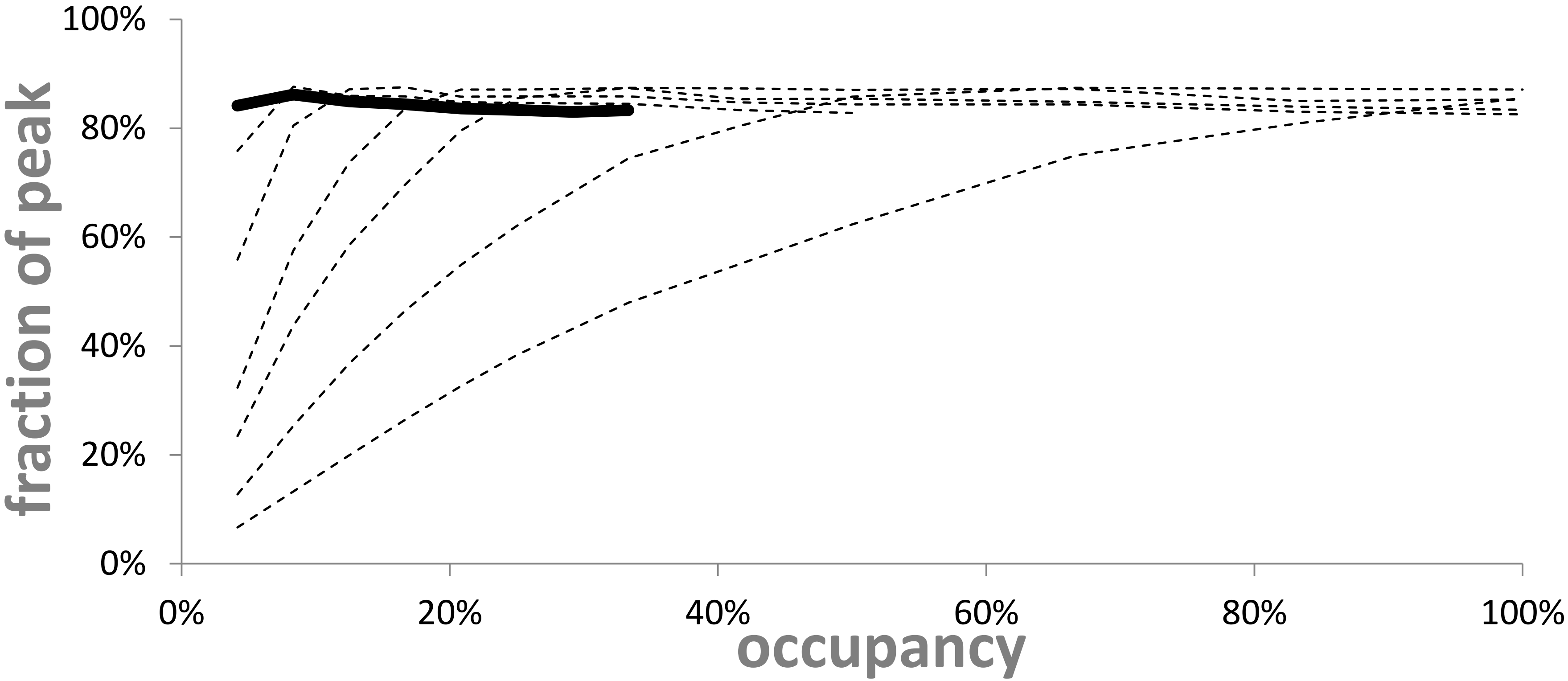


Copying 8 float4 values per thread



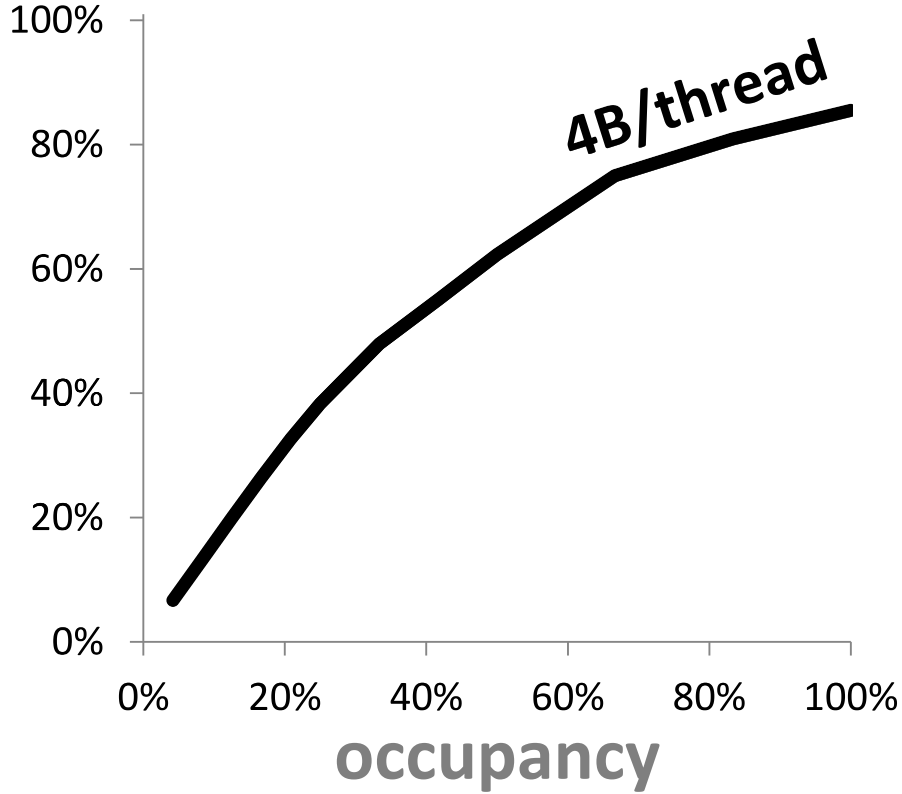
87% of pin bandwidth at only 8% occupancy!

