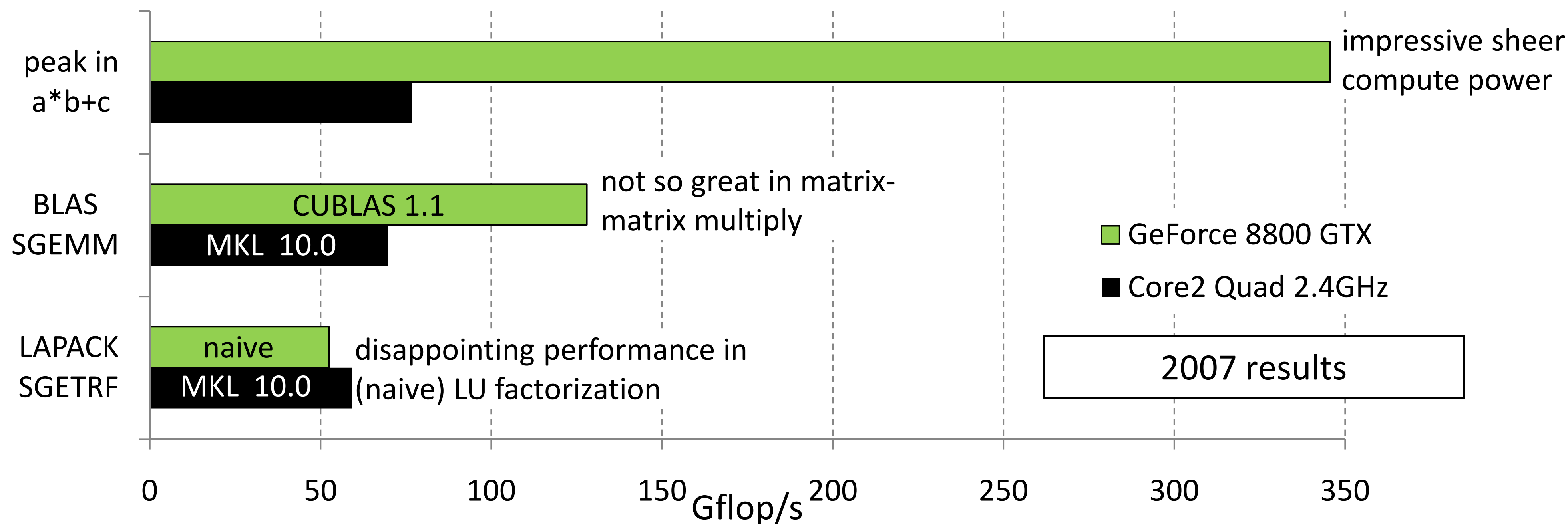
Understanding performance bottlenecks in numerical kernels on GPUs

Vasily Volkov
University of California, Berkeley
May 21, 2010

- LU factorization
- Matrix multiplication

Motivation

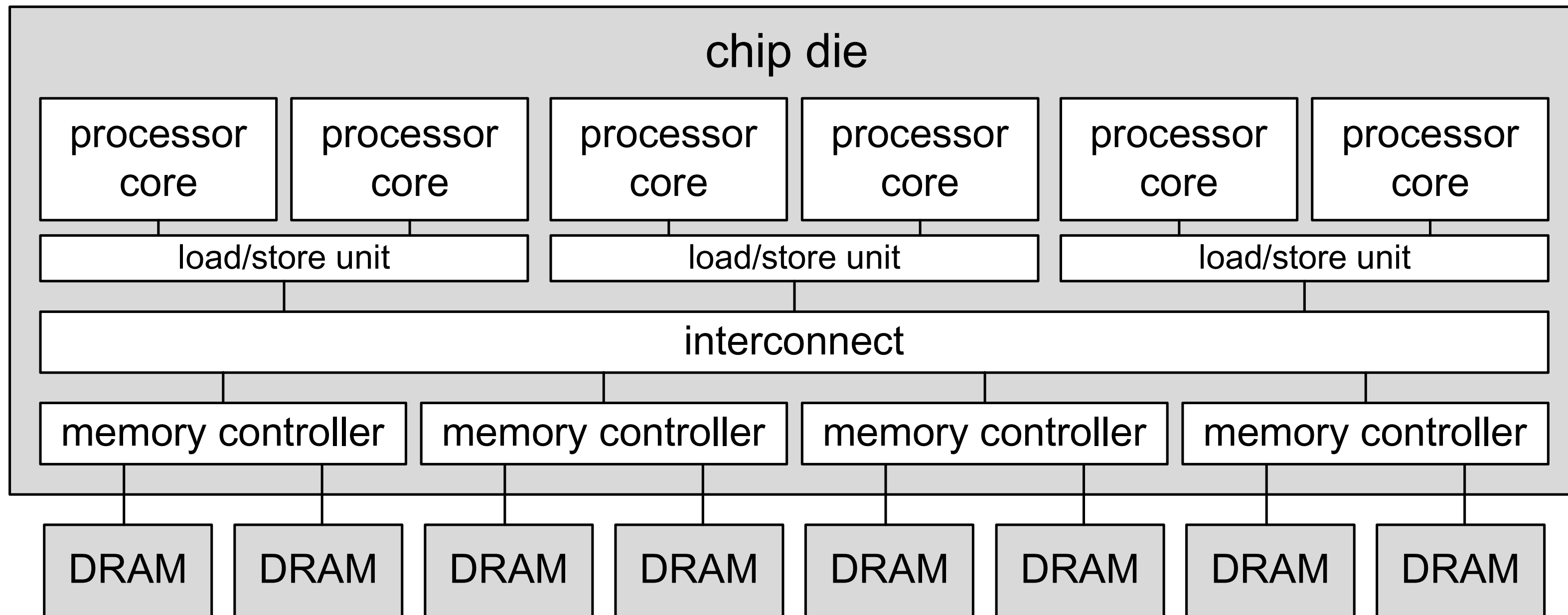
- NVIDIA released CUBLAS 1.0 in 2007, which is BLAS for GPUs
- This enables porting LAPACK to GPU in an easy and obvious way
- (We are concerned with single precision only throughout the talk)



- GPUs have many more ALUs on chip, thus higher sheer compute capacity
- Keeping ALUs busy is a challenge
- **Goal: understand bottlenecks in the dense linear algebra kernels**
  - This requires detailed understanding of the GPU architecture

GPU is a Chip Multiprocessor

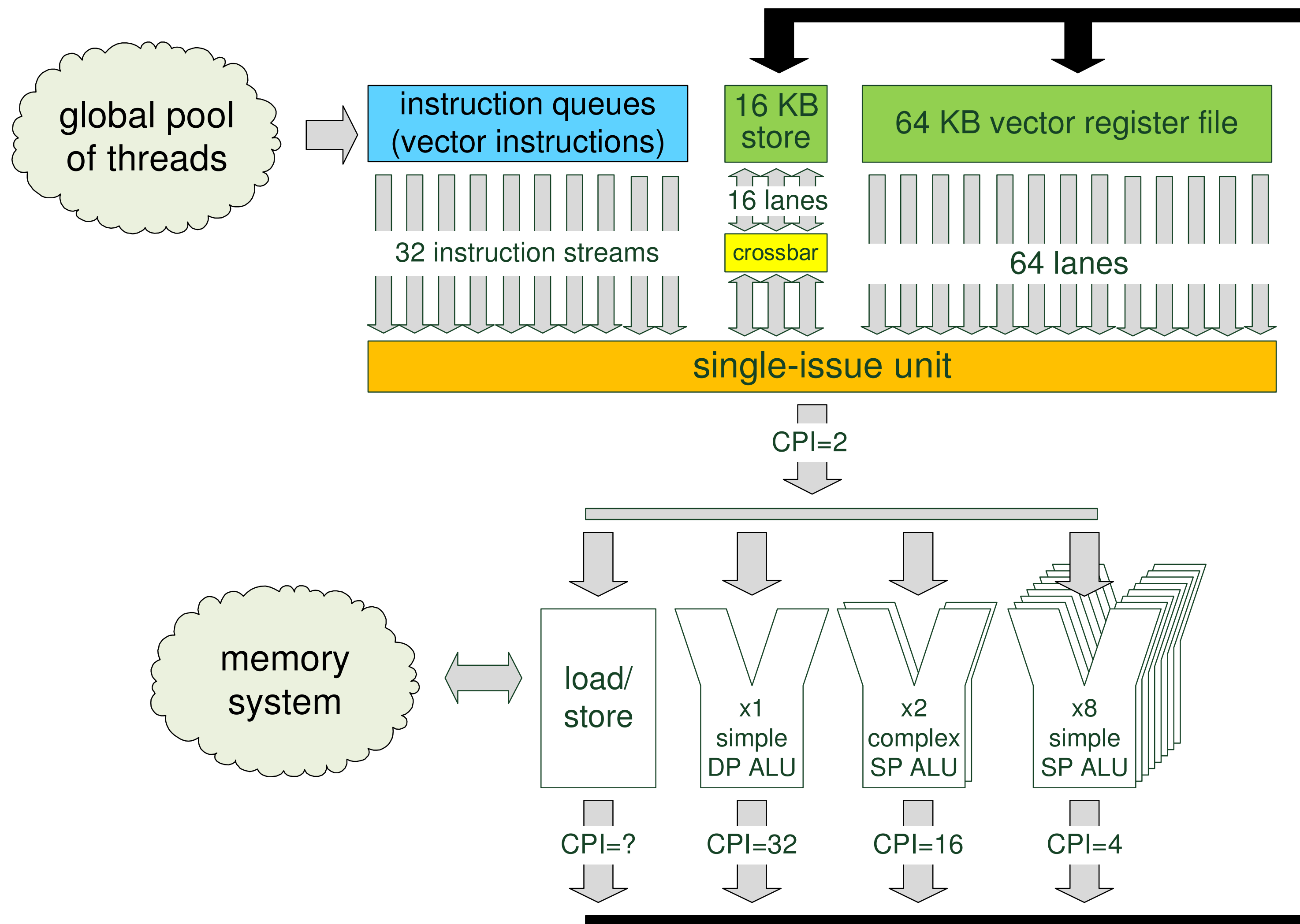
- Multiprocessor on chip = multi-core



- Compute capability scales with number of cores
- Memory bandwidth scales with number of memory controllers

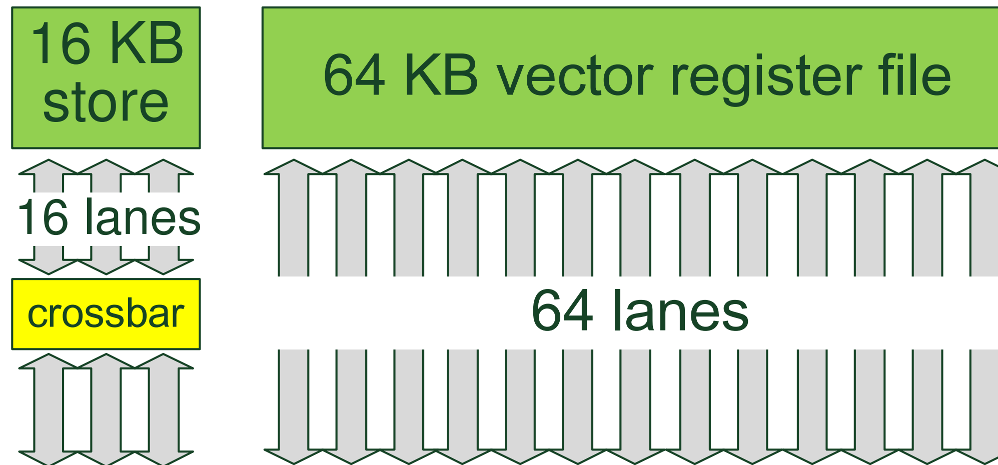
GPU (NVIDIA GeForce)	8600 GTS	9800 GTX	GTX 280
Processor cores	4	16	30
Compute capability (a+b*c)	93 Gflop/s	429 Gflop/s	624 Gflop/s
Memory controllers	2	4	8
Memory bandwidth	32 GB/s	70 GB/s	141 GB/s

GPU Cores are Multithreaded Vector Units



Purely vector ISA	Instructions operate on 32-element vectors
Fine-grain multithreading	32 concurrent instruction streams
“Multiple-issue”	Single issue that is faster (CPI=2) than execution (CPI≥4)

GPU Memory Hierarchy



- Register file is the fastest and the largest on-chip memory
  - **Keep as much data as possible in registers**
  - However, register file is constrained to vector operations
  - Can live with it — vectorized codes are common in HPC
- Shared memory permits indexed and shared access
  - However, it is 2–4x smaller and has 4x lower bandwidth than registers
    - Only 1 operand in shared memory is allowed versus 4 register operands
  - Moreover, some instructions run slower if using shared memory
  - **Use shared memory as a communication device only**
  - Avoid communication to improve performance

GPU Vector Processing

- GPU supports variable vector length (VL), VL=512 is maximum
  - Implemented via strip-mining into independent instruction streams
  - Instruction streams must barrier-synchronize if write to shared memory
  - More physical program counters per longer vectors improves performance in conditional codes
- GPU has no scalar capabilities at all
  - Pointers and scalars in registers have full vector length
  - Each pointer consumes 2KB in register file if VL=512
  - Counter **i** in "**for(int i = 0; i < n; i++)**" consumes 2KB in registers if VL=512
  - Similarly, **i++** translates into 512 increments if VL=512
  - *But only into 64 increments and 256 bytes if VL=64*
- Therefore, **use shorter vectors, not longer**
  - Strip-mine longer vectors into shorter at the program level if necessary
  - E.g. instead of using "**float a;**" and VL=512 use "**float a[8];**" and VL=64
  - Instead of "**a += b;**", VL=512 use "**a[0] += b[0]; …; a[7] += b[7];**", VL=64

Peak Throughput in Multiply-and-Add

- How much parallelism is *enough* to get the peak?

