Using GPUs in Pre-stack AVO Inversion for Performance Improvement

Arthur Lee

CREWES Weekly Friday Morning Meeting

Feb 8, 2019

NSERC CRSNG
CMC Containment & Monitoring Institute
UNIVERSITY OF CALGARY
FACULTY OF SCIENCE
Department of Geoscience

Outline

1. Objective and Motivation
2. Pre-stack AVO Inversion (Hampson et al., 2005) using Conjugate Gradient
3. Software Engineering Solutions to a Large Data Problem
4. Using GPUs in Conjugate Gradient
5. What is GPU Programming
6. Results
7. Conclusion
8. Discussions

Acknowledgements

Dan Hampson, CGG

CREWES
Kevin Hall, University of Calgary

Jorge Monsegny Parra, University of Calgary
Daniel Trad, University of Calgary
Larry Lines, University of Calgary

Objective and Motivation

Reflection Seismic Method

Acquisition → Processing → Interpretation and Reservoir Characterization

Acquisition → Processing → Interpretation:

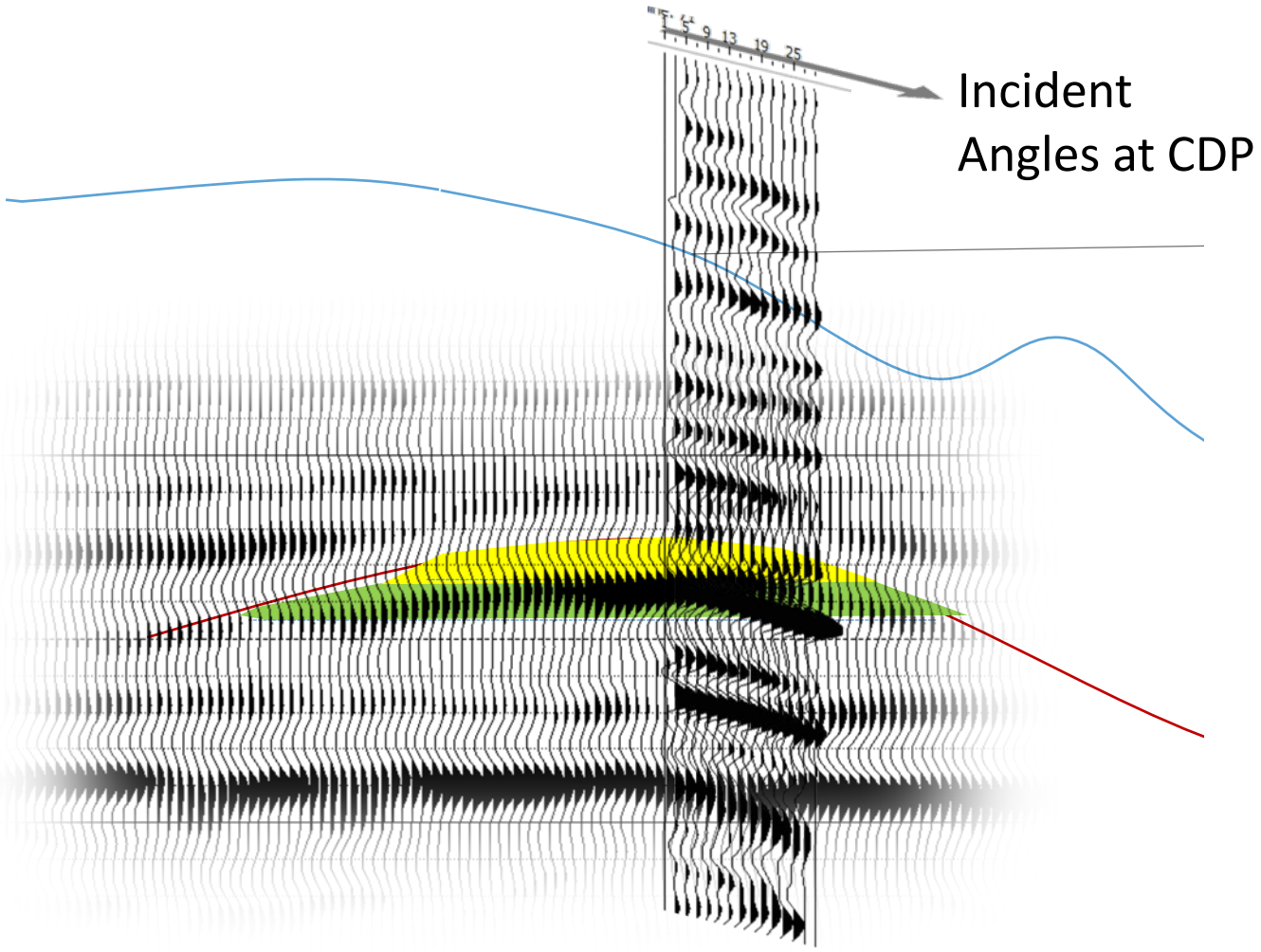
Acquisition → Processing → Pre-stack AVO Inversion → Elastic Properties (Zp, Zs, and Density)

Purposes:	(1) Reconnaissance Analysis and (2) Detailed Rock/Fluid Property Estimation
Requirements:	Interactive and Fast, and … Faster

Would GPU help?

