HPC16 - Speeding up the 2+2+1 method via GPU computing for estimating local travel time operators in nonlinear beamforming

Y. Sun[1], I. Silvestrov[2], A. Bakulin[2]

[1] Aramco Research Center; [2] Saudi Aramco

Summary

Nonlinear beamforming is an effective method to enhance the quality of noisy seismic data. It uses local traveltime operators to describe local wavefronts, and then stack neighboring traces guided by these operators to enhance data quality. The 2+2+1 method is a pragmatic solver for estimating local traveltime operators from input data, but its calculation efficiency is not satisfying when the solution space is big. We speed up the 2+2+1 method using graphics processing unit (GPU) computing with the Compute Unified Device Architecture (CUDA) programming language. We introduce our GPU-based 2+2+1 algorithm, and demonstrate its efficiency improvement using a field data example. A speed-up factor of ~10 is obtained compared to the CPU version of the 2+2+1 method.

Introduction

Nonlinear beamforming (NLBF) is a recently proposed technology for enhancing 3D seismic data quality (Bakulin et al., 2020; Sun et al., 2022a). NLBF first estimates local traveltime operators directly from data, and later stacks every trace in the input data with its neighboring traces along wavefronts determined by local traveltime operators. To some extent, NLBF is similar to many other operator-based technologies, including the Common Focus Point technology (Sun et al., 2014; Sun and Verschuur, 2014), the Common Reflection Surface technology (Jäger et al., 2001) and the multifocusing technology (Berkovitch et al., 2011), but NLBF stands out as due to its versatility: it can deal with data in different domains and it can also work with NMO-corrected data.

The key to the success of NLBF is to efficiently estimate local traveltime operators from data, as this usually has to solve millions or even billions of nonlinear optimization problems per input gather (Sun et al., 2022a). The 2+2+1 method is a pragmatic solver for estimating local traveltime operators (Bakulin et al., 2018), but its efficiency in general is not satisfying enough when the solution space becomes large.

GPU (graphics computing unit) computing via the CUDA (Compute Unified Device Architecture) programming language has become an important high-performance computing tool (NVIDIA, 2020), and nowadays more and more supercomputers are powered by GPUs, for example the Dammam-7 supercomputer (Moss, 2021). We hereby introduce our GPU-assisted 2+2+1 method, yielding a speed-up factor of ~10 over our previous 2+2+1 method running on the CPU platform.

The 2+2+1 method

Due to the space limitation, in this paper, we can only briefly introduce the 2+2+1 method in NLBF. For comprehensive details, please refer to Sun et al. (2022b).

The local traveltime operators in NLBF are defined by a second-order mathematical equation:

$$\begin{aligned}\Delta t(x, y; x_0, y_0) &= t(x, y) - t(x_0, y_0)\\ &= A \cdot (x - x_0) + B \cdot (y - y_0) + C \cdot (x - x_0) \cdot (y - y_0) + D \cdot (x - x_0)^2 + E \cdot (y - y_0)^2 ,\end{aligned} \qquad (1)$$

where $t(x, y)$ is the traveltime of the trace at (x, y), $t(x_0, y_0)$ is the traveltime of the NLBF parameter trace at (x_0, y_0), and $\{A, B, C, D, E\}$ are the unknown coefficients for a local traveltime operator at $t(x_0, y_0)$. To estimate these unknown coefficients $\{A, B, C, D, E\}$, we optimize a semblance-based cost function:

$$S(x_0, y_0) = \frac{\sum_{j=1}^{N}\{\sum_{i=1}^{M} u[x_i, y_i; t_j(x_0, y_0) + \Delta t(x_i, y_i; x_0, y_0)]\}^2}{M\sum_{j=1}^{N}\sum_{i=1}^{M}\{u[x_i, y_i; t_j(x_0, y_0) + \Delta t(x_i, y_i; x_0, y_0)]\}^2} , \qquad (2)$$

where $u(x_i, y_i; t)$ represents a time sample of the trace at (x_i, y_i) in an input gather, M is the total number of traces inside the spatial aperture of the local traveltime operator, and N is the total number of time samples within the temporal aperture of the local traveltime operator.