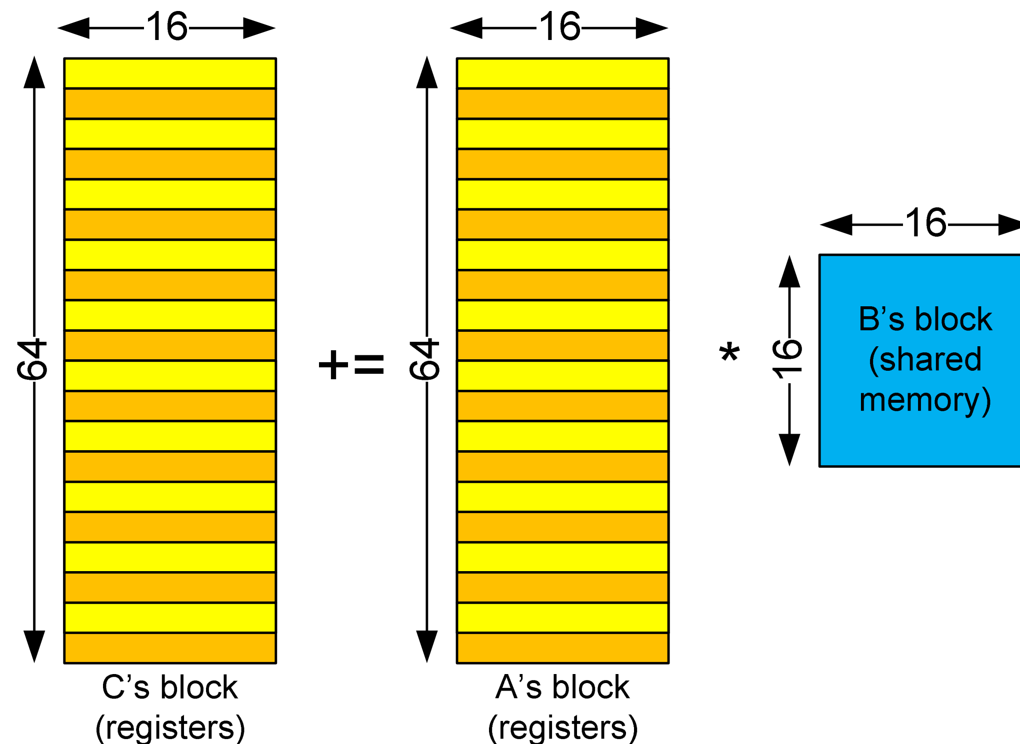
- Run **1 thread per processor core**
  - Purpose: smallest amount that can control all computing resources
- Assume **sufficient instruction-level parallelism** in the program
  - Purpose: hide pipeline latency
- Choose the shortest vector length that yields the peak
  - Purpose: satisfy inherent data-parallelism constraints
- Result: **98% of arithmetic peak at VL = 64**
  - Therefore, VL=64 is recommended for all compute-bound codes

- However, we never could surpass 66% of peak is using an operand in shared memory
  - We believe this is an inherent bottleneck in the architecture
  - We use this number in the throughput bounds below

Matrix-Matrix Multiply: $C = C + A*B$

- GPU requires using block algorithms in matrix-matrix multiply:
  - Peak rates on one of the latest GPUs are 624 Gflop/s and 141 GB/s
  - This corresponds to 0.23 bytes per flop
  - But naïve matrix-matrix multiply requires 4 bytes per flop
  - Thus, it is bandwidth-bound unless data is reused 18 times
  - Using MxN blocks in C yields $2/(1/M+1/N)$ average reuse
- Use vector algorithms to efficiently use vector registers
  - Such as used on IBM 3090 Vector Facility and Cray X1:
  - Keep A's and C's blocks in registers
  - Keep B's block in a shared storage
  - No other sharing is needed if C's height = VL. We know VL=64 is best
- Choose large enough width of C's block
  - 16 is enough as 2/(1/64+1/16) = 26-way reuse
- Choose a convenient thickness for A's and B's blocks

Local data layout in our SGEMM



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

- The resulting matrix-matrix multiply code fits on one slide

```
__global__ void sgemmNN( const float *A, int lda, const float *B, int ldb, float* C, int ldc, int k, float alpha, float beta )
{
    A += blockIdx.x * 64 + threadIdx.x + threadIdx.y*16;
    B += threadIdx.x + ( blockIdx.y * 16 + threadIdx.y ) * ldb;
    C += blockIdx.x * 64 + threadIdx.x + (threadIdx.y + blockIdx.y * ldc ) * 16;
    __shared__ float bs[16][17];
    float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    const float *Blast = B + k;
    do
    {
#pragma unroll
        for( int i = 0; i < 16; i += 4 )
            bs[threadIdx.x][threadIdx.y+i]  = B[i*ldb];
        B += 16;
        __syncthreads();
#pragma unroll
        for( int i = 0; i < 16; i++, A += lda )
        {
            c[0] += A[0]*bs[i][0];    c[1] += A[0]*bs[i][1];    c[2] += A[0]*bs[i][2];    c[3] += A[0]*bs[i][3];
            c[4] += A[0]*bs[i][4];    c[5] += A[0]*bs[i][5];    c[6] += A[0]*bs[i][6];    c[7] += A[0]*bs[i][7];
            c[8] += A[0]*bs[i][8];    c[9] += A[0]*bs[i][9];    c[10] += A[0]*bs[i][10];c[11] += A[0]*bs[i][11];
            c[12] += A[0]*bs[i][12];c[13] += A[0]*bs[i][13];c[14] += A[0]*bs[i][14];c[15] += A[0]*bs[i][15];
        }
        __syncthreads();
    } while( B < Blast );
    for( int i = 0; i < 16; i++, C += ldc )
        C[0] = alpha*c[i] + beta*C[0];
}
```

Compute pointers to the data

Declare the on-chip storage

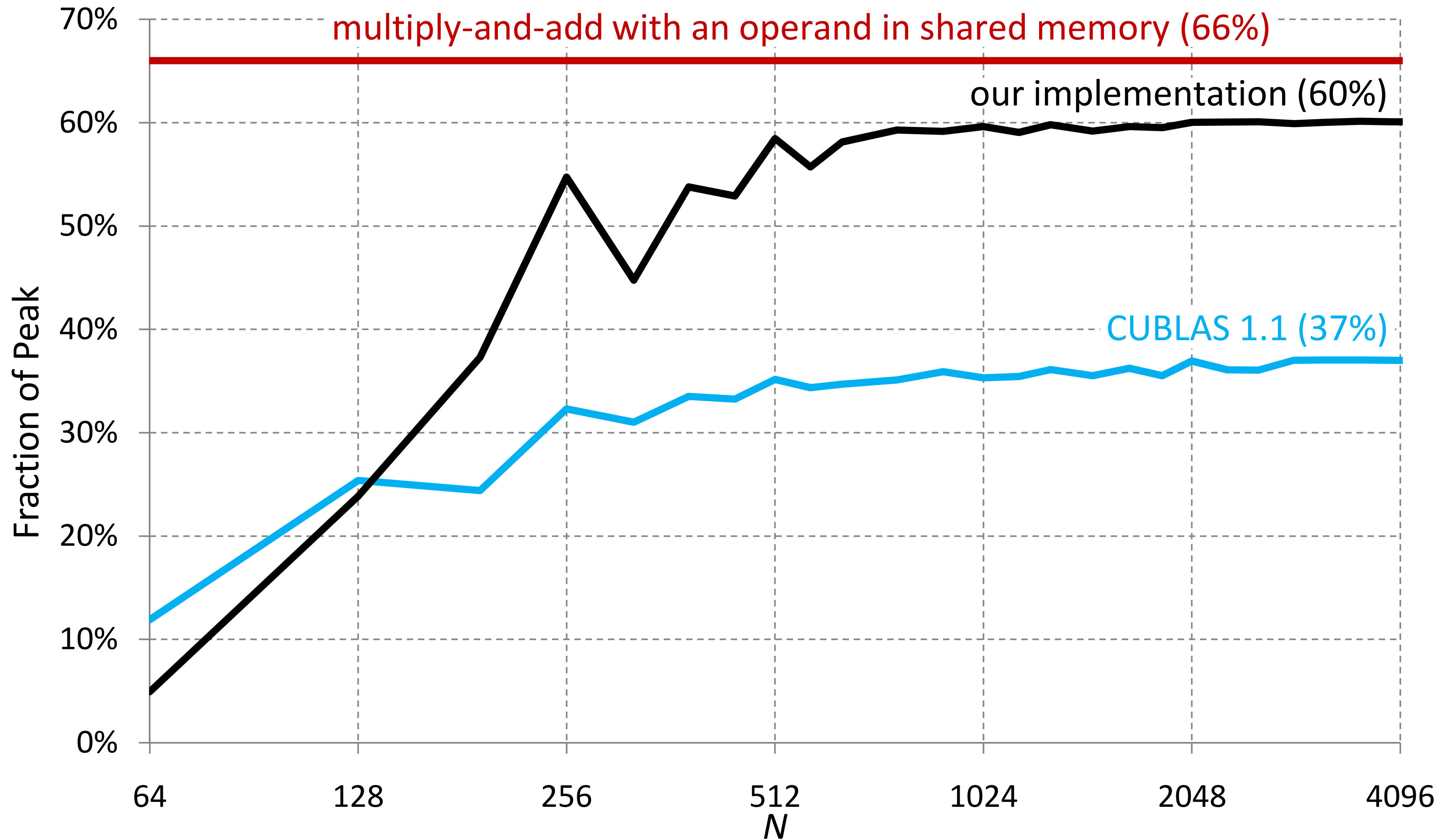
Read next B's block

The bottleneck:
Read A's columns
Do Rank-1 updates

Store C's block to memory

Our code vs. CUBLAS 1.1

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



What causes CUBLAS 1.1 to run slower than our code?

Our code vs. CUBLAS 1.1

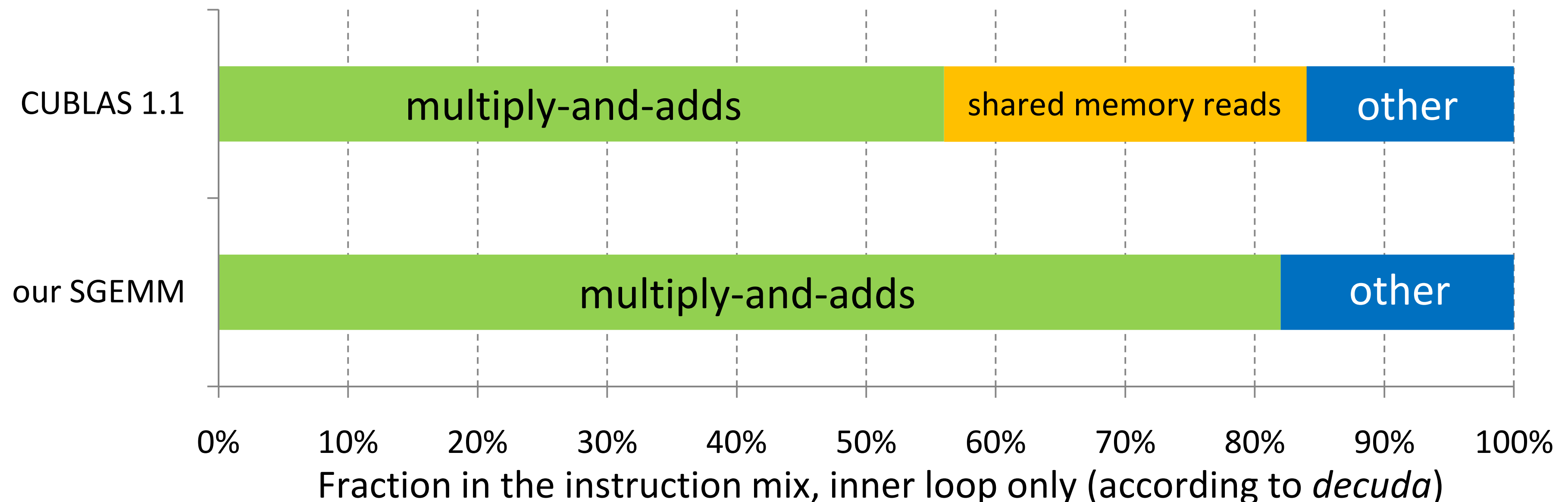
- Popular GPU programming guidelines recommend:
  - Minimize use of registers
  - Maximize use of shared memory
  - Use longer vectors
  - Maximize occupancy (number of concurrent instruction streams)
- CUBLAS 1.1 succeed in following all of them, but loses in performance:

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

- Can those guidelines be wrong?
- Note that both codes do the same amount of work per thread
  - 2048**K* flops per thread if multiplying *M*x*K* matrix by *K*x*N* matrix

