applied sciences

MDPI

Article

Parallel Dislocation Model Implementation for Earthquake Source Parameter Estimation on Multi-Threaded GPU

Seongjae Lee [1,2] and Taehyoun Kim [1,2,*]

[1] Department of Mechanical and Information Engineering, University of Seoul, Seoul 02504, Korea; seongjae.lee.1118@gmail.com
[2] Department of Smart Cities, University of Seoul, Seoul 02504, Korea
* Correspondence: thkim@uos.ac.kr; Tel.: +82-2-6490-2388

Abstract: Graphics processing units (GPUs) have been in the spotlight in various fields because they can process a massive amount of computation at a relatively low price. This research proposes a performance acceleration framework applied to Monte Carlo method-based earthquake source parameter estimation using multi-threaded compute unified device architecture (CUDA) GPU. The Monte Carlo method takes an exhaustive computational burden because iterative nonlinear optimization is performed more than 1000 times. To alleviate this problem, we parallelize the rectangular dislocation model, i.e., the Okada model, since the model consists of independent point-wise computations and takes up most of the time in the nonlinear optimization. Adjusting the degree of common subexpression elimination, thread block size, and constant caching, we obtained the best CUDA optimization configuration that achieves 134.94×, 14.00×, and 2.99× speedups over sequential CPU, 16-threads CPU, and baseline CUDA GPU implementation from the 1000 × 1000 mesh size, respectively. Then, we evaluated the performance and correctness of four different line search algorithms for the limited memory Broyden–Fletcher–Goldfarb–Shanno with boundaries (L-BFGS-B) optimization in the real earthquake dataset. The results demonstrated Armijo line search to be the most efficient one among the algorithms. The visualization results with the best-fit parameters finally derived by the proposed framework confirm that our framework also approximates the earthquake source parameters with an excellent agreement with the geodetic data, i.e., at most 0.5 cm root-mean-square-error (RMSE) of residual displacement.

Keywords: GPU; CUDA; nonlinear optimization; line search algorithm; remote sensing; Monte Carlo method; earthquake source parameter estimation

Citation: Lee, S.; Kim, T. Parallel Dislocation Model Implementation for Earthquake Source Parameter Estimation on Multi-Threaded GPU. *Appl. Sci.* **2021**, *11*, 9434. https://doi.org/10.3390/app11209434

Academic Editor: Jianbo Gao

Received: 11 August 2021
Accepted: 2 October 2021
Published: 11 October 2021

1. Introduction

The Monte Carlo method has widely been used for geophysical source parameter estimation in remote sensing [1–3]. The method repeats nonlinear optimization from multiple random starting points to derive the best-fit source parameters that minimize an objective function indicating the misfit between geodetic measurement and the dislocation model. Therefore, it enables avoiding local minima and evaluating the uncertainties of parameters [4].

Interferometric synthetic radar (InSAR) has been one of the most common methods for surface displacement measurement. It compares the phase offset of two or more complex-valued SAR images obtained from different times and locations [5]. InSAR can derive precise geophysical information with centimetric or millimetric accuracy. For this reason, InSAR data have been widely employed in remote sensing, including earthquakes, volcanic activities, landslides [6], thaw-derived slope failure [7], or glacial ice movement [8] investigations.

The dislocation model mathematically calculates the surface deformation of earthquakes or volcanic activities. The rectangular dislocation model, also known as the Okada

model, is a popular model for earthquake source modeling, which assumes finite rectangular source and isotropic half-space [9,10]. Other models include the prolates spheroid model [11] and the spherical source model [12] for magma source modeling.

Most nonlinear optimization algorithms iteratively search local minima using gradient or Hessian approximation of the objective function. However, since the dislocation model computes three-dimensional (3D) surface displacement of points in a two-dimensional (2D) mesh, computation of the objective function leads to many floating-point operations proportional to the mesh size. Therefore, gradient or Hessian approximation of the objective function becomes too expensive. Furthermore, the nonlinear optimization is iterated more than 1000 times for the Monte Carlo method, and thus the Monte Carlo method takes an extreme computational burden.

In the recent decade, graphics processing units (GPUs) have been gaining attention as a powerful solution to overcome the performance limitations of conventional multi-core CPUs, especially for applications demanding a massive amount of computations. Furthermore, the release of NVIDIA's computed unified device architecture (CUDA) [13], a new general-purpose parallel programming interface, also facilitated the use of GPU platforms for general-purpose computing applications. Since the CUDA architecture can reorganize the computation around data when the data can be processed independently, we can execute the independent computation in parallel on a vast number of threads to improve the overall performance significantly. Therefore, the Okada model, the dislocation model discussed in this paper, fits well for the GPU implementation because the 3D displacement computation of each point needs a massive number of floating-point operations and has no dependencies between these operations.

This paper proposes a performance acceleration framework based on CUDA GPU for the earthquake source estimation with the Okada dislocation model. For this purpose, (1) we analyze the effects of various combinations from options of CUDA optimization techniques and, based on the analysis, perform the CUDA kernel optimization for the parallel implementation of the Okada model; (2) we investigate the performance and correctness of line search algorithms used for the nonlinear optimization to derive the earthquake source parameters in an efficient manner; (3) finally, we verify that the earthquake source parameters derived by our proposed approach also fit the geodetic data correctly by visualizing the residual data between the geodetic data and our modeled data.

Considering the target application characteristics, three techniques, common subexpression elimination, thread block size adjustment, and constant caching, are employed for the CUDA kernel optimization. Then, the combinations generated from the options of these optimization techniques are evaluated in terms of efficiency and occupancy. As a result, the configuration with the shortest average computation time is selected as our best CUDA optimization configuration. As for the comparison of line search algorithms, we employ the root-mean-square-error (RMSE), computation time, 95% confidence interval, and parameter distribution on the real earthquake dataset as the evaluation metrics. Finally, the source parameters with the best consistency between geodetic and modeled displacement are determined and validated.

The paper is organized as follows; Section 2 introduces the related work and background of this study. Then, we define the problem addressed and describe our proposed approach in Section 3. Next, Section 4 presents the experimental results and discussions of our approach. Finally, we conclude the paper in Section 5.

2. Background and Related Work

2.1. Earthquake Source Parameter Estimation

In remote sensing, the Monte Carlo method has widely been used for earthquake or volcanic source parameter estimation. To reduce the computation time, many researchers used techniques that randomly sample only a portion of the entire geodetic measurement, such as quadtree sampling [14], or defined bound constraints based on previous seismic studies conducted in the same study area. De Novellis et al. [15] estimated the source pa-

rameters and its uncertainties for the 2015 Wolf volcano using the Okada and Yang models and the Levenberg–Marquardt method. Studies by Funning et al. [16] and Qu et al. [17] used the downhill simplex optimization and the Okada model to estimate the source parameters of the 2003 Bam earthquake and the 2009 Yao'an earthquake, respectively. Dicelis et al. [18] estimated the source parameters of the 2008 Quetame earthquake using the nonlinear interior-point optimization and the Okada model.

Recently, studies estimating the source parameters using a statistical approach or machine learning have also been proposed. Bagnardi and Hooper [19] and Dutta et al. [20] used the Bayesian approach based on the Markov Chain Monte Carlo. Šílený [21] and Picozzi et al. [22] applied a genetic algorithm and artificial neural network for source parameter estimation. Lee and Kim [23] proposed a parameter search space reduction method based on the principal component analysis to accelerate nonlinear optimization.

2.2. Nonlinear Optimization

In general, nonlinear optimization algorithms are classified into trust-region methods or line search methods. Both of them approximate an objective function as a quadratic model, but the methods proceed in different ways. Trust-region methods define a region around the current step that the quadratic model is adequate and find the approximate minimizer in the region. However, line search methods first find the descent direction and focus on searching appropriate step length [24]. Since most source parameters have their own upper and lower bound constraints, we should employ an optimization algorithm handling constrained problems.

The Levenberg–Marquardt algorithm is a popular trust-region method [25]. The algorithm can be thought of as a combination of the steepest descent and the Gauss–Newton method. Using the damping parameter μ, it operates like the steepest descent when the current point is far from the correct solution. On the other hand, when the current point is close to the correct solution, it behaves like the Gauss–Newton method [24]. We can generally use the algorithm to solve unconstrained nonlinear optimization problems, but the algorithm for the constrained problem was also developed in [26]. Other trust-region methods include trust-region reflective [27] and Dogleg [28] methods.

In line search methods, we typically use the steepest descent, Newton's method, or the quasi-Newton methods to derive the descent direction. In the k-th step, let f_k and d_k be the objective function and the descent direction, respectively. In the steepest descent, the computation of the descent direction is very simple, calculated as $d_k = -\nabla f_k$. However, it slowly converges to the local solution. Newton's method finds the descent direction using the inverse of Hessian matrix as $d_k = -\nabla^2 f_k^{-1} \nabla f_k$. It converges to the local solution faster than the steepest descent, but computing the inverse of $\nabla^2 f_k$ is too expensive. As an alternative to Newton's method, the quasi-Newton method approximates the inverse of the Hessian matrix using only the gradient of the objective function. The Broyden–Fletcher–Goldfarb–Shanno (BFGS) method is a representative quasi-Newton method, but the BFGS method also suffers from high memory usage. To resolve this problem, the limited memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) method [29] was developed. The L-BFGS-B method, which is a variant of L-BFGS, was also proposed to handle simple bound constraints [30].

In this paper, we used the L-BFGS-B algorithm to handle bound constraints with a small memory footprint. Another advantage of using L-BFGS-B is that the user can adjust the computational cost and memory requirement by tuning the parameter m, which denotes the number of stored previous points and gradients.

