Experience in accelerating linear algebra using GPUs

Vasily Volkov
UC Berkeley

October 6, 2011

Agenda

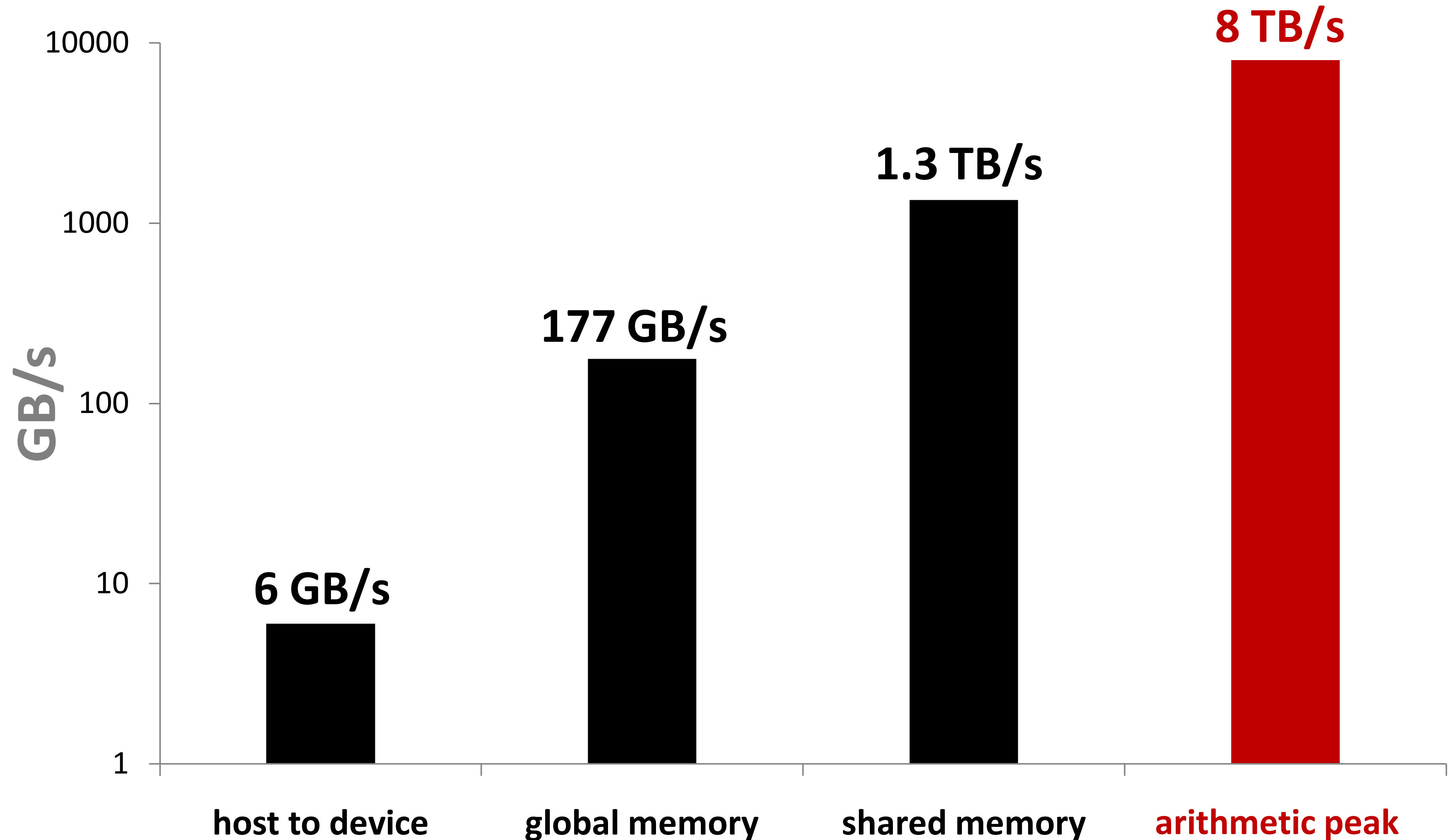
A few optimization patterns:

- Optimizations around bandwidth
- Optimizations for small problems
- Running faster by doing more flops

Part I:

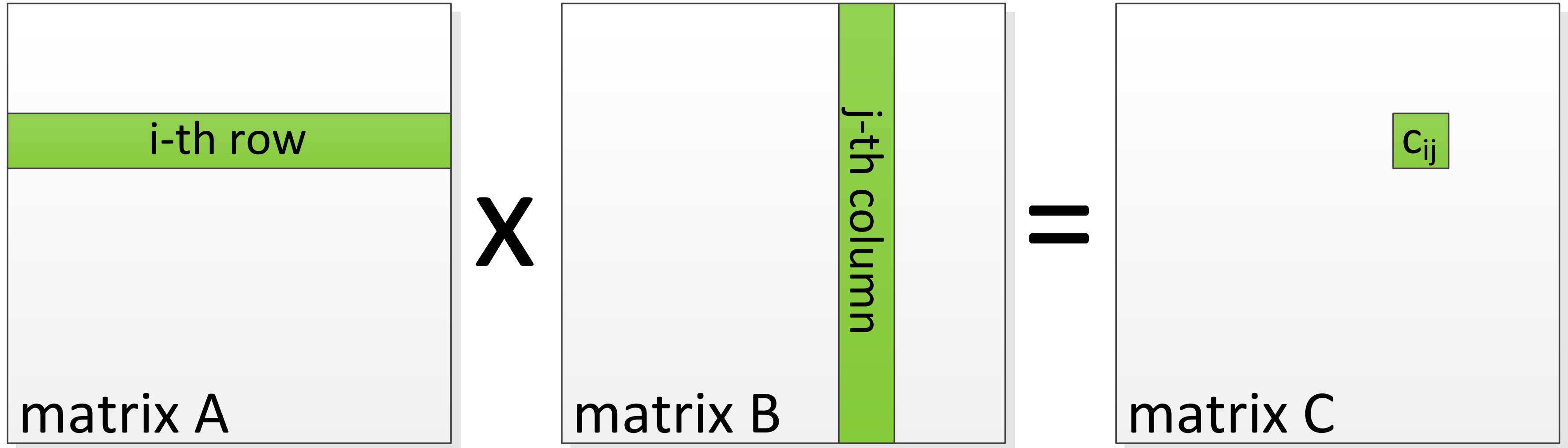
Workarounds around bandwidth

Memory hierarchy in a GPU system



Either can be the bottleneck

Simple example: matrix multiply



Each entry is computed as dot product:

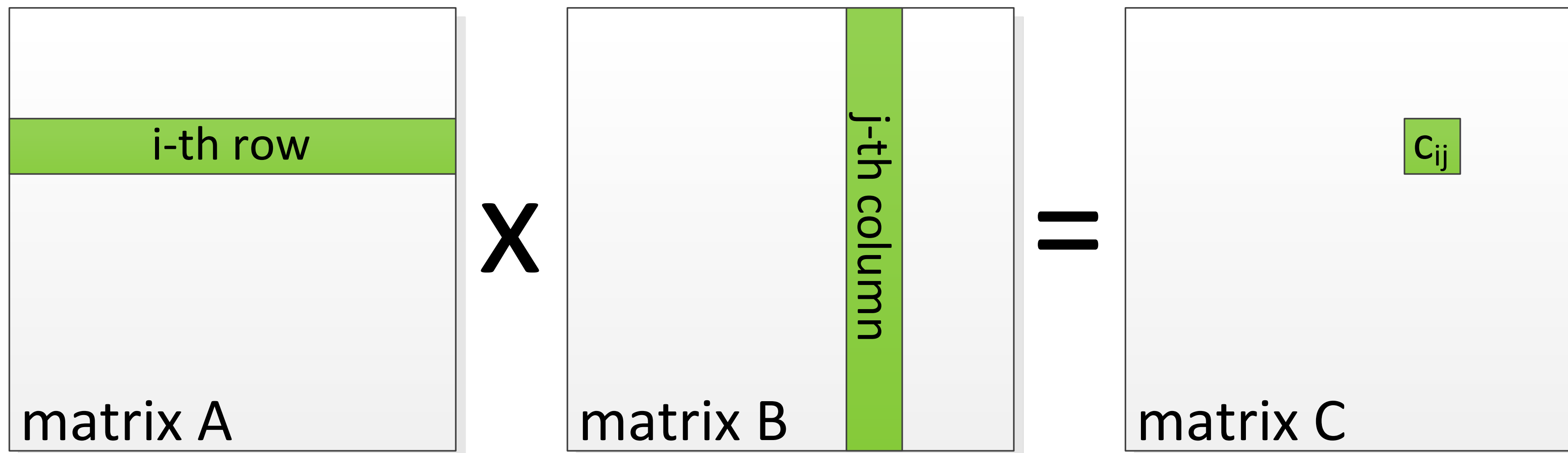
$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

Suppose we feed a_{ik}, b_{kj} from global memory

– This is 177 GB/s = 44 Gwords/s

– Need 2 words per 2 flops = 44 Gflop/s

How much faster with shared memory?



