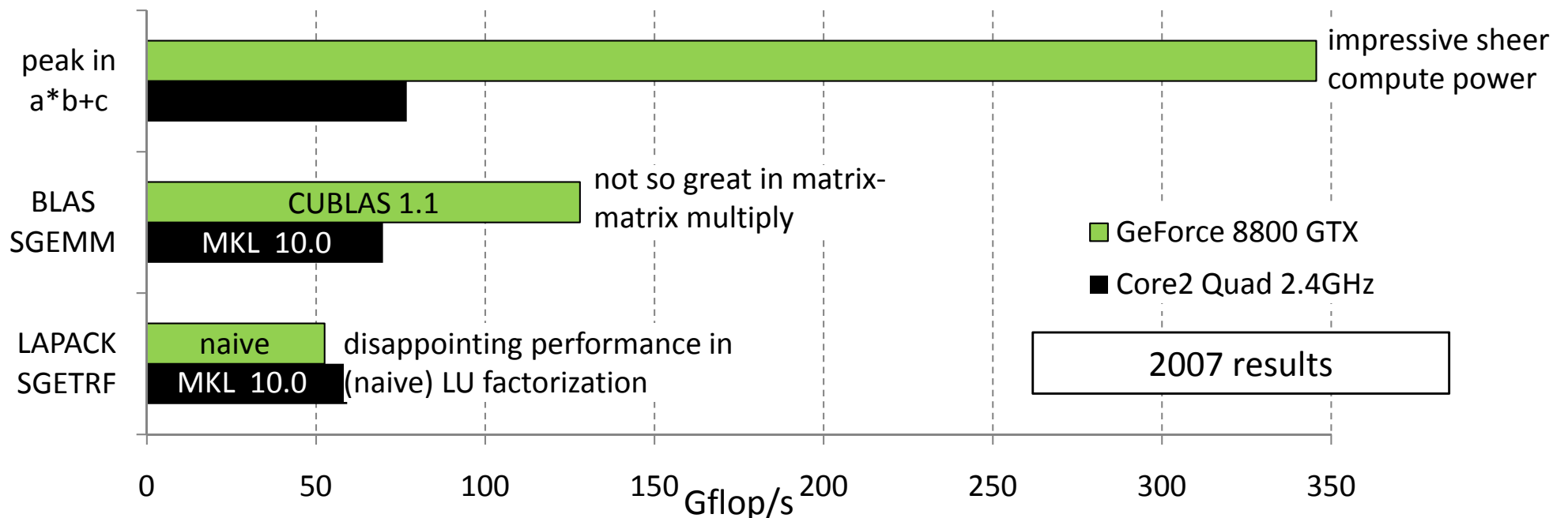


# Benchmarking GPUs to Tune Dense Linear Algebra

Vasily Volkov and James Demmel  
UC Berkeley

# Motivation

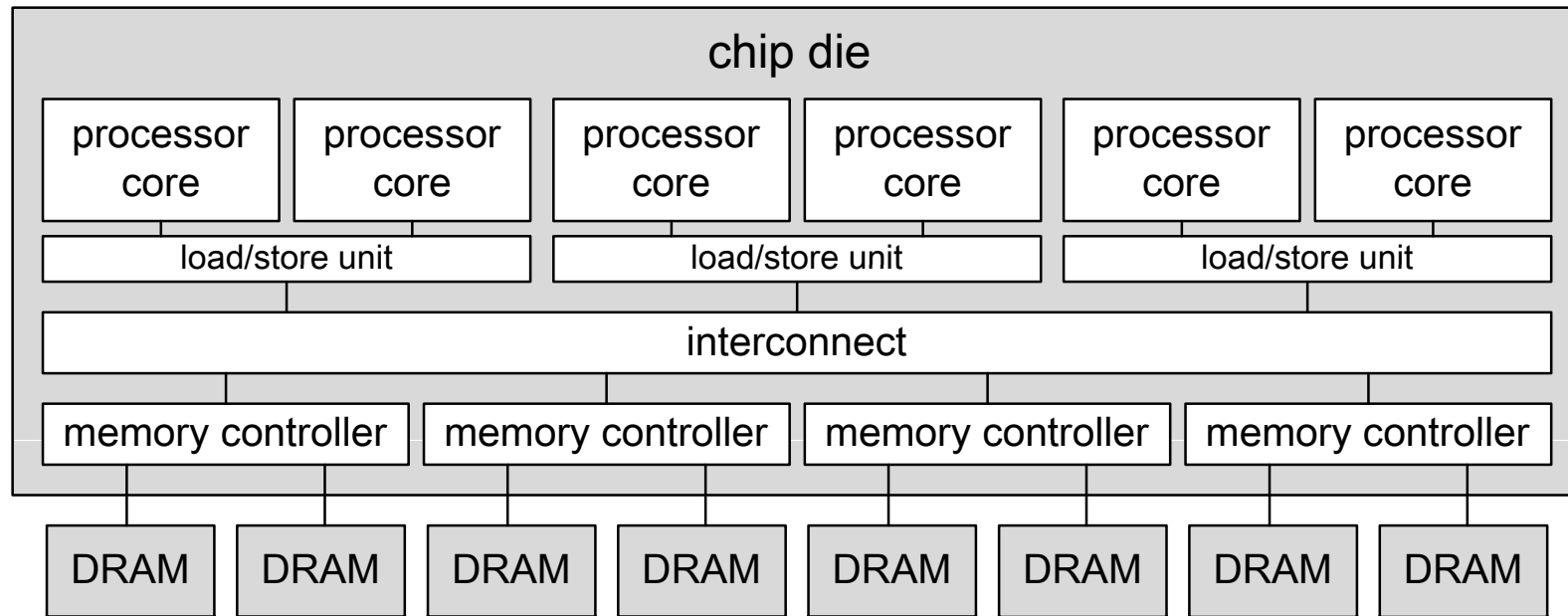
- NVIDIA released CUBLAS 1.0 in 2007, which is BLAS for GPUs
- This enables porting LAPACK to GPU in an easy and obvious way
- (We are concerned with single precision only throughout the talk)



