PMPP 2015/16

Performance Tuning (1)

Course Schedule

12.10.2015	Introduction to PMPP
13.10.2015	Lecture CUDA Programming 1
19.10.2015	Lecture CUDA Programming 2
20.10.2015	Lecture CUDA Programming 3
26.10.2015	Lecture Parallel Basics, Exercise 1 assigned
27.10.2015	Questions and Answers (Q&A), S3|19, Room 2.8
2.11.2015	Intro Final Proj., Ex. 1 due, Ex. 2 assigned, Lecture PRAM
3.11.2015	Lecture PRAM (2)
9.11.2015	Final Projects assigned, L. Parallel Sort., Exercise 2 due
10.11.2015	Questions and Answers (Q&A)
16.11.2015	Questions and Answers (Q&A)
17.11.2015	Questions and Answers (Q&A)
23.11.2015	1st Status Presentation Final Projects
24.11.2015	1st Status Presentation Final Projects (continued)
30.11.2015	Lecture Design Patterns
1.12.2015	Questions and Answers (Q&A)

(Preliminary) Course Schedule

→ you are here

7.12.2015	Lecture Design Patterns (2), Performance Tuning
8.12.2015	Questions and Answers (Q&A)
14.12.2015	Performance Tuning (2)
15.12.2015	Questions and Answers (Q&A)
11.1.2016	2nd Status Presentation Final Projects
12.1.2016	2nd Status Presentation Final Projects (continued)
18.1.2016	Performance Tuning (3)
19.1.2016	
25.1.2016	
26.1.2016	
1.2.2016	
2.2.2016	
8.2.2016	Final Presentation Final Projects
9.2.2016	Final Presentation Final Projects (continued)

Overview

- Instruction Level Parallelism
- Identifying Performance Limiters
  - Instruction Limited Kernels
  - Latency Limited Kernels
  - Bandwidth Limited Kernels
  - Local Memory Optimizations
- Multi GPU Multi Threading
  - Brief review of the scenarios
  - Single CPU process, multiple GPUs