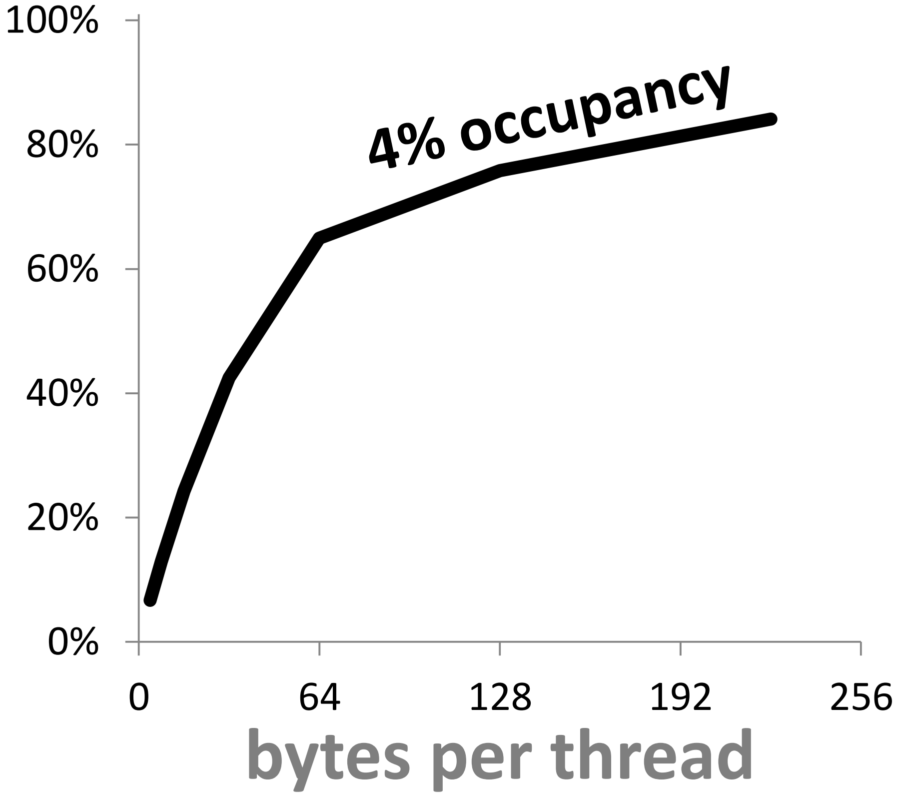
Copying 14 float4 values per thread



84% of peak at 4% occupancy

Two ways to hide memory latency





Fallacy:

"Low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation" (**CUDA Best Practices Guide**)

– *We've just seen 84% of the peak at mere 4% occupancy. Note that this is above 71% that cudaMemcpy achieves at best.*

Fallacy:

"In general, more warps are required if the ratio of the number of instructions with no off-chip memory operands (…) to the number of instructions with off-chip memory operands is low." (**CUDA Programming Guide**)

– *No, we've seen 87% of memory peak with only 4 warps per SM in a memory intensive kernel.*

Part III:

Run faster by using fewer threads

Fewer threads = more registers per thread

More threads

More registers per thread

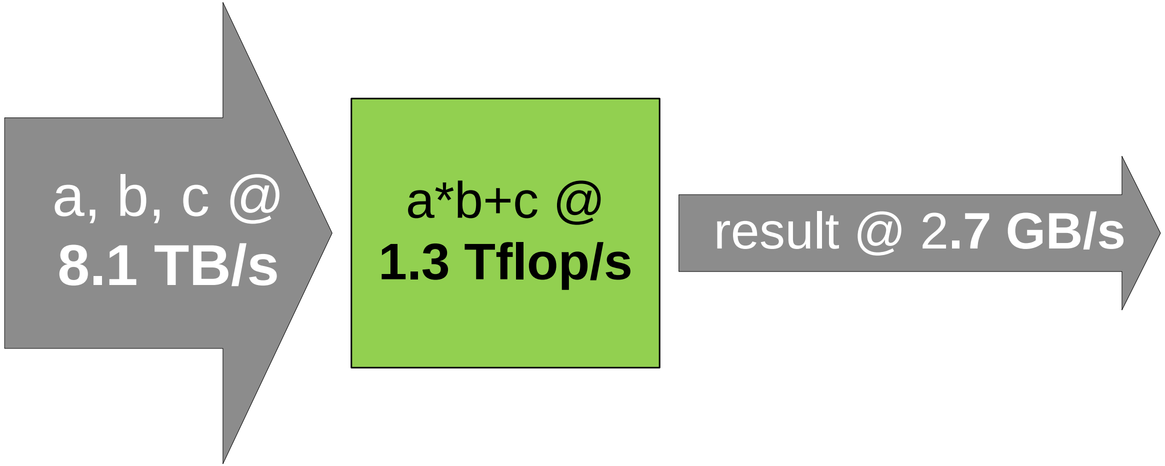
Registers per thread:

GF100: **20** at 100% occupancy, **63** at 33% occupancy — **3x**

GT200: **16** at 100% occupancy, ≈**128** at 12.5% occupancy — **8x**

Is using more registers per thread better?