In line search methods, computing the step length α is a key factor for the accuracy and performance of the algorithm. For the k-th iteration, let x_k, d_k, and α_k be the current point, descent direction, and step length, respectively. The ideal step length would be the global minimizer of the function $\phi(\cdot)$ defined by Equation (1). However, since it is too expensive to derive the global minimizer, there is a trade-off between the quality of a step length and the computation time [24].

$$\phi(\alpha_k) = f(x_k + \alpha_k d_k), \quad \alpha_k > 0 \tag{1}$$

Typical line search algorithms perform a sequential search until predefined conditions of $\phi(\cdot)$ are satisfied. A popular condition is the sufficient decrease condition, described in Equation (2) for some constant $0 < c_1 < 1$.

$$f(x_k + \alpha_k d_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^T d_k \tag{2}$$

This condition stipulates that α_k should give a sufficient decrease in the objective function. Since the sufficient decrease condition is not enough to rule out unacceptably short step lengths, we normally can use the curvature condition, defined by Equation (3) for $0 < c_1 < c_2 < 1$. Both the sufficient decrease condition and curvature condition are collectively known as the Wolfe conditions or the weak-Wolfe conditions [31]. In addition, we can modify the curvature condition to find a broad neighborhood of a local minimizer or stationary point of $\phi(\cdot)$, as described in Equation (4). The conditions satisfying both Equations (2) and (4) are called the strong-Wolfe conditions.

$$\nabla f(x_k + \alpha_k d_k)^T d_k \geq c_2 \nabla f_k^T d_k \tag{3}$$

$$|\nabla f(x_k + \alpha_k d_k)^T d_k| \geq c_2 |\nabla f_k^T d_k| \tag{4}$$

There are various algorithms for line search. Armijo line search (Armijo) [32], bisection method for weak-Wolfe conditions (BWW), and Móre-Thuente line search (MT) [33] find a step length that satisfies the sufficient decrease, weak-Wolfe, and strong-Wolfe conditions, respectively. All three methods use iterative steps to find the appropriate step length. BWW and MT need to recalculate both the objective function f and its gradient g for every iteration. However, Armijo requires only a new calculation of the objective function f.

2.3. GPU and CUDA

Unlike CPUs optimized for sequential performance by using sophisticated control logic and large cache memories, GPUs consist of a massive number of threads with small cache memories to achieve high throughput [34]. CUDA is a general-purpose parallel computing platform and programming model for the NVIDIA GPUs. CUDA is designed to develop scalable parallel applications while maintaining a low learning curve for developers [35]. The architecture also provides flexibility in the assignment of local resources to enhance performance, but this flexibility induces developers to perform hand-crafted optimization [36].

Figure 1 shows the general compilation and execution flow of CUDA C/C++ program [34]. As illustrated in Figure 1a, a single source program is first divided into the host, i.e., CPU, and device, i.e., GPU, code. The host code is standard C/C++ code, and the device code is written in data-parallel functions called kernels, specified by CUDA keywords. A general C/C++ compiler compiles the host code, and the device code is first converted into parallel thread execution (PTX) intermediate code that is the CUDA's instruction set architecture and then compiled to binary [37]. Figure 1b shows the execution and memory transfer of the CUDA program at runtime. The program starts with the host, and when the host launches the kernel, it is executed by thousands of threads on a device.

Thread scheduling is essential for enhancing parallel performance. CUDA runtime system organizes threads in a two-level hierarchy. Threads are grouped into thread blocks, and thread blocks form a grid. When a kernel function is launched, each thread block is assigned to a streaming multiprocessor (SM) during its execution, and each block is divided into warps of 32 threads. The warp is a thread scheduling unit in SMs and executes in single instruction, multiple data (SIMD) fashion. SMs perform zero-overhead thread scheduling that can interleave warps or select ready warps with no additional cost or time. It can also hide the latency of global memory access or long-latency instructions [38].

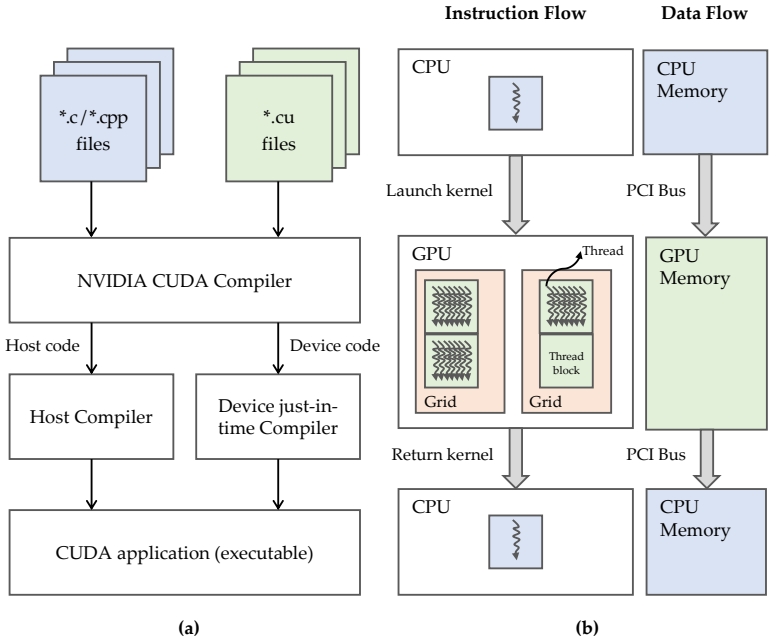

Figure 1. (**a**) Compilation process of CUDA C/C++ sources; (**b**) CUDA instructions and data flow at runtime.

Memory bandwidth optimization is also significant to boost the execution efficiency of CUDA kernels because the data to be processed by GPU should be transferred from the host memory to the device's global memory, as shown in Figure 1b. Since the global memory is implemented with dynamic random access memory (DRAM), applications can saturate the memory bandwidth easily. CUDA provides several programmable on-chip memories to reduce the demand for the global memory bandwidth of the applications. Table 1 summarizes the types of CUDA memory. Shared memory is on-chip memory with faster access speed than global and local memory, and variables in shared memory space are shared among threads in the same thread block. Therefore, we can eliminate the global memory access bottleneck by using shared memory as a software-managed cache. For read-only data referenced by many threads simultaneously, constant memory can help reduce the memory access latency via automatic caching.

Table 1. Properties of CUDA memories.

Name	Location	Scope	Speed
Register	On-chip	Thread	Extremely fast
Constant memory	Off-chip (not-cached), on-chip (cached)	GPU	Slow (not-cached), fast (cached)
Shared memory	On-chip	Thread block	Fast
Local memory	Off-chip	Thread	Slow
Global memory	Off-chip	GPU	Slow

GPUs are being actively used in remote sensing. Representatively, several efforts for the incorporation of GPU to hyperspectral image processing have been directed. Since the computational cost of hyperspectral image processing is expensive due to the high dimensionality of the image, GPUs can be a powerful solution for the processing, such as unmixing or dimensionality reduction [39,40]. In seismic studies, there was an implementation of double-difference seismic tomography with GPU to improve performance [41]. Venetis et al. [42] showed that GPU could significantly reduce the execution time of the grid search algorithm used in earthquake source parameter estimation [42].

3. Our Proposed Approach

This paper aims to accelerate earthquake source parameter estimation based on the Okada dislocation model by improving the most time-consuming parts in two folds, as highlighted in Figure 2.

First, we optimized the parallel implementation of the Okada model to enhance the model performance. We chose the CUDA optimization techniques regarding the computational characteristics of the Okada model and set 32 combinations from different options of the techniques. Each configuration was evaluated in terms of efficiency and occupancy to derive the effects of the optimization techniques on the computation time, and we selected a configuration with the shortest average computation time as the best CUDA optimization configuration. Second, in the earthquake source parameter estimation, we assessed the performance and correctness of line search algorithms, Armijo, MT, and BWW, for the L-BFGS-B optimization. We employed the misfit, the computation time, and the distribution of estimated source parameters for the performance assessment. In addition, the parallel implementation of the L-BFGS-B (cuLBFGSB) [43] was also evaluated. Finally, we derived the best-fit source parameters and visualized the modeled result to verify our approach.

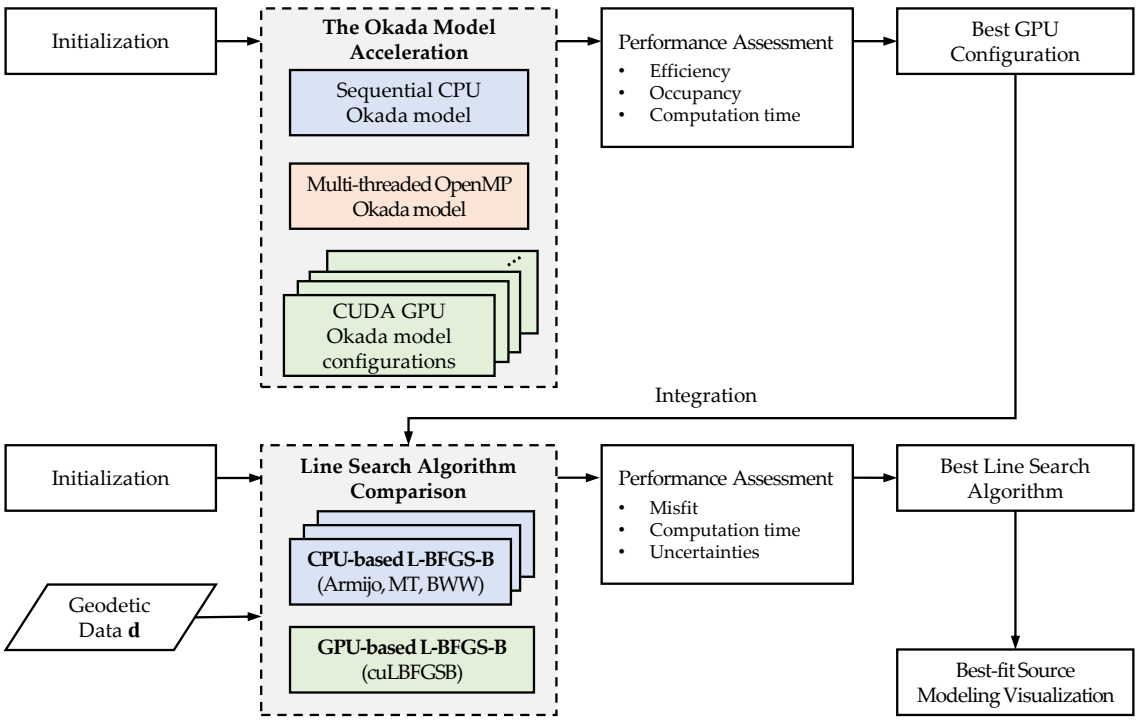

Figure 2. Schematic workflow of our approach.

