



Energy Industry Applications:

~Three examples of GPU Acceleration in the
Oil and Gas Industry using CUDA

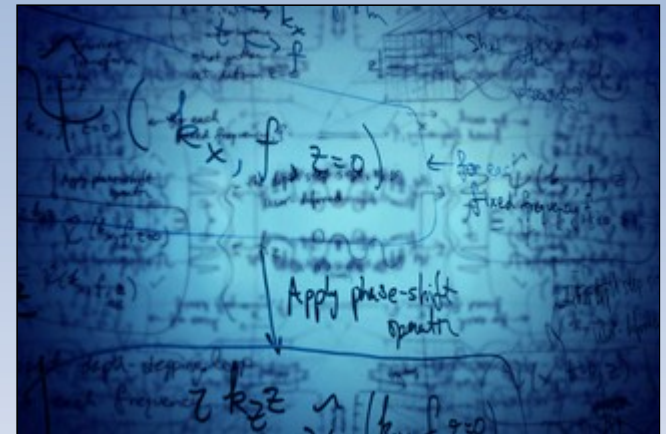
William Brouwer

Overview

- Company Background
- CUDA accelerates Geophysics:
 - Data Processing w/ Linear Algebra
 - Sparse Matrices
 - Seismic Imaging: Kirchhoff Time Migration
 - Frequency Filtering
 - Traveltime/Weight Calculations
 - GPU Migration
- Summary

Stone Ridge Technology

- Based in Maryland, USA
- Physics, CS, EE
- Services
 - Multi-core
 - GPU
 - FPGA
- Application areas
 - Energy (e.g. Seismic, Res. Sim.)
 - BioInformatics
 - Signal processing
 - FDTD E&M



Nvidia GPU + CUDA → Accelerated Geophysics

- The three key abstractions of CUDA are ideal for data processing and simulation: hierarchy of thread groups, shared memory, barrier synchronization
- Threads organized into blocks, within grid, can be identified via 1/2/3D index in kernel function, using threadIdx variable.
- Thread blocks execute independently; any order, parallel or series.
- Latency drastically reduced via reuse of *shared memory per thread block* and by avoiding host ↔ device exchanges

GPU Programming Model

C program
sequential
*.exe

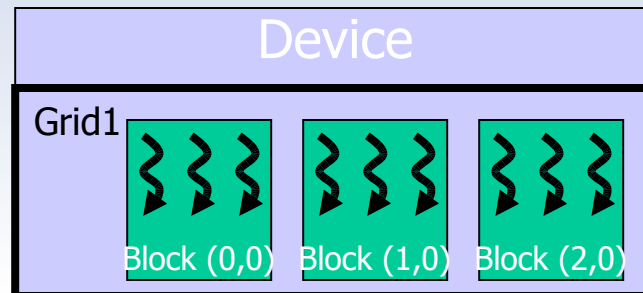
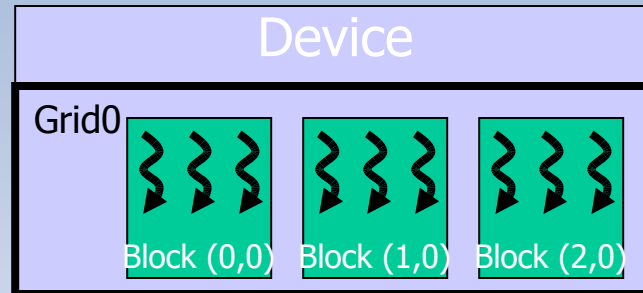


serial code

parallel
kernel00
<<<..>>>

serial code

parallel
kernel01
<<<..>>>



Geophysics Data Processing

- Many Science/engineering calculations are really just linear algebra calculations
- Seismic imaging (for instance) is an inverse calculation
- The following are more or less operators and adjoints in geophysics:

Operator	Adjoint
scatter/spray	sum/stack
Hyperbolic modeling	NMO & CDP stack
Derivative (slope)	-Derivative