Each entry is computed as dot product:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

Suppose we feed a_{ik}, b_{kj} from shared memory

- This is 1344 GB/s = 336 Gwords/s
- Need 2 words per 2 flops = 336 Gflop/s

Multiply-add in shared memory

How a[i] += b[i]*c[i] is executed
if a[], b[],c[] are in shared memory?

- Load a[i]
- Load b[i]
- Load c[i]
- Multiply-add
- Store new a[i]

On GF100: each load costs 2 arithmetic operations

- 3 loads cost 6x more than 1 multiply-add

Blocking saves bandwidth

- Must use registers to approach peak
- But: there are only 63 registers per thread
  - Use blocking
- Consider using *b*x*b* blocks
  - Load *b*x*b* submatrices from A and B: $2b^2$ words
  - Multiply them: $2b^3$ flops
  - Implies we do ***b*** flops per 1 word loaded
    - ***b*** times better than before

How big blocks do we need?

Assume feeding data from global memory (44 Gwords/s)

- Need to do 1344/44 = 31 flops per word
- I.e. 31x31 blocks are necessary to get the peak

Assume feeding from shared memory (366 Gwords/s)

- Need to do 1344/366 = 4 flops per word
- So, 4x4 blocks are necessary

Reality (CUBLAS):

- 6x6 register blocks
- 96x96 shared memory blocks
- Still, only around 60% of arithmetic peak

History of blocking

1952: First blocking in matrix multiply

Rutledge, J., Rubinstein, H. 1952. *High order matrix computation on the UNIVAC*. ACM '52, 181-186.

1969: First analysis of I/O complexity for MM

McKellar, A.C., Coffman, E.G., Jr. 1969. *Organizing matrices and matrix operations for paged memory systems*. CACM 12, 3, 153-165.

1981: First lower bound of I/O complexity for MM

Hong, J. W., Kung, H. T. 1981. *I/O complexity: The red-blue pebble game*. STOC '81, 326-333.

2003: First register blocking in matrix multiply on GPU

Moravánszky, A. 2003. *Dense Matrix Algebra on the GPU*.

There are lots of similar prior work

- Known under different names:
  - Out-of-core algorithms
  - External memory algorithms
  - Cache efficient algorithms
  - Communication-avoiding algorithms
- Same big idea, different low-level details
- Challenge: find matching solutions for GPUs

Example: FFT

- Idea: do more work locally in threads
- Cooley-Tukey FFT:
  - Divides big FFT into many smaller FFTs
- **Consider FFT on N = M*K**
  - Do *M* FFTs of size *K*
  - Multiply element-wise by constant twiddle factors
  - Shuffle the data
  - Do *K* FFTs of size *M*
- Apply until FFTs are small enough to fit in registers
  - But not too many times to keep fewer shuffles
- Reality (CUFFT): 8 to 16 pt FFTs in registers

More register locality = less occupancy

- Occupancy vs register locality: competing tradeoffs?
- Not necessarily
  - Occupancy is good, but *not* required
  - ILP also contributes to latency hiding
    - (ILP = Instruction Level Parallelism)
  - Register blocking tends to produce more ILP

- Here is an example with SGEMM for G80:

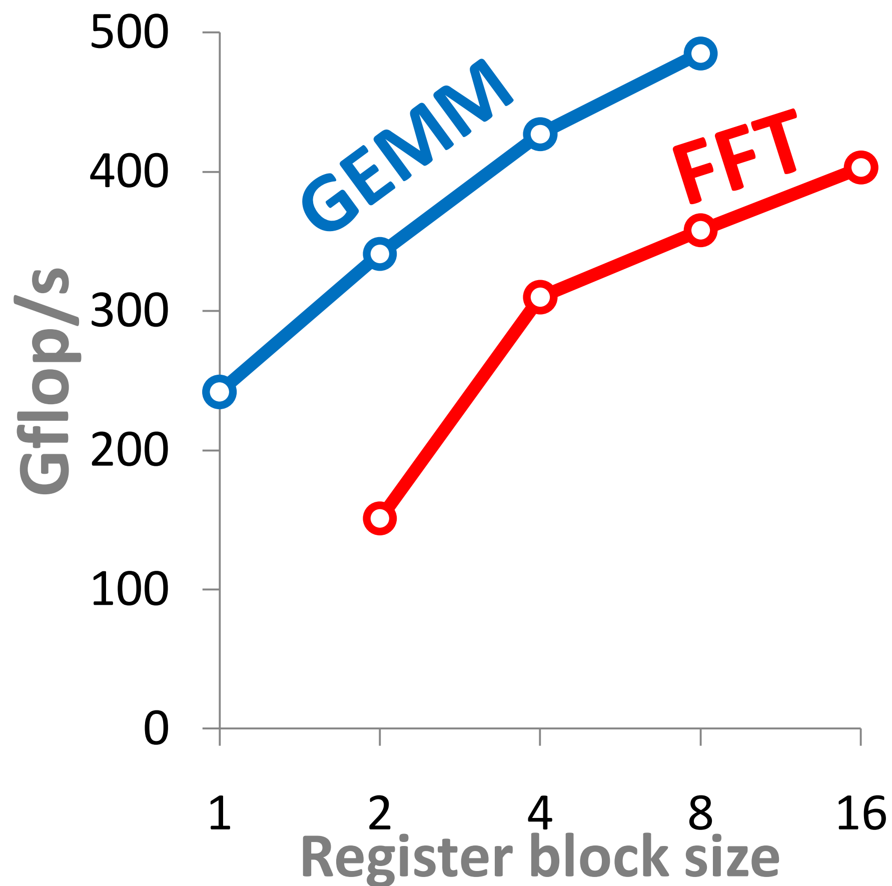
```
__global__ void sgemmNN( const float *A, int lda, const float *B, int ldb, float* C, int ldc, int k, float alpha, float beta )
{
    A += blockIdx.x * 64 + threadIdx.x + threadIdx.y*16;
    B += threadIdx.x + ( blockIdx.y * 16 + threadIdx.y ) * ldb;
    C += blockIdx.x * 64 + threadIdx.x + (threadIdx.y + 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];
}
```