    - GPU selection, UVA, P2P
  - Multiple processes
    - Needs CPU-side message passing
  - Dual-IOH CPU systems and NUMA

Images and slides partially by developer.nvidia.com

Instruction Level Parallelism (ILP)

- 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…
- Two common fallacies
  - Multithreading is the only way to hide latency on a GPU
  - Shared memory is as fast as register

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 by overlapping with other code

```
x = a + b; // takes ≈20 cycles to execute
y = a + c; // independent, can start any time
(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 ≈ 32ops/cycle (op = 32-bit load)
- Little’s Law
  - Required Parallelism: Latency x Throughput

Arithmetic Parallelism in Numbers

GPU model	Latency (cycles)	Throughput (cores/SM)	Parallelism (ops/SM)
G80-GT200	≈24	8	≈192
GF100	≈18	32	≈576
GF104	≈18	48	≈864
GF110	≈18	32	≈576
GK110	≈12	192	≈2304
GM200	≈8	128	≈1024

- Latency varies between different ops
- Can’t get 100% throughput with less parallelism
  - Not enough operations in flight = idle cycles

Thread Level Parallelism (TLP)

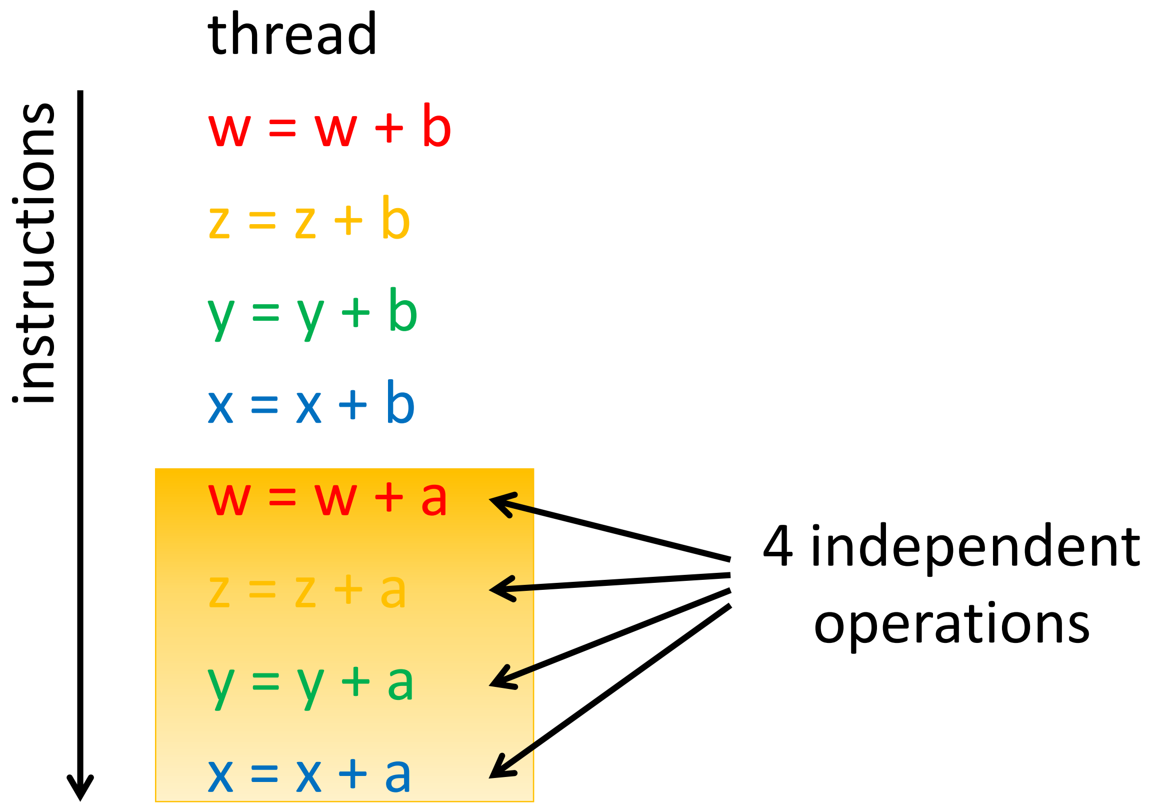
- It is usually recommended to use threads to supply the needed parallelism, e.g. 192 threads per SM on G80:

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:



You can use both ILP and TLP on GPU

- This applies to all CUDA-capable GPUs, e.g. 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% peak
  - 48 cores/SM, one instruction is broadcasted across 16 cores (half-warp)
  - So must issue 3 instructions per cycle
