FPGA Acceleration for Computational Glass-Free Displays

Zhuolun He* and Guojie Luo*[*][*]
Center for Energy-efficient Computing and Applications, School of EECS, Peking University, China*
Collaborative Innovation Center of High Performance Computing, NUDT, China[*]
PKU-UCLA Joint Research Institute in Science and Engineering[*]
{juicehe, gluo}@pku.edu.cn

ABSTRACT

The increasing computational power enables various new applications that are runtime prohibitive before. FPGA is one of such computational power with both reconfigurability and energy efficiency. In this paper, we demonstrate the feasibility of eyeglasses-free displays through FPGA acceleration. Specifically, we propose several techniques to accelerate the sparse matrix-vector multiplication and the L-BFGS iterative optimization algorithm with the consideration of the characteristics of FPGAs. The experimental results show that we reach a 12.78X overall speedup of the glass-free display application.

1. INTRODUCTION

FPGAs have been increasingly popular as accelerators for large scale compute-intensive applications because of their inherent parallelism. With massive on-chip logics and flexible on-chip memories, FPGAs can be expediently customized as high throughput, low latency computing systems. Nevertheless, the degree of computational parallelism is limited by either on-chip memory bandwidth or IO rate of external memory.

Light field, defined as a part of space studied from the standpoint of transmission of radiant energy within that space by Gershun [1], can now be recorded, manipulated and displayed with the advent of computers, color displays and inexpensive digital sensors [2]. While an image is a 2D slice of the 4D light field, computational light field display applications aim to construct a 4D light field from a set of 2D images of different angle's original view, where a significant amount of raw data is required. On the one hand, the compute-intensive nature of light field display applications makes them a particularly good fit for FPGA-based processing. On the other hand, relevant issues concerning the vast amount of data should be overcome to enhance the performance of the FPGA-based application.

We explore light field display application on FPGA. In particular, we make the following contributions:

1. We apply ideas of matrix compression and table look-up to minimize the data volume to transfer.
2. We search for the optimal partitioning factor of array to increase the concurrency of on-chip BRAM access.
3. We put forward an FPGA-friendly algorithm that effectively reduces both computational load and data transmission of the L-BFGS optimization method.

The remainder of this paper is organized as follows: Section 2 gives a brief introduction of our application and FPGA acceleration. Section 3 shows the overall structure of the system. Section 4 and 5 demonstrate our key techniques in SpMV and L-BFGS in detail. Section 6 shows our experimental results. Section 7 and 8 are related work and conclusions sections.

2. BACKGROUND

There are estimates that about 2.5 billion people (one-third of the world's population) could be affected by myopia by the end of this decade [3], and thus, they need eyeglasses or contact lenses to read or see clearly. Huang and Wetzstein [4] have introduced a computational display technology that predistorts the presented content for the observer so that the desired image is perceived without the need for eyewear. However, the long runtime of solving linear systems in the eyeglass-free display algorithm acts as a significant obstacle to process images at required rate. Therefore, we focus on the interesting and useful subject of accelerating the algorithm.

While section 2.1 gives a brief introduction of the algorithm, section 2.2 shows the time profiling. Section 2.3 discusses the basic idea of FPGA acceleration.

2.1 Light Field Reconstruction Problem

The goal of Huang and Wetzstein's algorithm is to present a 4D light field to the observer, such that a desired 2D retinal projection is perceived. Figure 1 demonstrates the application. Formally speaking, given the desired 2D image $u \in \mathbf{R}^M$ and the 2D-to-4D transformation matrix $P \in \mathbf{R}^{M \times N}$ encoding the projection of the screen-side light onto the retina, the light field reconstruction problem is to find a proper light field $x \in \mathbf{R}^N$ emitted by the display as follows:

$$\begin{array}{ll} \text{minimize} & f(x) = \|u - Px\|_2 \\ \text{subject to} & \mathbf{0} \leq x \leq \mathbf{1} \end{array} \tag{1}$$

Here, M is the discretized locations on the retina ($M = 128 \times 128$ in our program), and N is the number of emitted

