Abstract—Iris recognition is one of the most accurate biometric methods in use today. However, the iris recognition algorithms are currently implemented on general purpose sequential processing systems, such as generic central processing units (CPUs). In this work, we present a more direct and parallel processing alternative using field-programmable gate arrays (FPGAs), offering an opportunity to increase speed and potentially alter the form factor of the resulting system. Within the means of this project, the most time-consuming operations of a modern iris recognition algorithm are deconstructed and directly parallelized. In particular, portions of iris segmentation, template creation, and template matching are parallelized on an FPGA-based system, with a demonstrated speedup of 9.6, 324, and 19 times, respectively, when compared to a state-of-the-art CPU-based version. Furthermore, the parallel algorithm on our FPGA also greatly outperforms our calculated theoretical best Intel CPU design. Finally, on a state-of-the-art FPGA, we conclude that a full implementation of a very fast iris recognition algorithm is more than feasible, providing a potential small form-factor solution.

Index Terms—Biometrics, field-programmable gate arrays (FPGAs), iris recognition, parallel computing.

I. INTRODUCTION

Iris recognition stands out as one of the most accurate biometric methods in use today. Currently, iris recognition algorithms are deployed globally in a variety of systems ranging from personal computers to portable iris scanners. The overall performance, in terms of size, shape, speed, power, and accuracy, of these systems have become of great interest.

Currently, iris recognition algorithms are deployed globally in a variety of systems ranging from personal computers to portable iris scanners. These systems use central processing unit (CPU)-based systems for which the algorithm was originally designed. CPU-based systems are considered general purpose machines, designed for all types of applications. Therefore, CPU-based systems are known as sequential processing devices. Fig. 1 illustrates a simple processor pipeline. Instructions are first fetched, decoded, and finally executed by arithmetic logic units, or ALUs. State-of-the-art processors contain more than one ALU. Therefore, multiple instructions can be executed in parallel. However, modern processors are limited in the number of ALUs they possess, and most of today’s CPUs do not have more than four, as illustrated in Fig. 1. As pointed out in previous papers [24]–[26], this method of processing for which the recognition algorithm is currently designed is inefficient since parts of the iris recognition algorithm can be greatly parallelized.

The goal of this research is to find and exploit potential parallelism within iris recognition algorithms. Field-programmable gate arrays (FPGAs) are complex programmable logic devices that are essentially a “blank slate” integrated circuit that can be programmed with nearly any parallel logic function. They are fully customizable and the designer can prototype, simulate, and implement a parallel logic function without the costly process of having a new integrated circuit manufactured from scratch.

By programming the hardware directly, performance and energy consumption can be improved while the size and form factor of the system can be altered. The focus of this research is on parallelizing key portions of the iris recognition algorithm using an FPGA. This work demonstrates this by making the following contributions:

1) Introduction and calculation of the theoretical best performance of CPU-based machines executing key components of an iris recognition algorithm.

2) Measurements of CPU performance of key components of an iris recognition algorithm on a state-of-the-art computer.

3) Deconstruction and novel parallelization of three key iris recognition components: 1) Iris segmentation using local kurtosis, 2) iris template creation via filtering, and 3) template matching via hamming distance.

4) Evaluation of the benefits of parallelization in terms of performance and size.
II. RIDGE ENERGY DETECTION IRIS RECOGNITION ALGORITHM

The first and most popular iris recognition algorithm was introduced by pioneer Dr. J. Daugman [1], [16]. This patented algorithm is not available for open use, so an alternative for research purposes can be found in the implementation created by L. Masek [17]. There are also many other proposed methods [18]–[22] to perform iris recognition. Ives et al. have developed an alternate iris recognition algorithm, referred to as the ridge energy direction (RED) algorithm [2] that will be the focus of this work. The RED algorithm has been shown to have comparable performance to state-of-the-art algorithms that have been previously published [2]. In particular, the three parts of the RED algorithm have been fine-tuned to produce highly accurate iris recognition achieving 99.999% and 99.603% accuracy on the Bath2000 [28] and CASIA I [29] databases, respectively. In comparison, Masek’s implementation [17] of Daugman’s algorithm resulted in 99.526% and 99.630% accuracy on the same Bath2000 and CASIA I databases, respectively.

The steps of the RED algorithm are illustrated in Fig. 2. One of the first steps in preprocessing the image before extracting its unique features is to segment the iris from the rest of the image. Several means have been proposed to perform this segmentation [4]–[8], but this algorithm utilizes a different approach using local statistics [15] to aid in determining the outer boundary of the iris. After iris segmentation, feature extraction is performed based on directional filters. The iris is then transformed to polar coordinates, and the iris image is histogram equalized and then processed by each of two different directional filters (vertically and horizontally oriented). At every pixel location, the identity of the directional filter which provides the largest value of output is recorded, and represented with a single bit, thus generating a template. Templates are then matched using fractional Hamming Distance as the measure of closeness. We will now highlight three main components of the RED algorithm that help dictate performance of the overall algorithm: 1) segmentation of the outer boundary of the iris; 2) iris template generation; and 3) iris template matching.

