electronics

MDPI

Article

High-Throughput MPSoC Implementation of Sparse Bayesian Learning Algorithm

Jinyang Wang [1,2,3], El-Bay Bourennane [3], Mahdi Madani [3], Jun Wang [1,2], Chao Li [1,2], Yupeng Tai [1,2], Longxu Wang [1,2], Fan Yang [4] and Haibin Wang [1,2,*]

[1] State Key Laboratory of Acoustics, Institute of Acoustics, Chinese Academy of Sciences, Beijing 100190, China; wangj@mail.ioa.ac.cn (J.W.); wanglongxu@mail.ioa.ac.cn (L.W.)
[2] University of Chinese Academy of Sciences, Beijing 100049, China
[3] Laboratory ImViA, Université Bourgogne Franche-Comté, 21078 Dijon, France; mahdi.madani@u-bourgogne.fr (M.M.)
[4] Laboratory for Research on Learning and Developement—UMR CNRS 5022, University of Burgundy, 21000 Dijon, France; fanyang@u-bourgogne.fr
* Correspondence: whb@mail.ioa.ac.cn

Abstract: In the field of sparse signal reconstruction, sparse Bayesian learning (SBL) has excellent performance, which is accompanied by extremely high computational complexity. This paper presents an efficient SBL hardware and software (HW&SW) co-implementation method using the ZYNQ series MPSoC (multiprocessor system-on-chip). Firstly, considering the inherent challenges in parallelizing iterative algorithms like SBL, we propose an architecture based on the iterative calculations implemented on the PL side (FPGA) and the iteration control and input management handled by the PS side (ARM). By adopting this structure, we can take advantage of task-level pipelines on the FPGA side, effectively utilizing time and space resources. Secondly, we utilize LDL decomposition to perform the inversion of the Hermitian matrix, which not only exhibits the lowest computational complexity and requires fewer computational resources but also achieves a higher level in the parallel pipeline mechanism compared with other alternative methods. Furthermore, the algorithm conducts iterations sequentially, utilizing the parameters derived from the previous dataset as prior information for initializing the subsequent dataset's initial values. This approach helps to reduce the number of iterations required. Finally, with the help of Vitis HLS 2022.2 and Vivado tools, we successfully accomplished the development of a hardware design language and its implementation on the ZYNQ UltraScale+ MPSoC ZCU102 platform. Meanwhile, we have solved a direction of arrival (DOA) estimation problem using horizontal line arrays to verify the practical feasibility of the method.

Keywords: sparse Bayesian learning; ZYNQ; FPGA; task-level pipeline; LDL matrix inversion

Citation: Wang, J.; Bourennane, E.-B.; Madani, M.; Wang, J.; Li, C.; Tai, Y.; Wang, L.; Yang, F.; Wang, H. High-Throughput MPSoC Implementation of Sparse Bayesian Learning Algorithm. *Electronics* **2024**, *13*, 234. https://doi.org/10.3390/electronics13010234

Academic Editor: Juan M. Corchado

Received: 22 November 2023
Revised: 30 December 2023
Accepted: 3 January 2024
Published: 4 January 2024

1. Introduction

In the context of compressive sensing [1], sparse Bayesian learning (SBL) [2,3] is one implementation approach that offers several advantages over algorithms like Orthogonal Matching Pursuit (OMP) [4] and LASSO [5]. It exhibits superior stability, requiring no parameter tuning, and provideing lower sidelobes and higher resolution. These advantages make it a high potential for applications such as medical signal recovery, array signal processing, radar and sonar localization, etc. However, its main drawback lies in its higher algorithmic complexity. Applications such as direction of arrival (DOA) estimation in radar and sonar, channel estimation, and target tracking all have a significant demand for real-time performance. High complexity makes SBL challenging to be widely employed in practical applications.

In the normal process of SBL, parameter estimation is typically achieved using the Expectation-Maximization (EM) algorithm or direct derivation methods [6]. Parameters

are iteratively updated until convergence, and during each iteration, the inversion of an M-dimensional matrix is required, resulting in a complexity of $O(M^3)$, where M represents the number of sensors. Additionally, the matrix multiplication can lead to a maximum complexity of $O(M^2N)$, where N represents the number of basis functions, and often $N >> M$. Consequently, the overall complexity of the algorithm per iteration is $O(M^2N)$, but the difficulty and complexity of matrix inversion cannot be overlooked. Moreover, the limitation of SBL's practical application is further compounded by the necessity of multiple iterations to obtain results.

To improve computational efficiency, many efforts have been focused on reducing the complexity of SBL from the algorithmic perspective, leading to the emergence of various low-complexity fast SBL methods in recent years [7,8]. For instance, Tipping proposed a greedy method in [7], which starts from an empty set and continuously adds or removes basis functions to reduce computation time. Building on this approach, Babacan [9] developed a fast Bayesian algorithm based on the Laplace prior model, which is specifically used for recovering and reconstructing sparse images. In [10], Shutin proposed a fast SBL algorithm based on variational Bayesian inference by analyzing stationary points of the variational update expressions, which improved convergence speed. Additionally, there has been a gradual exploration of approximation-based methods. In [11,12], they replaced the E-step of the EM update formula in SBL with a low-complexity approach. In [11,13], the Bayesian model was transformed into a factor graph, and belief propagation was employed to calculate mean and variance. However, its application is limited by the strong assumption of sparsity in the basis matrix. To avoid matrix inversion, [12] used the generalized approximate message passing (GAMP) method as a replacement for the E-step. Although this avoids matrix inversion, it introduces another iterative method, leading to nested iterations in large-scale applications with inherent limitations. In [8], a Gaussian scale mixture prior model was used, and a block coordinate descent was employed to solve the resulting non-convex optimization problem, effectively reducing complexity through approximate Gaussian likelihood.

It is not difficult to notice that the methods mentioned above for reducing the complexity of SBL are predominantly achieved through approximations, which may lead to some performance loss. In order to enhance the computational efficiency of the SBL algorithm without loss in performance, this paper explores the hardware acceleration to achieve both high performance and high throughput.

Therefore, we analyzed the advantages and drawbacks of the three main families of embedded systems (FPGA, GPU, and CPU) to choose the hardware platform best suited to our objective (algorithm hardware acceleration). This analysis shows that the FPGA (Field-Programmable Gate Array) platforms are more suitable to achieve our objective, as discussed here. The FPGA offers several advantages over CPU or GPU [14]: flexibility that can be reconfigured and customized to perform specific tasks; parallel or pipeline computing; real-time processing; portability and power efficiency, etc. In [15], to accelerate the compression sensing algorithm, the authors present implementation structures for Cholesky decomposition on both FPGA and GPU. The results demonstrate a significant acceleration of 15 times and 38 times, respectively, compared to using LAPACK on CPU and a hybrid CPU/GPU system with MAGMA. This result is largely attributed to the limited on-chip hardware resources of FPGAs, making GPUs more suitable for large matrix operations, while FPGAs are better suited for small matrix computations. In [16], a study is conducted on three hardware accelerators: ARM57 CPU, Jetson TX2 GPU, and ZYNQ MPSoC ZCU102 FPGA, focusing on embedded vision applications. The findings reveal that relative to the CPU, the GPU achieves a 1.1–3.2 times reduction in energy/frame, while the FPGA's performance improves as the kernel complexity or pipeline complexity increases, reaching a substantial energy/frame reduction ratio of 1.2–22.3 times. However, the existing work [17] indicates that the throughput of the SBL algorithm implemented solely on FPGA is notably low, falling far short of real-time processing requirements. To leverage the parallel processing capabilities of the FPGA more effectively, the authors

propose a software-hardware co-processing system for target detection in [18] with the computationally intensive layer placed on the FPGA and the non-computational layer on the ARM. Inspired by this, we intend to explore the application of such a structure to accelerate the SBL algorithm.

By analyzing the solution process of the SBL algorithm, we can summarize three reasons that contribute to the high computational complexity and difficulty in accelerating the implementation of the SBL algorithm:

1. As an iterative solving algorithm, there is a strong data dependency between consecutive iterations, which limits the application of parallel acceleration in computations.
2. Within each iteration, there are operations such as matrix inversion and numerous matrix multiplications, which are the main sources of algorithmic complexity.
3. A large number of iterations are required to obtain the desired result, typically ranging from 300 to 500 iterations.

In this work, we explored the powerful properties of FPGA technology to achieve an efficient implementation method of the SBL algorithm. Therefore, to address the aforementioned three issues, we proposed the following solutions:

Firstly, by leveraging the capabilities of ZYNQ MPSoC (multiprocessor system-on-chip) [18], which combines FPGA and ARM processors, we propose a hardware and software (HW&SW) co-implementation method for SBL algorithm. The main body of the algorithm is implemented on the FPGA part, while the control of data input and iteration termination is handled by the ARM processor. The main body of the algorithm on the FPGA is divided into multiple subtasks based on functionality and computational complexity, and a task-level pipeline is implemented between these tasks. This processing architecture effectively utilizes the advantages of parallel computation on the FPGA.

Secondly, the algorithm utilizes LDL decomposition, triangular matrix inversion, and matrix multiplication in a three-step process to calculate the inverse of the Hermitian matrix. Compared to traditional methods such as LU decomposition, QR decomposition, or Cholesky decomposition, LDL decomposition for inversion offers the lowest computational complexity, minimal resource usage, and the highest numerical stability [19]. Then, we utilized an optimized complex multiplication structure and combined it with pipelining or loop-unrolling techniques to implement matrix multiplication efficiently.

Finally, considering that the sparse signal to be estimated in practical applications often exhibits continuity, a sequential estimation strategy is employed [20]. This strategy incorporates the prior information of the previous parameter estimation results into the initial parameters for the next set of data, thereby reducing the number of iterations required.

