# Using Register Files, Strip Mining and Global Synchronization on GPUs

Vasily Volkov and James Demmel

UC Berkeley

- Our goal: achieve peak performance on GPU
  - Where does the time go?
- One of the bottlenecks is memory:

| | Intel Q9550 | NVIDIA GTX280 | Potential speedup |
|---|---|---|---|
| Memory interface | 11 GB/s | 141 GB/s | 13x |
| Compute* | 91 Gflop/s | 936 Gflop/s | 10x |
| Flops/word | 34 | 27 | |

- Memory-bound unless reuse data 30 times
  - Reuse data in on-chip memory

*all Gflop/s numbers here and below are for single precision unless noted otherwise

- On-chip memory on CPU:

| | Intel Q9550 | |
|---|---|---|
| SSE registers | 256 B/core | |
| L1 cache | 32 KB/core | x100 of the above |
| L2 cache | 12 MB/die | x100 of the above |

- Memory bottleneck requires large blocks
  - Keep them in cache
- But cache is not fast enough
  - Apply same idea recursively
  - Use (smaller) blocks in faster registers

- GPU memory hierarchy:

| | NVIDIA 8800GTX (per multiprocessor) | NVIDIA GTX280 (per multiprocessor) | ATI HD4890 (per SIMD) |
|---|---|---|---|
| Registers | 32 KB | 64 KB | 256 KB |
| Shared memory | 16 KB | 16 KB | 16KB |

- Register file is larger
  - Keep blocks in register if possible
- Registers are distributed memory
  - Harder to program
  - Use shared memory for communication

# unrolling

```
for( int i = 0; i < n; i++ )
{
    y[i] += a*x[i];
}
```

Unroll 4x

- 4x fewer iterations
- 4x more work per iteration

```
for( int i = 0; i < n; i+=4 )
{
    y[i+0] += a*x[i+0];
    y[i+1] += a*x[i+1];
    y[i+2] += a*x[i+2];
    y[i+3] += a*x[i+3];
}
```

# stripmining

```
parallel for( int i = 0; i < n; i++ )//thread block of size n
{
    y[i] += a*x[i];
}
```

Stripmine 4x

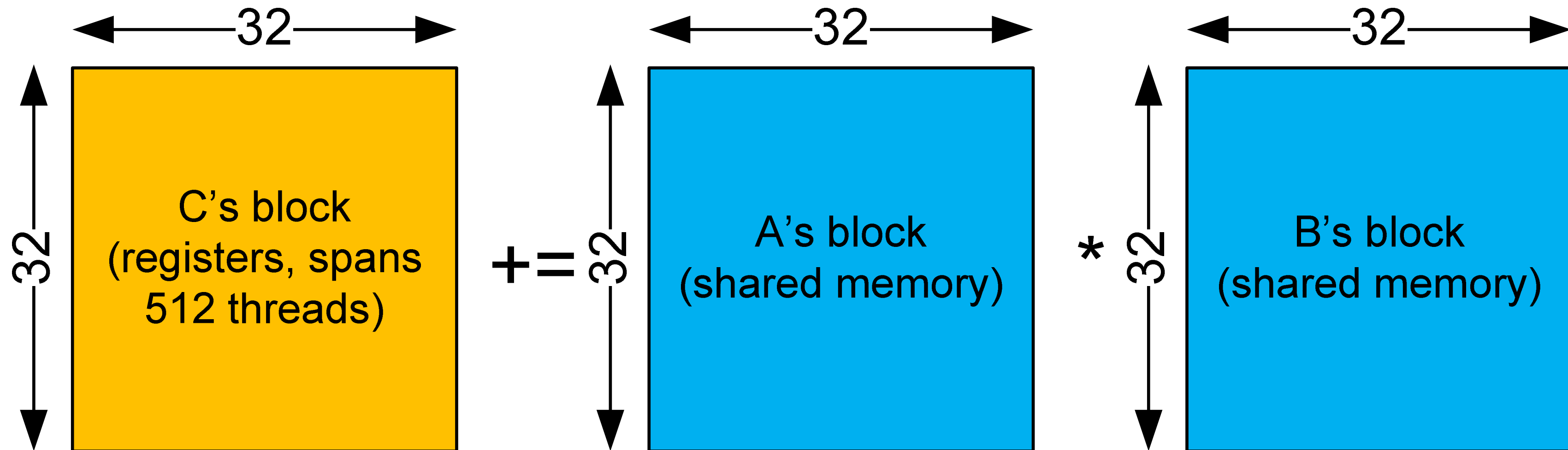
- 4x fewer threads
- 4x more work per thread

```
parallel for( int i = 0; i < n; i+=4 )
{
    y[i+0] += a*x[i+0];
    y[i+1] += a*x[i+1];
    y[i+2] += a*x[i+2];
    y[i+3] += a*x[i+3];
}
```

# Advantages of strip-mining

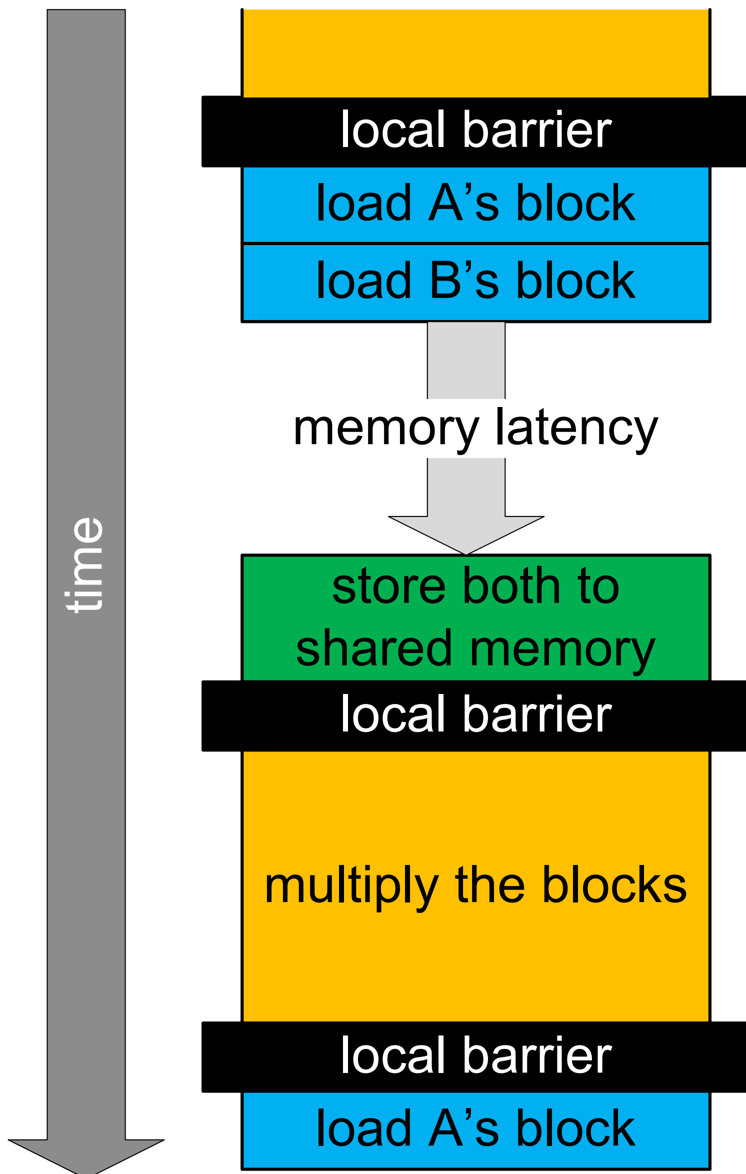
- Do same work using fewer threads
  - Wastes fewer registers
    - Threads eat registers pretty fast
  - So, can run more concurrent thread blocks
  - Or, use larger register blocks
- Still good enough to hide latency
  - Thread parallelism becomes instruction parallelism
  - As long as we have few concurrent thread blocks
- What is the smallest reasonable size of thread block?
  - 64 is enough to give the compute peak in multiply-add

