

# Optimizing GPU codes

Vasily Volkov

UC Berkeley

August 11, 2009

# Background

January 2008: we got 205 Gflop/s in **SGEMM** on G80

- CUBLAS was at ~128 Gflop/s
- Best published results were at 90-100 Gflop/s
  - Baskaran et al. 2008, Ryoo et al. 2008
- (Now our code is in CUBLAS)

That was not expected

- How did we do that!?

June 2008: 160 Gflop/s in **FFT** on G80

- CUFFT was at ~50 Gflop/s
- (Since June 2009 CUFFT matches our performance)

# Our code vs. CUBLAS 1.1

- Popular GPU programming guidelines recommend:
  - Minimize use of registers
  - Maximize use of shared memory
  - Use longer thread blocks
  - Maximize occupancy (number of concurrent instruction streams)
- CUBLAS 1.1 succeed in following all of them, but loses in performance:

	CUBLAS 1.1	Our code
Registers per thread	15	30
Shared memory per block	8.3 KB	1.1 KB
Thread block size	512	64
Occupancy (8800 GTX)	67%	33%
Performance (8800 GTX)	128 Gflop/s	205 Gflop/s

- Note that both codes do the same amount of work per block
  - $2048 * K$  flops per thread if multiplying  $M \times K$  matrix by  $K \times N$  matrix

# Memory latency hiding

## Little's law

$$\text{data in transit [B]} = \text{latency [s]} * \text{bandwidth [B/s]}$$

How to keep much data in transit?

- (Prefetch)
- Use long vectors: SIMD
- Use many threads: SMT
- Mixture: SIMT = SIMD + SMT

**It is all about memory concurrency, not threads**

# Little's law in numbers

	8800GTX	9800GTX	GTX280
Sustained bandwidth	76 GB/s	58 GB/s	127 GB/s
Sustained latency	320 ns	300 ns	335 ns
Little's law	24320 B	17400 B	42545 B
Max threads	12288	12288	30720
Min requests/thread	<b>2.0 B</b>	<b>1.4 B</b>	<b>1.4 B</b>

Hides latency if access is **very fine grain**  
≥2 bytes in independent requests in thread

**So small granularity may be unnecessary**

# Block/tiled algorithms

Workflow: **load block**, compute, **store block**, repeat  
- all in one thread block

Consider 32x32 block of single precision numbers

This is **4 KB** data/block or per multiprocessor

Hides latency no matter how many threads are run:

	8800GTX	9800GTX	GTX280
# of multiprocessors	16	16	30
Little's law	24320 B	17400 B	42545 B
Per multiprocessor	<b>1.5 KB</b>	<b>1.1 KB</b>	<b>1.4 KB</b>

**Tiled algorithms don't require many threads**

# Register files

	8800GTX	9800GTX	GTX280
Threads/multiprocessor	768	768	1024
Registers/multiprocessor	8192	8192	16384
Registers/thread	10	10	16
Registers, total	512 KB	512 KB	<b>1.9 MB</b>

Many threads require many registers

Up to  $\approx$  **2 MB** registers on die in total

- largest memory on die (4x shared memory)

**Got many registers – how to use them well?**

# Register usage in CUBLAS SGEMM

warp 1	warp 2	warp 3	warp 4	warp 5	warp 6	warp 7	warp 8	warp 9	warp 10	warp 11	warp 12	warp 13	warp 14	warp 15	warp 16
A's pointer															
A's pointer															
B's pointer															
B's pointer															
counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter	counter
C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data
C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data	C's data
C's index															
C's index															
B's pointer in shared memory															
lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda	lda
*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****
*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****
*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****
*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****

Runs many threads

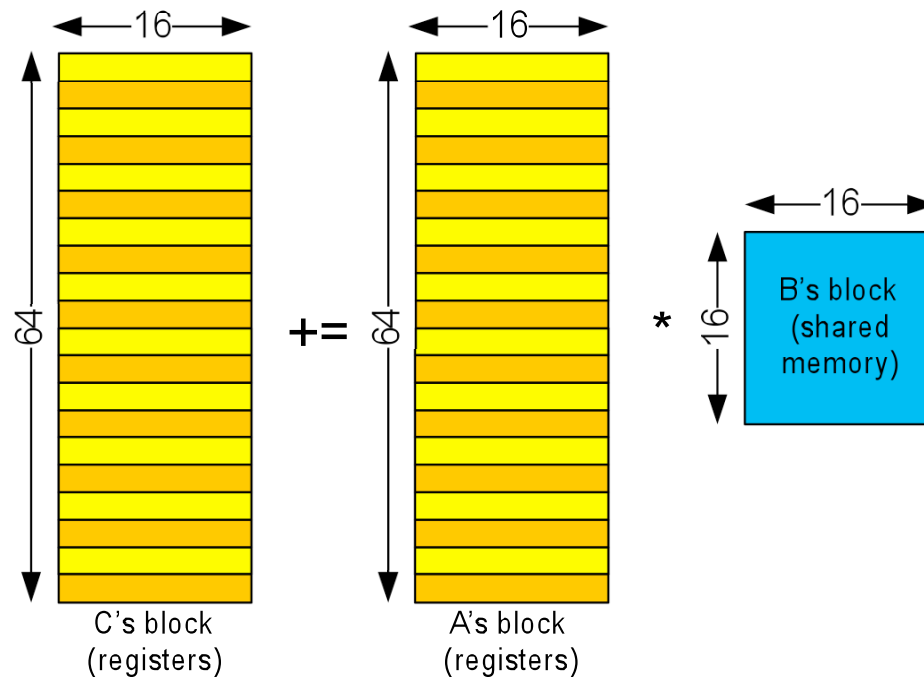
Registers are wasted for auxiliary information

