Hindawi Publishing Corporation
EURASIP Journal on Advances in Signal Processing
Volume 2010, Article ID 764838, 7 pages
doi:10.1155/2010/764838

Research Article

Comparing an FPGA to a Cell for an Image Processing Application

Ryan N. Rakvic,[1] Hau Ngo,[1] Randy P. Broussard,[2] and Robert W. Ives[1]

[1] *Department of Electrical and Computer Engineering, U.S. Naval Academy, Annapolis, MD 21402-5000, USA*
[2] *Department of Systems Engineering, U.S. Naval Academy, Annapolis, MD 21402-5000, USA*

Correspondence should be addressed to Ryan N. Rakvic, rakvic@usna.edu

Received 2 December 2009; Accepted 8 March 2010

Academic Editor: Yingzi Du

Modern advancements in configurable hardware, most notably Field-Programmable Gate Arrays (FPGAs), have provided an exciting opportunity to discover the parallel nature of modern image processing algorithms. On the other hand, PlayStation3 (PS3) game consoles contain a multicore heterogeneous processor known as the Cell, which is designed to perform complex image processing algorithms at a high performance. In this research project, our aim is to study the differences in performance of a modern image processing algorithm on these two hardware platforms. In particular, Iris Recognition Systems have recently become an attractive identification method because of their extremely high accuracy. Iris matching, a repeatedly executed portion of a modern iris recognition algorithm, is parallelized on an FPGA system and a Cell processor. We demonstrate a 2.5 times speedup of the parallelized algorithm on the FPGA system when compared to a Cell processor-based version.

1. Introduction

For most of the history of computing, the amazing gains in performance we have experienced were due to two factors: decreasing feature size and increasing clock speed. However, there are fundamental physical limits to this approach—decreasing feature size gets more and more expensive and difficult due to the physics of the photolithographic process used to make CPUs, and increasing clock speed results in a subsequent increase in power consumption and heat dissipation requirements. Parallel computation has been in use for many years in high performance computing; however, in recent years, multicore architectures have become the dominate computer architecture for achieving performance gains. The signal of this shift away from ever increasing clock speeds occurred when Intel Corporation cancelled development of its new single core processors to focus development on dual core technology. Executing programs in parallel on hardware specifically designed with parallel capabilities is the new model to increase processor capabilities while not entering into the realm of extensive cooling and power requirements.

The Cell processor is a joint effort by Sony Computer Entertainment, Toshiba Corporation, and IBM that began in 2000, with the goal of designing a processor with performance of an order of magnitude over that of desktop systems shipping in 2005. The result was the first-generation Cell Broadband Engine (BE) processor, which is a multicore chip comprised of a 64-bit Power Architecture processor core and eight synergistic processor cores. A high-speed memory controller and high-bandwidth bus interface are also integrated on-chip [1].

The Cell processor, shown in Figure 1, has a unique heterogeneous architecture compared to the homogeneous Intel Core architecture. It has a main processor called the Power Processing Element (PPE) (a two-way SMT PowerPC based processor), and eight fully functional coprocessors called the Synergistic Processing Elements, or SPEs. The PPE directs the SPEs where the bulk of the computation occurs. The PPE is intended primarily for control processing, running operating systems, managing system resources, and managing SPE threads. The SPEs are single-instruction, multiple-data (SIMD), shown in Figure 1, processors with a RISC core [2].

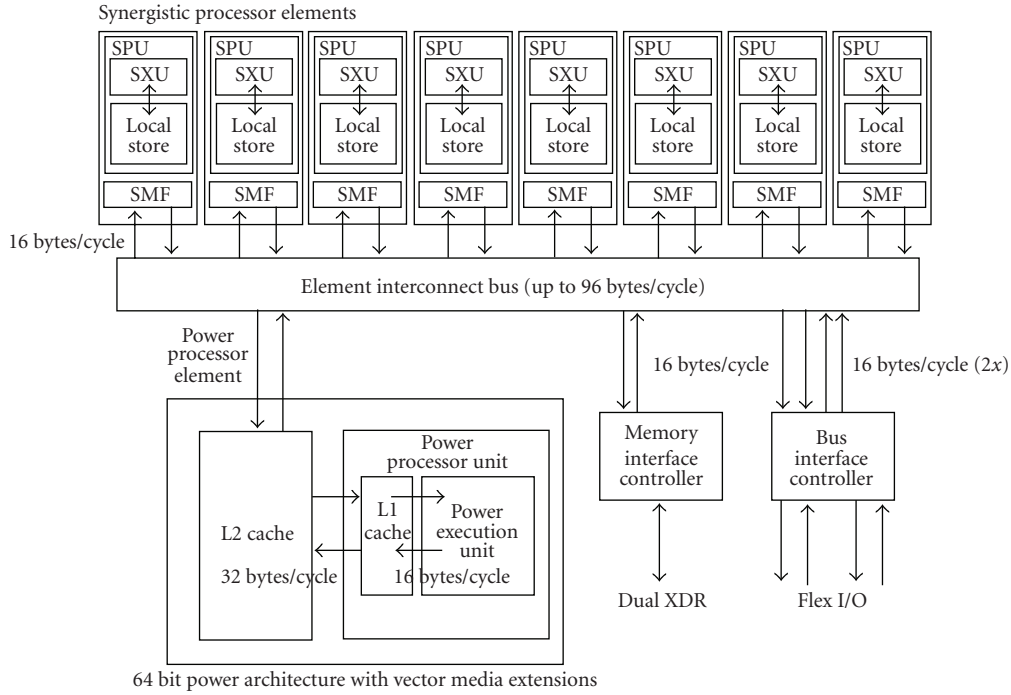

Figure 1: Cell BE high-level architecture diagram.