By applying these three optimization strategies, an accelerated implementation architecture is designed for the ZYNQ UltraScale+ MPSoC based on six ARM cores and FPGA logic, and simulation experiments are conducted using DOA estimation as an example.

The remainder of the paper is organized as follows. Section 2 provides the principle of the core part of the SBL algorithm. Section 3 presents the hardware acceleration architecture for the algorithm. Section 4 provides a resources utilization and latency analysis of the architecture. Section 5 presents the results of simulation experiments on DOA estimation. The final section summarizes the work and provides future prospects.

The notations used in this paper are as follows:

Vectors and matrices are represented by bold symbols. Particularly, $\mathbf{I}$ denotes the identity matrix with appropriate dimensions according to the context. $\mathbf{A}^T$ and $\mathbf{A}^H$ denote the transpose and Hermitian transpose of matrix $\mathbf{A}$, respectively. $\mathbf{A}^{-1}$ denotes the inverse of matrix $\mathbf{A}$. $\|\mathbf{x}\|_1, \|\mathbf{x}\|_2$ denote the ℓ_1 norm and ℓ_2 norm of vector $\mathbf{x}$, respectively. $\|\mathbf{A}\|_{\mathcal{F}}$ denotes the Frobenius norm of the matrix $\mathbf{A}$. $diag(\cdot)$ denotes a square diagonal matrix with the elements of vector on the main diagonal or the diagonal elements of matrix to form a vector. x^* represents the conjugate of complex number x. $\mathbb{R}$ and $\mathbb{C}$ represent the space of real numbers and the space of complex numbers, respectively.

2. Algorithm Principle

This section introduces and analyzes the principle part of the algorithm, mainly including an introduction to the SBL algorithm and derivation of the LDL matrix inversion.

2.1. SBL Algorithm Architecture

The general estimation model of sparse Bayesian learning (SBL) can be described using the multi-measurement vector (MMV) model Equation (1), which is an extension of the single measurement vector (SMV) model.

$$\mathbf{Y} = \mathbf{\Phi X} + \mathbf{E} \tag{1}$$

Without loss of generality, we can assume that the measurement vectors are complex-valued. $\mathbf{Y}$, $\mathbf{E} \in \mathbb{C}^{M\times T}$ are the observed signal and Gaussian white noise, respectively, received by M sensors and having the length T. $\mathbf{\Phi} \in \mathbb{C}^{M\times N}$ is the observation dictionary matrix, and it is fixed and independent of the signal $\mathbf{X}$. $\mathbf{X} \in \mathbb{C}^{N\times T}$ is the matrix to be solved, in which only a few rows are the sparse signals we need.

Assuming that all the sensors exhibit independent and complex Gaussian distributed noise with a zero mean and a covariance of $\sigma^2\mathbf{I}$, we can calculate the likelihood of variable Y as Equation (2) based on (1).

$$p(\mathbf{Y}|\mathbf{X},\sigma^2) = \frac{1}{(\pi\sigma^2)^{MT}} \exp\{-\frac{1}{\sigma^2}\|\mathbf{Y}-\mathbf{\Phi X}\|_{\mathcal{F}}^2\} \tag{2}$$

The prior of $\mathbf{X}$ is given by Gaussian distribution:

$$p(\mathbf{X};\boldsymbol{\alpha}) = \prod_{n=1}^{N} \frac{1}{\pi\alpha_n} \exp\{-\frac{\mathbf{X}_n^2}{\alpha_n}\} \tag{3}$$

where the $\mathbf{X}_n$ values are independent of each other and assigned a Gaussian distribution with mean zero and variance α_n.

Our objective is to recover $\mathbf{X}$; therefore, the next task is to provide the posterior probability of $\mathbf{X}$. According to Bayes rules, the posterior pdf (probability density function) for $\mathbf{X}$ can be derived as

$$p(\mathbf{X}|\mathbf{Y};\boldsymbol{\alpha},\sigma^2) = \frac{p(\mathbf{Y}|\mathbf{X};\sigma^2)p(\mathbf{X};\boldsymbol{\alpha})}{p(\mathbf{Y};\boldsymbol{\alpha},\sigma^2)} \tag{4}$$

in which the two terms in the numerator on the right-hand side can be found in Equations (2) and (3). The denominator can be obtained from the definition of marginal probability. The conditional probability of $\mathbf{Y}$ given parameters $\boldsymbol{\alpha}$ and σ^2 is obtained by integrating the product of Equations (2) and (3) with respect to $\mathbf{X}$. Hence,

$$\begin{aligned} p(\mathbf{Y};\boldsymbol{\alpha},\sigma^2) &= \int p(\mathbf{Y}|\mathbf{X};\boldsymbol{\alpha}^2)p(\mathbf{X};\boldsymbol{\alpha})d\mathbf{X} \\ &= \frac{1}{\pi^N|\mathbf{\Sigma}_y|} \exp\{-\mathbf{X}^H\mathbf{\Sigma}_y^{-1}\mathbf{X}\} \end{aligned} \tag{5}$$

where $\mathbf{\Sigma}_y = (\sigma^2\mathbf{I} + \mathbf{\Phi\Lambda\Phi}^H)$.

Substituting Equations (2), (3) and (5) into (4), we obtain

$$p(\mathbf{X}|\mathbf{Y};\boldsymbol{\alpha},\sigma^2) = \mathcal{CN}(\mathbf{X}|\,\boldsymbol{\mu}_x,\mathbf{\Sigma}_x) \tag{6}$$

with mean $\boldsymbol{\mu}_x$ and covariance $\mathbf{\Sigma}_x$:

$$\boldsymbol{\mu}_x = \mathbf{\Lambda\Phi}^H\mathbf{\Sigma}_y^{-1}\mathbf{Y} \tag{7}$$

$$\mathbf{\Sigma}_x = \mathbf{\Lambda} - \mathbf{\Lambda\Phi}^H\mathbf{\Sigma}_y^{-1}\mathbf{\Phi\Lambda} \tag{8}$$

The Maximum-A-Posteriori (MAP) of $\mathbf{X}$ can be regarded as $\boldsymbol{\mu}_x$. Thence, we can obtain the recovered signal $\mathbf{X}$ after estimating the parameters $\boldsymbol{\alpha}$ and σ^2. In the literature [6], the authors provide a summary of SBL algorithmic models using Gaussian distribution as a sparse non-informative prior. Due to different derivation methods for parameter updates, it can be classified as the EM-based M-SBL algorithm as well as the SBL and SBL1 algorithms which directly take the derivative of the likelihood function. They are summarized in Algorithm 1.

From Algorithm 1, it is evident that in SBL and SBL1, the iteration formula for α_i^{new} involves square root and division operations, while the update formula for $(\sigma^2)^{new}$ involves computing the pseudo-inverse matrix. Compared to the update formulas for the two parameters in M-SBL, the operations in SBL and SBL1 are more complex and not suitable for implementation on FPGA. Therefore, this paper will use M-SBL as an example to introduce the hardware implementation process of the algorithm. It should be noted that the acceleration processing framework described later in this paper is equally applicable to SBL and SBL1.

Algorithm 1: SBL Algorithm

Input : $\mathbf{Y}$, $\boldsymbol{\Phi}$

Initilize : $\boldsymbol{\alpha} = \mathbf{1}$, $\sigma^2 = 10^{-3}$, $\varepsilon_{\min} = 10^{-3}$, and $iter_{\max} = 500$

while $\varepsilon > \varepsilon_{\min}$ *and* $iter < iter_{\max}$ **do**

$$iter = iter + 1,\ \boldsymbol{\alpha}^{old} = \boldsymbol{\alpha}^{new}$$

$$\boldsymbol{\Sigma}_y = \sigma^2\mathbf{I} + \boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^H$$

$$\boldsymbol{\Sigma}_x = \boldsymbol{\Lambda} - \boldsymbol{\Lambda}\boldsymbol{\Phi}^H\boldsymbol{\Sigma}_y^{-1}\boldsymbol{\Phi}\boldsymbol{\Lambda}$$

$$\boldsymbol{\mu}_x = \boldsymbol{\Lambda}\boldsymbol{\Phi}^H\boldsymbol{\Sigma}_y^{-1}\mathbf{Y}$$

$$\alpha_i^{new} = \begin{cases} \frac{1}{T}\|(\boldsymbol{\mu}_x)_{i:}\|_2^2 + (\boldsymbol{\Sigma}_x)_{ii} & \text{(M-SBL)} \\ \frac{1}{\sqrt{T}}\|(\boldsymbol{\mu}_x)_{i:}\|_2 / \sqrt{\boldsymbol{\phi}_i^H\mathbf{S}_\mathbf{y}^{-1}\boldsymbol{\phi}_i} & \text{(SBL)} \\ \frac{1}{\sqrt{T}}\|(\boldsymbol{\mu}_x)_{i:}\|_2 / \sqrt{\boldsymbol{\phi}_i^H\boldsymbol{\Sigma}_y^{-1}\boldsymbol{\phi}_i} & \text{(SBL1)} \end{cases}$$

$$(\sigma^2)^{new} = \begin{cases} \frac{\frac{1}{T}\|\mathbf{Y}-\boldsymbol{\Phi}\boldsymbol{\mu}_x\|_\mathcal{F}^2 + (\sigma^2)^{old}\left(N - \sum_{i=1}^N \frac{(\boldsymbol{\Sigma}_x)_{ii}}{\alpha_i}\right)}{M} & \text{(M-SBL)} \\ \frac{1}{M-K}tr\left((\mathbf{I}_N - \boldsymbol{\Phi}_\mathcal{M}\boldsymbol{\Phi}_\mathcal{M}^+)\mathbf{S}_y\right) & \text{(SBL/SBL1)} \end{cases}$$

$$\varepsilon = \left\|\boldsymbol{\alpha}^{new} - \boldsymbol{\alpha}^{old}\right\|_1 / \left\|\boldsymbol{\alpha}^{old}\right\|_1$$

end

Output : $\boldsymbol{\alpha}$, σ^2, $\boldsymbol{\mu}_x$

Where :