A. Iris Segmentation With Local Kurtosis

After determining the location of the pupil, the center of the pupil is used as a reference for unwrapping the iris image to polar coordinates (50 rows by 180 columns). The center of the limbic boundary is not typically the same as the center of the pupil, so we consider a number of possible centers for the limbic boundary: the pupil center and the pupil center shifted by small increments to the left and right. Note that when unwrapping the iris from the correct center of the outer boundary, in the unwrapped image the outer boundary will appear as a horizontal line. For each of these unwrapped images, we compute the local kurtosis [9] in 3 × 3 neighborhoods of each pixel. In the resulting image, we observe that the values near the limbic boundary are relatively stable in small neighborhoods, compared with the appearance in the sclera and in the interior of the iris. Therefore, to create binary images, we assign a value of 1 (white) to locations for which kurtosis values within a 3 × 3 neighborhood are nearly constant. The correct center of the outer boundary will appear in the unwrapped image that has the longest nearly horizontal white border. Calculating the kurtosis repeatedly to aid with iris segmentation is a time-consuming portion of the RED algorithm, and hence we focus on it in this research.

B. Template Generation via Filtering

After determining the inner and outer boundaries and center of the pupil, the iris is again transformed into polar coordinates with the center of the pupil as the point of reference, into a 120 row by 180 column image. In this process, the radial extent of the iris is normalized in order to account for pupil dilation. Each row in the unwrapped iris image represents an annular region surrounding the pupil, and the columns represent radial information. Next, we consider the “energy” of the unwrapped iris image after an adaptive histogram equalization. Here, energy loosely refers to the prominence (pixel values) of the ridges that appear in the histogram equalized image: higher value reflects higher energy. This “energy” image is passed into each of two different directional filters (a vertical filter and a horizontal filter). These filters are used to indicate the presence of strong ridges, and the orientation of these ridges.

At every pixel location in the filtered image, the filter which provides the largest value of output is recorded and encoded with one bit to represent the identity of this directional filter. The iris image is thus transformed into a one-bit template that is the same size as the image in polar coordinates (120 rows by 180 columns). A template mask is also created during this filtering process. If both filter output values are not above a certain threshold, then a mask bit is cleared for that particular pixel location. The template mask is used to identify pixel locations where neither vertical nor horizontal directions are identified.

C. Template Matching via Hamming Distance

The template can now be compared to a stored template using the fractional Hamming distance as the measure of closeness (see Fig. 3). The fractional Hamming distance between two templates is defined as

\[
\text{HD} = \frac{||(\text{template } A \odot \text{template } B) \cap \text{mask } A \cap \text{mask } B||}{||\text{mask } A \cap \text{mask } B||},
\]

where \(\odot\) is the EXCLUSIVE OR operation to detect disagreement between the pairs of bits that represent the directions in the two templates, \(\cap\) represents the binary AND function, and masks \(A\) and \(B\) identify the values in each template.
that are not corrupted by artifacts such as eyelids/eyelashes and specularities. The denominator of (1) ensures that only the bits that matter are included in the calculation, after artifacts are discounted. Rotation (due to head-tilt) mismatch between irises is handled with left–right shifts of the coded one-bit template to determine the minimum Hamming distance. The lower the HD result, the greater the match between the two irises being compared. The fractional Hamming distance between two templates is compared to a predetermined threshold value and a match or nonmatch declaration is made. The time-consumption of template matching can greatly affect speed performance, since it is tied to the size of the database that it is being matched against.

III. FPGA PARALLELIZATION

FPGAs are complex programmable logic devices that are essentially a “blank slate” integrated circuit that can be programmed with nearly any parallel logic function. They are fully customizable and the designer can prototype, simulate, and implement a parallel logic function without the costly process of having a new integrated circuit manufactured from scratch. FPGAs are commonly programmed via VHIC Hardware Description Language (VHDL). VHDL statements are inherently parallel, not sequential. VHDL allows the programmer to dictate the type of hardware that is synthesized on an FPGA. For example, if you would like to have 2048 XOR logic gates that execute in parallel, then you program this directly in the VHDL code.

In this research, we will demonstrate a parallelization of three major components of the RED iris recognition algorithm: 1) iris segmentation with kurtosis, 2) template creation via filtering, and 3) template matching via hamming distance. We will conclude this section with a brief discussion of future parallelizations.

A. Iris Segmentation via Kurtosis

As stated previously, kurtosis is used to aid with iris segmentation. In the RED algorithm, repeated kurtosis calculations on $5 \times 5$ elements are made across the size of an image. Kurtosis is often used in probability theory to represent the “peakedness” of the probability distribution of a real-valued random variable [10]. Intuitively it is known as a measure of the volatility of volatility. It is described by the following mathematical function:

$$1/N \sum_{i=1}^{N} (x_i - \bar{x})^4$$

(2)

where $N$ represents the number of elements (25) for this calculation, $x$ represents the mean for those elements, and $x$ is the random variable representing their actual values.

Fig. 4 introduces the C++ code for the kurtosis function, and ave represents the average of the block of code. It is calculated inside a FOR loop, where the loop cycles through a $5 \times 5$ matrix and the index variable is, therefore, 25. Notice the sequential nature, since each matrix element is added one by one. The next FOR loop contains the calculations for variance and fourth moment by using the calculated ave above. For example, a difference is first calculated followed by four multiplications and a summation to create the fourth moment. This process is also repeated 25 times sequentially. After the loop, final computation for the kurtosis function is performed with three divisions and a multiplication.

