Using GPUs for Rapid Electromagnetic Modeling

Peter Messmer*, Travis Austin, John R. Cary, Paul J. Mullowney, Mike Galloy, Keegan Amyx, Nate Sizemore, Brian Granger, Dan Karipides, David Fillmore

*messmer@txcorp.com

Tech-X Corporation
5621 Arapahoe Ave., Boulder, CO 80303
www.txcorp.com

This work is partially supported by NASA SBIR Phase-II Grant #NNG06CA13C

SciDAC ComPASS All Hands Meeting, UCLA, December 3, 2008

Motivation

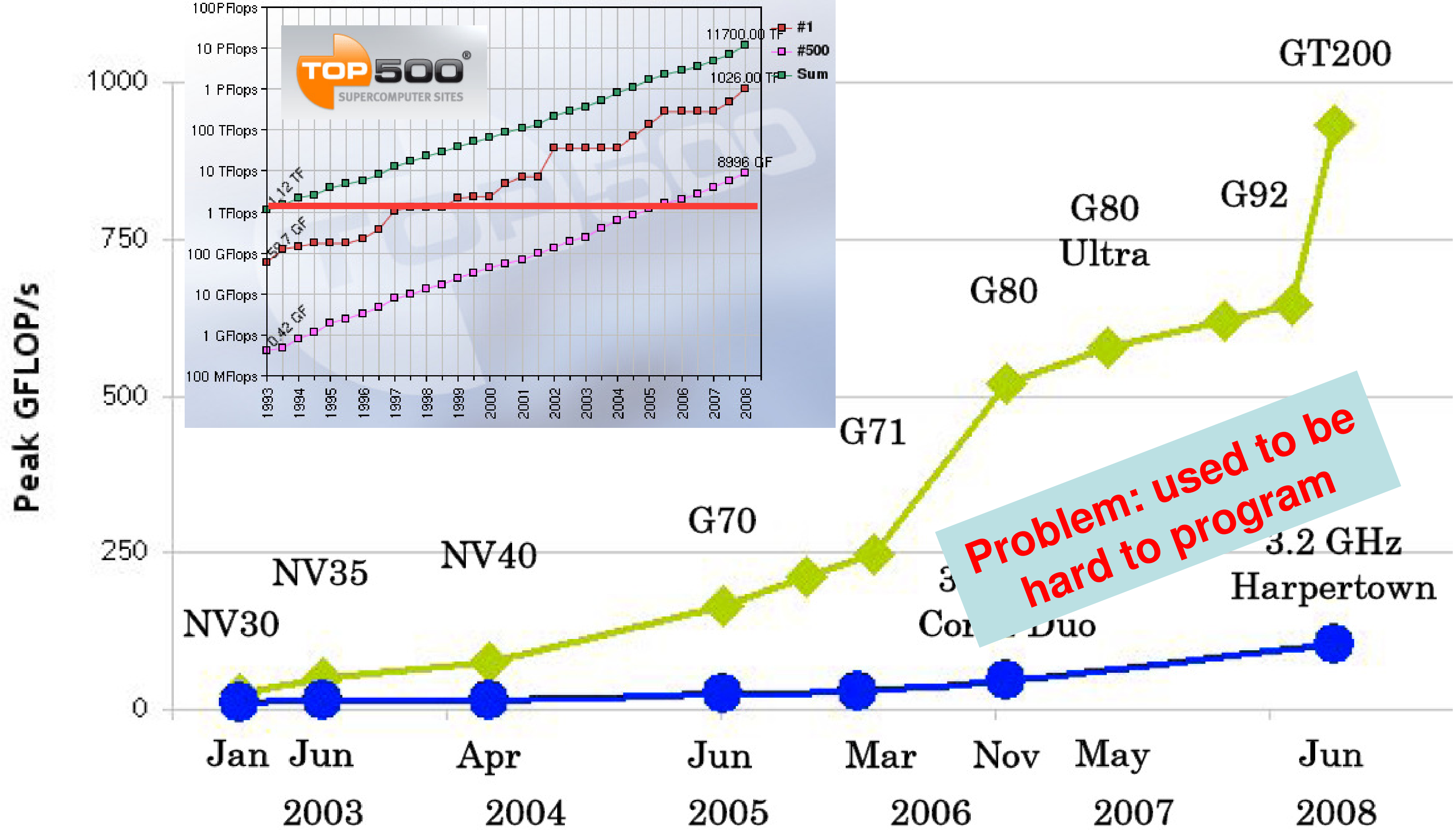
- **Need fast turnaround for FDTD simulations**
  - E.g. Frequency extraction (see Travis' talk), cavity optimizations
- **Parallelization of FDTD has limits**
  - Some problems too small: $N > (\tau_{latency} / \tau_{cell})/N + \tau_{comm} / \tau_{cell}$
  - "Time does not parallelize"
  - Access to large systems can be painful
- **FDTD highly memory bandwidth limited**
  - Almost no data reuse -> caches useless
  - Multi-core CPU makes it even worse