$iter_{\max}$: maximum iteration number

$\boldsymbol{\Lambda}$: diagonal matrix of $\boldsymbol{\alpha}$

$(\boldsymbol{\mu}_x)_{i:}$: i-th row of matrix $\boldsymbol{\mu}_x$

$\boldsymbol{\phi}_i$: i-th column of matrix $\boldsymbol{\Phi}$

$\mathbf{S}_\mathbf{y}$: The true covariance matrix of $\mathbf{Y}$

K: number of sparsity

$\boldsymbol{\Phi}_\mathcal{M}$: the active set of $\boldsymbol{\Phi}$

2.2. Matrix Inversion

2.2.1. LDL Factorization

It is evident that the variable $\boldsymbol{\Sigma}_y$ in Algorithm 1 is a Hermitian matrix, i.e., the diagonal elements are positive real numbers, and the other elements are conjugate symmetric along the diagonal. For a non-singular Hermitian matrix $\mathbf{A} \in \mathbb{C}^{N\times N}$, the most commonly known method for matrix inversion is based on Cholesky inversion, which involves Cholesky decomposition, triangular matrix inversion, and matrix multiplication. The Cholesky

decomposition can be expressed in the form of Equation (9), where $\mathbf{L}_1$ represents a complex lower triangular matrix with positive real diagonal elements.

$$\mathbf{A} = \mathbf{L}_1 \mathbf{L}_1^H \tag{9}$$

Cholesky decomposition has a computational complexity that is only half of LU decomposition. However, it involves square root operations. To avoid square root extraction, a closely related variant of Cholesky decomposition is LDL decomposition. By introducing an additional diagonal matrix D, we can use almost the same algorithm to compute and utilize the LDL decomposition, as shown in Equation (10).

$$\begin{aligned} \mathbf{A} &= \mathbf{L}_1 \mathbf{L}_1^H = \mathbf{L}\mathbf{D}^{1/2}(\mathbf{L}\mathbf{D}^{1/2})^H \\ &= \mathbf{L}\mathbf{D}\mathbf{L}^H \end{aligned} \tag{10}$$

where the diagonal elements of matrix $\mathbf{L}$ are all equal to 1, and matrix $\mathbf{D}$ is a diagonal matrix with positive real elements.

To solve for the elements in $\mathbf{L}$ and $\mathbf{D}$, we can express matrix $\mathbf{A}$ in the following form:

$$\mathbf{A} = \begin{bmatrix} a_{11} & \mathbf{b}^H \\ \mathbf{b} & \mathbf{A}_{N-1} \end{bmatrix} \tag{11}$$

where $\mathbf{b} \in \mathbb{C}^{(N-1)\times 1}$ is a column vector and $\mathbf{A}_{N-1} \in \mathbb{C}^{(N-1)\times(N-1)}$ is the sub-matrix. Therefore, the LDL factorization can be derived as:

$$\begin{bmatrix} a_{11} & \mathbf{b}^H \\ \mathbf{b} & \mathbf{A}_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \mathbf{s} & \mathbf{L}_{N-1} \end{bmatrix} \begin{bmatrix} d_{11} & 0 \\ 0 & \mathbf{D}_{N-1} \end{bmatrix} \begin{bmatrix} 1 & \mathbf{s}^H \\ 0 & \mathbf{L}_{N-1}^H \end{bmatrix} \tag{12}$$

Expanding the above equation is not difficult to obtain

$$d_{11} = a_{11} \tag{13}$$

$$\mathbf{s} = \mathbf{b}/a_{11} \tag{14}$$

From Equations (13) and (14), the first diagonal element of matrix $\mathbf{D}$ and the first column element of matrix $\mathbf{L}$ are solved. $\mathbf{A}_{N-1,new} = \mathbf{L}_{N-1}\mathbf{D}_{N-1}\mathbf{L}_{N-1}^H = \mathbf{A}_{N-1} - a_{11}ss^H$ remains a Hermitian matrix. We can continue to solve it using the above method like in [19], or we can directly expand Equation (10) as follows:

$$a_{ij} = \sum_{k=1}^{N} \left[l_{ik} d_{kk} l_{jk}^* \right] \quad (1 \le i,\, j \le N) \tag{15}$$

$$= \sum_{k=1}^{j-1} \left[l_{ik} d_{kk} l_{jk}^* \right] + l_{ij} d_{jj} \quad (1 < j \le i \le N) \tag{16}$$

The transformation of Equation (15) to (16) is due to the fact that matrix $\mathbf{L}$ is a lower triangular matrix with all diagonal elements equal to 1. Therefore, it is sufficient to calculate only the lower part of $\mathbf{L}$ excluding the diagonal elements.

By setting $i = j$ in Equation (16), we can obtain the updated Formula (17) for the other diagonal elements of matrix $\mathbf{D}$. Subsequently, we can derive the updated Formula (18) for the other elements in matrix $\mathbf{L}$.

$$d_{jj} = a_{jj} - \sum_{k=1}^{j-1} \left[l_{jk} d_{kk} l_{jk}^* \right] \quad (1 < j \le N) \tag{17}$$

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} \left[l_{ik} d_{kk} l_{jk}^* \right]}{d_{jj}} \quad (1 < j < i \le N) \tag{18}$$

Equations (13) and (17) are used together to solve for **D**, and Equations (14) and (18) are used together to solve for **L**. In Section 3.2, we will present a parallel computational structure for Equations (17) and (18).

2.2.2. Triangular Matrix Inversion

After LDL factorization, the inverse of matrix **A** becomes apparent, as shown in Equation (19).

$$\mathbf{A}^{-1} = (\mathbf{L}^{-1})^{H}\mathbf{D}^{-1}\mathbf{L}^{-1} \tag{19}$$

Since **D** is a diagonal matrix of real numbers, obtaining its inverse is straightforward by taking the reciprocal of each element. Therefore, our focus is on finding the inverse of **L**. There are several methods that can be employed, including equation solving, forward substitution, and block sub-matrix inversion. Regardless of the chosen method, the process of finding the inverse involves strong data dependencies. In this article, we will utilize the forward substitution method and provide a parallel implementation structure.

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & & & \vdots \\ l_{31} & l_{32} & 1 & & 0 \\ \vdots & & & \ddots & 0 \\ l_{N1} & l_{N2} & l_{N3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} l_{11}^{-1} & 0 & 0 & \cdots & 0 \\ l_{21}^{-1} & l_{22}^{-1} & & & \vdots \\ l_{31}^{-1} & l_{32}^{-1} & l_{33}^{-1} & & 0 \\ \vdots & & & \ddots & 0 \\ l_{N1}^{-1} & l_{N2}^{-1} & l_{N3}^{-1} & \cdots & l_{NN}^{-1} \end{bmatrix} = \mathbf{I} \tag{20}$$

Expanding the equation $\mathbf{L}\mathbf{L}^{-1} = \mathbf{I}$, we obtain Equation (20), where l_{ij}^{-1} represents the element in the i-th row and j-th column of matrix $\mathbf{L}^{-1}$. Continuing the calculation, let us take the first column of matrix $\mathbf{L}^{-1}$ multiplied by **L** as an example, and we obtain:

$$\begin{cases} l_{11}^{-1} = 1 \\ l_{21}^{-1} = -l_{21} \\ l_{31}^{-1} = -l_{31} - l_{32}l_{21}^{-1} \\ \quad \vdots \\ l_{N1}^{-1} = -l_{N1} - l_{N2}l_{21}^{-1} - \cdots - l_{N(N-1)}l_{(N-1)1}^{-1} \end{cases} \tag{21}$$

Based on the similarity of the results of other elements in $\mathbf{L}^{-1}$ and the form shown in Equation (21), we can summarize the formula for solving $\mathbf{L}^{-1}$ as:

$$\begin{cases} l_{jj}^{-1} = 1 & (1 \le j \le N) \\ l_{ij}^{-1} = -\sum_{k=j}^{i} l_{ik}l_{kj}^{-1} & (1 \le j < i \le N) \end{cases} \tag{22}$$

In Section 3.2, we will also present a parallel computational structure for Equation (22).

3. Proposed Hardware Architecture

This section consists of three parts: the top-level design approach for the hardware implementation of the SBL algorithm, the implementation architecture for LDL matrix inversion, and the optimized complex matrix multiplication.

3.1. Top-Level Description

By analyzing the computation process of the SBL algorithm, it becomes evident that in each iteration, the computational process executed is exactly the same, which means the operations are repeatable and there is a possibility for pipeline implementation. However, because the output of the previous iteration is required at the beginning of the next iteration, there is a data dependency that makes the actual computation process resemble a closed-loop finite state machine, which prevents the implementation of a pipeline. Even though

it is possible to optimize the internal computational process of each iteration by using parallel computing, vectorization, or other techniques to improve the execution speed of individual iterations, it is still not possible to fundamentally enhance the throughput rate and processing efficiency of the data from input to output.

Improving the data throughput on FPGA requires increasing the utilization of resources, including time and area resources. Returning to Algorithm 1 with M-SBL, if we treat each equation in the algorithm as a subtask, the workload of these subtasks remains unchanged regardless of the received data. This means that the area resources can be reused. However, when one subtask is executing, other subtasks are in a waiting state because they need to wait for the completion of the previous subtask to receive the data for processing. Clearly, this is a waste of time resources.

