1 Introduction

Background subtraction is a critical step in many video surveillance applications including traffic monitoring, vandalism deterrence, and suspicious object detection. In this project, we devise an automated end-to-end system for FPGA-based deployment of background subtraction applied to intelligent video surveillance. Our system can take the stream of video frames captured by surveillance cameras as its input and adaptively highlights the foreground information existing in each frame to subsequently facilitate objection detection/tracking process. We leverage a greedy sparse-coding routine called Orthogonal Matching Pursuit (OMP) to transform the raw video frames to a new set of frames by eliminating the background pixels. Note that processing the foreground information is computationally more efficient which, in turn, makes it practical to realize object detection/tracking on resource-constrained SoC platforms available on today's surveillance cameras. As such, it evades the need to transfer large video collections to a control station for further processing by enabling on-chip data analysis.

2 Background

Background subtraction can be cast as a sparse signal reconstruction problem. In particular, for the given n_f^{th} video frame (Y_{n_f}), its background can be sparsely represented as the linear combination of a few samples in a given dictionary D that contains common background patches. To perform the background subtraction, we aim to solve the following objective function using OMP algorithm:

$$min_{V^*_{n_f}} \|Y_{n_f} - DV_{n_f}\|_2 \;\; s.t. \;\; \|V^*_{n_f}\|_0 \leq k. \quad (1)$$

OMP is a well-known routine for solving sparse approximation problems. It takes a dictionary D and a sample Y_{n_f} as inputs and iteratively approximates the sparse representation of the sample by adding the best fitting element in every iteration. Parameter k controls the total number of non-zeros per column of V (a.k.a., sparsity level). Suppose the dictionary matrix D contains a set of background patches, thus the background in each frame Y_{n_f} can be reconstructed using $DV^*_{n_f}$, where $V^*_{n_f}$ is the optimal solution per Eq. 2. Thus, the foreground information can be represented as $Y_{n_f} - DV^*_{n_f}$ which would be sparse after background subtraction.

Algorithm 1 outlines the pseudocode of OMP. Performing the OMP routine for a given signal requires three main steps: (i) Finding the best matching sample in the dictionary matrix D, (ii) Least-Square (LS) optimization, (iii) Residual update. Implementing OMP on reconfigurable platforms such as FPGA enables a low-power and high-throughput computation on constrained settings with real-time analytic requirement. We use floating-point operations in our realization of OMP algorithm. Use of floating-point computation assures a much larger dynamic range which is particularly important in providing a generic solution for processing dynamic data collections where the range may be unpredictable as data evolves over time. In particular, the variant nature of ultimate learning tasks, as well as the unpredictability of data range make it infeasible to mathematically bound the possible effect of fix-point computation in providing a generic solution that is directly applicable to different applications.

Algorithm 1 OMP Algorithm

```
    Inputs: Dictionary $D$, video frame $\mathbf{Y}_{n_f}$, sparsity $k$, threshold error $\epsilon$.
    Output: Coefficient vector v, support set $\Lambda$.
1:  $r^0 \leftarrow \mathbf{Y}_{n_f}$
2:  $\Lambda^0 \leftarrow \emptyset$
3:  while $i \leq k$ and $r > \epsilon$ do
4:      $\Lambda \leftarrow \Lambda \cup argmax_j| < r^{i-1}, \mathbf{D}_j > |$ Find best fitting column
5:      $v^i \leftarrow argmin_v \|r^{i-1} - \mathbf{D}_{\Lambda^i} v\|_2{}^2$ LS Optimization
6:      $r^i \leftarrow r^{i-1} - \mathbf{D}_{\Lambda^i} v^i$ Residual Update
7:      $i \leftarrow i + 1$
    end while
```

To boost the computational performance for analyzing a large amount of data on FPGA, it is necessary to modify the OMP algorithm such that it maximally benefits from the available resources and incurs a scalable computational complexity. The Least Squares (LS) minimization step in OMP algorithm involves a variety of operations with complex data flows that introduce an extra hardware complexity. However, proper use of factorization techniques like QR decomposition or Cholesky method within the OMP algorithm would reduce its hardware implementation complexity and make it well-suited for hardware accelerators [1].

To efficiently solve the LS optimization problem, we decide to use QR decomposition (Algorithm 2). QR decomposition returns an orthogonal matrix $\mathbf{Q}$ and an upper-triangular matrix $\mathbf{R}$. It iteratively updates the decomposition by reusing the $\mathbf{Q}$ and $\mathbf{R}$ matrices from the last OMP iteration. In this approach, the residual (line 6 of Algorithm 1) can be updated by $\mathbf{r}^i \leftarrow \mathbf{r}^{i-1}\mathbf{Q}^i(\mathbf{Q}^i)^t\mathbf{r}^{i-1}$. The final solution is calculated by performing back substitution to solve the inversion of the matrix $\mathbf{R}$ in $\mathbf{v}^k = \mathbf{R}^{-1}\mathbf{Q}^t\mathbf{Y}_{n_f}$.

