Comparison of high performance computing methods for high resolution Radon transform and deblending

Kai Zhuang, and Daniel Trad

ABSTRACT

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. For our comparison, we will be utilizing the sparse Radon transform and least squares deblending as our focus algorithms that will benefit from parallelization. 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. When a user needs more processing power than what a single CPU can offer, then compute clusters are used in conjunction with OpenMP. 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 simpler processing tasks with a simpler instruction set, and cannot do complex single-core computations as efficiently as a CPU, such as speculative execution. However, for most scientific computations that mainly rely on simple matrix operations, the parallel workloads benefit much more from the sheer amount of parallel cores.

PARALLEL PROGRAMMING METHODS

openMP

OpenMP is an API that allows for parallel processing in C/C++, where the main thread on a CPU splits a task among a set of sub-threads that runs simultaneously. These parallel operations are defined explicitly with API calls to openMP. After the threads conclude their parallel operations the parallel threads then terminate, and then the program continues in single-threaded mode. The amount of threads used in a parallel call in openMP is dynamically allocated by the workload between the number of threads required or a defined maximum. This maximum is either the maximum number of threads in a system or otherwise arbitrarily defined by the user. Although openMP is relatively easy to implement and can be very useful given the proliferation of multi-core CPUs in modern systems, it has the problem that it does not scale up well for more than dozens of threads due to possible overhead issues.

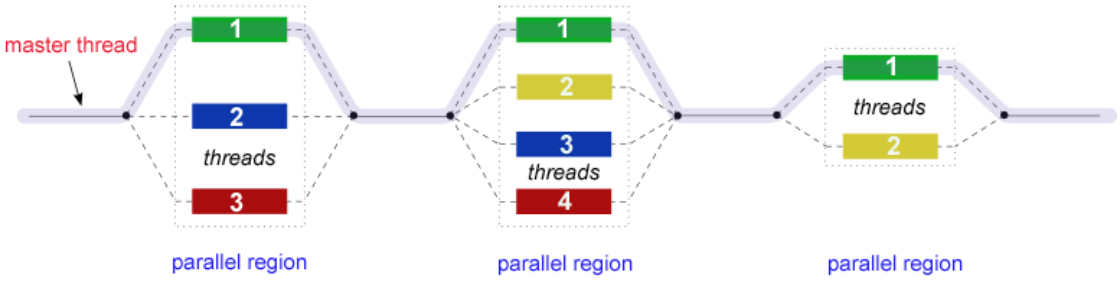

FIG. 1. Illustration of openMP parallelization (Barney, 2020).

CUDA

In 2007 the graphics card engineering company, Nvidia, introduced the Compute Unified Device Architecture API, otherwise known as CUDA. CUDA allows programmers to utilize the GPU rendering engine to be used for general-purpose parallel computing (Nickolls et al., 2008). CUDA is built on top of the C/C++ language and requires Nvidia's proprietary compiler called NVCC in addition to a standard C/C++ compiler. CUDA introduces many new qualifiers, data types, and API calls for processing on the GPU. With CUDA programming in C/C++, a set of new qualifiers for functions determines whether

the codes will run on the device (GPU) or the host (CPU). Functions that run on the GPU are called kernel functions. Usually, they are invoked by the CPU and run on the GPU, but can also be invoked from the GPU itself. Coding for CUDA is unique because we need to explicitly define which type of memory each operation will store information, and use special code to transfer the information required to perform each operation since the high speed "shared" memory is small and do no communicate directly across different parts of the GPU. These transfers are usually slow, so they have to be done in such a way that they do not outweigh performance gains. To get the most efficiency out of the program, CUDA requires an understanding of the low-level GPU architecture. Some important architectural details that impact the performance of a program include proper allocation of both global and shared memory, where shared memory can only be accessed by threads inside a block but have significantly lower access times. Thread warps are another factor we need to keep in mind when we program for CUDA as all threads under a warp execute the same instruction. Other concepts that can make a large difference in efficiency are newer features recently added to GPUs, like tensor cores, which are much faster than previous methods for matrix calculations on specifically sized matrices. The programmer must also keep in mind not to excessively transfer data from the system memory to the GPU memory due to limitations in PCI-E protocol signaling. This is due to compatibility for older x32 CPUs allowing only the exposure of 256MB of GPU ram at a time to the CPU through the PCI-E Base Address Register (BAR) for memory access.