Our code vs. CUBLAS 1.1

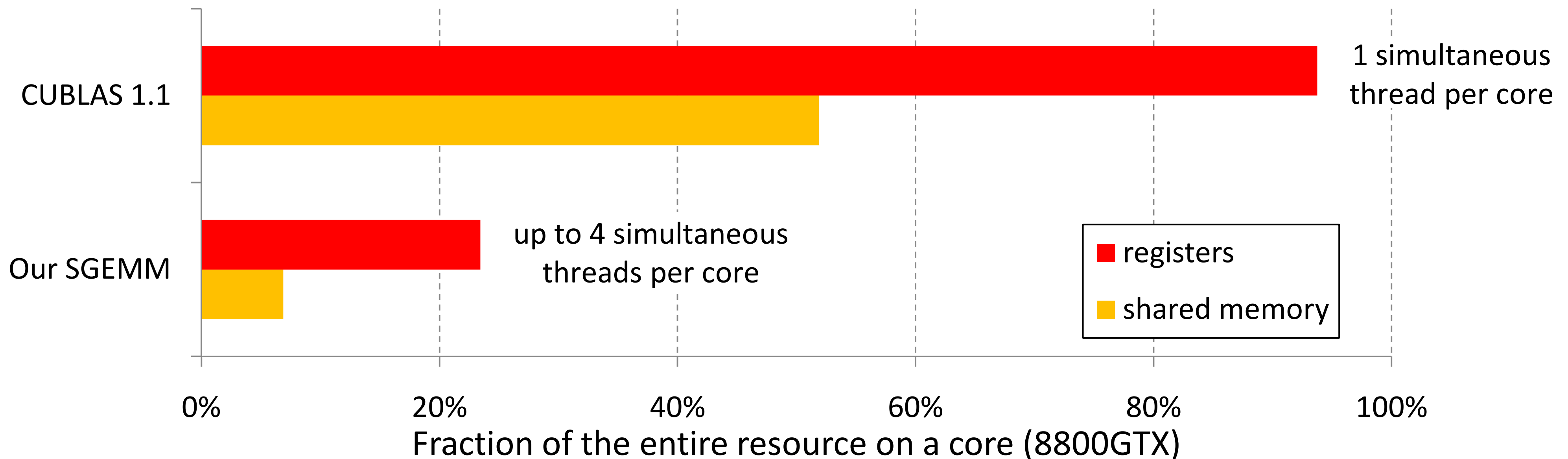
- Ideally, the code is instruction throughput bound
- Then the performance in Gflop/s is bound by the instruction mix



- Intensive use of shared memory in CUBLAS translates into instructions that move data, not compute
- This consumes cycles unless overlapped with computation
  - Does not overlap on pre-GTX280 GPUs
  - (Shared memory reads on GTX280 can go via the transcendental ALU)

Our code vs. CUBLAS 1.1

- Thread-level concurrency is helpful in hiding memory latency
- The concurrency is bound by the resource usage (less usage is better)



- CUBLAS uses space-inefficient square blocks in *A* and *B*
- CUBLAS uses fewer but longer vectors (VL = 512)
  - 3 pointers to the matrices and 1 loop counter alone consume 25% of the register file on 8800GTX

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

Register usage in our SGEMM

warp 1	warp 2
A's pointer	
B's pointer	
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
A's pointer	
ldb*12	ldb*12
ldb*4	ldb*4
ldb*8	ldb*8
*****	*****
*****	*****
*****	*****
*****	*****
*****	*****
*****	*****
*****	*****

Run fewer threads

Half registers are used for working set

Fewer threads – fewer registers wasted

Threads vs. Instruction Streams

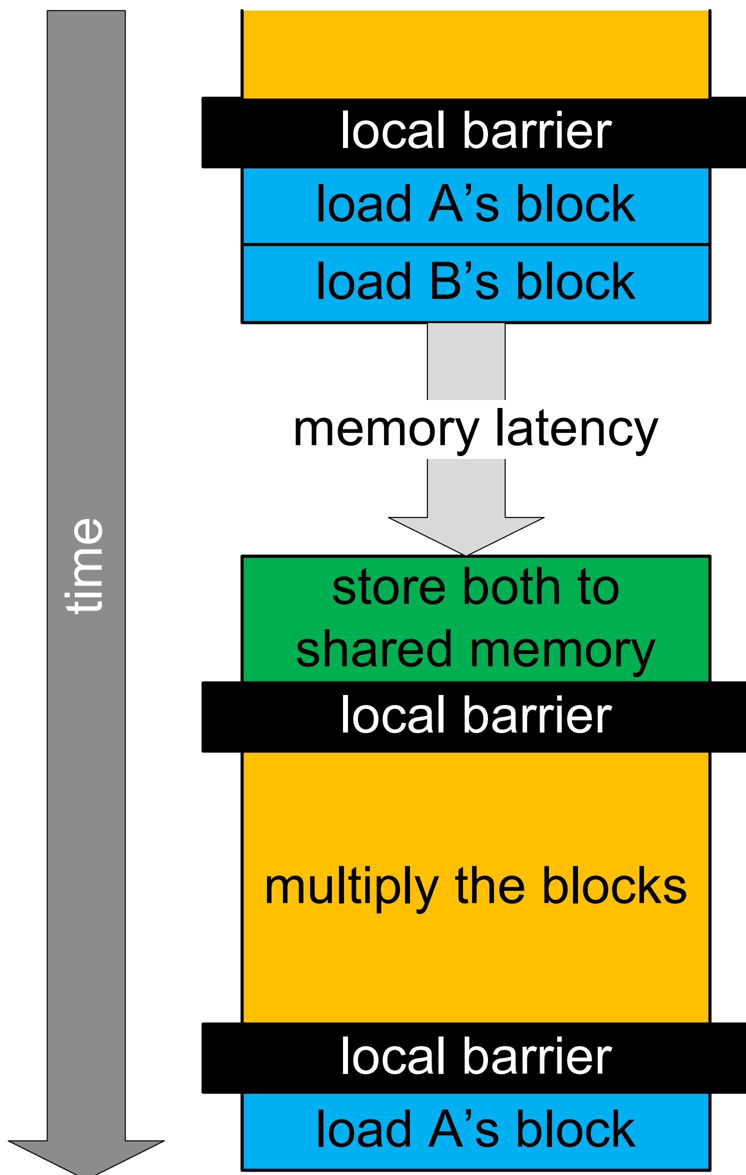
- CUBLAS 1.1 permits more concurrent instruction streams, but fewer concurrent threads:

	CUBLAS	Our code			
Threads/core	1 (max)	1	2	3	4
Instruction streams/core	16	2	4	6	8
Performance (8800 GTX)	37%	32%	49%	57%	60%
Theoretical bound (Instruction throughput, 8800 GTX)	44%	60%			

- Parallelism between asynchronously running threads is important
- Parallelism between tightly synchronized instruction streams within a thread is less important

- 3 threads/core are nearly enough to hide memory latency
  - This is 6 instructions streams/core or 33% occupancy on 8800 GTX

Is # of thread blocks important?



CUBLA SGEMM again

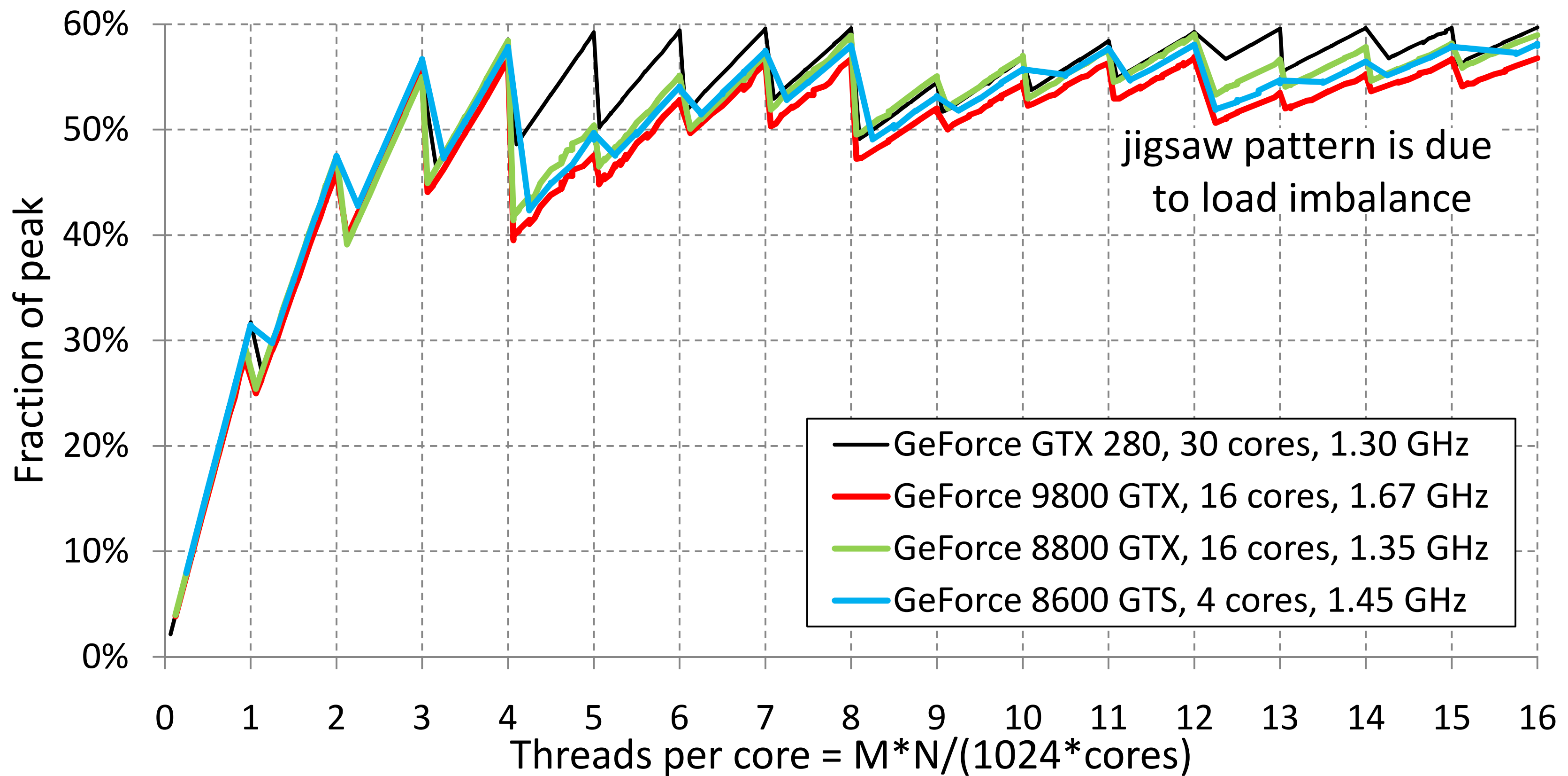
- 1 thread block per multiprocessor
- Many local barriers inside
- Memory latency doesn't overlap with computation

More thread blocks is good

- 2-4 often enough

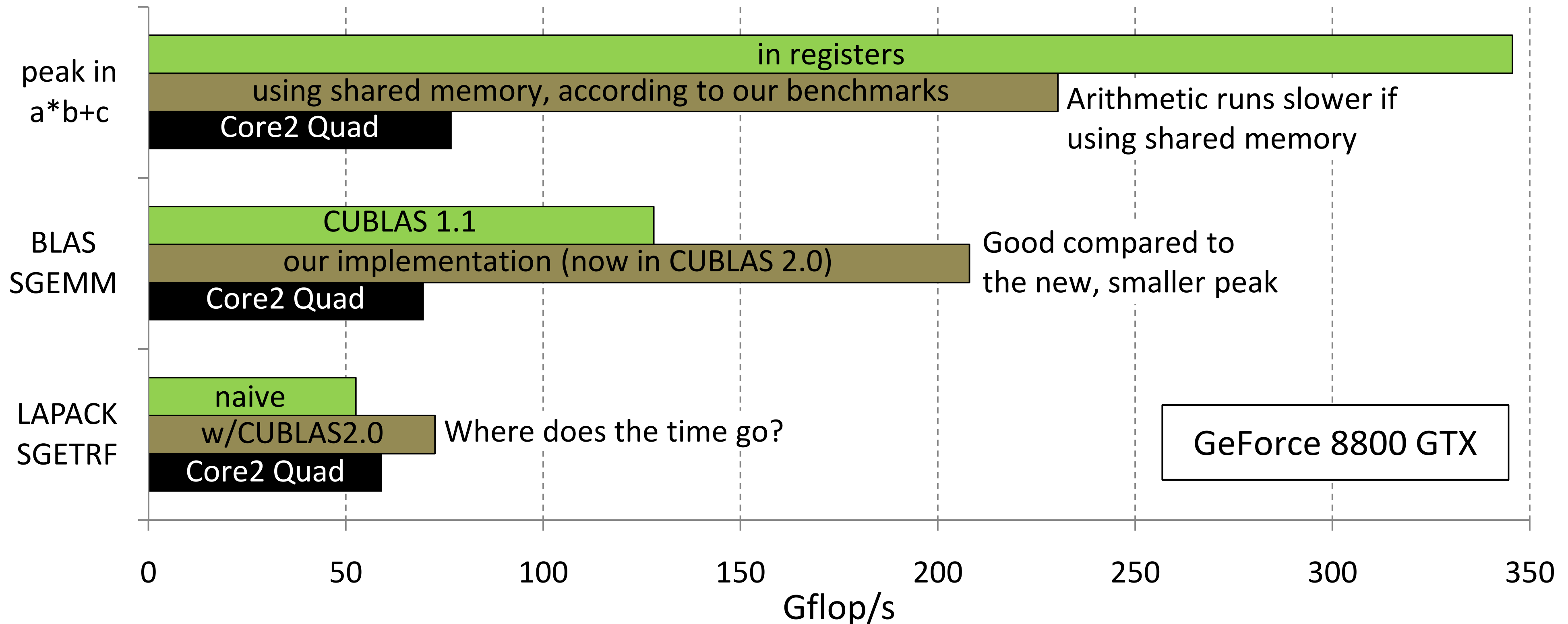
Scaled Performance of SGEMM

Performance plot for the cases when *A* is *M*x2048, *B* is 2048x*N*
(This is a sufficiently large workload with constant arithmetic intensity)