The Intel Assembly code for this calculation is shown in Fig. 3. In general, eight sequential instructions are required to perform the loop overhead, following that each calculation requires multiple instructions. For example, the fourth moment calculation requires three floating point multiplications. Floating point multiplications typically require dozens of processor cycles to execute. The total number of assembly instructions required to perform this kurtosis calculation is 741, of which 259 are floating point instructions. Instruction execution bandwidth for a processor is limited by the number of functional units that it contains. For example, a processor with four ALU functional units can maximally execute four instructions per cycle. Therefore, in theory, a processor executing four instructions per cycle at 4 GHz can maximally execute 16-G instructions per second. Therefore, assuming a processor speed of 4 GHz, four ALUs, and four instructions executed per cycle, the total theoretical best case time is 46.3 ns

$$\frac{741 \text{ inst}}{4 \text{ inst}} \times \frac{1 \text{ cycle}}{4 \text{ G cycles}} = 46.3 \text{ nsec}.$$
appears to be sequential. Each iteration however of a FOR loop is implemented in parallel. Therefore, all 25 iterations of the loop are performed concurrently. Due to the inherent multiplication required by the kurtosis algorithm, this compilation and synthesis results in an implementation that utilizes the limited on chip multipliers. We will discuss the hardware usage of our algorithm in the following section.

In addition to the previously described “intra” kurtosis function parallelization, “inter” kurtosis function parallelization can also be computed. Here, “inter” kurtosis function parallelization means computing multiple kurtosis function calculations simultaneously. We synthesized a different hardware version consisting of two kurtosis algorithms executing in parallel to demonstrate feasibility and potential. We will present results

```
ave=sdev=numer=0.0;
// line 36
for (ii=0;ii<index;ii++)
    ave+=n[iii];
aveFloat=(float)ave/(float)index;
// line 40
for (ii=0;ii<index;ii++)
{
    s=(float)ii-aveFloat;
    numer+=s*s*s*s;
    sdev+=s*s;
}
numer/=(float)index;
sdev/=(float)(index-1);
if (sdev != 0.0)
    kurt=numer/(sdev*sdev);
else
    kurt=0.0;
```

(a)

```
for i in 0 to size-1 loop
    mean := mean + i;
end loop;
mean := mean / size;
for i in 0 to size-1 loop
    diff := conv_integer(mat1(i)) - mean;
    var := var + diff * diff;
    fourth := fourth + diff * diff * diff * diff;
end loop;
fourth := fourth / size;
var := var(size-1);
var_squared := var * var;
kurt := (fourth) / var_squared;
```

(c)

```
36: for (ii=0;ii<index;ii++)
0040119C mov dword ptr [ebp-18h],0
004011A3 jmp main+136h (004011ae)
004011A5 mov eax,dword ptr [ebp-18h]
004011A8 add eax,1
004011AB mov dword ptr [ebp-18h],eax
004011AE mov ecx,dword ptr [ebp-18h]
004011B1 cmp ecx,dword ptr [ebp-30h]
004011B4 jge main+157h (004011c7)
37: ave+=n[eei];
004011B6 mov edx,dword ptr [ebp-18h]
004011B9 mov eax,dword ptr [ebp-54h]
004011BC mov ecx,dword ptr [ebp-50h]
004011BF add ecx,dword ptr [eax+edx*4]
004011C2 mov dword ptr [ebp-50h],ecx
004011C5 jmp main+135h (004011a5)
38: aveFloat=(float)ave/(float)index;
004011C7 fild dword ptr [ebp-50h]
004011CA fdiv dword ptr [ebp-30h]
004011CD fstp dword ptr [ebp-38h]
39:
40: for (ii=0;ii<index;ii++)
004011D0 mov dword ptr [ebp-18h],0
004011D7 jmp main+172h (004011e2)
004011D9 mov edx,dword ptr [ebp-18h]
004011DC add edx,1
004011DF mov dword ptr [ebp-18h],edx
004011E2 mov eax,dword ptr [ebp-18h]
004011E5 cmp eax,dword ptr [ebp-30h]
004011E8 jge main+1A6h (00401216)
41: 
42: s=(float)ii-aveFloat;
004011EA mov ecx,dword ptr [ebp-18h]
004011ED mov edx,dword ptr [ebp-54h]
004011F0 fild dword ptr [edx+ecx*4]
004011F3 fsub dword ptr [ebp-38h]
004011F6 fst dword ptr [ebp-44h]
43: numer+=s*s*s*s;
004011F9 fmul dword ptr [ebp-44h]
004011FC fmul dword ptr [ebp-44h]
004011FF fmul dword ptr [ebp-44h]
00401202 fadd dword ptr [ebp-40h]
00401205 fstp dword ptr [ebp-40h]
44: sdev+=s*s;
00401208 fild dword ptr [ebp-44h]
0040120B fmul dword ptr [ebp-44h]
0040120E fadd dword ptr [ebp-3Ch]
00401211 fstp dword ptr [ebp-3Ch]
45: }
00401214 jmp main+169h (004011d9)
```

(b)

Fig. 4. Kurtosis algorithm in (a) C++, (b) Intel Assembly, and (c) VHDL.
B. Template Creation via Filtering

The RED iris recognition algorithm uses a digital filter to generate the iris template. The energy data passed from the iris segmentation process is organized into a rectangular array. The rectangular array is formed into a single-holed torus to help account for edge effects. The filter passes over this periodic array taking in 81 (9 × 9) values at a time (note, in [2], 11 × 11 is used, although 9 × 9 provides very similar results). More specifically, the result is computed by first multiplying each filter value by

```c
for(row = HalfKernelSize; row < this->ActualRows-HalfKernelSize; row++)
for(col = HalfKernelSize; col < this->ActualCols-HalfKernelSize; col++)
{
    // convolve the Kernel with the neighborhood around row,col
    ConvolutionResult = 0;
    for(int i=row-HalfKernelSize; i<Kernel->GetActualRows()-1; i+=row+HalfKernelSize)
    for(int j=col-HalfKernelSize; j<Kernel->GetActualCols()-1; j+=col+HalfKernelSize)
        ConvolutionResult += tempImage->matrix[r][c]*Kernel->matrix[i][j];
    // store the result for point row,col
    this->matrix[row][col] = ConvolutionResult;
}
```

(shown below as “inter” with two copies) and discuss on-chip resource utilization in the following section.

