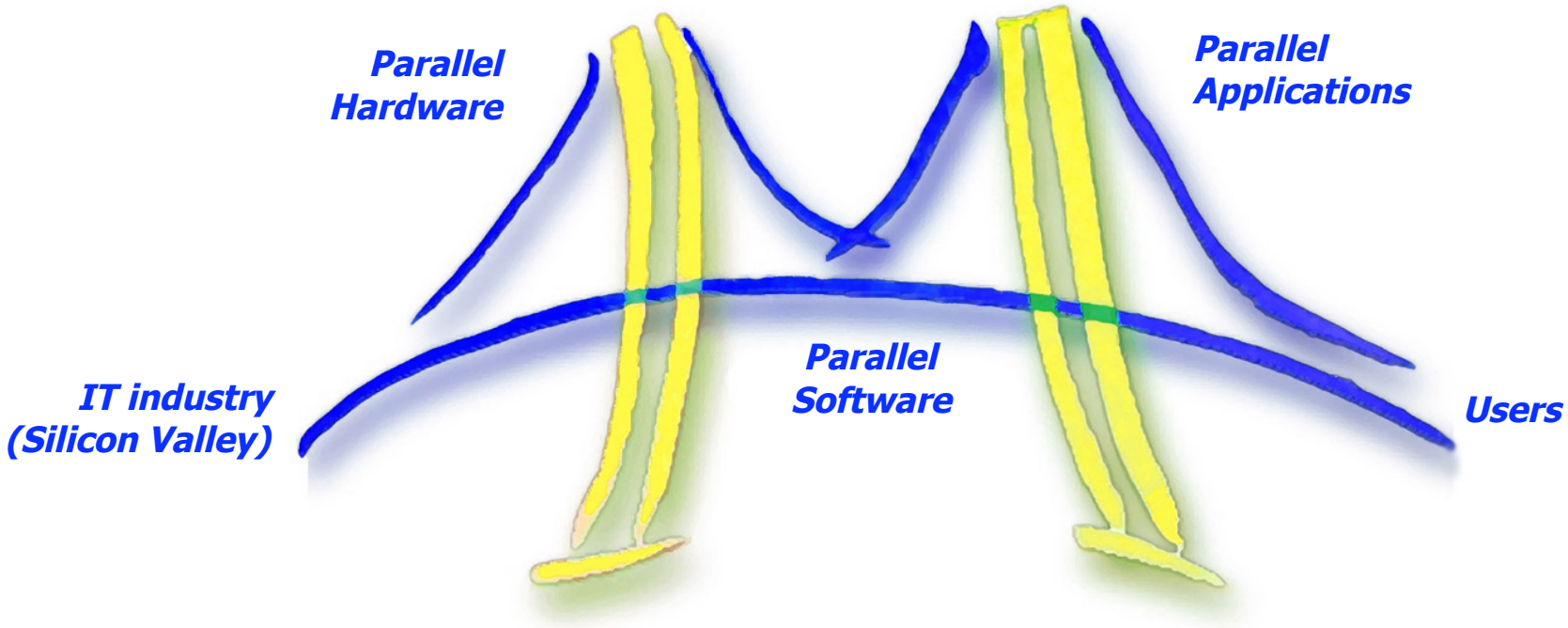


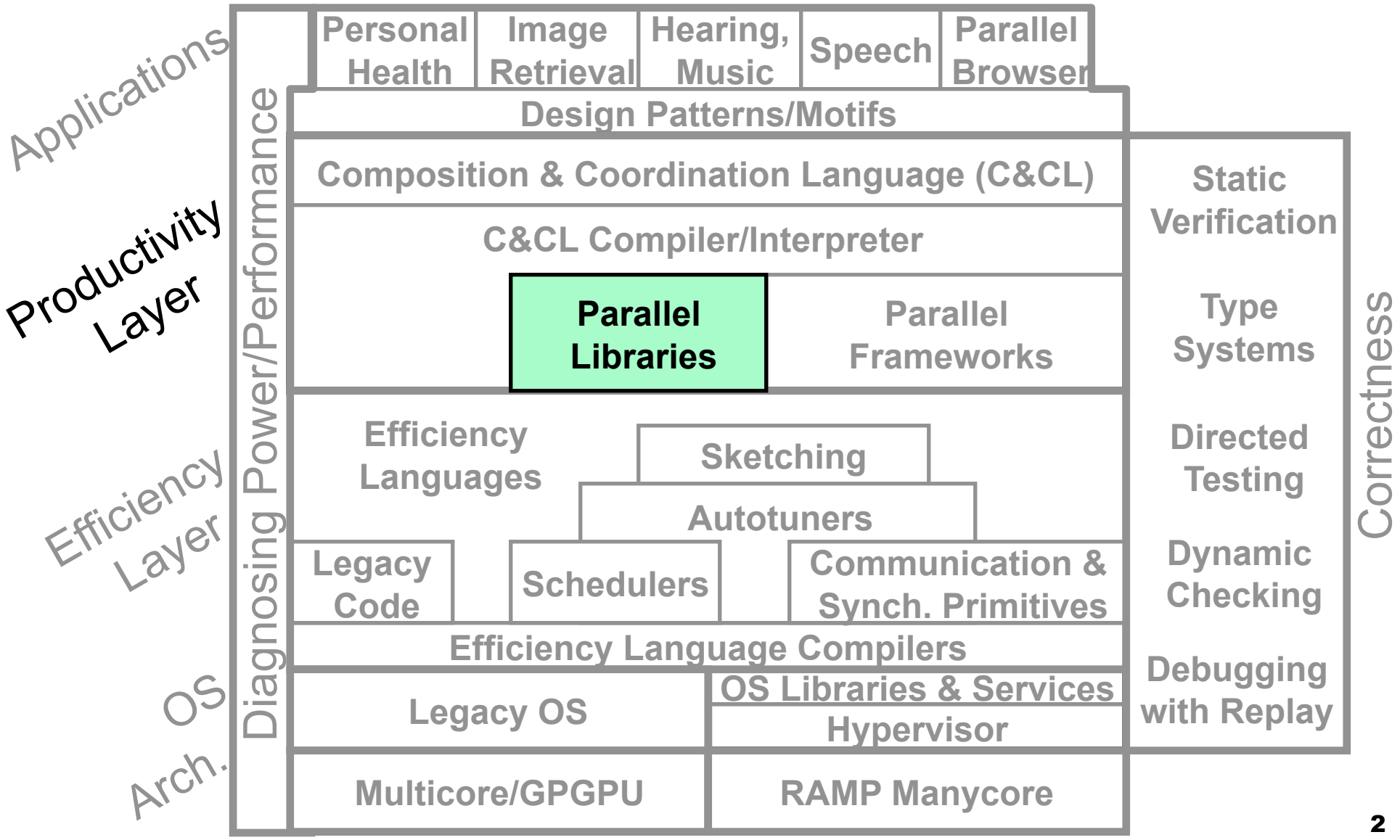
Heterogeneous (CPU+GPU) Performance Libraries