Algorithm 2 Incremental QR decomposition by modified Gram-Schmidt

```
    Inputs: New column $\mathbf{D}_{\Lambda^s}$, last iteration $\mathbf{Q}^{s-1}$, $\mathbf{R}^{s-1}$.
    Output: $\mathbf{Q}^s$ and $\mathbf{R}^s$.
1:
        $$R^s \leftarrow \begin{pmatrix} R^{s-1} & 0 \\ 0 & 0 \end{pmatrix}$$
2:  $\xi^s \leftarrow \mathbf{D}_{\Lambda^s}$
3:  for $j$ = 1,...,s-1 do
4:      $\mathbf{R}_{js}{}^s \leftarrow (\mathbf{Q}^{s-1})_j{}^H \xi^s$
5:      $\xi^s \leftarrow \xi^s - \mathbf{R}_{js}{}^s \mathbf{Q}_j{}^{s-1}$
    end for
6:  $\mathbf{R}^s_{ss} \leftarrow \sqrt{\|\xi^s\|_2{}^2}$
7:  $\mathbf{Q}^s \leftarrow [\mathbf{Q}^{s-1}, \frac{\xi^s}{\mathbf{R}_{ss}{}^s}]$
```

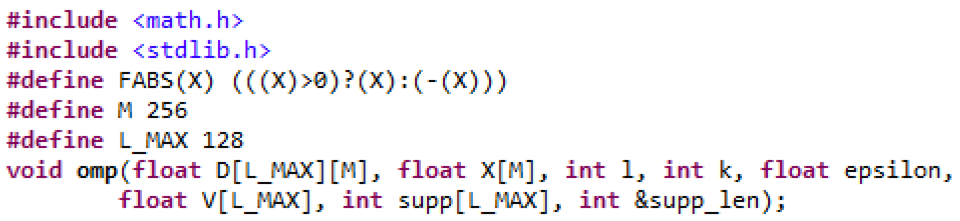
3 Implementation

3.1 Hardware Setting

For the purpose of this tutorial, we use Xilinx Virtex-7 VC709 Evaluation Kit (xc7vx690tffg1761-2) as our hardware accelerator platform. An Intel core i7-2600K processor running on the Windows OS is utilized as our general purpose processing unit for comparison with the software-based realization. All computations are performed using single precision floating point operations. Vivado HLS 2016.2 is used to synthesize and simulate our OMP kernel. Our target clock cycle period is $10ns$.

3.2 Baseline Implementation

Figure 1 shows the interface of OMP function and its input and output data. The inputs of the OMP function are as follows: dictionary matrix ($D[LMAX][M]$), input sample ($X[M]$), dictionary size l, sparsity level k, and desired error threshold ϵ (Algorithm 1). Parameter M is the feature space size of the dictionary matrix D which also represents the size of input data X (we used X instead of Y_{n_f} in our code for simplicity). The parameter L_MAX denotes the maximum number of samples in the dictionary matrix ($l \leq L\_MAX$). The OMP module then outputs the computed coefficient vector v, support set Λ (denoted as $supp[L\_MAX]$ in our code), and the number of non-zeros in the computed vector v as *supp_len*. The defined function denoted as $FABS$ is used in our OMP code to compute the absolute value of a number.



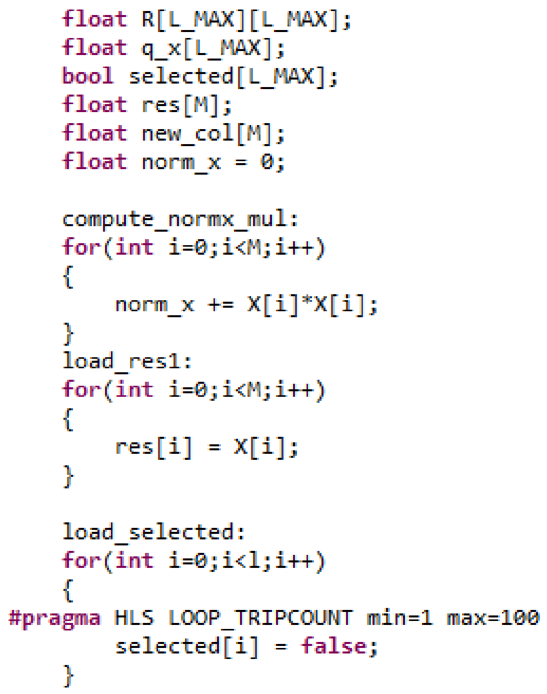
```
#include <math.h>
#include <stdlib.h>
#define FABS(X) (((X)>0)?(X):(-(X)))
#define M 256
#define L_MAX 128
void omp(float D[L_MAX][M], float X[M], int l, int k, float epsilon,
        float V[L_MAX], int supp[L_MAX], int &supp_len);
```

Figure 1: *h* file code (omp.h).

In the following of this section, we explain the realization of OMP module step by step. Figure 2 shows the declaration of intermediate variables and the initialization of the residual vector (*res*) per line 1 of Algorithm 1. The Boolean vector, *selected*, is used to indicate which sample (column) of the dictionary matrix is selected so far. This array is initially set to all *False*. We use the pragma *Loop_TRIPCOUNT* to explicitly specify the range of loop trip counts for the compiler optimization purposes. Not using this pragma did result in an undefined number for the maximum latency after synthesizing.

The main loop in the OMP algorithm can be divided into three inter-linked modular tasks (Figures 3, 4, and 5). In the first module (Figure 3), we find the index of best matching column of the dictionary matrix in each iteration. As we discuss in the Section 3.3.1, we leveraged the same set of block RAMs for the Q matrix and D. As such, in our *cpp* code Qt denotes the dictionary matrix. We check the norm of variable *dot_D_res* to make sure we are not dealing with very small correlations between the residual vector and dictionary samples.