Only registers are fast enough to get the peak



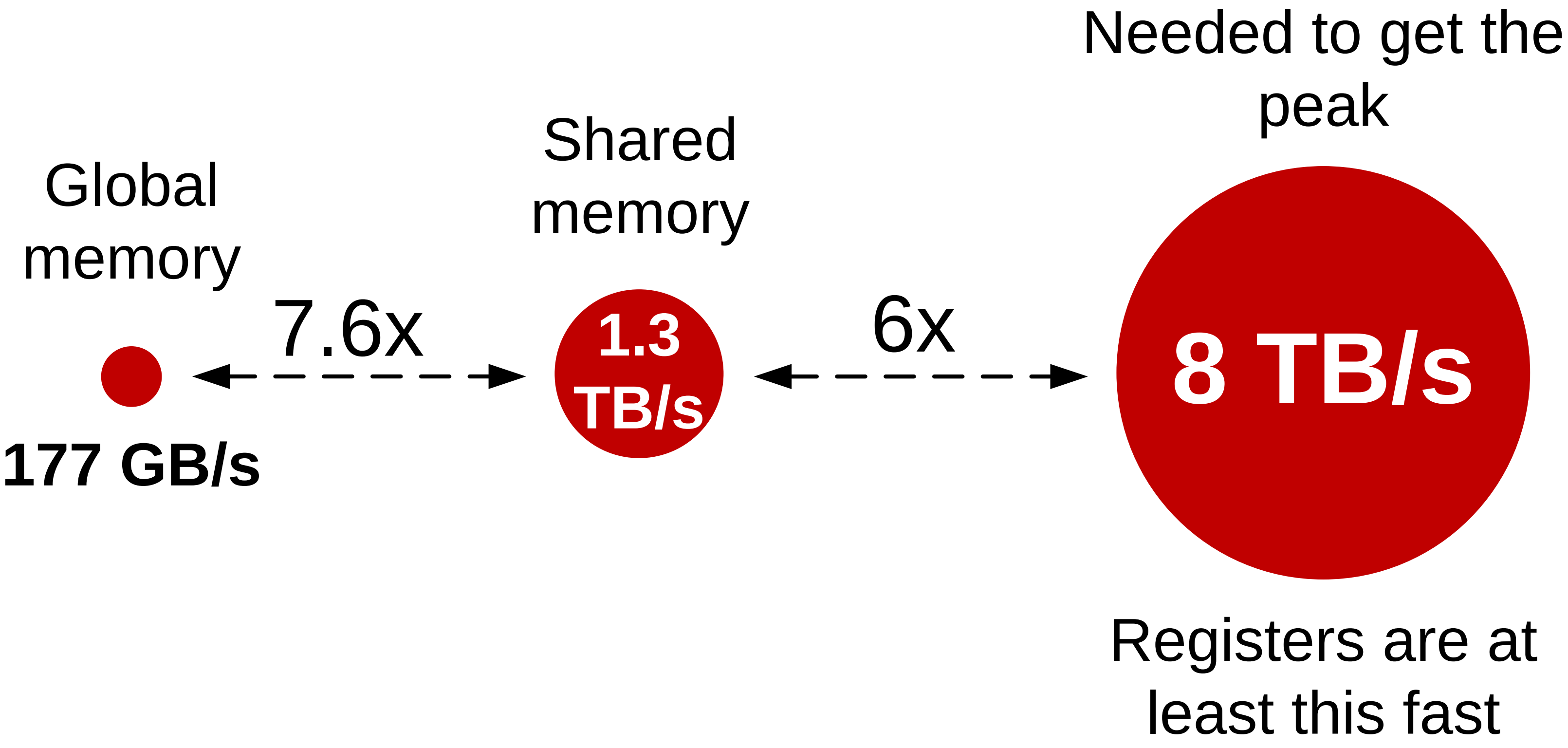
Consider a*b+c: 2 flops, 12 bytes in, 4 bytes out

This is **8.1 TB/s** for 1.3 Tflop/s!

Registers can accommodate it. Can shared memory?

– 4B*32banks*15 SMs*half 1.4GHz = 1.3TB/s only

Bandwidth needed vs bandwidth available



Fallacy:
"In fact, for all threads of a warp, accessing the shared memory is as fast as accessing a register as long as there are no bank conflicts between the threads.."
(**CUDA Programming Guide**)

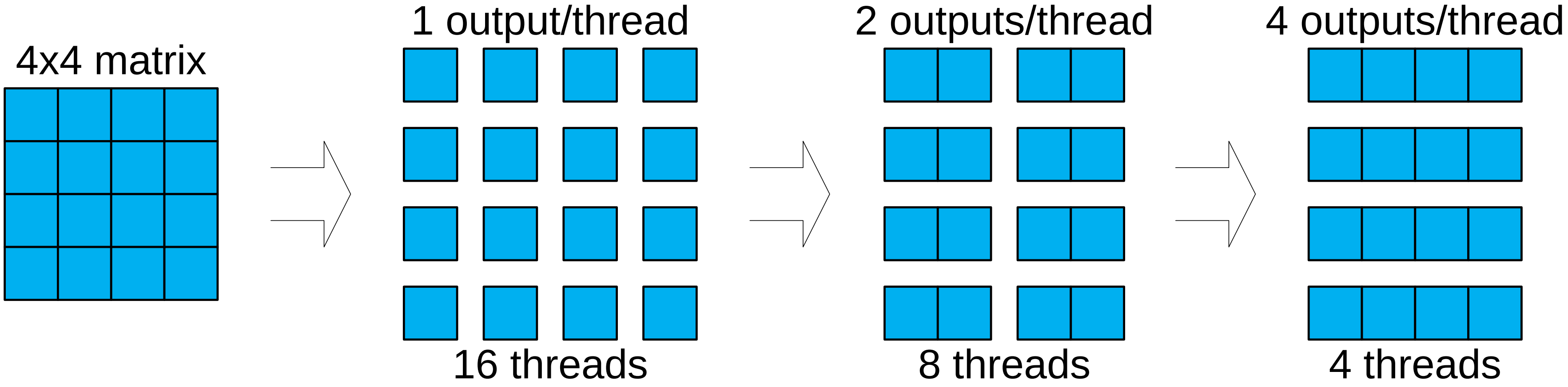
– *No, shared memory bandwidth is $\geq$ 6x lower than register bandwidth on Fermi. ($\geq$3x before Fermi.)*