Vasily Volkov

June 6, 2008

Par Lab Research Overview

Easy to write correct programs that run efficiently on manycore



GPU vs CPU

	GeForce 8800GTX	Quad Core 2.4GHz	ratio
Cores	16	4	4
Multiply-add, Gflop/s	346	76.8	4.5
Gflop/s/core	21.6	19.2	
Pin-bandwidth, GB/s	86.4	8.5	10

-Promises 5-10x speedup in high performance codes

-Unless that are hardware constrains that are not in the table

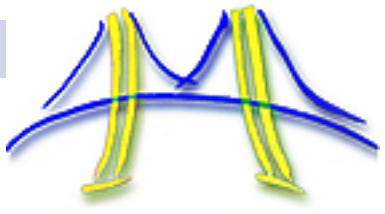
- Higher speedups (50x-100x) are common in applications

- compare serial program running on CPU vs parallel on GPU

- e.g. ignore vector extensions and cores of CPUs

- easier to write thread and data parallel programs on GPU

Challenges in programming GPUs



Not sufficient technical information on GPUs

- how many SIMD lanes register file has? Is memory sequentially consistent? How data is partitioned across memory banks?
- Reverse engineering may fill some of the spaces

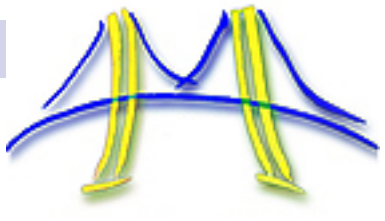
Hardware is (still) exposed via abstract program model

- DirectX in past, CUDA today
- Mixes virtual, abstract concepts and physical resources

Polluted by novel terminology

- What is “warp”, “SIMT”, “branch divergence”?
- Sounds like very new, scares off low-level programmers
- Does not sounds so new, when you figure out how it works

Challenges in programming GPUs



Few people understand how program GPUs for performance

We get 2x speedups vs. best other work in nearly everything we try
- Matrix-matrix multiply (1.6x speedup)
- FFT (2x speedup)
- LU/Cholesky factorization (3x/2x speedups)
- Tridiagonal eigenvalue solver (2x even running on slower GPU)