3.1. Dislocation Model and Dataset

For the forward modeling of the earthquake, we chose the Okada model that assumes a finite rectangular source and isotropic homogeneous half-space. Table 2 describes the physical source parameters that define a rectangular source, and Figure 3 shows the fault geometry of the Okada model, respectively. We assumed a Poisson's ratio ν of 0.23. The parameter E indicates the horizontal, i.e., E–W direction, distance of the upper-left point of the satellite image and fault, and N indicates the vertical one. The upper-left point of the measured deformation map is located at 36°10′57″ N, 129°17′03″ E.

Table 2. Source parameters of the Okada model.

Parameter	Unit	Description
E	km	Distance from the reference point to the east
N	km	Distance from the reference point to the north
Depth	km	Depth of source
Strike	degrees	Angle of fault relative north
Dip	degrees	Angle between the fault and a horizontal plane
Length	km	Length of fault
Width	km	Width of fault
Rake	degrees	Angle of slip relative to the width direction
Slip	cm	Dislocation in rake direction
Open	cm	Dislocation in tensile component

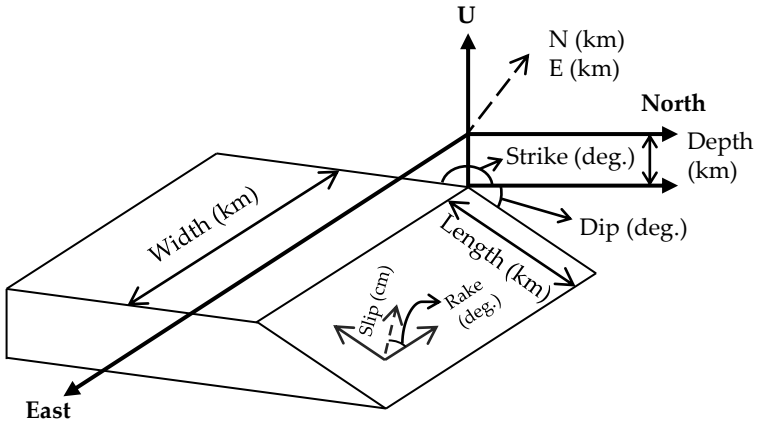

Figure 3. Fault geometry of the Okada model [23].

Since the source parameter estimation procedure needs a 3D surface displacement dataset, InSAR techniques for retrieving 3D displacement have been introduced [44]. Many studies used the differential InSAR technique that combines line-of-sight (LOS) SAR images from ascending and descending paths [15–17]. However, since this method suffers from low accuracy of north component displacement, stacking InSAR and multiple aperture interferometry (MAI) method [45] was proposed to solve this problem.

In this paper, we use the 2017 Pohang earthquake dataset as the three-dimensional geodetic displacement **d** for line search algorithm comparison. For accurate measurement, the stacking InSAR and MAI method [45] was used. Four SAR images were acquired from CSK and ALOS2. The data size, the pixel size, and the total observation area are 628 × 518, 30 m^2, and 18.84 × 15.54 km, respectively. In the rest of this paper, we use **G**, x, and **d** to denote the dislocation model, the input source parameters, and the measured surface displacement, respectively.

3.2. GPU Kernel Optimization

Since the Okada model has no dependence between point calculations, it is straightforward to implement the model in a CUDA kernel function. We first implemented a sequential C++ Okada model based on the MATLAB open-source [46] and then wrote the CUDA kernel function computing the Okada model. Figure 4 shows the difference between CPU and CUDA implementation. While CPU code sequentially calculates the displacement of the 2D mesh using a nested loop, in parallel implementation, a host code launches the kernel with a specific keyword "__global__", and thousands of threads simultaneously execute the kernel function.

In the parallel implementation of the Okada model, each thread performs a massive number of arithmetic operations, but there are no shared variables in thread block scope and no thread synchronizations. Therefore, maximizing instruction efficiency can be the prime

optimization goal of the Okada model. However, since efficient instructions generally need additional resources, there is a trade-off between the instruction efficiency and the number of active warps that determine thread-level parallelism. We set our optimization goal to minimize the computation time by striking a balance between instruction efficiency and thread-level parallelism. For this purpose, we used two performance metrics: efficiency [47] for instruction efficiency and occupancy [48] for thread-level parallelism. We use the NVIDIA Nsight compute profiling tool to calculate the performance metrics.

CPU Implementation

```
template <typename real>
void cpuOkada(...) {

    // Initialization

    for (int i = 0; i < n_rows; i++) {
        for (int j = 0; j < n_cols; j++) {

            // Displacement calculation
        }
    }

    return;
}
```

CUDA Implementation

```
template <typename real>
__global__ void cuOkadaKernel(...) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < n_rows && col < n_cols) {

        // Displacement calculation

    }

    return;
}

template <typename real>
void cuOkada(...) {

    // Initialization

    // Set grid and thread block size
    dim3 dG(ceil(n_cols/ tx), ceil(n_rows/ ty), 1);
    dim3 dB(tx, ty, 1);

    // Call the Kernel
    cuOkadaKernel<real><<<dG, dB>>>(...);

    return;
}
```

Figure 4. Difference between CPU and CUDA implementations.

Efficiency indicates the instruction efficiency of the configuration. It is calculated as the reciprocal of the total number of PTX instructions of the kernel [47], described as Equation (5). Thus, high efficiency means that the kernel can perform the same task with fewer instructions.

$$\text{Efficiency} = \frac{1}{\text{The number of instructions}} \tag{5}$$

Occupancy represents the warp scheduling performance, calculated as Equation (6). Maximizing the number of active warps in SM facilitates hiding the latency caused by the global memory access, branch divergence, or instruction stall. Since a GPU has limited resources such as shared memory, registers, or thread blocks per SMs, inefficient resource usage may decrease the occupancy.

$$\text{Occupancy} = \frac{\text{Active warps per SM}}{\text{Maximum warps per SM}} \tag{6}$$

This paper attempt to obtain the best CUDA optimization configuration by employing the following three techniques.

1. Common subexpression elimination (CSE);
2. Constant caching;
3. Thread block size.

There are other CUDA optimization techniques, such as tilling or loop unrolling. However, we did not employ the tiling technique using shared memory because all operations are performed independently in each thread, and there are no variables referenced at the

thread block scope. Furthermore, since our implementation has no inner loop, we do not use the loop unrolling technique.

CSE removes repeated calculations such as arithmetic or load operation by replacing repeated calculations with the precomputed value from the register, as shown in Figure 5 [38]. Since the register reference cost is less than the arithmetic operation cost, CSE can increase the instruction efficiency. However, CSE may degrade warp scheduling performance since CSE tends to use additional registers. We set four options according to the degree of CSE: low (L), medium-low (ML), medium-high (MH), and high (H). As the degree of CSE increases, more repeated expressions are removed by using more registers. Figure 6 shows a part of the source code of the L and H options. In the source code, the L option compute repeated expressions with fewer variable usage. However, the H option precomputes the repeated expressions just once, but it needs 15 more registers than the L option. Table 3 shows the total number of register variables substituting repeated expressions for each degree of CSE.

Constant memory is a read-only memory that all threads can access. We can place the variable in the CUDA constant memory by declaring a variable with the keyword "__constant__". The constant memory variables originally reside in global memory, but they are cached automatically in a per-SM constant cache. Since they are not modified during kernel execution, the cache coherence issue does not occur. Furthermore, since the constant cache is optimized for broadcasting a value to massive threads, we can effectively reduce the register usage and memory bottleneck by the constant caching. Therefore, we set two options for constant caching: using constant caching ("with CC") and not using constant caching ("without CC"). When constant caching is used, we copy an array of 14 variables globally referenced by all threads to constant memory space, and the kernel uses the array directly. On the other hand, when constant caching is not used, the kernel accesses those variables in the global memory space.

Table 3. CSE option description.

CSE Option	The Number of Precomputed Subexpressions
L	24
ML	28
MH	40
H	43

Without CSE

```
double A1(double x, double y) {...}
double A2(double x, double y) {...}
double A3(double x, double y) {...}

int main()
{
    // Calculate common expressions repeatedly
    double B1 = A1(x, y) + A2(x, y) + A3(x, y);
    double B2 = A1(x, y) * A2(x, y) * A3(x, y);
    // …
}
```

With CSE

```
double A1(double x, double y) {...}
double A2(double x, double y) {...}
double A3(double x, double y) {...}

int main()
{
    // Use additional registers
    double tmpA1 = A1(x, y);
    double tmpA2 = A2(x, y);
    double tmpA3 = A3(x, y);

    double B1 = tmpA1 + tmpA2 + tmpA3;
    double B2 = tmpA1 * tmpA2 * tmpA3;
    // …
}
```

Figure 5. CSE implementation example.