Data Processing (Linear Algebra) Examples

- Time derivative estimated by convolution with filter:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} -1 & 1 & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & -1 & 1 \\ \cdot & \cdot & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_5 \\ x_6 \end{bmatrix}$$

- Interpolation eg., $d(ata) \rightarrow m(odel)$ or $d \leftarrow m$

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \dots \\ \cdot & \dots & \cdot \\ \dots & w_{34} & w_{35} \end{bmatrix} \begin{bmatrix} m_0 \\ \vdots \\ m_5 \end{bmatrix}$$

- In any case, we can easily assign sub-matrices to separate blocks on CUDA grid using *thread/block indices*

Matrix Multiply in CUDA I

```
// Block index
int bx = blockIdx.x;
int by = blockIdx.y;

// Thread index
int tx = threadIdx.x;
int ty = threadIdx.y;

// First/last sub-matrix index of A processed by block in C=A*B & step
int aBegin = wA * BLOCK_SIZE * by;
int aEnd   = aBegin + wA - 1;
int aStep  = BLOCK_SIZE;

// First/last sub-matrix index of B processed by block
int bBegin = BLOCK_SIZE * bx;
int bStep  = BLOCK_SIZE * wB;

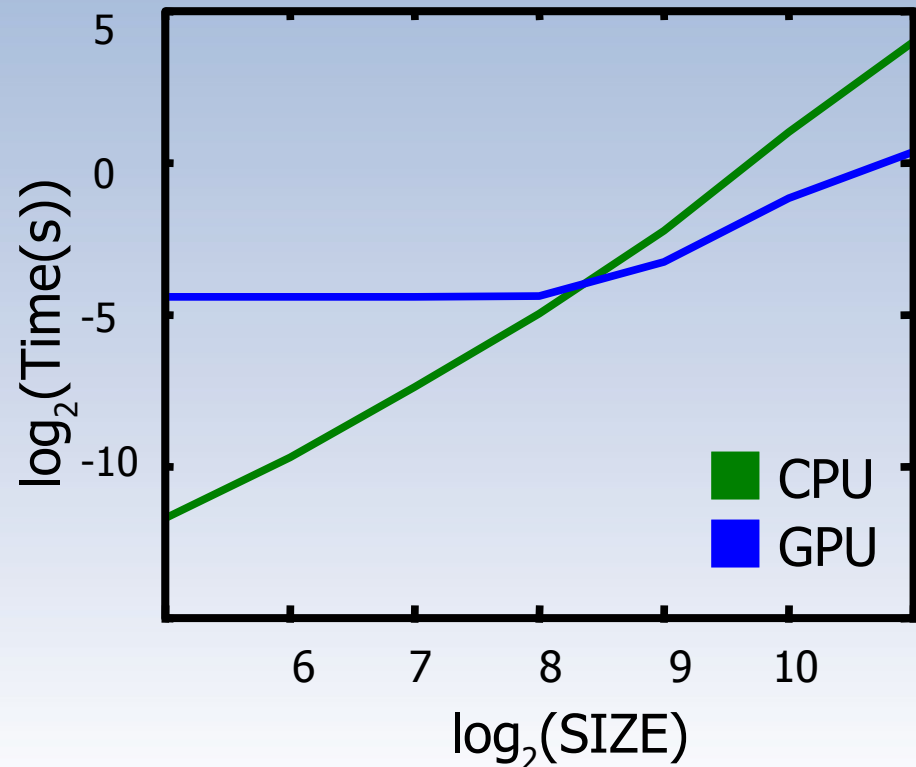
//storage container for threads computing sub-matrix of C
float Csub = 0;
```