- 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

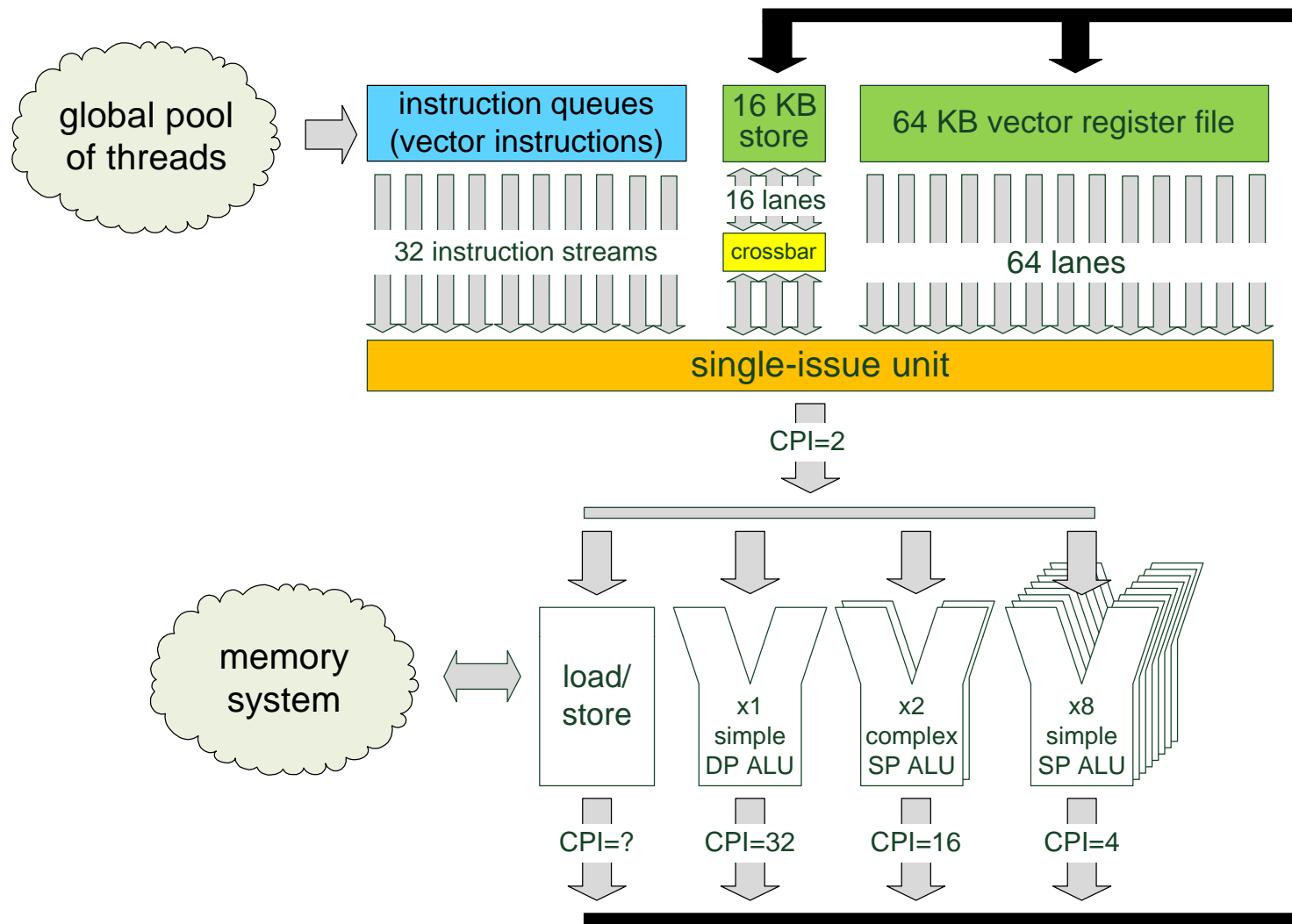
- Multiprocessor on chip = multi-core



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

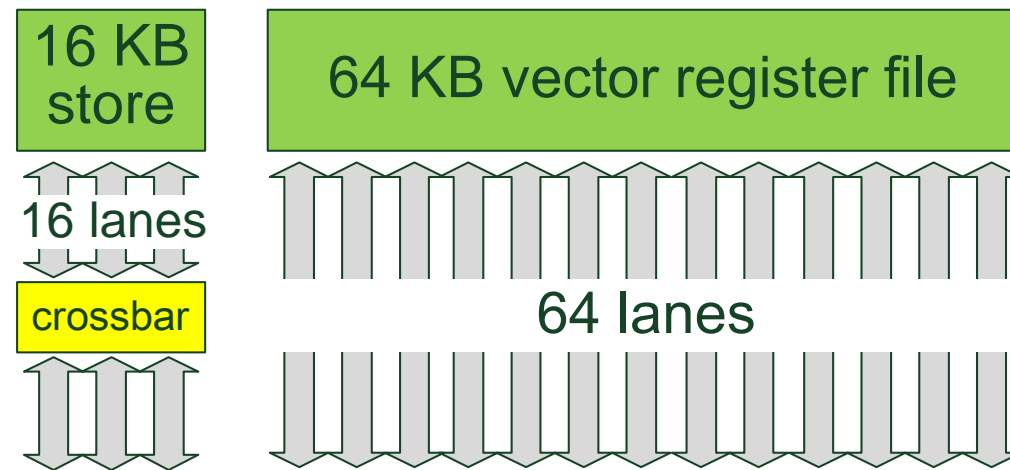
GPU (NVIDIA GeForce)	8600 GTS	9800 GTX	GTX 280
Processor cores	4	16	30
Compute capability (a+b*c)	93 Gflop/s	429 Gflop/s	624 Gflop/s
Memory controllers	2	4	8
Memory bandwidth	32 GB/s	70 GB/s	141 GB/s

# 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 if 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  $M \times N$  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

- 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;
```

} Compute pointers to the data

```
    __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;
```

} Declare the on-chip storage

```
    do  
    {
```

```
    #pragma unroll
```

```
        for( int i = 0; i < 16; i += 4 )  
            bs[threadIdx.x][threadIdx.y+i] = B[i*ldb];  
        B += 16;  
        __syncthreads();
```

} Read next B's block

```
    #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];  
        }
```

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