Pre-stack AVO Inversion

Seismic Inversion is “to extract geological model information from data”. (Lines, Chapter 16)



INPUT:

Pre-stack migrated, or NMO-corrected, and AVO friendly processed Angle gathers (AVO -> AVA)

OUTPUT:

Simultaneously solves for elastic properties:

- P-Impedance,
- S-Impedance and
- Density.

Available tools (linear/non-linear):

1. Artificial Intelligent
2. Statistical
3. Deterministic

Pre-stack AVO Inversion – Assumptions

The Forward Model Development …

Equation:

$$T = \mathbf{W}\, R$$

$$T = \mathbf{W}\ (c_1 R_P + c_2 R_S + c_3 R_D)$$

$$T = \mathbf{W}\ \left(\frac{1}{2} c_1 \mathbf{D} L_P + \frac{1}{2} c_2 \mathbf{D} L_S + c_3 \mathbf{D} L_D\right)$$

Assumptions:

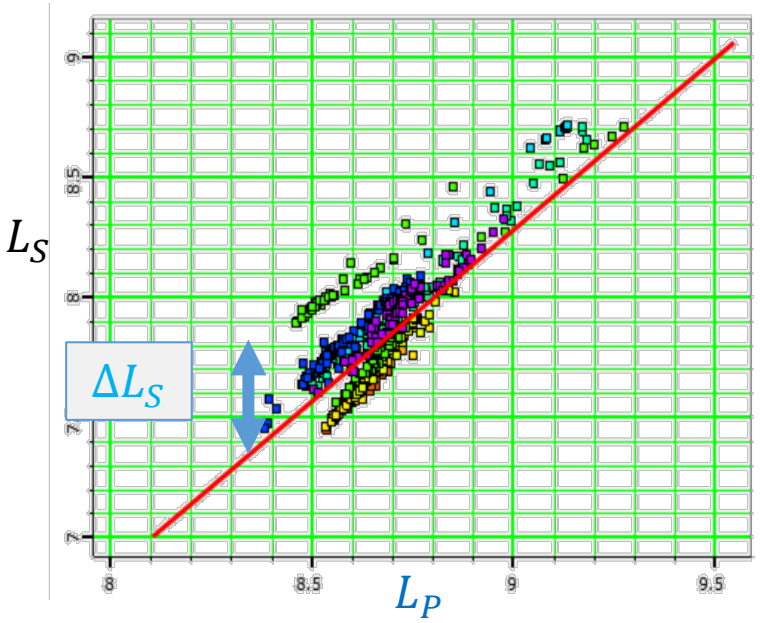
-- Convolutional Model

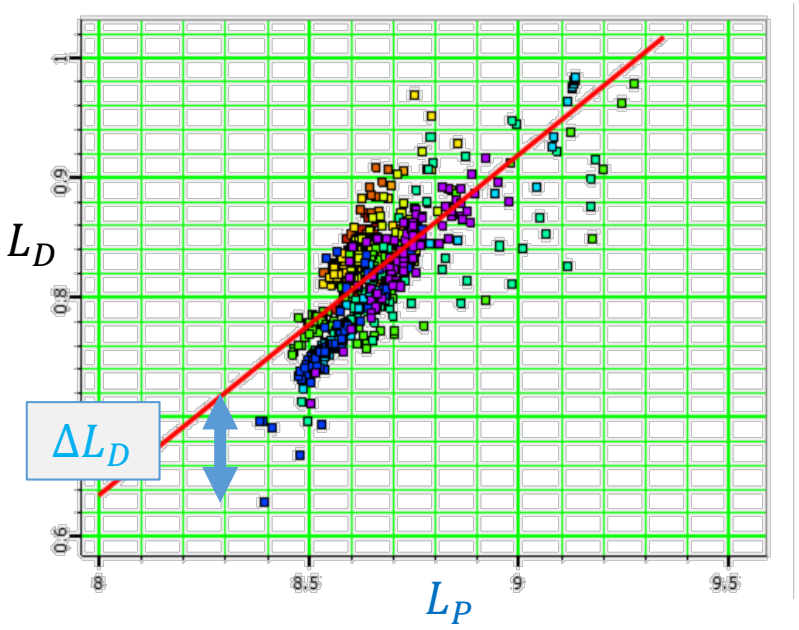
-- Linearization

-- Reflectivity as small changes in Elastic Parameters

Additional assumption by Hampson et al., 2005

Model parameters Z_S and ρ are related to Z_P.





Assume: $L_S = k\ L_P + k_c + \Delta L_S$

$L_D = m\ L_P + m_c + \Delta L_D$

Then: $\mathbf{D}L_S = k\ \mathbf{D}L_P + \mathbf{D}\Delta L_S$

$\mathbf{D}L_D = m\ \mathbf{D}L_P + \mathbf{D}\Delta L_D$

Pre-stack AVO Inversion – Assumptions by Hampson et al., 2005.

The Forward Model

Equation:

$$T = \mathbf{W}\, R$$

$$T = \mathbf{W}\, (c_1 R_P + c_2 R_S + c_3 R_D)$$

$$T = \mathbf{W}\, \left(\frac{1}{2} c_1 \mathbf{D} L_P + \frac{1}{2} c_2 \mathbf{D} L_S + c_3 \mathbf{D} L_D\right)$$

$$T = \mathbf{W}\, (\tilde{c_1} \mathbf{D} L_P + \tilde{c_2} \mathbf{D} \Delta L_S + \tilde{c_3} \mathbf{D} \Delta L_D)$$

Assumptions:

-- Convolutional Model

-- Linearization

-- Reflectivity as small changes in Elastic Parameters

-- Elastic Parameters are related.