Matrix Multiply in CUDA II

```
for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep){
    // Decln of the shared memory array for sub-matrices in C= A*B
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Load the matrices from device memory, one element / thread
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];
    //Barrier
    __syncthreads();
    //Multiply; each thread one element of C sub-matrix
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);
    // Barrier to
    __syncthreads();
} //write to global
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

Performance I

- After $N \sim 2^7$ GPU time benefits significantly from shared memory
- For $N \sim 2^{11}$ CPU is 16 x GPU time, and increasing at \gg higher rate
- There exists CUBLAS optimized libraries for even better performance



Sparse Matrix-Vector

- Many problems in classical physics and geophysics give rise to sparse matrices eg., numerical solutions to PDE via conjugate-gradient approach
- Various data representations exist eg., Coordinate (COO) and Compressed Sparse Row (CSR):

$$\begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 3 & 0 & 1 \end{bmatrix}$$



$$\begin{aligned} \text{ptr} &= [0235] \\ \text{col} &= [02102] \\ \text{data} &= [12131] \end{aligned}$$

- (Almost) regardless of format, sp matrix-vector multiply is increasingly expensive in CPU → *use GPU with data organized such that contiguous reads are possible*

SpMV in CUDA

```
__global__ void SparseMV_CSRKernel(const int numRows, const int *ptr, const int
    *indices, const float *data, const float *x, float *y){

    __shared__ float values[BLOCK];

    //thread, warp, thread/warp labels
    int thrLabel = blockDim.x * blockIdx.x + threadIdx.x;
    int warLabel = thrLabel / 32;
    int lane = thrLabel & (32-1);
    int row = warLabel;

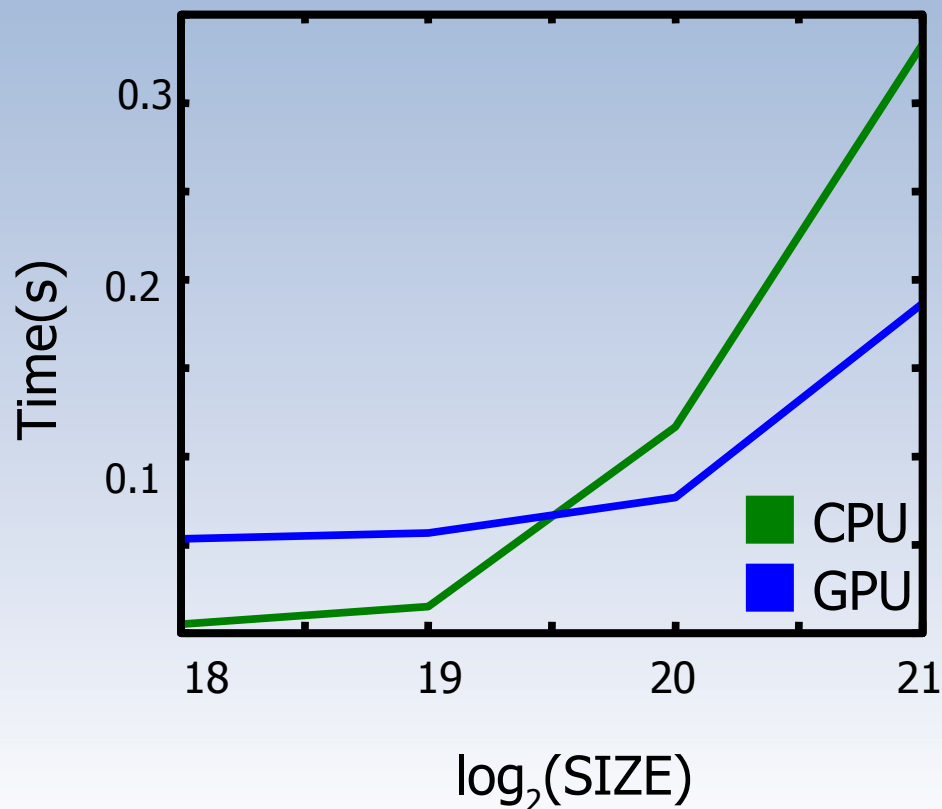
    if (row < numRows){
        int rowStart = ptr[row]; int rowEnd = ptr[row+1];