CSE option L

```
// Calculate param, x, p, q, ...
double uxss, uyss, uzss, uxds, uyds, uzds,
       uxtf, uytf, uztf;


// Identical subexpressions are repeatedly computed
uxss = cuUxss(param, x, p, q);
uyss = cuUyss(param, x, p, q);
uzss = cuUzss(param, x, p, q);

uxds = cuUxds(param, x, p, q);
uyds = cuUyds(param, x, p, q);
uzds = cuUzds(param, x, p, q);

uxtf = cuUxtf(param, x, p, q);
uytf = cuUytf(param, x, p, q);
uztf = cuUztf(param, x, p, q);
```

CSE option H

```
// Calculate param, x, p, q, ...
double uxss, uyss, uzss, uxds, uyds, uzds,
       uxtf, uytf, uztf;
double X, R, yb, db;
double I1, I2, I3, I4, I5, invtan, lnrn, lnrd;
double A0, A1, A2;

// Compute subexpressions
X = GETX(x, q);
R = GETR(x, p, q);
yb = GETYB(param, p, q);
db = GETDB(param, p, q);
invtan = GETINVTAN(x, p, q, R);
lnrn = GETLNRN(R, p);
lnrd = GETLNRD(R, db);
A0 = GETA0(q, R);
A1 = GETA1(p, q, R);
A2 = GETA2(x, q, R);
I5 = cuI5(x, p, q, yb, db, R, X, lnrd, lnrn);
I4 = cuI4(x, p, q, yb, db, R, X, lnrd, lnrn);
I3 = cuI3(x, p, q, I4, yb, db, R, X, lnrd, lnrn);
I2 = cuI2(x, p, q, I3, yb, db, R, X, lnrd, lnrn);
I1 = cuI1(x, p, q, I5, yb, db, R, X, lnrd, lnrn);

// Do not need to repeatedly compute subexpressions
uxss = cuUxss(param, x, p, q, I1, yb, db, R, X,
              invtan, A0, A1, A2);
uyss = cuUyss(param, x, p, q, I2, yb, db, R, X,
              invtan, A0, A1, A2);
uzss = cuUzss(param, x, p, q, I4, yb, db, R, X,
              invtan, A0, A1, A2);
uxds = cuUxds(param, x, p, q, I3, yb, db, R, X,
              invtan, A0, A1, A2);
uyds = cuUyds(param, x, p, q, I1, yb, db, R, X,
              invtan, A0, A1, A2);
uzds = cuUzds(param, x, p, q, I5, yb, db, R, X,
              invtan, A0, A1, A2);
uxtf = cuUxtf(param, x, p, q, I3, yb, db, R, X,
              invtan, A0, A1, A2);
uytf = cuUytf(param, x, p, q, I1, yb, db, R, X,
              invtan, A0, A1, A2);
uztf = cuUztf(param, x, p, q, I5, yb, db, R, X,
              invtan, A0, A1, A2);
```

Figure 6. Comparison of the L and H options for CSE.

The thread block size is an important factor for determining the number of active warps per SM because the CUDA restricts the maximum number of resident threads and thread blocks per SM. Since many active warps hide the latencies of stalled instructions, the thread block size can affect the performance of the kernel. Therefore, we set four options of the thread block size, i.e., 64, 128, 256, and 512, based on a rule of thumb that threads per block should be a multiple of warp size [35] and theoretical occupancy derived by the CUDA occupancy calculator [49].

All possible combinations from the options of the three optimization techniques are $4 \times 2 \times 4 = 32$. We conducted experiments to evaluate the performance of our optimization approach. First, we compared 32 combinations using efficiency and occupancy to estimate the effects of options on the instruction efficiency and thread-level parallelism. Then, based on the computation time, we evaluated the significance of the instruction efficiency and thread-level parallelism on the parallel implementation of the Okada model and select the fastest one as the best CUDA optimization configuration.

3.3. Line Search Algorithm for the L-BFGS-B

After selecting the best CUDA optimization configuration described in Section 3.2, we integrate our CUDA implementation of the Okada model into the Monte Carlo method estimating the earthquake source parameters with the L-BFGS-B optimization. Then, we analyze the computational cost of subroutines in the L-BFGS-B to find the most time-consuming part and alleviate the bottleneck.

The L-BFGS-B proceeds in the sequence of generalized Cauchy point computation (algorithm CP), subspace minimization, line search, objective function and gradient up-

date, and update of a new limited-memory Hessian approximation, as described in Algorithm 1 [30]. We define the objective function f as the misfit of geodetic measurement $\mathbf{d}$ and model displacement $\mathbf{G}(x)$ and use the central finite difference equation for gradient approximation, as formulated by Equations (7) and (8), respectively.

Algorithm 1 L-BFGS-B algorithm.

1: **Result**: optimized output x_{out}
2: **Initialization**: random starting point x_0, the number of BFGS corrections stored m, bound constraint l, u, termination condition $maxIter$, $ftol$, $gtol$, $k = 0$
3: **repeat**
4: Algorithm CP: compute generalized Cauchy point x_k^c
5: Subspace minimization: compute search direction d_k
6: Line search: select step length α_k
7: Update point: $x_{k+1} = x_k + \alpha_k d_k$
8: Compute objective function $f_{k+1} = \frac{1}{2}\|\mathbf{G}(x_{k+1} - \mathbf{d})\|_2^2$ and gradient g_{k+1}
9: Update limited memory BFGS matrix H_{k+1}
10: $k = k + 1$
11: Compute projected gradient $P(x_k - g_k, l, u)$ $\left(P(x,l,u)_i = \begin{cases} l_i, & \text{if } x_i < l_i \\ x_i, & \text{if } l_i \le x_i \le u_i \\ u_i, & \text{if } x_i > u_i \end{cases} \right)$
12: **until** $\frac{f_{k-1} - f_k}{\max(\|f_k\|, \|f_{k-1}\|, 1)} < ftol$ and $\|P(x_k - g_k, l, u) - x_k\|_\infty < gtol$ and $k > maxIter$
13: $x_{out} = x_k$

$$f = \frac{1}{2}\|\mathbf{G}(x) - \mathbf{d}\|_2^2 \tag{7}$$

$$g \approx \frac{f(x+h) - f(x-h)}{h} \tag{8}$$

As summarized in Table 4, we analyzed the computational cost of L-BFGS-B subroutines [30,50], where m is the number of BFGS corrections stored, n is the number of variables to be optimized, t is the number of free variables, n_{int} is the number of segment explorations of the algorithm CP, $liter$ is the number of iterations of the line search algorithm, w is the mesh width of the Okada model, and h is the mesh height of the model, respectively. We optimized nine parameters, all input parameters of the Okada model except *Open*, in this paper and usually used small values of m (e.g., $3 \le m \le 20$) [50]. However, the mesh size of the Okada model varies from hundreds to thousands of squares depending on the scale of geodetic measurement, and thus the subroutine that includes the most dislocation model calculations takes the highest computational burden. Therefore, the line search algorithm can be a bottleneck of the L-BFGS-B because the Okada model is repeatedly calculated.

Table 4. Computational cost of the L-BFGS-B subroutines.

Subroutine	Computational Cost
Algorithm CP	$(2m+2)n + O(m^2) \cdot n_{int}$
Subspace minimization (direct primal method)	$2m^2t + 6mt + 4t + O(m^3)$
Line search (Armijo)	$liter \cdot O(wh)$
Line search (MT, BWW)	$liter \cdot (2+2n)O(wh)$
Update limited memory BFGS matrix	$2n + O(m^3)$
Compute objective function f	$O(wh)$
Compute gradient g	$(1+2n)O(wh)$

To alleviate this bottleneck, we compared four different candidates of the line search algorithm: Armijo, MT, BWW, and cuLBFGSB. We performed the five thousand iterations of the Monte Carlo method for each candidate in the real earthquake data mentioned in

Section 3.1 and evaluated the performance and correctness of the results. As for performance metrics, we used the average computation time of Monte Carlo iterations, RMSE, and parameter distribution for estimation speed, misfit, and uncertainties of parameters, respectively. Armijo, MT, and BWW were implemented with CPU-based L-BFGS-B, and cuLBFGSB was fully implemented with CUDA GPU and used simplified algorithm CP and line search algorithm since they have strong sequential dependence [43]. We used the open-source of Fei et al. [43] for MT and cuLBFGSB and implemented Armijo and BWW based on Burke [51]'s research [51].

4. Experimental Result and Discussion

This section investigates the performance effects of different combinations of the CUDA optimization techniques on the parallel implementation of the Okada model. Then, we explore the runtime behavior of the line search algorithms for the L-BFGS-B to estimate the best-fit source parameters. Finally, the correctness of our source parameter estimation result is verified by the RMSE and the residual displacement.

Table 5 shows the specification of the experiment environment. We used the NVIDIA GeForce RTX 2080 SUPER GPU with 48 SMs, 64 CUDA cores per SM, and 8GB GDDR6 memory.

Table 5. Hardware and software specification for experiment environment.

Type	Specification
OS	Ubuntu 18.04
CPU	Intel® Core™ i9-10900K @ 3.70GHz
RAM	128GB
GPU	NVIDIA GeForce RTX 2080 SUPER
CUDA version	11.0
CUDA compute capability	7.5
Host compiler	g++ 7.5.0

4.1. Kernel Optimization and Evaluation

For the first-order analysis of the optimization techniques, we assessed efficiency and occupancy of 32 different combinations presented in Section 3.2. For this purpose, we wrote CUDA kernel functions that calculate the objective function $\frac{1}{2}\|\mathbf{G}(x) - \mathbf{d}\|_2^2$ for the 500×500 mesh size. The results acquired from this experiment are shown in Figure 7. In Figure 7, both metrics were scaled with a minimum value of 0 and a maximum value of 1.