-- Invert for elastic parameter changes from a trend.

Pre-stack AVO Inversion (Hampson et al., 2005)

The equation in Matrix Notation for one angle gather:
(with *M* angle traces per gather, and *N* samples per trace)

$$\begin{bmatrix} T(\theta_1) \\ \vdots \\ T(\theta_2) \\ \vdots \\ \ddots \\ T(\theta_M) \\ \vdots \end{bmatrix} = \begin{bmatrix} \mathbf{W_1}(\theta_1) \dots & \mathbf{W_2}(\theta_1) \dots & \mathbf{W_3}(\theta_1) \dots \\ \vdots & \vdots & \vdots \\ \mathbf{W_1}(\theta_2) \dots & \mathbf{W_2}(\theta_2) \dots & \mathbf{W_3}(\theta_2) \dots \\ \vdots & \vdots & \vdots \\ \ddots & \ddots & \ddots \\ \mathbf{W_1}(\theta_M) \dots & \mathbf{W_2}(\theta_M) \dots & \mathbf{W_3}(\theta_M) \dots \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} L_P \\ \vdots \\ \Delta L_S \\ \vdots \\ \Delta L_D \\ \vdots \end{bmatrix}$$

Wavelets and linearized AVO coefficients combined at different angles.

$$\Rightarrow \quad T = \mathbf{A}m$$

The basic equation.

Pre-stack AVO Inversion (Hampson et al., 2005)

Basic Equation: $T = \mathbf{A}m$

Normal Equation: $\mathbf{A}^{\mathrm{T}}T = \mathbf{A}^{\mathrm{T}}\mathbf{A}m$

RHS:
$$\mathbf{A}^{\mathrm{T}}\mathbf{A}m = \begin{bmatrix} \mathbf{D}^{\mathrm{T}}\mathbf{G_{11}}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G_{12}}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G_{13}}\mathbf{D} \\ \mathbf{D}^{\mathrm{T}}\mathbf{G_{21}}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G_{22}}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G_{23}}\mathbf{D} \\ \mathbf{D}^{\mathrm{T}}\mathbf{G_{31}}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G_{32}}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G_{33}}\mathbf{D} \end{bmatrix} \begin{bmatrix} L_P \\ \Delta L_S \\ \Delta L_D \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{D}^{\mathrm{T}}\mathbf{G_{11}}\mathbf{D}\, L_P + \mathbf{D}^{\mathrm{T}}\mathbf{G_{12}}\mathbf{D}\, \Delta L_S + \mathbf{D}^{\mathrm{T}}\mathbf{G_{13}}\mathbf{D}\, \Delta L_D \\ \mathbf{D}^{\mathrm{T}}\mathbf{G_{21}}\mathbf{D}\, L_P + \mathbf{D}^{\mathrm{T}}\mathbf{G_{22}}\mathbf{D}\, \Delta L_S + \mathbf{D}^{\mathrm{T}}\mathbf{G_{23}}\mathbf{D}\, \Delta L_D \\ \mathbf{D}^{\mathrm{T}}\mathbf{G_{31}}\mathbf{D}\, L_P + \mathbf{D}^{\mathrm{T}}\mathbf{G_{32}}\mathbf{D}\, \Delta L_S + \mathbf{D}^{\mathrm{T}}\mathbf{G_{33}}\mathbf{D}\, \Delta L_D \end{bmatrix}$$

Autocorrelation

where: $\mathbf{G_{jk}} = \sum_i^M \tilde{c}_j(\theta_i)\mathbf{W}^T(\theta_i)\mathbf{W}(\theta_i)\tilde{c}_k(\theta_i)$

Pre-stack AVO Inversion (Hampson et al., 2005)

Basic equation: $T = \mathbf{A}m$

Normal equation: $\mathbf{A}^{\mathrm{T}}T = \mathbf{A}^{\mathrm{T}}\mathbf{A}m$

Conjugate Gradient Algorithm by Hestenes and Stiefel (1952) as mentioned by Scales (1989):

Initialize {

Model Parameters Vector: $m = [0 \quad 0 \quad 0]^{\mathrm{T}}$ (With zero initial guess.)

Residual Vector: $r = \mathbf{A}^{\mathrm{T}}T - (\mathbf{A}^{\mathrm{T}}\mathbf{A})m$

Error Squared, a scalar: $r^{\mathrm{T}}r$ (Is a least squares solution.)

Search Vector: $p = r$

Search Multiplier, a scalar: $\beta = 0$

}

For all iterations {

If not the first iteration {

$\beta = r^{\mathrm{T}}r / r^{\mathrm{T}}r_{old}$ (Calculate error and save for the next iteration.)

$p = r + \beta p$ (Modify the search vector.)

}

$\alpha = r^{\mathrm{T}}r / \left(p^{\mathrm{T}}(\mathbf{A}^{\mathrm{T}}\mathbf{A})p\right)$ (Obtain a model step size. Stop if denominator is 0.)