- Nearly same scaled performance across three generations of GPUs
- More threads improves load balance

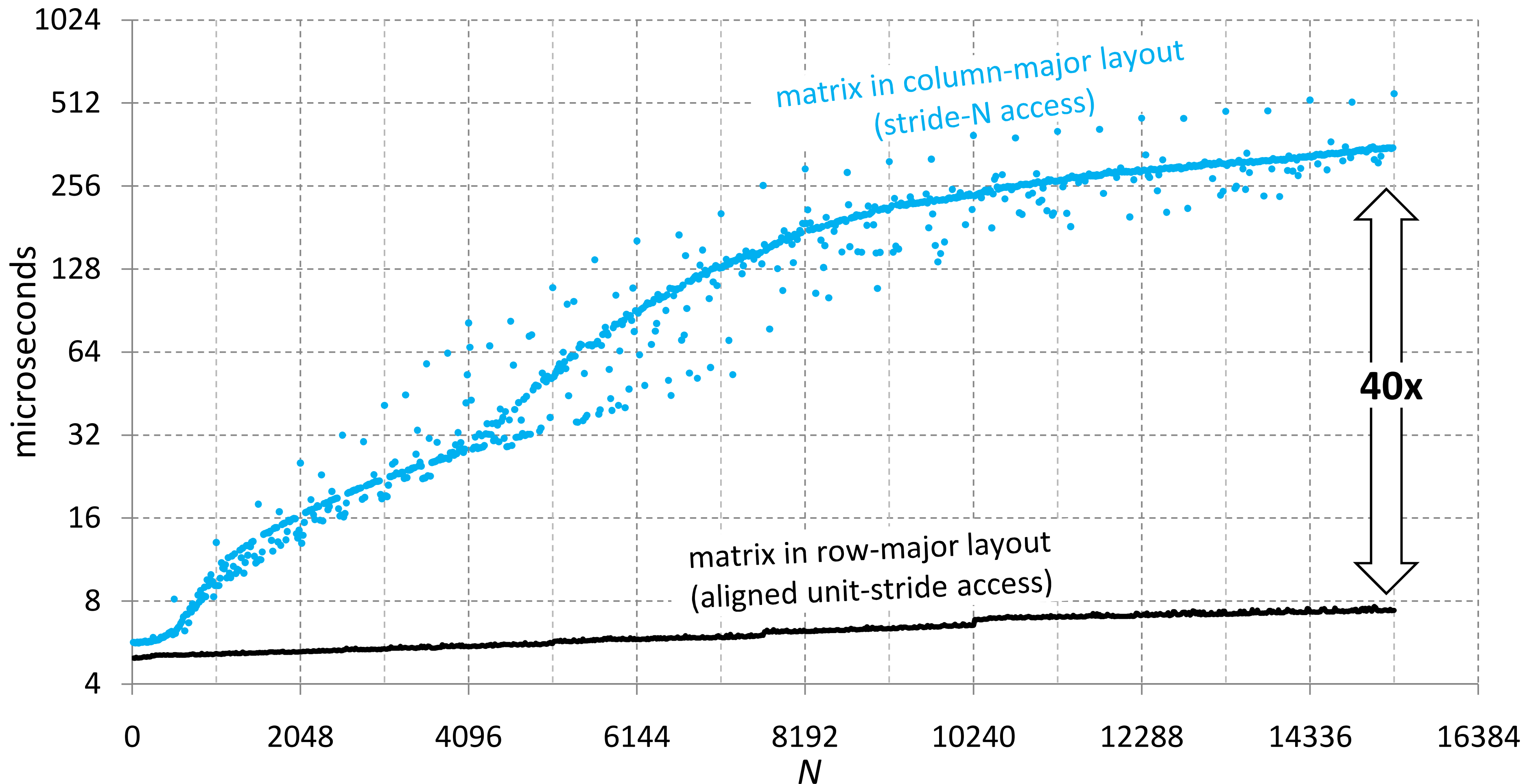
The Progress So Far



- We achieved predictable performance in SGEMM
  - Which does $O(N^3)$ work in LU factorization
- But LU factorization (naïve SGETRF) still underperforms
  - Must be due to the remaining $O(N^2)$ work done in BLAS1 and BLAS2
  - Why does $O(N^2)$ work take so much time?

Row-Pivoting in LU Factorization

Exchange two rows of an *N*x*N* matrix (SSWAP in CUBLAS 2.0):

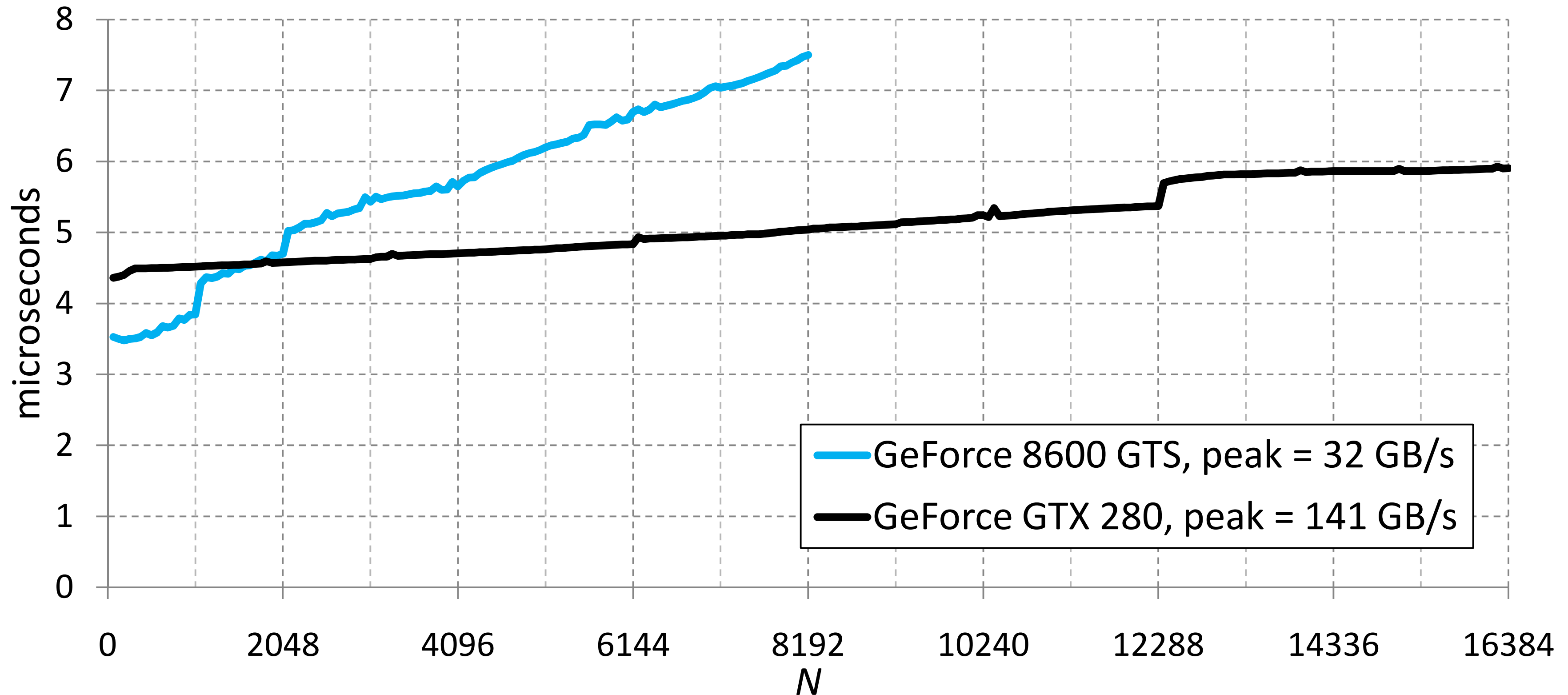


Row pivoting in column-major layout on GPU is very slow

This alone consumes half of the runtime in naïve SGETRF

BLAS1 Performance

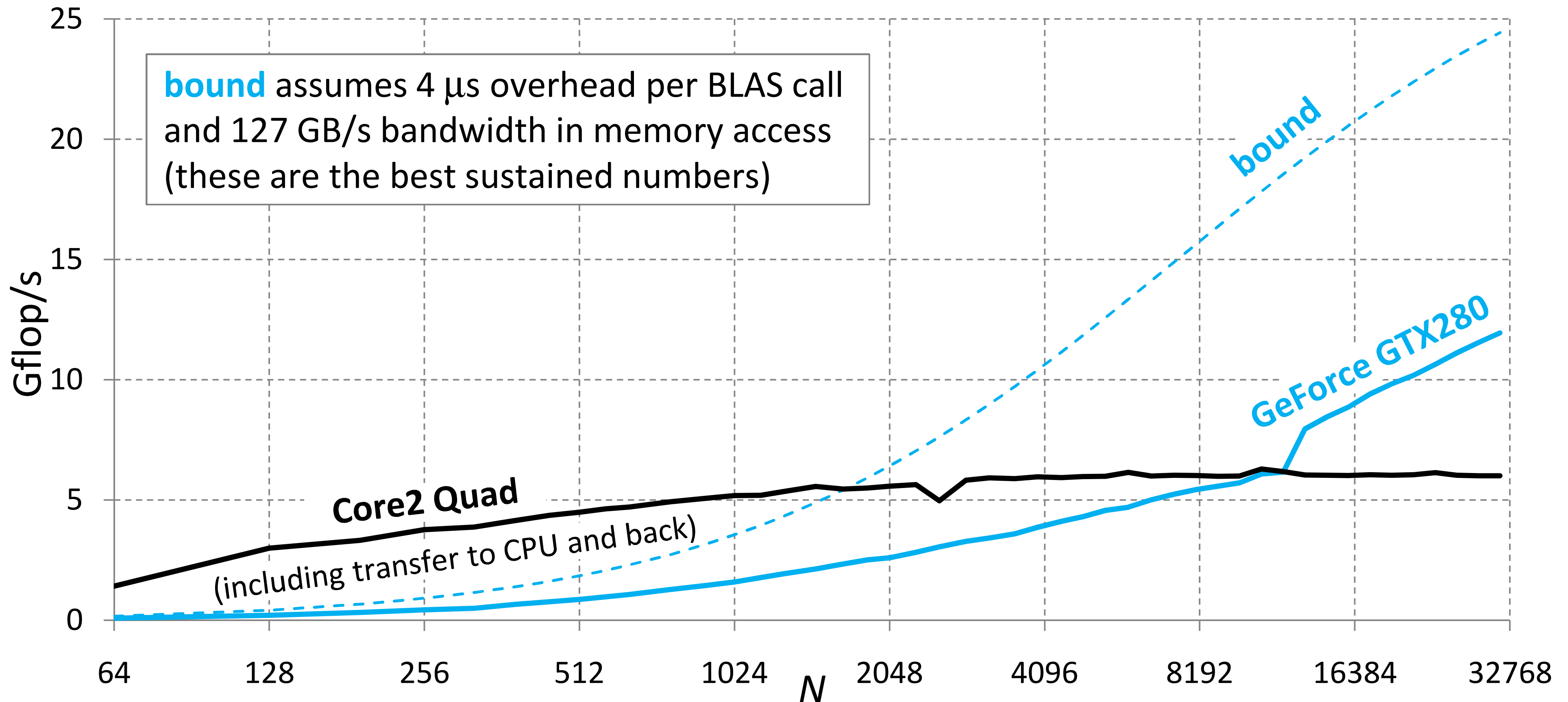
Scale a column of an *NxN* matrix that fits in the GPU memory (assumes aligned, unit-stride access)



- Peak bandwidth of these GPUs differs by a factor of 4.4
- But runtimes are similar
- **Small tasks on GPU are overhead bound**