$\Rightarrow$ **Need high memory bandwidth accelerator**

Outline

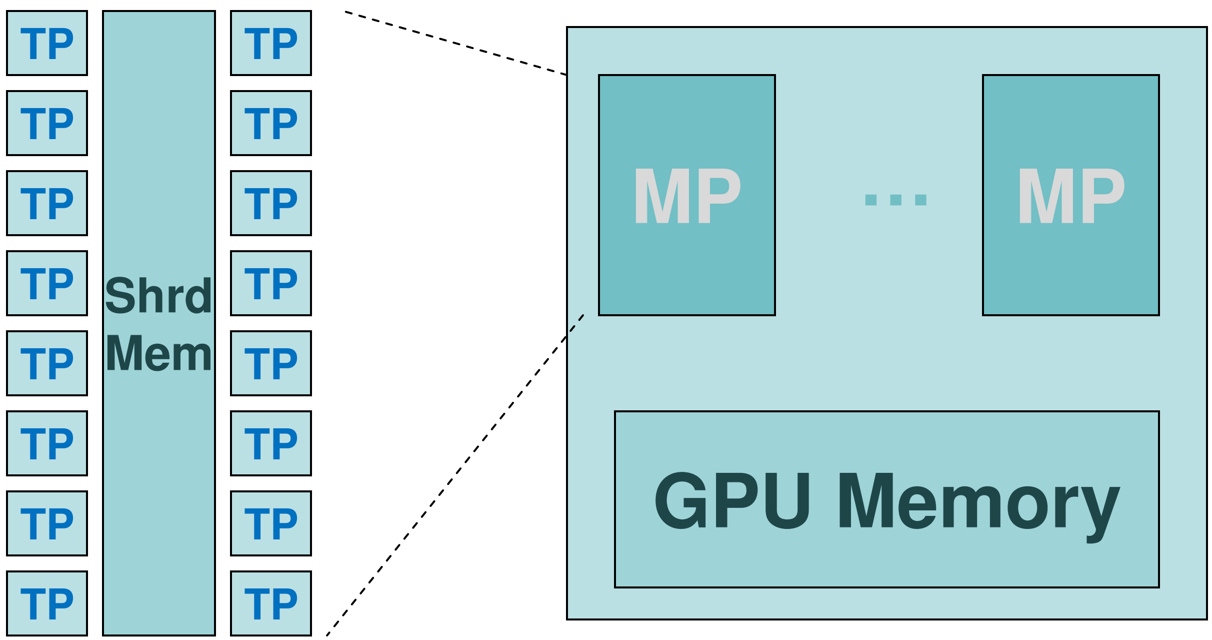
- **GPU architecture, programming**
- **GPULib: Simplification of GPU development**
- **Implementation of FDTD on GPUs**
- **Conclusion**

Why scientific computing on GPUs?