Running fast may require low occupancy

- ***Must* use registers to run close to the peak**
- The larger the bandwidth gap, the more data must come from registers
- This may require many registers = **low occupancy**

This often can be accomplished by *computing multiple outputs per thread*

Compute multiple outputs per thread



More data is local to a thread in registers

– *may* need fewer shared memory accesses

Fewer threads, but more parallel work in thread

– So, low occupancy should not be a problem

From Tesla to Fermi: regression?

The gap between shared memory and arithmetic throughput has increased:

- G80-GT200: **16** banks vs **8** thread processors (**2:1**)
- GF100: **32** banks vs **32** thread processors (**1:1**)
- GF104: **32** banks vs **48** thread processors (**2:3**)

Using fast register memory could help. But instead, register use is restricted:

- G80-GT200: up to ≈128 registers per thread
- Fermi: up to ≈64 registers per thread

Part IV:

Case study: matrix multiply

Baseline: matrix multiply in CUDA SDK

- I'll show very specific steps for SDK 3.1, GTX480
- Original code shows 137 Gflop/s
- First few changes:
  – Use larger matrices, e.g. 1024x1024 (matrixMul.cu)
    - "uiWA = uiHA = uiWB = uiHB = uiWC = uiHC = 1024 ; "
    - Get 240 Gflop/s
  – Remove "–maxrregcount 32" (or increase to 63)
    - Not important now, but will matter later
  – Increase BLOCK_SIZE to 32 (matrixMul.h)
    - Must add #pragma unroll (see next slide); 242 Gflop/s

Matrix multiply example in SDK

```
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();

#pragma unroll
    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;
```

Baseline performance

- One output per thread so far
- 242 Gflop/s
  - 2 flops per 2 shared memory accesses = 4 B/flop
  - So, bound by shared memory bandwidth to 336 Gflop/s
  - We’ll approach 500 Gflop/s in a few slides
- 21 register per thread (sm_20)
- 67% occupancy
- But only 1 block fits per SM
  - Can’t overlap global memory access with arithmetic

Two outputs per thread (I)

In the new code we use 2x smaller thread blocks

– But same number of blocks

matrixMul.cu:

```
// setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE/2); //32x16
dim3 grid(uiWC / BLOCK_SIZE, uiHC / BLOCK_SIZE);
```

2x fewer threads, but 2x more work per thread:

Two outputs per thread (II)

```
float Csub[2] = {0,0};//array is allocated in registers
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+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    __syncthreads();
```

Define 2 outputs and do 2x more loads

Two outputs per thread (III)

```
#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub[0] += AS(ty, k) * BS(k, tx);
        Csub[1] += AS(ty+16, 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+16) + tx] = Csub[1];
```

Do 2x more flops and stores

Two outputs per thread: performance

- Now 341 Gflop/s — **1.4x speedup**
  - Already above 336 Gflop/s bound
- 28 registers
  - 2x more work with only 1.3x more registers
- Now 2 threads blocks fit per SM
  - Because fewer threads per block, 1536 max per SM
  - Now can overlap of memory access with arithmetic
  - This is one reason for the speedup
- Same 67% occupancy

Shared memory traffic is now lower

- Date fetched from shared memory is now **reused**:

```
for (int k = 0; k < BLOCK_SIZE; ++k)
{
    Csub[0] += AS(ty, k) * BS(k, tx);
    Csub[1] += AS(ty+16, k) * BS(k, tx);
}
```

- Now 5B/flop in shared memory accesses
- New bound: 448 Gflop/s
  - We'll surpass this too

Four outputs per thread (I)

Apply same idea again

Shrink thread blocks by another factor of 2:

```
// setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE/4); //32x8
dim3 grid(uiWC / BLOCK_SIZE, uiHC / BLOCK_SIZE);
```

Four outputs per thread (II)

```
float Csub[4] = {0,0,0,0};//array is in registers
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];
    AS(ty+16, tx) = A[a + wA * (ty+16) + tx];
    BS(ty+16, tx) = B[b + wB * (ty+16) + tx];
    AS(ty+24, tx) = A[a + wA * (ty+24) + tx];
    BS(ty+24, tx) = B[b + wB * (ty+24) + tx];
    __syncthreads();
```

Four outputs per thread (III)

```
#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);
        Csub[2] += AS(ty+16, k) * BS(k, tx);
        Csub[3] += AS(ty+24, 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];
C[c + wB * (ty+16) + tx] = Csub[2];
C[c + wB * (ty+24) + tx] = Csub[3];
```

Four outputs per thread: performance

- Now 427 Gflop/s — 1.76x speedup vs. baseline!
  - Because access shared memory even less
- 41 registers
  - Only ≈2x more registers
  - So, ≈2x **fewer** registers per thread block
- 50% occupancy — 1.33x lower
  - **Better performance at lower occupancy**
- 3 thread blocks per SM
  - Because fewer registers per thread block