FPGA '17, February 22-24, 2017, Monterey, CA, USA

DOI: http://dx.doi.org/10.1145/3020078.3021728

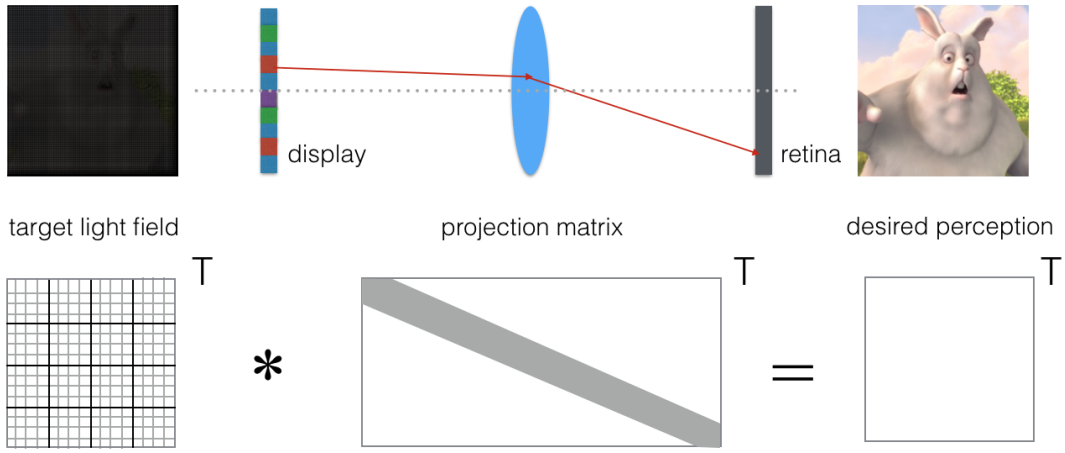

Figure 1: Glass-free Display Application

light rays ($N = 700 \times 700$ is applied for iPod Touch 4 according to Huang and Wetzstein).

The matrix P is constant under the rule of light field transformation [5], when viewing distance, pupil size, and other parameters are fixed. Therefore, the process of building the projection matrix P needs to be done merely once, and can be considered as pretreatment and input of the algorithm. In other words, for each input image, the only work of the algorithm is to calculate the light field by Equation 1.

Equation 1 can be solved using standard convex optimization algorithms, and we employ the L-BFGS algorithm [6] here. However, this is still of high calculation strength due to the high dimensionality of P and u.

2.2 The Algorithm and its Runtime Profiling

As mentioned above, since P can be treated as the input of the algorithm, we want to focus on the runtime of solving Equation 1 and other processes that need to be done respectively for each input image.

In a C++ prototype of the reconstruction algorithm[1], processing a 128×128 image needs approximately 124.5s, where L-BFGS accounts for 122.5s, namely 98.4% of runtime. The high percentage pushes us to concentrate on the acceleration of the L-BFGS algorithm.

Listing 1 is a rough outline of the L-BFGS algorithm. We timed respectively for the four primary procedures in L-BFGS algorithm in line 2-5, and the result is as Figure 2.

Besides, we also timed for the basic matrix-vector operations. The result is shown in Figure 3, from which we can see multiplication and inner product operations take up over 70% of runtime.

These profiling results indicate that the matrix-vector multiplication, the inner product, and the vector addition are the most time-consuming matrix-vector operations in the L-BFGS algorithm. They are the focus of acceleration.

[1]We test it using the same configuration as the baseline described in Section 6.1.

Listing 1 L-BFGS Algorithm Outline [7]

Input: starting point x_0, integer history size $m > 0$, $k = 0$;
Output: the position x with a minimal objective function

1: **while** not converge **do**
2: Calculate gradient $\nabla f(x_k)$ at position x_k;
3: Compute direction p_k using two-loop recursion as Listing 2;
4: Search for step length α_k which satisfies Wolfe conditions;
5: Update $x_{k+1} = x_k + \alpha_k p_k$, $k = k + 1$ and other information;
6: **end while**
7: **return** final x

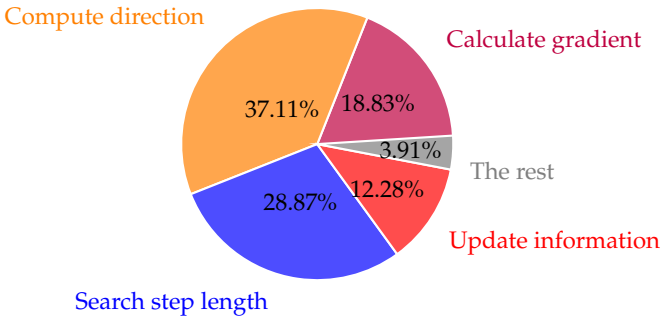

Figure 2: Time of Procedures (Totally 123.72s)

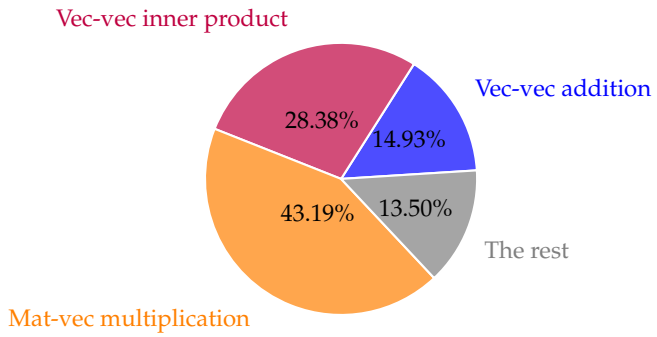

Figure 3: Time of Operations (Totally 122.48s)

2.3 FPGA Acceleration and Challenges

FPGA acceleration is a promising technique to implement energy-efficient and low-latency computational tasks. Let's take the computation of inner product as an instance here.

Ideally, the speed of inner product calculation is limited by the computational resources, assuming that on-chip memory is large enough to hold the two operand vectors. Suppose F_m floating-point multiplications and F_a floating-point additions are performed per second. For an inner product operation between two vectors in $\mathbf{R}^N$, totally N times multiplications and $N-1$ times additions are needed. Therefore, the sequential computation time $T_{comp} = N/F_m + (N-1)/F_a$. Thanks to the inherent parallelism structure of FPGA, we can directly implement a parallel design for inner product calculation. If the design consists of K multipliers and a reduction tree for summation, the computation time is reduced to $T_{comp} = N/(K \cdot F_m) + \log_2(N-1)/F_a$.