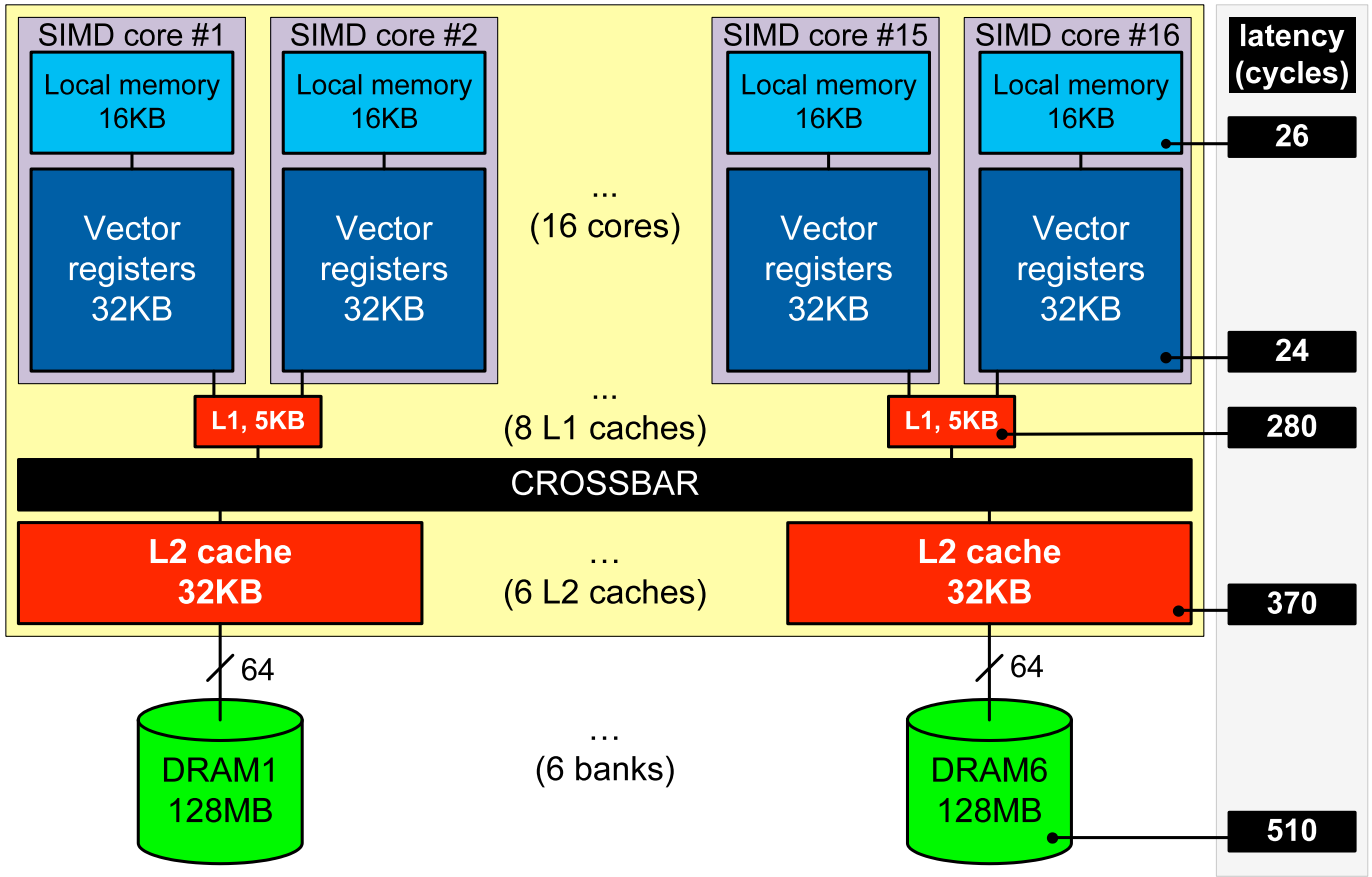
Example: partial pivoting in LU factorization
- Ad: “GPU supports gather/scatter with arbitrary access patterns”
- LAPACK code (sgetrf) spends 50% of time doing pivoting
- Gets worse with larger matrices
- Only 1% of time is spent in pivoting if matrix is transposed
    - 2x speedup!

What is GPU architecture

GPU is an array of vector units
- runs vector threads, 32-elements per vector
- memory accesses in 16-element, aligned, stride-1 vectors
    - can do any access pattern, but 10-100x slower
- 8 lanes for basic arithmetic operations (*, +, bitwise logic, etc)
- 2 lanes for divide, square root, log, etc
- vector registers are largest level of on-chip memory hierarchy
- algorithms used on vector computers are often the best
    - e.g. matrix-matrix multiply, FFT, eigenvalue solver

Particular features
- Multithreading using register windows to hide latencies
- Hardware queue for threads that didn't get window
- Skip branch paths not taken by any threads in predication

8800GTX Memory System



8-word cache line
L1: 8 x 5kB, each is 20-way associative
L2: 6 x 32kB, each is 4-way associative
TLB: 16 entries, fully associative, 512kB pages

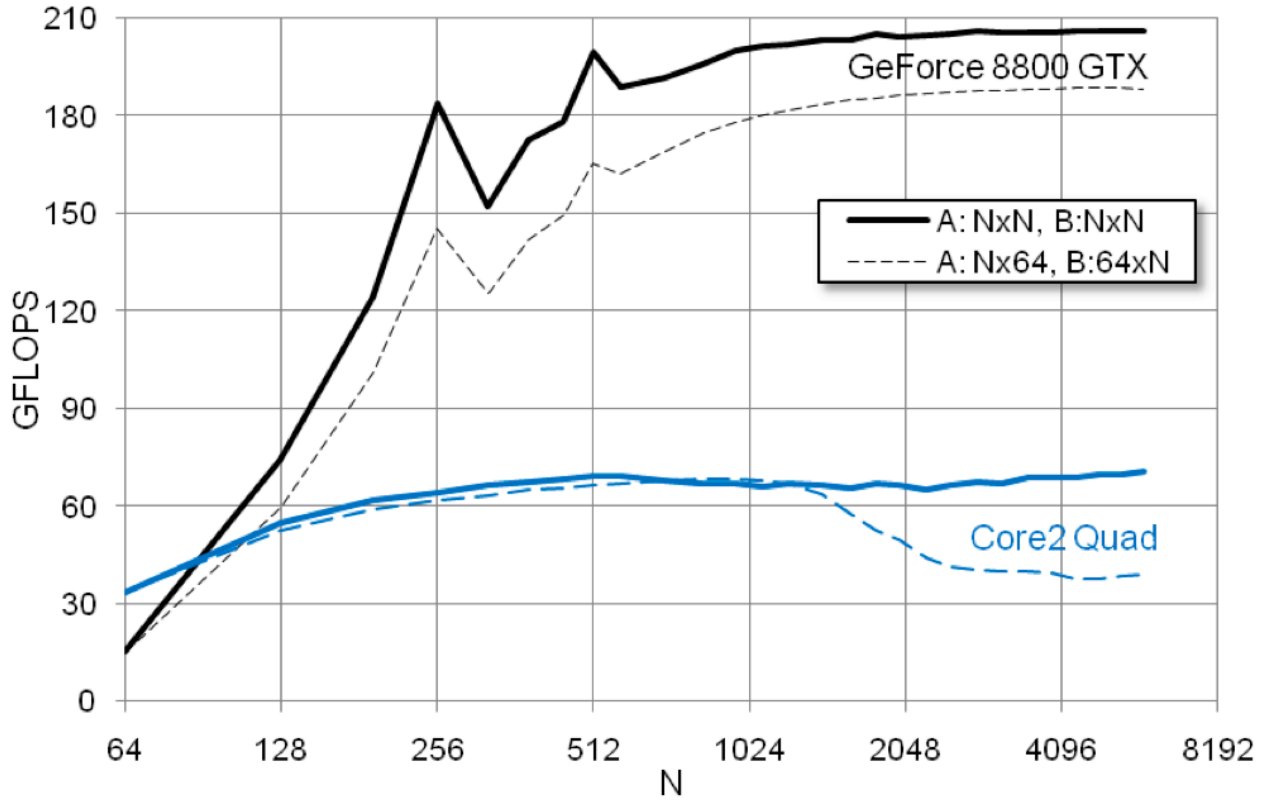
(Part of data is according to pointer chase benchmark)