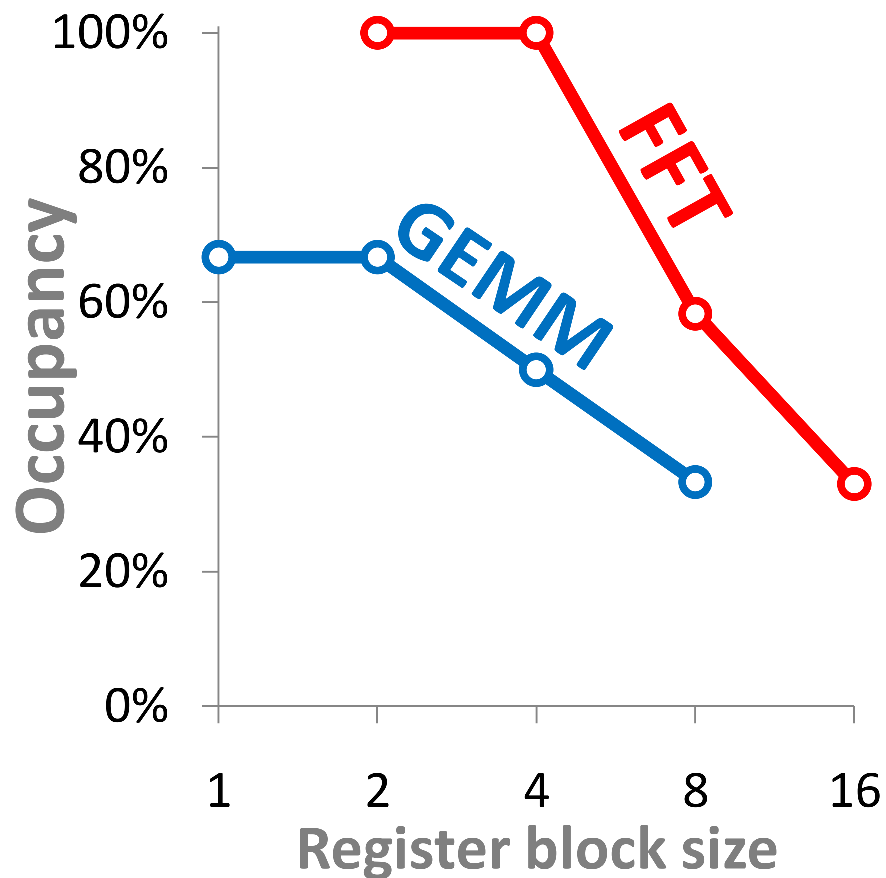
Register locality

ILP

ILP

Register locality: faster w/ fewer threads





Better locality may be better than better occupancy

Shared memory vs registers: size

- Shared memory is too small for shuffle in FFT
  - Consider 1024-pt FFT per thread block
  - This 8KB data + padding
  - Only 16 KB available on G80/GT200
  - Thus, only 1 thread block fits per multiprocessor
    - Devastating performance
- Solution: shuffle real and imaginary data separately
  - Now, enough shared memory
  - But either way, enough *registers* to fit both

Inverse memory hierarchy

- 48KB+16KB L1 storage per multiprocessor on GF100
- But 32768 registers - this is 128 KB!
- 2x more registers than entire L1 storage
  - Was 4x on GT200
- Aggregate L1 storage is also larger than L2 cache

- CPUs develop in the same direction:
  - SIMD and hyper-threading scales faster than L1 caches

New word in a classical concept?

Burk, Goldstine, von Neumann, 1946:

—*"We are therefore forced to recognize the possibility of constructing a* ***hierarchy of memories****, each of which has greater capacity than the preceding but which is less quickly accessible."*

(Konrad Zuse's Z1 in 1930s already had 2 levels)

Finite difference stencils

- Goal: apply 7-point stencil to 3D array
- Usually bandwidth bound, use blocking
  - Keep three BxB input slices in "fast memory"
  - Applying stencil gives one (B-2)x(B-2) output slice
  - I.e. produces $(B-2)^2$ outputs per every B^2 inputs
  - Waste less bandwidth if B is large
- Store 2D slices in registers!
  - Distributed across (B-2)x(B-2) thread block
- Have to keep middle slice into shared memory
- 27-point stencil: 1 input and 3 output slices

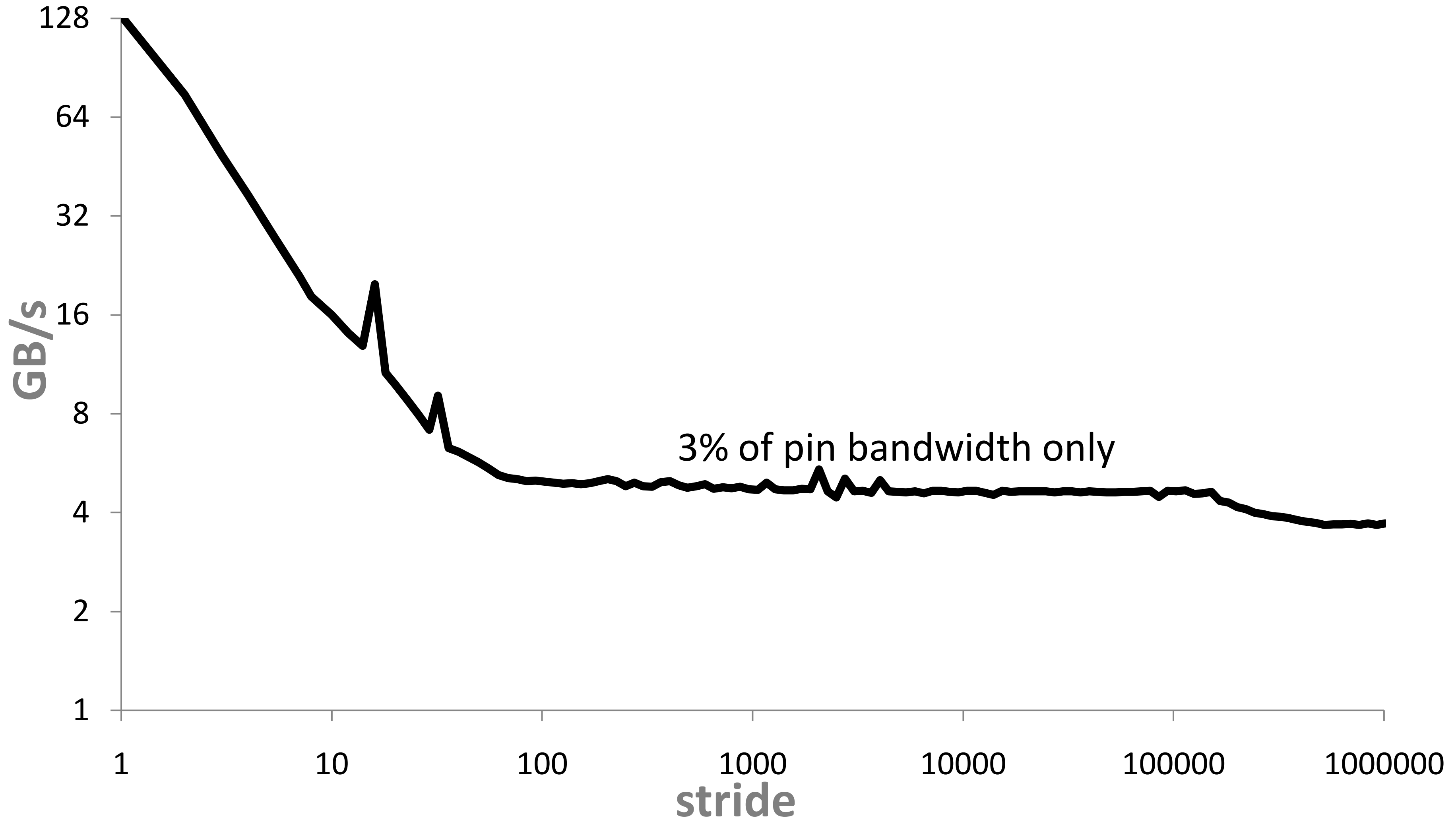
Finite difference stencils: references

- Original work:
  - Datta et al. SC08
  - Also: register blocking for bigger tiles
- Used in Reverse Time Migration:
  - Micikevicius, GPGPU-2, 2009
- Used in weather simulation:
  - Shimokawabe et al. SC10
  - Also: non-square tiles for fewer noncoalesced memory accesses

Random access to RAM is slow

- Ironically, Random Access Memory (RAM) underperforms in random access
  - 4B read pulls >=32B through memory interface
  - Can't open new DRAM pages too often
    - Bound by tRRD, tFAW, t32AW timing parameters
  - Random accesses thrash TLB
- *Spatial locality* is required for good performance