**Fig. 5.** (a) 9 × 9 filter computing circular convolution of top left portion of data. (b) Example two-directional filtering. (c) C++ code.

**Fig. 6.** Illustration of different possible digital filtering methods.
the corresponding input data value. Then a summation is performed, and the result is stored in a memory location that corresponds to the centroid of the filter. This process repeats for each pixel in the input data, stepping right, column-by-column, and down, row-by-row, until the filtering is complete, as shown in Fig. 5(a).

Finally, the template is generated by comparing the results of two different directional filters (horizontal and vertical) and writing a single bit that represents the filter with the highest output at the equivalent location. The output of each filter is compared and for each pixel, a “1” is assigned for strong vertical content or a “0” for strong horizontal content. These bits are concatenated to form a bit vector unique to the “iris signal” that conveys the identifiable information. This final step is demonstrated in Fig. 5(b). In this example, the two filters are represented in the middle of the diagram, while the image data is on the left. In this example, the filter provides higher output with the vertical filter, and hence a 1 bit is stored for this particular pixel location.

As described in prior art [3], there are three methods to compute a digital filter, as shown in Fig. 6. The first method uses a general purpose processor to compute the digital filter via software, executing a single instruction, and hence byte (shown as \( x \times 1 \)) at a time. This approach is acceptable if the digital filter is small; however, as the digital filter and data set to be filtered increases, the computational load rises exponentially. The second approach is to develop hardware specifically designed to compute the digital filter but by first loading all the data (all 81 bytes, illustrated \( x \times 81 \)) into an array of computing elements. By loading the data first (buffering the data into the hardware), this approach is similar to the sequential general purpose processor method due to overhead. Only partial calculations can be computed until the entire kernel is filled. The third and final approach is to redefine the filtering as a parallel process, allowing the hardware to take in a single byte of data and compute the maximum influence that data would have on the final results. The method is a truly parallel process and minimizes the need for overhead and buffering that other sequential processes use. Throughout this paper, the parallel method of approach will be investigated and compared in detail to sequential methods. Parallelizing the RED algorithm filtering can be done simply by doing the obvious—calculating the result of both filters at the same time and computing the single bit representation in the template simultaneously. Unfortunately, while this straightforward approach appears to cut out a large amount of intermediate memory, it in fact suffers from a very serious memory bottleneck. The energy data to compute a single result must first be fully loaded into a hardware representation of each filter. Assuming the filters could compute the final template bit in a single clock cycle, each filter must read 81 different 8-bit values of energy data before any filtering can be done. Therefore, a straightforward approach to digital filtering in hardware will suffer due to the overhead associated with waiting for data to be localized near the computing elements. In this case, since the computing elements are idle until the 81 data values are loaded, the process is fundamentally sequential much like the general purpose processor. Consequently, a straightforward filtering in hardware is inherently not parallel with a single bank of memory holding all the energy data.

To parallelize the digital filter, the memory itself must be parallelized. A possible but impractical option to parallelize the straightforward digital filter would be to create an 81 ported block of memory or 81 parallel blocks of memory that could supply all energy data to both filters in a single step. Either case would exceed the memory resources of even the most massive FPGAs while also potentially creating interconnect issues. Furthermore, even if the memory issue was inconsequential to the straightforward digital filtering, there is no way around the overhead associated with needing at least 81 energy data values and at least nine more properly arranged to step in either the horizontal or vertical directions.

There is an alternative way to accomplish the digital filtering that also offers greater opportunity for parallelization while also reducing the bandwidth requirements on memory. Our digital filtering is a commutative mathematical operation of the form

\[
 f(x, y) = g(x, y) \ast h(x, y)
\]

or

\[
 f(x, y) = h(x, y) \ast g(x, y).
\]

The resulting function \( f \) is either function \( g \) filtered with \( h \) or \( h \) filtered with \( g \). This concept is critical in modifying the digital filtering algorithm so that it can more easily be implemented in hardware. Typically, the filter is streamed overtop the input data, but for parallelizing purposes, the input data will be streamed over the centroid of the filter. This approach is analogous to dropping the energy data onto the centroid of the filter and computing the partial product of all results within scope. Commuting the filtering eliminates the overhead associated with buffering all the energy data into the filter to calculate a single result and greatly simplifies the hardware implementation (see Fig. 7).

The parallelized digital filters maximize the computations for any single 8 bits of energy data so that input buffering is kept at an absolute minimum while also offering quite a bit of freedom with regard to the form of the resulting data. The parallelized digital filter computes the partial products and writes the results to memory into the appropriately wrapped memory locations. While outside the scope of this paper, it is likely possible to compute the partial results directly into the finalized template, totally eliminating the need for intermediate memory but that is left to the discretion of the engineer to handle the filters in whatever way best fits the application.