However, the limited on-chip block RAM (BRAM) imposes a barrier for exposing the peak computational performance of FPGAs. If memory bandwidth is denoted by B elements per second, the memory I/O time $T_{I/O} = 2N/B$. Then the total time for inner product takes $T \geq \max(T_{comp}, T_{I/O})$. Typically, $T_{I/O}$ is the bottleneck.

In this paper, we apply the idea of data compression and data reuse to reduce the requirement of data transfer and improve the performance.

3. OVERALL ACCELERATOR DESIGN

Figure 4 shows the overall structure of our system. The Flow Controller assembles the modules to form the application. The Memory Manager sees to BRAMs management and off-chip data transfer management. Mul-Add, SpMV, and InnerProduct are the processing modules. Whenever the control flow reaches the three operations, the Memory Manger is invoked to read required data from external memory and store them in on-chip BRAMs, after which the corresponding processing module starts computation. Results are written back to external memory if necessary.

4. SPARSE MATRIX-VECTOR MULTIPLICATION

In this sparse matrix-vector multiplication section, we primarily focus on the memory issues within the problem, which significantly affect the performance. Specifically, we reduced the data transfer time by compressing the sparse matrix size

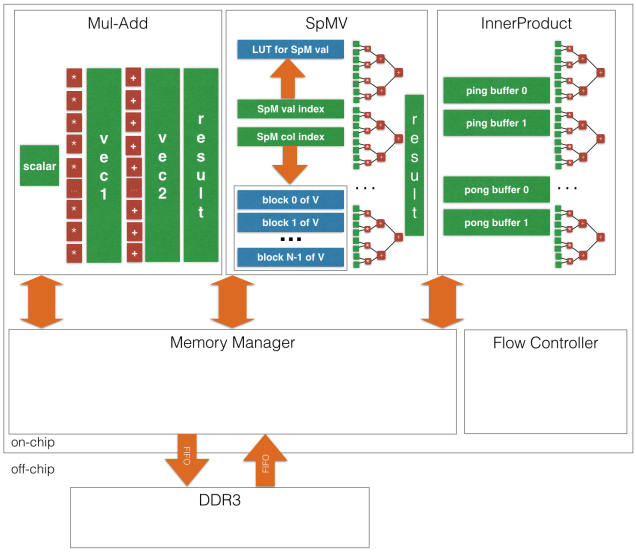

Figure 4: Overall Structure of Accelerator

(elaborated in section 4.1) and reduced the data access time by partitioning the vector (elaborated in section 4.2).

We have to emphasize here that our problem is not very similar to the traditional deconvolution problem, since the projection matrix (namely the sparse matrix here) is an analog result generated by the simulation of the projection process. That is to say, building the matrix involves analog phases, such as sampling and rounding, which makes the matrix anomalous and so that access pattern of the vector is not affine-access for the matrix.

4.1 Storage of the Sparse Matrix

4.1.1 *Bitwidth Reduction for COO and CRS*

Note that the projection matrix P is a large matrix with billions of elements, but only no more than a million of them are non-zero. Spontaneously, the matrix should be represented in a sparse format. And the simplest one, the coordinate list (COO), is to use a triplet $\langle (r,c), v \rangle$ to denote each non-zero element, where r and c are the row index and column index of an element and v is its corresponding value. Here, r and c are integers range from 0 to $M-1$ and 0 to $N-1$, and can be stored in a $\lceil \log_2 M \rceil$-bit integer and a $\lceil \log_2 N \rceil$-bit integer, respectively. v is a floating-point number range from 0-1, and takes up 32-bit width space as a single-precision floating-point number in IEEE 754. Suppose there are totally n_z non-zero elements in the matrix, the matrix can be stored in $(\lceil \log_2 M \rceil + \lceil \log_2 N \rceil + 32)n_z$ bits, namely $(\frac{1}{8}(\lceil \log_2 M \rceil + \lceil \log_2 N \rceil) + 4)n_z$ bytes space.

If all the non-zero elements in triplet are sorted in a row-major order and are placed together, we get a n_z-row table with three columns: row-index column, column-index column, and value column. Notice that the elements are sorted in row-major order, which means the elements from the same row are placed successively, and the duplicate row indices are redundant. Therefore, we can compress the row-index column. In this Compress Row Storage (CRS), the row-index column contains only M elements. The i^{th} element stores the row index in the table of the first non-zero element

in the i^{th} row. There are n_z non-zero elements so the row index of an entry is no more than $n_z - 1$, and thus can be store in $\lceil \log_2 n_z \rceil$ bits. With the column-index column and value column remained the same, the matrix can be stored in $M\lceil \log_2 n_z \rceil + \lceil \log_2 N \rceil + 32n_z$ bits, namely $\frac{M}{8}\lceil \log_2 n_z \rceil + (\frac{1}{8}\lceil \log_2 N \rceil + 4)n_z$ bytes.