```
        __syncthreads();
```

```
    } while( B < Blast );
```

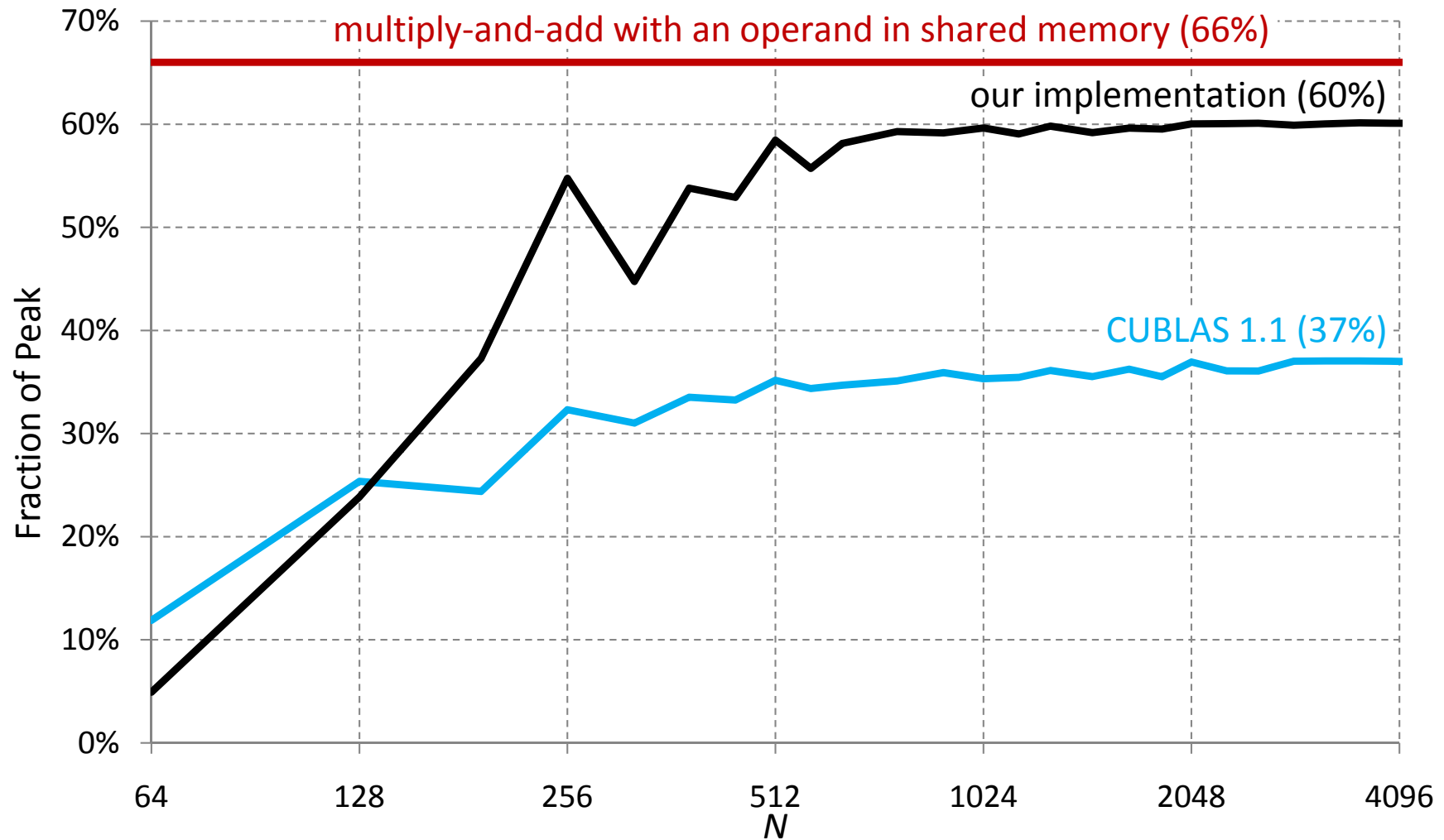
```
    for( int i = 0; i < 16; i++, C += ldc )  
        C[0] = alpha*c[i] + beta*C[0];
```

} Store C's block to memory