# SGEMM in CUBLAS 1.1



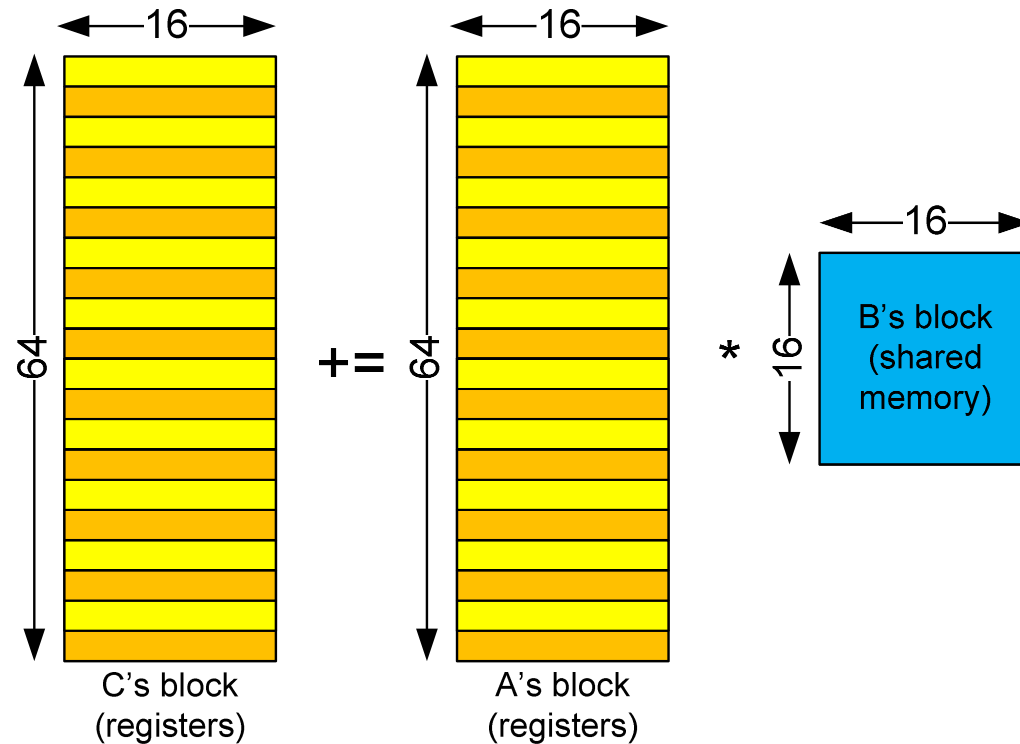

- Problems:
  - Two 32x32 blocks consume half of shared memory
    - Only one thread block per multiprocessor at a time
  - 3 reads from shared memory per 2 multiply-and-adds
    - Bound by shared memory access

# Latency hiding with one thread block




- One thread block/multiprocessor
- Memory access and computation are separated by barrier
- Can't overlap them, no matter how many threads in the block