4.1.2 Compression using Look-up Table of Values

We have different strategies to store and transfer the value column. As mentioned above, we need n_z 32-bit floating-point numbers to store the values directly, and that counts for $4n_z$ bytes storage space.

However, in the algorithm, the value is something like light intensity and is calculated as the number of samples falling on a discrete screen light field coordinate divided by total number of samples generated. Given that, since the total number is fixed, we can only store the number of samples for each coordinate, and calculates the actual value when necessary. According to the result of building projection matrix, the max number of samples is less than 600, and can be represented as a 10-bit integer, so we need only $1.25n_z$ bytes storage space, at the cost of an extra division for each value.

Moreover, we notice that there are only 350 different values, so the idea of the index table is under consideration here. We can use a 9-bit number to represent an unsigned integer less than 512, which means we reduce the storage space to $1.125n_z$ bytes for the values, but have to keep an index table in memory with a size of about 0.5 Kbytes (350×10 bits), and have to look up the index table and calculate for each value. The compressed row-index column and col-index column remain the same as that in section 4.1.1.

4.1.3 Overall Matrix Size Reduction

The effects of different storage methods are summarized in Table 1. Our CRS+LUT representation achieves a $1.81\times$ reduction in the matrix storage compared to the conventional CRS, and thus greatly relieve the bandwidth bottleneck of the sparse matrix-vector multiplication.

Format	Space complexity (bytes)	Storage (MB)
flat	$4MN$	32112.64
COO	$\frac{1}{8}\lceil \log_2 M \rceil n_z + (\frac{1}{8}\lceil \log_2 N \rceil + 4)n_z$	6.63
CRS	$\frac{M}{8}\lceil \log_2 n_z \rceil + (\frac{1}{8}\lceil \log_2 N \rceil + 4)n_z$	5.24
CRS+LUT	$\frac{M}{8}\lceil \log_2 n_z \rceil + (\frac{1}{8}\lceil \log_2 N \rceil + 1.125)n_z$	2.90

Table 1: Storage of the single-precision sparse matrix with $M = 16384$ rows, $N = 490000$ columns, and $n_z = 816272$ non-zero entries in our application. The CRS+LUT format has an overhead of 437.5 bytes for the look-up table (LUT).

4.2 Partitioning of the Vector

Typically, a block RAM in FPGA provides up to $16Kb$ storage size (if not using parity bits). In our problem, the vector consists of 490000 32-bit floating-point numbers, which takes 980 BRAMs to hold the whole vector. Due to the limited number of read ports of the BRAMs, the mapping from an array (e.g., a vector in SpMV) to the BRAMs is important to fetch enough data in every clock cycle during pipelining.

As we mentioned, the access pattern of the vector is irregular on the matrix rows, because the matrix simulates some analog processes. Therefore, conventional methods for stencil computation to partition the vector is no longer feasible. The good thing is that since the matrix is constant during computation and thus, the access pattern is statistically analyzable, it is possible to search for an optimal partitioning factor to reach a maximum access rate of the vector elements. With the sparse matrix donated as P and the vector donated as x, we divide the vector into N blocks to minimize the memory access latency. Formally, we search for such an N that:

$$\underset{N}{\operatorname{argmin}} \sum_{r=1}^{P.rows} \max_{b=1}^{N} acc_{r,b} \tag{2}$$

Here, $acc_{r,b} = k$ indicates that when we calculate the inner-product between the r^{th} row of P and x, there will be k read transactions for block b. Since different BRAMs can be read or written in parallel, the number of cycles for each row-vector multiplication is decided by the block with the most read transactions, and the equation 2 is rather straightforward.

We enumerate the factors of $|x|$ as potential partitioning factors, which is intuitive and avoids small storage fragmentation. We also try both block partition and cyclic partition. Considering both the search time and the BRAM resource limitation, we set an upper bound of 1500 for N during the enumeration. Table 2 shows part of the results. In this case, 980 is a perfect partitioning factor, which not only cause no conflict in memory access, but also brings about no storage fragmentation.

Factor N	Method	Min cyc/r	Max cyc/r	Total cyc
980	cyclic	1	1	16384
1225	cyclic	1	1	16384
1250	cyclic	1	2	19840
...	...	...	...	...
1400	block	4	18	188564
1250	block	5	18	193276
...	...	...	...	...
1	N/A	37	54	816272

Table 2: Partition Results of Various N

5. DECOMPOSED L-BFGS ALGORITHM

5.1 Vector-free L-BFGS

The major component of the overall algorithm is the iterative L-BFGS algorithm, where the computation of inner products consumes most of the time. The computational kernel of the L-BFGS algorithm is a two-loop recursion, as shown in Listing 2. The direction p_k depends on the the current gradient direction $\nabla f(x_k)$ and the last m updates $s_i = x_{i+1} - x_i$ and $y_i = \nabla f(x_{i+1}) - \nabla f(x_i)$ for $i = k - m, ..., k - 1$.