```
}
```

# Our code vs. CUBLAS 1.1

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



What causes CUBLAS 1.1 to run slower than our code?

# Our code vs. CUBLAS 1.1

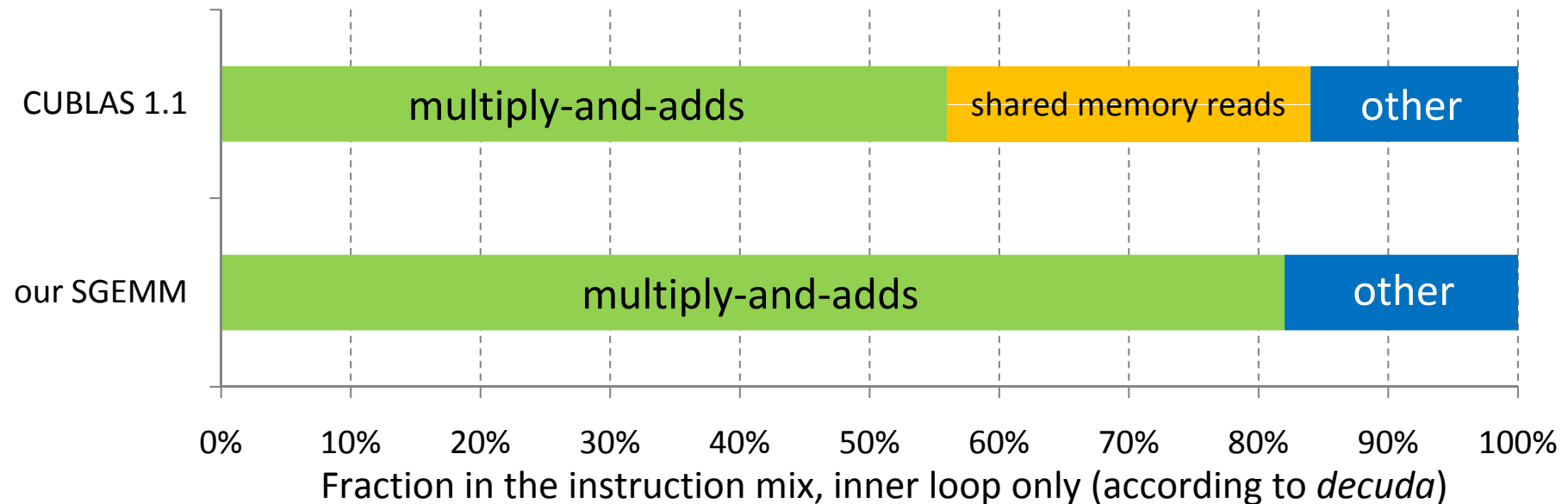
- Popular GPU programming guidelines recommend:
  - Minimize use of registers
  - Maximize use of shared memory
  - Use longer vectors
  - Maximize occupancy (number of concurrent instruction streams)
- CUBLAS 1.1 succeed in following all of them, but loses in performance:

	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  $M \times K$  matrix by  $K \times N$  matrix

