Using GPUs to Accelerate Linear Algebra Routines

Vasily Volkov and James Demmel

1. BLAS libraries

= building block for dense linear algebra routines

Consists of three layers:
BLAS1 and BLAS2: memory-bound = slow
BLAS3: computation-bound = fast = key for fast dense linear algebra

NVIDIA released BLAS for GPU in 2007. Peaks:

GEMM (matrix-matrix multiply):	125 Gflop/s
TRSM (triangular solve):	70 Gflop/s
SYR2K (symmetric update):	37 Gflop/s
Theoretical peak of GPU:	346 Gflop/s

Contrast: on CPU all these BLAS3 routines run at ≈100% of theoretical peak

Our improvements:
TRSM and SYR2K: >100 Gflop/s by formulating them via NVIDIA's GEMM. Use left-looking version to have lower bandwidth requirements.

GEMM: >180 Gflop/s by using non-square blocks that yields better cache use (GPUs are cache-poor) and careful coding.

Can we do better? Why not the peak rate?

Memory latency is already hidden and maximum memory bandwidth is achieved – can't improve.
Stuck at the cache bandwidth? Can't achieve more than peak/1.5 = 230 Gflop/s if one of the operands is in shared memory (i.e. "cache").
Cycles are wasted in logic and pointer arithmetic — unlike on superscalar CPUs that hide it behind flops.

2. Dense LU factorization

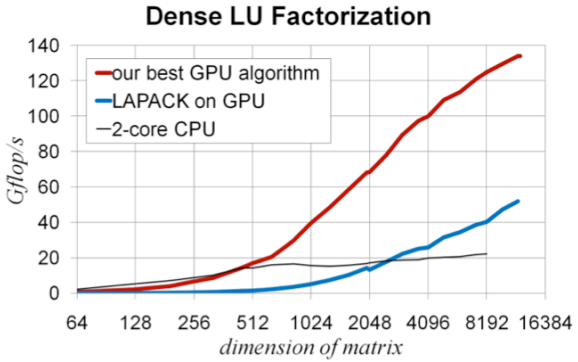
"#1 linear algebra algorithm"
≈ LINPACK benchmark

- Most of the work is done in BLAS3
- On the CPU runs nearly as fast as BLAS3
- Implemented in LINPACK, LAPACK, ScaLAPACK

LAPACK's code can be run on the GPU using NVIDIA's BLAS but is inefficient. Example: at n=11840 it spends 46% of total time in BLAS1 and BLAS2 doing 0.4% of total work. Reason: large kernel call overhead (~10μs), inefficiency of the GPU in solving sequences of small problems.

Our best design is **heterogeneous**:

Run BLAS1/BLAS2 codes on the CPU — there is not sufficient parallelism to benefit from the GPU anyway.
Overlap serial CPU computations (BLAS1/2) with parallel GPU computations (BLAS3).
Work with matrix in row-major format for faster row permutations (this requires extra transpositions).
Use left/up-looking algorithm to have lower bandwidth requirement in GEMM (faster than right-looking).
Substitute triangular solve with multiply by inverse matrix (for better parallelism and regularity). Stability ok?



3. Tridiagonal eigenvector solver (inverse iterations)

Embarrassingly parallel, bandwidth bound.
More complicated when eigenvalues are clustered.

- **trade bandwidth for flops**: store only part of factorization (only D in LDL^T). Reconstruct the rest upon need. (Best speedup vs. LAPACK so far: 120x.)
- Instead of using safe but slow codes, **use fast but reckless codes. If fails – solve on the CPU**. The bet is that it fails rarely.

4. Tridiagonal eigenvalue solver (bisection)

Most work is embarrassingly parallel, $O(n^2)$ flops.
- Traditionally vectorized.

The rest ($O(n)$ work) is nearly serial.

Such a solver is included in the NVIDIA SDK examples.
- Runs *entirely* on the GPU.
- Extreme effort applied to parallelize the $O(n)$ work.
- However: fails on many basic matrices.
- Difficult to parallelize O(n) part, not worth it.

Our solution: simple, reliable, fast.
- **GPU runs only embarrassingly parallel codes.**
- The rest is run on the CPU.
- **If problem is small, solve it entirely on the CPU**
- Increase the flop rate by doing useful redundant computations.
- Autotuning to decide the optimal redundancy and when involve the GPU.
- Use features of IEEE arithmetic to save another 15-30% of runtime.

Runs on an older GPU 2x faster than NVIDIA's code, run on a newer GPU; Up to 156x faster than LAPACK. Runs at up to 40 Gflop/s.

5. Conclusion

Our BLAS3 and LAPACK codes for the GPUs are the fastest known. The lessons:

Admit that GPU is not always faster.
Exploit heterogeneity: GPU is for large embarrassingly parallel tasks, CPU is for small or nearly serial tasks.
Trade fewer flops for lower bandwidth requirements.
Buy regularity and embarrassing parallelism.
Trade fewer, slower flops for more, faster flops.

Future work includes:
- parallelizing LU factorization over multiple GPUs
- other matrix factorizations
- eigenvector solver for clustered eigenvalues
- understand performance of GEMM on GPU better