openMPI

OpenMPI is an API for message passing parallel programming originally built for distributed memory architectures, where one processor may not have fast access to all of the allocated memory, mainly if the memory belongs to another processor. OpenMPI serves as a way for multiple CPUs or servers, referred to as nodes, to run a single program together. By splitting data and assigning them individually to different nodes, with their own allocated memory, each node executes their assigned function and then returns the results to the master node. Through proper implementation of openMPI, a single program can split up tasks and send them to each server to process independently. As data needs to be explicitly sent and received from each server node, openMPI is much more complex to program when compared to openMP. This complexity is a result of the need to dynamically assign the data to each node, which has to be coded manually. OpenMPI nodes can be simulated for testing on a single system during development by splitting each thread in a CPU as individual nodes. By utilizing openMPI on a single system, often can outperform openMP in compute time for eliminating the need for CACHE coherence in exchange for more RAM to duplicate information across nodes. Using openMPI with openMP is one of the most common ways of parallelizing operations for large datasets. Although openMPI is a powerful tool, it has limitations when implementing some algorithms that require large master-slave communication because of the low transfer speeds across nodes. Thus if an algorithm would require data to be sent often from and to the nodes the transfer time may offset the benefits of parallel processing. In recent years as GPU computation has grown to be a powerhouse in computation, openMPI is used with CUDA for GPU compute nodes with increasing frequency.

Radon Transform

The Radon transform is an important processing tool commonly used for multiple and noise attenuation, and velocity analysis in seismic data. We utilize the Hyperbolic Radon Transform (HRT) as the basis for our deblending algorithm which we implement for this comparison. We can express the formula for HRT as (Thorson and Claerbout, 1985)

$$m(p,\tau) = \int_{h_1}^{h_2} d(h, t = \sqrt{\tau^2 + p^2 h^2}) dh. \tag{1}$$

Here $m(p,\tau)$ is the Radon space data, p is the slowness, τ is the two way travel time, h_1 is the lower offset limit, h_2 the upper offset limit, and d is the data space to be transformed. The slowness p is then defined as the inverse of velocity $1/v$. This operator focuses events into points following hyperbolic traveltimes according to a range of moveout velocities. The HRT has been used in many applications such as velocity analysis (Thorson and Claerbout, 1985), multiple suppression (Foster and Mosher, 1992; Guitton, 2000), interpolation in the time domain (Sacchi and Ulrych, 1995; Trad et al., 2002), high speed formulation for hyperbolic Radon (Huo et al., 2012), and deblending via denoising (Ibrahim and Sacchi, 2015).