        //running sum/thread
        values[threadIdx.x]=0;
        for(int jj = rowStart + lane; jj< rowEnd; jj+=32){
            values[threadIdx.x] += data[jj]*x[indices[jj]];

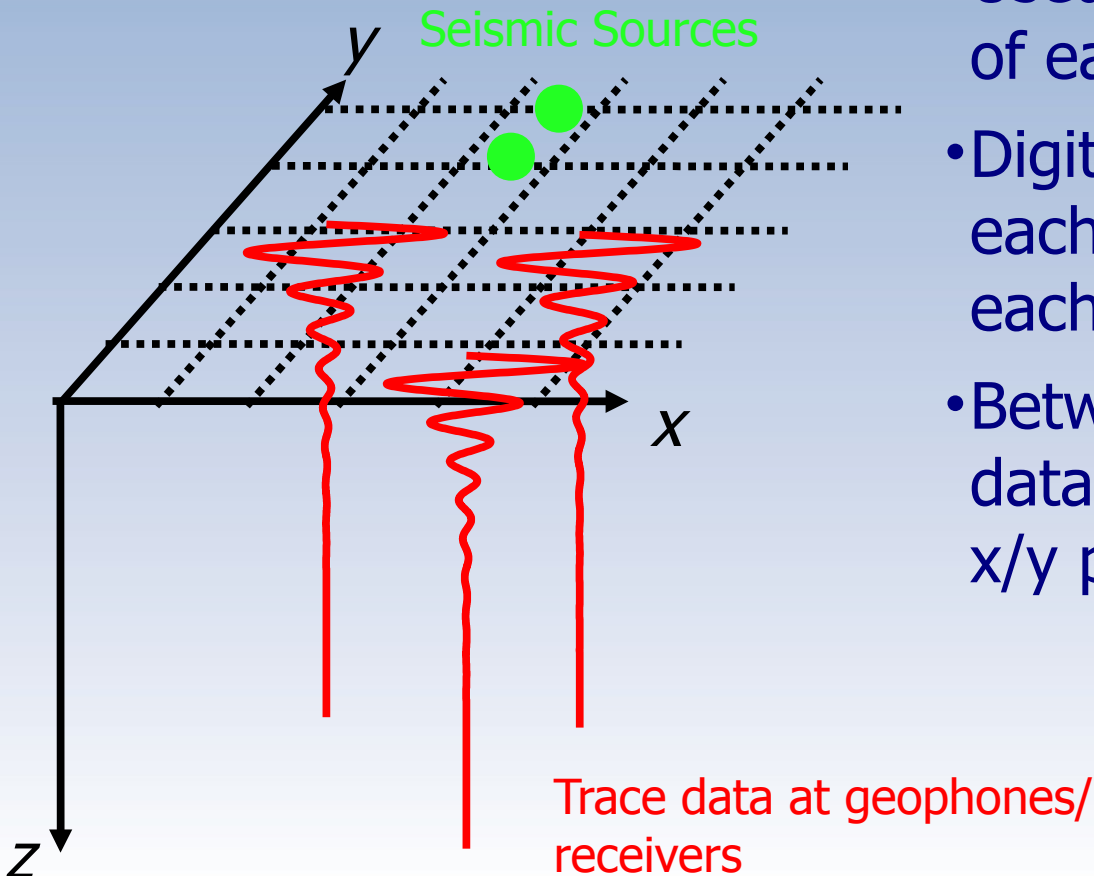
            //reduction in shared
            if (lane < 16) values[threadIdx.x] += values[threadIdx.x + 16];
            ...
            if (lane < 1) values[threadIdx.x] += values[threadIdx.x + 1];
        }
        //write results
        if (lane==0)
            y[row] += values[threadIdx.x]; }}
```

Performance II

- This vector kernel accesses indices and data contiguously
- Memory access takes place according to *warp*, whose length should be $<$ maximum entries per row
- Not optimized in this example, nonetheless significant benefits where total non-zero entries $>$ 1M



Seismic Imaging



- Used to create 2D/3D images of earth's interior
- Digitally sampled data trace for each src/rec pair, $\sim 1\text{k}$ points each, $\sim 4\text{ms}$ rate
- Between 10^4 - 10^6 traces per dataset, each acquired over 2D x/y plane

Kirchhoff Time Migration (KTM)

- Integrate over source/receiver pairs i.e., trace data, to produce image point beta:

$$\beta(x) = \int_D W(x, \xi) D(\xi, t(x, \xi)) d\xi$$

- D is seismic/trace data which is a function of both image space (x) and source/receiver space (ξ) by virtue of total, two-way traveltime t :

$$t = t_s + t_r = \sqrt{(\rho_s/v)^2 + \tau^2} + \sqrt{(\rho_r/v)^2 + \tau^2}$$

v =velocity, τ =one way traveltime, ρ =horizontal distance btwn src (s) or rec (r) and image point

- *The integral is replaced by a summation over this path*

KTM : Filtering

- KTM is essentially the Greens function solution to the scalar wave equation.
- Rho filters are applied to trace data in order to ultimately adjust the kernel of the integral to suit the dimensionality of the image and correct freq. response
- Depending on form of Greens function and thus dimension of problem space, one may filter trace data u as :

$$D(\xi, t) = \begin{cases} \int_{-\infty}^{\infty} |\omega| u(\xi, \omega) e^{-i\omega t} d\omega & 2D \\ \int_{-\infty}^{\infty} \sqrt{|\omega|} u(\xi, \omega) e^{-i\omega t + i\pi \text{sgn}(\omega)/4} d\omega & 2.5 \\ \int_{-\infty}^{\infty} i\omega u(\xi, \omega) e^{-i\omega t} d\omega & 3D \end{cases}$$

Convolution/filter in CUDA