# Our code vs. CUBLAS 1.1

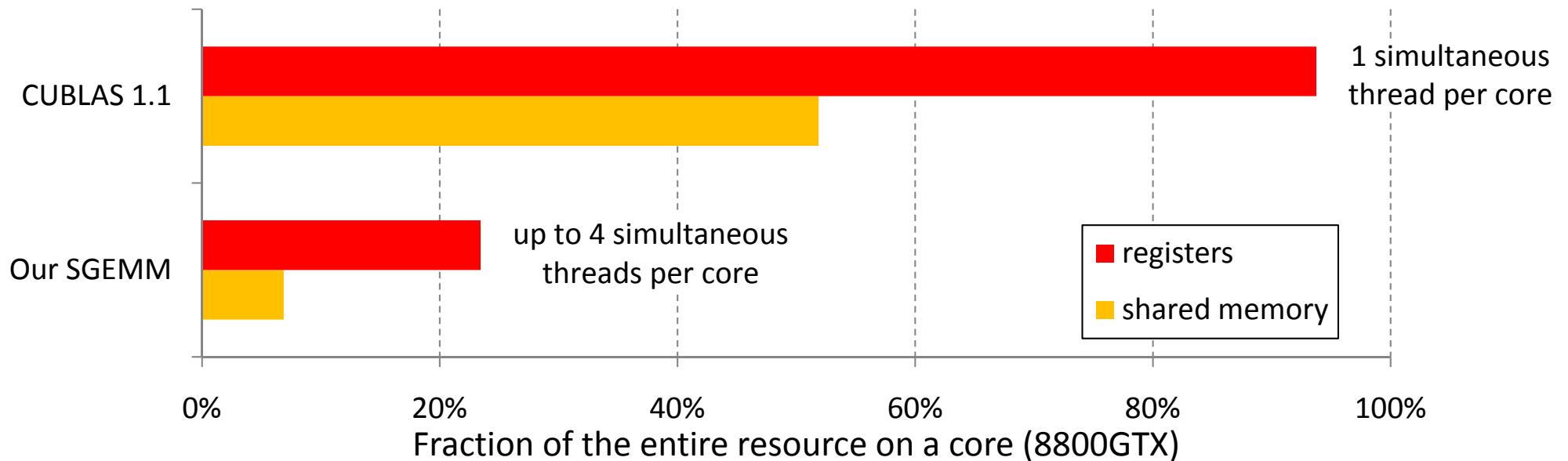
- Ideally, the code is instruction throughput bound
- Then the performance in Gflop/s is bound by the instruction mix



- 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

# Threads vs. Instruction Streams

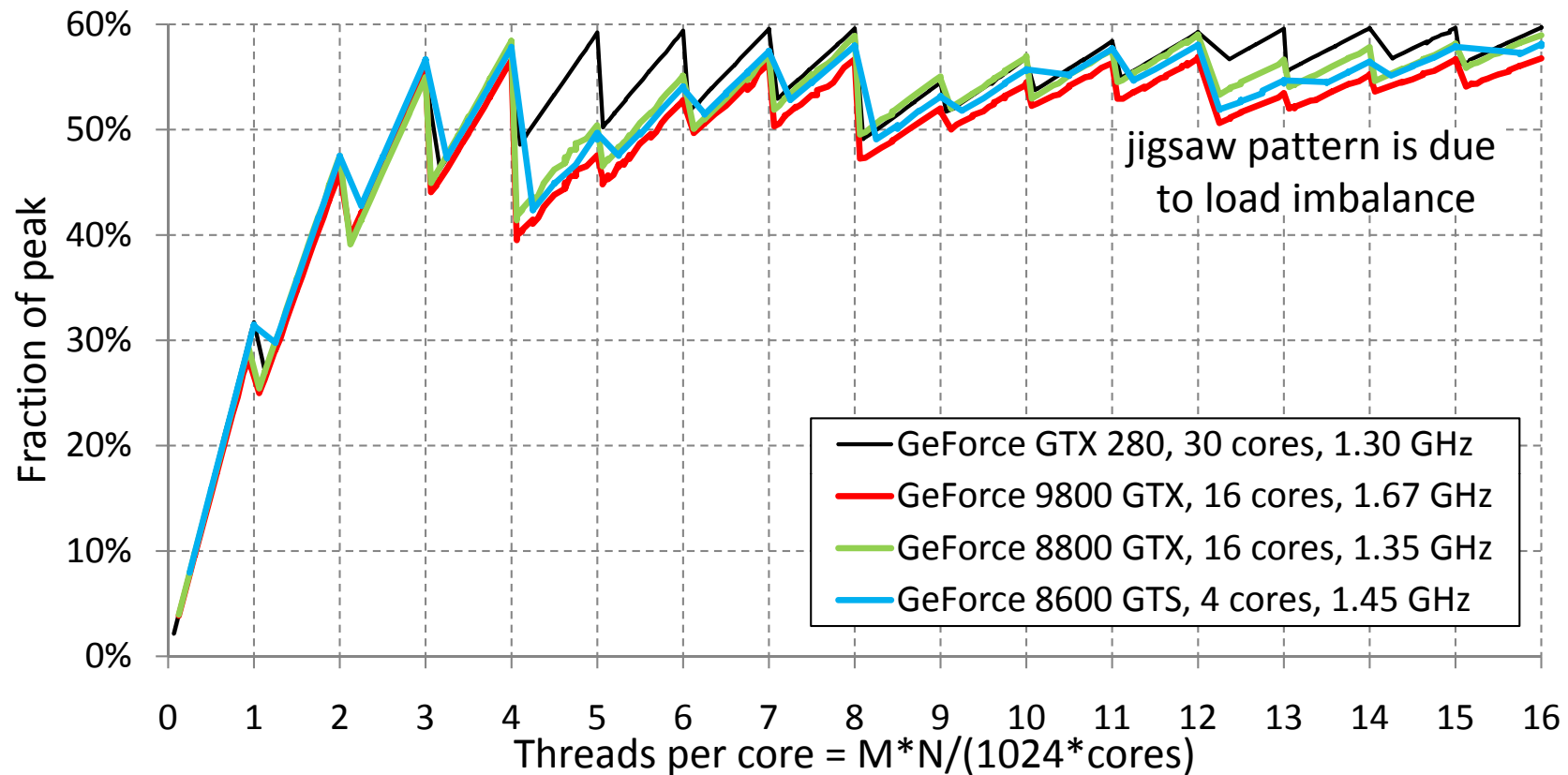
- CUBLAS 1.1 permits more concurrent instruction streams, but fewer concurrent threads:

	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

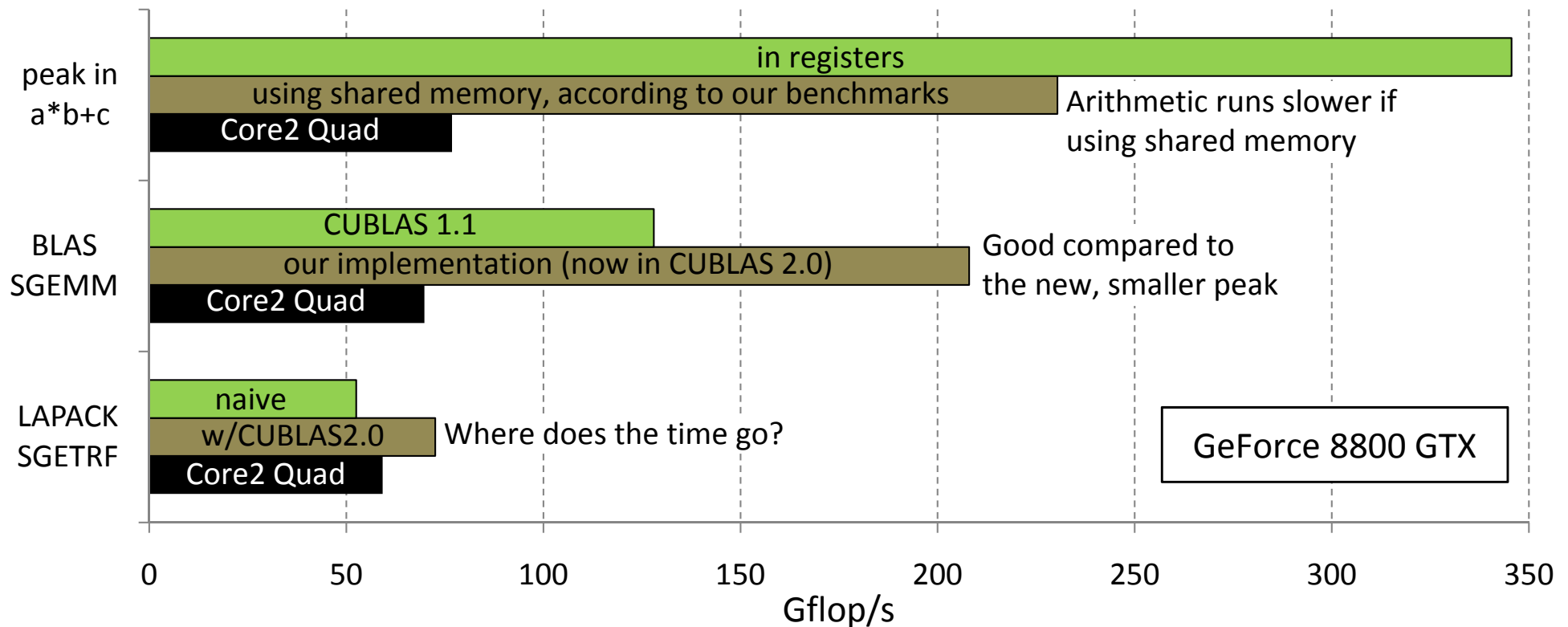
# Scaled Performance of SGEMM

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