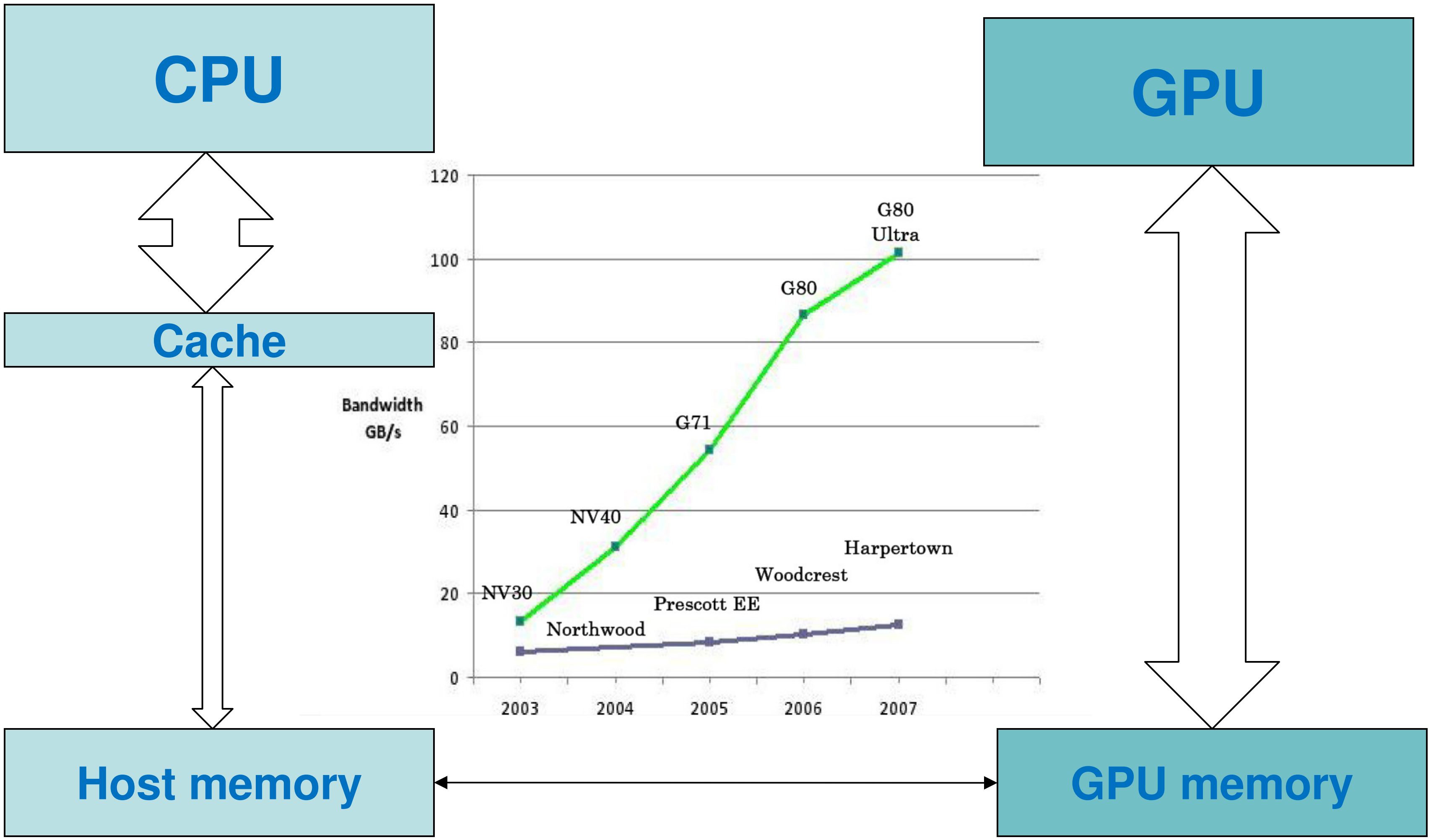


GPUs are Massively Parallel Floating-Point Co-Processors

- **Silicon used for ALUs, rather than large caches**
  - Up to 240 (!) processing elements (“thread processors”, TP)
  - running at 1.3 GHz, statically scheduled, 2 instructions / cycle
  - Small software managed caches (“shared memory”, Shrd Mem)
- **Organized as ‘Multi-processors’ (~ SIMD processors)**
  - Software managed caches shared within one multi-processor
  - Synchronization within MP, no light-weight global synchronization
- **Active thread management**
  - Work on next thread-set while waiting for a memory request



Another Advantage of GPUs: High Memory Bandwidth



The Flipside: GPUs like (=need!) regular “patterns”

- **Collection of SIMD processors**
  - Thread divergence handled by masked execution
    - E.g. two-way conditional takes sum of both branches
- **Needs large number of threads**
  - Keep all TP busy
  - Hide memory access latency with work
- **TPs need to access successive memory locations**
  - Results in a single memory request
  - “Memory coalescence”
- **Double precision FP currently slow**

$\Rightarrow$ **Want large number of (almost) identical floating point operations on contiguous block of memory**