The newly selected dictionary sample (*new_col*) is then used to update the R and Q matrices according to the Algorithm 2 (Figure 4). The updated Q and R matrices are leveraged to solve the LS optimization task and update the residual vector as we



```
    float R[L_MAX][L_MAX];
    float q_x[L_MAX];
    bool selected[L_MAX];
    float res[M];
    float new_col[M];
    float norm_x = 0;

    compute_normx_mul:
    for(int i=0;i<M;i++)
    {
        norm_x += X[i]*X[i];
    }
    load_res1:
    for(int i=0;i<M;i++)
    {
        res[i] = X[i];
    }

    load_selected:
    for(int i=0;i<l;i++)
    {
#pragma HLS LOOP_TRIPCOUNT min=1 max=100
        selected[i] = false;
    }
```

Figure 2: Variable initialization per lines 1-2 of Algorithm 1.

discussed in detail in Section 2. The algorithm terminates the main loop if the remainder (*residual*) is less than the given error threshold ϵ.

Upon completion of the main loop (either reaching the maximum sparsity level k or getting a residual less than ϵ), we compute the coefficient matrix v by solving the $\mathbf{v} = \mathbf{R}^{-1}\mathbf{Q}^t\mathbf{X}$ as discussed in Section 2. Note that the matrix R is an upper triangle matrix for which the inversion can be efficiently computed using back substitution as illustrated in the Figure 6.

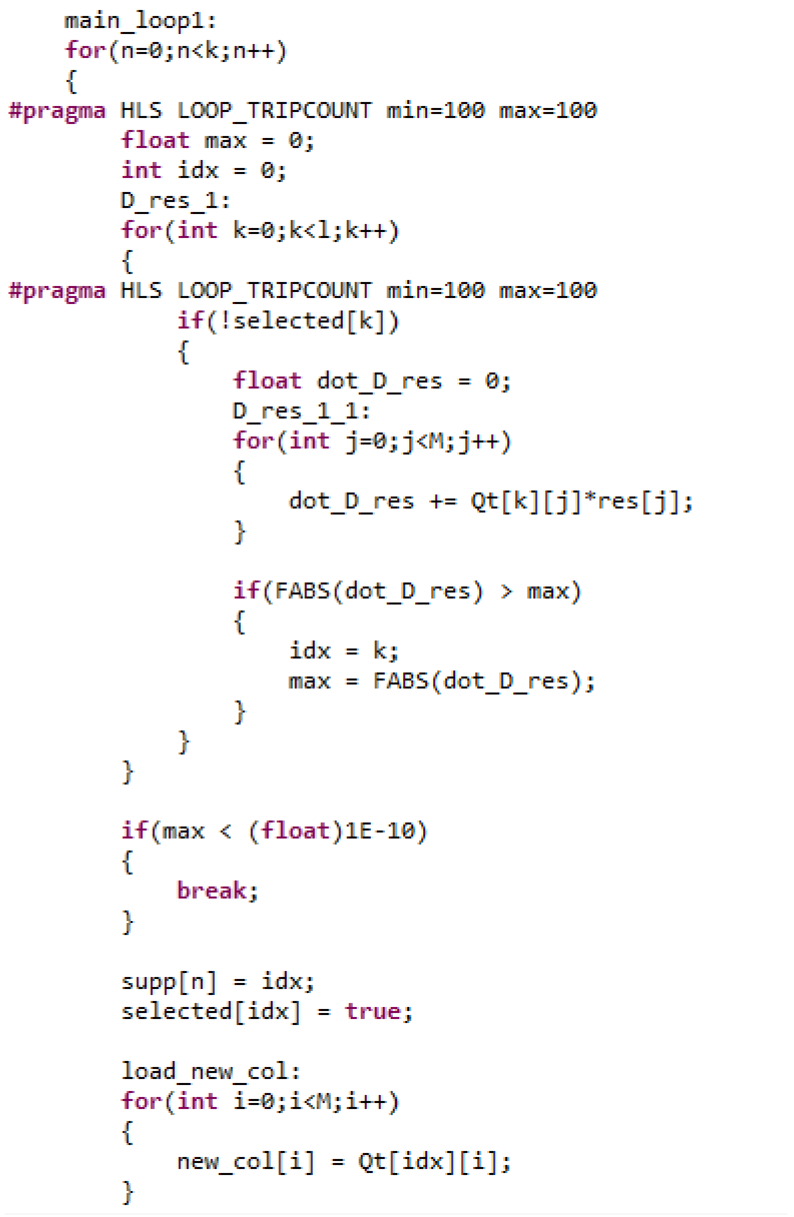
3.3 Optimized Solution

The general trend of our final code is similar to our baseline implementation discussed in Section 3.2. As we explain in the remainder of this section, we leverage both algorithmic and pragma optimization to provide an efficient implementation of the OMP routine.

3.3.1 Algorithmic Optimization

We use two sets of algorithmic optimization in our realization of OMP kernel by exploiting the parallelism that exists in the algorithm:

(i) We observe that the OMP algorithm includes multiple dot product computations which result in frequent appearance of for-loops requiring an operation similar to $a += b[i] \times c[i]$. Due to the sequential nature of such operations (Figure 7a) simple use of pipelining/unrolling pragmas does not help. Thereby, We suggest to transform such sequential operations ($a += b[i] \times c[i]$) into a series of operations that can be independently run in parallel (e.g., $temp[i] = b[i] \times c[i]$). We then use a *tree-based reduction module* by implementing a tree-based adder to find the final sum value a by adding up the values stored in *temp* (Figure 7b). Optimizing the execution of such operations accelerates the dot product and norm computation steps that appear frequently in the