**Extensive thread parallelism eats registers fast**



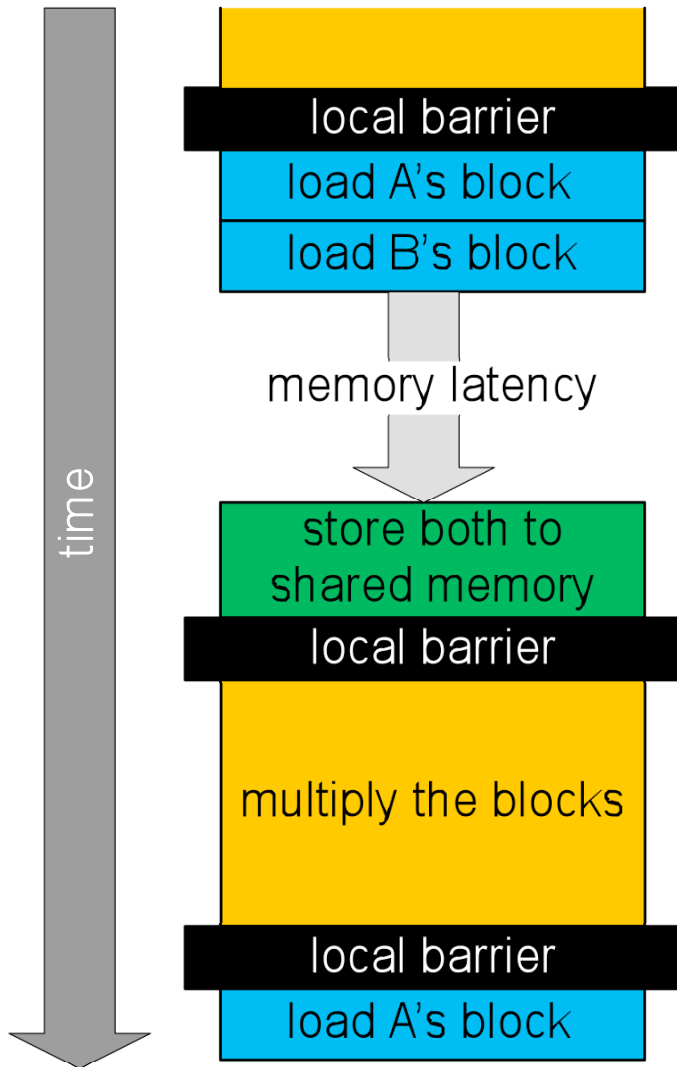
- We want to use registers for working set
- But they are not shared
  - **Use distributed memory algorithms?**

# Local data layout in our SGEMM



- Blocks in A and C are row-cyclic distributed across threads
  - **Little inter-thread communication required**

# Is # of thread blocks important?



CUBLA SGEMM again

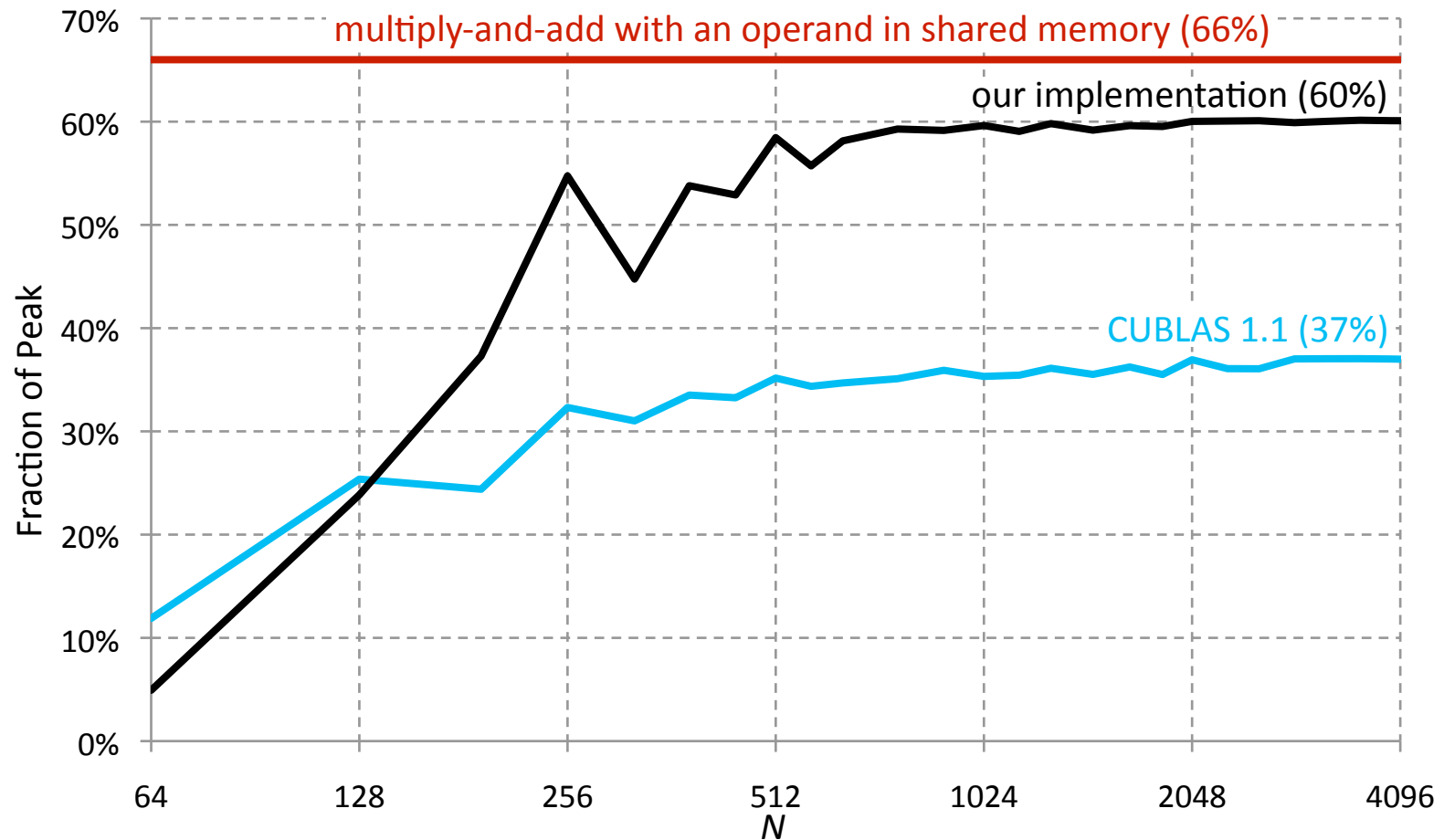
- 1 thread block per multiprocessor
- Many local barriers inside
- Memory latency doesn't overlap with computation