The RED algorithm digital filtering is designed and implemented in VHDL software. The parallel hardware filter consists of 81 accumulator cores arranged in a nine by nine array to represent the original filter. The accumulators are grouped within the array by row since the energy data from the segmentation process (or most other common process) emerges left to right, top to bottom, easing the memory access as the filter array steps through the incoming data. Each row is accommodated with a block of simple dual ported ram to support all the simultaneous reading and writing needs of the accumulator core row. Also among each row is a multiplexer to narrow the control of memory to a single accumulator core and a row controller to grant any accumulator within the row memory control and issue the proper control signals to the row’s accumulator cores. Finally, the array is fitted with a first-in-first-out (FIFO) memory buffer to ease incoming data and a comparator module that judges which filter had the highest output and issues the appropriate bit to the finalized template for matching. The full
system is shown in Fig. 8. Due to the complexity, the VHDL code for this portion of our parallelization is not shown here.

C. Template Matching via Hamming Distance

The focus of this section is on parallelizing the template matching portion of the algorithm. The template matching portion of the algorithm is important due to its repeated execution (depending on the number of iris comparisons necessary). Illustrated in Fig. 9 is optimized C++ code for computing the fractional HD between two templates. In order to help count the number of 1’s in a binary number, the code contains a lookup table (Value). The lookup table holds the number of 1’s that are in that particular 16 bit number. This lookup table is small enough to fit into the cache and has been shown to execute faster because it requires less assembly instructions than a recursive or loop-based method.

We would like to highlight the sequential nature of this code. For example, since the XOR function is performed 32 bits at a time, a loop (FOR loop denoted below) is necessary. Since it is computing 2048 bits, this loop is executed 64 times. Also, note that the XOR and AND computations are also performed sequentially. These instructions could be scheduled to execute in parallel, but a modern CPU has a limited number of functional units, therefore limiting the amount of parallel execution. Finally, a score is computed as a ratio of the number of differences between the templates to the total number of bits that are not masked.

Illustrated in Fig. 9 is the associated assembly code created for the hamming distance calculation. The code is compiled for a Xeon Processor, and hence IA-32 assembly code is produced [11]. For each C++ computation, there are at least five assembly language instructions required. For example, for the AND computation that is in C++ code, there are four MOV instructions and one AND instruction. The MOV instructions are required to move data in and out of registers in preparation for the logical instruction. This AND instruction is a 32-bitwise computation performed by an ALU functional unit in the processor. Illustrated in Fig. 10 are the assembly instructions that are required for the loop instruction. Loop instructions require the aforementioned overhead. For each iteration of the loop, there is required a total of 38 assembly instructions. As stated above, this code requires 64 loops to perform a template match. Therefore, according to the following formula, a template match on a perfect 4-GHz 4 IPC microprocessor is 151 ns per match:

$$\frac{38 \text{ inst}}{\text{loop}} \times \frac{64\text{ loops}}{\text{match}} \times \frac{1 \text{ cycle}}{4\text{ inst}} \times \frac{1 \text{ sec}}{4 \text{ G cycles}} = \frac{151 \text{ nsec}}{\text{match}}.$$  

Ideally, if 2048 matching elements could fit onto the FPGA, then the iris recognition algorithm could be many times faster. Fig. 11 illustrates our VHDL code that implements the HD cal-
for(IntPtr1=(unsigned int *)&matrix[row][0],
    IntPtr2=(unsigned int *)&InMatrix->matrix[0][0],
    MaskPtr1=(unsigned int *)&Mask1->matrix[row][0],
    MaskPtr2=(unsigned int *)&Mask2->matrix[0][0];
    IntPtr1+=1,IntPtr2+=1,MaskPtr1+=1,MaskPtr2+=1;
}
    // AND two Masks using 32 bit pointers
    Mask = *MaskPtr1 & *MaskPtr2;
    // XOR templates, then AND with Masks using 32 bit pointers
    XOR = (*IntPtr1 ^ *IntPtr2) & Mask;
    // Sum lower 16 bits of XOR using lookup table
    Sum += Value[XOR & 0x000000ff];
    // Sum upper 16 bits of XOR
    Sum += Value[XOR>>16] & 0x000000ff;  
    // Sum lower 16 of Mask
    MaskSum += Value[Mask & 0x000000ff];
    // Sum upper 16 of Mask
    MaskSum += Value[(Mask>>16) & 0x000000ff];
}
Score = (float)Sum/(float)MaskSum;

Mask = *(MaskPtr1 ^ *MaskPtr2); // AND two Masks using 32 bit pointers
0x0401D63 mov ecx,dword ptr [ebp-24h]
0x0401D66 mov edx,dword ptr [ebp-28h]
0x0401D69 mov eax,dword ptr [ecx]
0x0401D6B mov eax,dword ptr [edx]
0x0401D6D mov dword ptr [ebp-30h],eax
XOR = (*IntPtr1 ^ *IntPtr2) & Mask; // XOR templates, then AND with Masks using 32 bit pointers
0x0401D70 mov ecx,dword ptr [ebp-1ch]
0x0401D73 mov edx,dword ptr [ebp-20h]
0x0401D76 mov eax,dword ptr [ecx]
0x0401D78 xor eax,dword ptr [edx]
0x0401D7A and eax,dword ptr [ebp-30h]
0x0401D7D mov dword ptr [ebp-2ch],eax
Sum = Value[XOR & 0x000000ff]; // Sum lower 16 bits of XOR using lookup table
0x0401D80 mov ecx,dword ptr [ebp-2ch]
0x0401D83 mov ecx,0FFh
0x0401D89 mov edx,dword ptr [ebp-34h]
0x0401D8C add edx,dword ptr [ecx+4+4519E0h]
0x0401D93 mov dword ptr [ebp-34h],edx
Sum = Value[XOR>>16] & 0x000000ff; // Sum upper 16 bits of XOR
0x0401D96 mov eax,dword ptr [ebp-2ch]
0x0401D99 shr eax,10h
0x0401D9C mov eax,0FFh
0x0401DA1 mov ecx,dword ptr [ebp-34h]
0x0401DA4 add ecx,dword ptr [eax+4+4519E0h]
0x0401DAB mov dword ptr [ebp-34h],ecx
MaskSum += Value[Mask & 0x000000ff]; // Sum lower 16 bits of Mask
0x0401DAE mov edx,dword ptr [ebp-30h]
0x0401DB1 mov edx,0FFh
0x0401DB7 mov eax,dword ptr [ebp-38h]
0x0401DBA add eax,dword ptr [edx*4+4519E0h]
0x0401DC1 mov dword ptr [ebp-38h],eax
MaskSum += Value[(Mask>>16) & 0x000000ff]; // Sum upper 16 bits of Mask
0x0401DCA mov ecx,dword ptr [ebp-30h]
0x0401DC7 shr ecx,10h
0x0401DCA mov ecx,0FFh
0x0401DD0 mov edx,dword ptr [ebp-38h]
0x0401DD3 add edx,dword ptr [ecx+4+4519E0h]
0x0401DDA mov dword ptr [ebp-38h],edx