  - But have only 2 half-warp schedulers
  - Instead, it can issue 2 instructions per half-warp in the same cycle

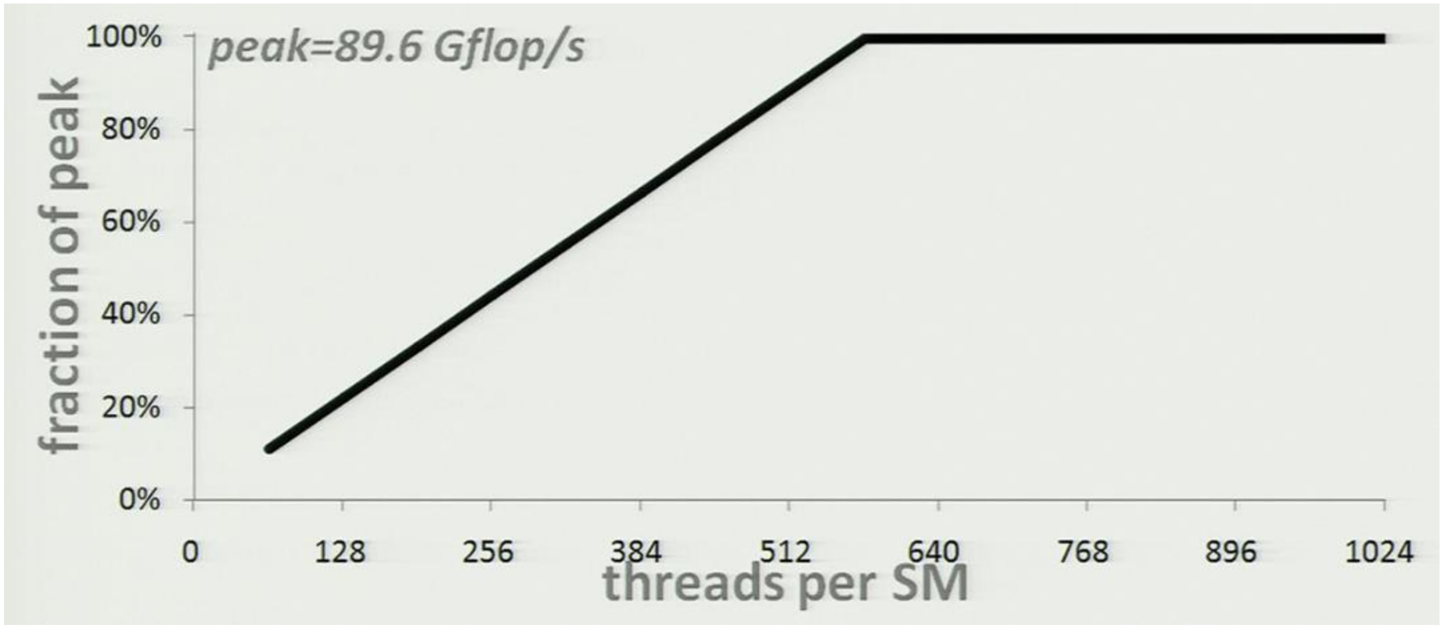
Let’s check it experimentally

- 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 use later
- Run 1 block (use 1 SM), vary block size
  - See what fraction of peak (1.3TFLOPS/15) we get (gf100)

Experimental results (GTX480)



- No ILP, need 576 threads to get 100% utilization

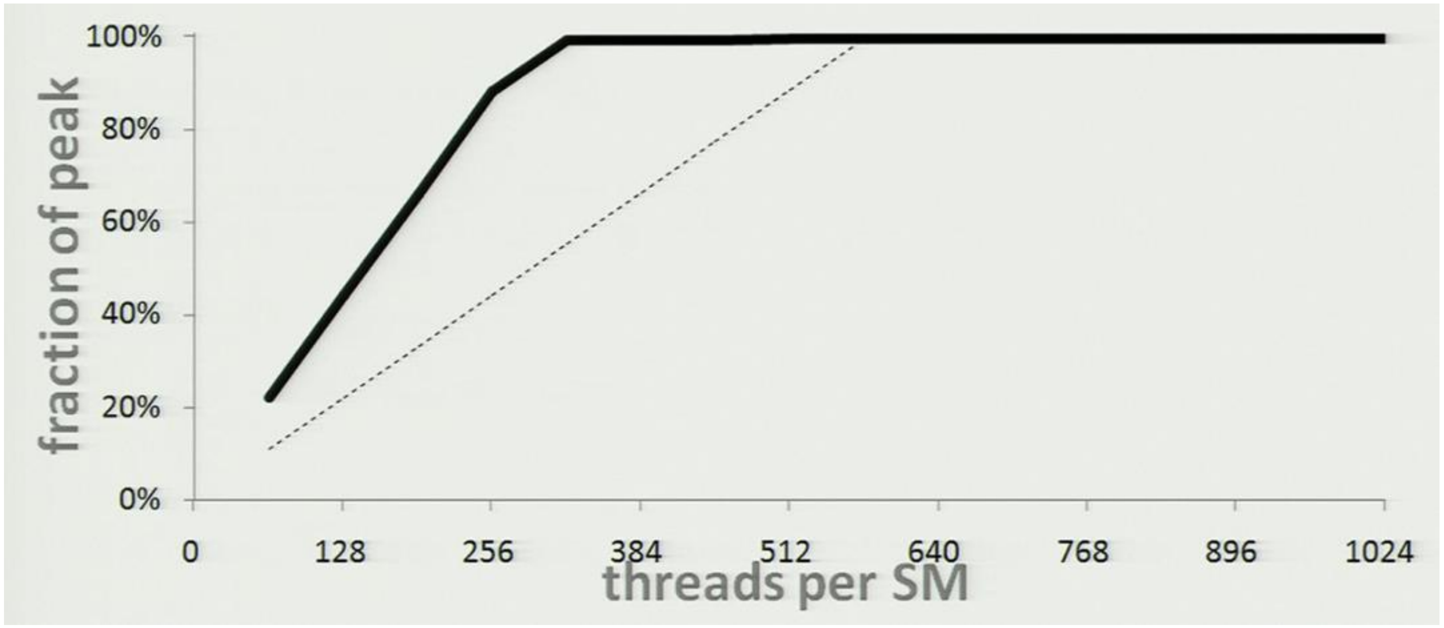
Introduce Instruction Level Parallelism

- Try ILP=2: two independent instructions 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

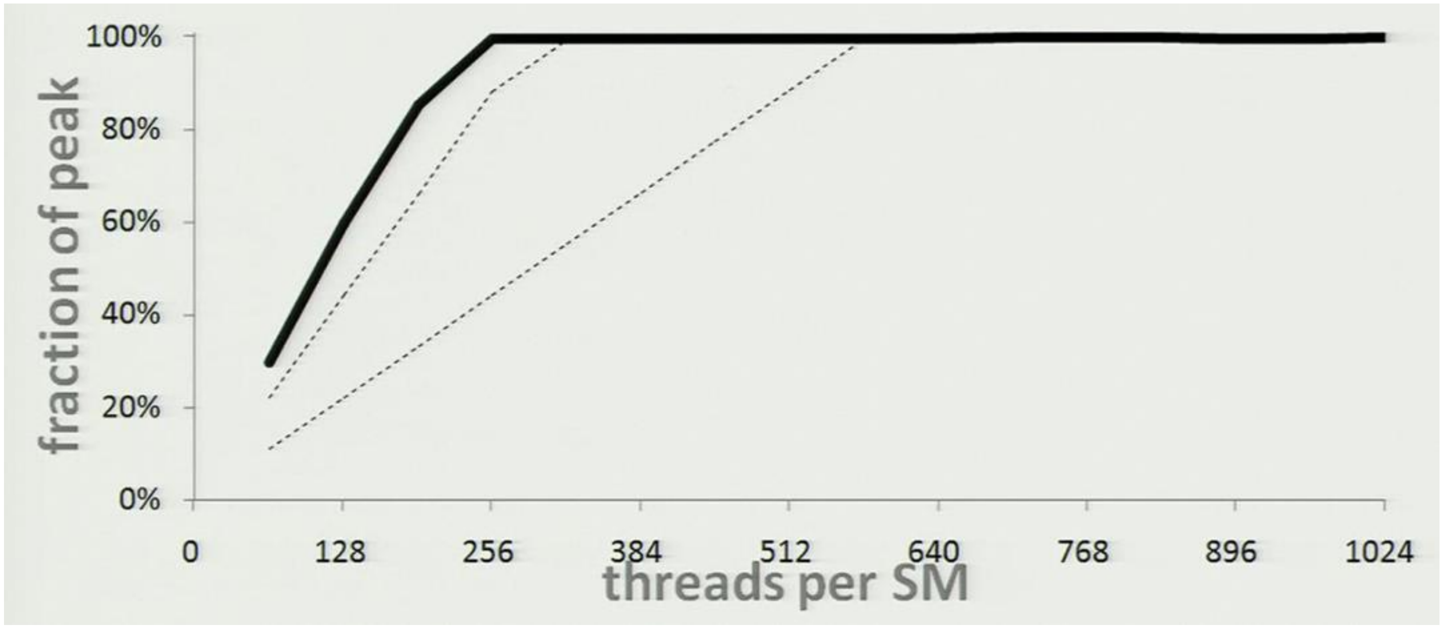
Add more Instruction Level Parallelism

- Try ILP=3: three independent instructions per thread