Lower spatial locality – lower throughput

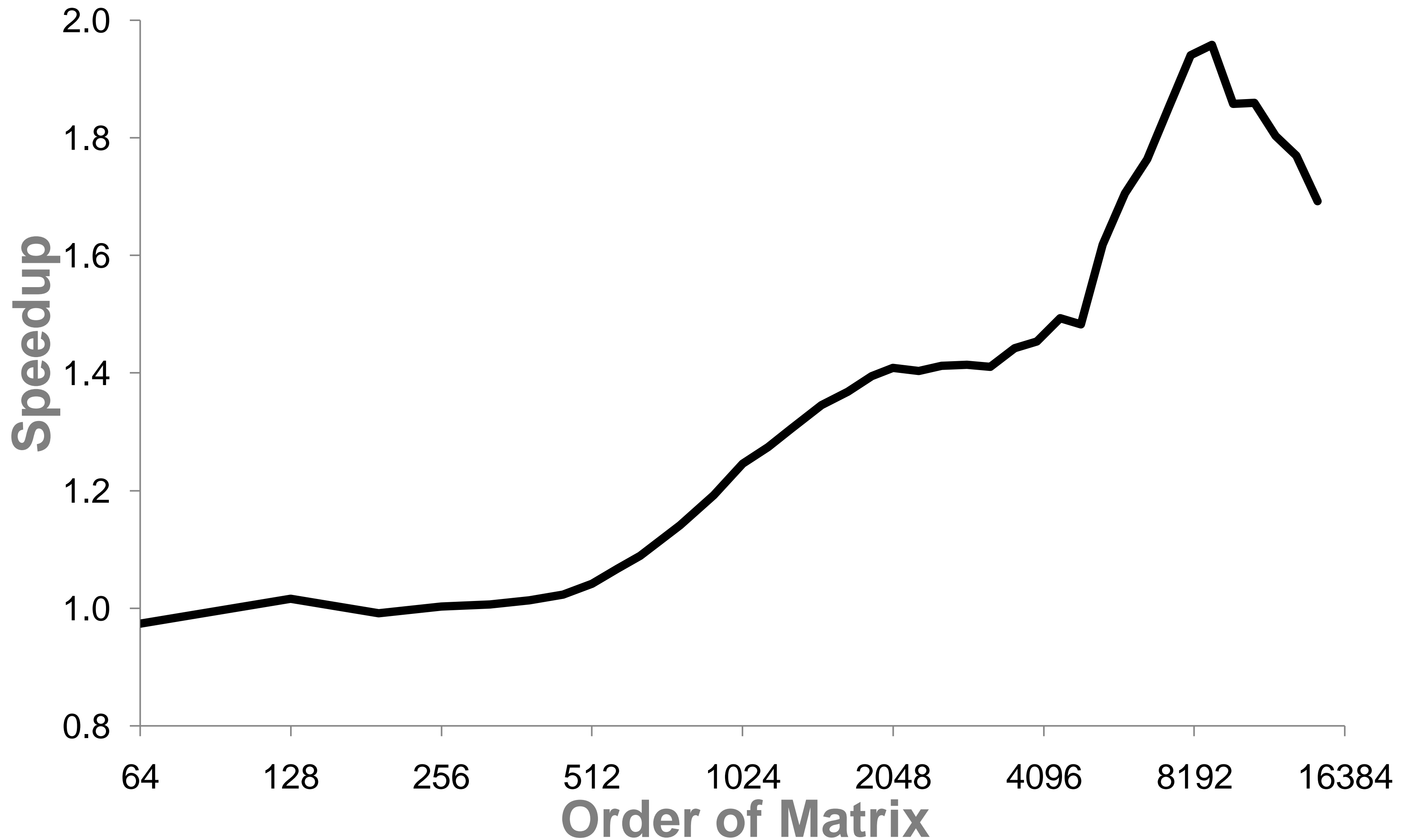


Thread *k* performs *array*[*k**stride*]++ in a big array

Workaround in LU factorization

- Problem: exchange matrix rows in column-major layout
- Solution: transpose the matrix
  - We transpose at entry and exit to LU factorization
  - Also, transpose panels during the factorization
- Have to pay small extra cost for the transposes
  - But avoid expensive strided access

Impact on LU factorization (GTX280)



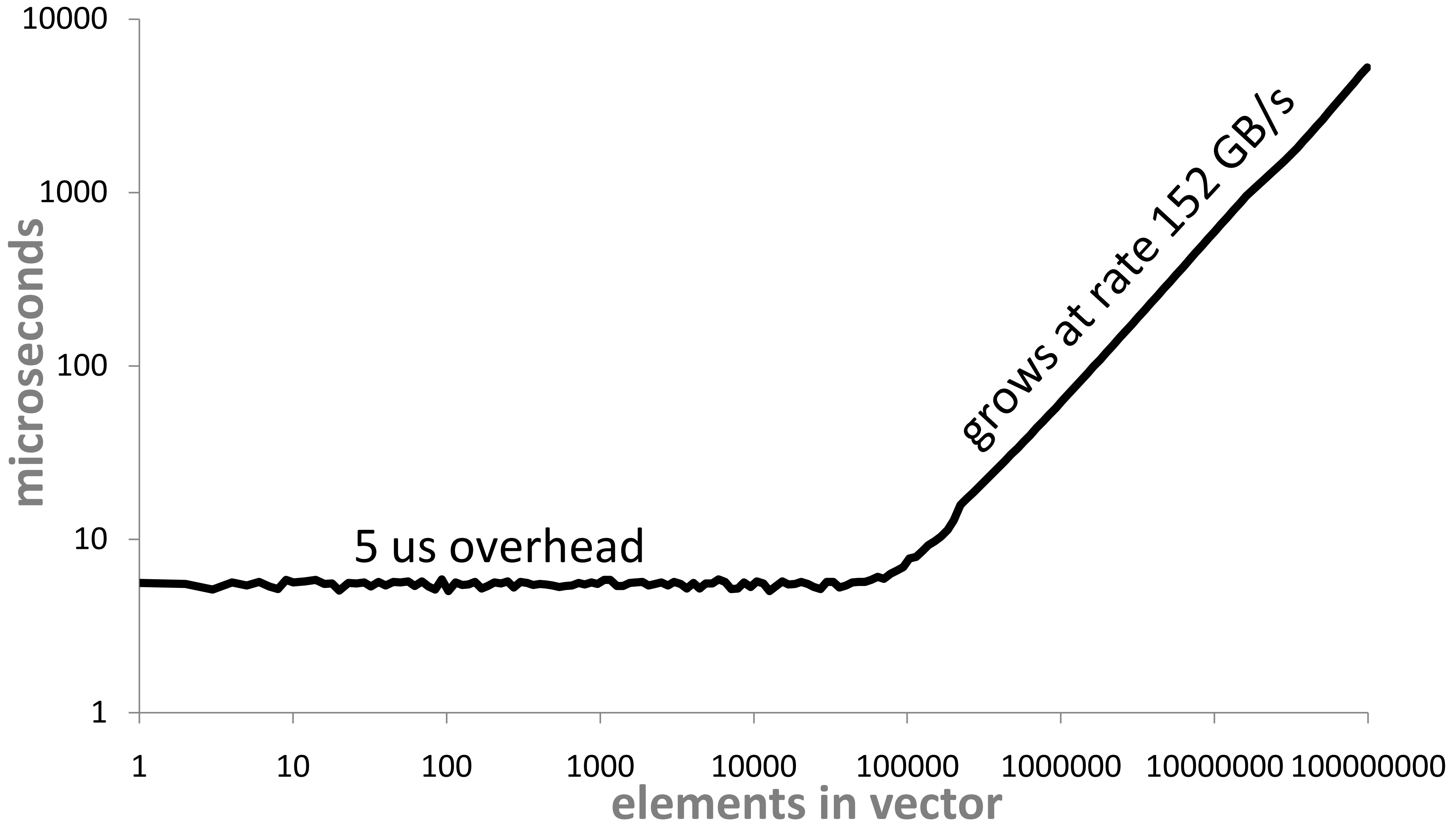
What if can’t avoid scattered access?