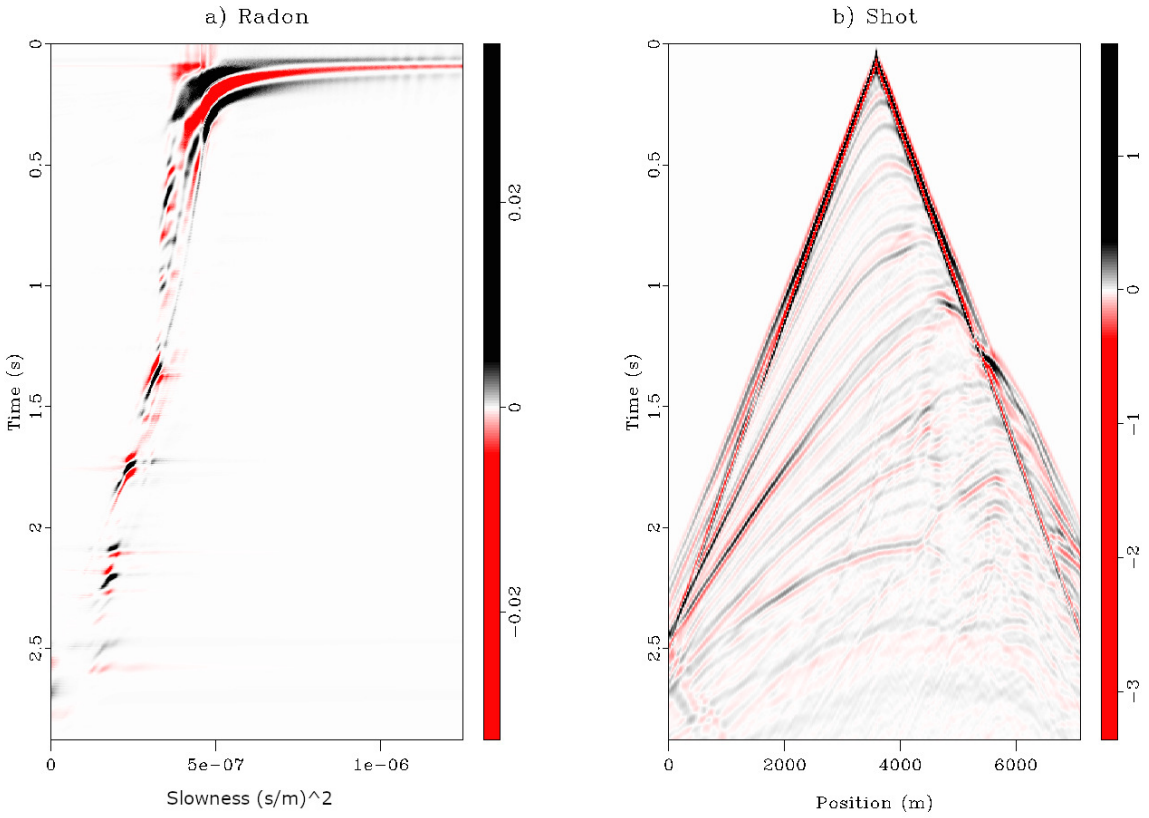

FIG. 2. Sparse Radon transform in common midpoint domain: Transformed Radon domain in a, shot domain data in b.

The sparse hyperbolic Radon transform is used in our deblending algorithm to re-format

the ill-posed blending operator to an inversion problem that uses sparse regularization. This Radon operator allows us to benchmark the performance of a highly parallelizable operation using each of the parallelization methods. To properly code the HRT 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

function RADON
  #PRAGMA OMP PARALLEL FOR ▷ insert for parallelization of loop
  for q = slowness **do**
    for h = offset **do**
      $moveout = h^2 * q$
      for it = 0 to nt **do**
        $time = \sqrt{(it * dt)^2 + moveout}$
        $model[iqNt + it] += data[ihNt + INT(time/dt)]$
      end for
    end for
  end for
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

__global__
function CUDA_ RADON
  int it = blockIdx.x * blockDim.x + threadIdx.x
  int iq = blockIdx.y * blockDim.y + threadIdx.y
  int ih = blockIdx.z * blockDim.z + threadIdx.z
  if (iq >= nq || ih >= nh || it >= nt) return
  int ihNt = ih*nt
  int iqNt = iq*nt
  double timemax = dt*nt
  double $moveout = h^2 * q$
  double $time = \sqrt{(it * dt)^2 + moveout}$
  $atomicAdd(\&model[iqNt + it], data[ihNt + INT(time/dt)])$
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.

Least squares deblending

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).

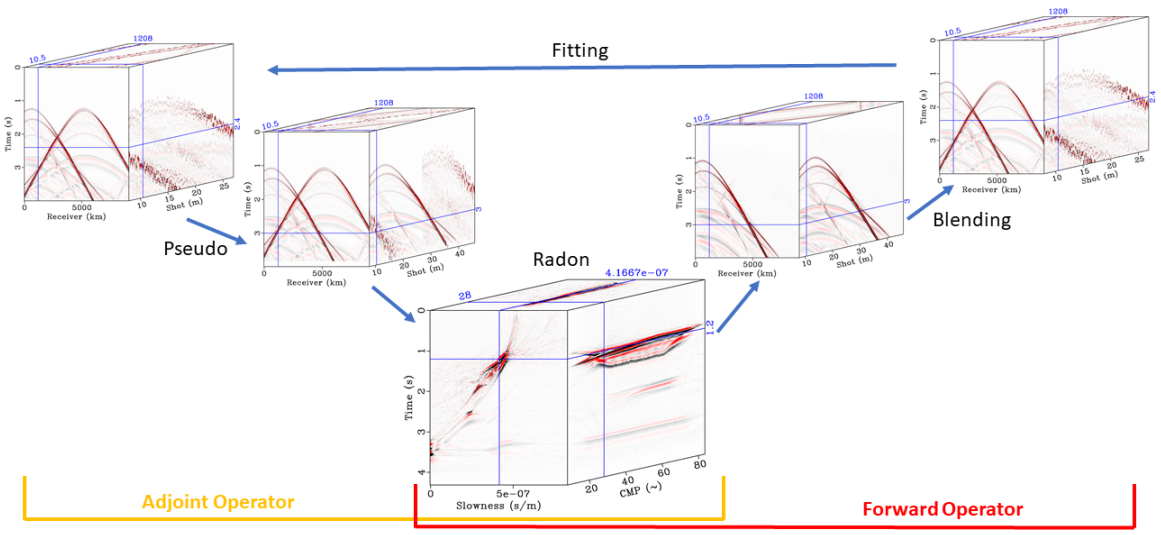

FIG. 3. Least squares deblending process: least-squares deblending as the fitting of a combined Radon and blending operator.

The test dataset used is the Marmousi dataset which can be seen in figure 4 for the least-squares deblending, the data is sorted into the CMP domain after the blending operator which is then sent to the Radon operator. By sorting into the CMP domain we can account

for the multiple apexes in a standard shot record without adding complexity to the Radon operator.