The 2+2+1 method is a local search method, and it estimates $\{A, B, C, D, E\}$ in three steps: by setting $\{B, C, E\}$ to 0s, it first finds the optimal values of $\{A, D\}$ using a subset of data extracted along the y direction via a brute-force search; next and likewise, it finds the optimal values of $\{B, E\}$ via brute-force searching a data subset extracted along the x direction by setting $\{A, C, D\}$ to 0s; finally, it finds

the optimal value of $\{C\}$ via another brute-force search using the complete dataset with $\{A, B, D, E\}$ fixed at their already estimated values. As the 2+2+1 method requires three brute-force searches, when the solution space is large, the computation efficiency is still a challenge.

The 2+2+1 CPU / GPU algorithm

Figure 1 presents our 2+2+1 method implemented on both the CPU platform and the GPU platform. The CPU algorithm, shown in the left picture of Figure 1, is a straightforward implementation of the 2+2+1 method, and the foreach loop in green is parallelized via OpenMP directives for efficiency.

```
Pseudocode for the 2+2+1 CPU Algorithm

    Input: a seismic dataset, user-specified parameters to define the solution space for {A, B, C, D, E}
    Output: {A, B, C, D, E, S} for each gather in the input dataset, where {S} is the semblance value
            corresponding to the parameters {A, B, C, D, E}
1   Use user-specified parameters to set search schemes for {A, D}, {B, E} and {C}
2   Set different apertures (shown in Figure 1) for {A, D}, {B, E} and {C}
3   foreach gather_i ∈ the input seismic dataset do
4       Set locations of NLBF parameter traces for this gather;
5       foreach y_i ∈ y direction of NLBF parameter traces do
6           foreach x_i ∈ x direction of NLBF parameter traces do
7               Collect traces falling into the aperture of {C};
8               Collect traces falling into the aperture of {A, D};
9               Collect traces falling into the aperture of {B, E};
10              foreach t_i ∈ time direction of NLBF parameter traces do
11                  Set {B, C, E} to 0s and scan {A, D} in their solution space using traces in
                        its aperture (Figure 1(a)) to pick out the combination that
                        maximizes equation (2);
12                  Set {A, C, D} to 0s and scan {B, E} in their solution space using traces in
                        its aperture (Figure 1(b)) to pick out the combination that
                        maximizes equation (2);
13                  Fix {A, B, D, E} to the estimated values, and scan {C} in its solution space
                        using traces in its aperture (Figure 1(c)) to pick out the value that
                        maximizes the semblance value S defined by equation (2);
14                  Save these estimated {A, B, C, D, E, S} at (x_i, y_i, t_i);
15              end
16          end
17      end
18  end
19  Output {A, B, C, D, E, S} of all NLBF parameter traces.
```

```
Pseudocode for the 2+2+1 GPU Algorithm

    Input: a seismic dataset, user-specified parameters to define the solution space for {A, B, C, D, E},
           heap size and stream amount for the GPU
    Output: {A, B, C, D, E, S} for each gather in the input dataset, where {S} is the semblance value
            corresponding to the parameters {A, B, C, D, E}
1   Use user-specified parameters to set search schemes for {A, D}, {B, E} and {C};
2   Set different apertures (shown in Figure 1) for {A, D}, {B, E} and {C};
3   Set the user-specified heap size on the GPU;
4   Adaptively build the block and grid for the CUDA calculation;
5   Create a pool of CUDA streams;
6   foreach gather_i ∈ the input seismic dataset do
7       Set locations of NLBF parameter traces for this gather;
8       foreach y_i ∈ y direction of NLBF parameter traces do
9           foreach x_i ∈ x direction of NLBF parameter traces do
10              Fetch an available CUDA stream from the stream pool;
11              Collect traces falling into the aperture of {A, D};
12              Asynchronously copy these traces in the aperture of {A, D} to the GPU, with the
                    fetched stream as the stream identifier;
13              Collect traces falling into the aperture of {B, E};
14              Asynchronously copy these traces in the aperture of {B, E} to the GPU, with the
                    fetched stream as the stream identifier;
15              Collect traces falling into the aperture of {C};
16              Asynchronously copy these traces in the aperture of {C} to the GPU, with the
                    fetched stream as the stream identifier;
17              Run the 2+2+1 CUDA kernel to calculate {A, B, C, D, E, S} at (x_i, y_i),with the
                    fetched stream as the stream identifier;
18              Asynchronously copy {A, B, C, D, E, S} at (x_i, y_i) from the GPU to the memory;
19          end
20      end
21  end
22  Output {A, B, C, D, E, S} of all NLBF parameter traces.
```