**More thread blocks is good**

- 2-4 often enough

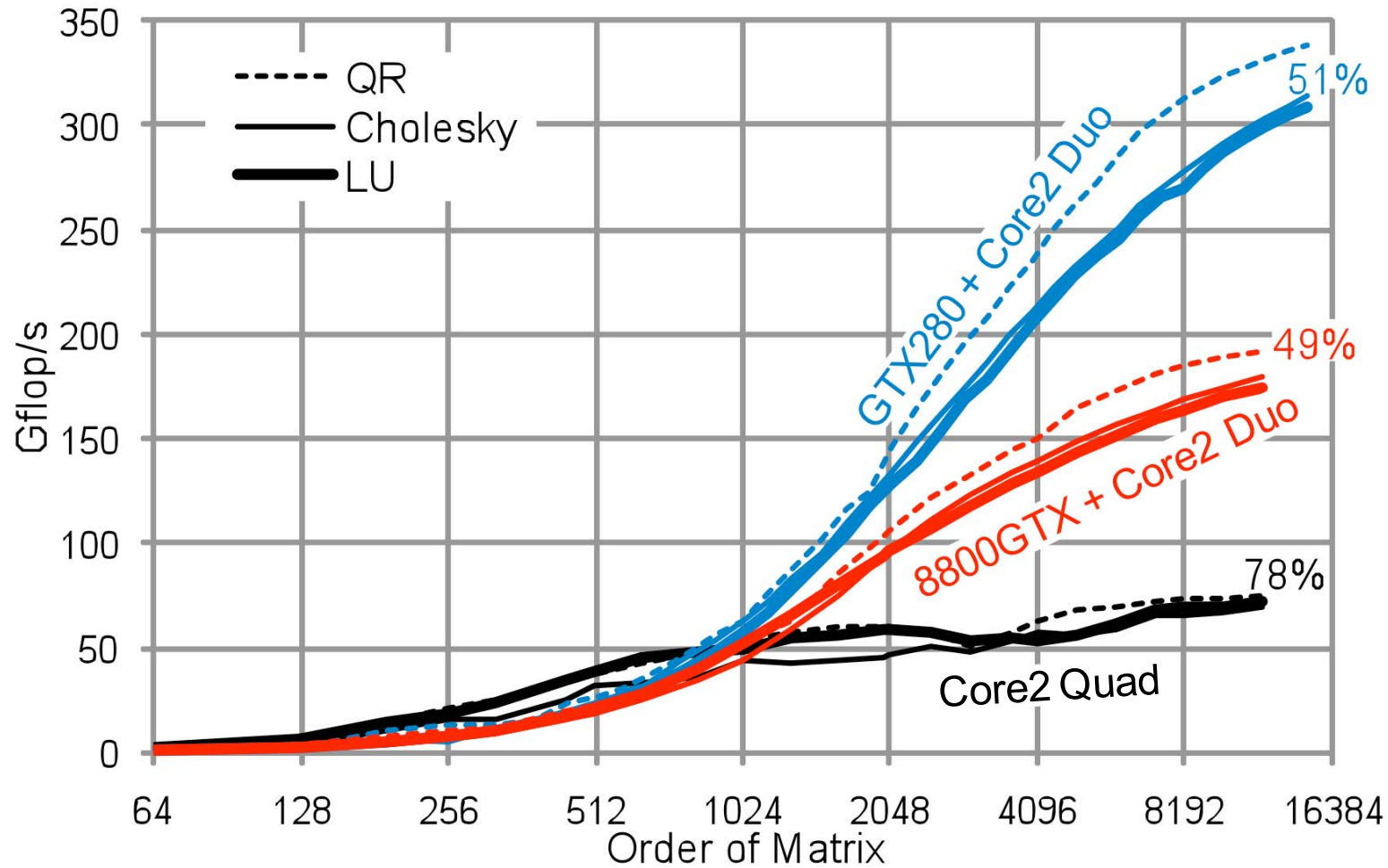
# Our code vs. CUBLAS 1.1

Performance in multiplying two  $N \times N$  matrices on GeForce 8800 GTX:



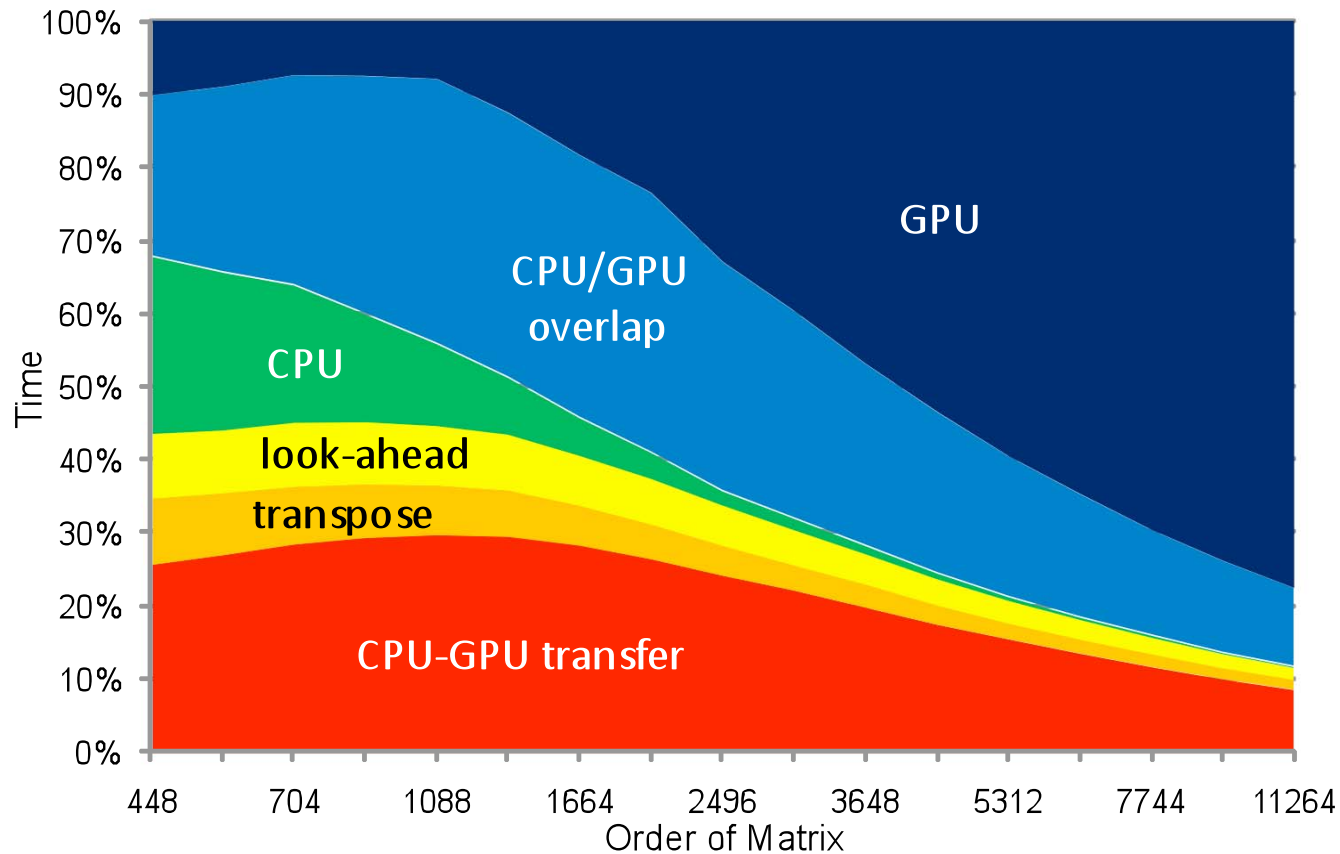
Our SGEMM is used in CUBLAS 2.0 and later; open-source