Eight outputs per thread: performance

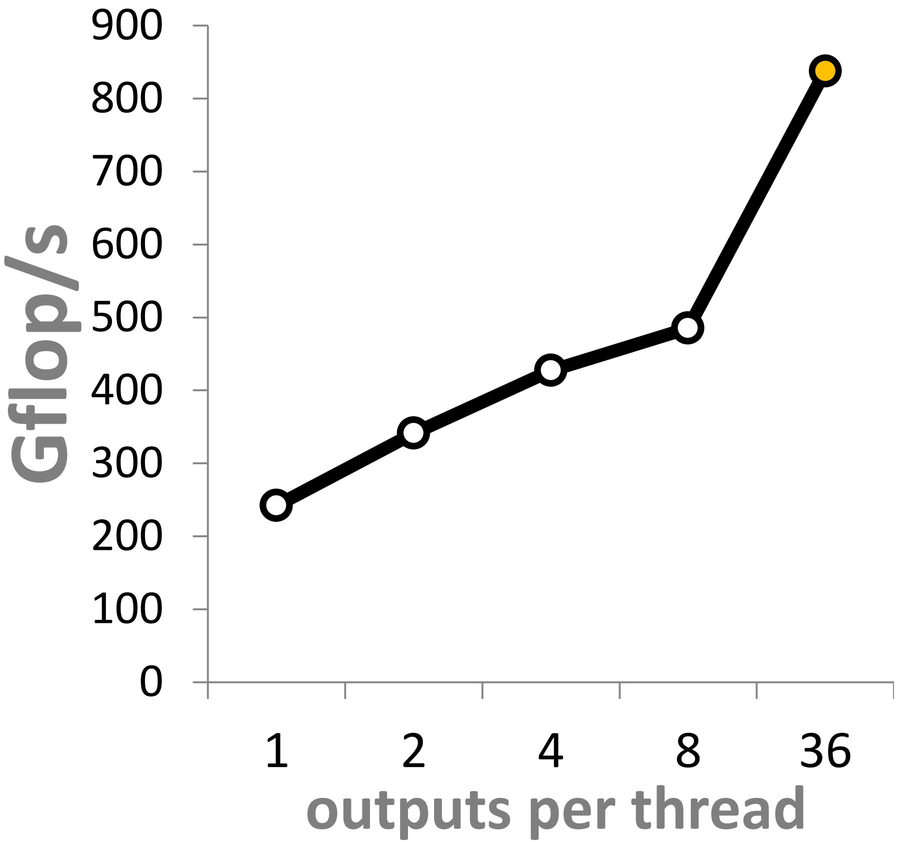
- Now 485 Gflop/s — 2x speedup vs. baseline!
  - Only 2.25 B/flop — 1.8x lower
- 63 registers — 3x more
  - But do 8x more work!
- 33% occupancy — 2x lower
  - **Better performance at lower occupancy**
- 4 thread blocks per SM

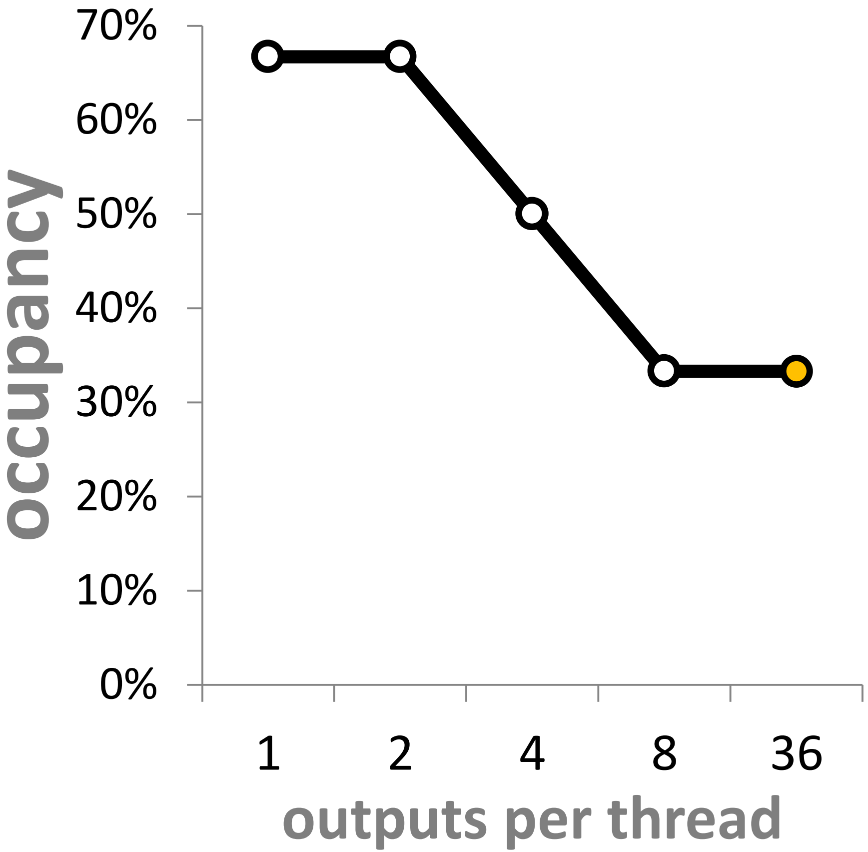
How much faster we can get?