We can easily come up with the idea of using a pipeline to fully utilize each subtask and save time resources. However, before a set of data iterations is completed, it is always this set of data that flows through the subtasks, creating a closed loop that cannot be pipelined. Consequently, we formulated the subsequent concept: exclusively deploying the subtasks within the iteration body onto FPGA for computation while entrusting the management of data input and iteration termination assessment to the external CPU, as depicted in Figure 1. This breaks the original closed loop. After all the computations in FPGA are completed, the result is output to ARM. The output results can be temporarily stored in a memory through a demultiplexer and determined by the CPU whether it still needs iteration. Then, we use the CPU to control its re-entry into FPGA through the multiplexer at the appropriate time. Multiple sets of data can be stored in memory, corresponding to multiple subtasks in FPGA. Therefore, each task can process different data at the same time, achieving task-level pipelining. The ZYNQ development board is able to assist us in implementing the aforementioned idea. The pseudocode for this approach is shown in Algorithm 2.

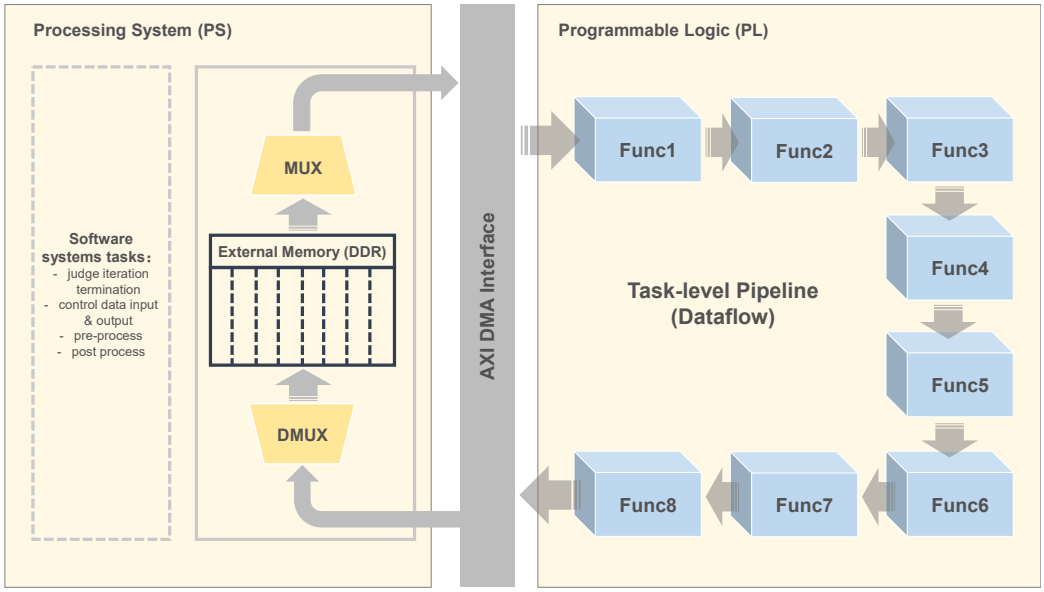


Figure 1. Overall architecture of the proposed HW&SW co-design task-level pipeline implementation system. Note: PL (Programmable Logic) refers to the FPGA side, while PS (Processing System) refers to the processing system side independent of the FPGA. MUX: Multiplexer. DMUX: Demultiplexer.

The "PS part" represents the ARM side in the ZYNQ development board, while the "PL part" represents the FPGA side. The algorithm is developed in C++ using the Vitis HLS 2022.2 platform.

We will provide a detailed explanation of Algorithm 2, beginning from the hardware part implemented on the FPGA.

Algorithm 2: Task-level pipelined M-SBL Algorithm

PS part :
Initilize : $\boldsymbol{\alpha}_{Memory} = \mathbf{1}$, $\sigma^2_{Memory} = 10^{-3}$, $\varepsilon_{Memory} = \mathbf{1}$, $iter_{Memory} = \mathbf{0}$, $\varepsilon_{\min} = 10^{-3}$, $iter_{\max} = 500$
while 1 **do**
  for $n = 1 : size(\mathbf{Y}_{Memory})$ **do**
    $\varepsilon = \varepsilon_{Memory}(n)$
    $iter = iter_{Memory}(n)$
    if $\varepsilon < \varepsilon_{\min}$ *or* $iter > iter_{\max}$ **then**
      Output : $\boldsymbol{\alpha}_{Memory}(n)$, $\sigma^2_{Memory}(n)$
      $\mathbf{Y}_{Memory}(n) = \mathbf{Y}_{new}$
      $\boldsymbol{\alpha}_{Memory}(n) = (1-\eta)\boldsymbol{\alpha}_{Memory}(n) + 0.01\eta\mathbf{1}$
      $iter_{Memory}(n) = 0$
    end
    $\boldsymbol{\alpha} = \boldsymbol{\alpha}_{Memory}(n)$
    $\sigma^2 = \sigma^2_{Memory}(n)$
    $\mathbf{Y} = \mathbf{Y}_{Memory}(n)$

    PL part :
    Input : $\mathbf{Y}$, $\boldsymbol{\alpha}$, σ^2
    # pragma HLS dataflow
      Func1 :
        $\boldsymbol{\Sigma}_y = \sigma^2\mathbf{I} + \boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^H$
        $Copy\ \mathbf{Y}$, $\boldsymbol{\alpha}$, σ^2
      Func2 :
        $\mathbf{L}$, $\mathbf{D} = LDL^T(\boldsymbol{\Sigma}_y)$
        $Copy\ \mathbf{Y}$, $\boldsymbol{\alpha}$, σ^2
      Func3 :
        $\mathbf{L^{-1}}$, $\mathbf{D^{-1}} = inverse(\mathbf{L}, \mathbf{D})$
        $Copy\ \mathbf{Y}$, $\boldsymbol{\alpha}$, σ^2
      Func4 :
        $\boldsymbol{\Sigma}_y^{-1} = (\mathbf{L^{-1}})^{\mathbf{T}}\mathbf{D^{-1}}\mathbf{L^{-1}}$
        $Copy\ \mathbf{Y}$, $\boldsymbol{\alpha}$, σ^2
      Func5 :
        $\boldsymbol{\Psi} = \boldsymbol{\Phi}^H\boldsymbol{\Sigma}_y^{-1}$
        $Copy\ \mathbf{Y}$, $\boldsymbol{\alpha}$, σ^2
      Func6 :
        $\mathbf{temp} = diag(\boldsymbol{\Psi}\boldsymbol{\Phi})$
        $diag(\boldsymbol{\Sigma}_x) = \boldsymbol{\alpha} - \mathbf{temp}.*(\boldsymbol{\alpha}^2)$
        $Copy\ \mathbf{Y}$, $\boldsymbol{\alpha}$, σ^2, $\boldsymbol{\Psi}$
      Func7 :
        $\boldsymbol{\mu}_x = \boldsymbol{\Lambda}\boldsymbol{\Psi}\mathbf{Y}$
        ${\alpha_i}^{new} = \frac{1}{T}\|(\boldsymbol{\mu}_x)_{i:}\|_2^2 + diag(\boldsymbol{\Sigma}_x)_i,\ \forall i$
        $Copy\ \mathbf{Y}$, $\boldsymbol{\alpha}$, σ^2, $diag(\boldsymbol{\Sigma}_x)$
      Func8 :
        $(\sigma^2)^{new} = \frac{\frac{1}{T}\|\mathbf{Y}-\boldsymbol{\Phi}\boldsymbol{\mu}_x\|_{\mathcal{F}}^2 + (\sigma^2)^{old}\left(N - \sum_{i=1}^{N}\frac{diag(\boldsymbol{\Sigma}_x)_i}{\alpha_i}\right)}{M}$
    Output : $\boldsymbol{\alpha}^{new}$, $(\sigma^2)^{new}$

    $\varepsilon_{Memory}(n) = \left\|\boldsymbol{\alpha}^{new} - \boldsymbol{\alpha}^{old}\right\|_1 / \left\|\boldsymbol{\alpha}^{old}\right\|_1$
    $iter_{Memory}(n) = iter + 1$
    $\boldsymbol{\alpha}_{Memory}(n) = \boldsymbol{\alpha}^{new}$
    $\sigma^2_{Memory}(n) = (\sigma^2)^{new}$
  end
end

The part between the two horizontal lines represents the processing flow in the FPGA. The inversion of the LDL matrix is divided into three subtasks Func2–4. The $\mathbf{\Phi}^H\mathbf{\Sigma}_y^{-1}$ operation is treated as a independent subtask because its result is reused. The $\boldsymbol{\mu}_x$ and $\boldsymbol{\alpha}$ computations are combined into one subtask. Finally, the M-SBL algorithm is divided into eight subtasks based on the functionality and complexity of each subtask: Func1 to Func8.

In each subtask, apart from their respective tasks, there is an additional *Copy* operation. This is due to the strict single-producer-single-consumer principle in implementing the task-level pipeline. It means that each parameter can only be produced by one task and consumed by the immediately following task. It is not allowed to supply a parameter to two tasks or skip intermediate subtasks and supply it to later tasks. The purpose of the *Copy* operation is to address this constraint, effectively encapsulating each subtask as an independent black box.