# Performance Results



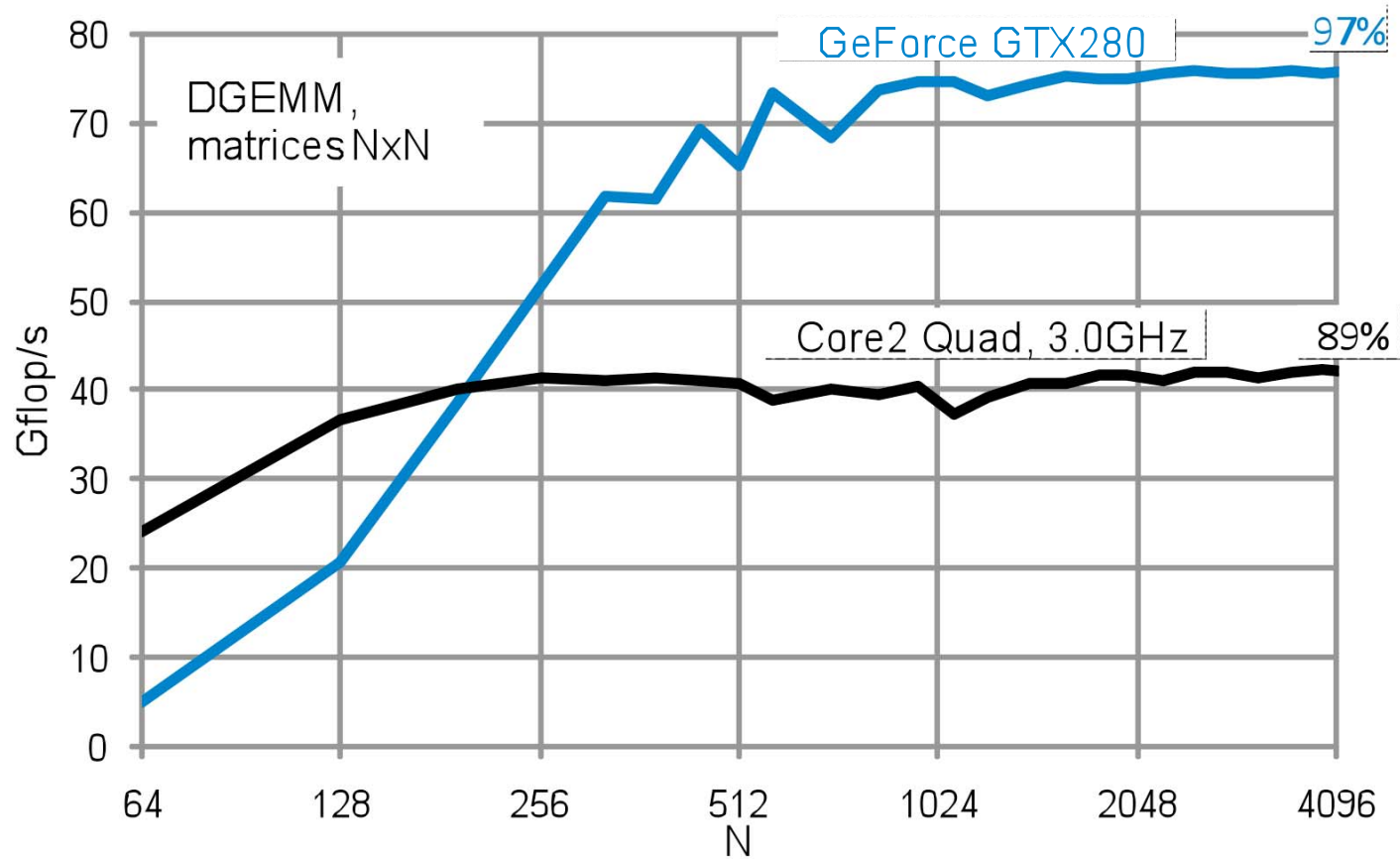
Our solution runs at ~50% of the system's peak (shown on the right)  
Open-source