- Such as in hash
- Then might not need to optimize anything else
  - Can do ~30 flops per word loaded if streaming access
  - **But ~1200 flops per word loaded if random access**
- Fast hash on GPU:
  - Alcantara et al. 2009, ACM TOG 28, 5, 154:1-154:9
- Simpler and faster hash on GPU:
  - Alcantara et al. 2011, GPU Computing Gems

Part II

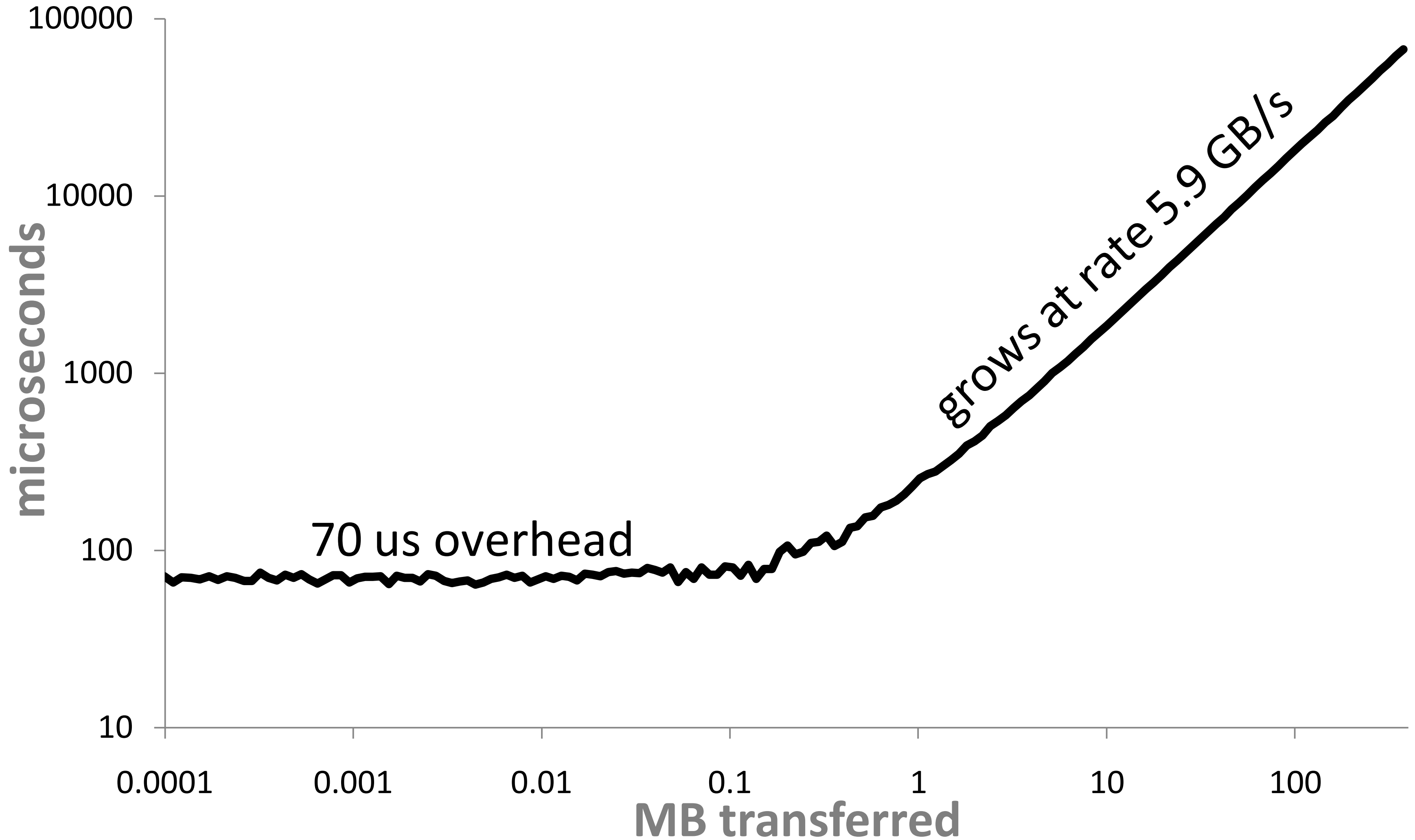
Problems with small problems

Bandwidth is not the only cost



Runtime for SSCAL in CUBLAS 4.0 on GTX480

Overhead/startup may dominate



Runtime for host to device memcpy, Windows Vista

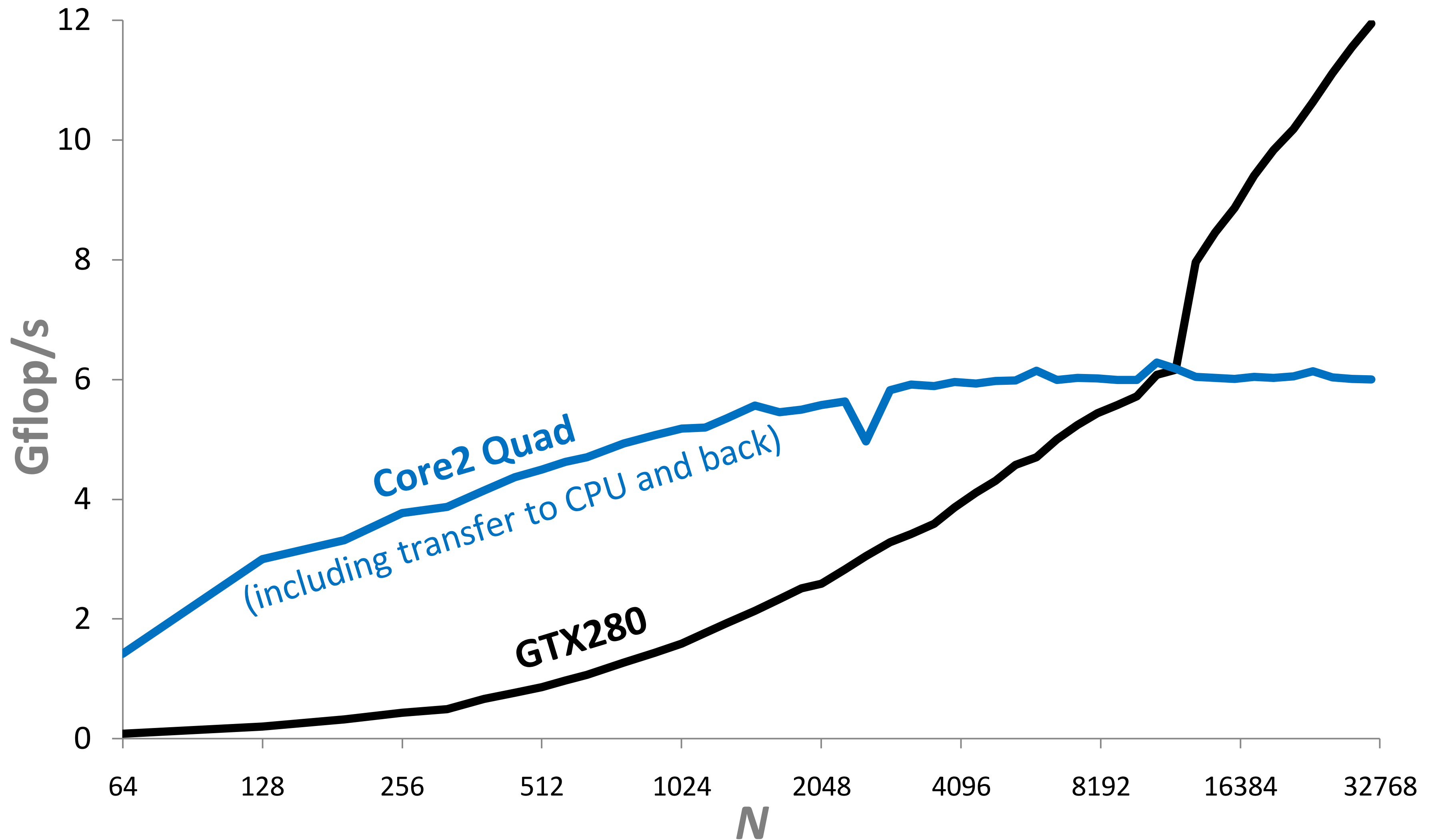
Performance dies for “small” problems

- Consider factorizing 1000x1000 matrix
  - 1,000,000 entries – looks large enough
  - SGEMM runs at ~775 Gflop/s at these sizes
  - Can LU approach this?
- LAPACK: ~4N BLAS invocation for NxN matrix
  - Find pivot, swap rows, scale column, rank update
- Assume each invocation is 5us
  - 20 milliseconds in total for N = 1000
  - This is only 33 Gflop/s