We adopt the vector-free L-BFGS algorithm [7] which can reduce both the efforts in computation and data transfer. The key idea is to represent the vector p_k as a linear combination of the precomputed inner products, as shown in figure 5. Specifically, let $b^k_j = s_{k-m-1+j}$, $b^k_{m+j} = y_{k-m-1+j}$ for

Listing 2 The original L-BFGS two-loop recursion [7]

Input: $\nabla f(x_k)$, s_i, y_i for $i = k-m, ..., k-1$
Output: new direction p_k

$p_k = -\nabla f(x_k)$
for $i = k-1$ **to** $k-m$ **do**
  $\alpha_i = \frac{s_i \cdot p_k}{s_i \cdot y_i}$
  $p_k = p_k - \alpha_i y_i$
end for
$p_k = \frac{s_{k-1} \cdot y_{k-1}}{y_{k-1} \cdot y_{k-1}} p_k$
for $i = k-m$ **to** $k-1$ **do**
  $\beta = \frac{y_i \cdot p_k}{s_i \cdot y_i}$
  $p_k = p_k + (\alpha_i - \beta) s_i$
end for

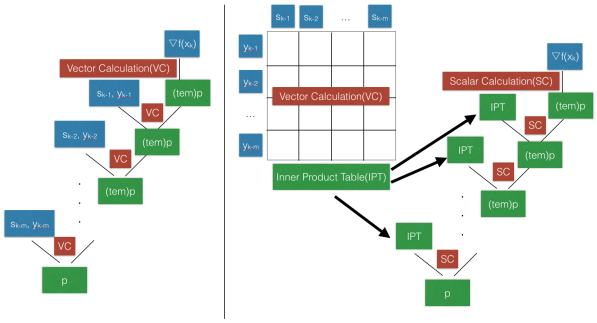

Figure 5: Two-loop recursion in the original L-BFGS (the left) and the vector-free L-BFGS (the right)

Listing 3 The vector-free L-BFGS two-loop recursion [7]

Input: inner product table $T^k[\,][\,]$ of size $(2m+1)^2$
Output: coefficients δ_i^k for $i = 1, 2, ..., 2m+1$

for $i = 1$ **to** $2m+1$ **do**
  $\delta_i^k = (i < 2m)\ ?\ 0 : -1$
end for
for $i = k-1$ **to** $k-m$ **do**
  $j = i - (k-m) + 1$
  $\alpha_i = \frac{\sum_{l=1}^{2m+1} \delta_l^k T^k[l][j]}{T^k[j][m+j]}$
  $\delta_{m+j}^k = \delta_{m+j}^k - \alpha_i$
end for
for $i = 1$ **to** $2m+1$ **do**
  $\delta_i^k = \frac{T^k[m][2m]}{T^k[2m][2m]} \delta_i^k$
end for
for $i = k-m$ **to** $k-1$ **do**
  $j = i - (k-m) + 1$
  $\beta = \frac{\sum_{l=1}^{2m+1} \delta_l^k T^k[m+j][l]}{T^k[j][m+j]}$
  $\delta_j^k = \delta_j^k + (\alpha_i - \beta)$
end for

$j = 1, 2, ..., m$, and $b_{2m+1}^k = \nabla f(x_k)$, we can write the direction p_k as $\sum_{i=1}^{2m+1} \delta_i^k b_i^k$. In this way, we can replace the vector computations in Listing 2 by the scalar computations in Listing 3.

Each time when executing Listing 3, we need a look-up table for the inner products among $\nabla f(x_k)$, s_i and y_i for $i = k-m, ..., k-1$. There are $2m+1$ vectors in total, so the table size is $(2m+1)^2$. The entry $T^k[i][j]$ stores the value of the inner product $b_i^k \cdot b_j^k$.

When iterating from k to $k+1$, we only need to update the look-up table by discarding the outdated vectors and inserting new entries. The new entries include the inner products between $\nabla f(x_{k+1})$, s_k, y_k and the remaining vectors. Noticing that $s_k = x_{k+1} - x_k$ and $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$, there are many computations that can be shared when computing the new inner products relating to s_k or y_k.

5.2 TESC for VL-BFGS

Here we put forward an efficient algorithm to update the look-up table in VL-BFGS called "Transfer Equation and Shared Computation" (TESC). Our algorithm is shown in Listing 4, and it has the minimum inner product computations so that it highly improves the performance of VL-BFGS.

TESC is based on the following two observations:

1. When entering an iteration, most of the vector entries are overlapped with the previous ones.

2. When finishing an iteration, $\nabla f(x_k)$ is the only vector that cannot be linearly represented by previous vectors.