$\Rightarrow$ **Redundant computation is ok, if it optimizes memory access**

$\Rightarrow$ **Avoid CPU/GPU transfers**

CUDA: Code development environment for (NVIDIA) GPUs

- **Early GPGPU efforts heroic**
  - Graphics API (OpenGL, DirectX) no natural fit for scientific computing
- **Compute Unified Device Architecture (http://www.nvidia.com/cuda)**
  - Supported on all modern NVIDIA GPUs (notebook GPUs, high-end GPUs, mobile devices)
  - Future: Co-Existence with OpenCL
- **Single Source for CPU and GPU**
  - Host code C or C++
  - GPU code C(++) with extensions
    - “Kernel” describes on thread
    - Host invokes a collection of threads
  - nvcc: NVIDIA cuda compiler
- **Runtime libraries**
  - Data transfer, kernel launch, ..
  - BLAS, FFT libraries
- **Simplified GPU development, but still “close to the metal”!**

GPULib: One way to simplify GPU development

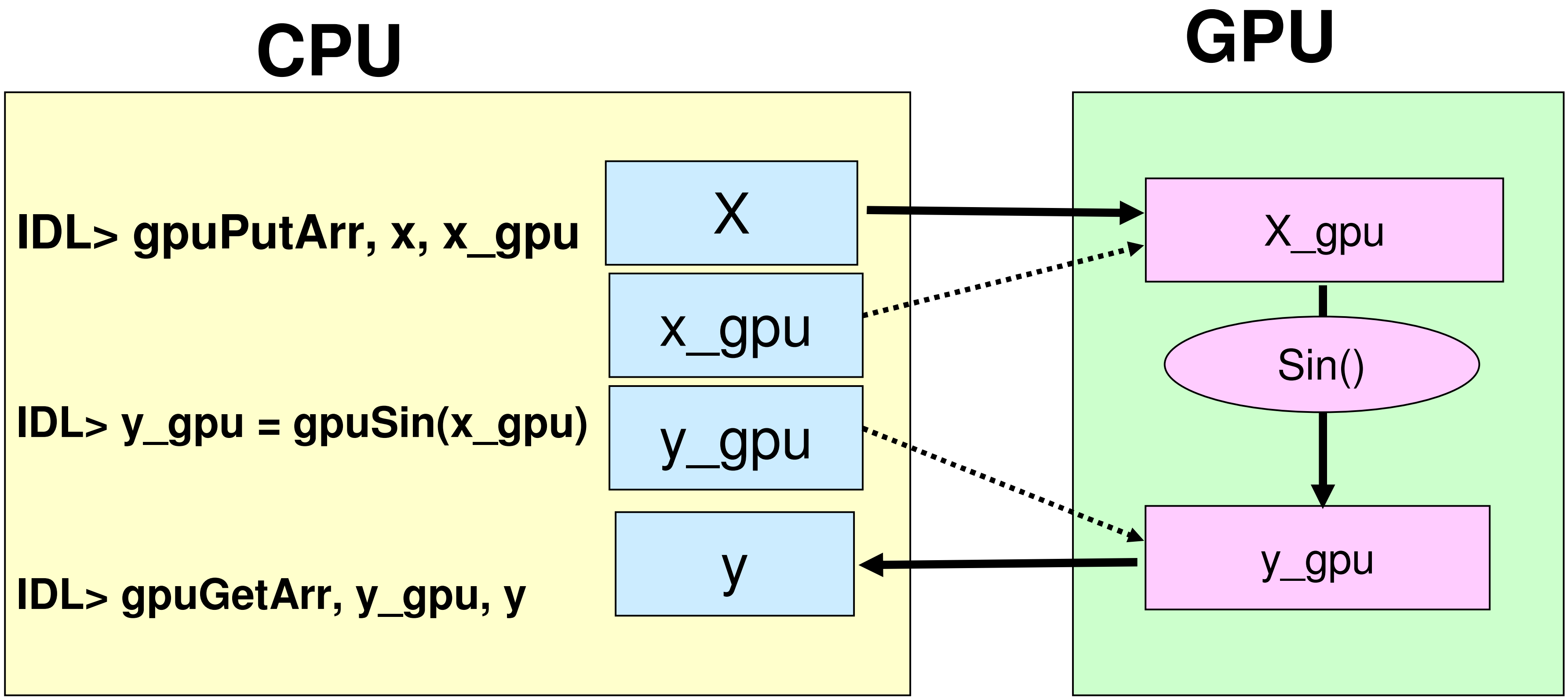
- **Provide access to GPUs in Very High-Level Languages**
  - IDL, MATLAB, (Python)
  - Seamless integration into host language
- **Data objects on GPU represented as structure/object on CPU**
  - Contains size information, dimensionality and pointer to GPU memory
- **GPULib provides a large set of vector operations**
  - Data transfer GPU/CPU, memory management
  - Arithmetic, transcendental, logical functions
  - **Support for different types (float, double, complex, dcomplex)**
  - Data parallel primitives, reduction, masking (**total, where**)
  - Array operations (reshaping, interpolation, range selection, **type casting**)
  - NVIDIA's cuBLAS, **cuFFT**