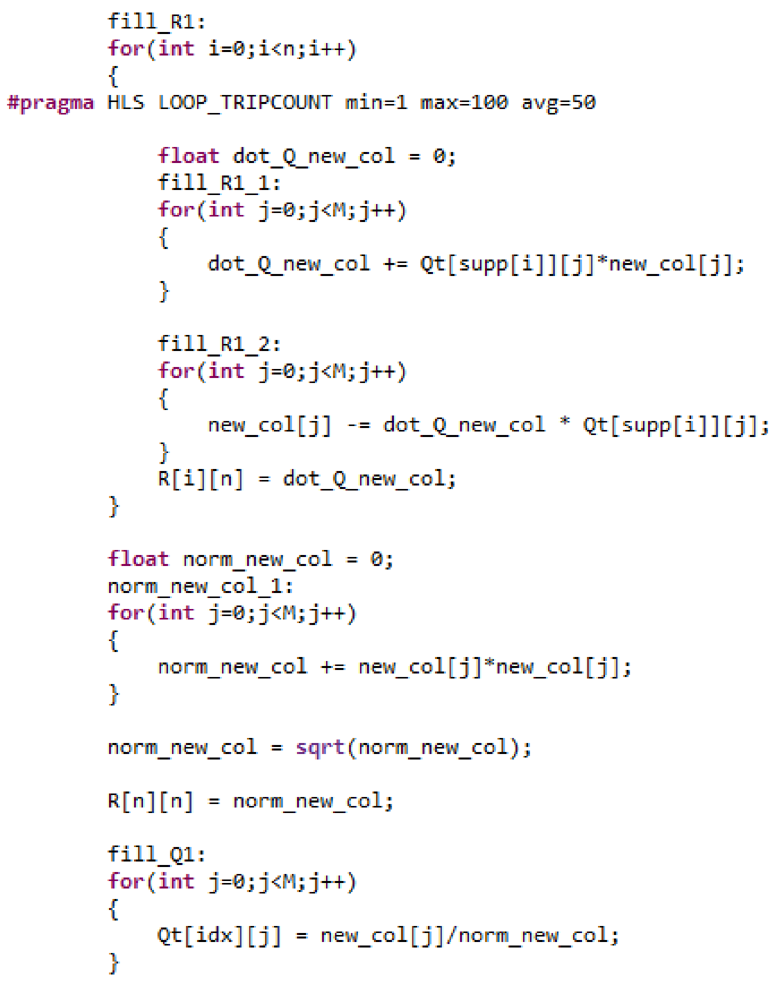


```
    main_loop1:
    for(n=0;n<k;n++)
    {
#pragma HLS LOOP_TRIPCOUNT min=100 max=100
        float max = 0;
        int idx = 0;
        D_res_1:
        for(int k=0;k<l;k++)
        {
#pragma HLS LOOP_TRIPCOUNT min=100 max=100
            if(!selected[k])
            {
                float dot_D_res = 0;
                D_res_1_1:
                for(int j=0;j<M;j++)
                {
                    dot_D_res += Qt[k][j]*res[j];
                }

                if(FABS(dot_D_res) > max)
                {
                    idx = k;
                    max = FABS(dot_D_res);
                }
            }
        }

        if(max < (float)1E-10)
        {
            break;
        }

        supp[n] = idx;
        selected[idx] = true;

        load_new_col:
        for(int i=0;i<M;i++)
        {
            new_col[i] = Qt[idx][i];
        }
```

Figure 3: Finding best fitting column.

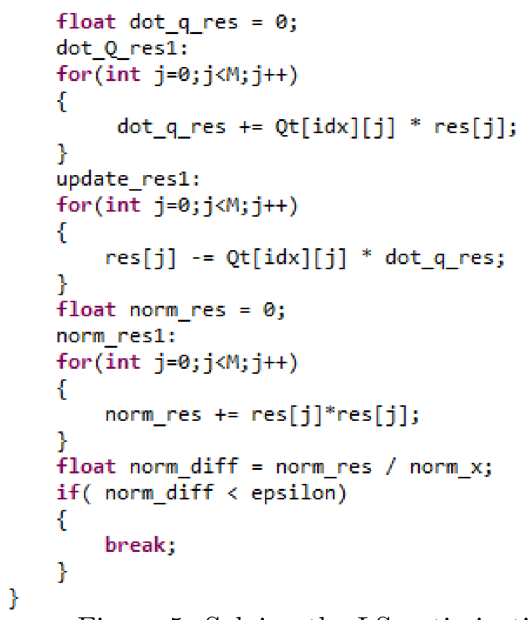
OMP routine. By means of the reduction module, we were able to reduce the Iteration Interval (II) and handle more operations simultaneously. As such, our implementation requires multiple concurrent loads and stores from a particular RAM. To cope with the concurrency, instead of having a large block RAM for matrices D and Q, we use multiple smaller sized block memories and fill these block RAMs by cyclic interleaving (*cyclic array partitioning*). Thus, we can perform a faster computation by accessing multiple successive elements of the matrices and removing the dependency in the for-loops.

Figure 9 outlines the realization of the tree-based reduction module. We pipeline and unroll this function to provide a more efficient solution as we later discuss in Section 3.3.2. The reduction function takes two arrays of size $2M$ as its inputs ($temp0$ & $temp1$). It incurs 4 different modes. In mode 0, our function reduces $temp0[0 : M-1]$ by using $temp1[0 : M-1]$ as temporary array and return the result. In mode 1, it adds