From the first observation, we obtain the following transfer equation:

$$\begin{cases} b_i^k = b_{i+1}^{k-1}, & \text{for } i = 1, \ldots, m-1 \\ b_{m+i}^k = b_{m+i+1}^{k-1}, & \text{for } i = 1, \ldots, m-1 \end{cases} \tag{3}$$

From the second observation, we can conclude that the inner products involving the newly produced vector $\nabla f(x_k)$ are inevitable. In fact, with equation 3, the rest of the look-up table entries can be calculated from the previous look-up table and other scalars. For example:

$$\begin{aligned} b_m^k \cdot b_m^k &= (\alpha_{k-1} \sum_{i=1}^{2m+1} \delta_i^k b_i^{k-1}) \cdot b_m^k \\ &= \alpha_{k-1} \sum_{i=1}^{2m+1} (\delta_i^k b_i^{k-1} \cdot (\alpha_{k-1} \sum_{j=1}^{2m+1} (\delta_j^k b_j^{k-1}))) \\ &= \alpha_{k-1}^2 \sum_{i=1}^{2m+1} \sum_{j=1}^{2m+1} (\delta_i^k \delta_j^k b_i^{k-1} \cdot b_j^{k-1}) \\ &= \alpha_{k-1}^2 \sum_{i=1}^{2m+1} \sum_{j=1}^{2m+1} (\delta_i^k \delta_j^k T^{k-1}[i][j]) \end{aligned} \tag{4}$$

The expression to calculate $b_m^k \cdot b_m^k$ above consists of only scalar operations, so no direct inner product is needed here. Most of the derivations are evident, so we just omit them here. Some operations involve the two discarded vectors after the last iteration, so we just postpone discarding until finishing updating the look-up table.

Listing 4 shows the outline of TESC.

5.3 Complexity Analysis and Comparison

In TESC, with the history size m and the vector dimension d, the computational complexity of updating the inner product table for each iteration is $(2m+4)d$, compared to the origin $6md$ in VL-BFGS. Even better, the $2m+4$ inner product computations involve the same vector, which means we can

Listing 4 TESC for VL-BFGS

Input: previous look-up table $T^{k-1}[\,][\,]$, $\nabla f(x_k)$, and the relevant scalars and vectors
Output: new look-up table $T^k[\,][\,]$

```
for i = k − m to k − 1 do
    compute ∇f(x_k) · s_i and ∇f(x_k) · y_i
end for
compute ∇f(x_k) · ∇f(x_k) and ∇f(x_k) · ∇f(x_{k−1})
compute ∇f(x_k) · (s_{k−m−1}) and ∇f(x_k) · (y_{k−m−1})
for r = 1 to 2m + 1 do
    for c = 1 to r do
        Update T^k[r][c] using T^{k−1}[ ][ ], the inner product
        results above and the relevant scalars
    end for
end for
```

keep that vector in the BRAM on FPGA during the computations, and thus reduce the data transfer. We will discuss the details in the following paragraphs.

Originally, L-BFGS needs $2 \cdot (2m)$ inner product computations in its two-loop recursion, and another $(2m+2)d$ multiplications to calculate the new direction p_k. Therefore, the total complexity of the origin algorithm is $(6m+2)d$ multiplication for each iteration. Intuitively, $(6m+2)d$ multiplications bring about $(12m+4)d$ data transfer. However, since some data can be shared between the multiplications due to the certain calculation pattern here, data transfer can be lowered to $(8m+4)d$ per iteration.

For VL-BFGS, with the consideration of the commutative law of multiplication since $si \cdot yj \equiv yj \cdot si$, each new iteration only need to calculate $6m$ new dot products which involve new s_k, y_k and g_k. The other and the final step is to calculate the new direction p based on δ^k and the base vectors, i.e. $b_1^k, \ldots, b_{2m+1}^k$. The complexity is another $2md$ multiplications, which means the overall complexity of the algorithm is $8md$ multiplications. Similarly, the idea of data reuse can be applied here. For example, when we are calculating inner products related to new s_k, we simply have s_k stored in the block RAM of FPGA and fetch the other party of the product from on-board memory one by one. Therefore, data transfer can also be reduced to $8md$ per iteration.

Things get much more amazing when it comes to VL-BFGS with TESC. As mentioned above, the computational complexity of updating the inner product table is $2m+4$ inner products involving the same vector, so data transfer complexity is $2m+4$ here. With the addition of the calculation for the new direction p, the overall complexity is $(4m+4)d$ multiplications with $(4m+4)d$ data transfer.

Table 3 shows and compares the analysis results.

	Multiplication	Data transfer
L-BFGS	$(6m+2)d$	$(8m+4)d$
VL-BFGS	$8md$	$8md$
VL-BFGS with TESC	$(4m+4)d$	$(4m+4)d$

Table 3: Complexity Analysis and Comparison

6. EXPERIMENTAL EVALUATIONS

6.1 Experimental Setup

We tested our optimizations on VC707 evaluation board featuring the Virtex-7 XC7VX485T-2FFG1761C FPGA. Designs are implemented as IP cores with Vivado High-Level Synthesis v2015.2 and synthesized and place-and-route by Vivado v2015.2.

The Huang and Wetzstein's original algorithm was implemented in MATLAB. And we convert the original implementation into C++ as the baseline for the following comparisons. The baseline uses Eigen (a C++ template library for linear algebra) and CppNumericalSolvers (an L-BFGS implementation in C++11 based on Eigen) for the matrix-vector manipulation and optimization. We tested on a server with a 20-core Intel Xeon CPU E5-2630 v3 @ 2.30GHz and $64GB$ main memory, hereinafter inclusive.