Figure 1: Pseudo code for the 2+2+1 CPU algorithm (left) and the 2+2+1 GPU algorithm (right).

```
Pseudocode for the CUDA kernel in the 2+2+1 GPU algorithm

    Input: solution dimensions N_A, N_B, N_C, N_D and N_E of parameters A, B, C, D and E; traces in the
           apertures of {A, D}, {B, E} and {C}, respectively
    Output: none
1   __shared__ float a_best, b_best, c_best, d_best, e_best, s_best;
2   if threadIdx.x == 0 do
3       Allocate memories for several float arrays: semblance[blockDim.x], a[blockDim.x],
            b[blockDim.x], c[blockDim.x], d[blockDim.x], e[blockDim.x];
4   end
5   __syncthreads();
6   semblance[threadIdx.x] = 0.0;
7   idx = threadIdx.x;
8   while idx < N_A·N_D do
9       Get a_thread and d_thread corresponding to this idx;
10      Evaluate equation (2) using { a_thread, 0, 0, d_thread, 0} for semb with the traces in the aperture of {A,
            D};
11      if semblance[threadIdx.x] < semb do
12          semblance[threadIdx.x] = semb;
13          a[threadIdx.x] = a_thread;
14          d[threadIdx.x] = d_thread;
15      end
16      idx = idx − blockDim.x;
17  end
18  __syncthreads();
19  if threadIdx.x == 0 do
20      Pick out a_best and d_best corresponding to the best value in semblance[blockDim.x];
21  end
22  __syncthreads();
23  semblance[threadIdx.x] = 0.0;
24  idx = threadIdx.x;
25  while idx < N_B·N_E do
26      Get b_thread and e_thread corresponding to this idx;
27      Evaluate equation (2) using { 0, b_thread, 0, 0, e_thread } for semb with the traces in the aperture of {B,
            E};
28      if semblance[threadIdx.x] < semb do
29          semblance[threadIdx.x] = semb;
30          b[threadIdx.x] = b_thread;
31          e[threadIdx.x] = e_thread;
32      end
33      idx = idx + blockDim.x;
34  end
35  __syncthreads();
36  if threadIdx.x == 0 do
37      Pick out b_best and e_best corresponding to the best value in semblance[blockDim.x];
38  end
39  __syncthreads();
40  semblance[threadIdx.x] = 0.0;
41  idx = threadIdx.x;
42  while idx < N_C do
43      Get c_thread corresponding to this idx;
44      Evaluate equation (2) using {a_best, b_best, c_thread, d_best, e_best} for semb with the traces in the aperture of
            {C};
45      if semblance[threadIdx.x] < semb do
46          semblance[threadIdx.x] = semb;
47          c[threadIdx.x] = c_thread;
48      end
49      idx = idx − blockDim.x;
50  end
51  __syncthreads();
52  if threadIdx.x == 0 do
53      Pick out c_best corresponding to s_best, which is the best value in semblance[blockDim.x];
54      Pass { a_best, b_best, c_best, d_best, e_best, s_best } to the global memory;
55  end
56  __syncthreads();
57  Return.
```

Figure 2: Pseudo code for the CUDA kernel in the 2+2+1 GPU algorithm.

Our GPU algorithm for the 2+2+1 method is designed to make the best usage of GPUs' high instruction throughput, and the right picture in Figure 1 shows its pseudo code. Our GPU algorithm follows a hierarchic design:

- As data has to be first organized and transferred from memory to GPU, we overlap data organization with data transfer via the help of CUDA streams.
- Since GPUs are designed as a natural vector-calculation machine, we unwrap the foreach loops in the CPU version via adaptively building thread blocks and block grids. One thread block is used

to deal with a local traveltime operator at a single time window of an NLBF parameter trace. Since the time dimension of a seismic trace is definitely smaller than the maximum x dimension of a block grid, which is 2^{31}-1 (NVIDIA, 2020), we just set the block grid to be $(N_t, 1, 1)$, where N_t is the time dimension of an NLBF parameter trace. One thread in each thread block evaluates one parameter combination. Since the maximum number of threads in a thread block can only be 1024 (NVIDIA, 2020) but the amount of parameter combinations is unpredictable, we adaptively build the thread-block size $thread_{max}$: first take the maximum solution dimension as its trial solution, $thread_{max}=max(N_A{\cdot}N_D,\ N_B{\cdot}N_E,\ N_C)$, where N_A, N_B, N_C, N_D, and N_E are the solution dimensions of parameters $\{A, B, C, D, E\}$; next, round $thread_{max}$ to the least integer multiple of the warp size, which is 32 by definition (NVIDIA, 2020); finally, take the larger value between the current $thread_{max}$ and the maximum thread amount in a block as the final $thread_{max}$, i.e., $thread_{max}=max(thread_{max},\ 1024)$.