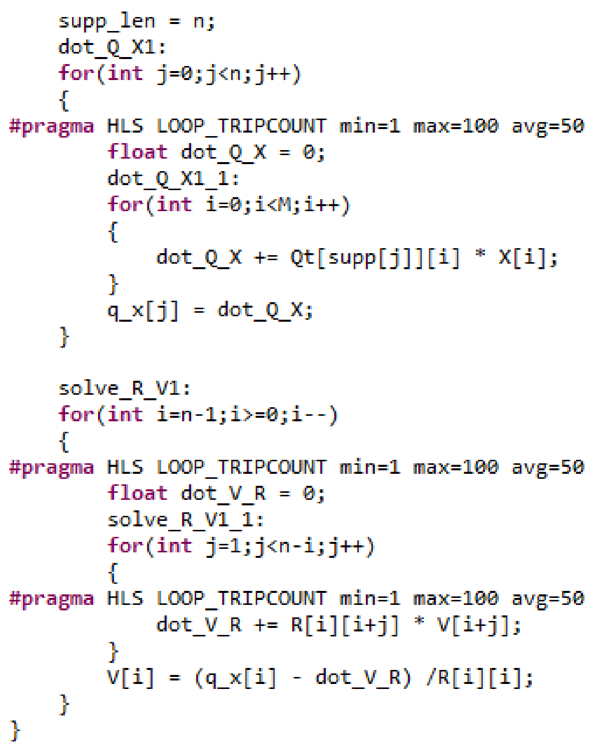
```
        fill_R1:
        for(int i=0;i<n;i++)
        {
#pragma HLS LOOP_TRIPCOUNT min=1 max=100 avg=50

            float dot_Q_new_col = 0;
            fill_R1_1:
            for(int j=0;j<M;j++)
            {
                dot_Q_new_col += Qt[supp[i]][j]*new_col[j];
            }

            fill_R1_2:
            for(int j=0;j<M;j++)
            {
                new_col[j] -= dot_Q_new_col * Qt[supp[i]][j];
            }
            R[i][n] = dot_Q_new_col;
        }

        float norm_new_col = 0;
        norm_new_col_1:
        for(int j=0;j<M;j++)
        {
            norm_new_col += new_col[j]*new_col[j];
        }

        norm_new_col = sqrt(norm_new_col);

        R[n][n] = norm_new_col;

        fill_Q1:
        for(int j=0;j<M;j++)
        {
            Qt[idx][j] = new_col[j]/norm_new_col;
        }
```

Figure 4: Updating Q and R matrices.



```
        float dot_q_res = 0;
        dot_Q_res1:
        for(int j=0;j<M;j++)
        {
             dot_q_res += Qt[idx][j] * res[j];
        }
        update_res1:
        for(int j=0;j<M;j++)
        {
            res[j] -= Qt[idx][j] * dot_q_res;
        }
        float norm_res = 0;
        norm_res1:
        for(int j=0;j<M;j++)
        {
            norm_res += res[j]*res[j];
        }
        float norm_diff = norm_res / norm_x;
        if( norm_diff < epsilon)
        {
            break;
        }
    }
```

Figure 5: Solving the LS optimization task and updating the residual vector



```
    supp_len = n;
    dot_Q_X1:
    for(int j=0;j<n;j++)
    {
#pragma HLS LOOP_TRIPCOUNT min=1 max=100 avg=50
        float dot_Q_X = 0;
        dot_Q_X1_1:
        for(int i=0;i<M;i++)
        {
            dot_Q_X += Qt[supp[j]][i] * X[i];
        }
        q_x[j] = dot_Q_X;
    }

    solve_R_V1:
    for(int i=n-1;i>=0;i--)
    {
#pragma HLS LOOP_TRIPCOUNT min=1 max=100 avg=50
        float dot_V_R = 0;
        solve_R_V1_1:
        for(int j=1;j<n-i;j++)
        {
#pragma HLS LOOP_TRIPCOUNT min=1 max=100 avg=50
            dot_V_R += R[i][i+j] * V[i+j];
        }
        V[i] = (q_x[i] - dot_V_R) /R[i][i];
    }
}
```

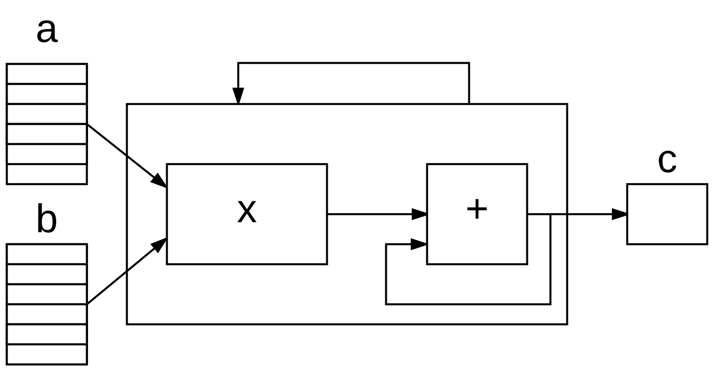
Figure 6: Computing coefficient vector v.

$temp1[0 : M-1]$ to $temp0[M : 2 \times M-1]$ (used for storing res). In mode 2, it adds $temp0[0 : M-1]$ to $temp1[M : 2 \times M-1]$ (used for storing new_col). In mode 3, we reduce $temp1[0 : M-1]$ by using $temp0[0 : M-1]$ as temporary array and return result. We use HLS pragma *Inline* to instruct the HLS compiler to dissolve and absorb this function into its top level.