6.2 Analysis of the SpMV Optimization

In section 4.1, we introduced the idea of matrix compression using bitwidth reduction and look-up table, which achieves a $2.28X$ reduction in storage compared with the conventional CRS representation. Naturally, it saves 56.1% data transfer time of the sparse matrix, and that is 43.0% of IO for each SpMV execution.

In section 4.2, we partitioned the vector with the optimal factor to achieve the maximum on-chip memory bandwidth. Besides, we also notice that there are approximately fifty to sixty non-zero elements in a sparse matrix row, and none of them holds more than 64 non-zeros. To simplify our design, we propagate a constant of 64 elements down the processing unit chain and the reduction tree each trip, and the accesses of corresponding vector elements cause no conflict because of the array partition.

To take full advantage of previous ideas, we first rewrite the original MATLAB-style SpMV operation (which looks like a dense matrix-matrix multiplication) as an FPGA-friendly one (which uses CRS and LUT as mentioned) in C++. The change greatly reduces the runtime of the CPU version application from over two minutes to 65.49 seconds.

We employ ping-pong buffer in our FPGA design, and the performance is entirely limited by the external memory bandwidth in this situation. Runtime comparison and source utilization are listed in section 6.4.

6.3 Analysis of the L-BFGS Enhancement

By adopting vector-free L-BFGS, as shown in Listing 3, inter-data dependencies between iterations are removed in the two-loop recursions of L-BFGS. Not only can we divide the operand vectors into blocks for parallel computing, but we may parallelize different inner-product operations. The greater granularity of parallelism is also applicable to FPGA clusters or distributed systems.

Listing 4 shows our TESC algorithm that updates the look-up table in VL-FBGS efficiently using transfer equation and shared computation. With TESC, the runtime of the CPU version decreases to around 47.47 seconds.

Before we load certain computations onto FPGA, we tested different history size m (see Listing 1), and we take $m = 3$ as the history size hyperparameter, which guarantees convergence and lowers calculation cost. In this case, the runtime is reduced to 25.26 seconds.

6.4 FPGA Speedup of each Component

With the consideration of the nature of the algorithm and the optimizations, we conclude three most basic operations in VL-BFGS with TESC:

1. Sparse matrix-vector multiplication (SpMV) ($vector_{res} = SpM * vector$)
2. Inner product of vectors (InnProd) ($scalar_{res} = vector_1 \cdot vector_2$)
3. Scalar multiplication and vector addition (Mul-Add) ($vector_{res} = scalar \cdot vector_1 + vector_2$)

It is reasonable to compare the runtime between FPGA implementation and multi-threading implementation of each operation. We use OpenMP to implement the multi-threading version from the C++ code, where Eigen is adopted as the linear algebra library, as mentioned above.

According to the analysis of VL-BFGS with TESC in section 5.3, there are $2m + 4$ consecutive inner products in the updating phase and $2m$ multiple-additions in the phase of calculating new direction p. Therefore, we test the two operations in loops that repeat them for corresponding times and add OpenMP directives to the loops. The SpMV operation will not be tested with repetitive computation, and we add the OpenMP directives to the for loops inside the operation.

Table 4 shows the runtime comparison of each component, while figure 6 illustrates the speedup comparison. Table 5 shows the resource utilizations of each component and the summation on FPGA.

	SpMV	InnProd	Mul-Add
FPGA	0.0085s*	0.0237s	0.0635s
1-thread	0.0312s	0.1247s	0.1920s
2-threads	0.0204s	0.1120s	0.1437s
4-threads	0.0121s	0.0741s	0.1091s

* estimated

Table 4: Runtime Comparison of each Component

	LUT	FF	BRAM	DSP
SpMV	8290	6038	1058	10
InnProd	21722	26372	39	0
Mul-Add	27834	40959	169	0
Total	57846	73369	1266	10
Available	303600	607200	2060	2800
Utilization(%)	19	12	61	∼0

Table 5: Resource Utilizations of each Component

6.5 The Overall Speedup and Discussions

Finally, we can obtain the overall speedup. The FPGA version takes totally 9.74 seconds to accomplish, which reaches a 12.78× speedup compared with the baseline. 3.64× speedup is closely FPGA-related, including L-BFGS enhancement and FPGA acceleration of some operations. L-BFGS enhancement contributes 1.38× to the speedup directly and more importantly, it enables the employment of FPGA. Yet our system didn't fully utilize the external memory bandwidth. Currently, the peak memory bandwidth in our experiment is less than $800MB/s$. Therefore, there is still considerable room for performance improvement.

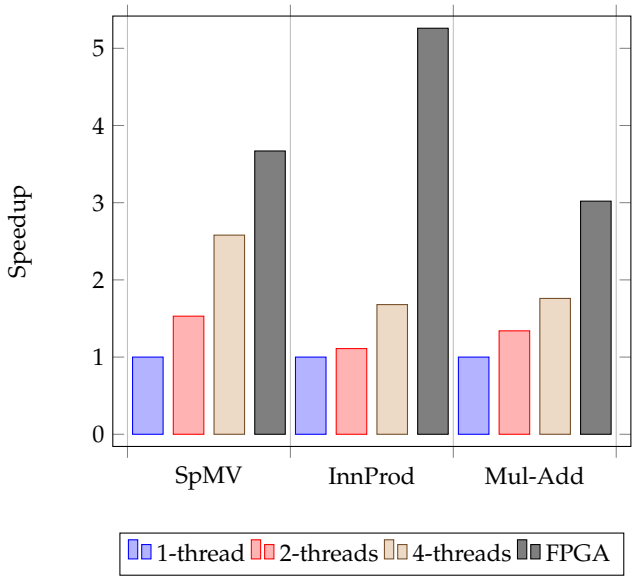

Figure 6: Speedup Comparison

7. RELATED WORK

Though our FPGA acceleration of the light field reconstruction problem is quite different from the previous problems, some challenges are solved by our novel techniques, which are partly inspired by the following works.