```
// CUFFT plan
    cufftHandle plan;
    cufftSafeCall(cufftPlan1d(&plan, mem_size, CUFFT_C2C, 1));
// Transform signal and kernel
    cufftSafeCall(cufftExecC2C(plan, (cufftComplex *)d_signal, (cufftComplex
*)d_signal, CUFFT_FORWARD));
    cufftSafeCall(cufftExecC2C(plan, (cufftComplex *)d_filter_kernel,
(cufftComplex *)d_filter_kernel, CUFFT_FORWARD));
// Multiply the coefficients together and normalize the result
    ...
// Transform signal back
    cufftSafeCall(cufftExecC2C(plan, (cufftComplex *)d_signal, (cufftComplex
*)d_signal, CUFFT_INVERSE));
// Copy device memory to host
Complex* h_convolved_signal = h_signal;
cutilSafeCall(cudaMemcpy(h_convolved_signal, d_signal, mem_size,
                        cudaMemcpyDeviceToHost));
```

Performance III

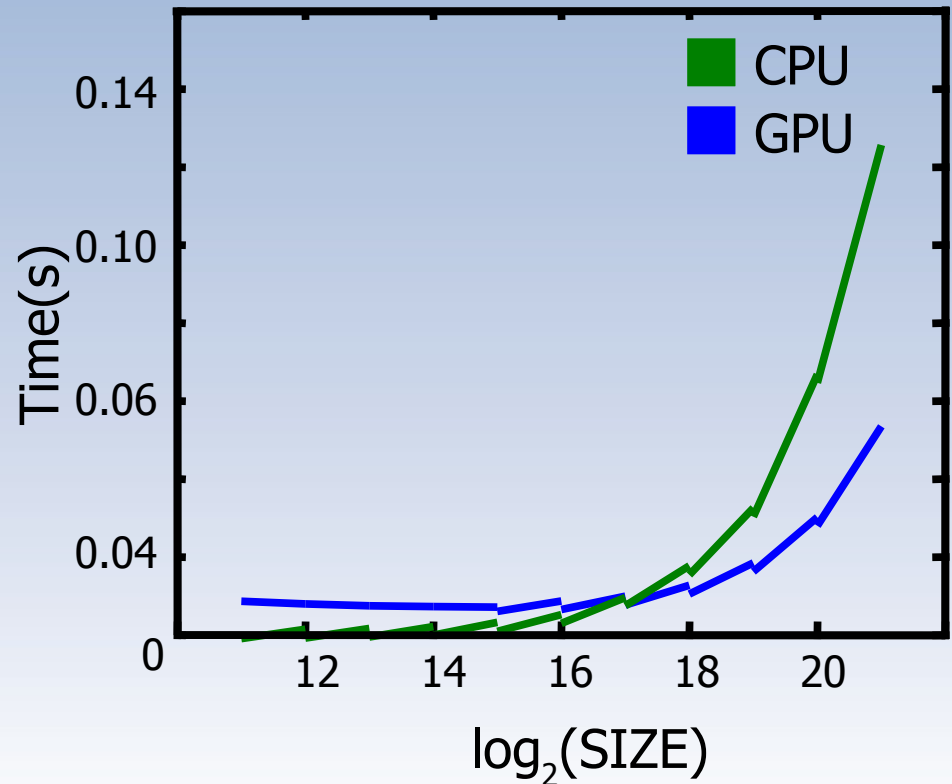
- Fourier transform is linear → apply fft/iff to:

data batch x **filter kernel**

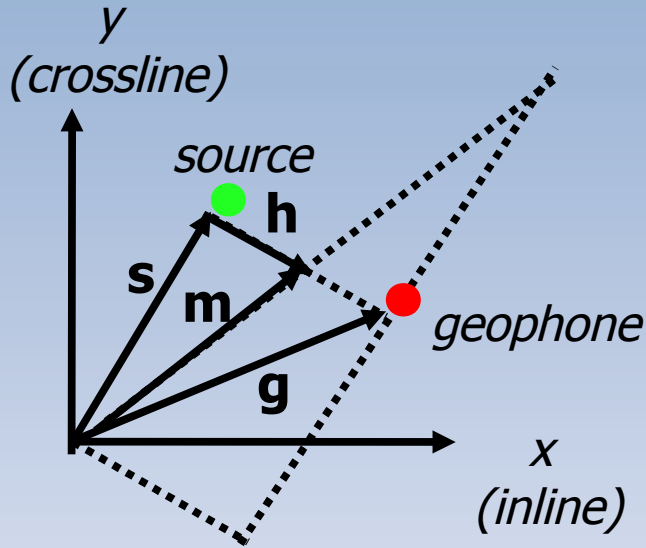
- Filter kernel has period of trace data length within batch

- Significant savings over CPU after $N \sim 128k$

- One can take advantage of existing CUFFT library



KTM: 2D Common Offset Migration

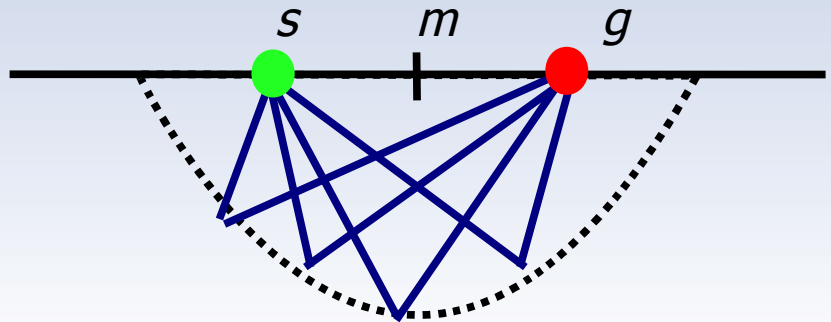


- In 2D the locus of scatterers with *constant traveltime* for src/rec is an ellipse
- Summation selects values from traces falling on these paths, for points in image plane
- Weights in summation also fn of traveltime

- One works in terms of offset-midpoint coordinates; eg., in 2D:

$$\text{Offset } h = (g-s)_x/2;$$

$$\text{Midpoint } m = (g+s)_x/2;$$



Travel Time calculation in CUDA

