

## Application of HPC methods for Radon and Deblending

*Kai Zhuang, Daniel O. Trad*  
*CREWES – University of Calgary*

### Summary

We implemented both a sparse Radon transform and a least-squares deblending algorithm in C++ using different parallel processing methods. In this paper, we compare three different Application Programming Interfaces (APIs) to perform parallel computing on large datasets, using openMP, openMPI, and CUDA. Our goal is to understand the scaling of different parallel processing methods with our codes and explore the advantages and drawbacks of each API. The sparse Radon transform is easily parallelizable as each Radon frame is calculated independently from each other, which results in greatly reduced calculation time proportional to the resources given. On the other hand, the least-squares deblending algorithm is not efficient when implemented with openMPI as the least-squares gradient requires the application of the blending forward and adjoint operators that involve resorting the data from the entire dataset at each iteration. Therefore, the openMPI implementation of least-squares deblending requires collecting all the data in a single main node at every iteration, thus adding a significant overhead because of data transfer that often outweigh the computing performance gain. By implementing the deblending in CUDA on a single local machine, we then significantly reduce data copying overhead and increase the computational speed of the gradient calculation. Most likely a CUDA implementation with distributed GPUs (GPU clusters) would suffer from similar issues as the openMPI version for inversion but is not tested for this report.

### Introduction

As datasets become larger and data processing becomes more complex, parallel processing methods are needed to efficiently process these large and often repetitive workflows. Advancements in technology, especially in the consumer GPU market in recent years provide a very high-performance alternative to the traditional CPU based parallel processing methods for mathematical operations. Although many seismic workflows are highly parallelizable, depending on the implementation some algorithms may see limited benefits from some parallelization methods. As multi-core CPUs became more mainstream, a way to easily write codes that uses these extra cores was created under the name "Open Multi-Processing" or openMP for short. OpenMP is an application programming interface in the form of a compute library created for C and C++ to parallelize data using shared memory. Due to the simplicity of its implementation, openMP is widely used as an easy way to parallelize mathematical operations in scientific computation. Open Message Passing Interface (openMPI) is a library created for computation using multiple CPUs together, where each CPU exists on distinct systems and where memory is not shared. When openMPI is used in conjunction with openMP many jobs can be done quickly in parallel. While originally used only for graphics processing, over the years advancements in GPU technology have made general parallel computing on the GPU more and more appealing, known as general-purpose GPU computing or GPGPU. As the graphics processing unit (GPU) is traditionally used for computer graphics their architecture is purpose-built differently from a CPU's. A GPU is normally built with an abundance of cores, often two or three orders of magnitude more

than the core number for normal CPUs. Compared to CPUs, GPU threads are designed to do the same task repeatedly which is very useful for scientific processing on large datasets.

## Implementation

To properly code the hyperbolic Radon Transform to run efficiently in openMP, we send outer offset loops to separate threads to calculate the transform in the adjoint operator, for the forward operator we then send each outer slowness loop instead. The pseudo-code for the Radon transform can be written as seen in Algorithm 1.

---

### Algorithm 1 Radon pseudo code

---

```

1: function RADON
2:   #PRAGMA OMP PARALLEL FOR           ▷ insert for parallelization of loop
3:   for q = slowness do
4:     for h = offset do
5:        $moveout = h^2 * q$ 
6:       for it = 0 to nt do
7:          $time = \sqrt{(it * dt)^2 + moveout}$ 
8:          $model[iqNt + it] += data[ihNt + INT(time/dt)]$ 
9:       end for
10:    end for
11:  end for
12: end function

```

---

For our implementation in CUDA, we set up a three-dimensional thread configuration one dimension for each loop iteration in the standard code, then calculate the transform in a single step with each thread doing a single calculation. To implement the HRT in openMPI we send the data pertaining to each shot in the transform to separate nodes and then calculate transform independently on each node using either the CUDA or openMP methods stated above, we then collect the data at the master at the end of the calculation. The CUDA pseudo-code can be seen in Algorithm 2.



---

**Algorithm 2** CUDA Radon pseudo code

---

```
1: __global__
2: function CUDA_RADON
3:   int it = blockIdx.x * blockDim.x + threadIdx.x
4:   int iq = blockIdx.y * blockDim.y + threadIdx.y
5:   int ih = blockIdx.z * blockDim.z + threadIdx.z
6:   if (iq >= nq || ih >= nh || it >= nt) return
7:   int ihNt = ih*nt
8:   int iqNt = iq*nt
9:   double timemax = dt*nt
10:  double moveout =  $h^2 * q$ 
11:  double time =  $\sqrt{(it * dt)^2 + moveout}$ 
12:  atomicAdd(&model[iqNt + it], data[ihNt + INT(time/dt)])
13: end function
```