# Time Breakdown for LU on GeForce 8800 GTX

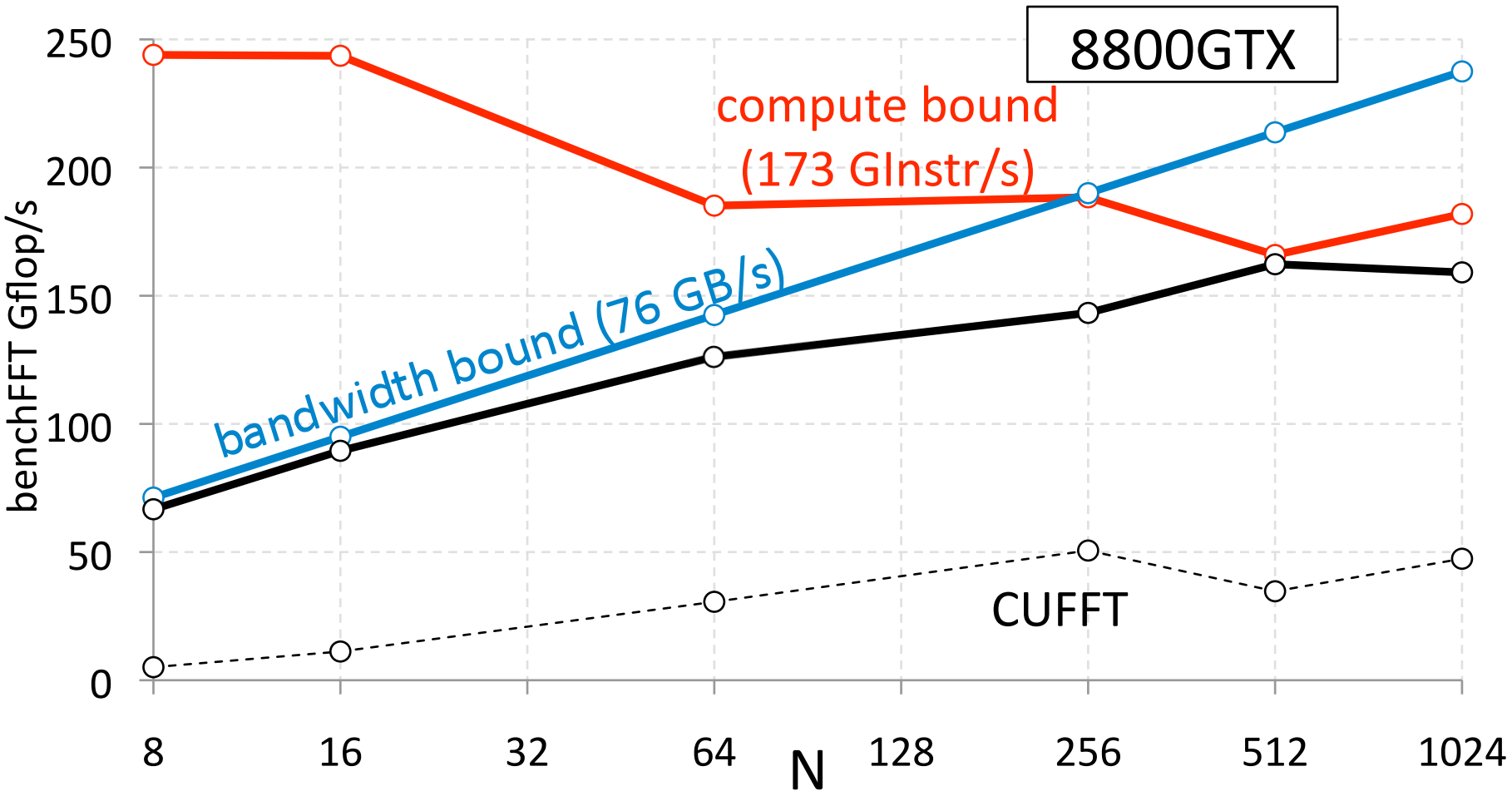


- If we compute on CPU anyway, do that in parallel with computing on GPU

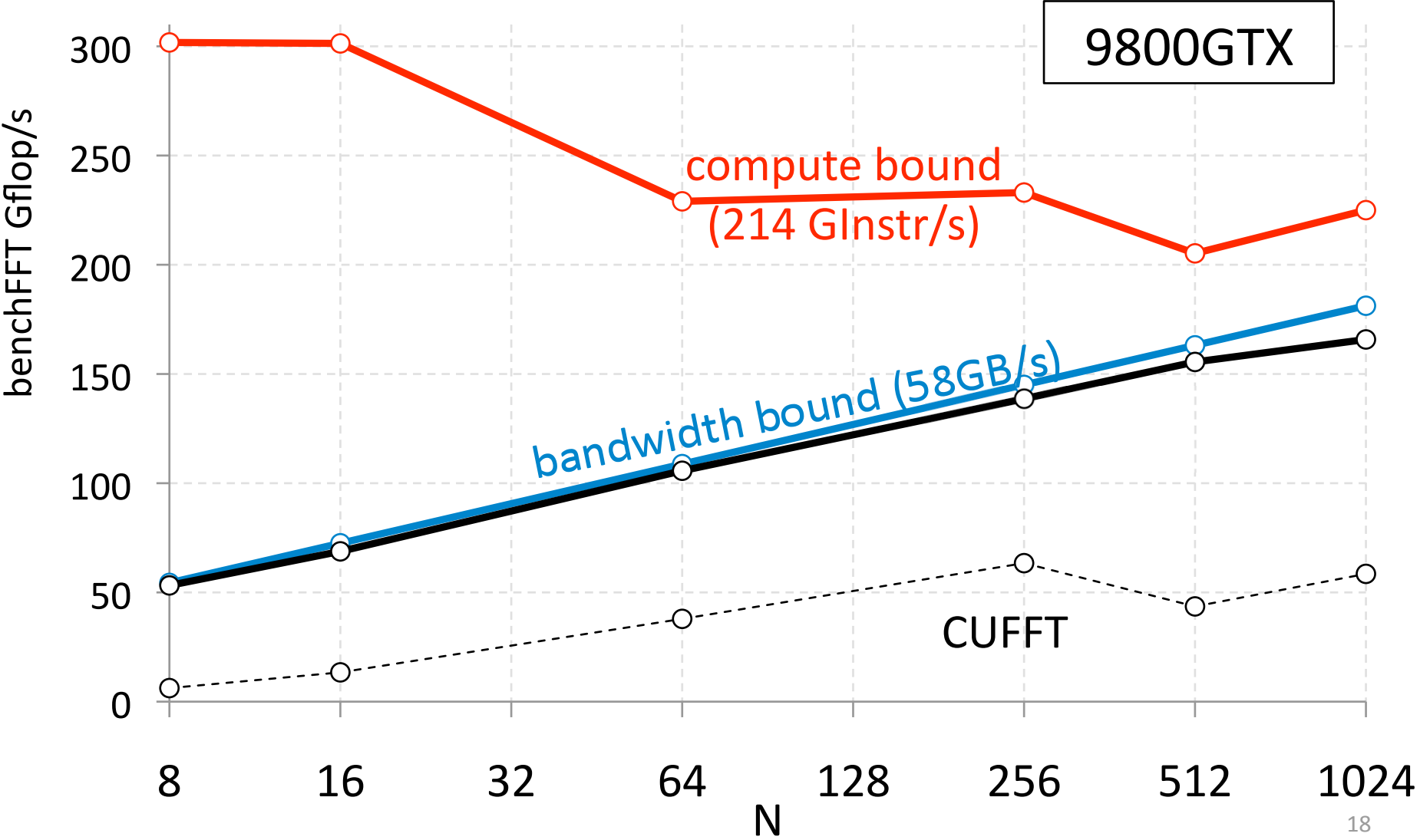
# Similar algorithm in double precision

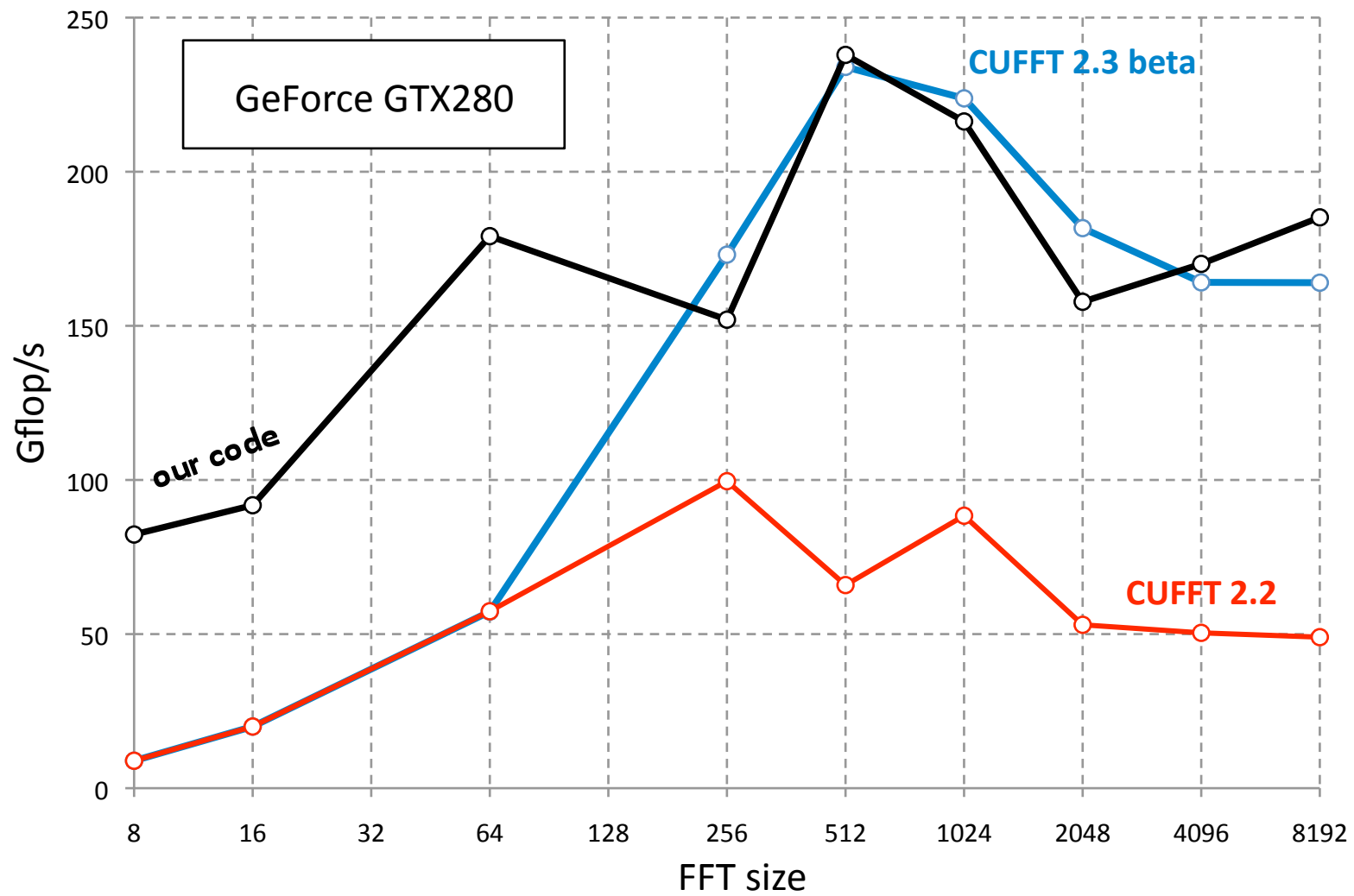


# FFT Performance on NVIDIA 8800GTX

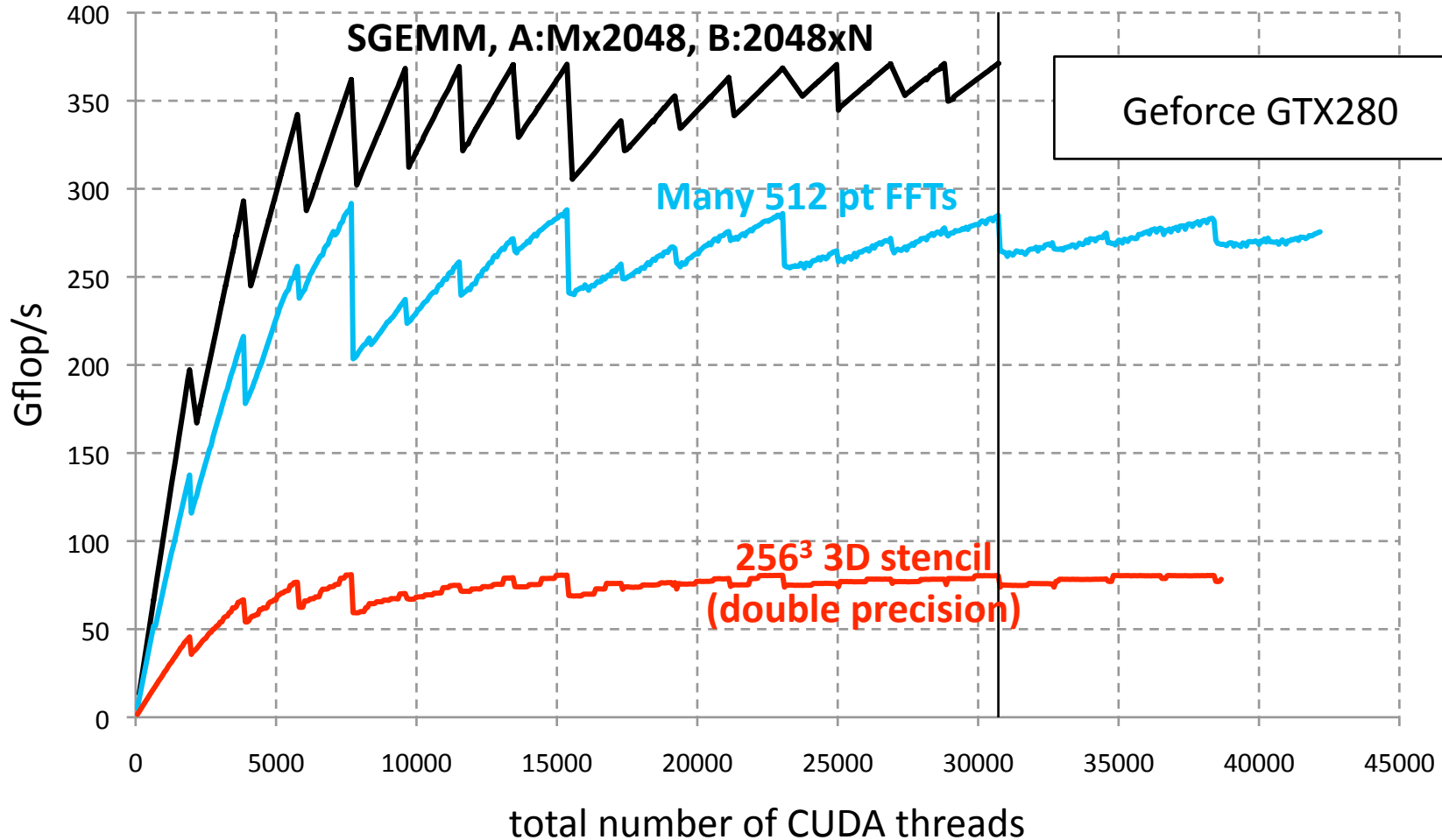