$r = r - \alpha\,(\mathbf{A}^{\mathrm{T}}\mathbf{A})p$ (Update residual error.)

(Stop if error $r^{\mathrm{T}}r$ increases.)

$m = m + \alpha\, p$ (Update model parameters with a gradient step.)

}

Pre-stack AVO Inversion (Hampson et al., 2005)

Hampson's conjugate gradient implementation concentrates on the combined $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ operator.

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}m = \begin{bmatrix} \mathbf{D}^{\mathrm{T}}\mathbf{G}_{11}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G}_{12}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G}_{13}\mathbf{D} \\ \mathbf{D}^{\mathrm{T}}\mathbf{G}_{21}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G}_{22}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G}_{23}\mathbf{D} \\ \mathbf{D}^{\mathrm{T}}\mathbf{G}_{31}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G}_{32}\mathbf{D} & \mathbf{D}^{\mathrm{T}}\mathbf{G}_{33}\mathbf{D} \end{bmatrix} \begin{bmatrix} L_P \\ \Delta L_S \\ \Delta L_D \end{bmatrix}$$

- It employs 9 convolutions which can be running in ***parallel***.
- The implementation should fit well into a typical multi-threading optimization style.

Software Engineering Solutions to a Large Data Problem

Typical software engineering solutions (using existing science):

Coarse-grained Parallelization

Multiple Nodes (CPU's / Machines / Clusters)
Network connection. Non-shared memory. E.g. MPI

Medium-grained Parallelization

Multiple threads in one CPU
Threads ~= cores. Shared memory. E.g. pthread, OpenMP or MPI

Fine-grained Parallelization

Multiple threads (in one CPU) E.g. pthread, OpenMP
Multiple GPU cards connected by PCI-e