Panel Factorization

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

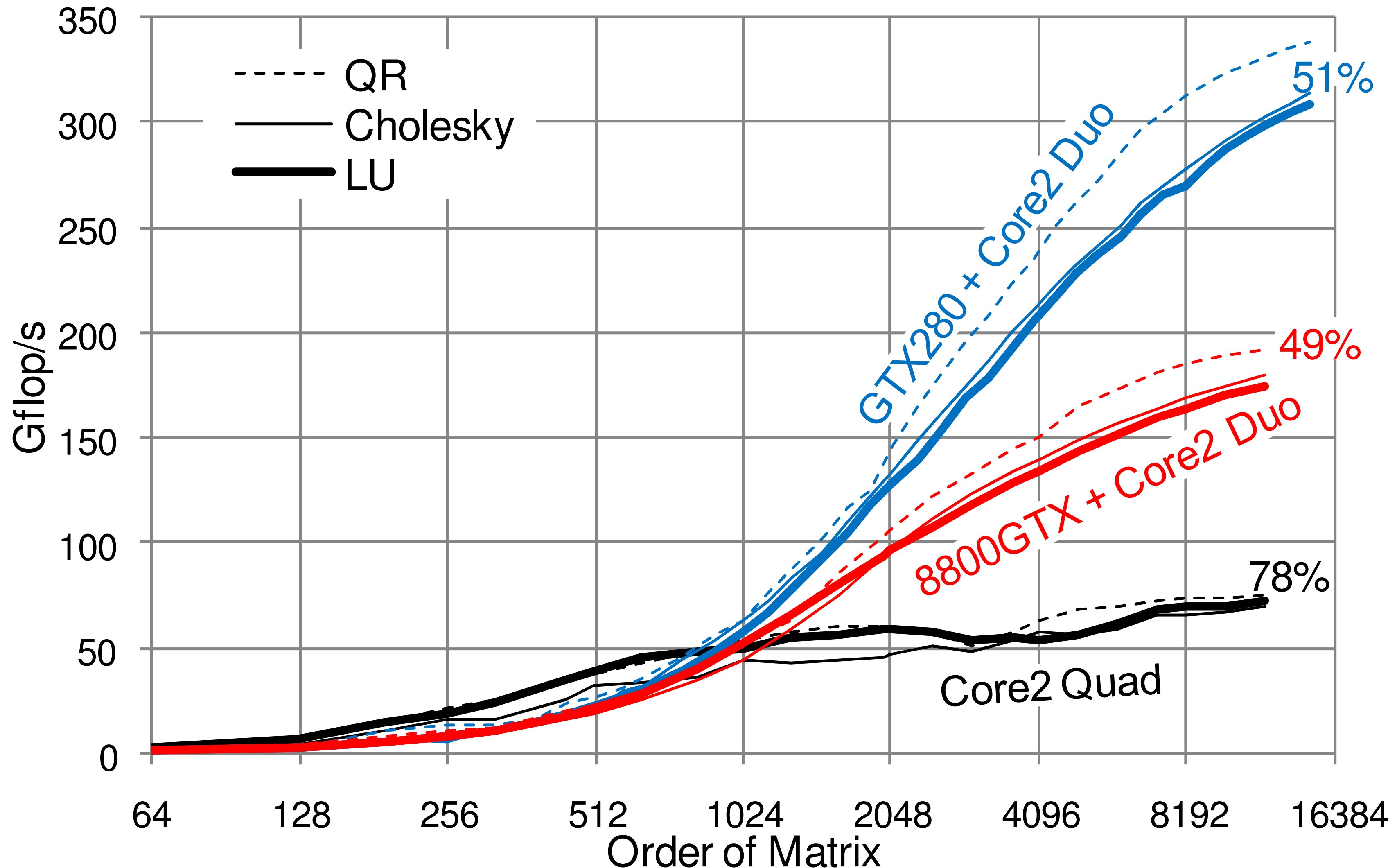


- Invoking small BLAS operations on GPU from CPU is slow
- Can we call a sequence of BLAS operations from GPU?
  - Requires barrier synchronization after each parallel BLAS operation
  - Barrier is possible but requires sequential consistency for correctness

Fast Matrix Factorizations using GPUs

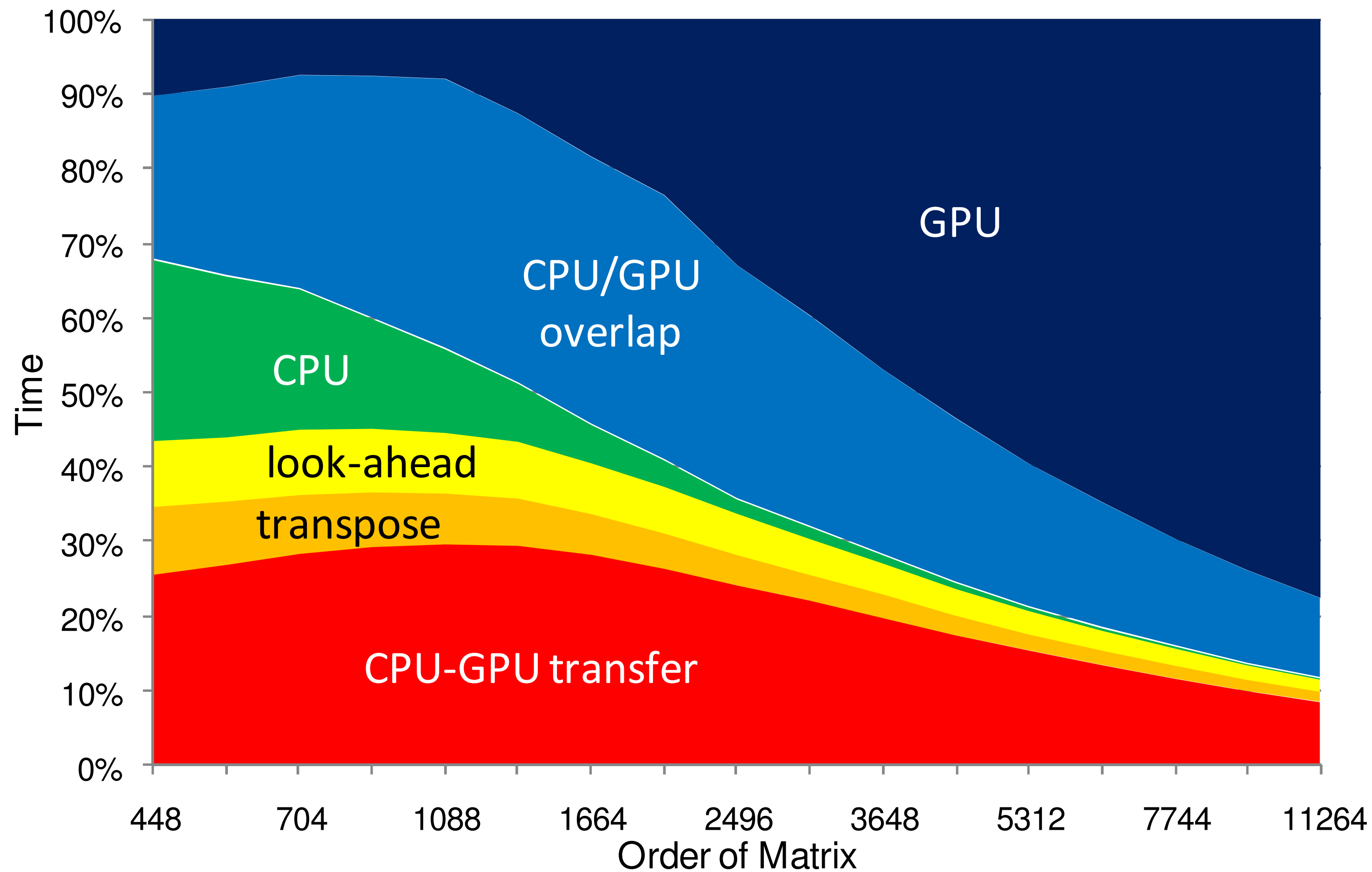
- Use GPU to compute matrix-matrix multiplies only
- Factorize panels on the CPU
- Use look-ahead to overlap computations on CPU and GPU
- Use right-looking algorithms to have more threads in SGEMM
  - Better load balance in the GPU workload, better latency hiding
- Use row-major layout on GPU in LU factorization
  - Requires extra (but fast) matrix transpose for each CPU-GPU transfer
- Substitute triangular solves $LX=B$ with multiply by L^{-1}
  - Provably stable if we do this only when $||L^{-1}|| < fixed_threshold$
  - Small pivot growth nearly always assumes small $||L^{-1}||$
  - Accuracy of LU assumes small pivot growth anyway
- Use two-level and variable size blocking as finer tuning
  - Thicker blocks impose lower bandwidth requirements in SGEMM
  - Variable size blocking improves CPU/GPU load balance
- Use column-cyclic layout when computing using two GPUs
  - Requires no data exchange between GPUs in pivoting
  - Cyclic layout is used on GPUs only so does not affect panel factorizations

Performance Results



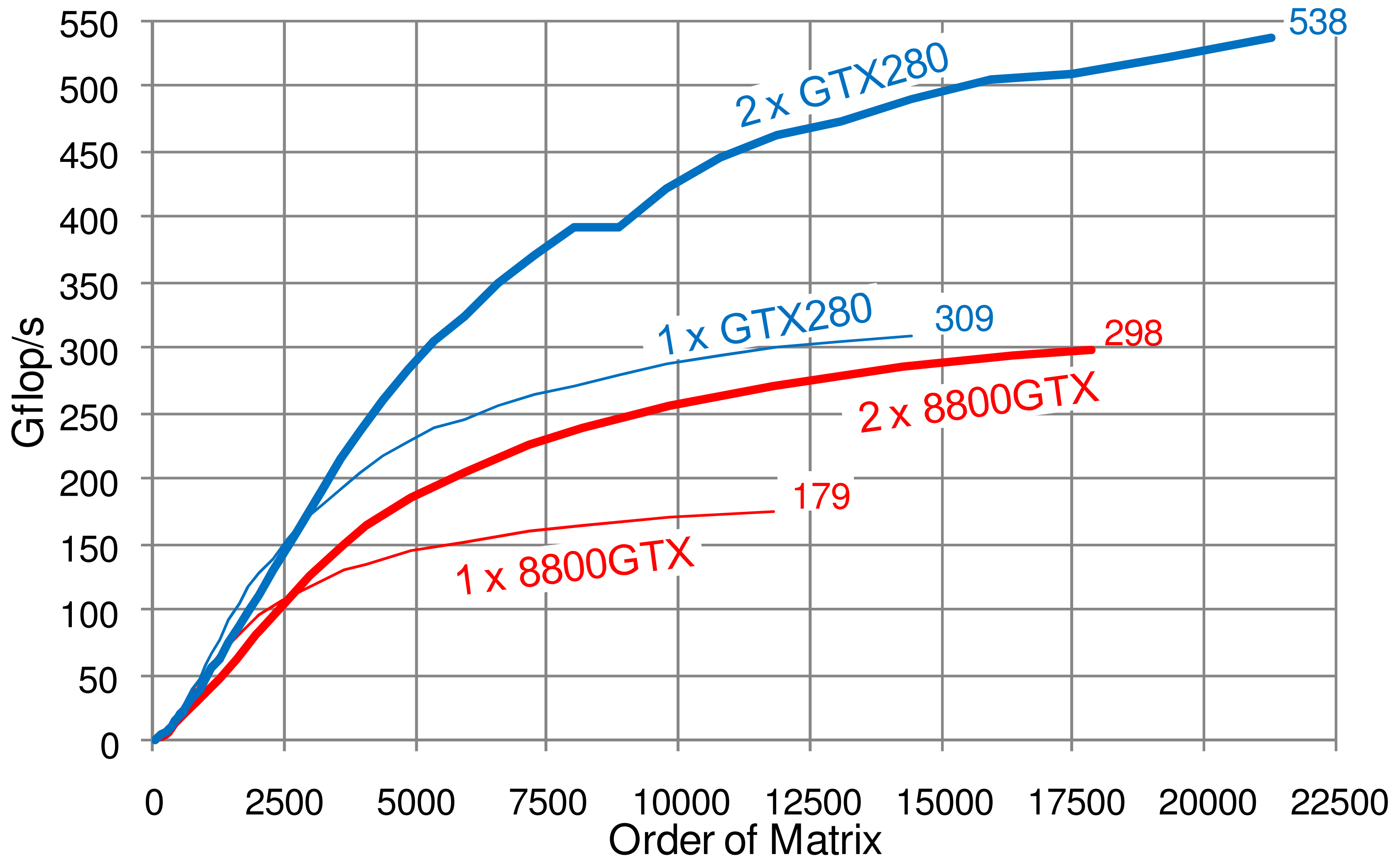
Our solution runs at ~50% of the system's peak (shown on the right)
It is bound by SGEMM that runs at 60% of the GPU-only peak

Time Breakdown for LU on GeForce 8800 GTX



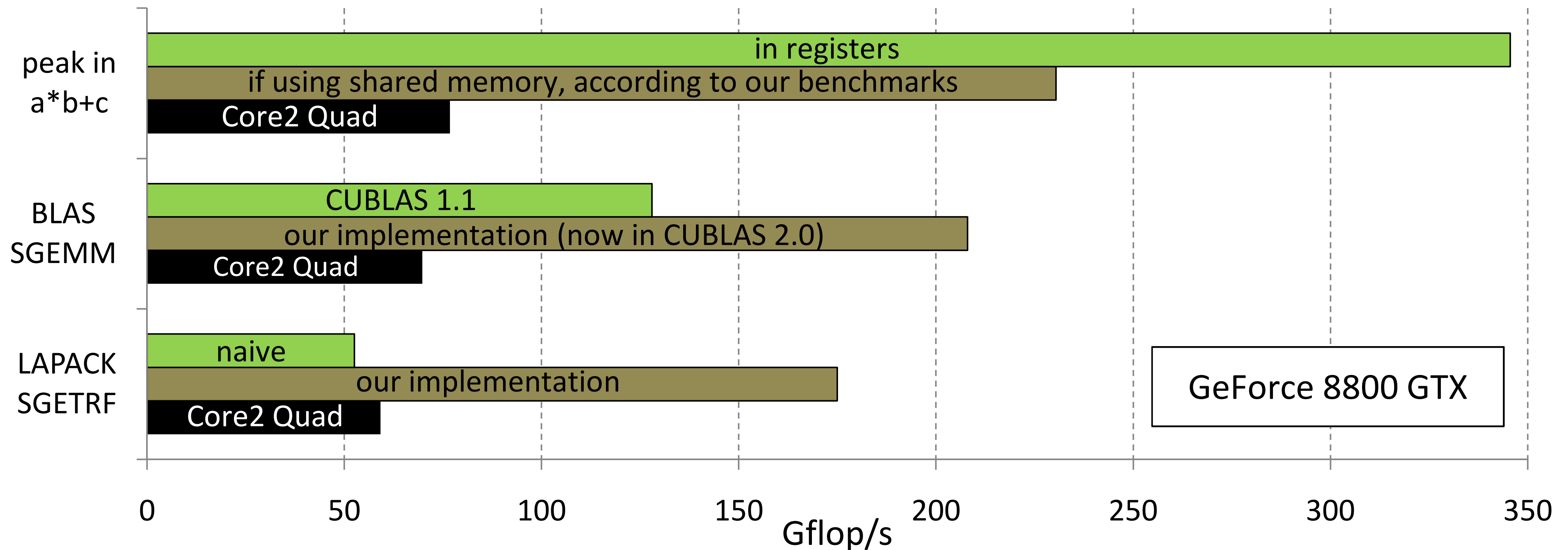
- If we compute on CPU anyway, do that in parallel with computing on GPU