In the Vitis HLS tool, there are two implementation approaches for task-level pipelining. One involves structuring each subtask with a "for" loop, while the other entails representing each subtask as a function. Considering the complexity of our subtasks, it is evident that the latter is more suitable for implementation. Therefore, we encapsulate each subtask into an independent function. Then, in the top-level function, when applying the HLS directive "#pragma HLS dataflow", the HLS tool automatically recognizes the individual functions and implements task-level pipelining between them.

When transferring data between subtasks, Vitis HLS may employ ping-pong RAM or FIFO, depending on its ability to determine whether the data are sequentially accessed. As our data transmissions primarily involve arrays, and configuring FIFO depth may lead to issues, we manually configure the use of ping-pong RAM here through the "config_dataflow".

As for the implementation of individual subtasks, as evident from Algorithm 2, tasks other than LDL decomposition and inverting triangular matrices mainly involve basic matrix multiplication and fundamental arithmetic operations. These tasks can be implemented using basic "for" loops. During "for" loops, adjusting resource usage and latency can be achieved by setting appropriate parameters for loop pipeline intervals or loop unrolling factors.

Now, we will provide an explanation of the software part that has been implemented on the ARM processor.

In the ARM memory on the device, there is a temporary storage area for various parameters, i.e., $\boldsymbol{\alpha}_{Memory}$, σ^2_{Memory}, ε_{Memory}, $iter_{Memory}$. Firstly, the memory for these parameter values is initialized. In the subsequent loop, it can be observed that the "while" loop is the outer loop of the "for" loop, and the "for" loop acts as a counter, continuously selecting different data inputs with varying identifiers. This approach differs from the conventional iteration, where a set of data is computed repeatedly before moving on to the next set of data inputs. By reversing the order of these two loops, we break the original closed loop structure, laying the foundation for task-level pipelining to simultaneously process multiple sets of data in an interleaved manner.

Next, we retrieve the data to be processed from each memory block. It is important to note that there is a termination condition judgment for the iteration, which originally follows the "while" statement but has been moved here. When a set of data satisfies the termination condition, we first output the required parameters. Then, we extract a new $\mathbf{Y}$ from the data stream to fill in the $\mathbf{Y}_{Memory}(n)$ occupied by this set of data. The new $\boldsymbol{\alpha}$ will include the $\boldsymbol{\alpha}$ result of the completed data as part of the prior, which corresponds to the sequential estimation strategy we mentioned before. The weight of the previous iteration estimate is given by $(1-\eta)$, where η can take the value of 0.2. The term $0.01\eta\mathbf{1}$ is used to prevent the new initial iteration value of $\boldsymbol{\alpha}$ from being 0.

After processing in Func1 to Func8 on the PL side, termination condition ε can be calculated, along with other parameters, which are stored in their respective memories.

3.2. Implementation of LDL Matrix Inversion

3.2.1. Implementation of LDL Factorization

To fully leverage the parallel computing capabilities of FPGA, we designed parallel operation structures for Equations (17) and (18), as illustrated in Figures 2 and 3, respectively. The structures of the two are similar because the main computation of both involves a summation from $k = 1$ to $j - 1$. We did not implement a pipeline for this summation because the upper limit $j - 1$ is a variable, and variable bounds cannot be easily pipelined or fully unrolled. Therefore, we added a conditional operation before each unrolled computation to check the value of j and determine whether to execute the subsequent computation. This approach allows for minimal cost full unrolling. Although it increases the usage of multiplier and other components, it reduces latency. The subsequent summations can be unrolled accordingly.

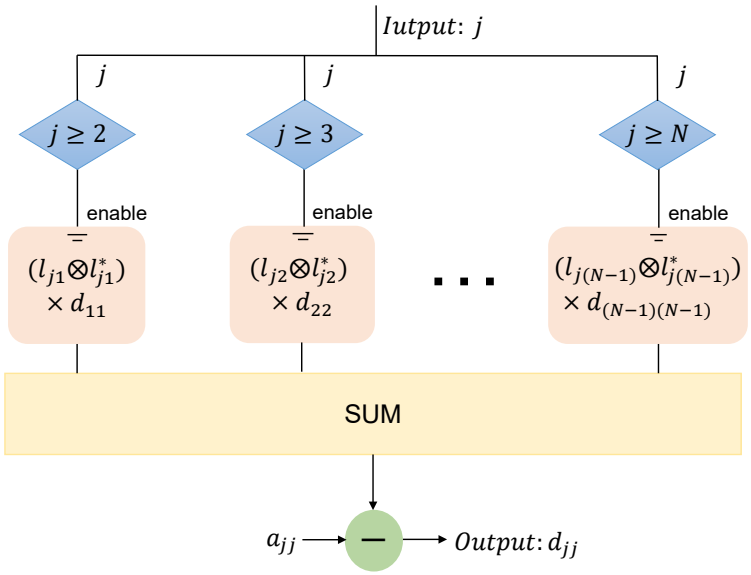


Figure 2. Parallel structure for solving D in LDL factorization. (Note: $\otimes$ specifically refers to optimized complex number multiplication, and $\times$ represents real number multiplication. The rules remain the same thereafter.)

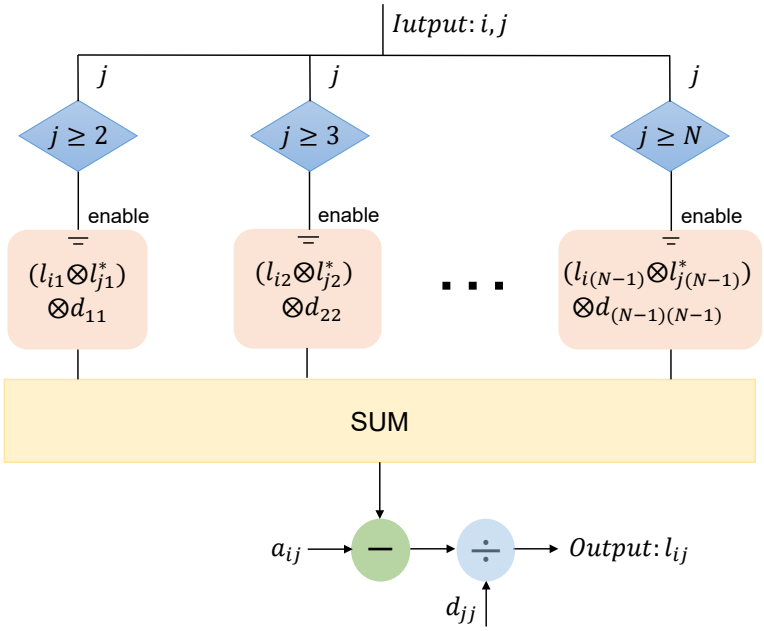


Figure 3. Parallel structure for solving L in LDL factorization.

Apart from using parallel structures internally, the outer loops for i and j can be pipelined. However, it is important to note that the principles of LDL factorization result in data dependencies between elements. For example, computing l_{32} requires the use of l_{31}, which means that l_{31} needs to be calculated before it can be involved in the computation of l_{32}. Therefore, the data initial interval (II) in the pipeline will be relatively large.

3.2.2. Implementation of Triangular Matrix Inversion

The parallel operation structure for Equation (22) is depicted in Figure 4. Since both the lower limit j and the upper limit i are variables, two conditional operations are required to check the values of i and j separately. The results of the two judgments are then connected through an AND gate to jointly determine whether to execute the subsequent computation. As mentioned before, when employing a pipeline externally, there will be a relatively large initial interval due to the same reasons.

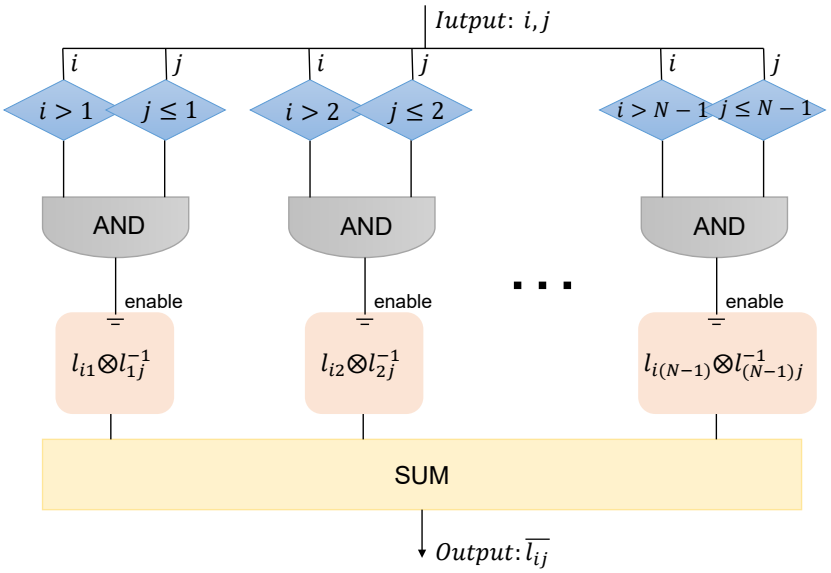


Figure 4. Parallel structure for lower triangular matrix inversion.

3.3. Implementation of Complex Multiplication

As is commonly known, DSPs (Digital Signal Processors) are precious resources in FPGA designs due to their limited availability. These resources are primarily utilized for demanding numerical operations, such as multiplication. Conversely, addition and subtraction operations are generally implemented using extensive lookup tables, which are numerous compared to DSP. Traditionally, one complex multiplication of $(a + bi) \times (c + di) = Re + Imi$ requires four multipliers and two adders because

$$\begin{aligned} Re &= a \times c - b \times d \\ Im &= a \times d + b \times c \end{aligned} \tag{23}$$

To economize DSP resources, we have employed an optimized architecture for complex multiplication:

$$\begin{aligned} A &= (a + b) \times d \\ B &= (b - a) \times c \\ C &= (c + d) \times a \\ Re &= C - A \\ Im &= B + C \end{aligned} \tag{24}$$

As can be seen, the optimization technique changes the resource usage to three multipliers and five adders, resulting in a remarkable 1/4 DSP resource saving during large-scale computations. Although this optimized approach introduces a slight increase in latency, this increase becomes nearly negligible when deploying a pipelined design and dealing with a high volume of computations like matrix multiplication. Overall, our optimized complex multiplication technique presents an efficient solution to save the limited DSP resources in FPGA designs.