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

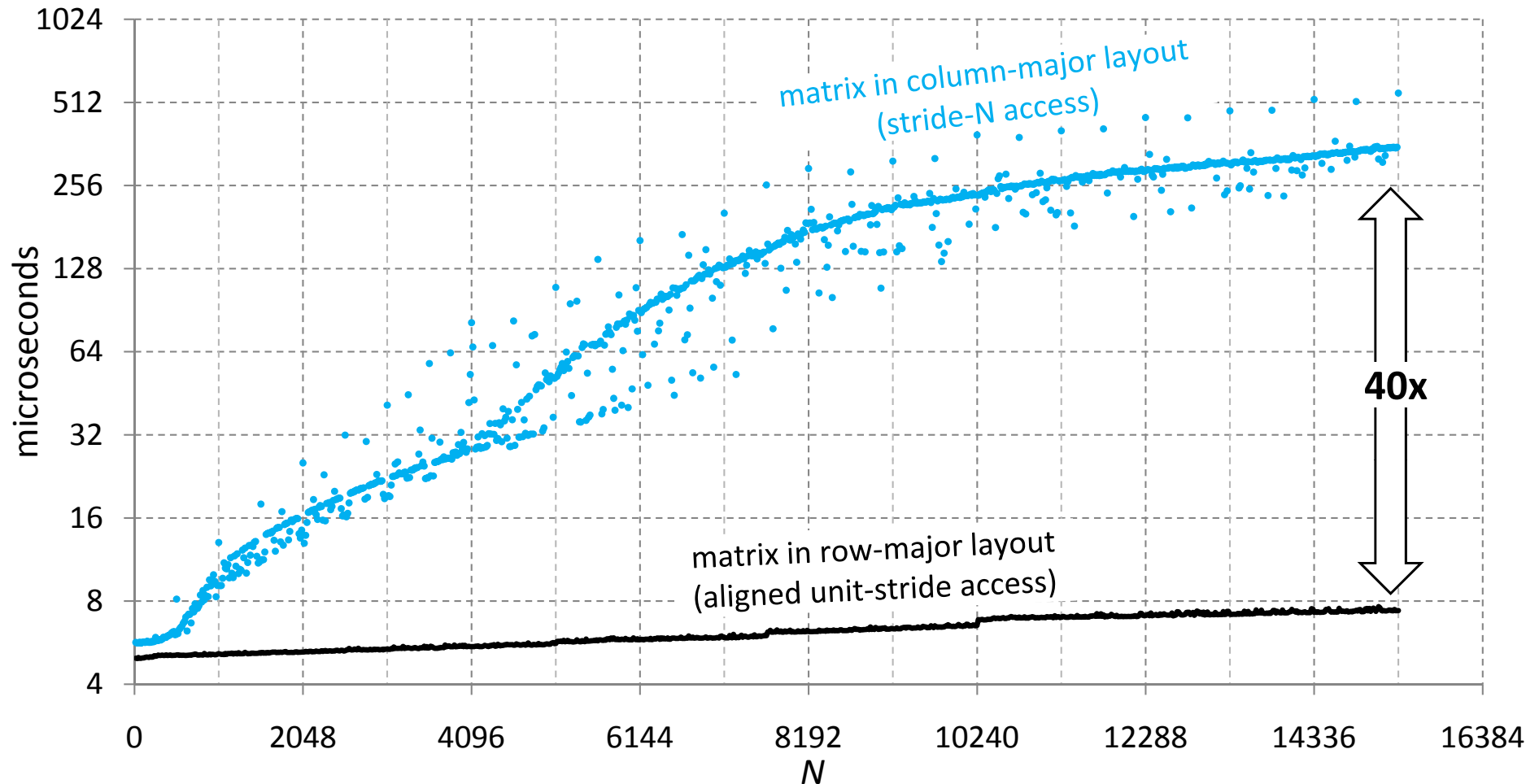
# The Progress So Far



- 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

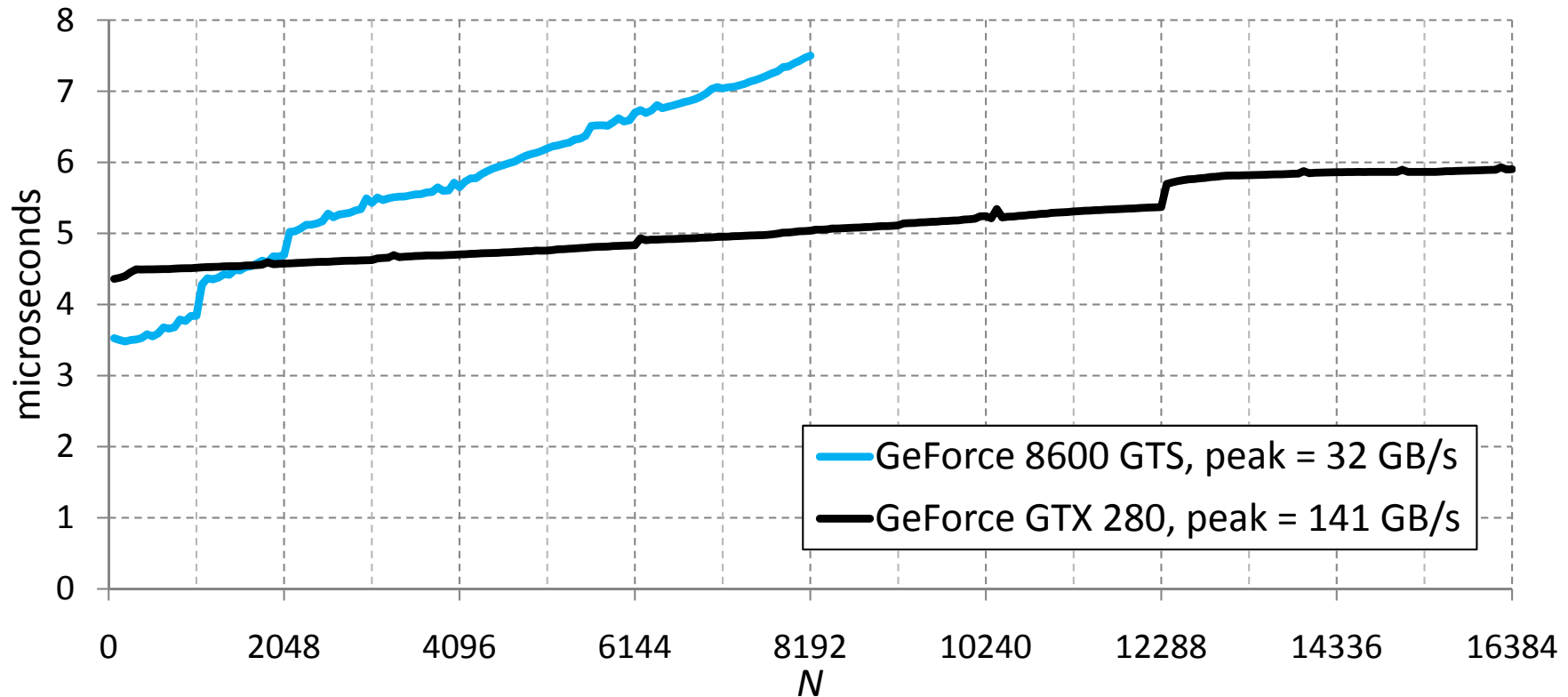
Exchange two rows of an  $N \times N$  matrix (SSWAP in CUBLAS 2.0):



Row pivoting in column-major layout on GPU is very slow  
This alone consumes half of the runtime in naïve SGETRF

# BLAS1 Performance

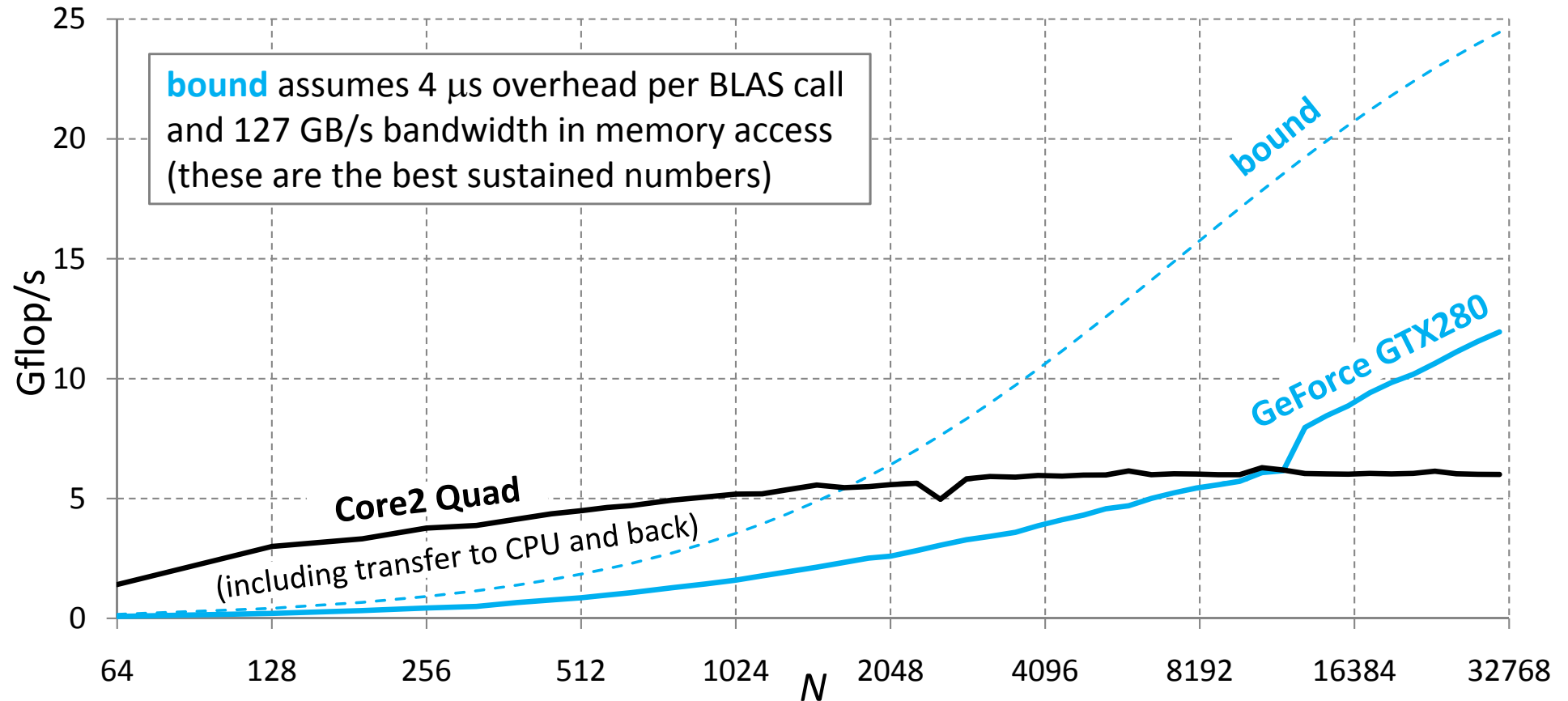
Scale a column of an  $N \times N$  matrix that fits in the GPU memory (assumes aligned, unit-stride access)