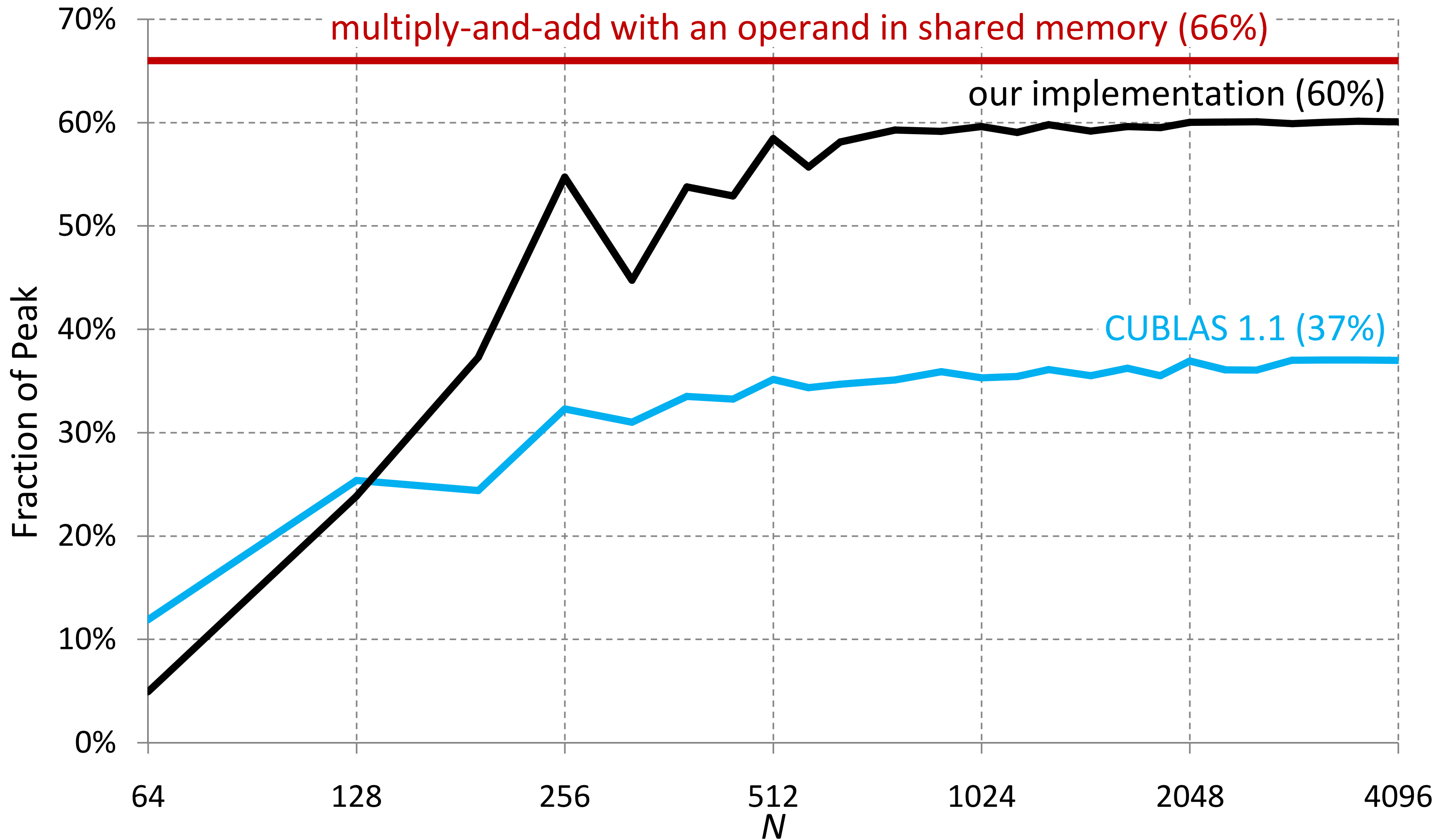
# Our SGEMM




- Blocks in A and C are row-cyclic distributed across threads
  – Don't need to pass A's block via shared memory
- Low usage of shared memory
- Can run many concurrent thread blocks

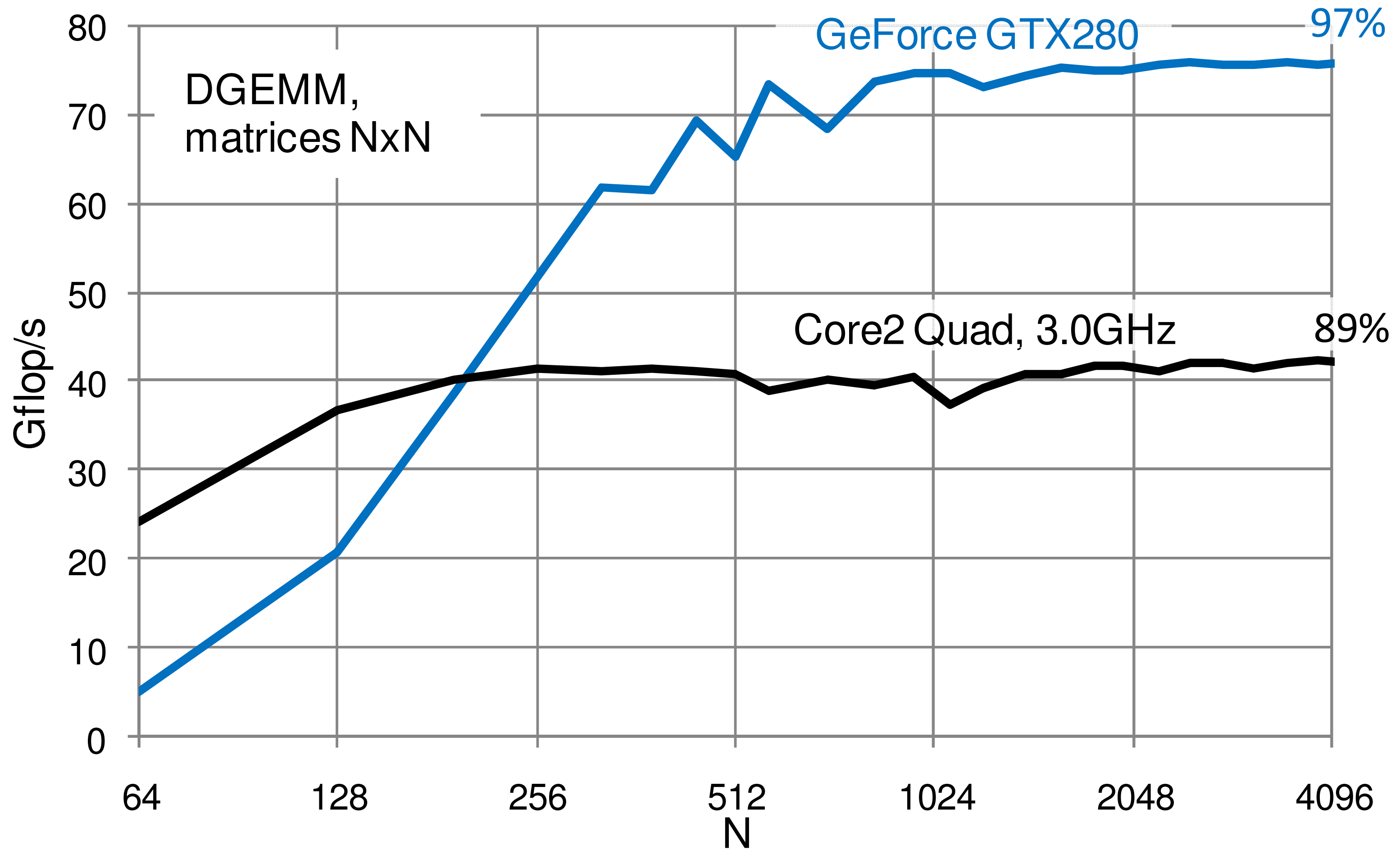
# Our code vs. CUBLAS 1.1

Performance in multiplying two *N*x*N* matrices on GeForce 8800 GTX:




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

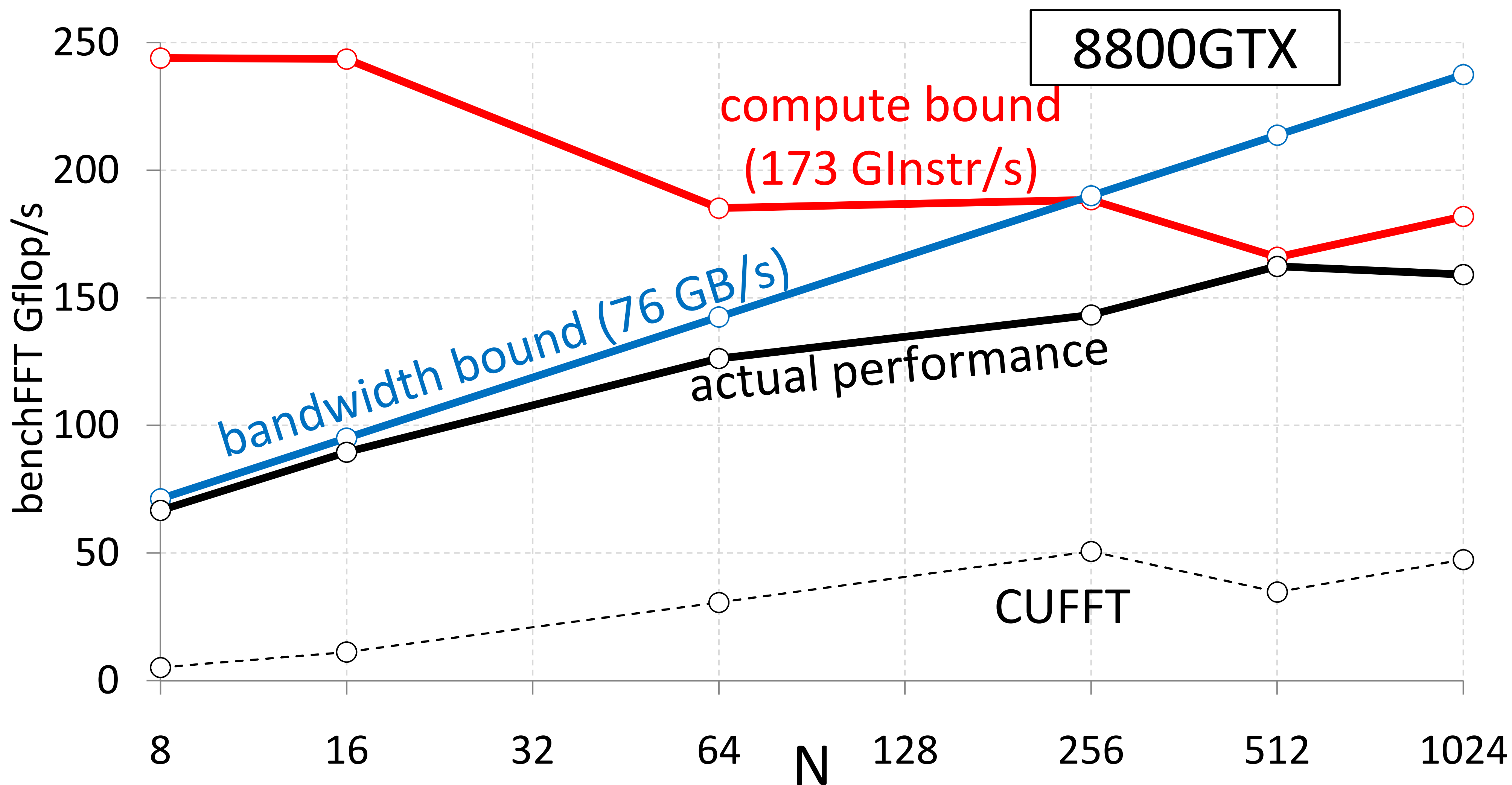
# Similar algorithm in double precision



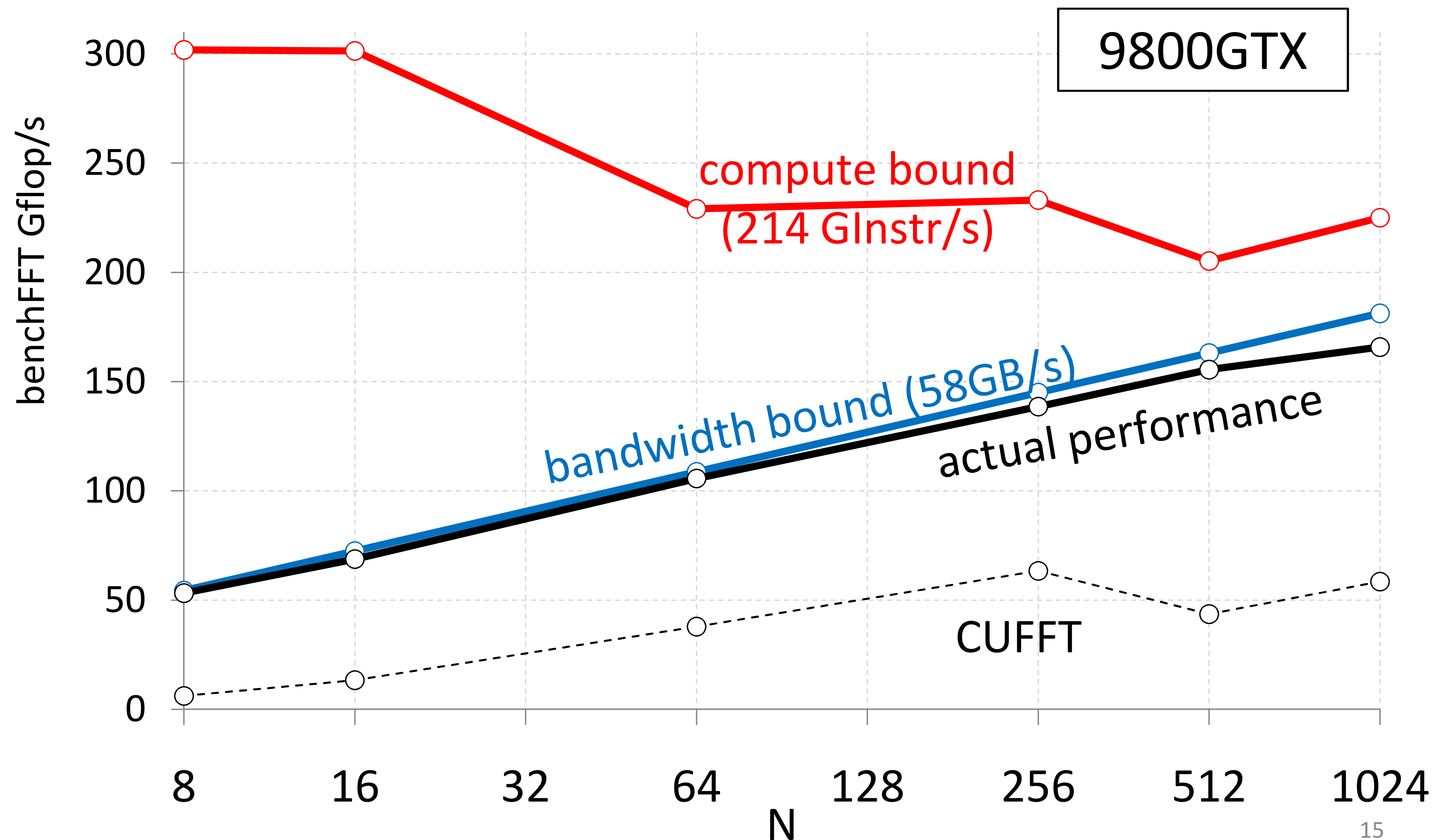
# Batches of small 1D FFTs

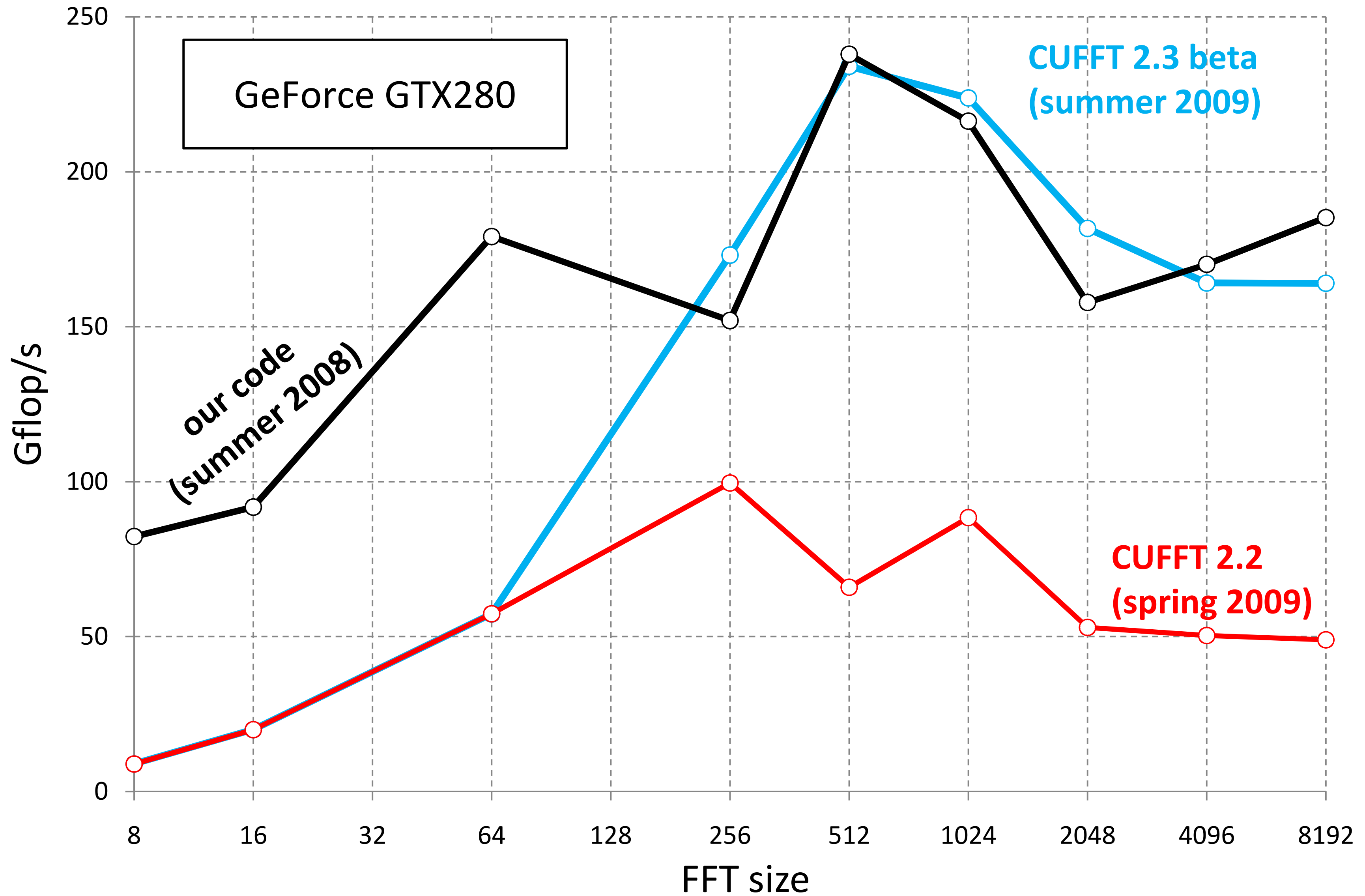
- CUFFT:
  - Keep working set in shared memory
  - Fetch it to registers to do radix-2 or radix-4 FFT
  - Runs n/2 or n/4 threads for n-point FFT (n is sufficiently small to fit data in shared memory)
- Our solution:
  - Keep all data in registers
  - Use shared memory only for permutations (needs 2x less shared memory)
  - Strip-mine down to 64 threads
  - This enables 8 and 16 pt FFT entirely in registers
  - Only <=2 permutations are needed for n <= 1024