What is not GPU Architecture

The following features do not belong to GPU architecture

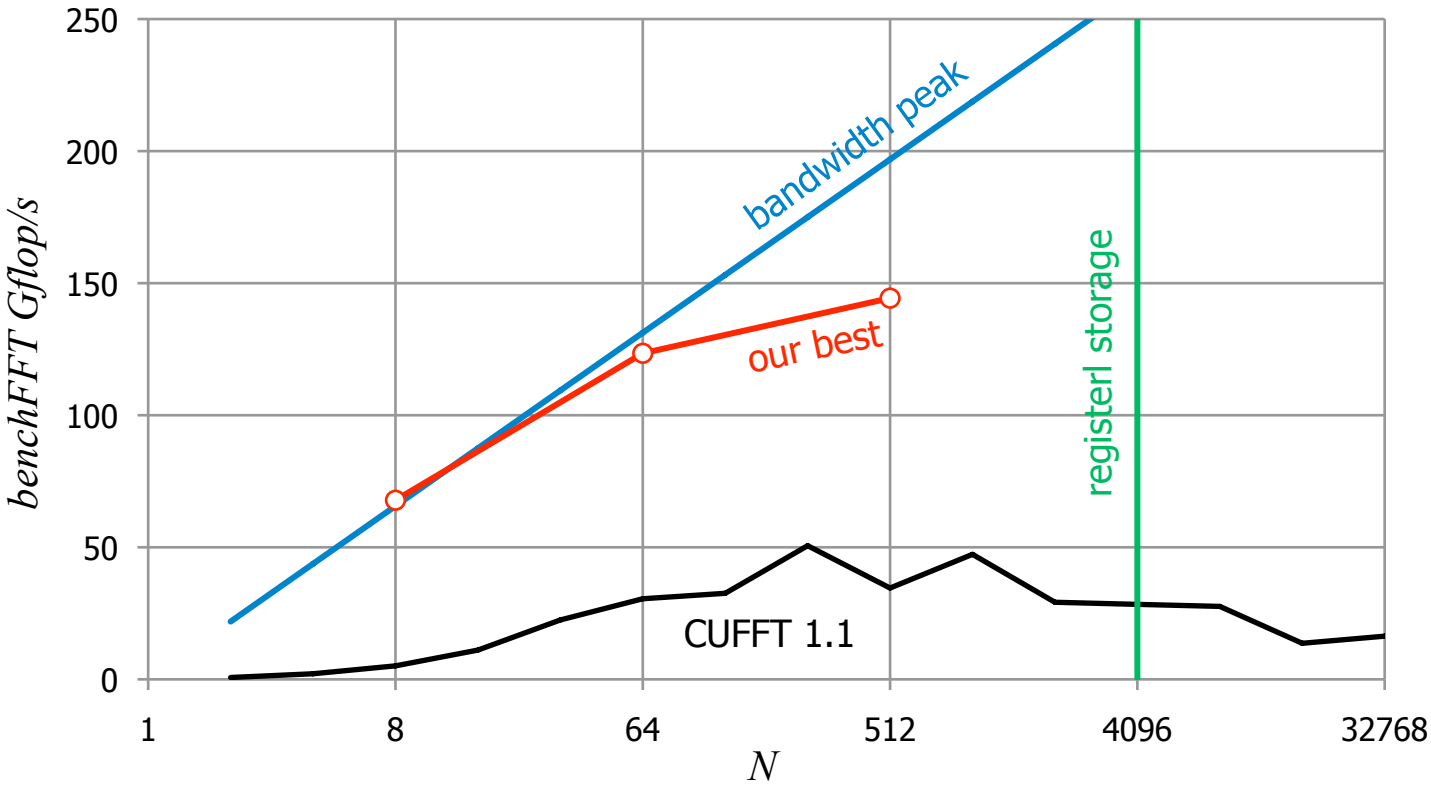
- "GPUs run scalar threads"
    - Scalar thread is an abstraction used in program model
    - There is one physical program counter per 32 scalar threads
    - Vector processors can also be viewed as running scalar threads
- "millions of threads"
    - Not required for GPU and can be useful on other processors
    - It is solution for load balance, scalability and amortization
    - Common in graphics
    - 98% of arithmetic peak with 2 threads per core
        - need > 1 per core due to structure of register file
- "no global communication"
    - may be true if memory is not sequentially consistent
    - (we don't know if it is)
    - otherwise can implement global barrier via DRAM