Solution: heterogeneity

- **Solve small problems on CPU**
  - Transfer the data from GPU if necessary
  - Involves overhead, but may worth it
- LU factorization:
  - Factorize panels on CPU
  - Do the rest (GEMM, pivoting) on GPU

Factorizing Nx64 matrices



Faster to solve on CPU, even if transfer cost included

Solution: batch processing

- **Solve many small problems on GPU at once**
  - This is how small FFTs are done fast
  - Batch pivoting in LU factorization (64 per kernel)
- Same 1000x1000 matrix example:
  - Now assume doing 10 factorizations at once
  - Still need only ~4N BLAS invocations
    - But now it is custom BLAS, each call works on 10 independent matrices
  - So, still 20 millisecond bound, but 10x more flops
  - Get a better 333 Gflop/s bound now

Solution: persistent threads

- Launch 1 kernel that does all the work
  - Threads are recycled to do different work
- How to resolve dependencies?
  - I.e. threads need data produced by other threads
  - Must synchronize
- Custom global synchronization?
  - Still exists mostly at experimental level

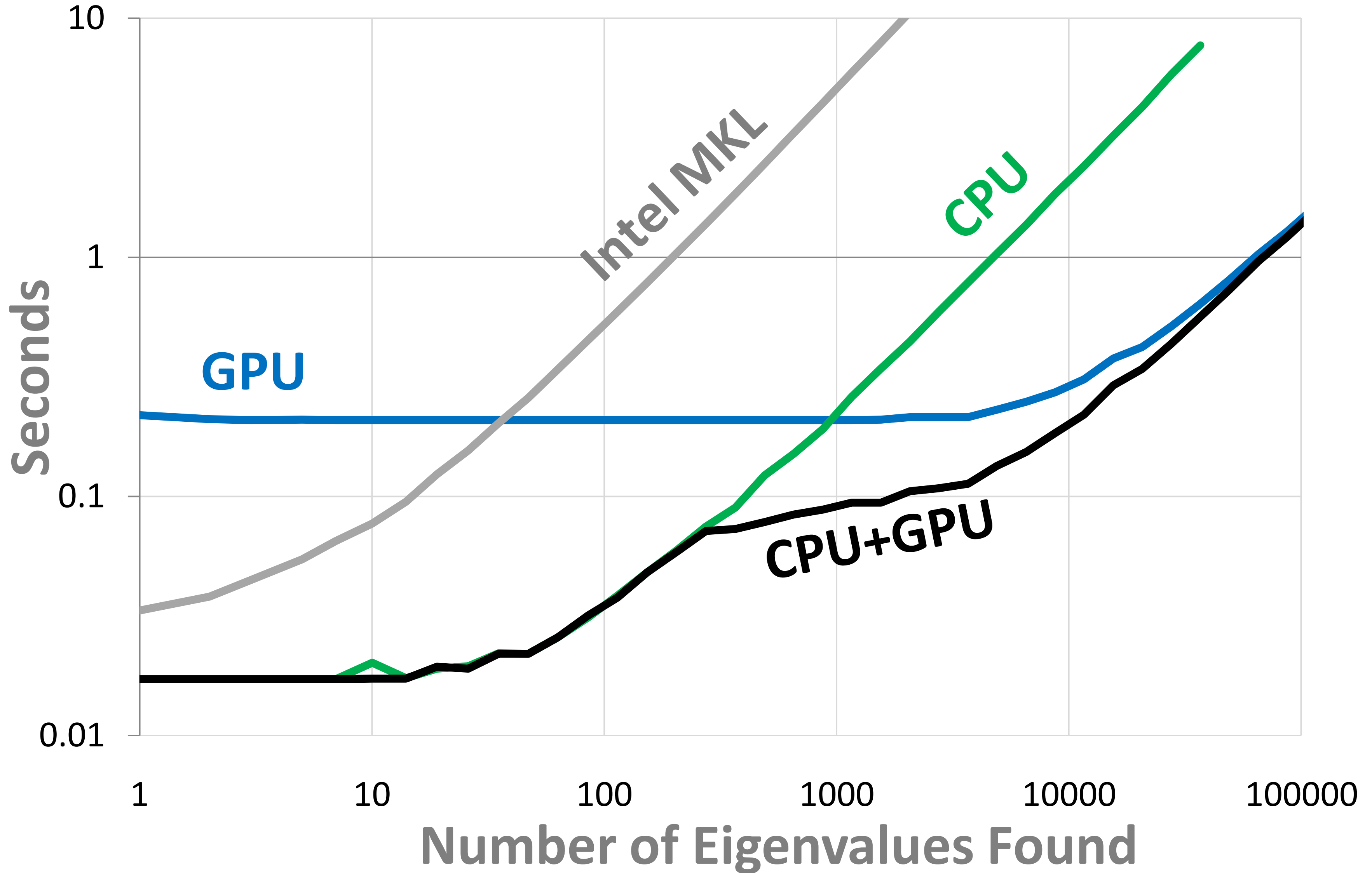
Problem: tridagonal eigenvalue solver

- Real, symmetric, tridiagonal NxN matrix T
- Bound spectrum w/ Gershgorin theorem
  - Gives interval [L,R] where all N eigenvalues are
- Bisect the interval using inertia theorem
  - Factorize $T - xI = LDL^T$ for x = (L+R)/2
  - Count negative entries in *D* (serial algorithm)
  - This equals #eigenvalues in left half of [L,R]
- Now we have two intervals
  - Bisect them in parallel, recursively
  - Stop when intervals get small enough
  - Discard intervals that contain no eigenvalues

Where to run it?

- Run on GPU when many intervals, on CPU otherwise
- How to decide when is “many”?
  - We build an empirical runtime model
  - Estimate execution time on CPU and GPU
  - Choose the smallest one
  - Switch to GPU and back if necessary