---

Because a significant amount of parallel applications use multidimensional data, the CUDA API groups thread blocks into multiple dimensions, up to three. This is beneficial to the implementation of Radon as there are generally three data dimensions for the transform, one for offset, time, and velocity. By assigning each of our variables to a dimension we can eliminate their respective loops within our CUDA kernel which otherwise contains the same calculations as our standard function. The elimination of loops within the CUDA kernel is important as it would otherwise introduce a significant slowdown in the kernel because of the limited GPU instruction set. If a loop is required, the loop should be inserted outside of the kernel for maximum efficiency. One issue we encountered in creating the CUDA code for the RADON transform is that due to non-linear access of data for the adjoint modeling (along a hyperbolic path), we were not able to utilize shared memory access to decrease compute times. For the forward operator, we were able to read from shared memory as access was linear through time mapping to hyperbolic arrivals. The least-squares deblending algorithm that we implemented for this report uses a combination of the previous sparse Radon transform operator in conjunction with the blending operator to deblend seismic data. The deblending operator is used to map data to and from the blended domain to the standard shot domain. Due to the blending operator being ill-posed, an inverse cannot be assessed, thus we combine the blending operator with the Radon operator, where the sparse Radon transform is used as a sparse constraint along with hyperbolic arrivals for the inversion. By combining the two operators into a single operator for the least-squares fit we can then solve for the sparsest solution in the Radon domain that maps to the blended data effectively deblending it (Zhuang et al., 2019). This algorithm is interesting in that most of it can be optimized in parallel but there exist many challenges to proper parallelization. The main challenge of optimizing and parallelizing the algorithm is due to the large memory swaps required every iteration. These memory issues arise due to the constant need to change domains in the blending operator resulting in large-scale non-sequential access to the data. The other challenge we face is that this algorithm cannot be properly parallelized across servers used openMPI as the gradient calculation at every iteration needs to be calculated using the blended operator which requires resorting all data.

## Results and Conclusions

Processor (API)	Radon Transform Time (s)	LS Deblending Time (s)
i7 8750H 6-core (openMP)	3026	1625
i7 9800x 8-core (openMP)	3630	1973
TR 3960x 24-core (openMP)	997	570
GTX 1080 (CUDA)	789	458
RTX 2060 MAX-Q (CUDA)	1273	786
RTX 2060 Super (CUDA)	1010	588
TR 3960x (openMPI)	420	—

Table 1. Table of average runtimes for Radon and Least squares deblending on different machines.

It can be seen that there is a significant computational advantage moving from CPU processing (openMP) to GPU based processing (CUDA) by significant amounts. The openMPI results however show a significant boost compared to both the CUDA and openMP results, this difference can mainly be attributed to the overhead from the least-squares optimization. In standard Radon transform testing, the CUDA-based code reduced compute time by over an order of magnitude compared to the openMP version, after the introduction of the least-squares algorithm the advantage diminished by a large amount. Due to the openMPI distributing a work including the LS optimization across the nodes, it can run the LS optimization in a parallel fashion that neither the openMP nor CUDA code can, resulting in an overall faster computational time versus both the CUDA and openMP implementations. It must be noted however that open-MPI used the most amount of ram in this exercise, about 122GB, compared to the openMP and CUDA implementations which used 8GB of ram and 8GB + 8GB VRAM respectively.

Compared to CPUs, GPUs are easier for scaling and upgrading as multiple GPUs can be installed into a single system without the need to upgrade other parts. CPUs in comparison are only supported to a maximum of two per motherboard while also requiring a motherboard upgrade when switching to newer generation CPUs. In our comparison, the only comparable CPU in our lineup is significantly more expensive than its GPU counter-parts. In a comparison of APIs, implementation with openMPI is the most complex as well as the most scalable, though it was not tested due to equipment limitations, an openMPI implementation using in conjunction with CUDA on multi GPU nodes would yield better results compared to openMPI implemented with openMP. With current advancements in graphics processing technology, it is clear through our benchmarks that GPU processing for scientific computing is both significantly faster than comparable CPUs as well as easy to implement on nvidia GPUs using CUDA.

## Acknowledgements

We thank the sponsors of CREWES for continued support. This work was funded by CREWES industrial sponsors, NSERC (Natural Science and Engineering Research Council of Canada) through the grants CRDPJ 461179-13 and CRDPJ 543578-19.

## References

- Barney, B., 2020, Openmp: Lawrence Livermore National Laboratory.  
URL <https://computing.llnl.gov/tutorials/openMP/>
- Foster, D. J., and Moshier, C. C., 1992, Suppression of multiple reflections using the radon transform: *GEOPHYSICS*, 57, No. 3, 386–395.
- Guitton, A., 2000, Prestack multiple attenuation using the hyperbolic radon transform: Stanford Exploration Project Report, 103, 181–201.
- Huo, S., Luo, Y., and Kelamis, P. G., 2012, Simultaneous sources separation via multidirectional vector-median filtering: *GEOPHYSICS*, 77, No. 4, V123–V131.
- Ibrahim, A., and Sacchi, M. D., 2015, Fast simultaneous seismic source separation using Stolt migration and demigration operators: *GEOPHYSICS*, 80, No. 6, WD27–WD36.
- Nickolls, J., Buck, I., Garland, M., and Skadron, K., 2008, Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?: *Queue*, 6, No. 2, 40–53.  
URL <https://doi.org/10.1145/1365490.1365500>
- Sacchi, M. D., and Ulrych, T. J., 1995, High-resolution velocity gathers and offset space reconstruction: *GEOPHYSICS*, 60, No. 4, 1169–1177.
- Thorson, J. R., and Claerbout, J. F., 1985, Velocity-stack and slant-stack stochastic inversion: *GEOPHYSICS*, 50, No. 12, 2727–2741.
- Trad, D. O., Ulrych, T. J., and Sacchi, M. D., 2002, Accurate interpolation with high-resolution time-variant radon transforms: *GEOPHYSICS*, 67, No. 2, 644–656.
- Zhuang, K., Trad, D., and Ibrahim, A., 2019, Sparse inversion based deblending in cmp domain using radon operators: *CREWES Research Report*, 31.