Matrix-matrix multiply (SGEMM)



- Uses software prefetching, strip-mining, register blocking
- Resembles algorithms used for Cray X1 and IBM 3090 VF
- 60% of peak, bound by access to on-chip shared memory

FFT, complex-to-complex, batched



Results as on GeForce 8800GTX
Resembles algorithms used for vector processors
Radix-8 in registers, transposes using shared memory

Heterogeneous Computing

Q: should you port entire applications to GPU?

- Assume that you have plenty of time to do that
- Should get speedup, right?

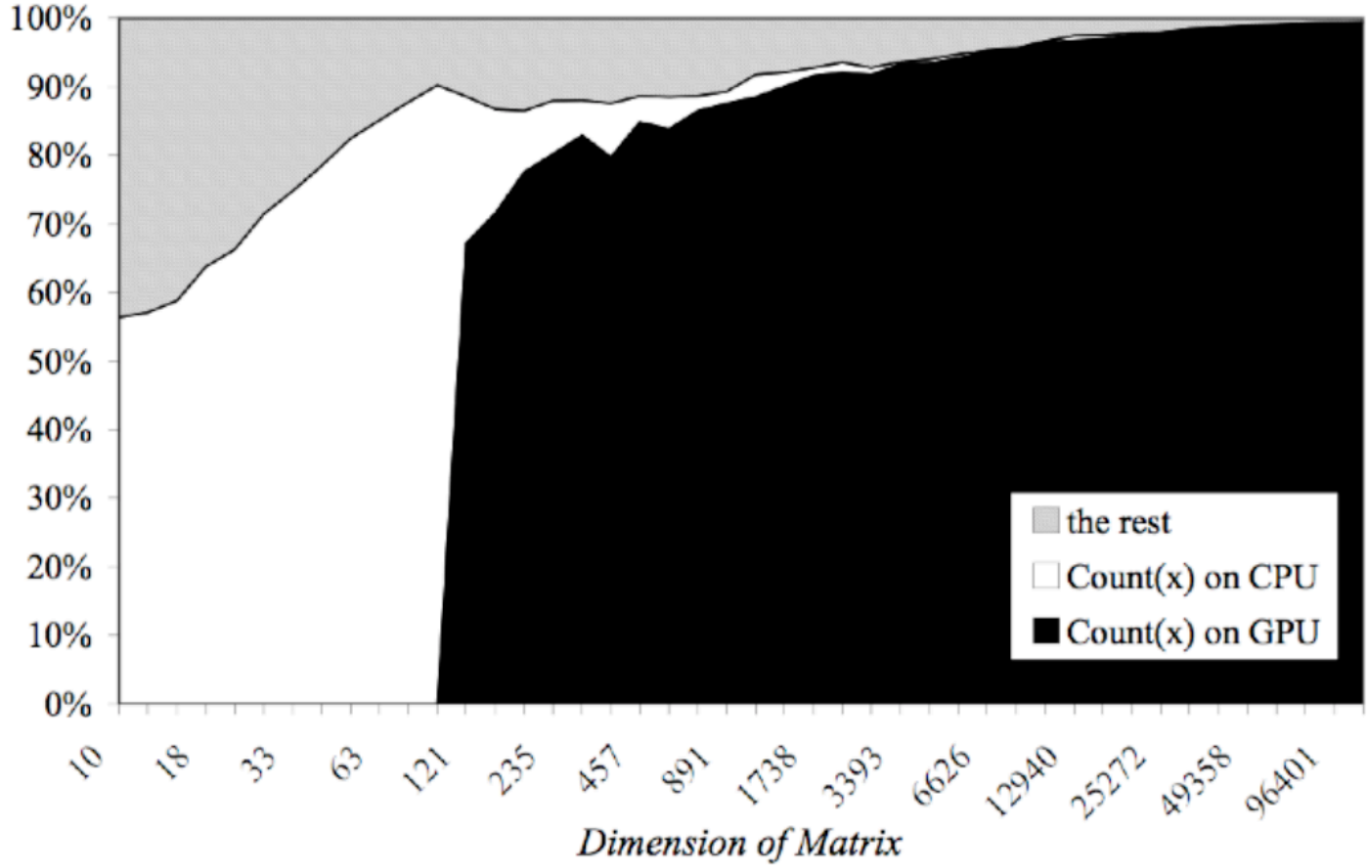
Example: Eigenvalue solver (bisection, LAPACK's sstebz)

- Most work ($O(n^2)$) in vectorized, embarrassingly parallel code
  - do it on GPU and finely tune
  - run on CPU when doesn't worth to use GPU
  - use performance model to choose CPU or GPU version
- Little work ($O(n)$) in nearly serial code
  - do it on CPU: time-consuming and non-trivial to port