heterogenous

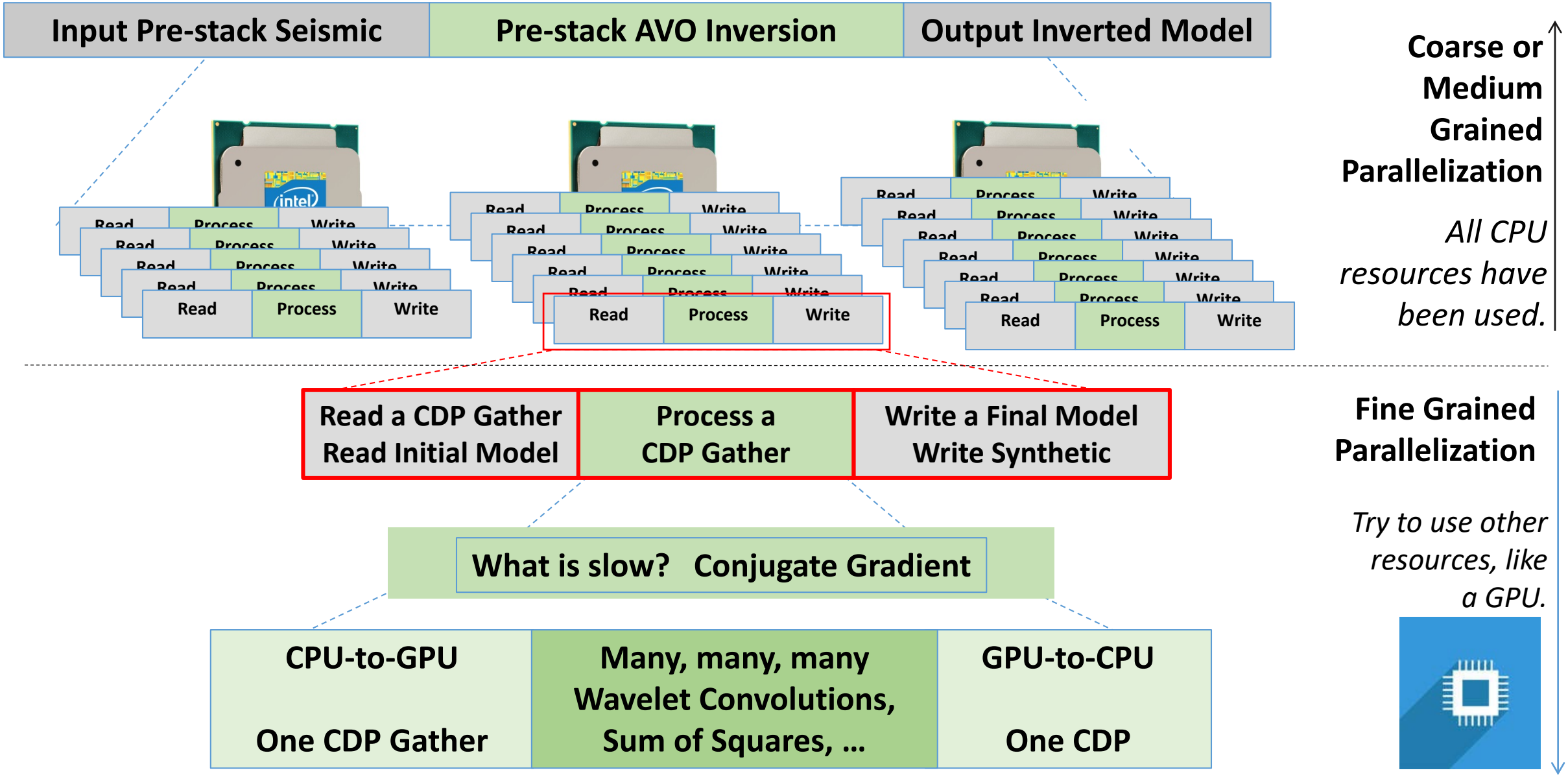
Ultra-fine-grained Parallelization

Multiple threads in a single GPU card
Array or Vector Processors

E.g. "OpenACC", "**CUDA**",… } **SIMD**

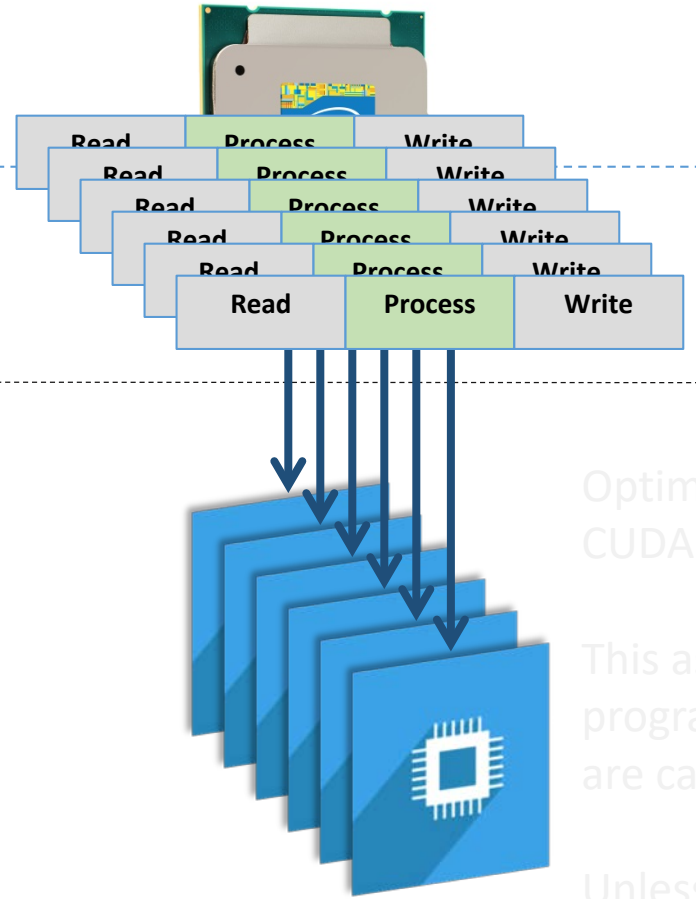
(Not seen anymore in geophysical industry.)

Using GPUs in *the* Pre-stack AVO Inversion



Using GPUs in AVO Inversion

Input Pre-stack Seismic	Pre-stack AVO Inversion	Output Inverted Model



Coarse or Medium Grained Parallelization

Fine Grained Parallelization using GPUs

Optimally, we want to have one CUDA device for one CPU thread.

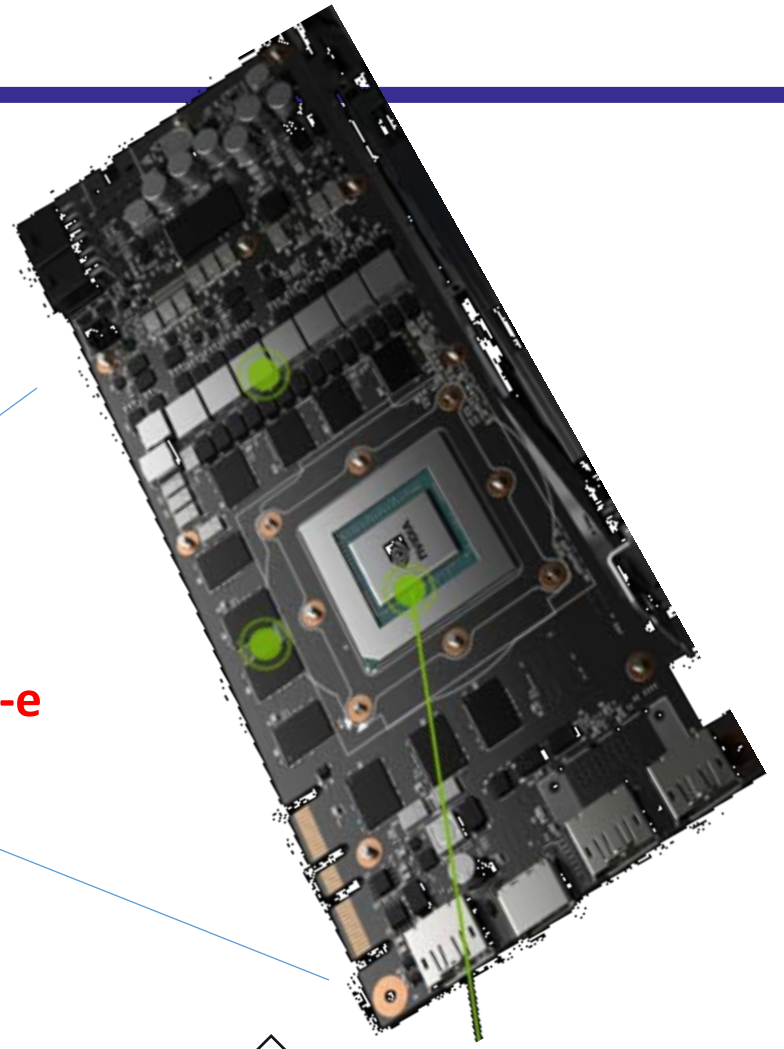
This assumes that our programming skill is good and we are capable to fully occupy a GPU!