# FFT Performance on NVIDIA 9800GTX





Also used in OpenCL FFT



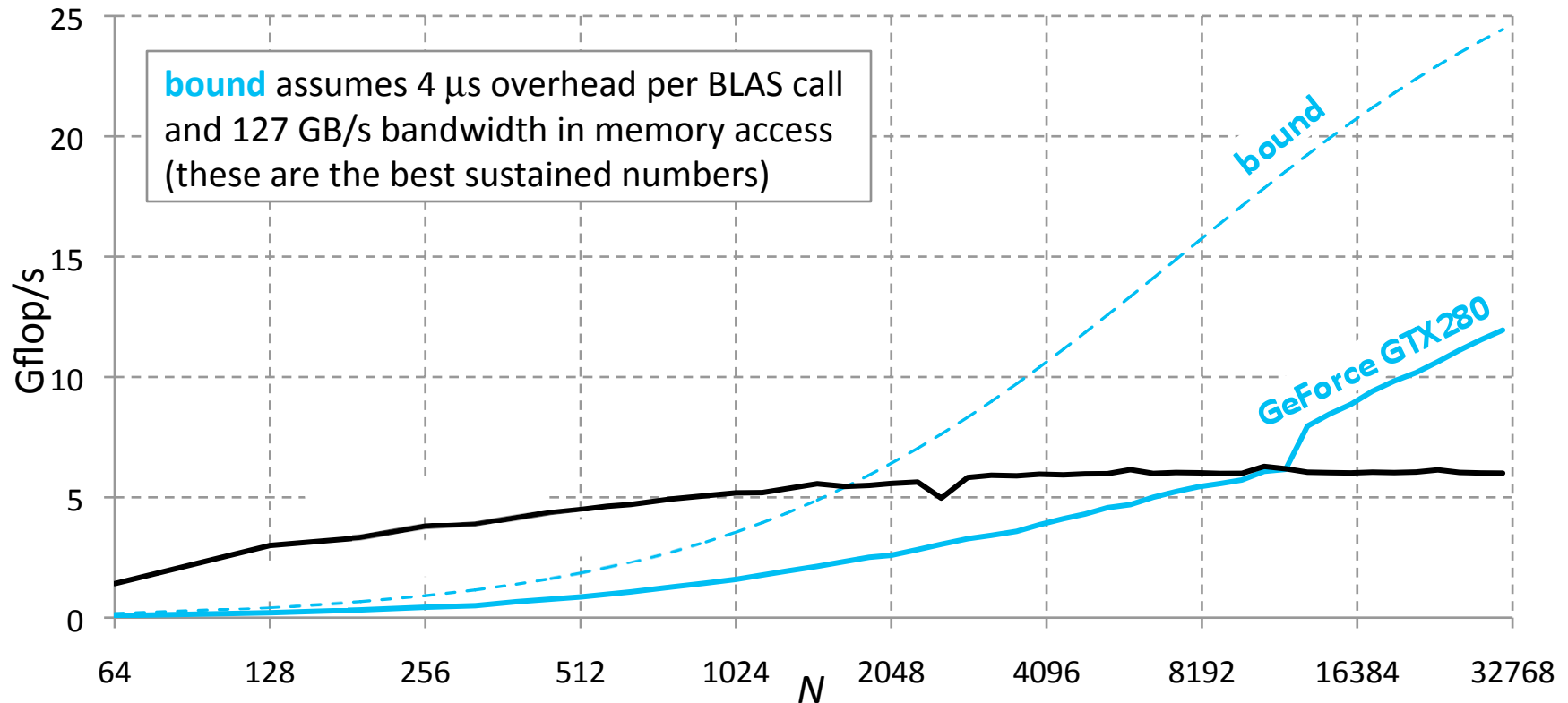
(3D stencil flop count doesn't include CSE done by compiler)  
 Performance doesn't get better after 10,000 threads  
**Running more threads than can fit at a time is not critical**

# Global synchronization

- Global synchronization  $\approx$  launch new kernel
- Launch new kernel  $\approx 3\div 7$   $\mu$ s in overhead
- LU factorization of  $N \times N$  matrix  $\approx 4N$  kernel invocations
  - This is  $12\div 28$  ms for  $1000 \times 1000$  matrix
  - This is  $24\div 56$  Gflop/s upperbound
  - But you get 50 Gflop/s on quad-core CPU
- May not worth implementing on GPU