Independent project:

- Did everything on the GPU
- Used advanced parallel algorithms for O(n) work
- Up to 2x slower running on faster GPU (compared to us)

Breakdown of Runtime (sstebz)



"Count(x)" is the $O(n^2)$ work, use either SSE or GPU
"the rest" is the $O(n)$ work, doesn't worth to optimize

Fine-grain operations on GPU

CUDA has 5μs startup times
- Dominates at small problems

Example: SAXPY
- Semantics: **y** = α**x** + **y** where **y** and **x** are vectors, α is scalar
- SAXPY byte-to-flop ratio: 6
- GPU (8800GTX) byte-to-flop ratio: 346/75 ≈ 0.22
- Thus, SAXPY is bandwidth bound on GP
- May run at up to r_∞ = 75 GB/s / 6 bytes/flop = 12.5 Gflop/s
- Achieves half of that at $n_{1/2}$ ≈ 30,000
- Runs at 5-10μs at n = 1…30,000
- GPU memory fits only up to 14,000x14,000 matrices
- Runs at nearly constant time in practice

May be even slower if not well-optimized
- isamax in NVIDIA BLAS 1.1 never takes less than 60μs
- faster to copy to CPU and solve there for n < 8,000

LU Factorization

LU factorization:

- $O(n^3)$ work in bulky BLAS3 operations
- $O(n^2)$ work in fine-grain BLAS1 / BLAS2 (panel factorization)

Many found that entire panel factorization is faster on the CPU

- We reported it first, see poster @ Winter 2008 ParLab retreat
- Didn't manage to program GPU well? Any other reason?

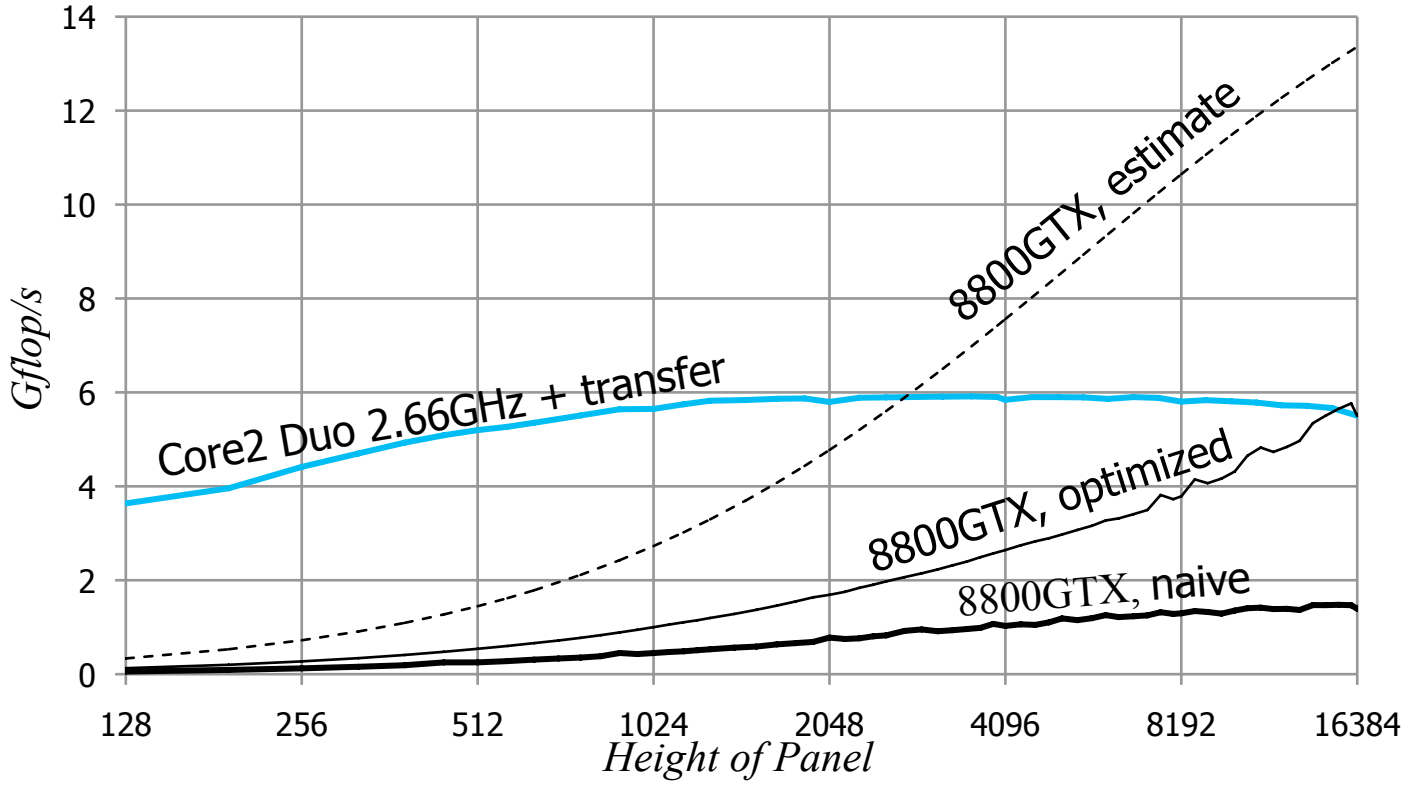
Basic analysis:

- Peak sustained bandwidth of 8800GTX is 75GB/s
- Kernel launch overhead is 5μs
- BLAS1 and BLAS2 are bandwidth bound, so run at

$$Time = 5\mu s + \frac{bandwidth\ \ required}{75GB/s}$$

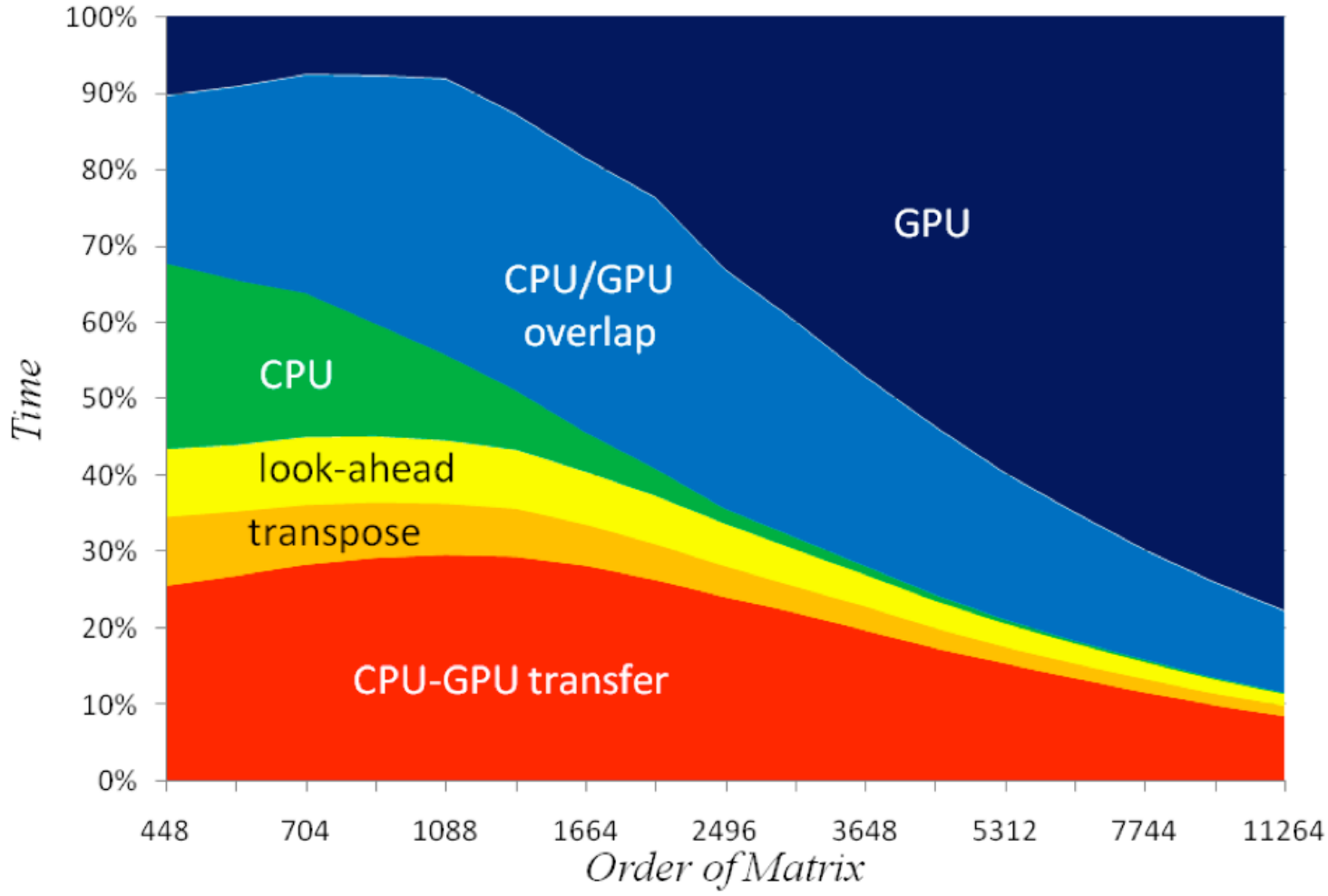
- compare vs. CPU handicapped by CPU-GPU transfers