  => Reduces need for CPU/GPU transfers
- **Download from http://gpulib.txcorp.com**

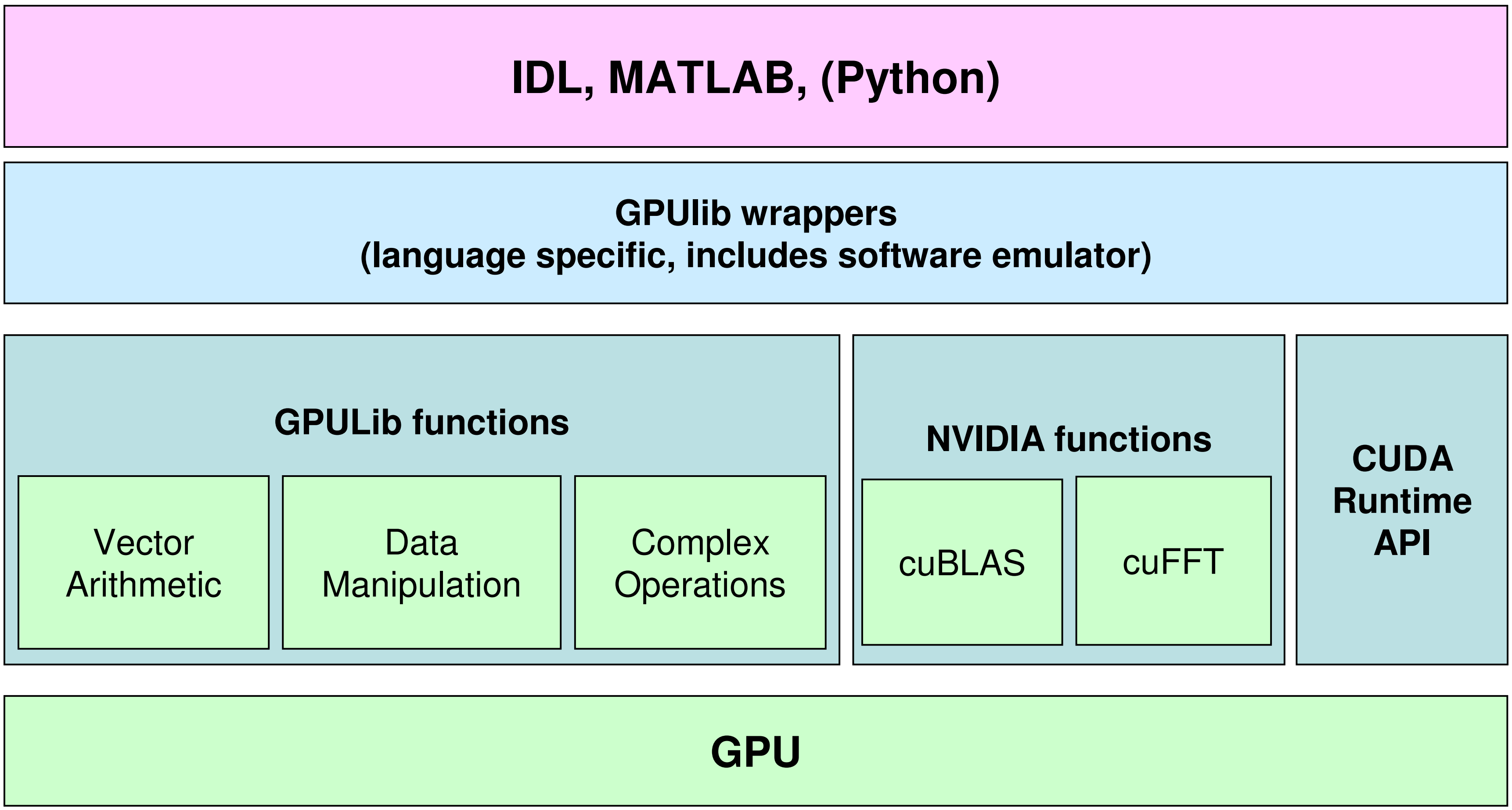
  (free for non-commercial use)