D. Future Topics
Segmentation in iris recognition algorithms requires substantial computer vision and digital signal processing techniques to accurately detect the inner and outer boundaries of an iris embedded in a bitmap image. Initially, finding the pupil as a large dark object in a bitmap image can be complicated by glare and eccentricities which can easily throw off an algorithm. To mitigate anomalies in the pupil, a series of dilations and erasures expand the black pupil, close gaps created by glare, and return the pupil to its original shape. After dilation and erosion, a computer can better detect the shape of the pupil as a characteristic

calculation. Here we perform the same function as the aforementioned C++ code. However, we are doing this computation completely in parallel. There are 2048 XOR gates and 4096 AND gates required for this computation. In addition, adders are required for summing and calculating the score.

The iris templates must be stored either in memory on the FPGA or off-chip. In one instance of our implementation, we have implemented a 2048-bit-wide memory in VHDL. We have added this to our code to verify that a small database can be stored on chip. One of the two templates compared is received from this dual-ported 2048-bit-wide single cycle cache implemented on our FPGA. Therefore, once per clock cycle, a 2048-bit vector is fetched from on-chip memory, and the HD calculation is performed. We have successfully implemented and tested the HD calculation with and without a memory device.

Fig. 9. Template matching C++ and intel assembly.

```
1764: for(row = 0; row < ActualRows; row++)
0x0401C5D mov dword ptr [ebp-14h],0
0x0401CDE jmp Matrix<unsigned char>::HammingDistanceFast_NoShift+2F7h (0x0401ce7)
0x0401CED mov ecx,dword ptr [ebp-14h]
0x0401CE1 add ecx,1
0x0401CE4 mov dword ptr [ebp-14h],ecx
0x0401CE7 mov edx,dword ptr [ebp-10h]
0x0401CEA mov eax,dword ptr [ebp-14h]
0x0401CED cmp eax,dword ptr [edx+4]
0x0401CF0 jge Matrix<unsigned char>::HammingDistanceFast_NoShift+449h (0x0401e39)
1765:
```

Fig. 10. Overhead assembly instructions for loop.
large dark object with a center point and radius, setting it apart from eyebrows and eyelashes which remain irregular in shape.

In the RED iris recognition algorithm, dilation and erosion represent approximately 30% of the segmentation execution time in software. A proposed alternative implements the dilation and erosion as an array of cells within hardware-based architecture such as an FPGA. We expect a parallel FPGA implementation to be complete after only a few clock cycles compared to the millions needed in the software dilation and erosion method.

IV. RESULTS AND ANALYSIS

A. Hardware Details

The CPU experiment is executed on an Intel Xeon X5355 [12] workstation class machine. The processor is equipped with eight cores, 2.66-GHz clock, and an 8-MB L2 cache. While there are eight cores available, only one core is used to perform this test, therefore allowing all cache and memory resources for the code under test. The HD code was compiled under Windows XP using the Visual Studio software suite. The code has been fully optimized to enhance performance. Additionally, millions of matches were executed to ensure that the templates are fully cached in the on-chip L2 cache.

The RED algorithm optimized C++ code time is faster than some of the times reported in the literature for other commercial iris recognition implementations [27]. For example, the template matching time published in [27] is 10 μm, whereas our template matching time is 383 ns. We attribute this difference to: 1) our use of a faster, more modern CPU operating at 2.66 GHz vice 300 MHz; 2) the RED algorithm is fully optimized; and 3) the RED algorithm is different from [27]. An extrapolation of the execution times of [27] on a modern computing environment at 2.66 GHz would leave it still slower, but closer to the
RED execution times. Our goal here is not to provide a direct comparison of the execution times of these two iris recognition algorithms. However, this indicates that our C++ code is a reasonable target for comparison to the FPGA performance. Furthermore, due to iris recognition algorithm similarity, one can reasonably expect similar improvements from application mapping to FPGA technology for other iris recognition algorithms.

The FPGA experiment is executed on a DE2 [13] board provided by Altera Corporation. The DE2 board includes a Cyclone-II EP2C35 FPGA chip, as well as the required programming interface. The DE2 board is illustrated in Fig. 12. Although the DE2 board is utilized for this research, only the Cyclone-II chip is necessary to execute our algorithm. The Cyclone-II [14] family is designed for high-performance low-power applications. It contains over 30 000 logic elements (LEs) and over 480 000 embedded memory bits. In order to program our VHDL onto the Cyclone-II, we utilize the Altera Quartus software for implementation of our VHDL program. The Quartus suite includes compilation, synthesis, simulation, and programming environments. We are able to determine the size required of our program on the FPGA, and the resulting execution time.