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

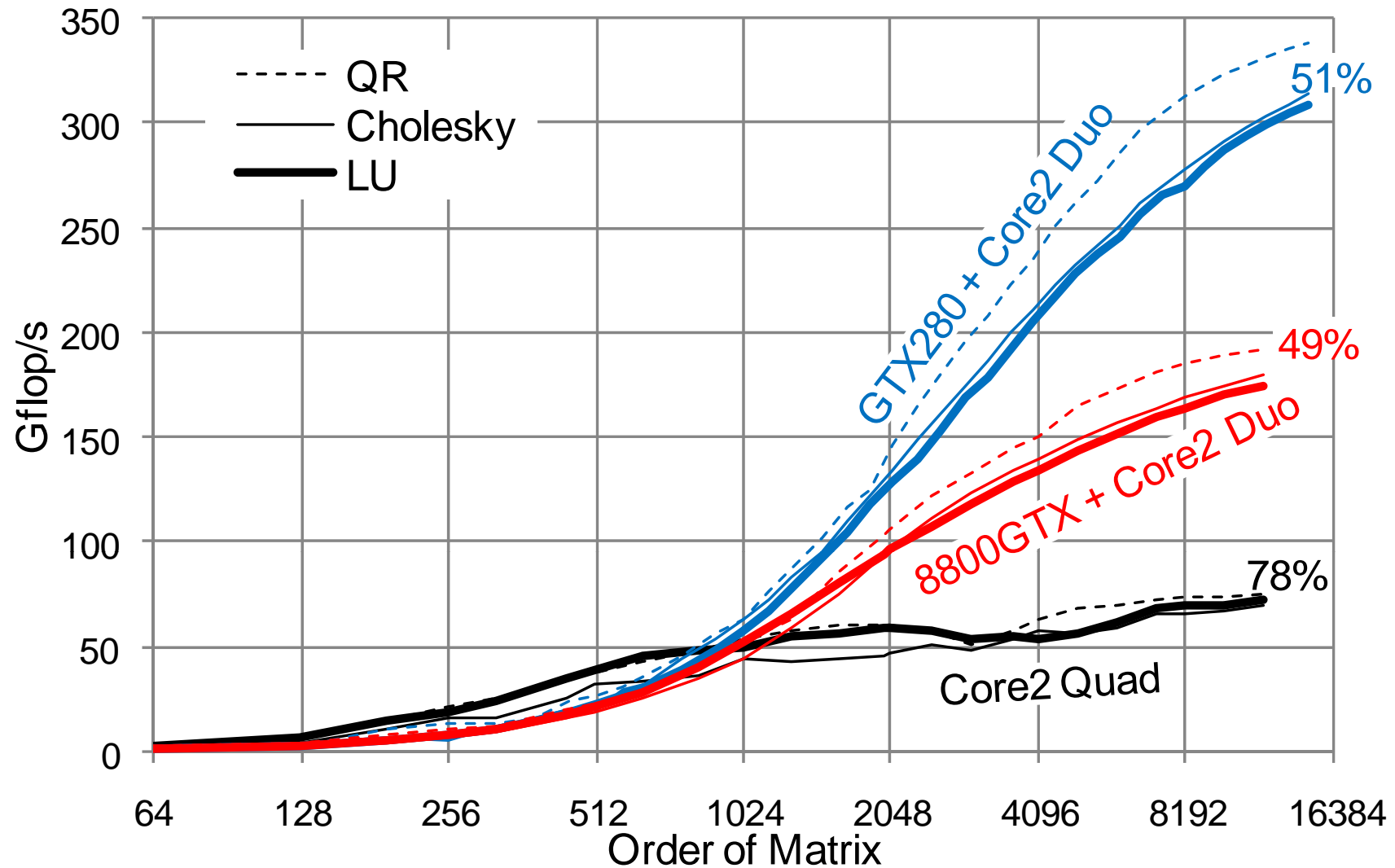


- Invoking small BLAS operations on GPU from CPU is slow
- Can we call a sequence of BLAS operations from GPU?
  - Requires barrier synchronization after each parallel BLAS operation
  - Barrier is possible but requires sequential consistency for correctness

# Fast Matrix Factorizations using GPUs

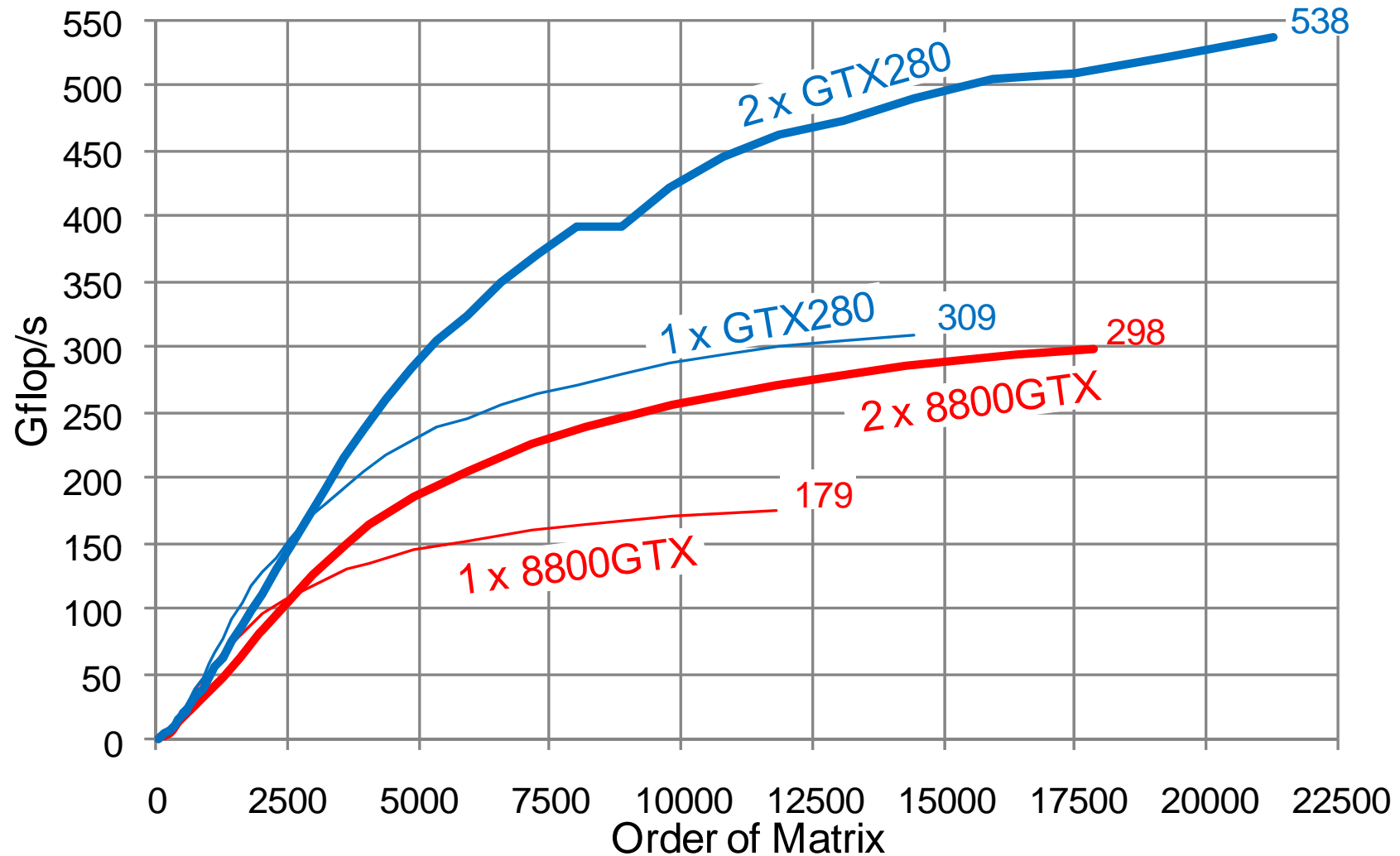
- Use GPU to compute matrix-matrix multiplies only
- Factorize panels on the CPU