4. Hardware Implementation

In this section, we will introduce the hardware development approach employed in this research. Subsequently, we will optimize the results for two different data scales and compare them with existing works.

In the past, the specialized nature of FPGA design required dedicated FPGA engineers, leading to a high barrier to entry. Fortunately, in recent years, tools like HDL Coder [21] and Vitis HLS [22] have emerged, enabling the conversion of Matlab code or C/C++ code into register transfer level (RTL) languages and assisting synthesis and simulation. While automatically generated code may be more redundant and slightly less computationally efficient compared to hand-written Verilog or VHDL code, it is essential to consider that creating complex algorithmic RTL code manually demands an enormous amount of work from hardware engineers. Therefore, these tools have bridged the gap between software engineers and hardware engineers and dramatically accelerated the process of developing and optimizing architectures on FPGA targets. Thus, in this study, we will leverage the Vitis HLS 2022.2 tool to achieve high throughput implementation of the SBL algorithm on FPGA through C/C++ to Verilog conversion. The main development process includes the following steps:

1. C/C++ Simulation;
2. C/C++ Synthesis;
3. C/C++ and RTL Co-Simulation;
4. Export RTL Design;
5. Hardware Implementation.

For task-level pipelines mentioned in last section, it is common for each subtask to have different latencies. One major factor that affects the performance of a task-level pipeline is the maximum subtask latency as it determines the initial interval of each subtask after implementation. This phenomenon is analogous to the "weakest link" or "bottleneck" effect in a barrel. Therefore, our optimization goal is to minimize the maximum subtask latency while ensuring that the latencies of other subtasks are close to this latency. This strategy allows us to achieve the same throughput performance with minimal resource consumption.

In fact, Vitis HLS can automatically assist us in achieving the most optimizations. However, for the specific goals mentioned above, certain details still require us to manually add directives. Apart from using "Dataflow" in the top function, the most frequently used directives in the sub-functions include "PIPELINE", "UNROLL", "LOOP FLATTEN", "ARRAY PARTITION", and others. The data type adopts a 48-bit word length fixed-point number. The evaluation board adopts ZCU102 based on the Zynq® UltraScale+™ XCZU9EG-2FFVB1156E MPSoC. As a typical application of SBL, we conduct experiments using DOA estimation as an example, which finds wide engineering use in many areas such as radar, sonar, communication, and so on. The results will now be presented using two different data scales.

4.1. Condition A: Number of Sensors M = 8, Number of Angles N = 90, Number of Snapshots T = 8

After optimization, the delays, intervals, and resource usage of the top-level function and each sub-module are displayed in Table 1. The functions are listed in sequential order from Func1 to Func8, which are named based on their respective functionalities. First looking at the delays, the LDL decomposition has the highest delay, which is 1322 cycles. The delays of other tasks are smaller than this value. Consequently, the initial interval for

all subtasks, including the top-level function, is $1322 + 1 = 1323$ cycles. The addition of 1 is because after completing a task on a particular pipeline, a function cannot immediately start executing the next task. It requires waiting for one clock cycle.

Table 1. Latency, interval and resource utilization for each module in condition A. (BRAM_18K: 18K bit Block RAM, DSP: Digital Signal Processing slice, FF: Flip-Flop, LUT: Look-Up Table).

Modules	Latency (Cycles)	Interval	BRAM_18K	DSP	FF	LUT
Top-Function	8974	1323	148	1112	45,076	77,927
cal_C	896	1323	10	193	1546	7890
LDLT	1322	1323	8	86	18,522	28,892
Linverse	961	1323	0	189	1443	3068
Linv_mul	1054	1323	0	70	402	1430
PhiCinv	1224	1323	8	144	1298	4362
cal_Sigma	1273	1323	4	31	398	926
cal_alpha	1270	1323	0	252	1460	4331
cal_sigma2	967	1323	12	147	11,449	14,211

Actually, contemporary FPGAs can perform floating-point operations, and we initially implemented our solution in this manner. However, upon converting to fixed-point representation, we observed that it allowed for lower resource utilization and reduced latency. Simultaneously, the error remained within an acceptable range. Therefore, after experimenting with various fixed-point number lengths and comparing precision, we ultimately chose to implement the solution using a 48-bit fixed-point representation. The resource and latency comparison under condition A for both approaches is presented in Table 2.

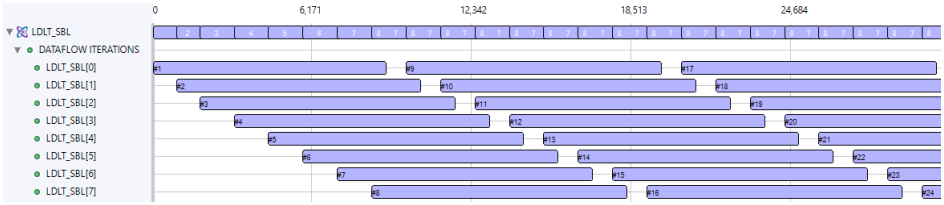
Table 2. Latency, interval, and resource utilization for different floating-point and 48-bit fixed-point data types in condition A.

Data Type	Latency (Cycles)	Interval	BRAM_18K	DSP	FF	LUT
floating-point	23,189	6278	328 (17.98%)	1695 (67.26%)	163,953 (29.91%)	185,652 (67.74%)
48-bit fixed-point	8974	1323	148 (8.11%)	1112 (44.13%)	45,076 (8.22%)	77,927 (28.43%)
2× fixed-point modules	18,388	1327	378 (20.72%)	1962 (77.86%)	71,226 (12.76%)	153,630 (56.05%)

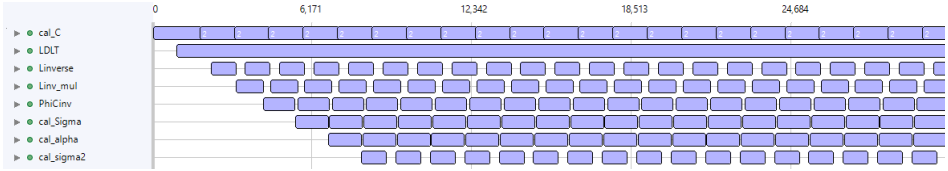
Figure 5 illustrates the timeline of (a) data processing and (b) subtask processing within the FPGA. In (a), each row represents a set of data, and each set of data executes its iterations along the timeline. Simultaneously, other sets of data do not need to wait for the completion of the previous data but can be executed in an interleaved manner based on time intervals. This forms a task-level pipeline. The fundamental reason for this can be understood from (b). Each row in (b) represents the execution status of a subtask, and it is evident that each subtask processes different data in an interleaved manner, maximizing the utilization of time resources.

After synthesis and place-and-route, the clock frequency has been achieved at 110 MHz. Using the given data, we can calculate the FPGA data throughput as:

$$\begin{aligned} Throughput &= \frac{Clock\ frequency}{Interval} \\ &= \frac{110\ \text{MHz}}{1323} = 83.2\ \text{kHz} \end{aligned} \tag{25}$$



(**a**)



(**b**)

Figure 5. (**a**) Data processing timeline trace and (**b**) subtask run timeline trace in condition A.

Assume that under the sequential estimation strategy, the average number of iterations required for each set of data is 150. Considering the effect of pipelining, the FPGA is processing eight sets of data simultaneously, which means that within a very short time after the first set of data processing is completed ($7 \times Interval$), the other seven sets of data will also complete their iterations. Therefore, the data output rate can be calculated using the following equation:

$$\begin{aligned} Output &= \frac{Data\ throughput}{Iterations} \times Number\ of\ Modules \\ &= \frac{83.2\ \text{kHz}}{150} \times 8 = 4.44\ \text{kHz} \end{aligned} \tag{26}$$

On the ZYNQ Ultrascale+ ZCU102 platform, the resource utilization on the FPGA is also shown in Table 2. Considering that resource utilization is consistently below 50%, replicating all modules could be a viable option to enhance throughput. We implemented this idea to demonstrate its feasibility, and the results are presented in the last row of Table 2. Although the resource utilization is not precisely twice that of the pre-replication state, the data input interval remains nearly unchanged. This validates that this operation can approximately double the throughput. The feasibility of this operation may vary depending on the available resources on different platforms and can be adjusted flexibly accordingly.

4.2. Condition B: Number of Sensors M = 16, Number of Angles N = 512, Number of Snapshots T = 8

To compare with the existing work, we have set up the same simulation conditions as presented in reference [17], which is a FPGA implementation of the SBL1 algorithm in Algorithm 1. Note in [17] that the number of snapshots processed in a single iteration is $T = 64$. However, in our case, we have set $T = 8$. Nonetheless, since we can simultaneously process eight groups of data, it is effectively equivalent to $T = 64$. Our results are displayed in Table 3, and the comparative results are displayed in Table 4.

From the comparison results shown in Table 4, it can be observed that our implementation method, compared to [17], employed fewer BRAM, FF, and LUT resources since we reduced the scale of computations in a single iteration. However, to enhance parallelism and minimize intervals, we opted to utilize additional DSP resources when they were available.