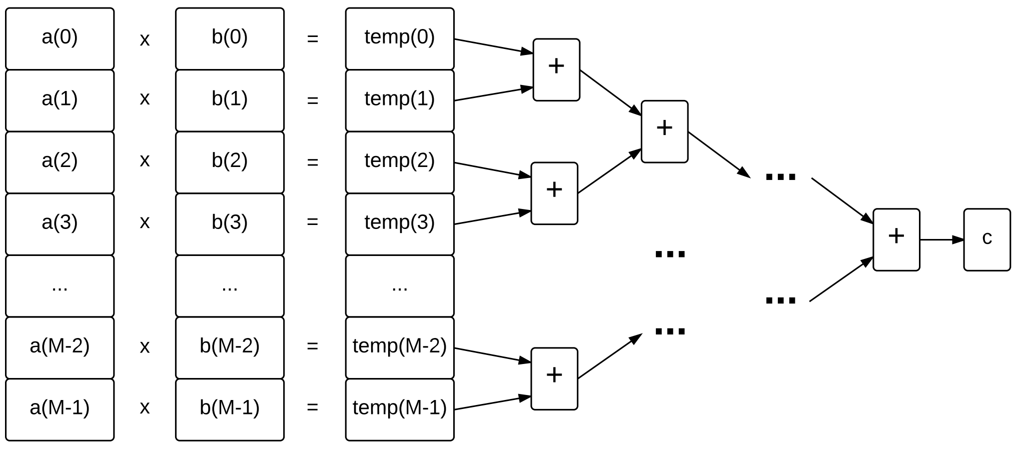
(ii) Using the block RAM is desirable in FPGA implementations because of its fast access time. The number of block RAMs on one FPGA is limited, so it is important to optimize the amount of utilized block memories. We reduce block RAM utilization in our realization by a factor of 2 compared to the naive implementation. This reduction is a consequence of our observation that none of the columns of matrix $\mathbf{D}$ would be selected twice during one call of the OMP algorithm. This is particularly because the updated residual at the end of each iteration is made orthogonal to the selected dictionary samples. Thereby, for computing line 4 of Algorithm 1 we only use the indices of $\mathbf{D}$ that are not selected during the previous OMP iterations. We instead use the memory space that was originally assigned to the selected columns of $\mathbf{D}$ to store the newly added columns of matrix $\mathbf{Q}$. Figure 10 illustrates the inner structure of our OMP realization.

3.3.2 Pragma Optimization

Table 1 summarizes the pragma optimization used in our final implementation. We simultaneously use pipelining and unrolling with factor 16 to optimize each of our inner for-loops (we experimentally select unroll factor 16 as it gives a reasonable trade-off between the throughput and resource utilization). To make this optimization work, we use cyclic array partitioning and tree-based reduction as discussed in Section 3.3. Figure 11



(a)



(b)

Figure 7: Facilitating dot product and norm computation steps that frequently appear in the OMP routine. (a) Conventional sequential approach. (b) Proposed tree-based model. Our approach reduces the II of for-loops that involves operation such as $a += b[i] \times c[i]$ to 1.

shows we used Array_Map to make sure that each of our intermediate variables are stored in a unified sets of BRAMs in case one BRAM is not enough to store the whole variable. This is to help the compiler decode the input addresses when accessing different elements of the array.

Table 1: High-level overview of optimization pragmas used in our final code and baseline implementation.

	Baseline	**Optimized Solution**
Loop Unrolling		x
Loop Pipelinining		x
Function Inlining		x
Array Optimization		x
Data Dependency		x
Loop Trip Count	x	x

```
float reduce_temp(float temp0[2* M], float temp1[2* M], int mode)
{
#pragma HLS INLINE
    if(mode==0 || mode==3)
    {
        int n = M/2;
        bool use_temp0 = (mode==0);
        float reduce_ret = 0;

        compute_normx_sum_1:
        while(n>0)
        {
            if(use_temp0)
            {
                compute_normx_sum_2_0:
                for(int i=0;i<n;i++)
                {
#pragma HLS LOOP_TRIPCOUNT min=1 max=128
#pragma HLS UNROLL factor=16
#pragma HLS PIPELINE
                    temp1[i] = temp0[2*i] + temp0[2*i+1];
                }
            }
            else
            {
                compute_normx_sum_2_1:
                for(int i=0;i<n;i++)
                {
#pragma HLS LOOP_TRIPCOUNT min=1 max=128
#pragma HLS UNROLL factor=16
#pragma HLS PIPELINE
                    temp0[i] = temp1[2*i] + temp1[2*i+1];
                }
            }
            use_temp0 = !use_temp0;
            n = n>>1;
        }
        if(use_temp0)
        {
            reduce_ret = temp0[0];
        }
        else
        {
            reduce_ret = temp1[0];
        }

    }
    else if(mode==1)
    {
        equal_add_1:
        for(int i=0;i<M;i++)
        {
#pragma HLS DEPENDENCE variable=temp0 array inter false
#pragma HLS UNROLL factor=16
#pragma HLS PIPELINE
            temp0[i+M] += temp1[i];
        }
    }
    else if(mode==2)
    {

        equal_add_2:
        for(int i=0;i<M;i++)
        {
#pragma HLS DEPENDENCE variable=temp1 array inter false
#pragma HLS UNROLL factor=16
#pragma HLS PIPELINE
            temp1[i+M] += temp0[i];
        }
    }
    return 0;
}
```