- Use look-ahead to overlap computations on CPU and GPU
- Use right-looking algorithms to have more threads in SGEMM
  - Better load balance in the GPU workload, better latency hiding
- Use row-major layout on GPU in LU 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  $\|L^{-1}\| < \textit{fixed\_threshold}$
  - 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



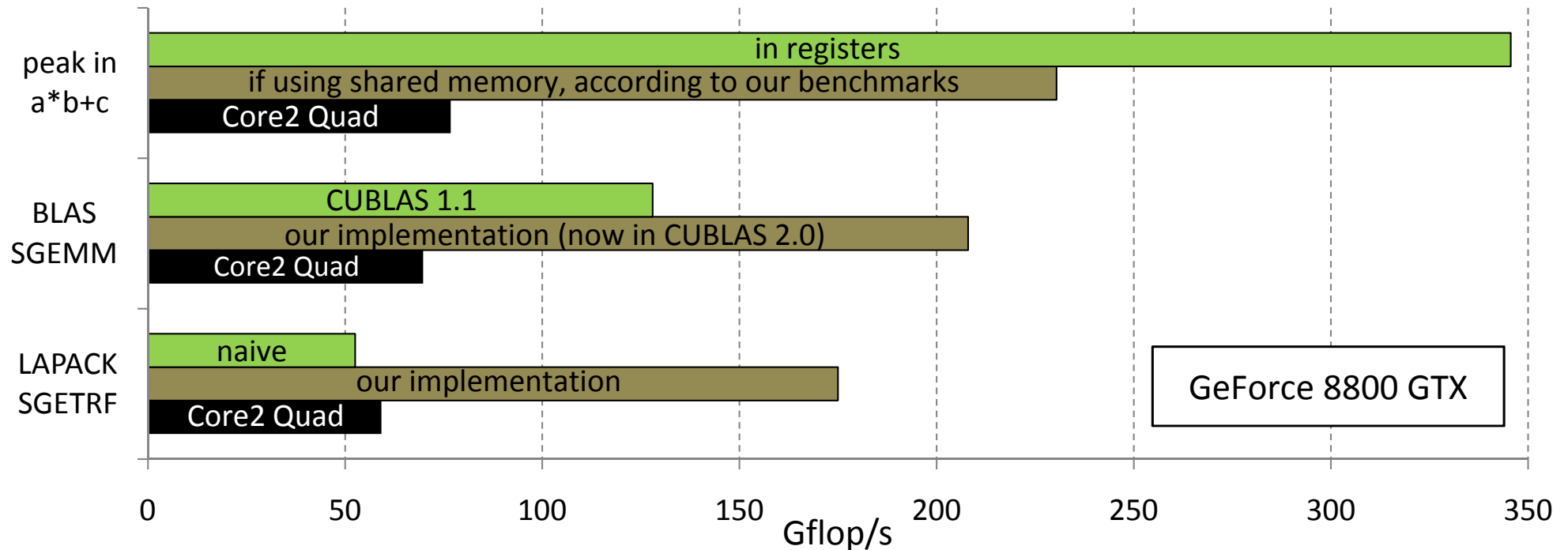
Our solution runs at ~50% of the system's peak (shown on the right)  
It is bound by SGEMM that runs at 60% of the GPU-only peak

# 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