```
__global__ void initTravelTimeCuda(float *travelTime, float *time, float
    *slowness, float *midpoints, float offset)
{
const int thread_x = threadIdx.x + blockIdx.x * blockDim.x;
const int thread_y = threadIdx.y + blockIdx.y * blockDim.y;
const int thread_xy = thread_x + thread_y * TRACE_PTS;

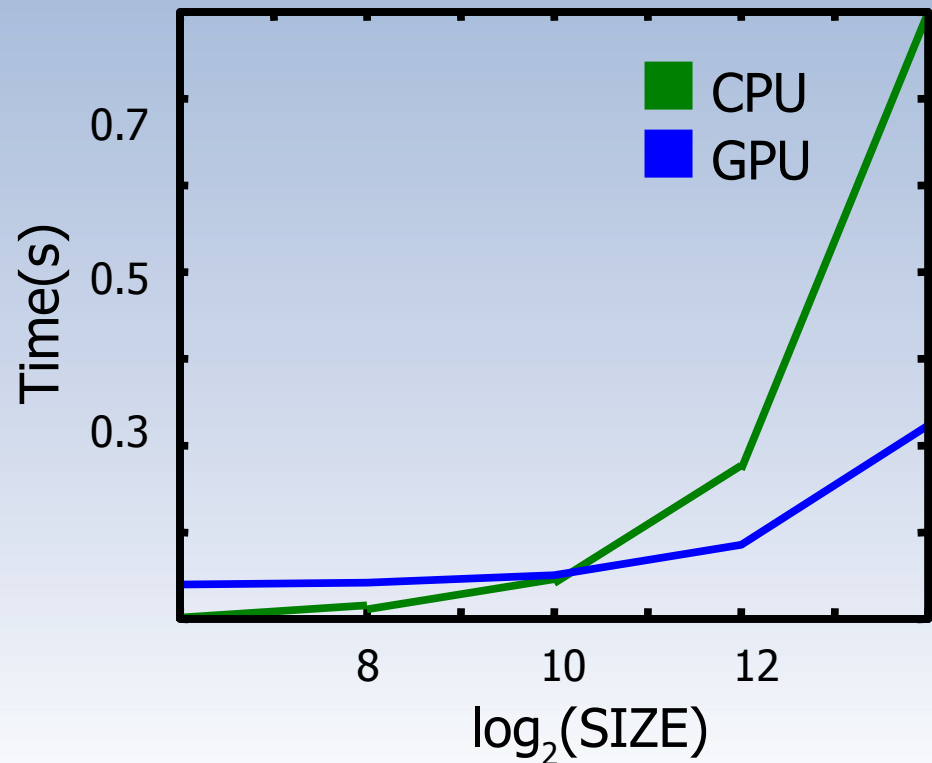
const float _time = time[thread_x];
const float _midpoint = midpoints[thread_y];
const float _slowness = slowness[thread_x];

float value = 0.0f;
value += sqrt( _time * _time + 4.0f * (_midpoint - offset) * (_midpoint -
    offset) * _slowness * _slowness);
value += sqrt( _time * _time + 4.0f * (_midpoint + offset) * (_midpoint +
    offset) * _slowness * _slowness);