MAGMA BLAS — up to 838 Gflop/s

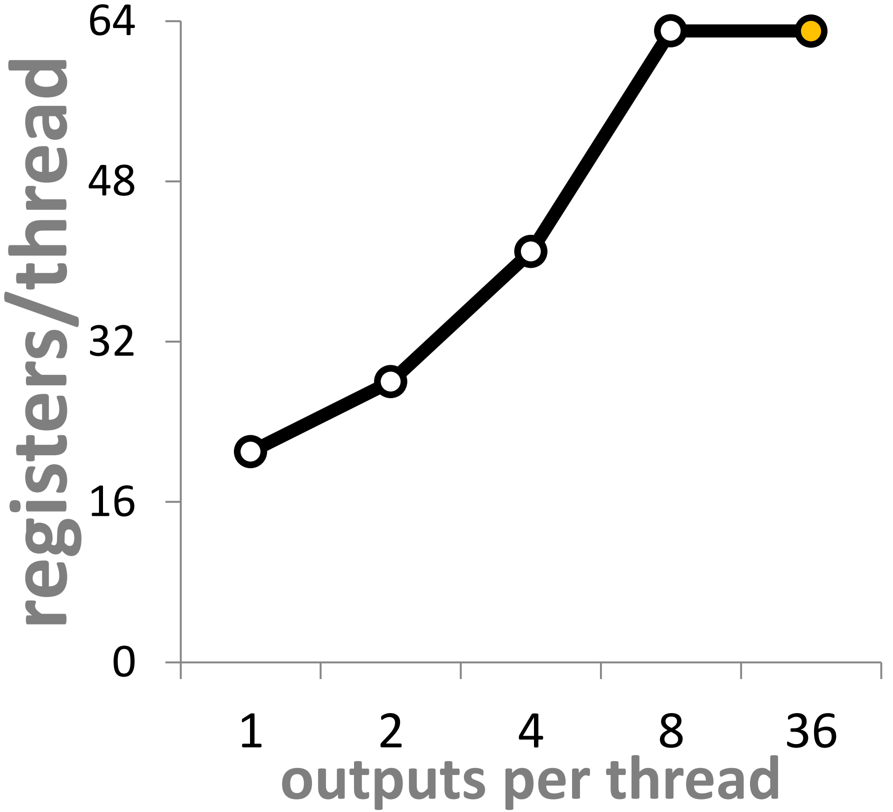
- 36 outputs per thread
- 0.67 B/flop only — 6x lower
- 33% occupancy
- 2 thread blocks per SM

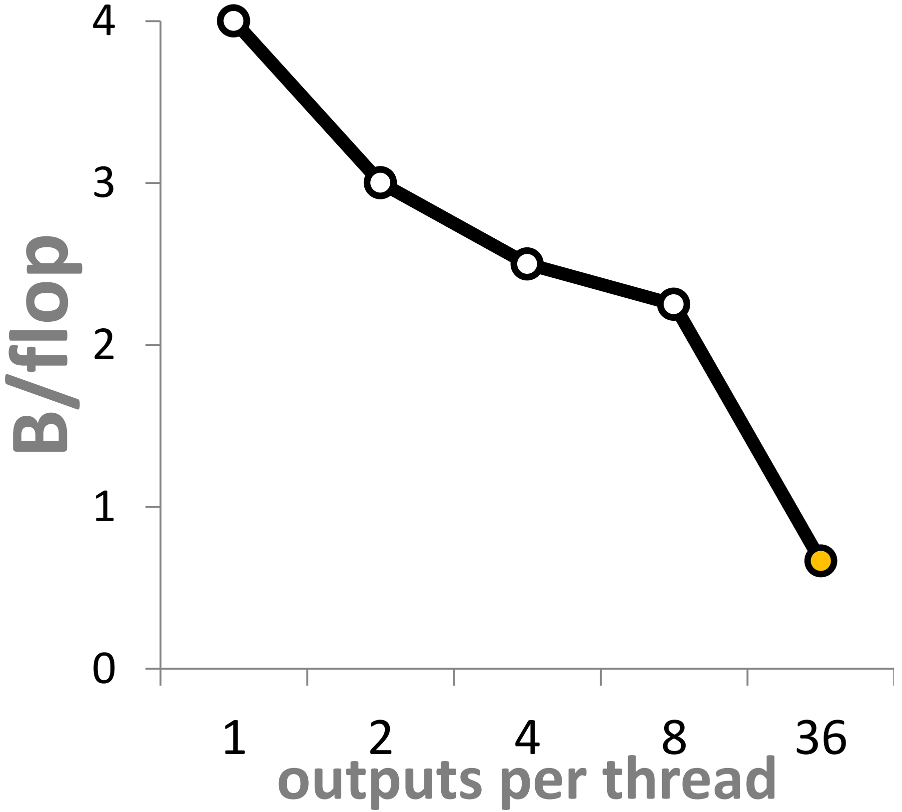
GFLOPS go up, occupancy goes down





Register use goes up, smem traffic down

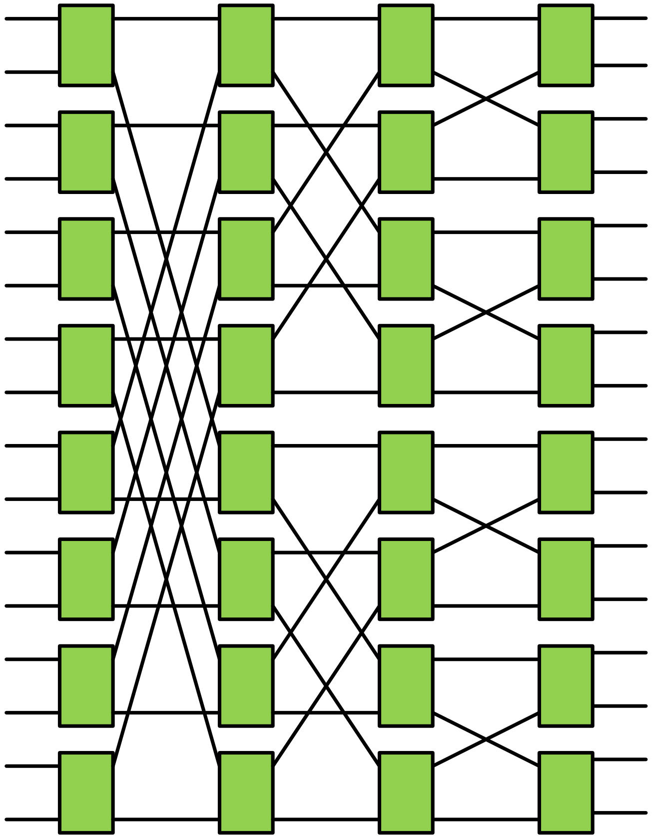




Part V:

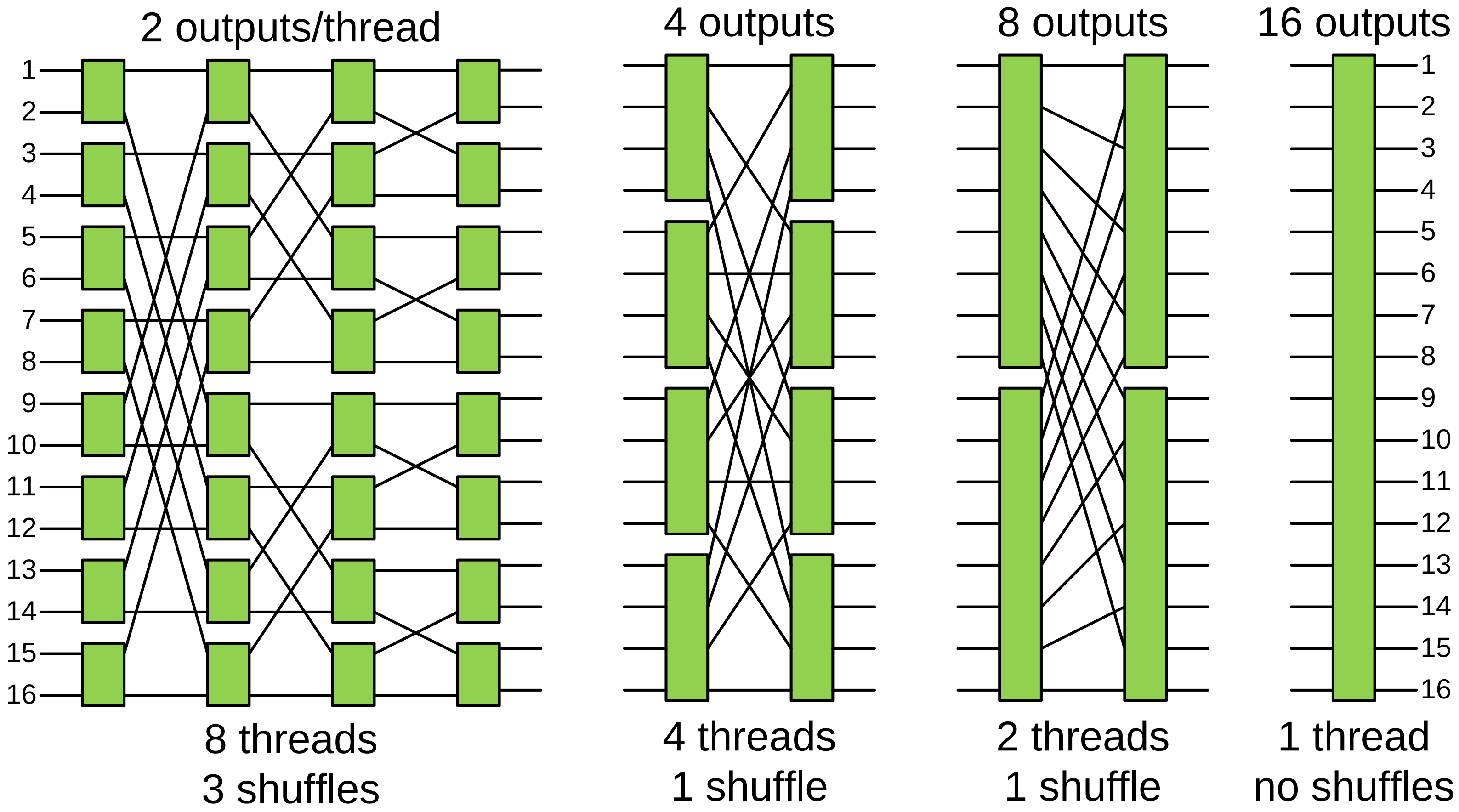
Case Study: FFT

Mapping Cooley-Tukey to GPU



- Cooley-Tukey splits large FFT into smaller FFTs
- Assume FFT fits into thread block
- Small FFT are done in registers
- Shuffles are done using shared memory