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

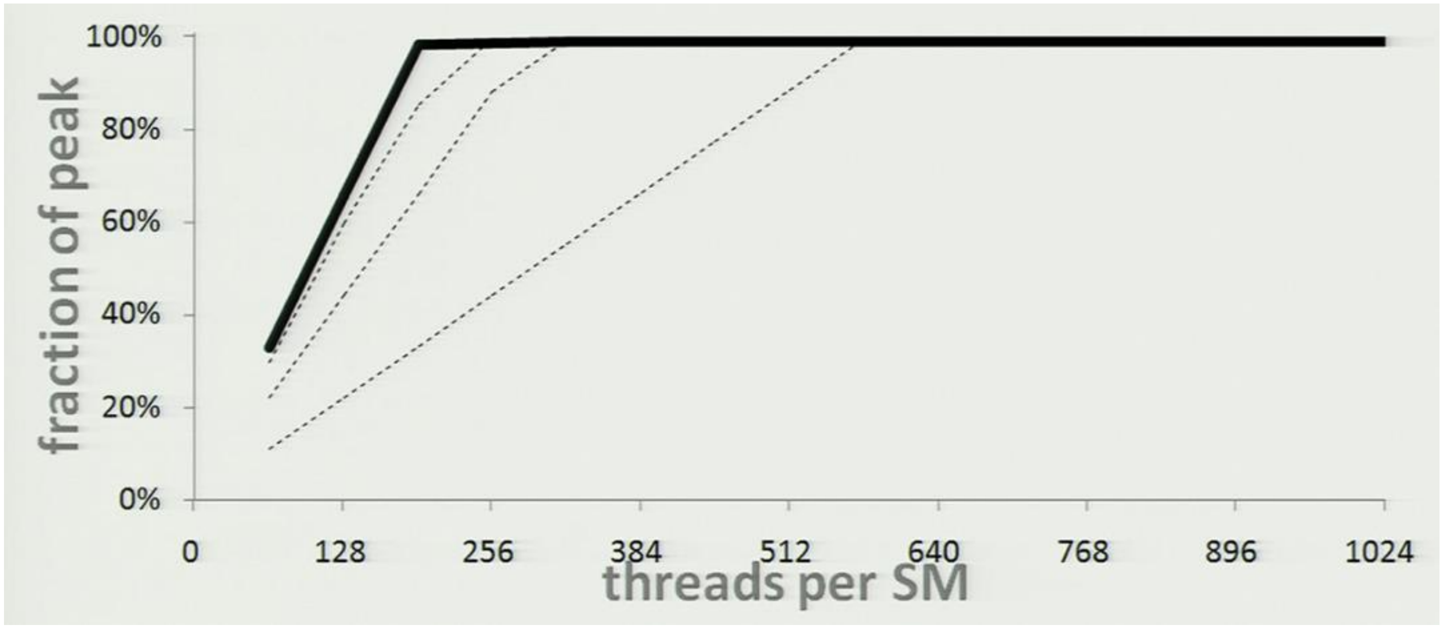
- How far can we push it?

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

Hiding Memory Latency

- Apply same formula but for memory operations
    - Needed parallelism = Latency x Throughput

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

- So, hide memory latency = keep 100 KB in 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 multiple threads to get 100 KB in 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 100B/thread – need 1,000 threads

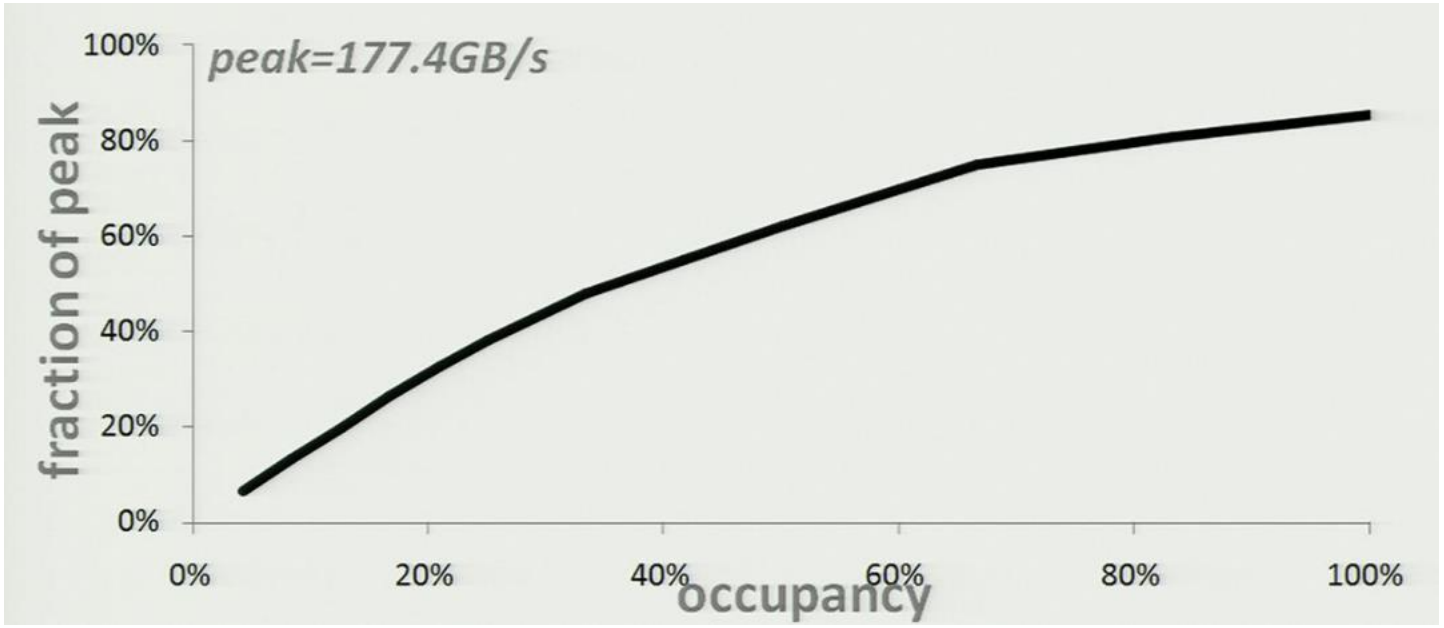
Empirical Validation

- 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 one float per thread (GTX480)



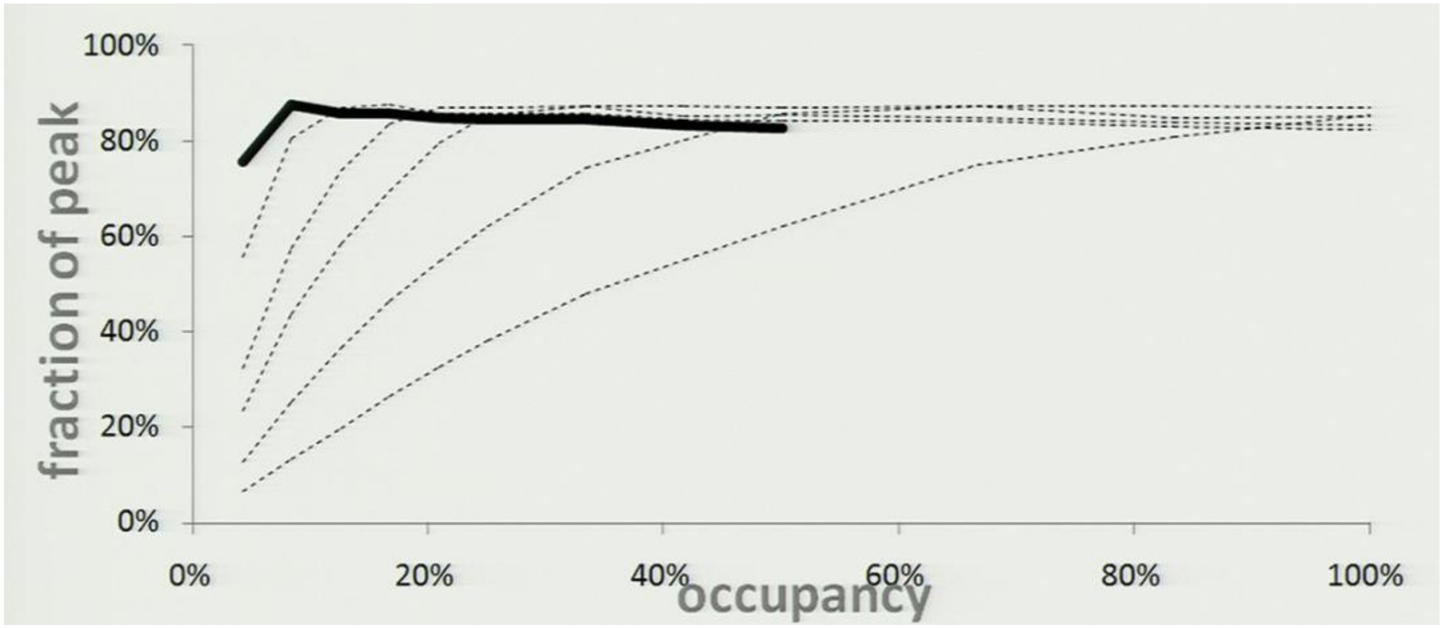
- Must maximize occupancy to hide latency?