- The CUDA kernel directly estimates parameters $\{A, B, C, D, E\}$ for one local traveltime operator at a single time window of an NLBF parameter trace, and its detailed pseudo code is shown in Figure 2.

Example

We use a 3D single-sensor field dataset acquired from a desert environment (Sun et al., 2022a) to demonstrate the efficiency improvement of the 2+2+1 GPU algorithm over the 2+2+1 CPU algorithm. The test environment for the 2+2+1 CPU algorithm comprises 2 Xeon Gold 6136 CPUs (3.00 GHz, 12 cores), and all 24 cores are engaged in the calculation via OpenMP directives. The environment for the 2+2+1 GPU algorithm comprises 1 NVIDIA Tesla V100 GPU with 32 GB onboard memory, and only 1 core of a Xeon Gold 6142 CPU (2.60 GHz, 16 cores) is engaged. The operation system for the CPU (GPU) test is Red Hat 7.x (CentOS 7.x). Both algorithms are implemented by C++, and the compilation environments are: for the CPU version, Intel Parallel Studio XE 2017 Update 2 is used; for the GPU version, CUDA 9.0 Update 4 along with Intel Parallel Studio XE Update 4 are used.

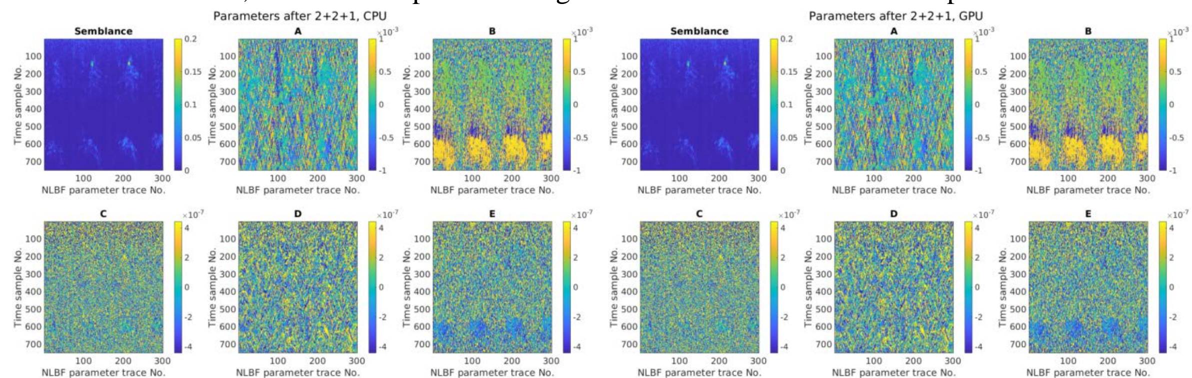

Figure 3: (Left/Right) Semblance values and parameters {A, B, C, D, E} for some selected parameter traces in the field data example calculated by the 2+2+1 CPU/GPU algorithm.

Several receiver lines of the raw data are shown in the left picture of Figure 4. This single-sensor dataset is highly contaminated by the near-surface scattering noise, and hence no signals can be easily recognized. In our calculation, the spacing between NLBF parameter traces is 50 m in both x and y direction, and the relevant apertures for different parameters are: 300 m by 35 m for {A, D}, 35 m by 300 m for {B, E} and 300 m by 300 m for {C}. The concrete search schemes for different parameters are [-10^{-3} s/m : $1.33{*}10^{-5}$ s/m : 10^{-3} s/m] for {A, B} and [$-4.44{*}10^{-7}$ s/m^2 : $4.44{*}10^{-8}$ s/m^2: $4.44{*}10^{-7}$ s/m^2] for {C, D, E}. The estimated semblance values and parameters {A, B, C, D, E} for some selected NLBF parameter traces are shown in Figure 3, and clearly we can observe that they are visually identical although they differ slightly at the floating-number level as they are calculated in different hardware and software environments. In terms of calculation time, the 2+2+1 GPU version takes about 147.39 s while the 2+2+1 CPU version takes about 1639.94 s. So on this test example, the GPU algorithm yields a speed-up factor of 11.1 over the CPU algorithm. The middle and right

pictures in Figure 4 show the correspondingly enhanced receiver lines by applying these estimated NLBF operators. Similar to Figure 3, we can see that these results are visually identical although they differ a little bit at the floating-number level.