Messmer, Mullowney, Granger, *"GPULib: GPU computing in High-Level Languages",* Computers in Science and Engineering, 10(5), 80, 2008.

A GPULib example in IDL

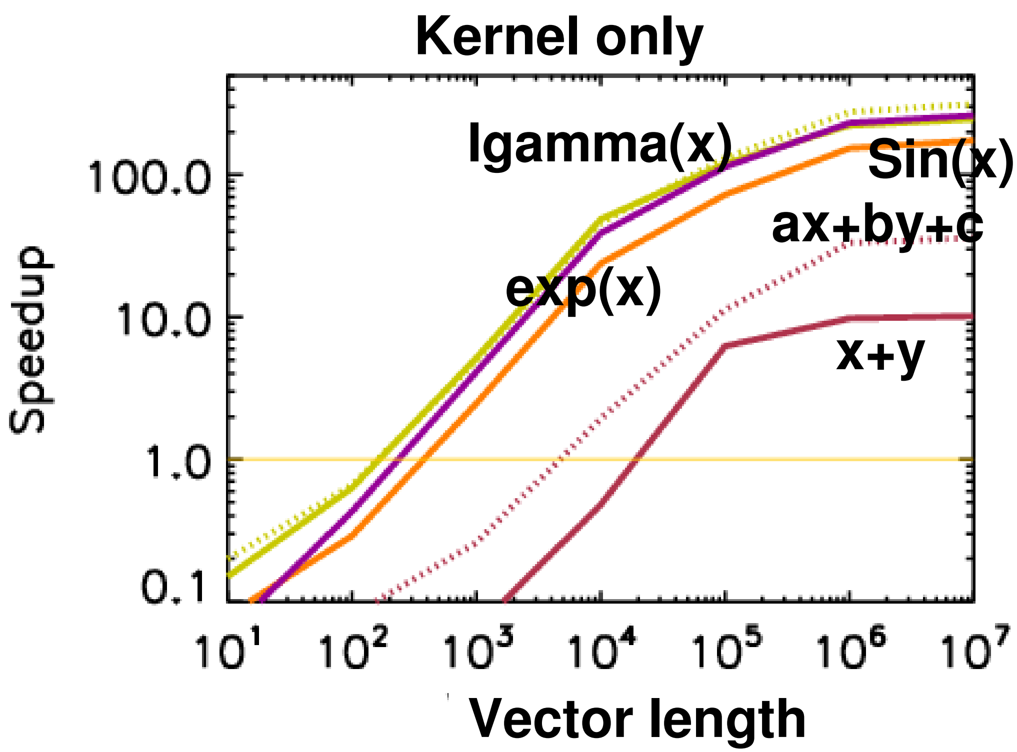


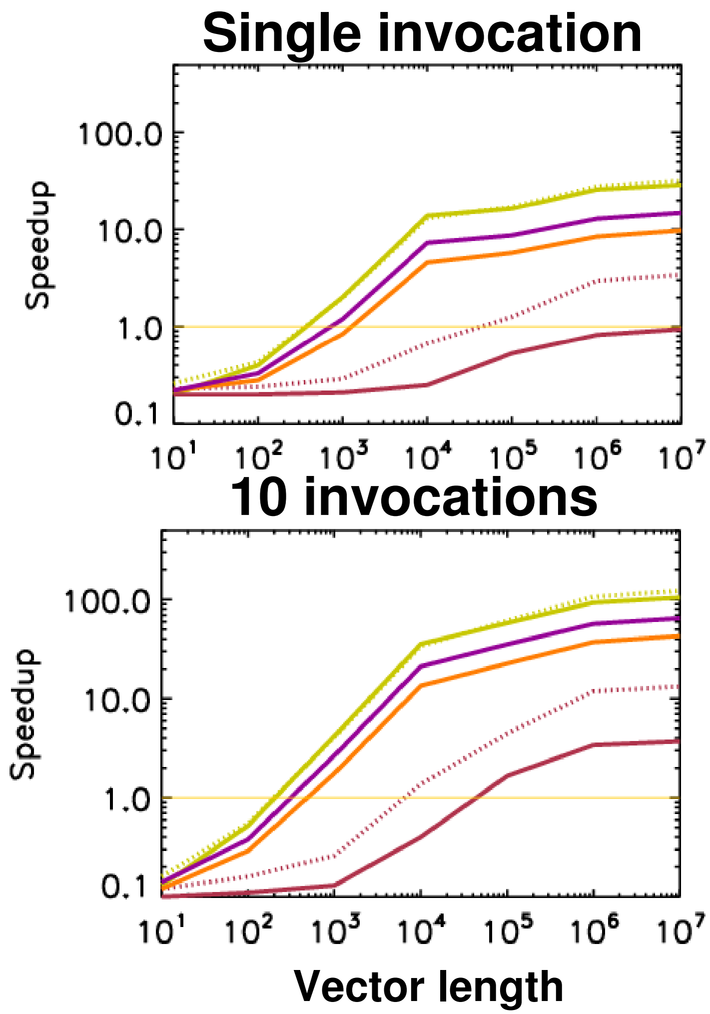
GPULib layered architecture is easily extensible



How to get performance?

- **Kernels are very fast, GPU<->CPU data transfer is slow**





Example: Image Deconvolution

- Image is convolved with detector point-spread function:

$$I_{obs}(x, y) = \int I_{true}(x-u, y-v)P(u,v)dudv$$

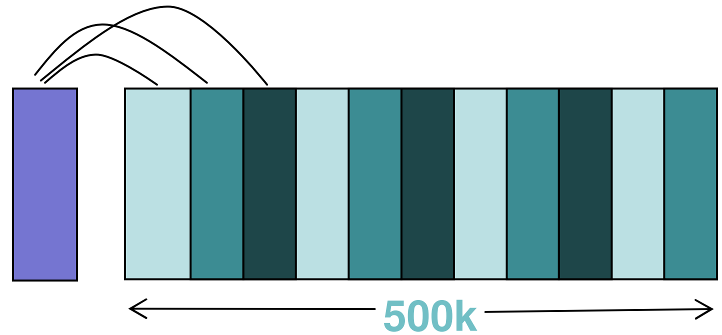