LU Factorization using Two GPUs



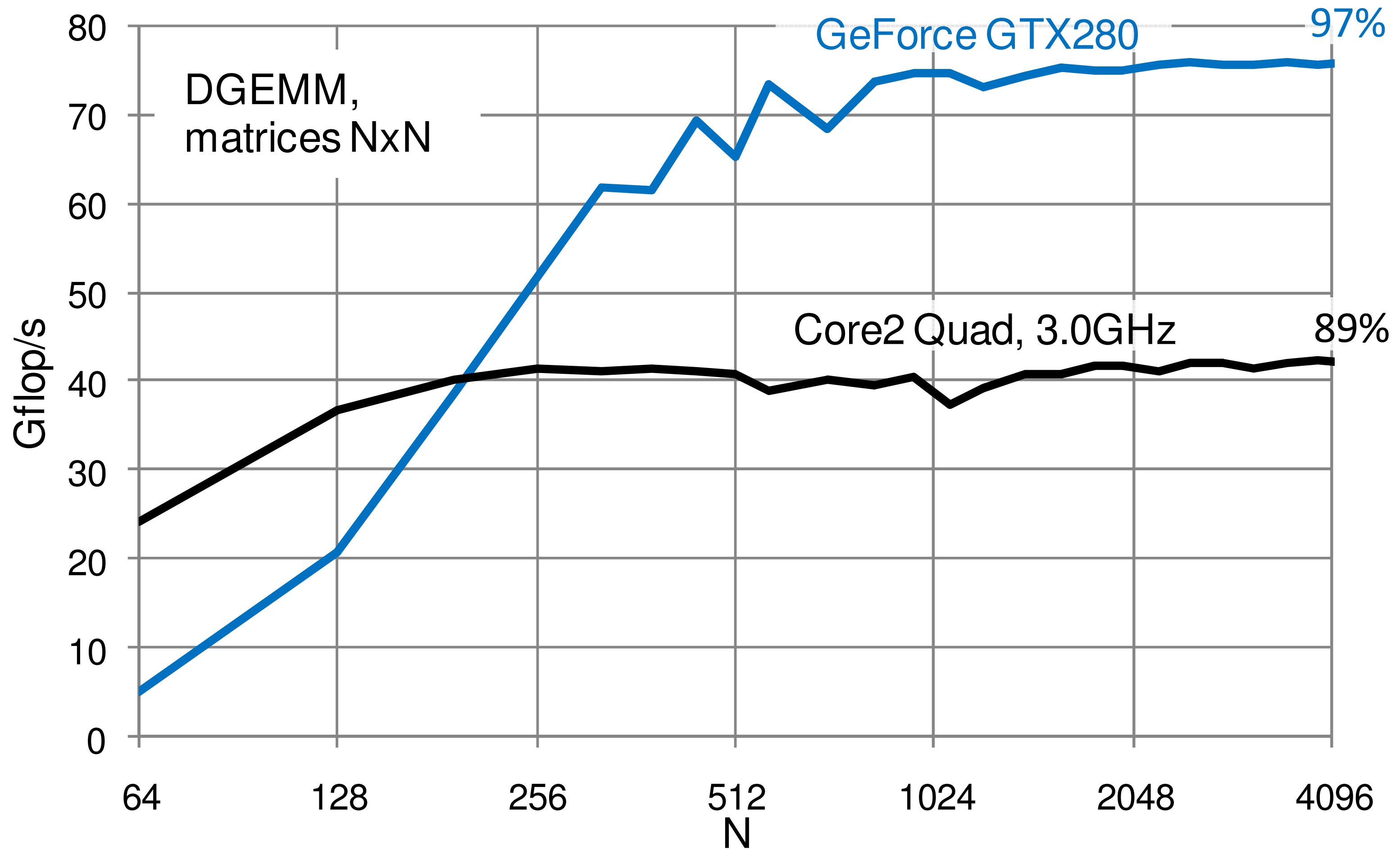
- Second GPU allows 1.7x higher rates
- More than half-teraflop using two GPUs

Conclusion

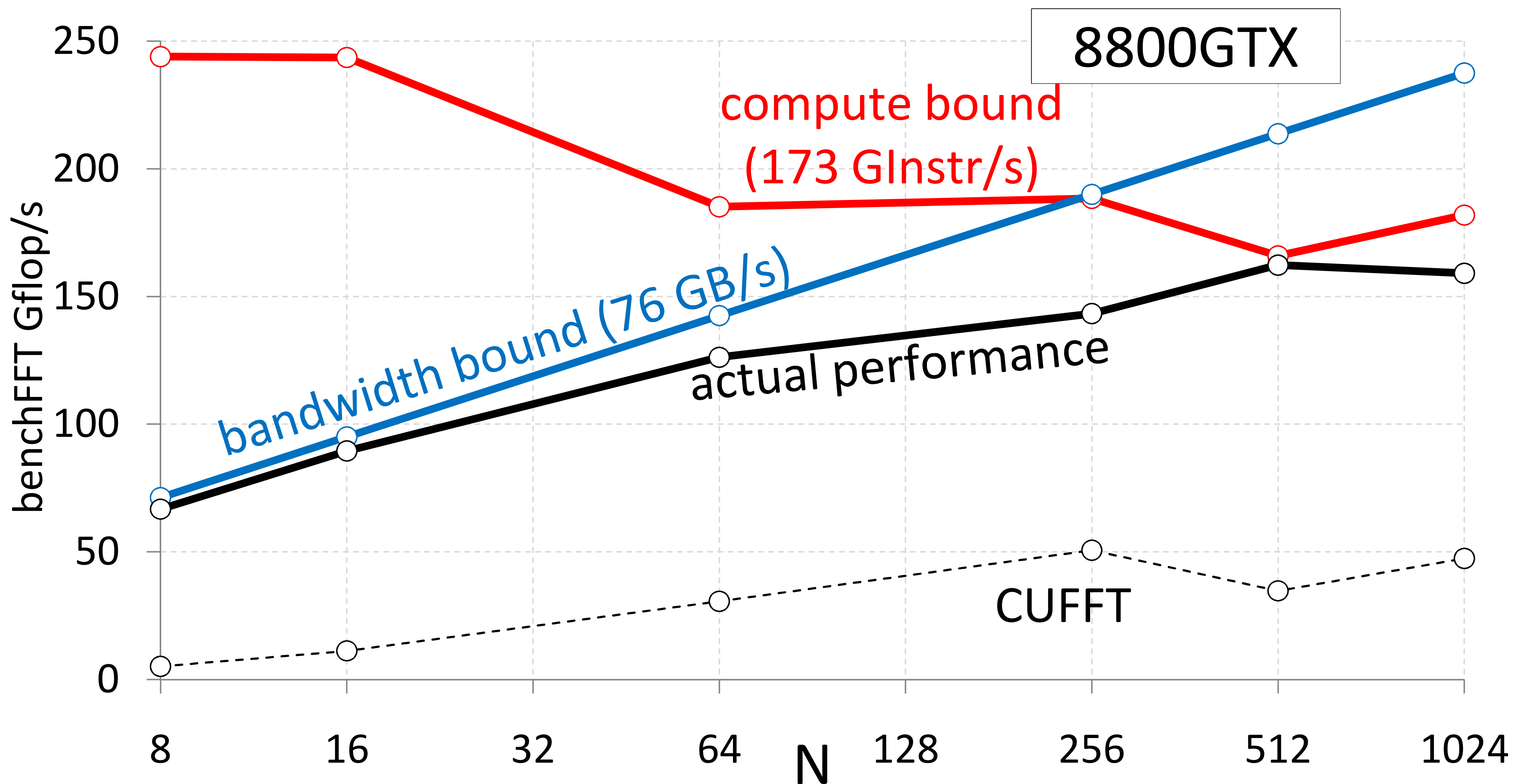


- What we've achieved:
  - Identified new, lower compute peak when using shared memory
  - Achieved a large fraction of this peak in matrix multiply
  - Achieved a large fraction of the matrix multiply rate in dense factorizations
- Our improved performance guidelines resulted in other fast kernels:
  - One of the currently fastest 1D FFTs on GPU for some $N \leq 1024$ (poster at SC08)
  - Close to optimal performance in stencil computations on GPU [Datta et al., SC08]
- Future work: extend results to other important kernels