Unless the CPU program cannot be multi-threaded.

What is a GPU? - NVIDIA GeForce GTX 1080

CPU

PCI-e



Nvidia Device

Programming Language:
CUDA

Nvidia Device

GPU

What is CUDA?

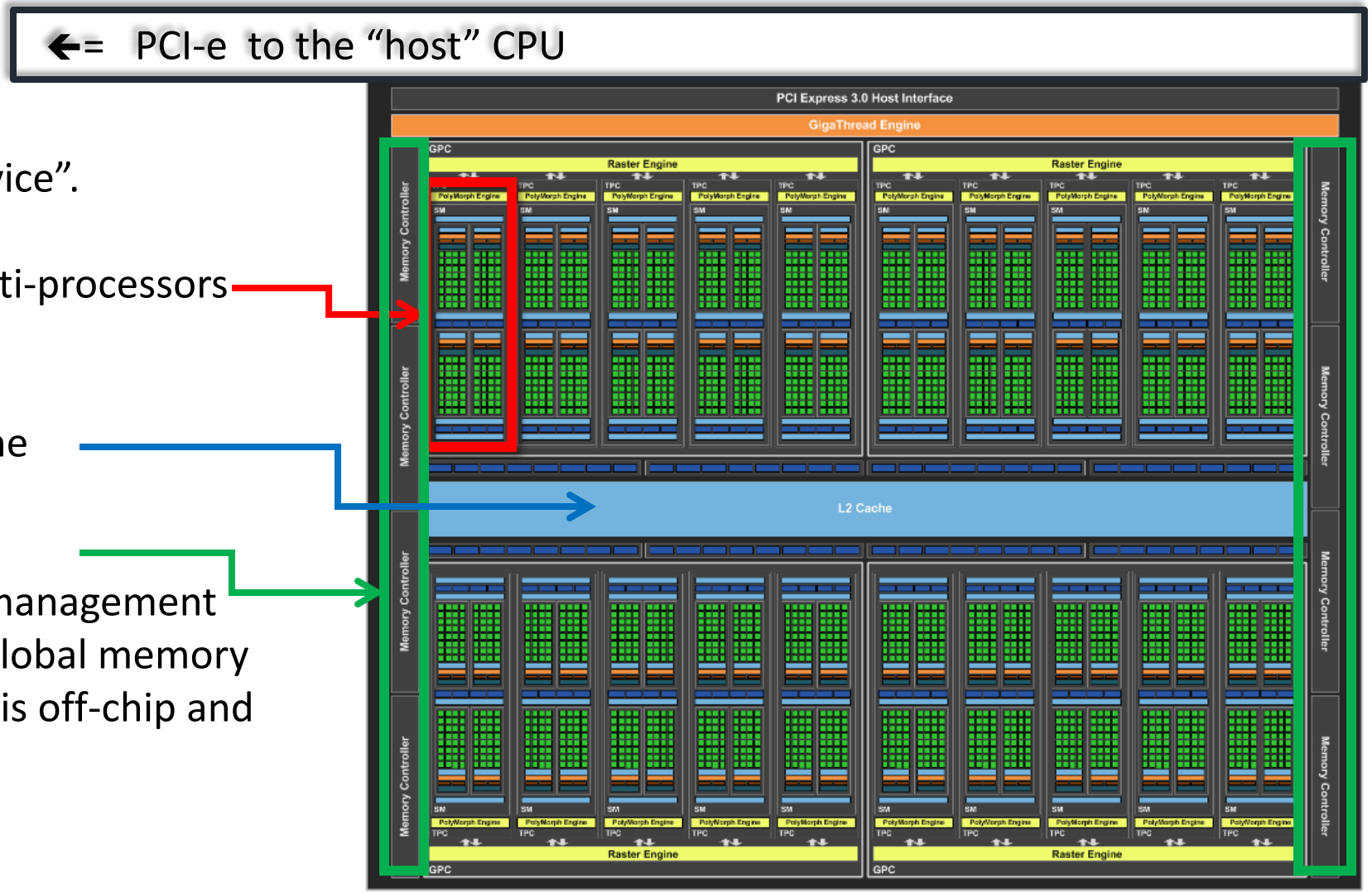
As described in www.nvidia.com,

- parallel computing architecture
- many computing cores → mathematical calculations
- supports OpenCL and Direct*X*
- supports C and Fortran.

In other words, it is a C/C++ pre-processor and compiler which comes with some libraries.

A Hard way to explain the Software operations

Block Diagram of the GP104 chip of GeForce GTX 1090, a 4 cluster card.



This is called one GPU or one "device".

- 4 clusters of 5 = 20 Stream Multi-processors (**SM** or **MP**)
- 8 x 256KB = 2,048KB of **L2** cache
- 8 x 32-bit memory controllers (i.e. 256 bit wide) for channel management between L2 cache and device global memory GDDR. Global memory (GDDR) is off-chip and not shown here.