Figure 7a shows that CSE is the most dominant factor for efficiency. From the experimental results, we observe that efficiency tends to be proportional to the degree of CSE. We also notice that the thread block size and constant caching options have little effect on efficiency since the thread block size and constant caching affect only computing resource allocation and utilization of the GPU. By contrast, a higher degree of CSE decreases occupancy in most cases, as depicted in Figure 7b. In the "without CC" option, there is little difference in occupancy between the L and ML options and the MH and H options. This is because the difference in the number of precomputed subexpressions between the L and ML options and the MH and H options is relatively small, as shown in Table 3. However, in the "with CC" option, the H option shows higher occupancy than the MH option.

To analyze this cause, we compared the reduced number of reduced register usage by constant caching for each CSE option. Constant caching decreases register usage per thread in all CSE options. For example, in the L, ML, and H options, the number of register usage per thread reduced by constant caching becomes 12, 8, and 6, respectively. However, in the MH option, only two registers per thread are saved by constant caching, and thus constant caching has only a trivial effect on occupancy when the MH option is used.

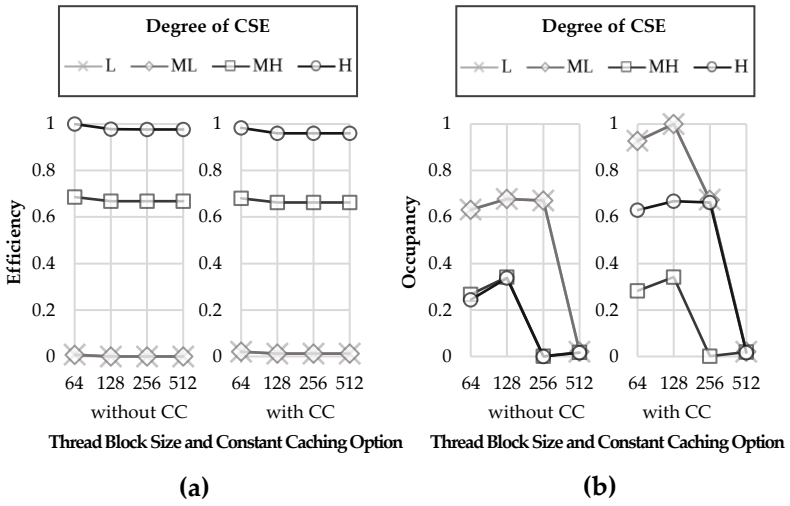

Figure 7. Performance metrics of configurations. (**a**) Efficiency; (**b**) Occupancy.

Since the maximum number of resident threads per SM is limited, the number of thread blocks in an SM decreases when a thread block size increases. For this reason, the larger the thread block size, the greater the loss of active warps per reduced thread block by resource limitation in SM. Therefore, we can conclude that a small thread block size leads to high occupancy and better thread-level parallelism.

Then, we compared the computation time of each configuration to select the best CUDA optimization configuration with the shortest average computation time. To this end, we measured the computation time of 1000 test runs for the 500 × 500 mesh size. As shown in Figure 8, the most remarkable enhancement appears when the degree of CSE changes from the ML to the MH, and the H option is the best configuration of CSE regardless of the other options of the two optimization techniques. As for the thread block size option, the computation time increases as the thread block size is enlarged. Furthermore, thus, the thread block size of 64 shows the best performance, up to a 5.3% reduction in the computation time than other thread block size options. Constant caching also slightly reduces the computation time by up to 3.7%.

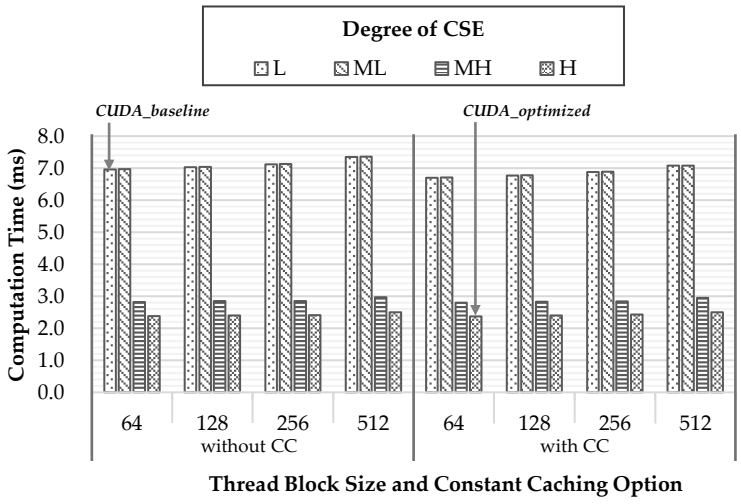

Figure 8. Average objective function computation time of configurations.

Finally, we selected [CSE, constant caching, thread block size]=[H, "with CC", 64] as the best CUDA optimization configuration for our application, called *CUDA_optimized*. We also set a baseline configuration for the performance comparison, called *CUDA_baseline*,

with the configuration [L, "without CC", 64]. The *CUDA_optimized* shows a 2.38 ms computation time, which is 2.93 times faster than the *CUDA_baseline*. This result indicates that the instruction efficiency has more effect on computation time than thread-level parallelism because a configuration with higher efficiency shows shorter computation time regardless of occupancy.

After the CUDA optimization, we also compared the computation time of our *CUDA_optimized* implementation with sequential CPU, multi-threaded CPU using OpenMP, and the *CUDA_baseline* for various mesh sizes ranging from 100 × 100 to 1000 × 1000. We generated 2D meshes by calculating grid coordinates from the reference point and the pixel size mentioned in Section 3.1.

Figure 9 shows that the *CUDA_optimized* has the shortest computation time, followed by the *CUDA_baseline*, 16/8/4/2 threads CPU, and sequential CPU implementations. We also observe that utilizing more threads improves the performance in multi-threaded CPU implementations. However, the performance gap between 8 and 16 threads is relatively small due to the limitation of CPU cores. Furthermore, in our experiments, using 32 and 64 threads showed almost the same performance as the 16 threads implementation.

Table 6 summarizes the speedup ratios achieved by the *CUDA_optimized* over CPU-based and the *CUDA_baseline* implementations. According to Table 6, the *CUDA_optimized* greatly improves the computation time of target objective functions compared with other implementations. For instance, it achieves up to 134.94×, 14.01×, and 2.99× speedups over sequential CPU, 16 threads CPU, and the *CUDA_baseline* for the 1000 × 1000 mesh size. Significantly, the performance gap between the *CUDA_baseline* and the *CUDA_optimized* is noteworthy. From the result, we can state that it is crucial to use the application-specific, hand-crafted optimization of CUDA kernel code to improve the performance, not merely write parallel code.

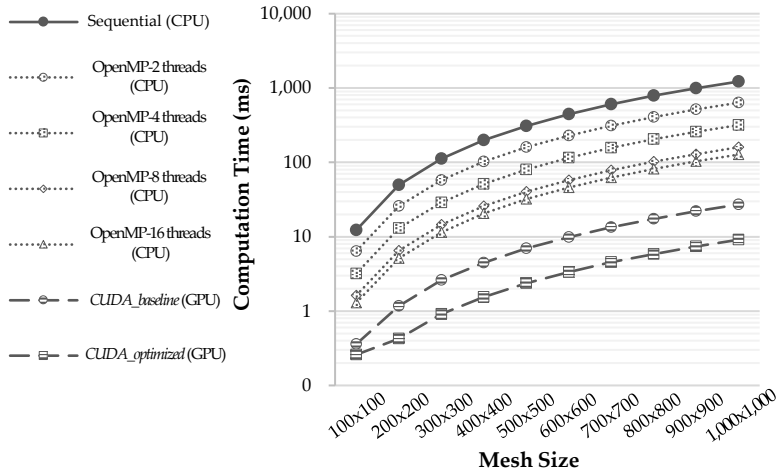

Figure 9. Average objective function computation time of sequential CPU, multi-threaded CPU, and parallel GPU implementations.

We also compared the results with previous work by Venetis et al. [42]. Although they used a different source parameter estimation algorithm from our study, they also used the Okada model and evaluated the GPU acceleration effect with the OpenMP implementation. The GPU implementation by Venetis et al. [42] achieves about 32× and 145× speedups over OpenMP-8 threads and sequential CPU implementations, respectively. To summarize the comparison result of performance evaluation, speedups over sequential CPU are similar. However, the speedup achieved over OpenMP-8 threads in [42] is about twice compared with our result in Table 6. We consider that this is because our study used the Intel® Core™ i9-10900K @ 3.70GHz processor with 10 CPU cores and 20 threads that is roughly two times

faster than the platform used in [42], i.e., Intel® Core™ i7-3770K @ 3.50GHz with 4 CPU cores and 8 threads.

Table 6. Speedup ratio achieved by the *CUDA_optimized* implementation.