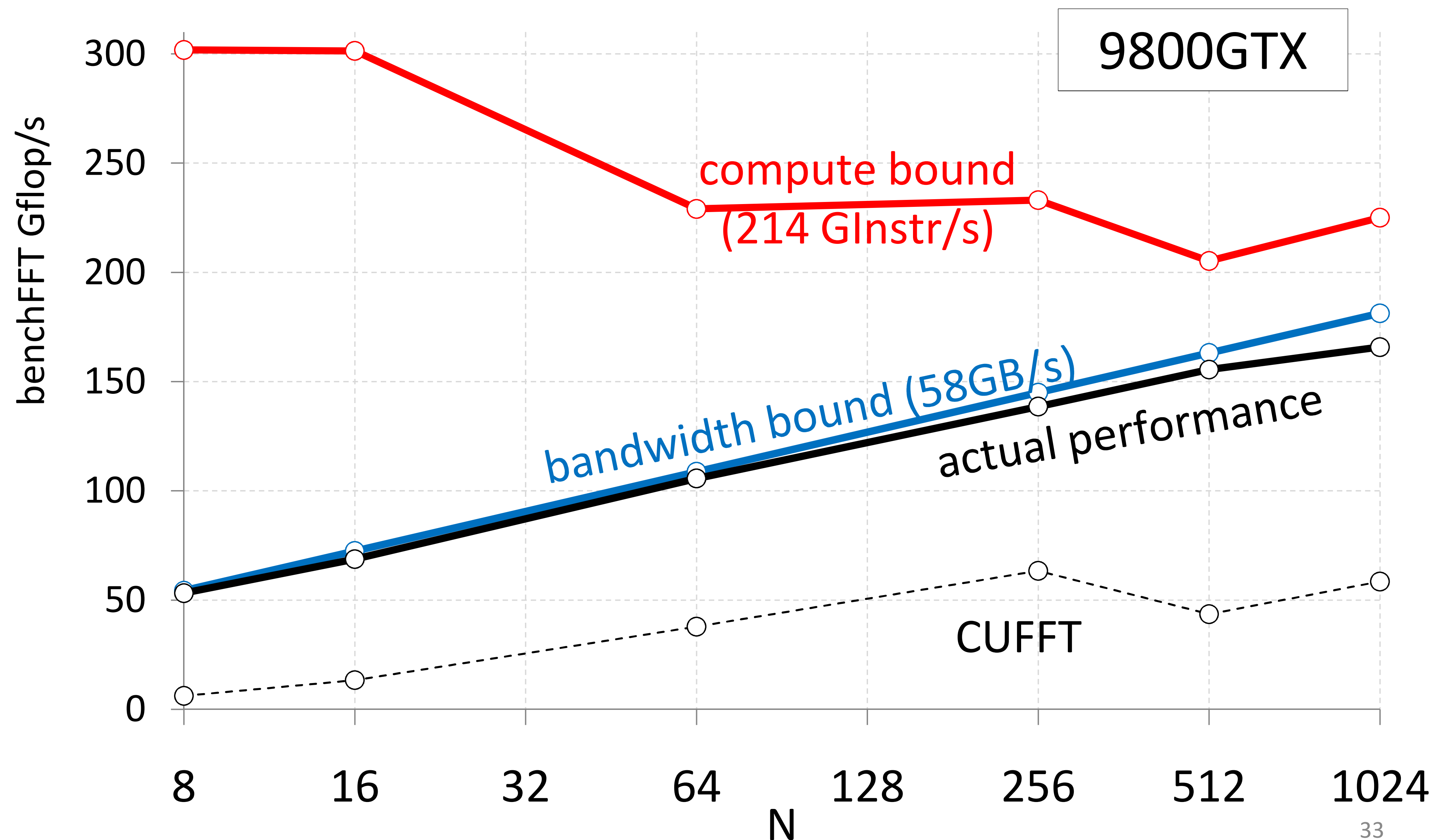
Matrix multiply in double precision

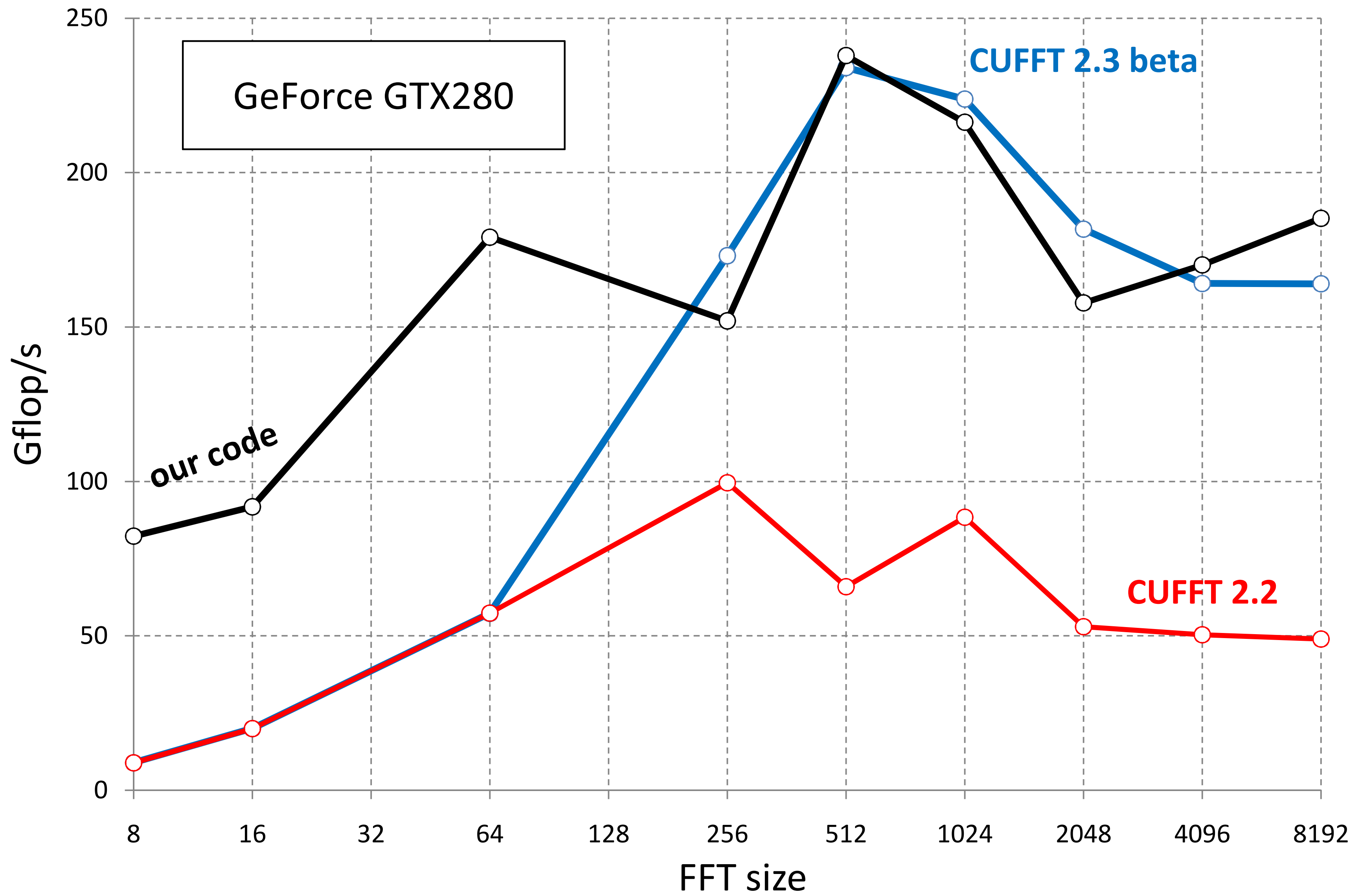


FFT Performance on NVIDIA 8800GTX



FFT Performance on NVIDIA 9800GTX





Also used in OpenCL FFT

- Eigensolvers

Introduction

- Flops are getting cheaper, e.g. NVIDIA GTX280:

Peak arithmetic throughput	624 Gflop/s	(single precision a*b+c)
Peak memory throughput	141 GB/s	18 flops per (32-bit) word
Kernel launch overhead	~5μs	~3,000,000 flops

- If bound by overhead, doing extra work may be free
  - If doing 1 flop is 5μs, doing 300,000 flops is 5.1μs
- Reduce memory traffic at the cost of extra flops
  - Recompute matrix factors instead of retrieving them
- Try fast algorithms that might fail (but rarely)
  - Check for failure, recompute if needed

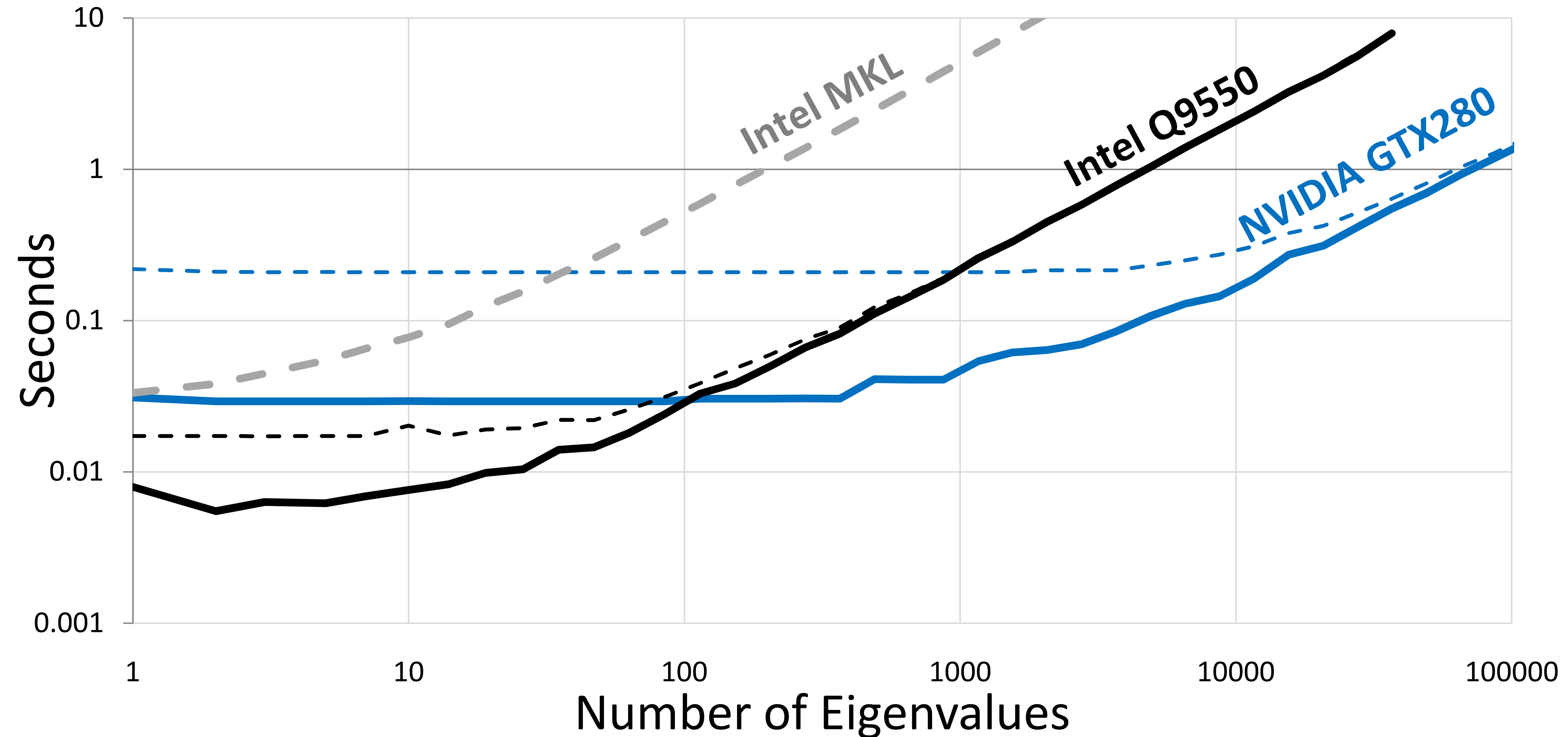
Bisection Eigensolver (LAPACK's SSTEBZ)

- Finding eigenvalues of symmetric tridiagonal T in [lb,ub]
  - Let a(1...N) hold diagonal of T, b(0…N-1) offdiagonal (b(0)=0)

```
Function Count(x,T) … # eigenvalues of T < x
      Count = 0
       d = 1
       for j = 1 to N
           d =a(j) – x – b(j-1)^2/d
(*)        if (d<0) then Count = Count + 1
```

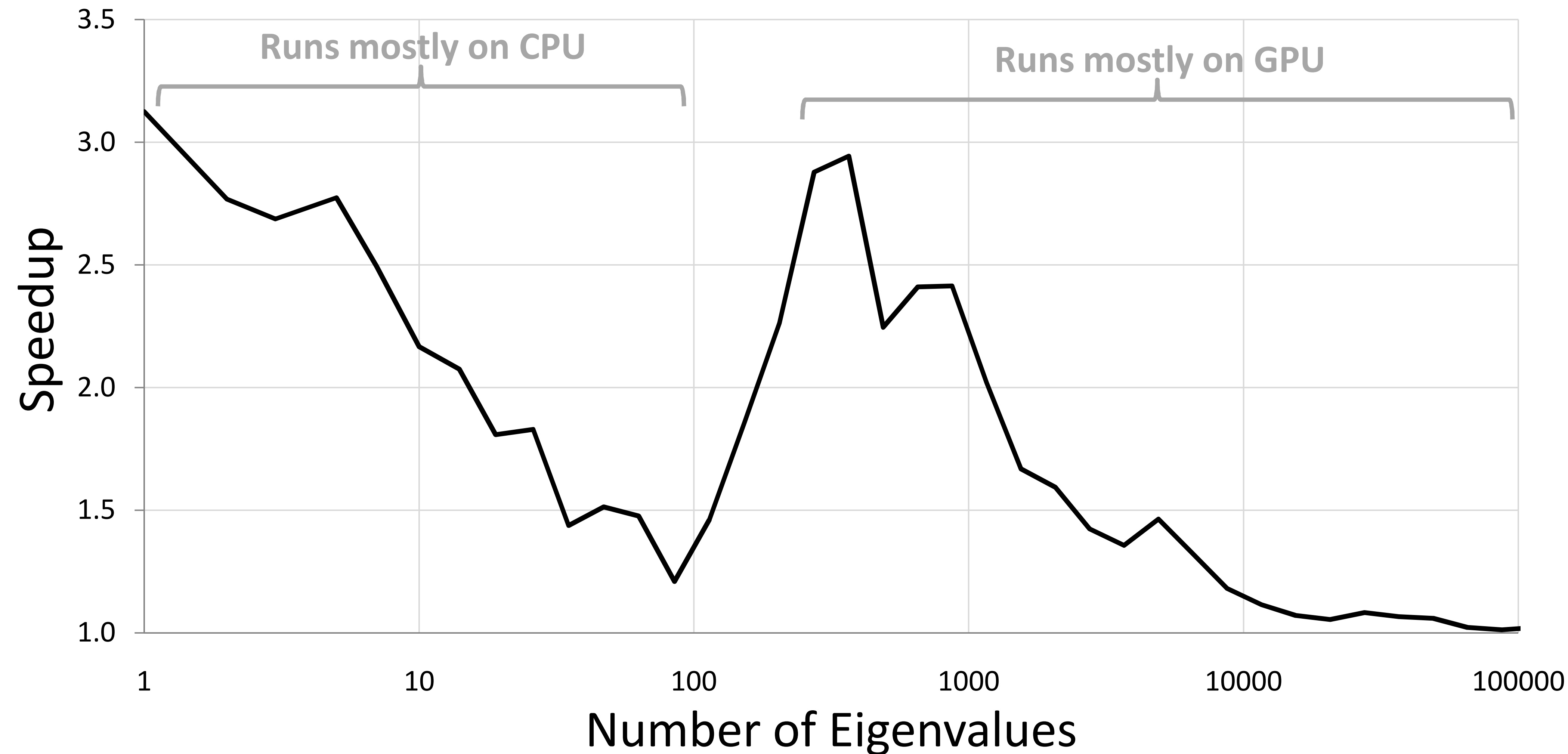
- If many values of x available, parallelize over them
  - Usual approach: repeated bisection of [lb,ub]
    - Minimizes flops needed to find narrow intervals containing each eigenvalue
    - Not much parallelism at start (when there are only 1,2,4,… intervals)
  - Faster approach: Multisection of intervals into $M \geq 2$ subintervals
    - Choose M on-the-fly by maximizing Efficiency = log(M)/time(M)
    - Use formula for time(M) using off-line benchmarking and data fitting

Find a Subset of Eigenvalues, N≈100,000



- (dashed line – bisection, solid line – multisection)
- CPU is still faster at small problems!
- Crossover point depends on eigenvalue distribution

Speedup for Heterogeneous Algorithm



- Runs on CPU or GPU depending on N and current #intervals
  - Decide between CPU and GPU every iteration of multisection

Inverse Iterations (LAPACK's SSTEIN)

- Inverse iteration for eigenvectors
  - Solve $(T - \lambda I) z = x(i)$, $x(i+1) = z / ||z||$
- LAPACK's implementation: BLAS1 parallelism only
  - Computes one eigenvector at a time
  - Not vectorized, doesn't scale with #cores
- We compute all eigenvectors in parallel
  - Challenge: how to keep them orthogonal?
    - This is future work. (Post-process with QR?)