Furthermore, considering the time utilization, the main factor that affects throughput is the data interval. Due to the utilization of task-level pipelining, our initial interval is two orders of magnitude smaller than [17]. Therefore, even though our achieved clock frequency of 100 MHz is lower than their 150 MHz, our achievable throughput can reach

$100M/(9730 \times 8) = 1284.7$ Hz according to Equation (25), while the extra division by 8 is to ensure fairness in comparison with T = 64. Finally, our result is 30 times higher than their result of 43 Hz.

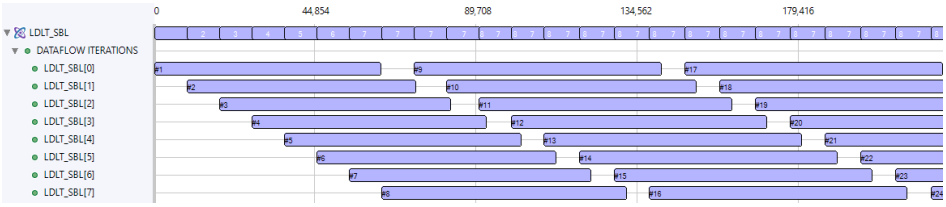
Figure 6 illustrates the timeline of data processing and subtask processing in condition B, similar to Figure 5 in condition A. The power consumption under condition A is 2.4 W, while under condition B, it is 3.0 W. In our study, when parameters deviate from the two specified conditions, the outcomes vary. When synthesized directly, the HLS tool will automatically propose optimization solutions, although they may not be the most optimal. It is essential to readjust optimization directives to achieve the best results. The above project code is uploaded in link https://github.com/WJinyang/FPGA-implementation-of-SBL (accessed on 1 January 2024).

Table 3. Latency, interval, and resource utilization for each module in condition B.

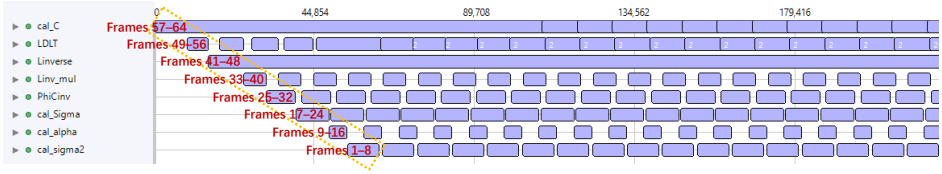
Modules	Latency (Cycles)	Interval	BRAM_18K	DSP	FF	LUT
Top-Function	62,603	9730	572 (31.36%)	1970 (78.17%)	73,148 (13.34%)	146,080 (53.30%)
cal_C	8987	9730	162	384	2719	31,779
LDLT	6002	9730	8	181	35,433	45,098
Linverse	9729	9730	0	378	2845	5325
Linv_mul	6404	9730	0	70	423	1692
PhiCinv	8194	9730	16	288	2512	10,673
cal_Sigma	9225	9730	0	30	286	11,620
cal_alpha	5130	9730	3	468	2420	7657
cal_sigma2	8925	9730	36	171	12,442	14,853

Table 4. Comparision of our result with existing works under the same conditions.

Modules	FPGA Type	Latency (Cycles)	Interval	BRAM	DSP	FF	LUT	Frequency	Throughput
SBL1 [17]	Kintex-7 XC7K325T	$> 10^6$	$> 10^6$	890	840	407,600	203,800	150 MHz	43 Hz
Our M-SBL	Zynq Ultrascale+ ZCU102	$\approx 6.26 \times 10^4$	9730	572	1970	73,148	146,080	100 MHz	1284.7 Hz



(**a**)



(**b**)

Figure 6. (**a**) Data processing timeline trace and (**b**) subtask run timeline trace in condition B.

5. Performance Analysis of DOA Estimation

To evaluate our method, we continue to adopt the application scenario of DOA estimation. We test the performance of the proposed approach from the following three aspects: computational precision, number of iterations, and comparison across different computational platforms.

We assume the reception of signals using an $M = 8$ elements uniform linear array with a measurement angle range between $[0^\circ, 180^\circ]$. The angular spacing between measurements is 2°, resulting in a total of $N = 90$ measurement angles. Therefore, the measurement matrix takes the form of an array manifold matrix, where the element in the m-th row and n-th column is denoted as $a_m(\theta_n) = e^{i2\pi d \cos\theta_n}$ (d is element interval). Assume there are two target sources located at 60° and 88°, respectively. For the multi-snapshot processing, we set the number of snapshots to $T = 8$. The received signals $\mathbf{Y}$ are normalized to the range $[-1, 1]$ for both the real and imaginary parts. Noise $\mathbf{E}$ is Gaussian white noise. The above conditions correspond to condition A in Section 4, thus achieving the same hardware resource utilization results as in condition A.

Since FPGAs are better at fixed-point arithmetic, for most variables, we utilize 48-bit fixed-point signed numbers, with 14 integer bits, 33 fractional bits, and 1 sign bit. Adopting the fixed-point data type not only helps conserve storage area resources but also allows us to transform some floating-point division operations into efficient shift operations, thus saving computational resources.

5.1. Fixed-Point Precision Test

Firstly, we validate the precision difference between our 48-bit fixed-point results and floating-point results. The value of $\boldsymbol{\alpha}$ can be used as a measure of signal power, representing the DOA estimation results. Under a signal-to-noise ratio (SNR) of 20 dB, we conduct 100 Monte Carlo experiments and calculate the absolute error and relative error between the floating-point and fixed-point results. The results are presented in Figure 7, where the absolute error and relative error are computed as follows:

$$Absolute\ error = floating\text{-}point - fixed\text{-}point\ results \tag{27}$$

$$Relative\ error = \frac{Absolute\ error}{floating\text{-}point\ results} \tag{28}$$

Figure 7a,b display the results of a particular experiment, representing the floating-point results obtained in MATLAB and the fixed-point results obtained in the FPGA, respectively. It is evident that the two results are nearly identical, accurately locating the positions of the target sources. Moving on to the analysis of (c) absolute error and (d) relative error, both plots show the mean values of 100 experimental results. Regarding absolute error, the largest values occur at the angles of the two target sources, 60° and 88°, with absolute values approaching 3×10^{-5}. However, in the case of relative error results, the lowest points appear at 60 and 88 degrees, while the highest relative error values are observed at other angles, peaking at around 3.5%.

These results can be attributed to the nature of the sparse Bayesian learning algorithm, which leverages the sparsity of signals. The algorithm yields large α values at certain points, while the rest of the results tend toward zero, leading to sparsity in the outcome. Consequently, both the floating-point and fixed-point results exhibit larger α values at 60° and 88° while being considerably smaller at other points, resulting in a higher absolute error at these two angles. Therefore, due to the inherently larger α values as denominators, at 60° and 88°, the relative error is lower compared to the other angles where α values are already quite small, resulting in higher relative errors. Overall, considering both absolute and relative errors, all results fall within an acceptable range, validating the accuracy and reliability of the fixed-point estimation method.

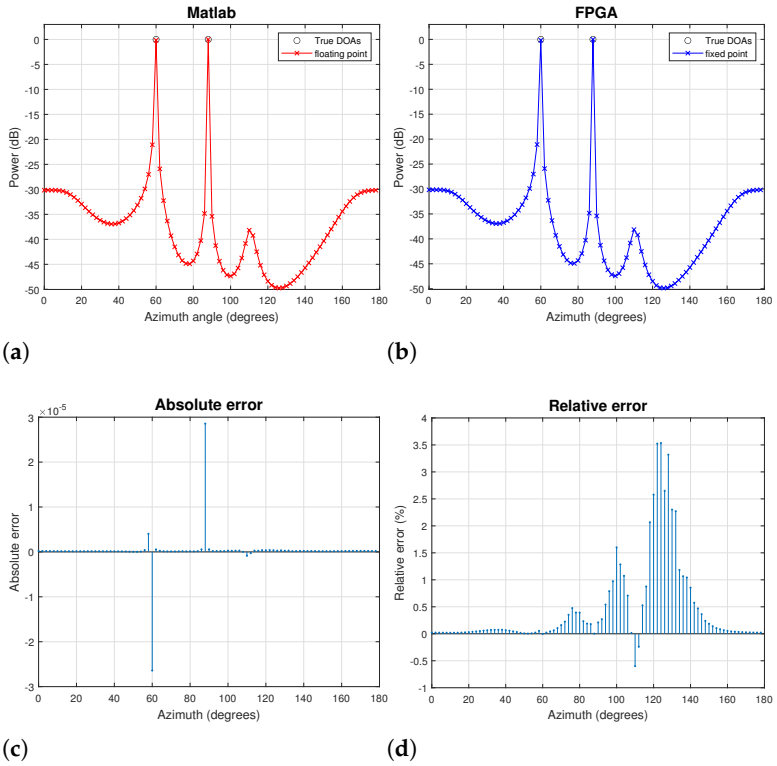


Figure 7. DOA estimation results and the calculation errors. (**a**) MATLAB result. (**b**) FPGA result. (**c**) Absolute error. (**d**) Relative error.

5.2. Number of Iterations Test

Next, we tested whether the M-SBL algorithm's iteration numbers significantly reduced under the sequential estimation strategy. Within the SNR range of 0 to 20 dB, with a 0.5 dB interval, we conducted 100 Monte Carlo experiments at each SNR level. We recorded the algorithm's iteration count both with and without using the sequential estimation strategy. The results are presented in Figure 8.