Programming Model	Mesh Size									
	100 × 100	200 × 200	300 × 300	400 × 400	500 × 500	600 × 600	700 × 700	800 × 800	900 × 900	1000 × 1000
Sequential (CPU)	47.3466	116.4604	122.0598	128.4378	130.2648	132.4892	133.0033	134.3299	134.0760	134.9387
OpenMP-2 threads (CPU)	24.5715	60.4006	63.0262	66.2300	67.2731	68.5220	68.7152	69.3884	69.3488	69.6882
OpenMP-4 threads (CPU)	12.3336	30.5240	31.5752	33.2385	33.7230	34.3336	34.4576	34.8817	34.8903	35.0240
OpenMP-8 threads (CPU)	6.2591	15.3084	15.8690	16.7954	16.9538	17.2413	17.2489	17.4243	17.4365	17.5468
OpenMP-16 threads (CPU)	4.9698	12.0188	12.5567	13.2865	13.5082	13.7300	13.7568	13.9138	13.9230	14.0098
CUDA_baseline (GPU)	1.3831	2.7500	2.8542	2.9088	2.9322	2.9490	2.9651	2.9782	2.9864	2.9883

On the other hand, for the mesh sizes from 100 × 100 to 400 × 400, the speedup ratio increases considerably, but in the mesh sizes from 500 × 500 to 1000 × 1000, the speedup ratio no longer increases noticeably. As shown in Figure 10, we can also observe that the Giga floating-point operations per second (GFLOPS) of the *CUDA_optimized* code for different mesh sizes changes in agreement with the results in Table 6. Therefore, it is considered that the GFLOPS of the target application represents the throughput achievable on the GPU, which limits the degree of performance improvement.

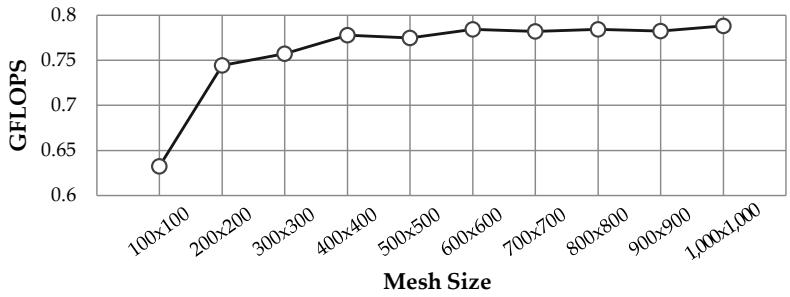

Figure 10. GFLOPS of *CUDA_optimized* implementation.

In summary, the CSE technique that enhances the instruction efficiency is the most effective optimization technique for parallel implementation of the Okada model. Our *CUDA_optimized* implementation achieves more than an order of magnitude speedup compared to traditional multi-threaded CPU-based implementations. Moreover, our optimization approach shows significant performance improvement, almost three times more speedup than before CUDA optimization.

4.2. Line Search Algorithm Comparison and Correctness Verification

We applied our parallel implementation to the earthquake source parameter estimation procedure using the L-BFGS-B. In doing so, we evaluated four different solutions introduced in Section 3.3, which are Armijo, MT, BWW, and cuLBFGSB, to derive the best-fit source parameters. Five thousand iterations of the Monte Carlo method were performed for each candidate. We defined the lower and upper bound of source parameters based on the study of Lee [52], as summarized in Table 7. Starting points of the Monte Carlo iterations were randomly sampled from a uniform distribution in the boundaries, and termination conditions of the L-BFGS-B were set to $maxIter = 1000$, $ftol = 10^{-3}$, $m = 8$, and $gtol = 10^{-3}$.

Table 7. Bound constraints of source parameters.

Constraints	Parameters								
	E (km)	*N* (km)	*Depth* (km)	*Strike* (deg.)	*Dip* (deg.)	*Length* (km)	*Width* (km)	*Rake* (deg.)	*Slip* (cm)
Lower bound	4.8	−12.0	3.0	110.0	33.0	4.0	4.0	80.0	10.0
Upper bound	10.0	−6.0	5.7	235.0	55.0	6.5	6.7	150.0	30.0

We used the RMSE, computation time, and 95% confidence interval obtained from the Monte Carlo iterations as the indicators to evaluate and compare the performance and correctness of candidates. Table 8 lists the statistics of the RMSE and the computation time measured for the candidate implementations. As shown in Table 8, Armijo and cuLBFGSB outperform MT and BWW in the mean and the standard deviation values of RMSE. Although Armijo, MT, and cuLBFGSB show a similar mean computation time, Armijo has the lowest mean and standard deviation values of the computation time. In addition, the summary of the best-fit parameters minimizing the RMSE and their 95% confidence interval is given in Table 9. The best-fit parameters derived from candidates are almost identical, but Armijo has a much smaller confidence interval than the other candidates, with a 42∼84% interval size reduction. This result remarks that the best-fit parameters determined by Armijo are the most reasonable.

Table 8. RMSE and the computation time statistics of four candidates.

Algorithm	RMSE (cm)				Computation Time (s)			
	Min.	Max.	Mean.	Std.	Min.	Max.	Mean.	Std.
Armijo	0.4741	0.5103	0.4784	0.0030	0.2465	1.9259	1.0198	0.2481
MT	0.4745	0.8243	0.5291	0.0589	0.2309	3.6965	1.1638	0.5253
BWW	0.4748	0.8164	0.5273	0.0613	0.1737	11.4612	1.8269	0.9649
cuLBFGSB	0.4739	0.9450	0.4860	0.0258	0.1163	2.5101	1.1922	0.3514

Table 9. The best-fit parameters and a 95% confidence interval of the Monte Carlo results.

Algorithm	Parameters								
	E (km)	*N* (km)	*Depth* (km)	*Strike* (deg.)	*Dip* (deg.)	*Length* (km)	*Width* (km)	*Rake* (deg.)	*Slip* (cm)
Armijo	6.7756 ± 0.1814	−8.0579 ± 0.1563	3.7138 ± 0.2603	203.9841 ± 6.5069	38.7941 ± 2.3886	5.0784 ± 0.4434	5.3181 ± 0.4994	115.3062 ± 4.7805	12.5695 ± 2.0428
MT	6.8278 ± 0.8900	−7.9336 ± 0.9688	3.6811 ± 1.0870	201.7411 ± 33.9702	38.5657 ± 7.9274	4.9229 ± 0.8697	5.3982 ± 1.0252	112.9773 ± 20.5970	12.9347 ± 5.2915
BWW	6.7314 ± 1.0308	−8.0493 ± 1.0263	3.7909 ± 0.8933	202.7314 ± 31.6251	38.7331 ± 8.3463	5.0749 ± 0.8959	5.6685 ± 0.9614	115.2043 ± 18.3948	12.2949 ± 5.4515
cuLBFGSB	6.7366 ± 0.3813	−7.9948 ± 0.4724	3.6931 ± 0.7714	204.2539 ± 12.2758	38.4440 ± 4.1287	5.0455 ± 1.4588	5.4938 ± 1.8056	114.8131 ± 10.1586	12.3322 ± 12.0922

To estimate the uncertainties of parameters in detail, we illustrated the histogram of each parameter in Figure 11. All parameters from Armijo feature a unimodal histogram with a small standard deviation. However, most source parameters estimated from MT show multiple peaks, and the *Dip* derived from BWW peaks around both the lower and upper bound. Furthermore, cuLBFGSB shows that most results of the *Length* and the *Width* converge on the lower bound, and the *Slip* does not converge on the unique solution either with a high standard deviation. It seems that the highly biased results of cuLBFGSB were caused by its simplified line search algorithm. This result confirms again that Armijo has the best agreement with the earthquake source parameter estimation.

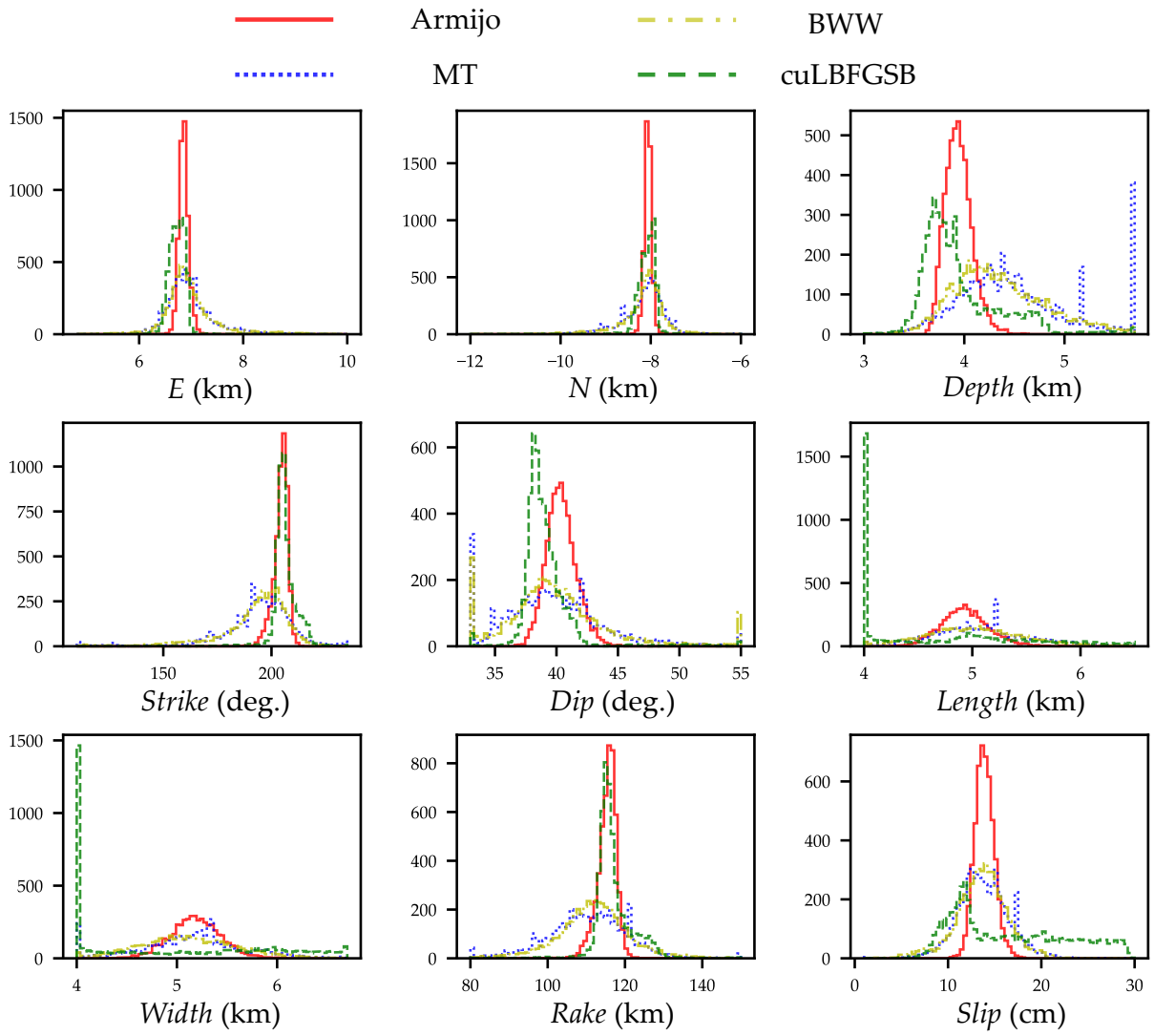

Figure 11. Parameter distribution of the Monte Carlo results.