- Clean image by (complex) division in Fourier space:

$$I_{true}(x, y) = FFT^{-1}(FFT(I_{obs}) / FFT(P))$$

- Large computational load per CPU-GPU data transfer
- Speedup ranging from 5x – 28x for 256x256 – 3kx3k images

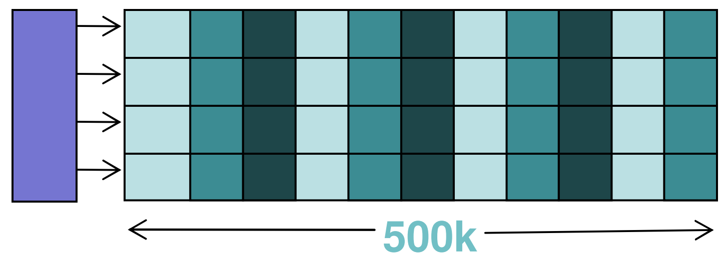
Example: Database search

- **Find closest match in 500k words with 128 characters each**
- **Less than 10ms**
- **CPU: ~200 ms**



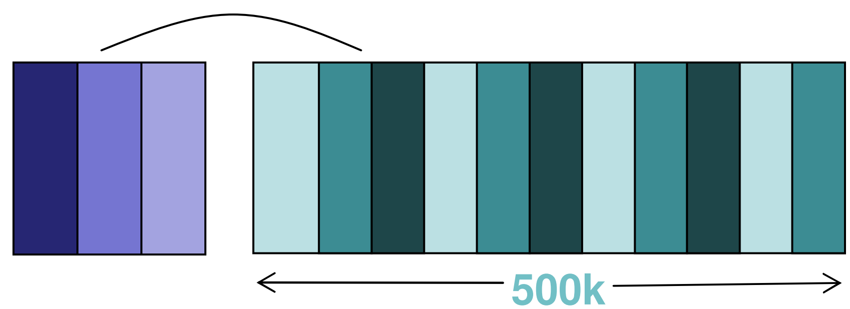
- **GPULib 1: 500k dot-products**
  - Need test vector on GPU
  - Vectors short
  - Huge number of kernel invocations