Fewer threads – lower shared memory traffic



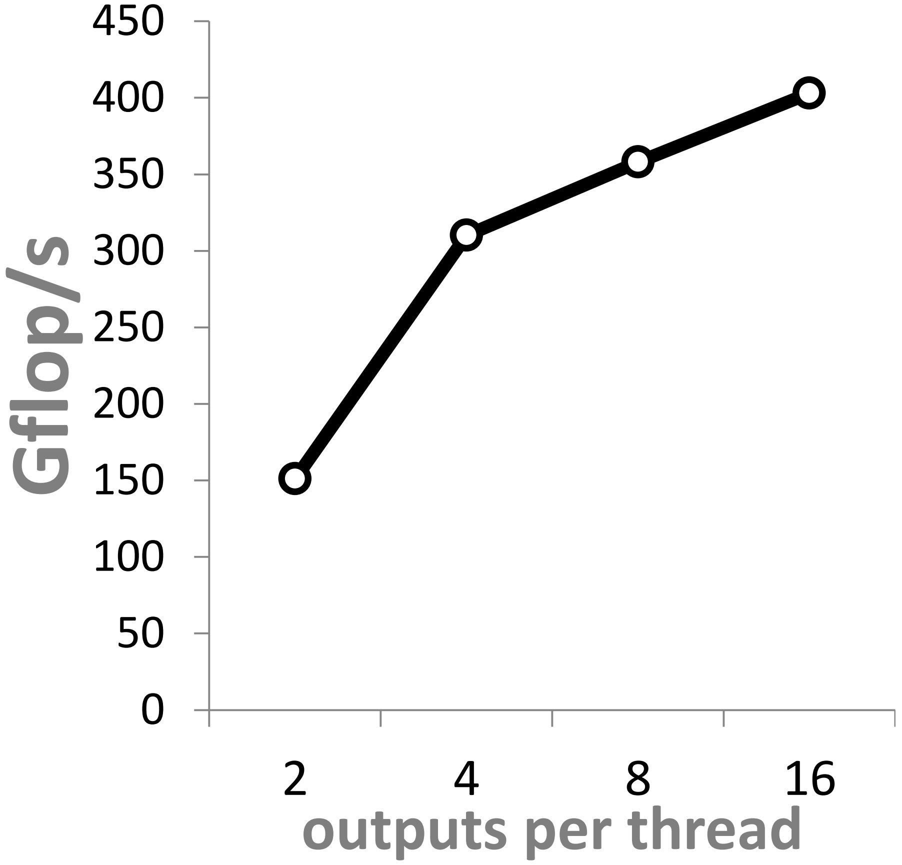
Two outputs per thread

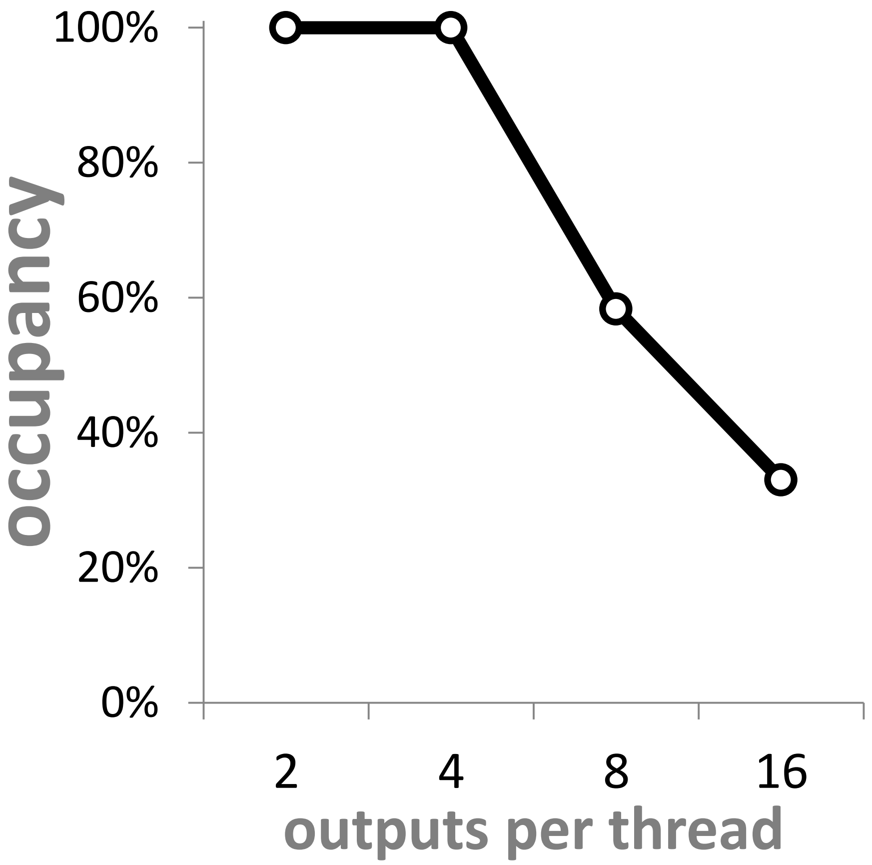
```
__global__ void FFT1024( float2 *dst, float2 *src ){
  float2 a[2]; int tid = threadIdx.x;
  __shared__ float smem[1024];
  load<2>( a, src+tid+1024*blockIdx.x, 512 );
  FFT2( a );
#pragma unroll
  for( int i = 0; i < 9; i++ ) {
      int k = 1<<i;
      twiddle<2>( a, tid/k, 1024/k );
      transpose<2>( a, &smem[tid+(tid&~(k-1))], k, &smem[tid], 512 );
      FFT2( a );
  }
  store<2>( a, dst+tid+1024*blockIdx.x, 512 );
}
```

Sixteen outputs per thread

```
__global__ void FFT1024( float2 *dst, float2 *src ){
   float2 a[16]; int tid = threadIdx.x;
   __shared__ float smem[1024];
   load<16>( a, src+tid+1024*blockIdx.x, 64 );
   FFT4( a, 4, 4, 1 );// four FFT4
   twiddle<4>( a, threadIdx.x, 1024, 4 );
   transpose<4>( a, &smem[tid*4], 1, &smem[tid], 64, 4 );
#pragma unroll
   for( int i = 2; i < 10-4; i += 4 ) {
     int k = 1<<i;
     FFT16( a );
     twiddle<16>( a, threadIdx.x/k, 1024/k );
     transpose<16>( a, &smem[tid+15*(tid&~(k-1))], k, &smem[tid], 64 );
   }
   FFT16( a );
   store<16>( a, dst+tid+1024*blockIdx.x, 64 );
}
```

GFLOPS go up, occupancy goes down





Summary

- Do more parallel work per thread to hide latency with fewer threads
- Use more registers per thread to access slower shared memory less
- Both may be accomplished by computing multiple outputs per thread

Compute more outputs per thread