### B. Comparisons

1) **Speedup:** All VHDL code is fully synthesizable and is downloaded onto our DE2 for direct hardware execution. As discussed above, our code is fully contained within a "process" statement. The process statement is only initiated when a signal in its sensitivity list changes values. The sensitivity list of our process contains the clock signal and, therefore, the code is executed once per clock cycle. In this code, the clock signal is drawn from our DE2 board which contains a 50-MHz clock. Therefore, every 20 ns, our calculation is computed. Note that for template generation, this process takes many clock cycles, and therefore, the total time for template generation with our FPGA is 98.6 us.

Fig. 13 illustrates the execution times and acceleration achieved for our implemented FPGA version on the Cyclone-II EP2C35 versus a Xeon-based C++ version. For example, the optimized C++ version takes 96.8 ns per match while the FPGA takes 20 ns per match for the kurtosis function. For the kurtosis implementation where we have two concurrent implementations on our FPGA, execution time is cut in half to 10 ns.

The main result of this research is that the implementation on a modest sized FPGA is approximately 9.6, 324, and 19 times faster than a state-of-the-art CPU design for the kurtosis, filtering, and hamming distance calculations, respectively. Also illustrated in Fig. 13 are the theoretical best performances for a future Intel-based machine, assuming a 4-GHz clock speed and four ALUs. The parallel algorithm on our FPGA is approximately 2.3, 4.7, and 7.5 times faster than the theoretical best CPU design for the kurtosis, filtering, and hamming distance calculations, respectively. If Intel were to design a much faster microprocessor with a perfect compiler, it still would be orders of magnitude slower than an off-the-shelf inexpensive low-end FPGA.

2) **Hardware Usage:** Fig. 14 presents the utilization of the FPGA LEs for our hardware implementations. For example, our implementation of the "inter" with two-copy kurtosis function parallelization utilizes 9% of our Cyclone-II FPGA. We found that the single-copy version utilized 0% of the LEs, but exhausted all of the on-chip multipliers. Only 9% of our chip is utilized by the second kurtosis function implementation.

For the template generation code, a much more parallel and hardware intensive approach is utilized, therefore extending the

<table>
<thead>
<tr>
<th></th>
<th>Optimized C++ Code on Xeon CPU (ns)</th>
<th>Theoretical Best on Any CPU</th>
<th>Cyclone-II EP2C35 (50 MHz)</th>
<th>Stratix IV estimated@ 500 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kurtosis</td>
<td>96.8 ns</td>
<td>46 ns</td>
<td>20 ns (“intra”)</td>
<td>2 ns (“intra”)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10 ns (“inter” with 2 copies)</td>
<td>1 ns (“inter” with 2 copies)</td>
</tr>
<tr>
<td>Digital</td>
<td></td>
<td></td>
<td>98.6 us</td>
<td>9.8 us</td>
</tr>
<tr>
<td>Filtering</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hamming</td>
<td>383 ns</td>
<td>151 ns</td>
<td>20 ns</td>
<td>2 ns</td>
</tr>
<tr>
<td>Distance</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 13. FPGA versus CPU comparison for iris match execution.

<table>
<thead>
<tr>
<th></th>
<th>% of Logic Elements on Cyclone II</th>
<th>% Logic Elements estimated on Stratix IV</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kurtosis</td>
<td>9%</td>
<td>9%</td>
<td>Utilizes faster on-chip multipliers (19/19 consumed). After multipliers are exhausted, each parallel instantiation of kurtosis consumes 9% of Cyclone II.</td>
</tr>
<tr>
<td>Digital</td>
<td>71%</td>
<td>7.1%</td>
<td>This algorithm is hardware intensive. It also consumes 17% of on-chip memory.</td>
</tr>
<tr>
<td>Filtering</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hamming</td>
<td>73%</td>
<td>7.3%</td>
<td>Storing a template consumes 0.7% of Cyclone Memory. Therefore, approximately 230 irises can be stored on the Cyclone II. A tenfold increase can be stored on the Stratix IV.</td>
</tr>
</tbody>
</table>

Fig. 14. FPGA hardware usage.
usage of the Cyclone-II chip. As illustrated, the template generation consumes 71% of the Cyclone-II LEs, and 17% of memory bit availability.

Our implementation of the hamming distance algorithm utilizes 73% of our Cyclone-II FPGA. In terms of on-chip memory usage, one of the two templates compared is stored in this dual-ported 2048-bit-wide single-cycle cache implemented on our Cyclone-II FPGA. Each stored template consumes 0.7% of on-chip memory. We have added this to our code to verify that a small database of approximately 230 can be stored on chip.

3) Projections and Other Issues: The Cyclone-II is not a state-of-the-art design. A projection of the performance and hardware usage of a state-of-the-art Stratix IV at 500-MHz [23] FPGA is given in Figs. 13 and 14, respectively. For example, given traditional scaling of FPGAs, we anticipate that a Stratix-IV at 500 MHz would be able to perform the hamming distance computation in only 2 ns. Therefore, a state-of-the-art FPGA can outperform a state-of-the-art computer by 90, 1500, and 190 times faster than a state-of-the-art CPU design for the kurtosis, filtering, and hamming distance calculations, respectively. The hardware usage of the Stratix-IV for our three components is also drastically reduced. For example, for the kurtosis function, parallelism can be increased since each 5 × 5 matrix computation will now only consume less than 1% of the chip. All told, the three algorithms now only consume 0.9%, 7.1%, and 7.3% of the Stratix-IV for kurtosis, filtering, and hamming distance calculations, respectively. Therefore, a full implementation of a very fast iris recognition algorithm is more than feasible.