Figure 9: Tree-based reduction function

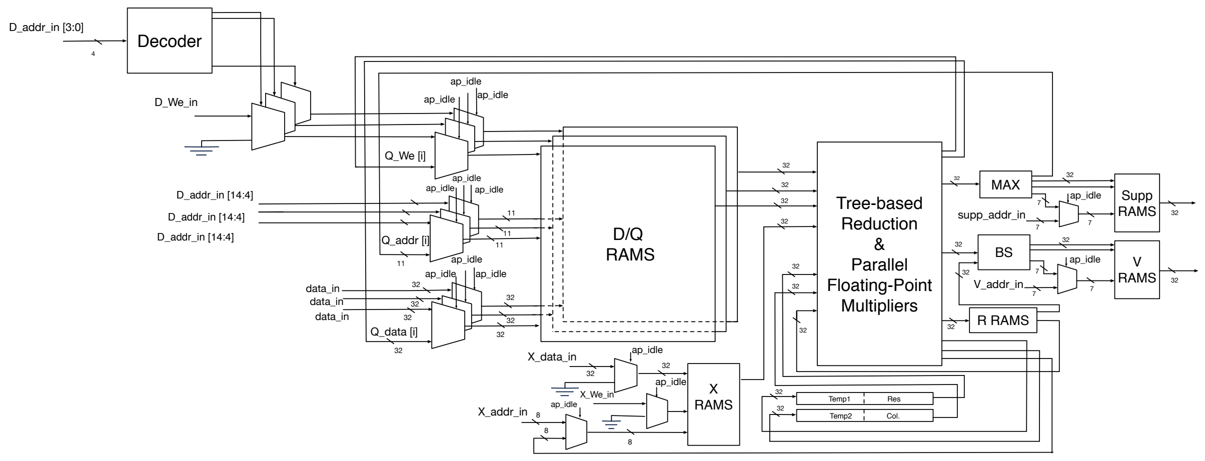


Figure 10: Schematic depiction of our OMP kernel.

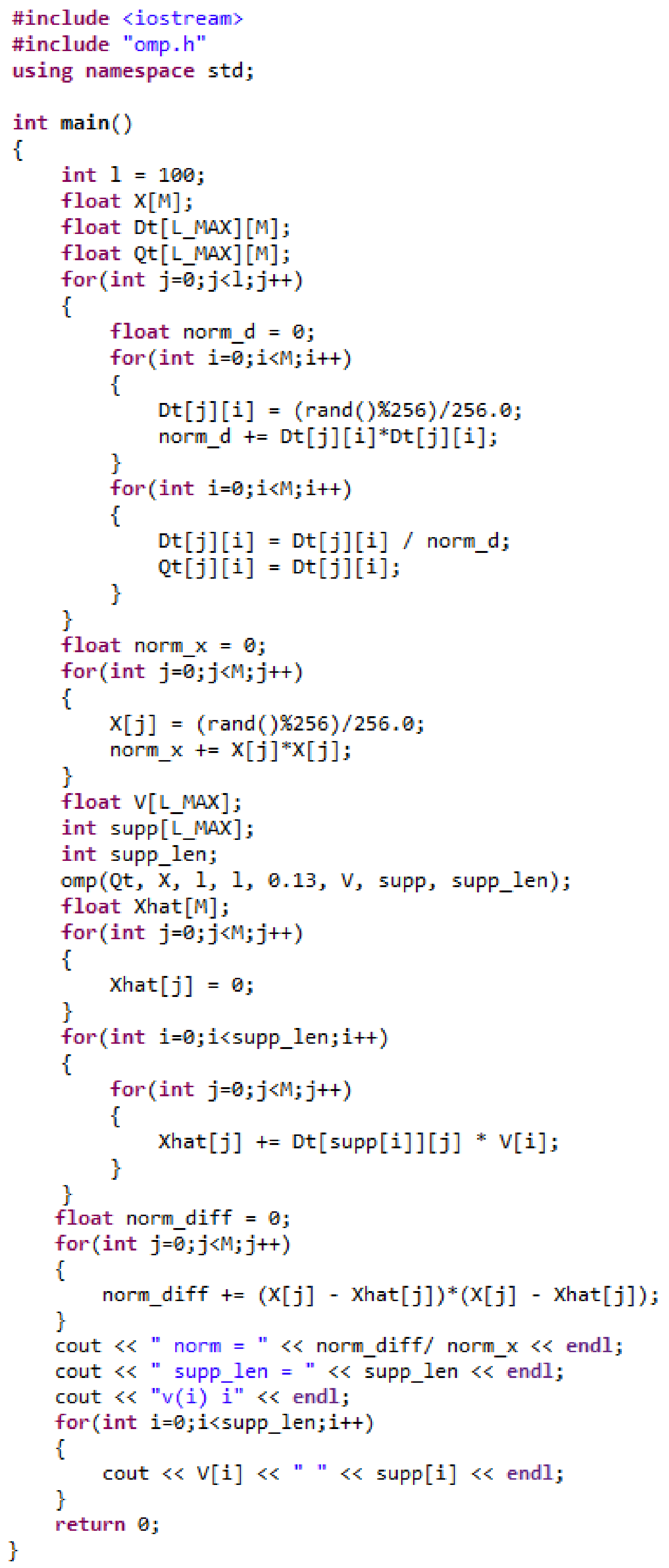
```
#pragma HLS RESOURCE variable=X core=RAM_1P_BRAM
#pragma HLS ARRAY_PARTITION variable=Qt cyclic factor=16 dim=2
#pragma HLS ARRAY_MAP variable=X horizontal

#pragma HLS ARRAY_MAP variable=supp horizontal
#pragma HLS ARRAY_MAP variable=V horizontal

    float R[L_MAX][L_MAX];
    float q_x[L_MAX];
    bool selected[L_MAX];

 #pragma HLS ARRAY_MAP variable=R horizontal
 #pragma HLS ARRAY_MAP variable=selected horizontal
 #pragma HLS ARRAY_MAP variable=q_x horizontal

    float temp0[2*M];
    float temp1[2*M];

    #pragma HLS ARRAY_PARTITION variable=temp0 cyclic factor=16 dim=1
#pragma HLS ARRAY_PARTITION variable=temp1 cyclic factor=16 dim=1
```

Figure 11: Array optimization.