# Panel Factorization

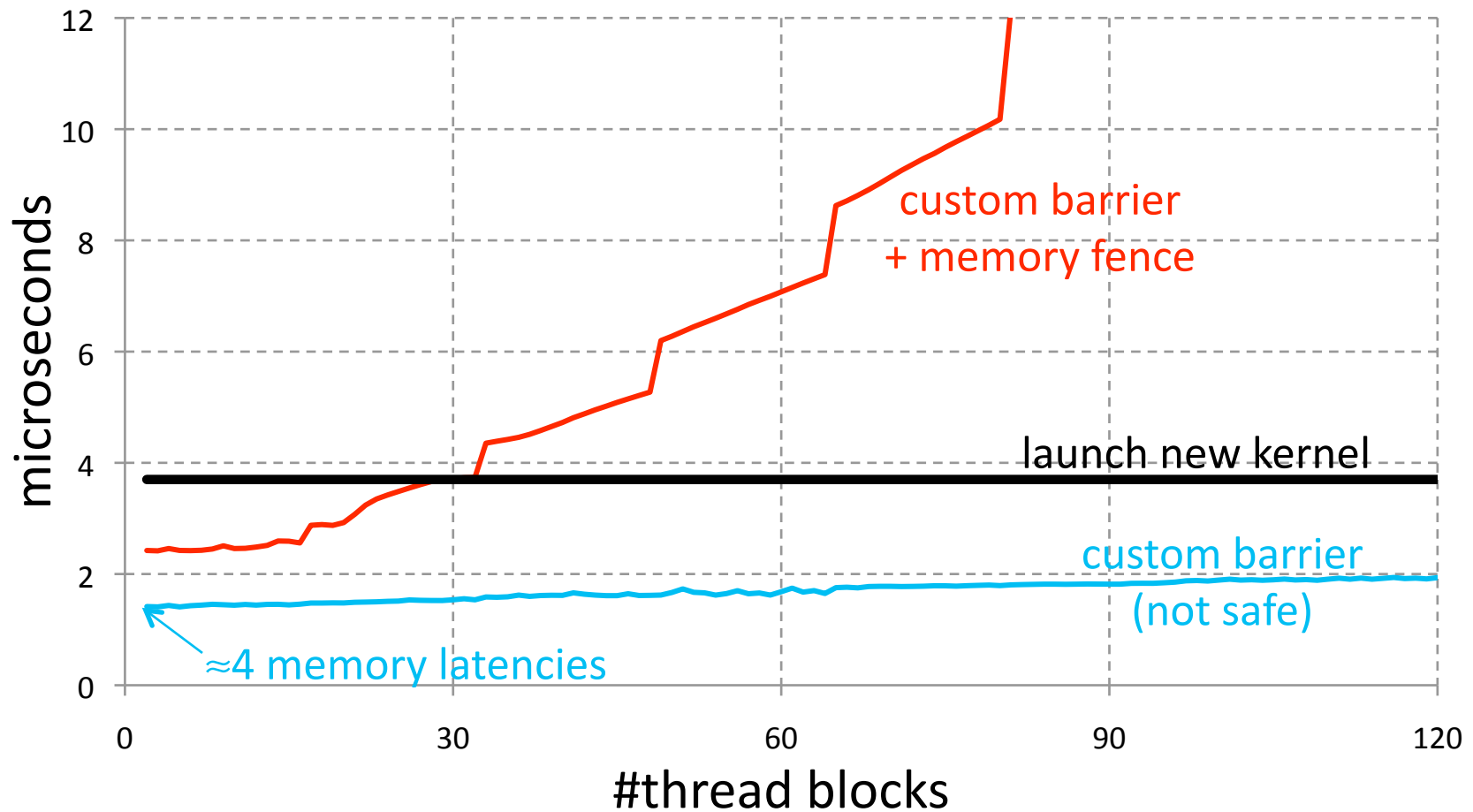
Factorizing  $N \times 64$  matrix in GPU memory using LAPACK's SGETF2:



- When optimizing CPU-GPU communication, mind:
  - Not only #bytes transferred
  - But also #kernels called

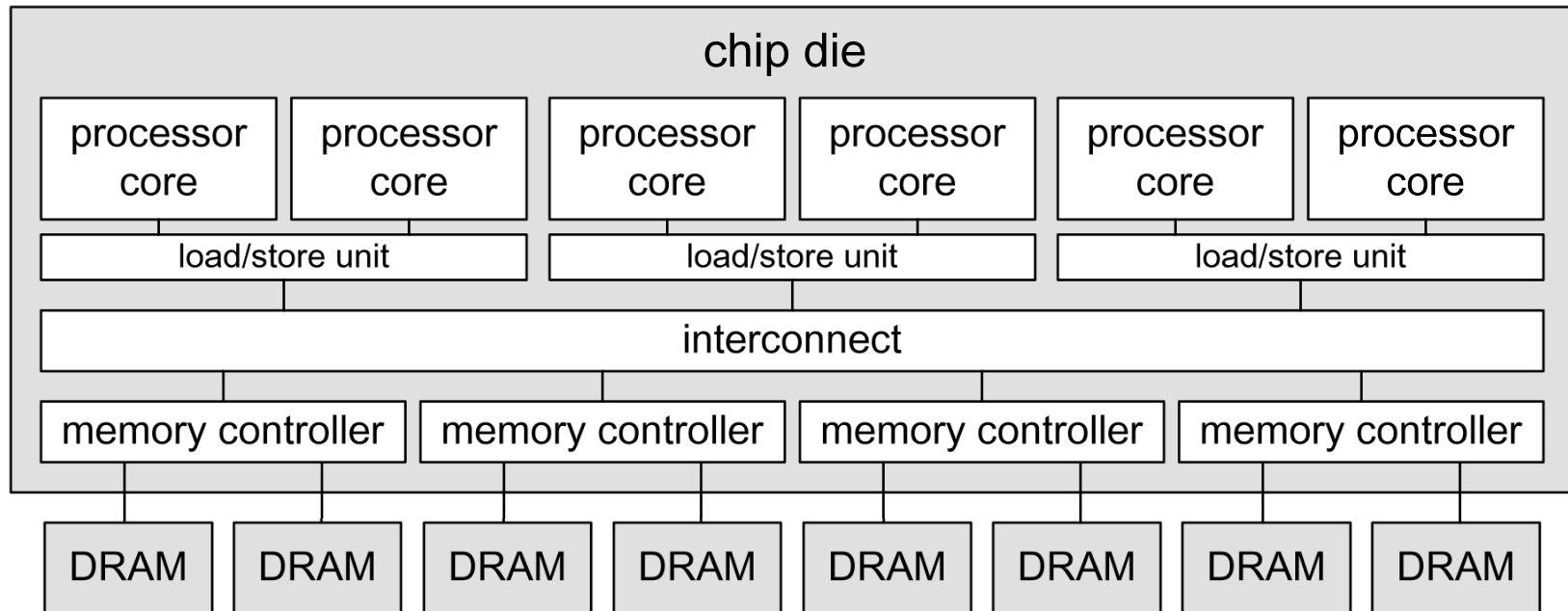
# Alternative global synchronization

- 3  $\mu\text{s}$  is 10x memory latencies – can do better?
  - Why not do all work in one kernel?
- Threads can globally communicate via DRAM
- Can implement custom barrier
  - Requires no atomic operations
  - Requires memory consistency
    - Memory fence will do (available since CUDA 2.2)



- Trends: fast on-chip global synchronization
  - Coherent caches on Intel Larrabee
  - ATI GPUs have global shared memory (GDS)

# Fast on-chip communication?



- GPU has a memory crossbar anyway
  - Can we use it for on-chip global communication?