How far can we go?

- Copy 8 float4 per thread

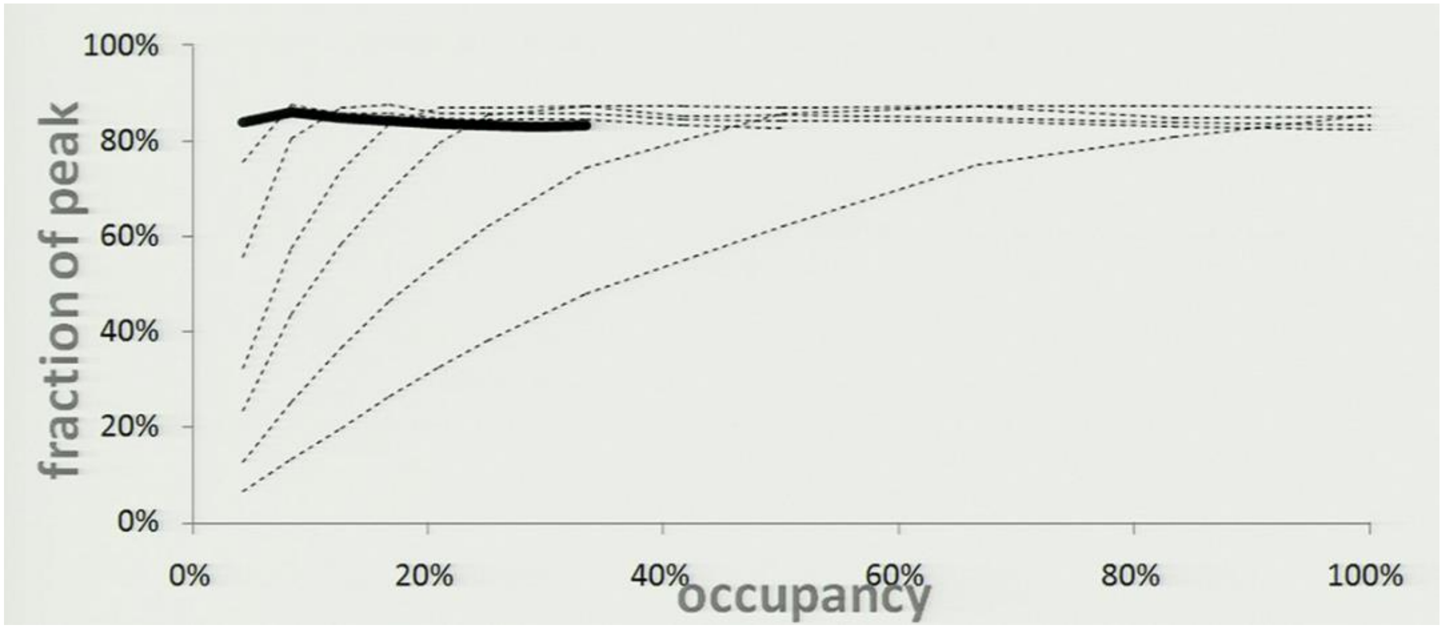
```
__global__ void memcpy(float *dst, float *src)
{
  int block = blockIdx.x + blockIdx.y * gridDim.x;
  int index = threadIdx.x + block * blockDim.x;
  index = index * 8;

  float4 a[8];
  for (int i = 0; i < 8; i++)
    a[i] = src[index + i];
  for (int i = 0; i < 8; i++)