As we mentioned earlier, we use *Inline* pragma to ensure that the *reduce_temp* function and its submodules are dissolved and raised to the top level. We leverage the *Trip_Count* pragma within each for-loop to avoid unspecified maximum latency during the synthesis of the code. This is to explicitly specify the range of loop trip counts for the compiler optimization purposes. We also use HLS *Dependence* pragma in the *reduce_temp* function to avoid unnecessary dependency in the temp arrays and run the operations in parallel.

3.4 Test Bench

In our test bench module, we use randomly generated matrix D and input vector X to evaluate OMP performance in term of the accuracy (Figure 12). Our test bench outputs the L2 norm of $\|X - DV\|_2$ as the approximation error due to sparse representation of input data X. As we discuss in Section **??**, we replace the random variables for sample images for realization of a particular object detection task. We also compare our result with a golden model generated by MATLAB. Our comparison shows less than 10^{-9} difference. This variation is because of the floating point operations and precision difference of MATLAB and Vivado.



```cpp
#include <iostream>
#include "omp.h"
using namespace std;

int main()
{
    int l = 100;
    float X[M];
    float Dt[L_MAX][M];
    float Qt[L_MAX][M];
    for(int j=0;j<l;j++)
    {
        float norm_d = 0;
        for(int i=0;i<M;i++)
        {
            Dt[j][i] = (rand()%256)/256.0;
            norm_d += Dt[j][i]*Dt[j][i];
        }
        for(int i=0;i<M;i++)
        {
            Dt[j][i] = Dt[j][i] / norm_d;
            Qt[j][i] = Dt[j][i];
        }
    }
    float norm_x = 0;
    for(int j=0;j<M;j++)
    {
        X[j] = (rand()%256)/256.0;
        norm_x += X[j]*X[j];
    }
    float V[L_MAX];
    int supp[L_MAX];
    int supp_len;
    omp(Qt, X, l, l, 0.13, V, supp, supp_len);
    float Xhat[M];
    for(int j=0;j<M;j++)
    {
        Xhat[j] = 0;
    }
    for(int i=0;i<supp_len;i++)
    {
        for(int j=0;j<M;j++)
        {
            Xhat[j] += Dt[supp[i]][j] * V[i];
        }
    }
    float norm_diff = 0;
    for(int j=0;j<M;j++)
    {
        norm_diff += (X[j] - Xhat[j])*(X[j] - Xhat[j]);
    }
    cout << " norm = " << norm_diff/ norm_x << endl;
    cout << " supp_len = " << supp_len << endl;
    cout << "v(i) i" << endl;
    for(int i=0;i<supp_len;i++)
    {
        cout << V[i] << " " << supp[i] << endl;
    }
    return 0;
}
```

Figure 12: Test bench module.

3.5 Example Background Subtraction

The sparse representation of each image (the foreground information) can be analyzed to locate a desired object. To do so, we use a locally adaptive thresholding method to transform the gray-scale output of background subtraction unit into a binary image based on the local statistics of the neighborhood pixels such as range and variance [2]. In our sample evaluation, we use the following criteria to compute the threshold corresponding to each patch of data:

$$T(x,y) = m(x,y) + [1 + \gamma(\frac{\sigma(x,y)}{R} - 1)], \tag{2}$$

where $m(x,y)$ and $\sigma(x,y)$ are the local sample mean and variance respectively. Figure 13 shows the output of background subtraction unit for an example image in which multiple cars are located near each other in a parking lot. In this experiment, we used $R = 256$, and $\gamma = 0.5$ for binarization. The dictionary size l is 256, and the patch size is set to (32×8) with an overlap of (8×2) pixels.

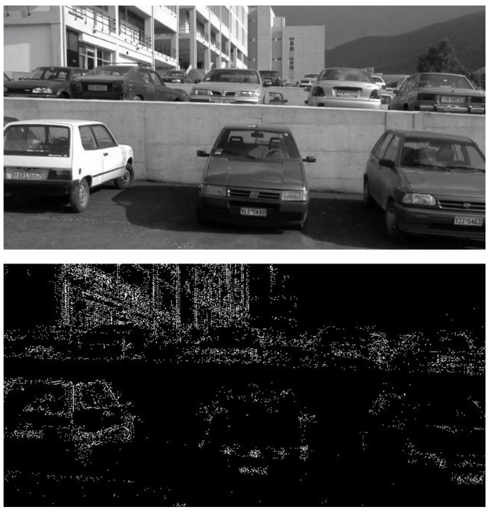


Figure 13: Example output of background subtraction unit after binarization. In this experiment the sparsity level k is set to 16.

Following the image binarization, Connected Component Analysis (CCA) can be applied to determine the group of pixels that are probably related to one another. CCA is developed based on the assumption that pixels in a connected component (object) share similar pixel intensity values and are connected with their adjacent pixels. Once all groups have been determined, each pixel is labeled with a value according to the component to which it was assigned. To detect/track a particular object in the newly labeled set of patches, one can make use of simple morphological characteristics such as the aspect ratio (a.k.a., eccentricity), and object orientation. For instance, to locate license plates in a surveillance video we can use $1.5 \leq \mathit{aspectratio} \leq 4.5$ and $10° \leq \mathit{orientation} \leq 60°$ as effective criteria to successfully locate license plate regions in an image. We emphasize that our approach is generic and can be used for a variety of surveillance applications other than license plate recognition.

References

[1] Lin Bai, Patrick Maechler, Michael Muehlberghuber, and Hubert Kaeslin. 2012. High-speed compressed sensing reconstruction on FPGA using OMP and AMP. In Electronics, Circuits and Systems (ICECS), 2012 19th IEEE International Conference on. IEEE, 5356.

[2] J. Sauvola and M. Pietikäinen, “Adaptive document image binarization,” *Pattern recognition*, vol. 33, no. 2, pp. 225–236, 2000.