# FFT Performance on NVIDIA 8800GTX



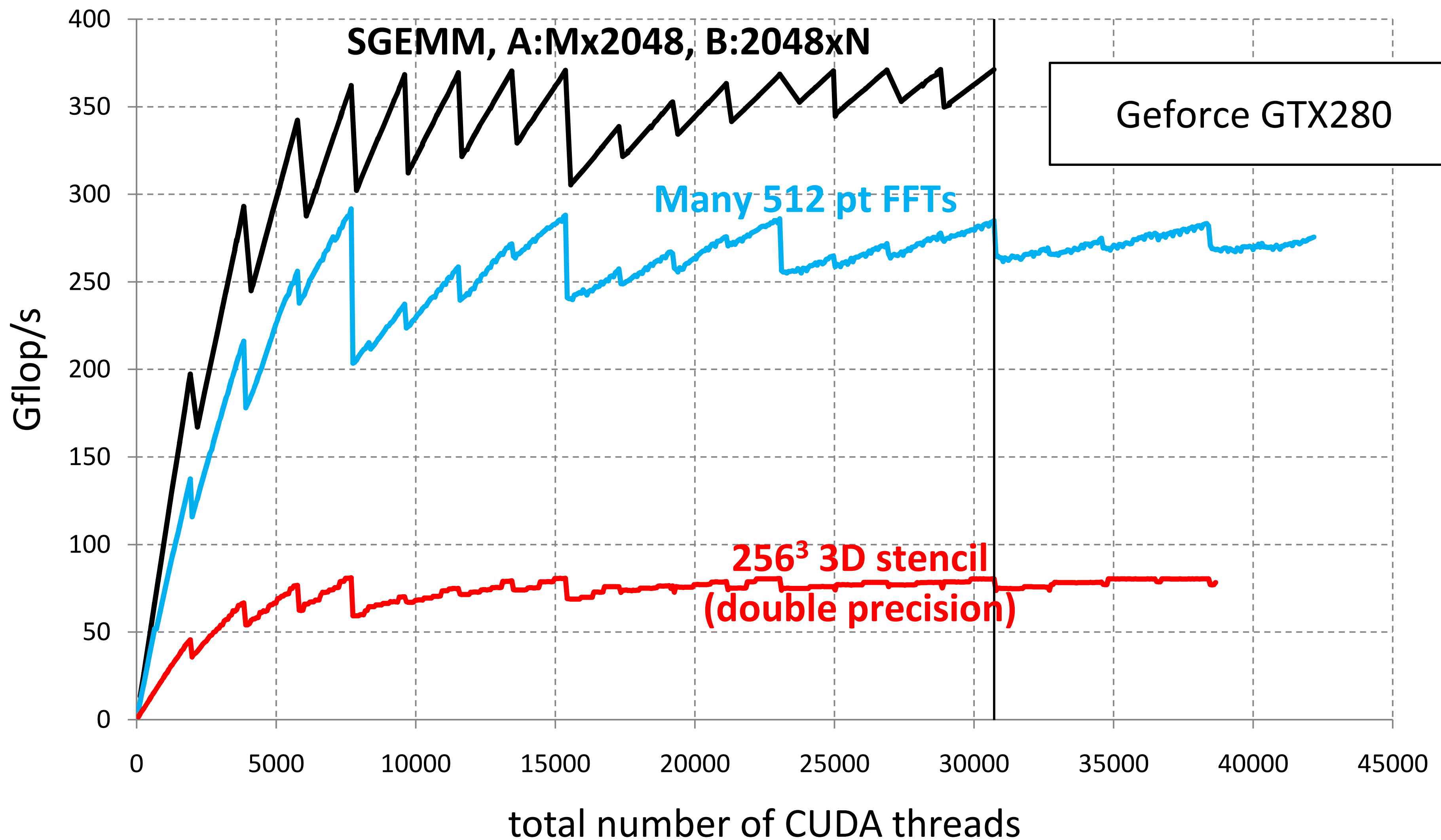
# FFT Performance on NVIDIA 9800GTX






Open source
Adopted by Apple in OpenCL FFT

— How many threads is enough to run fast?
(the more the better?)




(3D stencil flop count doesn't include CSE done by compiler)

Performance doesn't get better after 10,000 threads

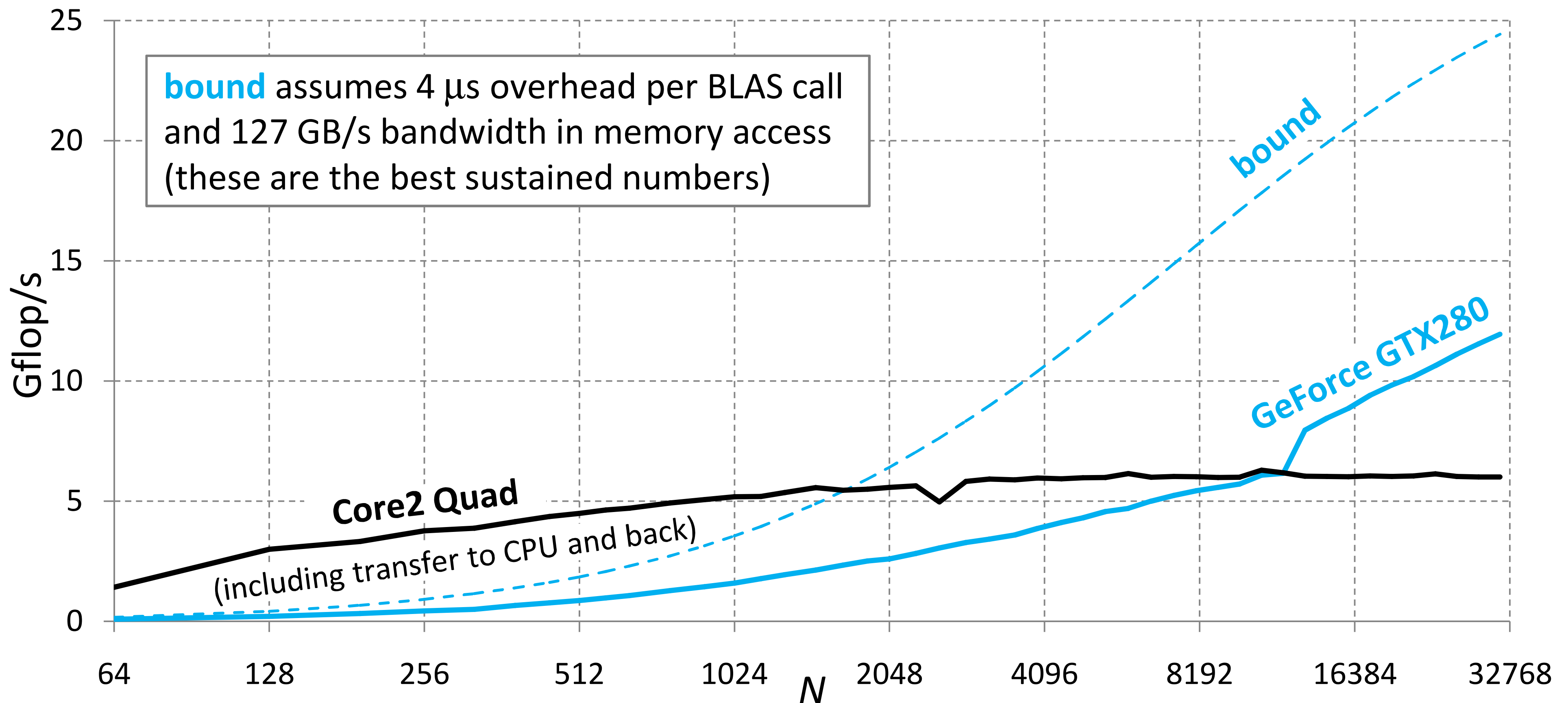
Running more threads than can fit at a time is not critical