On-chip memory for the Stratix-IV is also much larger with 22.4 Mbits of on-chip storage [23]. This is important for both iris template storage and digital filtering. For example, a database consisting of 10,000 irises can be stored on the Stratix-IV. We anticipate this storage scaling trend to continue into the future, with larger and larger database storage available. If a larger database is necessary, we propose an implementation where a DRAM chip is provided as part of the package, and the on-chip database is concurrently loaded while hamming distances are being computed.

Although we have not yet empirically analyzed it, another advantage of this FPGA-based technique is power and energy reduction. FPGAs, in general, consume far less power than their CPU counterparts. A detailed study of the power and energy advantages of this implementation is outside the scope of this study and left for future work.

V. CONCLUSION

In this paper, we have provided novel hardware implementations which enabled us to discover that three key portions of an iris recognition algorithm can be parallelized. The main result is that the implementation on a modest sized FPGA is approximately 9.6, 324, and 19 times faster than a state-of-the-art CPU design for portions of iris segmentation, template creation, and template matching components, respectively. Furthermore, the parallel algorithm on our FPGA also greatly outperforms our calculated theoretical best Intel CPU design. Finally, on a state-of-the-art FPGA, we conclude that a full implementation of a very fast iris recognition algorithm is more than feasible, providing a potential small form-factor solution.

FPGAs have been on an impressive scaling trend over the last 10 years. They are increasing in capacity, in terms of LEs, increasing in memory, in terms of on-chip bit storage, increasing in speed, and decreasing in power consumption. We expect this scaling trend to continue in the short term and we are excited to study the different advantages of FPGAs, including power consumption. Given the commercial and military demands of an accurate, timely, and small iris recognition system, we believe an FPGA-based option presents an exciting parallel alternative for this parallel application.

REFERENCES

C. Park, J. Lee, M. Smith, and K. Park, “Iris-based personal authentica-
tion using a normalized directional energy feature,” in *Proc. 4th Int.

R. P. Broussard, R. N. Rakvic, and R. W. Ives, “Accelerating iris tem-
plate matching using commodity video graphics adapters,” in *2008
IEEE Int. Conf. Biometrics: Theory, Applications and Systems*, Crystal

N. A. Schmid, M. Ketkar, H. Singh, and B. Cukic, “Performance analysis 
of iris-based identification system at the matching score level,”
2006.

F. Hao, J. Daugman, and P. Zielinski, “A fast search algorithm for a 


D. M. Monro, S. Rakshit, and D. Zhang, Iris Image Database University 
pages/kipg/irisweb

CASIA Iris Image Database [Online]. Available: http://www.sinobi-
ometrics.com

Ryan N. Rakvic (M’01) received the B.S. degree in computer engineering from 
the University of Michigan and the M.S. and Ph.D. degrees in com-
puter engineering from Carnegie Mellon University.

He is currently an Assistant Professor in the Electrical and Computer Engineering Department, U.S. Naval Academy, Annapolis, MD. Prior to 
that, he spent five years in the Computer Research Laboratory at Intel Corporation, Santa Clara, CA. During his time at Intel, he invented creative ways to increase the performance of Pentium micropro-
cessors. More recently, he is attempting to utilize the emerging efficiency of field-programmable gate arrays (FPGAs) to outperform a general purpose microprocessor both in terms of performance and power efficiency.

Randy P. Broussard received the B.S. degree in electrical engineering from Tulane University in 1986, the M.S. degree in computer engineering from Florida Institute of Technology in 1991, and the Ph.D. degree in electrical engineering from the Air Force Institute of Technology in 1997.

He is currently an Assistant Professor in the Department of Systems Engineering, U.S. Naval Academy, where he teaches computer engineering, computer vision, and control systems. His research includes the areas of pattern recognition, iris recogni-
tion, neural networks, breast cancer detection.

Robert W. Ives (S’96–M’97–SM’03) received the B.S. degree in mathematics from the U.S. Naval Academy, Annapolis, MD, the M.S. degree in elec-
trical engineering from the U.S. Naval Postgraduate School, Monterey, CA, and the Ph.D. degree in electrical engineering from the University of New 
Mexico, Albuquerque.

He served 20 years in the U.S. Navy, including 5 1/2 years as a military researcher at Sandia Na-
tional Laboratories, Albuquerque, and three years as an Assistant Professor in the Electrical Engineering Department, U.S. Naval Postgraduate School. After retiring from the Navy, he returned to the U.S. Naval Academy, where he is currently an Associate Pro-
fessor, and is the Director of the USNA Center for Biometric Signal Processing. His research interests include biometric signal processing, image processing and compression, adaptive signal processing, information theory, and communica-
tions.

Neil Steiner received the B.A. degree from Wheaton 
College, the B.S.E.E. degree in electrical engineering from 
Illinois Institute of Technology, the M.S. degree in electrical engineering from Virginia Tech., and the Ph.D. degree in electrical engineering from Virginia Tech. in 2008.

He is a Research Programmer at the University of Southern California’s Information Sciences Institute. His research interests include autonomous computing systems, hardware–software interaction, low-level FPGA configurations, and trusted inte-
grated circuits.