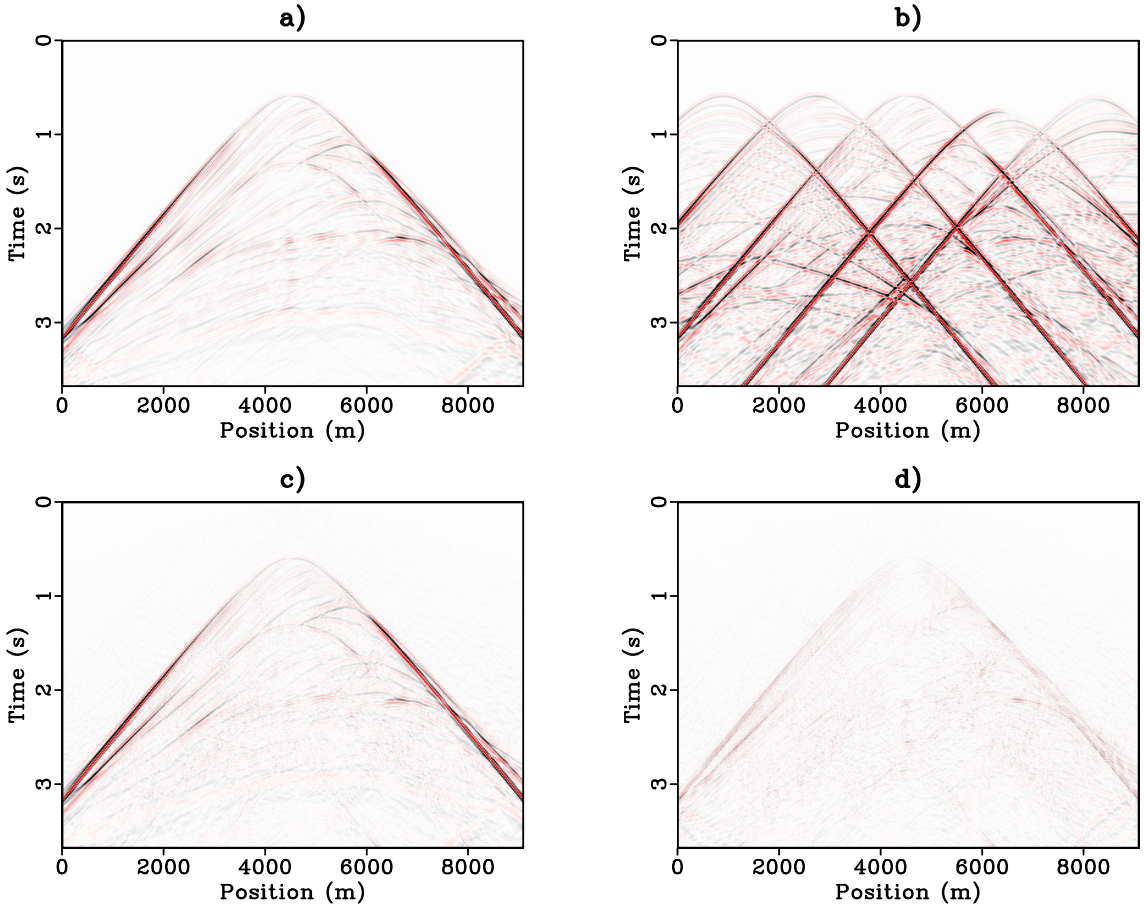

FIG. 4. Least squares deblending: least-squares deblending of the marmousi dataset with unblended in a, blended in b, result in c and difference in d.

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. Since distributed memory in clusters requires to split the data set across nodes instead of keeping it all in the master RAM, the resorting process imposes a very difficult challenge for an openMPI implementation. Due to this requirement, each node in a cluster would need to send data back after every calculation to calculate the next step, and then the master will need to resend the results back to each node. For our implementation in CUDA instead of using multiple nodes, we utilize a single high-performance machine, where both the Radon the least-squares descent algorithm was rewritten using CUDA kernels.

RESULTS

We ran the test algorithms on a range of different hardware to show the performance of the algorithms in difference conditions, a laptop, two high end workstations as well as our inhouse server cluster. The laptop utilizes an intel i7 8750H 6-core CPU with a Nvidia RTX 2060 MAX-Q gpu, the first workstation is equipped with an AMD Threadripper 3960x 24-core CPU, with a Nvidia GTX 1080, the second workstation has an intel i7 9800x 8-core cpu equipped with a Nvidia RTX 2060-super gpu. These results can be seen in table 1.

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 should be noted that although the RTX 2060 series GPUs are newer they have a lower CUDA core count ($\approx 75\%$) with respect to the GTX 1080 which results in the difference in computational times seen in the table. 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 openMPI 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.

CONCLUSION

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 counterparts. 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.

ACKNOWLEDGMENTS

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 Mosher, 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**.