# Global synchronization

- Global synchronization ≈ launch new kernel
- Launch new kernel ≈ 3÷7 µs in overhead
- LU factorization of $N$x$N$ matrix ≈ 4$N$ kernel invocations
  - This is 12÷28 ms for 1000x1000 matrix
  - This is 24÷56 Gflop/s upperbound
  - But you get 50 Gflop/s on quad-core CPU
- May not worth implementing on GPU

# Panel Factorization

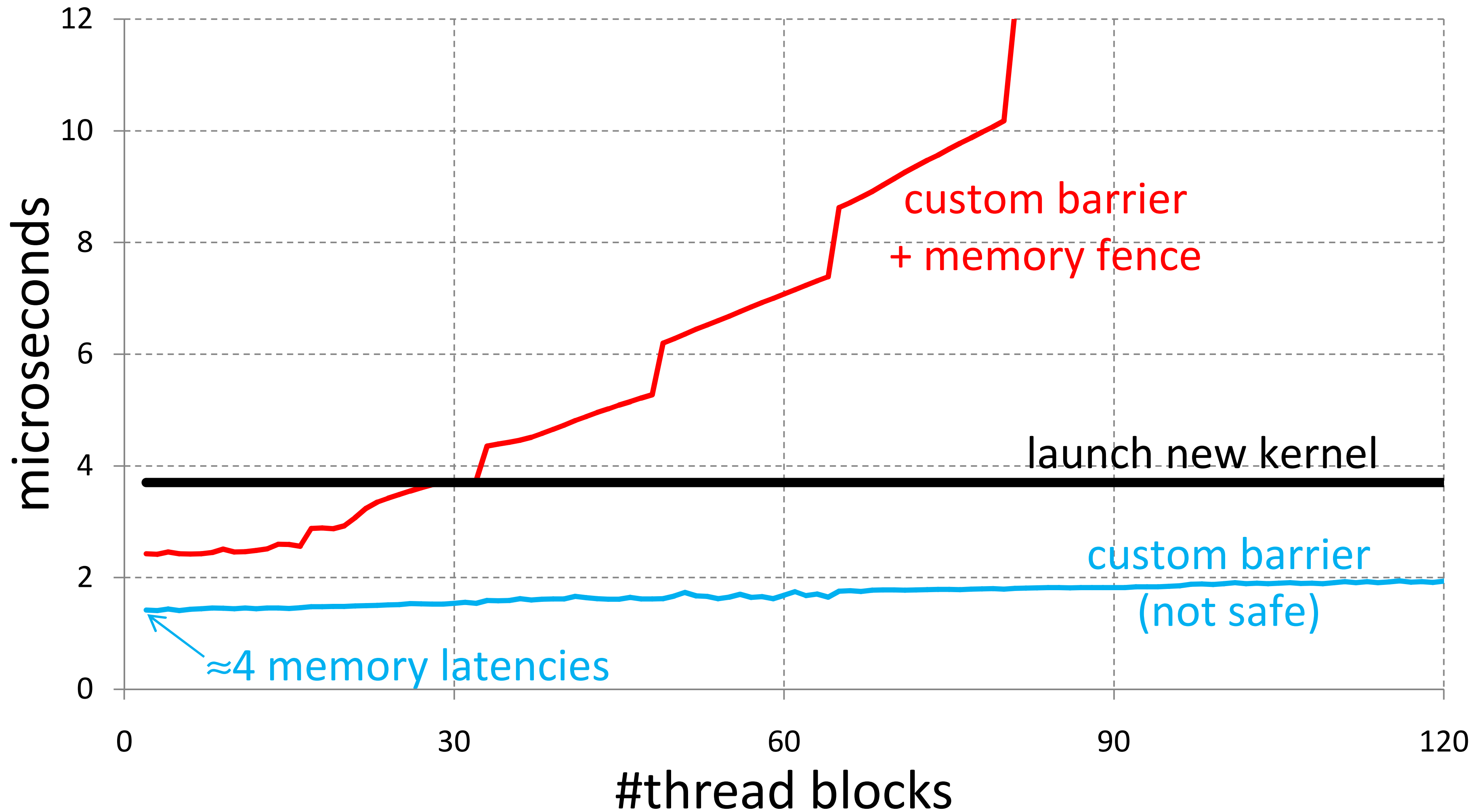
Factorizing *N*x64 matrix in GPU memory using LAPACK's SGETF2:




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

# Alternative global synchronization

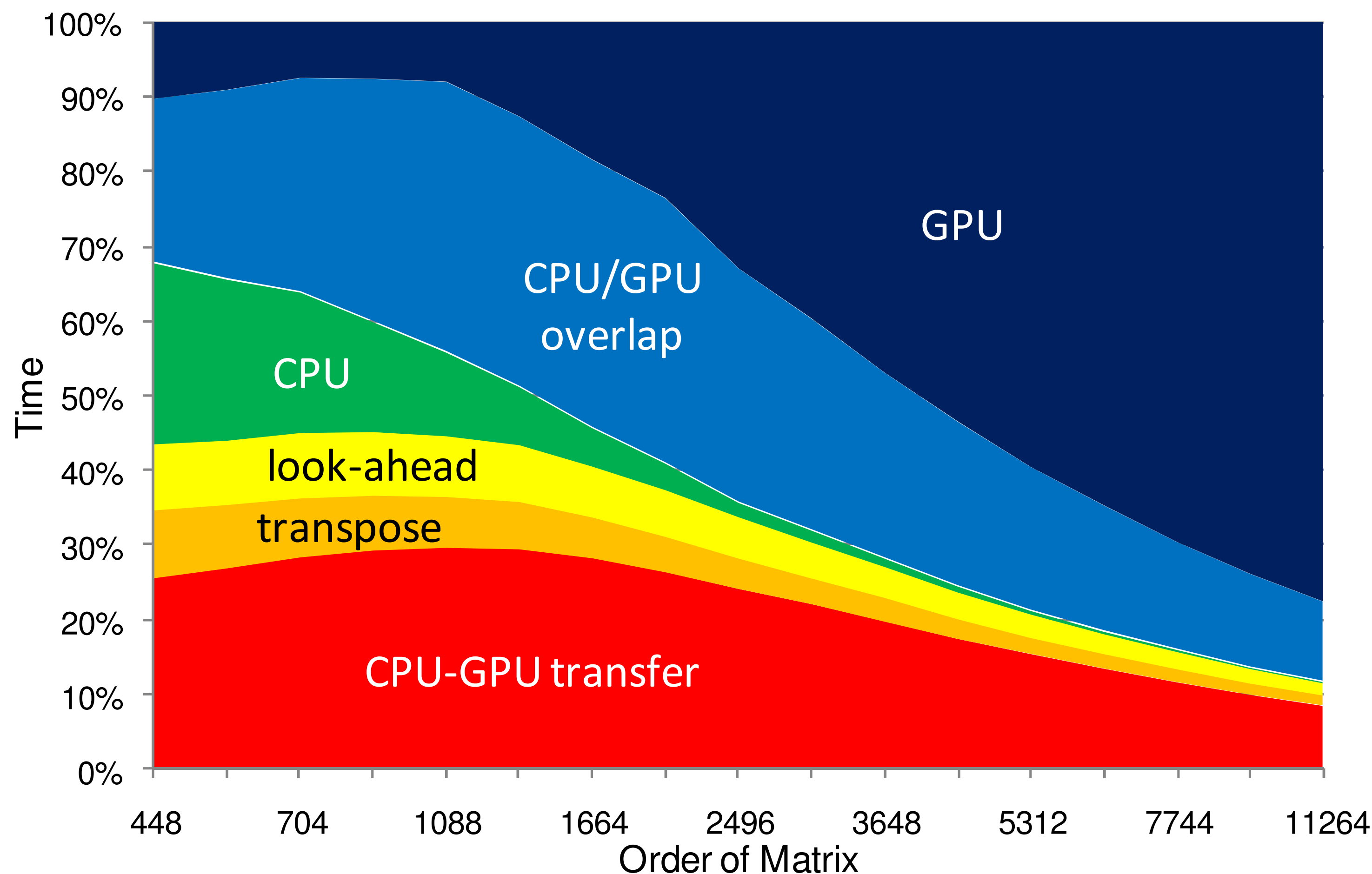
- 3 μs is 10x memory latencies – can do better?
  - Why not do all work in one kernel?
- Threads can globally communicate via DRAM
- Can implement custom barrier
  - Requires no atomic operations
  - Requires memory consistency
    - Memory fence will do (available since CUDA 2.2)




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

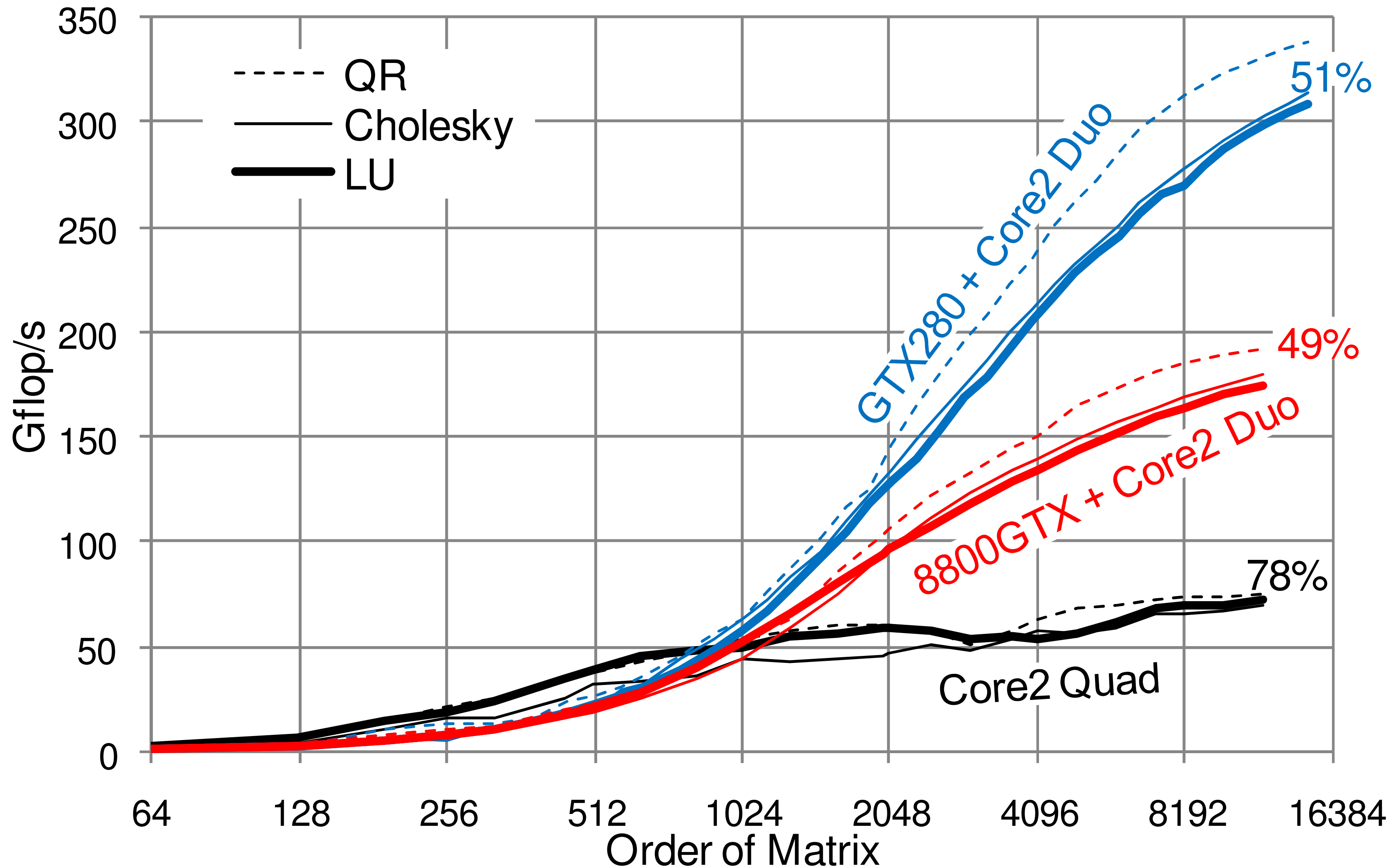
# Extra Slides

# Time Breakdown for LU on GeForce 8800 GTX




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

# Performance Results




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

Open-source