SpMV on FPGA The sparse matrix-vector multiplication is one of the most important kernels in scientific computing. P. Grigoras and et al. [8] have used the Bounded CSRVI Format to compress the sparse matrix values on CPU and decompress in runtime. Y. Umuroglu and M. Jahre [9] [10] have described a scalable backend architecture that exploits column-major traversal and interleaving to achieve high bandwidth utilization. They have also proposed a hardware-software caching scheme that exploits preprocessing to enable performant and area-effective SpMV acceleration. S. Guo and et al. [11] have presented a deeply-pipelined SpMV accelerator by exploiting a hardware-friendly storage scheme.

Memory Partition Memory partitioning can efficiently map data elements in the same logical array onto multiple physical banks so that the accesses to the array are parallelized. Y. Wang and et al. [12] have proposed a generalized memory-partitioning framework using a polyhedral model. J. Cong and et al. [13] have presented an automatic memory partitioning technique which can efficiently improve throughput and reduce the energy consumption of pipelined loop kernels for given throughput constraints and platform requirements.

System Optimization It is always important to balance the resource usage of different processing modules to reach better performance. P. Li and et al. [14] have developed an algorithm to determine the optimal resource usage and initiation intervals for each loop in the applications to achieve maximum throughput within a given area budget.

8. CONCLUSIONS AND FUTURE WORK

In our experiment, we achieve a $12.78X$ speedup for the light field reconstruction application using FPGA. On-chip memory bandwidth and IO rate turn out to be the greatest

limitation for FPGA performance, so we compress the matrix and partition the vector in SpMV operations to get better properties. Besides, we come up with a novel and efficient algorithm to update the look-up table in the vector-free L-BFGS.

In the future, we would like to explore real-time light field reconstruction technique with a more customized system. Besides, the system should apply to various viewing distance with eyeball tracking feature, which bound to be attractive and practical.

9. ACKNOWLEDGMENT

This work is partly supported by National Natural Science Foundation of China (NSFC) Grant 61520106004, Beijing Natural Science Foundation (BJNSF) Grant 4142022, and the Peking University Principal Undergraduate Research Foundation
(URTP2015PKU004).

10. REFERENCES

[1] A. Gershun, "The light field," *Journal of Mathematics and Physics*, vol. 18, no. 1-4, pp. 51–151, 1939.

[2] M. Levoy, "Light fields and computational imaging," *Computer*, vol. 39, no. 8, pp. 46–55, 2006.

[3] E. Dolgin, "The myopia boom," *Nature*, vol. 519, pp. 276–278, mar 2015.

[4] F. Huang and G. Wetzstein, "Eyeglasses-free display: towards correcting visual aberrations with computational light field displays," *ACM Transactions on Graphics (SIGGRAPH)*, pp. 1–12, 2014.

[5] F.-c. Huang, G. Wetzstein, and B. A. Barsky, "Supplementary Material : Eyeglasses-free Display : Towards Correcting Visual Aberrations with Computational Light Field Displays Derivation of the Light Field Transport,"

[6] B. J. Nocedal, "Updating Quasi-Newton Matrices With Limited Storage," vol. 35, pp. 773–782, 1980.

[7] W. Chen, Z. Wang, and J. Zhou, "Large-scale L-BFGS using MapReduce," in *Proceedings of Neural Information Processing Systems (NIPS 2014)*, 2014.

[8] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk, "Accelerating spmv on fpgas by compressing nonzero values," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, pp. 64–67, 2015.

[9] Y. Umuroglu and M. Jahre, "An energy efficient column-major backend for fpga spmv accelerators," in *IEEE International Conference on Computer Design*, pp. 432–439, 2014.

[10] Y. Umuroglu and M. Jahre, *A Vector Caching Scheme for Streaming FPGA SpMV Accelerators*. 2015.

[11] S. Guo, Y. Dou, Y. Lei, and G. Wu, "A deeply-pipelined fpga-based spmv accelerator with a hardware-friendly storage scheme," *Ieice Electronics Express*, vol. 12, no. 11, pp. 20150161–20150161, 2015.

[12] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *Acm/sigda International Symposium on Field-Programmable Gate Arrays*, pp. 199–208, 2014.

[13] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization.," in *IEEE/ACM International Conference on Computer-aided Design*, pp. 697–704, 2011.

[14] P. Li, P. Zhang, and J. Cong, "Resource-Aware Throughput Optimization for High-Level Synthesis,"