  => Bad idea



- **GPULib 2: 128 accumulations**
  - No need to transfer entire vector
  - Large vectors
  - Smaller number of kernel invocations

  => ~27 ms

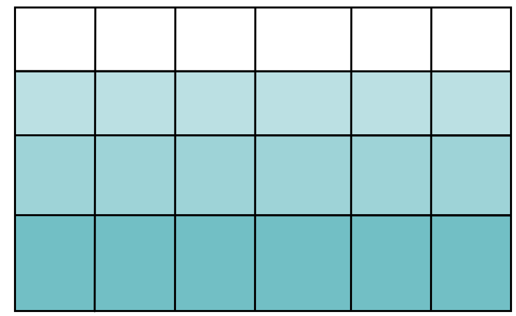


- **Hand crafted implementation**
  - Transfer data to GPU
  - Perform 128 dot products concurrently

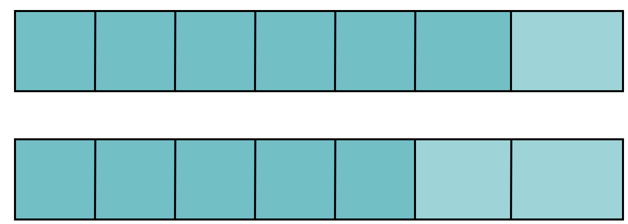
  => < 8 ms (old GeForce 8800 GTX)

FDTD fits well on GPUs

- **Data remains on GPU**
  - Memory large enough for interesting problems
  - For distributed memory use 1D/2D/3D memcpy



- **Avoid operations on short vectors**
  - Stencil picture may be misleading



- **Treat 3D domain as large 1D vectors**
  - Shifted vector operations ‘cheap’
    - Pointer arithmetic possible on GPUs
    - Regular operation on large vector -> ideal for GPU
  - ‘Dirt’ at domain boundaries due to wrap-around
    - Removed by applying boundary conditions

(Canadian Company Acceleware sells GPU-based FDTD accelerators: www.acceleware.com)

GPULib enabled rapid development of FDTD on GPUS

- **3D FDTD**
  - Cut-Cell (Dey-Mittra) and Stair-Stepped boundaries
- **Reads VORPAL geometry output**
  - Simulations should result in
- **Entire computation on rectangular domain**
  - Compute update outside of conformal boundaries for simplified memory access
- **Entirely GPULib based**
  - Written in IDL -> integrated visualization, visual debugging
  - Quickly demonstrate potential of GPU based FDTD
  - Parallelization using mpIDL (http://www.txcorp.com/products/FastDL)
- **Custom Kernel**
  - Optimize for performance, reduce memory transfer

Preliminary performance results highly promising

- **Performance (preliminary)**
  - Up to 470 Mcells/s on GPU including cut-cells boundaries
    - Currently at ~70% theoretical memory bandwidth, so still potential
  - ~10 Mcells/s on CPU

⇒ **~ 40-50x speedup compared to CPU based implementation**
  - Comparable to ~48 Franklin cores

- **Question: How bad is effect of single precision FP?**
  - Needs detailed evaluation
  - Think about your units!
- **Question: What about large problems?**
  - Currently no huge GPU systems available, may change
  - 2.6x speedup on a 3GPU ‘cluster’ (PSC)

Summary/Conclusions

- **GPUs offer large for accelerating scientific applications**
- **CUDA significantly simplifies code development**
  - Still requires understanding of hardware
- **GPULib enables GPU development from within VHLLs**
  - Provides large set of vector operations with unified interface
  - Enables rapid development of GPU accelerated algorithms
  - No hardware knowledge required
- **FDTD solver on GPU**
  - Loosely coupled to VORPAL (tighter integration planned)
  - Both stair-stepped and cut-cell boundaries
- **GPUs yields ~40x speedup compared to CPU**
  - Problems that take O(minutes) become O(seconds)
  - Compute on your desktop, rather than at HPC center