Therefore, as the final step, we verified the correctness of our source parameter estimation framework with the best-fit parameters derived by Armijo. Figure 12 compares the geodetic displacement and the modeled displacement acquired from the final source parameters. Each row of the figure represents the surface displacement for the E–W direction, the N-S direction, and the depth direction. The third column also shows the residual displacement between the geodetic and the modeled displacements. The residual displacement demonstrates that our proposed framework approximates the geodetic displacement quite precisely, where the RMSE for each direction is 0.49, 0.31, and 0.33 cm for the E–W, the N–S, and the depth direction, respectively.

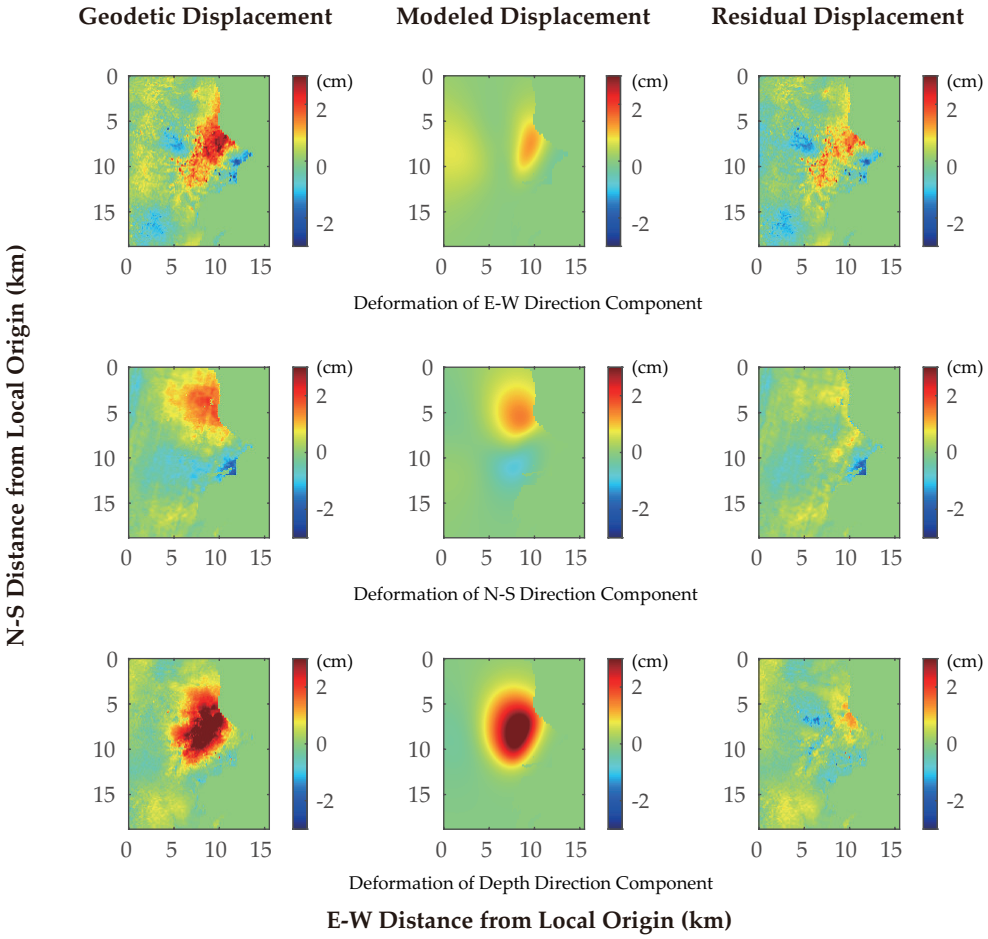

Figure 12. Final results of source modeling.

5. Conclusions

This paper presented a performance acceleration framework for earthquake source parameter estimation based on the Okada dislocation model using CUDA GPU. To this end, we first carefully set the optimization technique candidates considering the characteristics of the target application and analyzed the effects of various combinations from the suggested optimization options in terms of efficiency and occupancy. Then, based on the analysis, we performed the CUDA kernel optimization of the Okada model and evaluated its performance improvement compared to multi-threaded CPU-based and baseline GPU implementations. We also explored the performance and correctness of different line search algorithms for the L-BFGS-B optimization, which is one of the most time-consuming parts of our target problem. Finally, using the assessment result, we selected the Armijo line search as the most efficient one.

The performance evaluation results for CUDA kernel optimization showed that the *CUDA_optimized* implementation achieved up to 2.99× and 14.00× speedups over the *CUDA_baseline* and 16 threads CPU implementations, respectively. We also observed that Armijo shows a 42∼84% reduction in confidence interval size over other candidates, and all parameters derived from Armijo feature a unimodal histogram with a small standard deviation. The best-fit parameters from Armijo had the best consistency between geodetic and modeled displacement, at most 0.50cm RMSE for each direction. Consequently, the results demonstrated that our proposed approach successfully accelerates earthquake source parameter estimation procedure with correctness verification of the best-fit parameters.

Author Contributions: Conceptualization, S.L. and T.K.; methodology, S.L.; implementation, S.L.; evaluation, S.L.; writing, S.L. and T.K.; supervision, T.K.; funding acquisition, T.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the 2020 Research Fund of the University of Seoul.

Data Availability Statement: The surface deformation maps were provided by Jung and Baek. They have generated the maps from the ALOS PALSAR-2 interferometric data that were provided through the JAXA's ALOS-2 research program (RA4, PI No. 1412).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

InSAR	Interferometric synthetic aperture radar
3D	Three-dimensional
2D	Two-dimensional
GPU	Graphics processing unit
CUDA	Compute unified device architecture
RMSE	Root-mean-square-error
LOS	Line-of-sight
MAI	Multiple aperture interferometry
BFGS	Broyden–Fletcher–Goldfarb–Shanno
L-BFGS	Limited memory Broyden–Fletcher–Goldfarb–Shanno
L-BFGS-B	Limited memory Broyden–Fletcher–Goldfarb–Shanno with boundaries
Armijo	Armijo line search
BWW	Bisection method for weak-Wolfe conditions
MT	Móre-Thuente line search
PTX	Parallel thread execution
SM	Streaming multiprocessor
SIMD	single instruction, multiple data
DRAM	Dynamic random access memory
cuLBFGSB	Parallel implementation of the L-BFGS-B
CSE	Common subexpression elimination
GFLOPS	Giga floating point operations per second

References

1. Oldenburg, D.W.; Li, Y. Inversion for applied geophysics: A tutorial. In *Near-Surface Geophysics*; Society of Exploration Geophysicists: Tulsa, OK, USA, 2005; pp. 89–150.
2. Clarke, P.J.; Paradissis, D.; Briole, P.; England, P.C.; Parsons, B.E.; Billiris, H.; Veis, G.; Ruegg, J. Geodetic investigation of the 13 May 1995 Kozani-Grevena (Greece) Earthquake. *Geophys. Res. Lett.* **1997**, *24*, 707–710. [CrossRef]
3. Feigl, K. Estimating earthquake source parameters from geodetic measurements. In *International Geophysics*; Elsevier: Amsterdam, The Netherlands, 2002; Volume 81, pp. 607–cp1.
4. Aster, R.C.; Borchers, B.; Thurber, C.H. *Parameter Estimation and Inverse Problems*; Elsevier: Amsterdam, The Netherlands, 2018.
5. Moreira, A.; Prats-Iraola, P.; Younis, M.; Krieger, G.; Hajnsek, I.; Papathanassiou, K.P. A tutorial on synthetic aperture radar. *IEEE Geosci. Remote. Sens. Mag.* **2013**, *1*, 6–43. [CrossRef]
6. Schlögel, R.; Doubre, C.; Malet, J.P.; Masson, F. Landslide deformation monitoring with ALOS/PALSAR imagery: A D-InSAR geomorphological interpretation method. *Geomorphology* **2015**, *231*, 314–330. [CrossRef]
7. Hu, B.; Wu, Y.; Zhang, X.; Yang, B.; Chen, J.; Li, H.; Chen, X.; Chen, Z. Monitoring the thaw slump-derived Thermokarst in the Qinghai-Tibet plateau using satellite SAR interferometry. *J. Sens.* **2019**, *2019*, 1698432. [CrossRef]
8. Gray, L. Using multiple RADARSAT InSAR pairs to estimate a full three-dimensional solution for glacial ice movement. *Geophys. Res. Lett.* **2011**, *38*, 132–140. [CrossRef]
9. Okada, Y. Surface deformation due to shear and tensile faults in a half-space. *Bull. Seismol. Soc. Am.* **1985**, *75*, 1135–1154. [CrossRef]
10. Okada, Y. Internal deformation due to shear and tensile faults in a half-space. *Bull. Seismol. Soc. Am.* **1992**, *82*, 1018–1040. [CrossRef]
11. Yang, X.M.; Davis, P.M.; Dieterich, J.H. Deformation from inflation of a dipping finite prolate spheroid in an elastic half-space as a model for volcanic stressing. *J. Geophys. Res. Solid Earth* **1988**, *93*, 4249–4257. [CrossRef]
12. McTigue, D. Elastic stress and deformation near a finite spherical magma body: Resolution of the point source paradox. *J. Geophys. Res. Solid Earth* **1987**, *92*, 12931–12940. [CrossRef]
13. CUDA Toolkit. Available online: https://developer.nvidia.com/cuda-toolkit (accessed on 22 May 2021).
14. Wang, C.; Ding, X.; Li, Q.; Jiang, M. Equation-based InSAR data quadtree downsampling for earthquake slip distribution inversion. *IEEE Geosci. Remote. Sens. Lett.* **2014**, *11*, 2060–2064. [CrossRef]