Find subset of eigenvalues, N≈100,000



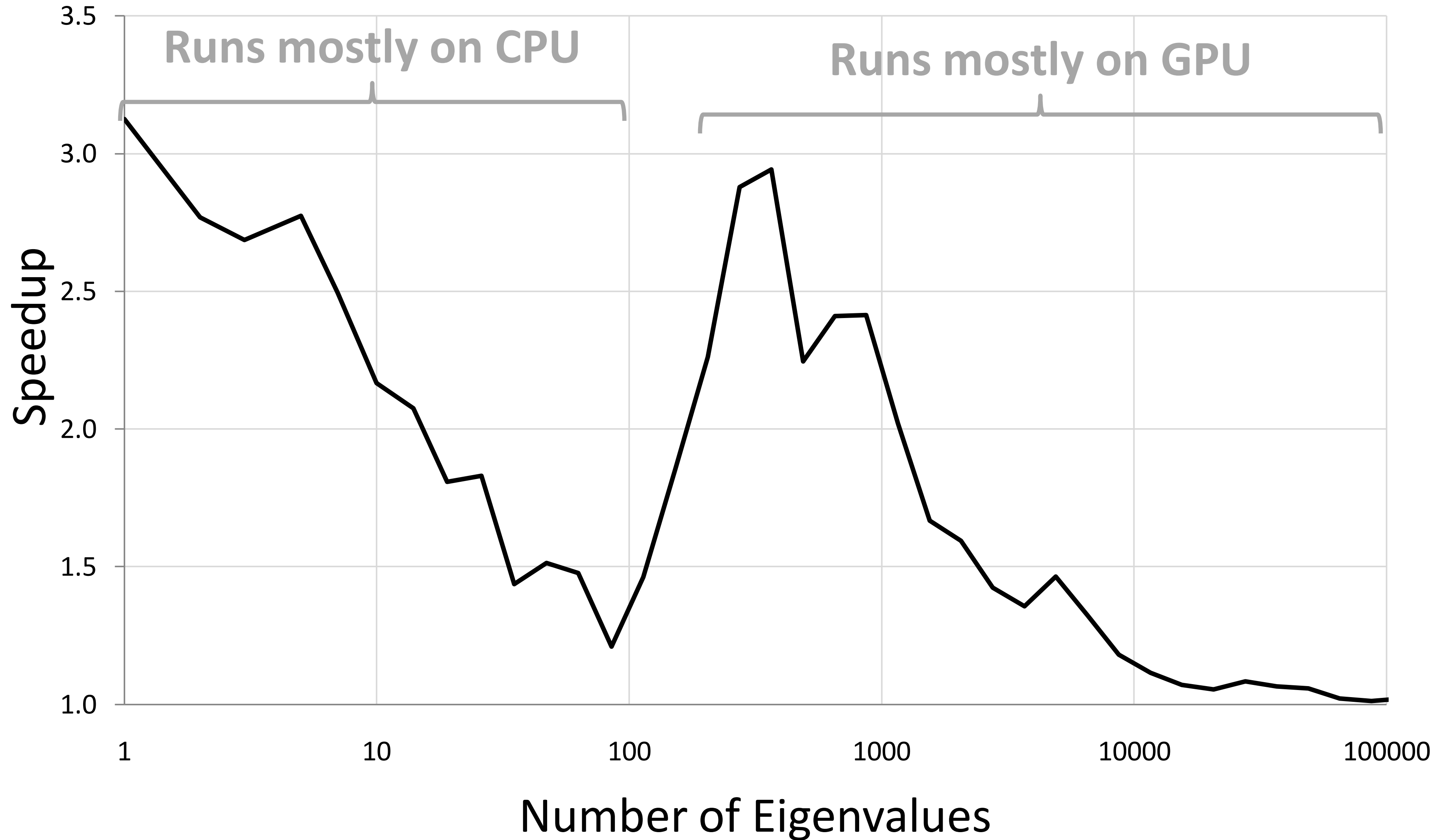
Part III

Flops are free, trade them for other costs

Multisection

- Lo et al. 1987; Simon 1989; Katagiri et al. 2006
- Few intervals to split: most resources are idle
- Split using multiple points for better utilization
- How many?
  - 4-section ≈ bisection applied twice
  - Prefer 4-section if it is faster than 2 bisections
  - In general, optimize log(M)/time(M) for M-section

Speedup with multisection, N≈100,000



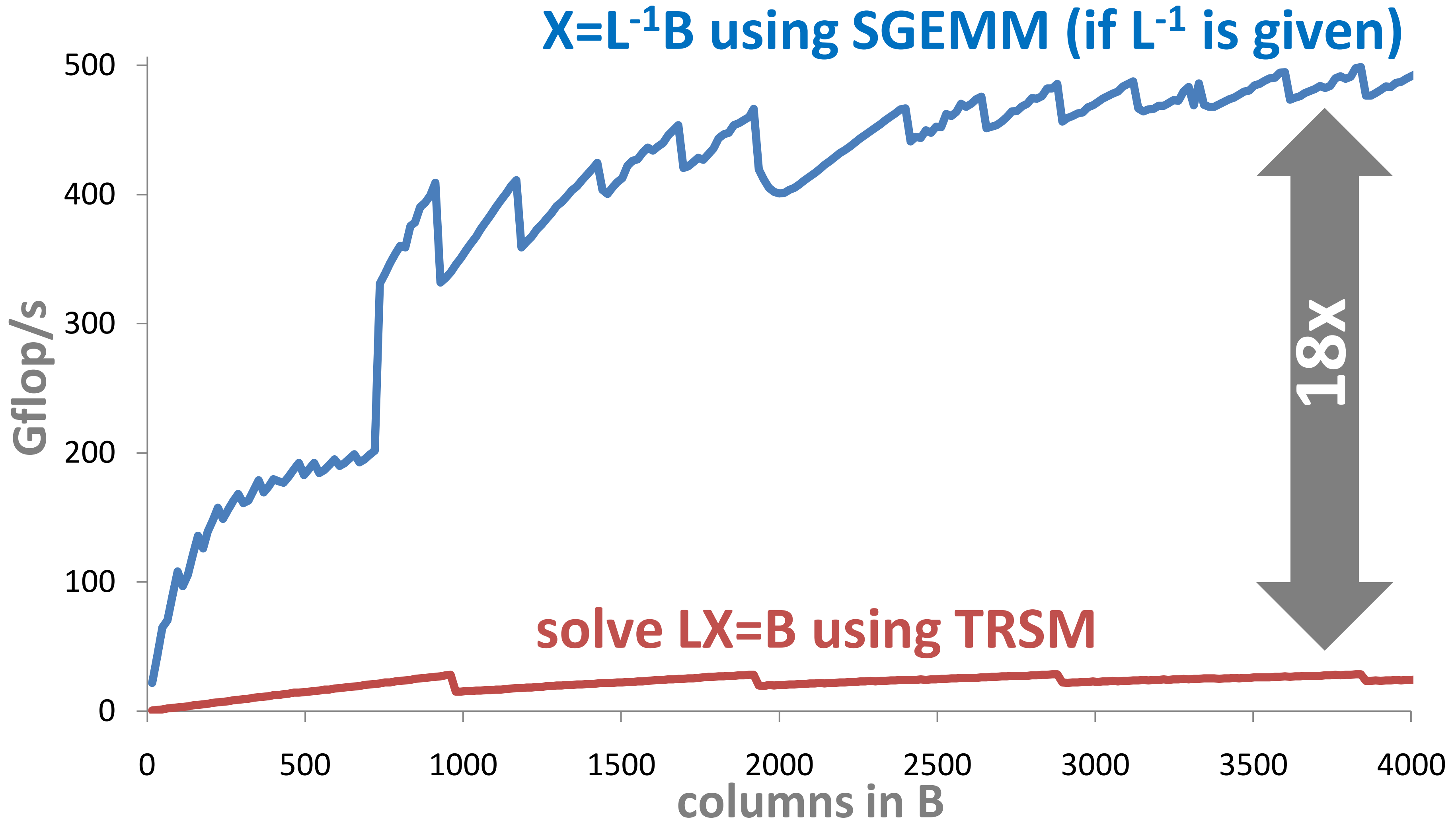
Tridiagonal eigenvector solver

- Inverse iterations
  - Solve $(T - \lambda I) z = x(i)$, $x(i+1) = z / ||z||$
  - LAPACK: use LU factorization (4N numbers)
  - Another 4N for reading/writing iterate twice
  - Bandwidth-bound for large enough problems
  - Cost: $8N*n_{iterations} + 4N$
- New solution:
  - Use LDL^T factorization with no pivoting
    - Fails in 0.06% cases in practice: recompute using safe solution
  - Store only D, reconstruct L on the fly from D and T
  - Cost: $5N*n_{iterations}+N$, 1.75x faster for $n_{iterations}=3$

Accelerating small triangular solves

- Need to solve LX=B in LU factorization
  - L is a small triangular matrix, e.g. 64x64
  - B is large, e.g. 64x10000
- Solution: forward substitution
  - Implemented in CUBLAS STRSM
  - Problem: very slow for small L