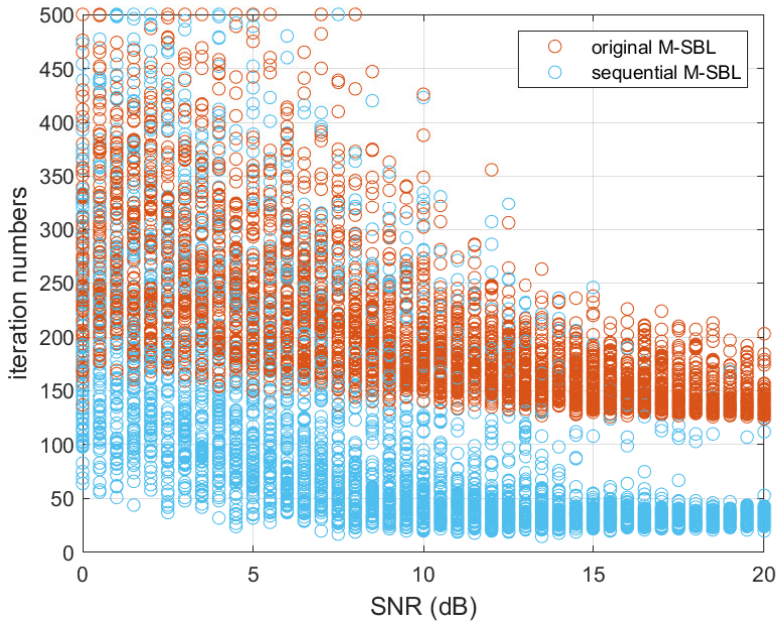


Figure 8. Comparison of algorithm's iteration numbers.

From Figure 8, we observe that as the SNR increases, the iteration numbers for both algorithms gradually decreases. However, when considering the overall trend, the algorithm with the sequential estimation strategy significantly reduces the iteration numbers compared to the one without it. By calculating the mean iteration numbers for the two algorithms, we found that the original algorithm's mean count was 207, while the sequential estimation algorithm's mean count was 106. Consequently, we conclude that under the sequential estimation strategy, the algorithm's iteration count can be reduced by 50% approximately.

5.3. Comparison across Different Computational Platforms

Finally, we compare the computational efficiency on different platforms under identical conditions, namely Matlab R2021b on a Intel(R) Core(TM) i5-10500 CPU at 3.10GHz, C++ on a Intel(R) Core(TM) i5-10500 CPU at 3.10GHz, and ZYNQ UltraScale+ MPSoC ZCU102.

After conducting 1000 independent repeated experiments at an SNR of 20 dB, the results are obtained through the average execution time and shown in Table 5. From the results, it can be observed that as a computational tool and programming language, Matlab, despite its internal matrix operation optimizations, still exhibits slightly slower execution speeds compared to lower-level languages like C++. Nevertheless, the most noteworthy aspect is that the FPGA's computational speed significantly surpasses both Matlab and C++, which demonstrates the superiority of our implementation method.

Table 5. Comparision of throughput (Hz) on different platforms under the same conditions.

Matlab	C++	FPGA
3310	3870	83.2 k

Based on the results from Section 4 and this section, we can conclude that our SBL ZYNQ implementation method significantly improves computational efficiency while ensuring a certain level of calculation accuracy.

6. Conclusions

In this paper, we achieved accelerated implementation of the sparse Bayesian learning algorithm using the ZCU102 evaluation board for rapid prototyping based on the Zynq® UltraScale+™ XCZU9EG-2FFVB1156E MPSoC (multiprocessor system-on-chip). To address challenges such as strong data correlation, matrix inversion, matrix multiplication, and high iteration numbers, we proposed task-level pipelining, LDL matrix inversion, optimized complex multiplication structure, and sequential estimation strategies to solve the above difficulties successfully. We conducted hardware implementations for two different data sizes, in one of which our results demonstrate a speed improvement of 30 times compared to existing approaches. Finally, we validated the practical engineering feasibility of our method by applying it to DOA estimation. Due to space constraints, we plan to conduct future implementations and comparisons of other compressed sensing algorithms. This will allow us to explore the performance and efficiency of different algorithms in FPGA environments, providing valuable insights for real-world applications.

Author Contributions: Conceptualization, J.W. (Jinyang Wang); methodology, E.-B.B.; validation, J.W. (Jinyang Wang); formal analysis, J.W. (Jun Wang), E.-B.B. and M.M.; writing—original draft preparation, J.W. (Jinyang Wang); writing—review and editing, E.-B.B., M.M., J.W. (Jun Wang), C.L., L.W. and Y.T.; supervision, E.-B.B. and H.W.; funding acquisition, F.Y. and C.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by China Scholarship Council (No.202002527014).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are contained within the article.

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

References

1. Donoho, D.L. Compressed sensing. *IEEE Trans. Inf. Theory* **2006**, *52*, 1289–1306. [CrossRef]
2. Tipping, M.E. Sparse Bayesian learning and the relevance vector machine. *J. Mach. Learn. Res.* **2001**, *1*, 211–244.
3. Palmer, J.; Rao, B.; Wipf, D. Perspectives on sparse Bayesian learning. *Adv. Neural Inf. Process. Syst.* **2003**, *16*, 249–256 .
4. Tropp, J.A.; Gilbert, A.C. Signal recovery from random measurements via orthogonal matching pursuit. *IEEE Trans. Inf. Theory* **2007**, *53*, 4655–4666. [CrossRef]
5. Roth, V. The generalized LASSO. *IEEE Trans. Neural Netw.* **2004**, *15*, 16–28. [CrossRef] [PubMed]
6. Gerstoft, P.; Mecklenbräuker, C.F.; Xenaki, A.; Nannuru, S. Multisnapshot sparse Bayesian learning for DOA. *IEEE Signal Process. Lett.* **2016**, *23*, 1469–1473. [CrossRef]
7. Tipping, M.E.; Faul, A.C. Fast marginal likelihood maximisation for sparse Bayesian models. In Proceedings of the International Workshop on Artificial Intelligence and Statistics. PMLR, Key West, FL, USA, 3–6 January 2003; pp. 276–283.
8. Zhou, W.; Zhang, H.T.; Wang, J. An efficient sparse Bayesian learning algorithm based on Gaussian-scale mixtures. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *33*, 3065–3078. [CrossRef] [PubMed]
9. Babacan, S.D.; Molina, R.; Katsaggelos, A.K. Bayesian compressive sensing using Laplace priors. *IEEE Trans. Image Process.* **2009**, *19*, 53–63. [CrossRef] [PubMed]
10. Shutin, D.; Buchgraber, T.; Kulkarni, S.R.; Poor, H.V. Fast variational sparse Bayesian learning with automatic relevance determination for superimposed signals. *IEEE Trans. Signal Process.* **2011**, *59*, 6257–6261.
11. Tan, X.; Li, J. Computationally efficient sparse Bayesian learning via belief propagation. *IEEE Trans. Signal Process.* **2010**, *58*, 2010–2021. [CrossRef]
12. Al-Shoukairi, M.; Schniter, P.; Rao, B.D. A GAMP-based low complexity sparse Bayesian learning algorithm. *IEEE Trans. Signal Process.* **2017**, *66*, 294–308. [CrossRef]
13. Zhang, C.; Yuan, Z.; Wang, Z.; Guo, Q. Low complexity sparse Bayesian learning using combined belief propagation and mean field with a stretched factor graph. *Signal Process.* **2017**, *131*, 344–349. [CrossRef]
14. Asano, S.; Maruyama, T.; Yamaguchi, Y. Performance comparison of FPGA, GPU and CPU in image processing. In Proceedings of the 2009 International Conference on Field Programmable Logic and Applications, Prague, Czech Republic, 31 August–2 September 2009. IEEE: Piscataway, NJ, USA, 2009; pp. 126–131.
15. Yang, D.; Peterson, G.D.; Li, H. Compressed sensing and Cholesky decomposition on FPGAs and GPUs. *Parallel Comput.* **2012**, *38*, 421–437. [CrossRef]
16. Qasaimeh, M.; Denolf, K.; Lo, J.; Vissers, K.; Zambreno, J.; Jones, P.H. Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels. In Proceedings of the 2019 IEEE International Conference on Embedded Software and Systems (ICESS), Genoa, Italy, 27–29 November 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 1–8.
17. Groll, H.; Mecklenbräuker, C.; Gerstoft, P. Sparse Bayesian learning for directions of arrival on an FPGA. In Proceedings of the 2018 IEEE Statistical Signal Processing Workshop (SSP), Freiburg, Germany, 10–13 June 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 623–627.
18. Tesema, S.N.; Bourennane, E.B. Resource-and Power-Efficient High-Performance Object Detection Inference Acceleration Using FPGA. *Electronics* **2022**, *11*, 1827. [CrossRef]
19. Zhang, X.W.; Zuo, L.; Li, M.; Guo, J.X. High-throughput FPGA implementation of matrix inversion for control systems. *IEEE Trans. Ind. Electron.* **2020**, *68*, 6205–6216. [CrossRef]
20. Park, Y.; Meyer, F.; Gerstoft, P. Sequential sparse Bayesian learning for time-varying direction of arrival. *J. Acoust. Soc. Am.* **2021**, *149*, 2089–2099. [CrossRef] [PubMed]
21. Pant, H.; Bourai, H.; Rana, G.S.; Yadav, S. Conversion of MATLAB code in VHDL using HDL Coder & Implementation of Code on FPGA. *HCTL Open Int. J. Technol. Innov. Res. (IJTIR)* **2015**, *14*, 1–9.
22. Cong, J.; Lau, J.; Liu, G.; Neuendorffer, S.; Pan, P.; Vissers, K.; Zhang, Z. FPGA HLS today: Successes, challenges, and opportunities. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* **2022**, *15*, 1–42. [CrossRef]