Figure 4: Block Diagram of the GP104 GPU

GP104 Multiprocessor Diagram

Available Cores:

In one **Stream Multiprocessor (SM or MP)**:

- 4 **Warps**
- Each warp has 4 x 8 = 32 **Cores**

In one **Cluster**:

- 5 SM x 4 Warps/SM x 32 Cores/Warps
- = 640 Cores

In one **Device**:

- 4 Clusters x 640 Cores/Cluster
- = 2560 Cores (or Threads)

Available Memory:

- register file capacity
  (16,384x32bit) x 4 = (16Kx4)x4 = 256KB

- 48 KB L1 cache

- 96 KB shared memory

Figure 5: GP104 SM Diagram

CUDA Basics – Write your own kernel function

CPU: C/C++ simple example:

```
for (int i =0; i < N; i++) { function(i, a, b, c …); }
```

GPU: corresponding CUDA kernel function:

```
function<<< Block Dimension, Grid Dimension >>>(a, b, c )
{
    //Code for one thread, asynchronous:
    int i =    (blockIdx.x * blockDim.x ) + threadIdx.x;
    if (i < N) { …. function(i, a,b,c);  }
}
```

← Row major linear arrangement of blocks of threads.

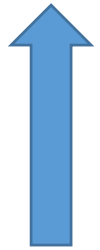
CUDA Basics – Provided kernel functions e.g. FFT

CPU: C/C++ simple example:

```
:
fftwf_plan                d_plan_forward =
fftwf_plan_dft_r2c_1d     ( myNFFT,  d_real,  d_complex,  FFTW_ESTIMATE);
:
fftwf_execute             ( d_plan_forward );
:
```

Corresponding CUDA GPU kernel launches:

```
:
cufftHandle          d_plan_forward = 0;
cufftPlan1d          (&d_plan_forward,  myNFFT,  CUFFT_R2C,  N );
:
cufftExecR2C         (  d_plan_forward,  d_real,  d_complex );
:
```



A Batch Option

GPU Programming – Only two main ideas are involved

1) Putting serial workloads to the CPU (host) and parallel workloads to the GPU (device).

Conjugate gradient iterative solver:

Loop (a serial process) in CPU.

FFT, additions and subtractions (parallel data) in GPU.

For all iterations {

If not the first iteration {

$\beta = r^{\mathrm{T}} r / r^{\mathrm{T}} r_{old}$

$p = r + \beta p$

}

$\alpha = r^{\mathrm{T}} r / \left(p^{\mathrm{T}} (\mathbf{A}^{\mathrm{T}} \mathbf{A}) p\right)$

$r = r - \alpha (\mathbf{A}^{\mathrm{T}} \mathbf{A}) p$

$m = m + \alpha p$

}

2) Coalescing data access to avoid memory bandwidth limitations. (To be discussed in the future.)

Results

Current Status and Expectation:

- Similar performance after changing the conjugate gradient from **CPU** to **GPU**. The CPU thread can be freed up for other activities.
- Amazon's Best CAD$.
- In our tests, the CPU is running in a single core. Two times will roughly balance the cost only. We hope that the GPU can run ~10 times faster.

Future Work:

- Optimize the conjugate gradient implementation.
- Invert multiple CDPs in parallel.

C$1817
Intel
Xeon
W3680
6 cores

C$581
Quadro
P2000
1024 cores

C$388
GeForce
GTX 1060
1280 cores

Conclusion

Converted from CPU to GPU a linearized pre-stack AVO least-squares inversion.

Pros:

Highly parallel and scalable. Suitable for conjugate gradient methods. Work well for a large number of data samples and number of iterations.

Cons:

Steep learning curve.

Sensitive to hardware setup.

Code changes may easily run --- slower!

Extra attention is needed for performance profiling and tuning.

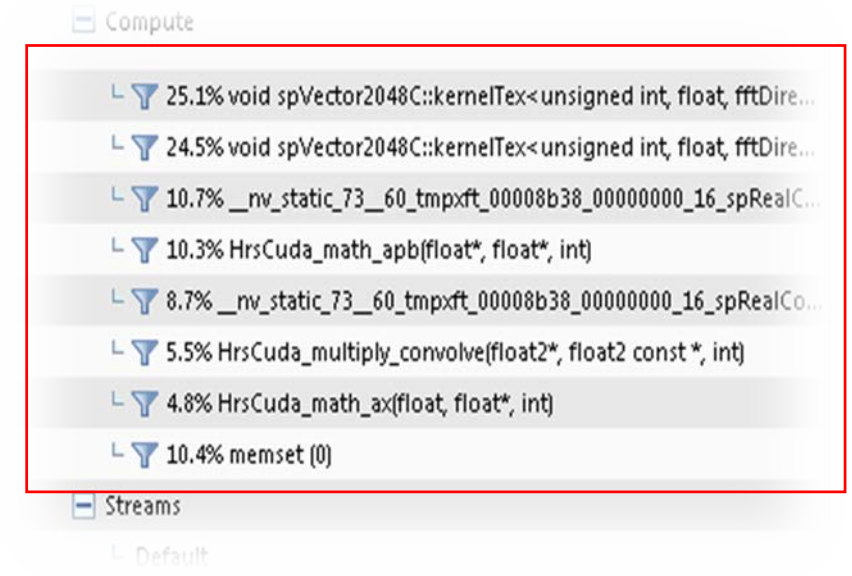
Takeaway – Three steps to deliver.