STRSM is much slower than GEMM

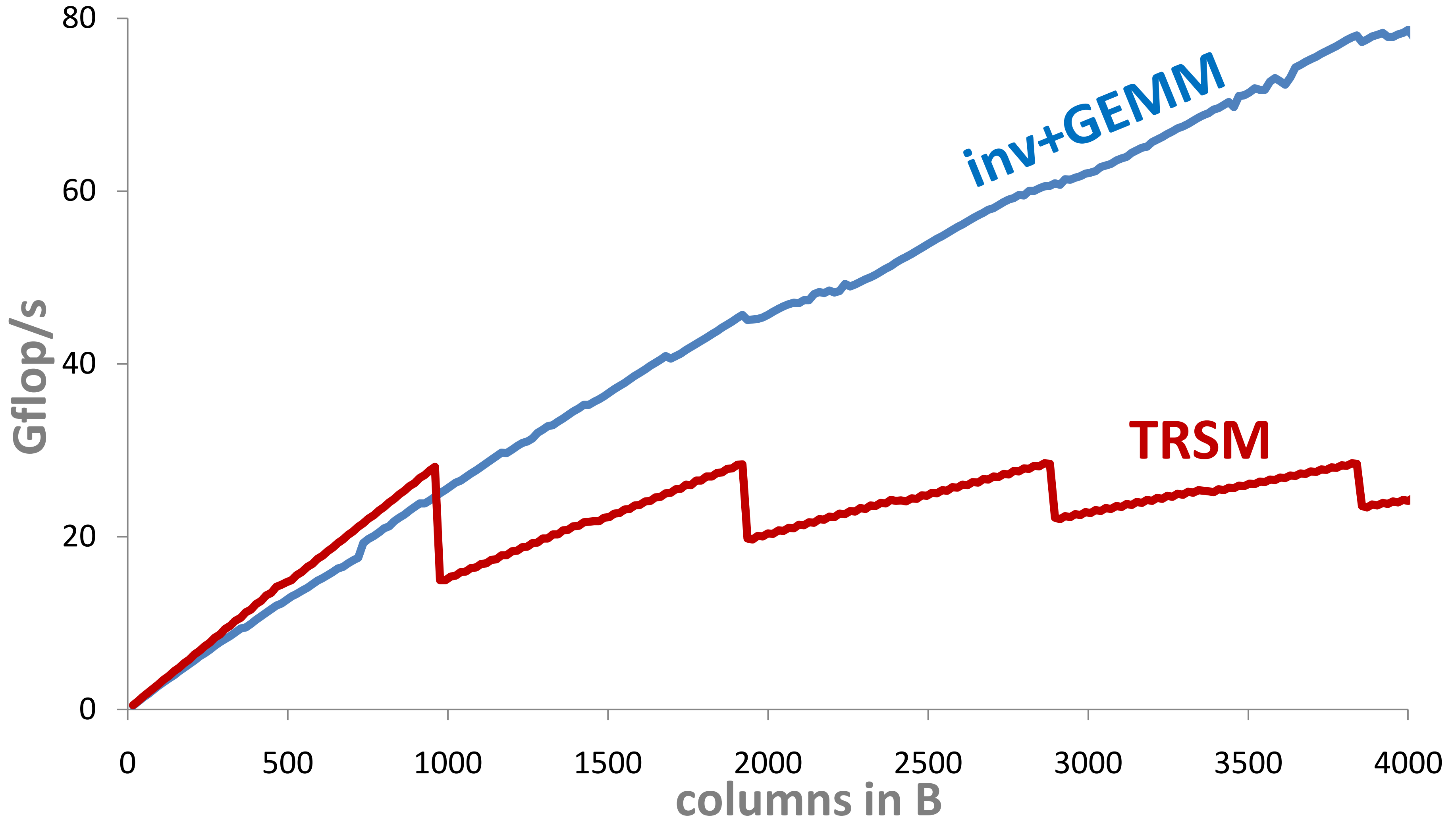


Measured on GTX480, CUBLAS 4.0, L is 64x64

Use inv+SGEMM instead of STRSM

- Solve LX=B as $X=L^{-1}B$ computing L^{-1} explicitly
  - Run TRSM to compute LY=I, I is identity matrix
  - Run GEMM to compute X = YM
  - 2x more work but at ~20x higher rate
- Numerically stable?
  - Yes, if $\|L^{-1}\|$ is small
  - This is usually the case in LU factorization
  - If $\|L^{-1}\|$ is large, run the usual algorithm
  - (Requires computing $\|L^{-1}\|$ norm)

Improvement if computing L^{-1} on GPU

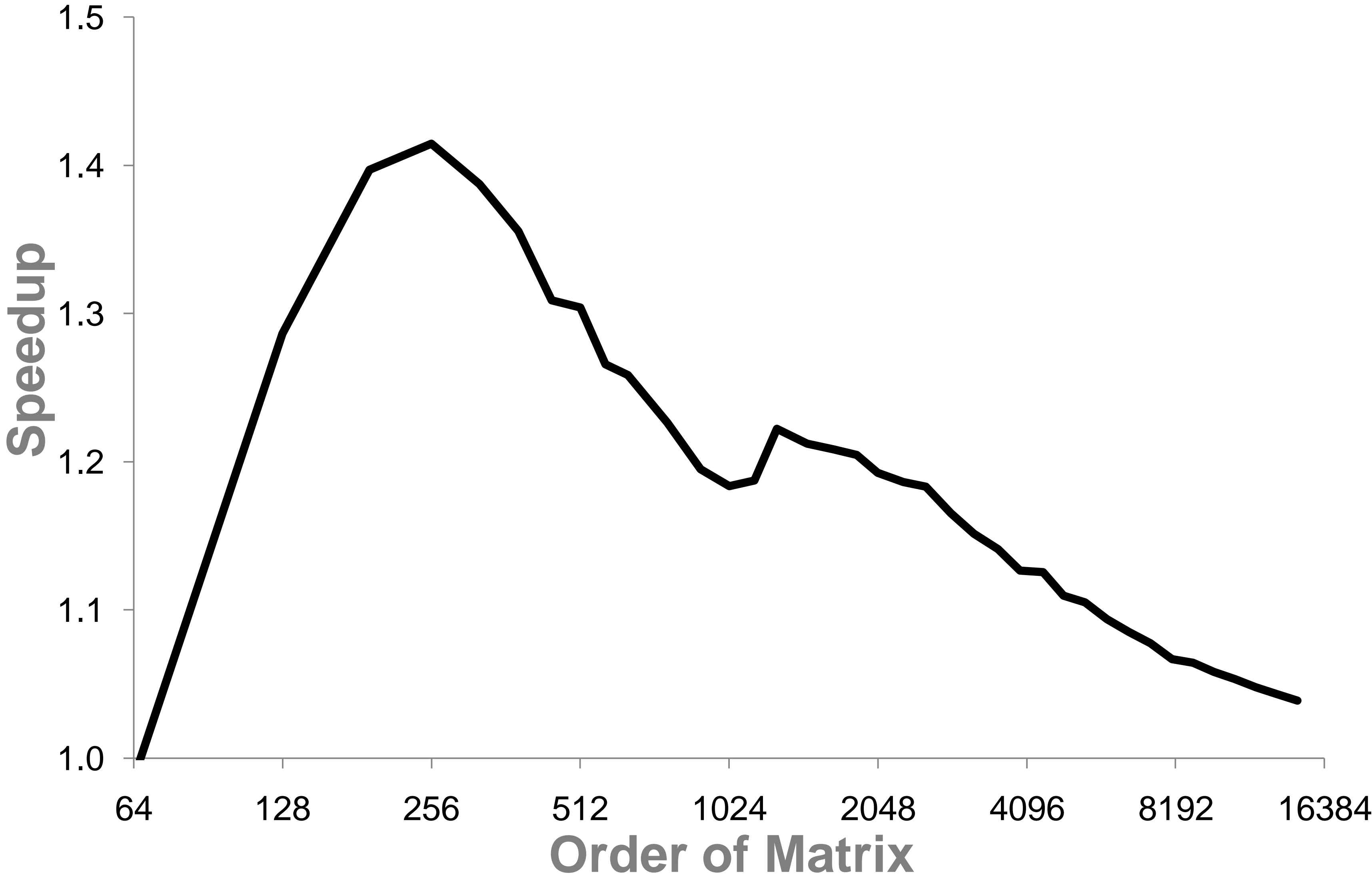


For Gflop/s rate here, flops counts are assumed same

Still nowhere close to GEMM rates?

- Computing L^{-1} on GPU takes 136 us
  - This is 30 Mflop/s or 0.002% of GPU peak
- Compute it on CPU instead
  - We do panel factorization on CPU anyway
  - More heterogeneity

Impact on LU factorization (GTX280)



Conclusion

- Must use register locality to approach peak
- Keep more data in the plentiful register space
- May worth to change layout for spatial locality
- Offload small problems to CPU
- Use free flops to alleviate other bottlenecks