Inverse iteration compute bound on GPU?

- Estimate required parallelism using Little’s law
  - ALUs: 24 cycle latency, 240 ops/cycle throughput
  - 5760 independent operations must be in the flight
    - Otherwise – poor utilization of hardware
- Little parallelism in finding one eigenvector
  - Must solve for thousands eigenvectors in parallel
- How much space we have per eigenvector solve?
  - ~3MB on GPU in total
    - Registers, shared memory, texture caches
  - 3MB/5760 $\approx$ 140 single precision numbers
    - If N > 140, data doesn’t fit on chip => **bandwidth bound**

Reducing Memory Traffic in SSTEIN

- Memory traffic in LAPACK's implementation:
  - Factorizing $T - \lambda I$ produces 4N words
  - Factorization must be read once per iteration (+4N)
  - Each iteration updates right hand side twice (+4N)
  - Total cost is $4N+8Nn_{iter}$ per one eigenvector
- First, abandon pivoting
  - I.e. $T - \lambda I = LDL^T$ for diagonal D, unit bidiagonal L
  - New total cost is $2N+6Nn_{iter}$
- Second, store only D^{-1}, recompute L using T
  - T is same for all problems – fits cache
  - New total cost is $N+5Nn_{iter}$, 1.75x speedup for $n_{iter}=3$

Conclusion

- Total memory traffic is 16N per eigenvector
- Transferring result to CPU costs another N
  - 16x smaller, but runs at 20x lower throughput!
    - **Consumes half of the runtime**
    - But may still outperform CPU
- Practice: 120x faster than LAPACK on CPU
  - However, computation failed in 0.06% cases
    - Other time 8x lower accuracy was detected
    - This is the outcome of using no pivoting
    - Future work: recompute using safe algorithm

- Hiding latency
- Global synchronization

Memory latency hiding

Little's law

data in transit [B] = latency [s] * 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	**2.0 B**	**1.4 B**	**1.4 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	**1.5 KB**	**1.1 KB**	**1.4 KB**

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	**1.9 MB**

Many threads require many registers

Up to ≈ **2 MB** registers on die in total

- largest memory on die (4x shared memory)

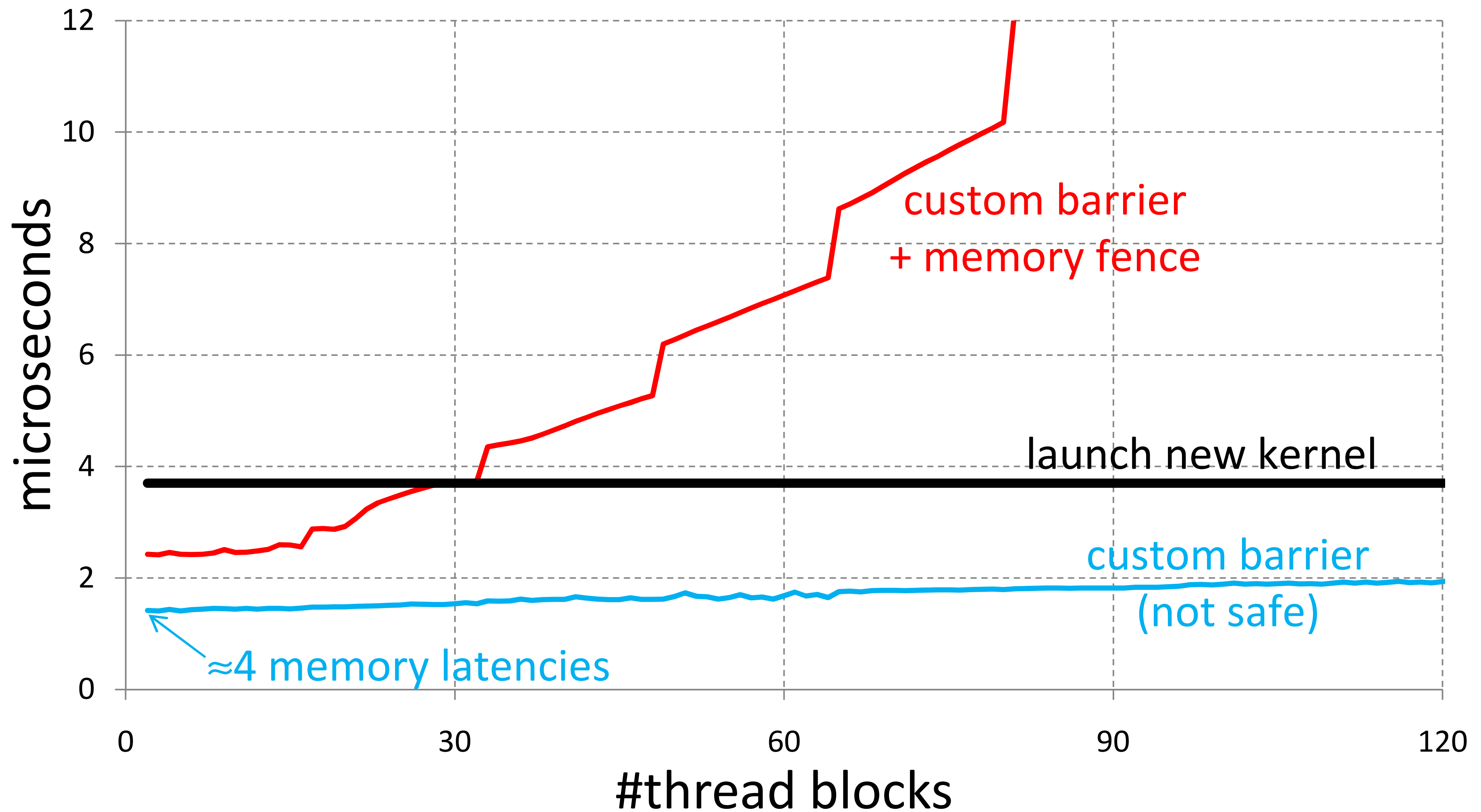
Got many registers – how to use them well?

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

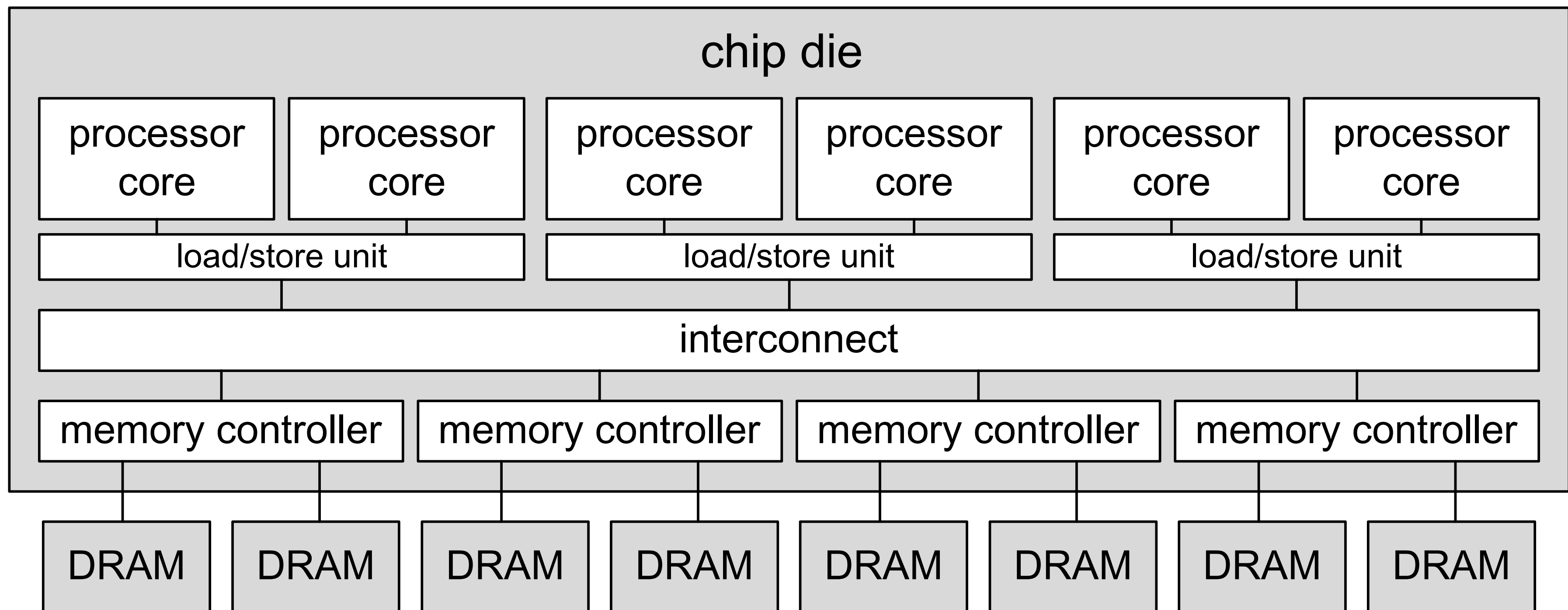
Alternative global synchronization

- 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)



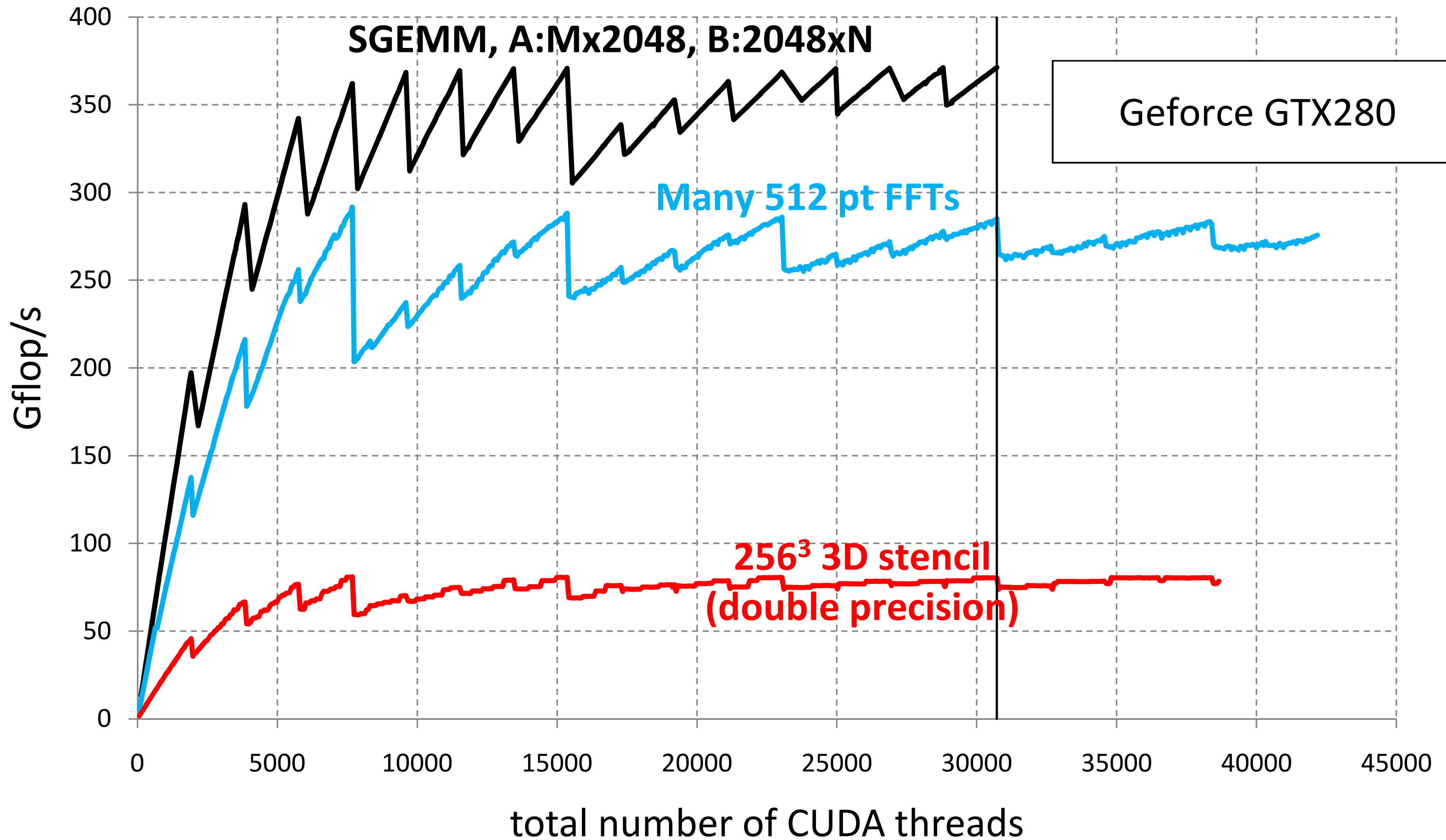
- Trends: fast on-chip global synchronization
  - Coherent caches on Intel Larrabee
  - ATI GPUs have global shared memory (GDS)

Fast on-chip communication?



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

Won't global barrier deadlock on GPU?



It will if you run too many threads, but you don't have to

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