15. De Novellis, V.; Castaldo, R.; De Luca, C.; Pepe, S.; Zinno, I.; Casu, F.; Lanari, R.; Solaro, G. Source modelling of the 2015 Wolf volcano (Galápagos) eruption inferred from Sentinel 1-A DInSAR deformation maps and pre-eruptive ENVISAT time series. *J. Volcanol. Geotherm. Res.* **2017**, *344*, 246–256. [CrossRef]
16. Funning, G.J.; Parsons, B.; Wright, T.J.; Jackson, J.A.; Fielding, E.J. Surface displacements and source parameters of the 2003 Bam (Iran) earthquake from Envisat advanced synthetic aperture radar imagery. *J. Geophys. Res. Solid Earth* **2005**, *110*. [CrossRef]
17. Qu, W.; Zhang, B.; Lu, Z.; Kim, J.W.; Zhang, Q.; Gao, Y.; Hao, M.; Zhu, W.; Qu, F. Source parameter estimation of the 2009 Ms6. 0 Yao'an Earthquake, Southern China, using InSAR observations. *Remote Sens.* **2019**, *11*, 462. [CrossRef]
18. Dicelis, G.; Assumpção, M.; Kellogg, J.; Pedraza, P.; Dias, F. Estimating the 2008 Quetame (Colombia) earthquake source parameters from seismic data and InSAR measurements. *J. S. Am. Earth Sci.* **2016**, *72*, 250–265. [CrossRef]
19. Bagnardi, M.; Hooper, A. Inversion of surface deformation data for rapid estimates of source parameters and uncertainties: A Bayesian approach. *Geochem. Geophys. Geosyst.* **2018**, *19*, 2194–2211. [CrossRef]
20. Dutta, R.; Jónsson, S.; Wang, T.; Vasyura-Bathke, H. Bayesian estimation of source parameters and associated Coulomb failure stress changes for the 2005 Fukuoka (Japan) earthquake. *Geophys. J. Int.* **2018**, *213*, 261–277. [CrossRef]
21. Šílený, J. Earthquake source parameters and their confidence regions by a genetic algorithm with a 'memory'. *Geophys. J. Int.* **1998**, *134*, 228–242. [CrossRef]
22. Picozzi, M.; Oth, A.; Parolai, S.; Bindi, D.; De Landro, G.; Amoroso, O. Accurate estimation of seismic source parameters of induced seismicity by a combined approach of generalized inversion and genetic algorithm: Application to The Geysers geothermal area, California. *J. Geophys. Res. Solid Earth* **2017**, *122*, 3916–3933. [CrossRef]
23. Lee, S.; Kim, T. Search Space Reduction for Determination of Earthquake Source Parameters Using PCA and-Means Clustering. *J. Sens.* **2020**, *2020*, 8826634. [CrossRef]
24. Nocedal, J.; Wright, S. *Numerical Optimization*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2006.
25. Moré, J.J. The Levenberg–Marquardt algorithm: Implementation and theory. In *Numerical Analysis*; Springer: Berlin/Heidelberg, Germany, 1978; pp. 105–116.
26. Shan, S. A Levenberg–Marquardt Method for Large-Scale Bound-Constrained Nonlinear Least-Squares. Ph.D. Thesis, University of British Columbia, Vancouver, BC, Canada, 2008.
27. Coleman, T.F.; Li, Y. On the convergence of interior-reflective Newton methods for nonlinear minimization subject to bounds. *Math. Program.* **1994**, *67*, 189–224. [CrossRef]
28. Voglis, C.; Lagaris, I. A rectangular trust region dogleg approach for unconstrained and bound constrained nonlinear optimization. In Proceedings of the WSEAS International Conference on Applied Mathematics, Corfu Island, Greece, 16–19 August 2004; Volume 7.
29. Liu, D.C.; Nocedal, J. On the limited memory BFGS method for large scale optimization. *Math. Program.* **1989**, *45*, 503–528. [CrossRef]
30. Byrd, R.H.; Lu, P.; Nocedal, J.; Zhu, C. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.* **1995**, *16*, 1190–1208. [CrossRef]
31. Wolfe, P. Convergence conditions for ascent methods. *SIAM Rev.* **1969**, *11*, 226–235. [CrossRef]
32. Armijo, L. Minimization of functions having Lipschitz continuous first partial derivatives. *Pac. J. Math.* **1966**, *16*, 1–3. [CrossRef]
33. Moré, J.J.; Thuente, D.J. Line search algorithms with guaranteed sufficient decrease. *ACM Trans. Math. Softw. TOMS* **1994**, *20*, 286–307. [CrossRef]
34. Kirk, D.B.; Hwu, W.-M. *Programming Massively Parallel Processors: A Hands-On Approach*; Morgan Kaufmann: Cambridge, MA, USA, 2016.
35. CUDA C++ Programming Guide. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (accessed on 22 May 2021).
36. Ryoo, S.; Rodrigues, C.I.; Baghsorkhi, S.S.; Stone, S.S.; Kirk, D.B.; Hwu, W.m.W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Salt Lake City, UT, USA, 20–23 February 2008; pp. 73–82.
37. NVCC. Available online: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html (accessed on 22 May 2021).
38. Ryoo, S.; Rodrigues, C.I.; Stone, S.S.; Stratton, J.A.; Ueng, S.Z.; Baghsorkhi, S.S.; Hwu, W.-M. Program optimization carving for GPU computing. *J. Parallel Distrib. Comput.* **2008**, *68*, 1389–1401. [CrossRef]
39. Plaza, A.; Du, Q.; Chang, Y.L.; King, R.L. High performance computing for hyperspectral remote sensing. *IEEE J. Sel. Top. Appl. Earth Obs. Remote. Sens.* **2011**, *4*, 528–544. [CrossRef]
40. Sánchez, S.; Ramalho, R.; Sousa, L.; Plaza, A. Real-time implementation of remotely sensed hyperspectral image unmixing on GPUs. *J. Real-Time Image Process.* **2015**, *10*, 469–483. [CrossRef]
41. Liao, P.C.; Lii, C.C.; Lai, Y.C.; Chang, P.Y.; Zhang, H.; Thurber, C. A graphics processing unit implementation and optimization for parallel double-difference seismic tomography. *Bull. Seismol. Soc. Am.* **2014**, *104*, 953–961. [CrossRef]
42. Venetis, I.E.; Saltogianni, V.; Stiros, S.; Gallopoulos, E. Multivariable inversion using exhaustive grid search and high-performance GPU processing: a new perspective. *Geophys. J. Int.* **2020**, *221*, 905–927. [CrossRef]
43. Fei, Y.; Rong, G.; Wang, B.; Wang, W. Parallel L-BFGS-B algorithm on gpu. *Comput. Graph.* **2014**, *40*, 1–9. [CrossRef]
44. Hu, J.; Li, Z.; Ding, X.; Zhu, J.; Zhang, L.; Sun, Q. Resolving three-dimensional surface displacements from InSAR measurements: A review. *Earth Sci. Rev.* **2014**, *133*, 1–17. [CrossRef]

45. Jung, H.S.; Lu, Z.; Won, J.S.; Poland, M.P.; Miklius, A. Mapping three-dimensional surface deformation by combining multiple-aperture interferometry and conventional interferometry: Application to the June 2007 eruption of Kilauea volcano, Hawaii. *IEEE Geosci. Remote Sens. Lett.* **2010**, *8*, 34–38. [CrossRef]
46. Beauducel, F. Matlab/Octave Tools for Geophysical Studies. 2014. Available online: https://www.ipgp.fr/~beaudu/matlab.html (accessed on 1 October 2021).
47. Ryoo, S.; Rodrigues, C.I.; Stone, S.S.; Baghsorkhi, S.S.; Ueng, S.Z.; Stratton, J.A.; Hwu, Hwu, W.m.W. Program optimization space pruning for a multithreaded gpu. In Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '08, Boston, MA, USA, 5–9 April 2008; pp. 195–204. [CrossRef]
48. Harris, M. Optimizing cuda. In Proceedings of the Tutorial at the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), Reno, NV, USA, 10–16 November 2007; Volume 60.
49. CUDA Occupancy Calculator. Available online: https://docs.nvidia.com/cuda/cuda-occupancy-calculator/ (accessed on 22 May 2021).
50. Zhu, C.; Byrd, R.H.; Lu, P.; Nocedal, J. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw. TOMS* **1997**, *23*, 550–560. [CrossRef]
51. Burke, J.V. Nonlinear optimization. *Lect. Notes Math.* **2014**, *408*, 80.
52. Lee, K.K. *Final Report of the Korean Government Commission on Relations between the 2017 Pohang Earthquake and EGS Project*; Technical Report; The Geological Society of Korea: Gangnam-gu, Seoul, 2019.