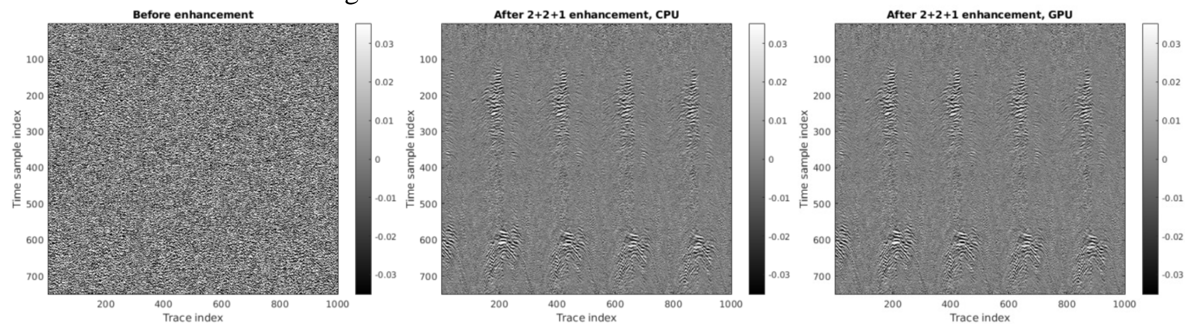

Figure 4: (Left) Several receiver lines of the raw data. (Middle/Right) The corresponding enhanced data by NLBF with the 2+2+1 CPU/GPU algorithm.

Conclusions

GPU computing is an effective way to improve the calculation efficiency of the 2+2+1 method to estimate local traveltime operators, and this will benefit the geophysics community to use the nonlinear beamforming technology for enhancing low-quality field data at a much larger scale.

Acknowledgements

For computer time, this research used the resources of the Supercomputing Laboratory at King Abdullah University of Science & Technology (KAUST) in Thuwal, Saudi Arabia.

References

Bakulin, A., Silvestrov, I., Dmitriev, M., Neklyudov, D., Protasov, M., Gadylshin, K., and Dolgov, V. [2018] Nonlinear beamforming for enhancing prestack seismic data with a challenging near surface or overburden. First Break, **36** (12), 121-126.
Bakulin, A., Silvestrov, I., Dmitriev, M., Neklyudov, D., Protasov, M., Gadylshin, K., and Dolgov, V. [2020] Nonlinear beamforming for enhancement of 3D prestack land seismic data. Geophysics, **85** (3), V283-V296.
Berkovitch, A., Deev, K. and Landa, E. [2011] How nonhyperbolic multifocusing improves depth imaging. First Break, **29** (9), 95-103.
Jäger, R., Maan, J., Höcht, G. and Hubral, P. [2001] Common-reflection-surface stack: Images and attributes. Geophysics, **66** (1), 97-109.
NVIDIA [2020] CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/archive/11.1.0/cuda-c-programming-guide/index.html
Moss, S. [2021] Aramco and stc launch 55.4 petaflops Cray supercomputer, Dammam 7. https://www.datacenterdynamics.com/en/news/aramco-and-stc-launch-554-petaflops-cray-supercomputer-dammam-7/.
Sun, Y., Verschuur, E. and Vrolijk, J. W. [2014] Solving the complex near-surface problem using 3D data-driven near-surface layer replacement. Geophysical Prospecting, **62** (3), 491-506.
Sun, Y. and D. J. Verschuur. [2014] A Self-Adjustable Input Genetic Algorithm for the Near-Surface Problem in Geophysics. IEEE Transactions on Evolutionary Computation, **18** (3), 309-325.
Sun, Y., Silvestrov, I., and Bakulin, A. [2022a] Enhancing 3D land seismic data using nonlinear beamforming based on the efficiency-improved Genetic Algorithm. IEEE Transactions on Evolutionary Computation, doi: 10.1109/TEVC.2022.3149579.
Sun, Y., Silvestrov, I., and Bakulin, A. [2022b] Accelerating the 2+2+1 method for estimating local traveltime operators in nonlinear beamforming using GPU graphics cards. Journal of Geophysics and Engineering, accepted.