CPU vs GPU, panel factorization



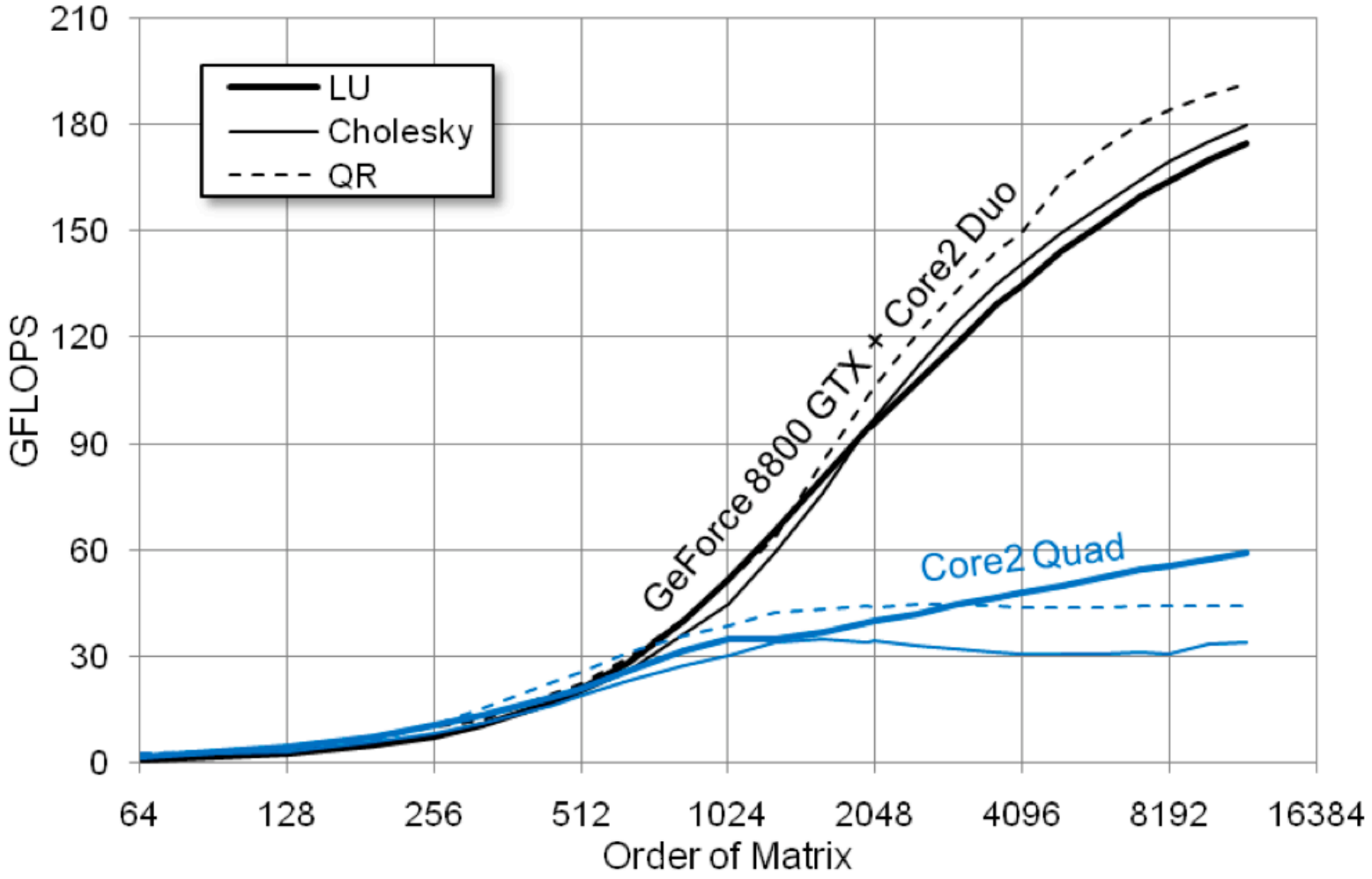
Panel width: 64 (used in practice)
GPU eventually gets faster than CPU
But hard to beat CPU at practical cases (in-core, square matrix)
Instead, we overlap sgemm on GPU w/ panel factorization on CPU

Overlap in LU factorization



Overlap panel factorization on CPU with sgemm on GPU

Overall Performance



Small Triangular Solver

Part of LU solver is solving small (e.g 64x64) triangular system with many right hand sides

- Large grain problem, overhead is not important
- *many* right hand sides, should be enough parallelism
- Up to 13 Gflop/s only if using NVIDIA's BLAS
    - Structure does not fit to GPU constraints?
    - Faster to do on CPU?
    - Faster to multiply by inverse - 80 Gflop/s on GPU
        - matrix-matrix multiply has more parallelism, faster on GPU
        - may get lower numerical accuracy
        - use safe approach if norm of the inverse is too large
            - happens in corner cases, not often

Tridiagonal eigenvector solver

(Inverse Iterations)
Embarrassingly parallel
- Runs *n* independent tridiagonal solves
- (More difficult if eigenvalues are clustered)

LAPACK version (sstein) uses pivoting for better stability
Not using pivoting is faster and easier to vectorize (but may fail)
- run reckless parallel version (e.g. on GPU), no pivoting
- expect that it solves most cases, fails in <1% of cases
- catch those failures, recompute using safe version
    - few of them - recompute on CPU
- 120x faster than LAPACK so far
    - (LAPACK's sstein is not parallel)

Reduction to tridiagonal form

LAPACK's ssytrd
Used in symmetric eigensolvers
LAPACK's version is 50% BLAS3, 50% BLAS1/BLAS2
- BLAS1/BLAS2 ops are fine-grain = slow on GPU
- BLAS3 = 60% of arithmetic peak so far, scales well

Solution:
- Reduce to band matrix on GPU (~100% BLAS3)
- Reduce further on CPU (asymptotically insignificant work)
- Extra overhead if eigenvectors are needed
    - Two orthogonal transforms instead of one

Conclusion

Trend: CPU-GPU communication improves
- Was asymmetric 0.35 / 1.8 GB/s few years ago (X1900XT)
- Was symmetric 3.3GB/s until recent (G80)
- ~5GB/s today (G90?)
- on the same chip in the future?
    - at least in the same memory system?

Trend: many realize importance of heterogeneity
- Program CPU-GPU system, not simply GPU
- Was “port all to GPU, see the speedups” few years ago
- Get speedup by porting to CPU today