travelTime[thread_xy] = 0.5f * value;
}
```

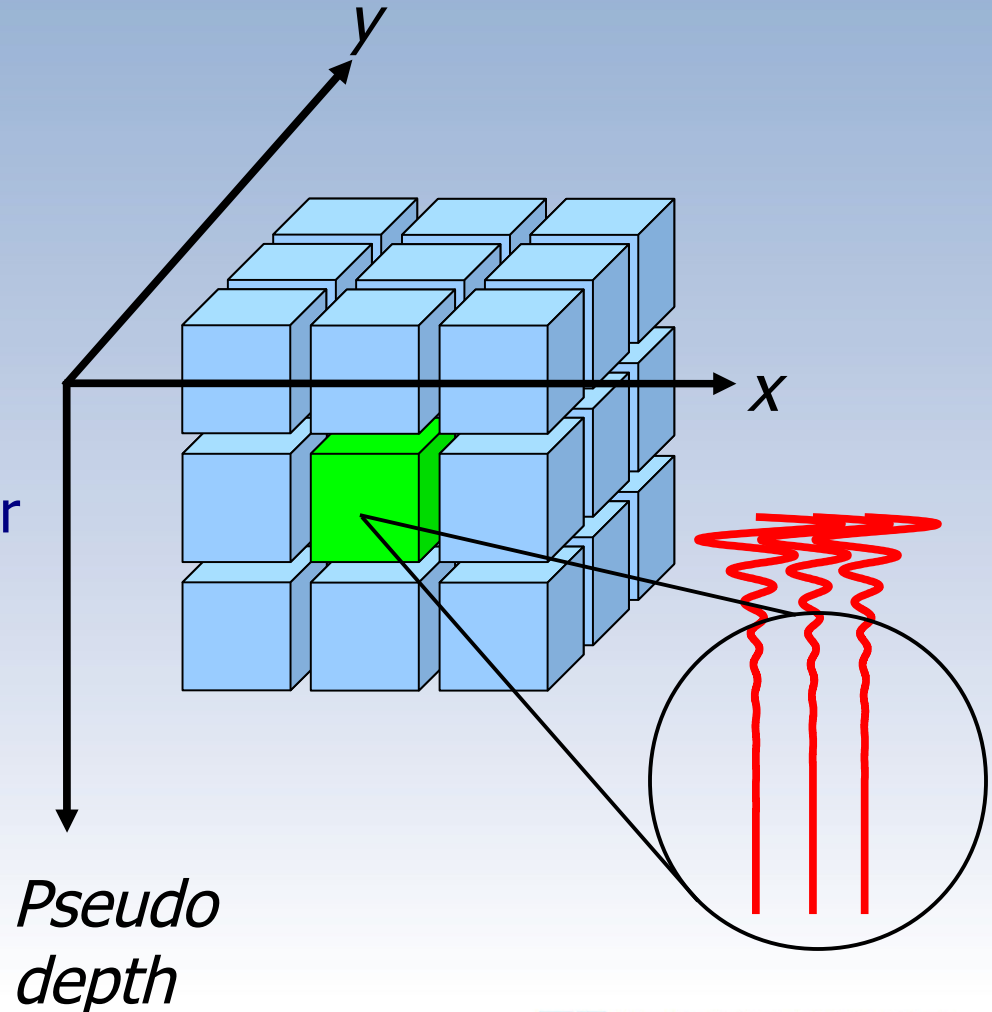
Performance IV

- Traveltime tables can be very large, obvious benefits to using GPU for $\text{SIZE} > 2^{10}$
- Makes sense to calculate on GPU, since only need to transfer $O(N)$ values, vs $O(N^2)$ - $O(N^5)$ for full table in 2/3D
- Also, *in situ* calculation of these values allows for integration with actual summation/migration step



GPU Migration

- Using architecture of GPU, migration/summation may take place using **blocks** assigned to regions of **input data**, where relevant trace data are weighted & selected according to traveltime for image point or section of image plane
- Preliminary results show an order of magnitude improvement over CPU
- Further strategies are being tested & optimized



Summary

- Heterogenous, massively parallel nature of Nvidia GPU computing ideal for Geophysics data processing and simulation
- CPU better for smaller datasets, obvious benefits from using GPU when shared block memory is utilized, host/device exchanges minimized
- Many preexisting libraries can be used (eg., CUFFT, CUBLAS etc), relatively easy to do performance hacks eg., optimize CUDA grid structure, coalesced memory reads
- Can combine visualization + numerical work using for example **Stone Ridge Cuda Visualization Framework (SCVF)**

Many thanks to Nvidia for the opportunity to speak