Three steps to deliver:

1. Profile on CPU, identify and evaluate the performance bottleneck carefully.

2. Code and debug in CUDA environment. Use existing libraries.

3. Analyze and improve core occupancy, memory bandwidth and other GPU resource usages with tools provided.



Takeaway – CUDA Features

The following may interest you:

1. Linear Algebra and Sparse Matrix handling. (**cuBLAS** and **cuSPARSE**)

2. Linear Solvers like Cholesky, SVD. Spare and dense matrix. (**cuSOLVER**)

3. Artificial Intelligent (**cuDNN**)

4. Shared memory (low latency memory) →
   Laplace operator stencils in Finite Difference methods.

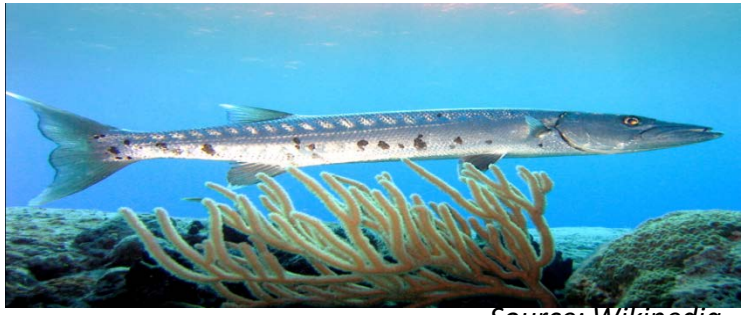
5. Texture memory →
   trace data interpolation ?

Resources

Hardware resources at University of Calgary: barracuda

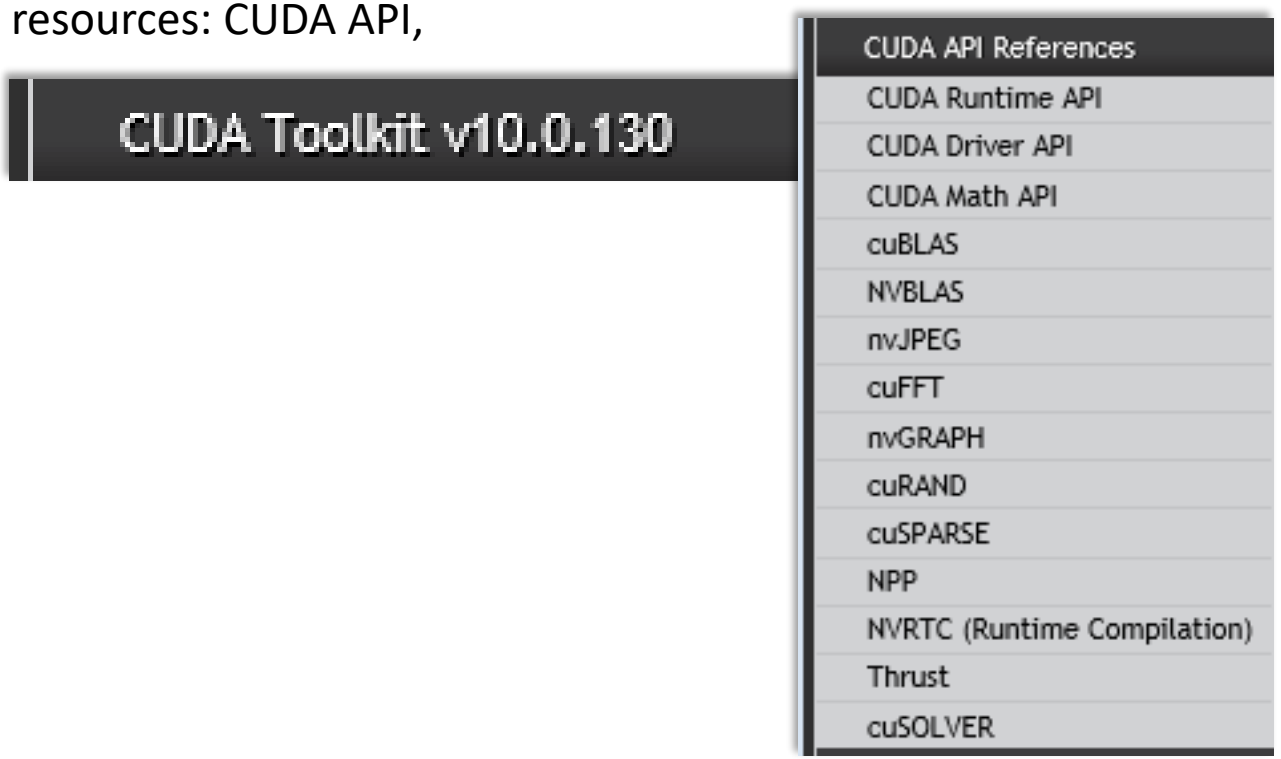
Two CUDA enabled graphic cards.

Tesla K10 GK104GL dual Kepler (v3) processors.



Source: Wikipedia

Software resources: CUDA API,



and YOU to build up the CREWES library!