    dst[index + i] = a[i];
}
```

Copying 8 float4 per thread (GTX480)



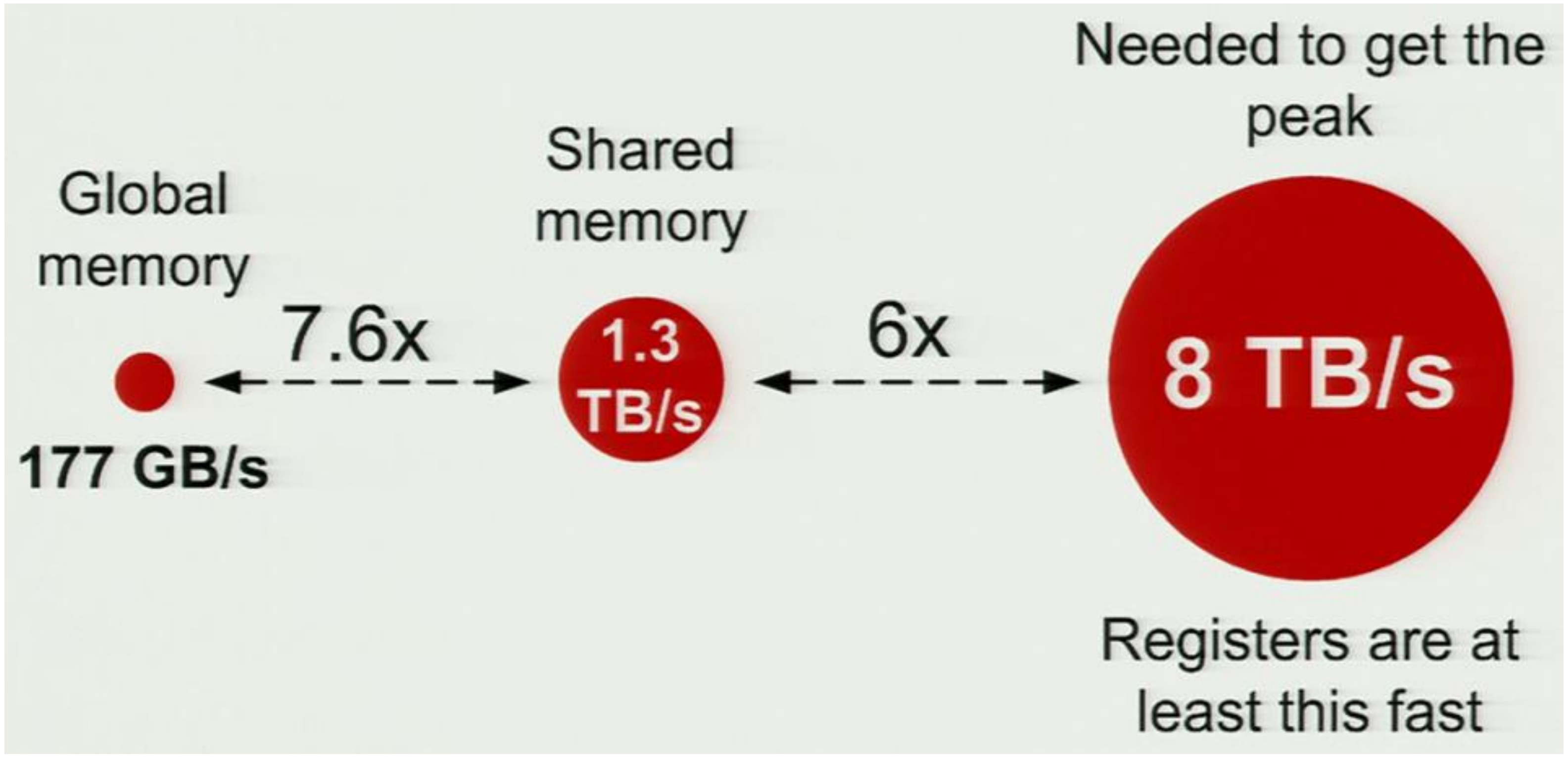
- 87% of peak bandwidth at only 8% occupancy!

Copying 14 float4 per thread (GTX480)



- 84% of peak bandwidth at only 4% occupancy!

Shared Memory Pitfall



Running fast may require less 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