According to IBM, the Cell BE is capable of achieving in many cases 10 times the performance of the latest PC processors [3]. The first major commercial application of the Cell processor was in Sony's PlayStation3 game system. The PlayStation3 has only 6 SPU cores available due to one core being reserved by the OS and 1 core being disabled in order to increase production yields. Sony has made it very easy to install a new Linux-based operating system onto the PlayStation3, thereby making the game system a popular choice for experimenting with the Cell BE.

Historically programmers have thought in sequential terms, and programming these multicore processors can be difficult. Often times, this involves completely redesigning an existing program from the ground up and implementing complex synchronization protocols. Parallel programming is based on the simple idea of division of labor—that large problems can be broken up into smaller ones that can be worked on simultaneously. Making it more challenging is the fact that the SPEs in the Cell do not share memory with the PPE. Additionally, they are not visible to the operating system, thereby leaving all management of SPE code and data to the programmer.

Another popular approach to parallelization is to use Field Programmable Gate Arrays (FPGAs). FPGAs are complex programmable logic devices that are essentially a "blank slate" integrated circuit from the manufacturer and 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 VHDL (VHSIC Hardware Description Language). 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 many ALUs that execute in parallel, then you program this in the VHDL code.

In this work, we have parallelized a repeatedly executed portion of an image processing algorithm with both an FPGA and a Cell processor. In Section 2 we present the Iris Recognition Algorithm and iris template matching. In Section 3, we present an approach to iris matching utilizing parallel logic with field-programmable gate arrays and cell processors. In Section 4 we demonstrate this efficiency with a comparison between the FPGA, the Cell processor, and a sequential processor. We provide concluding statements in Section 5.

2. Iris Recognition Algorithm

Iris recognition stands out as one of the most accurate biometric methods in use today. One of the first iris recognition algorithms was introduced by pioneer Dr. John Daugmann [4]. An alternate iris recognition algorithm, referred to as the Ridge Energy Direction (RED) algorithm [5], will be the basis for this work. There are many iris detection algorithms. What follows is a brief description of the RED algorithm. Since this research is focused on computational acceleration, we refer the reader to [6–12].

The iris is the colored part of the eye, protected by the cornea that extends from the pupil to the white of the eye. Its patterns remain stable over a lifetime. An example iris image is depicted in Figure 2. Typically an iris image is captured in

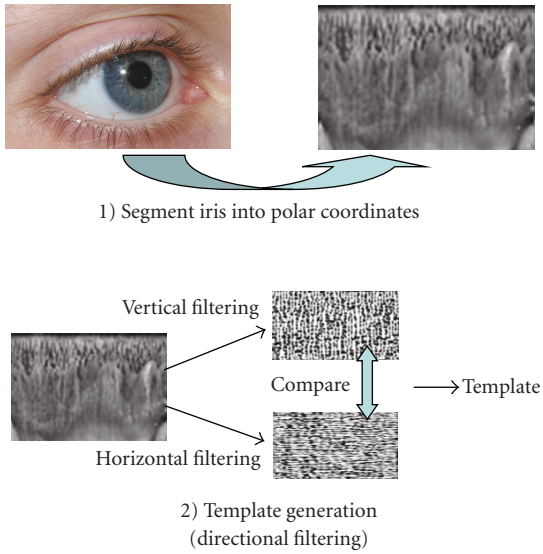

FIGURE 2: Red iris recognition algorithm. Visible is the associated two dimensional encoding of the iris image into energy data [14].

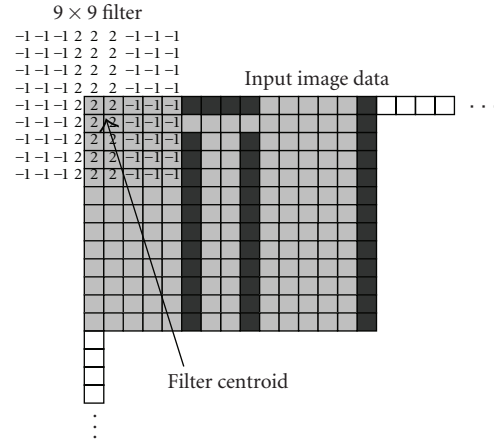

FIGURE 3: 9 × 9 filter computing the filtering of the top left portion of hypothetical input energy data. In this instance, each coefficient of the filter is multiplied by the corresponding image data within the scope of the filter (filter kernel) where some of the data is repeated from the opposite side. These filter coefficient and input data products make up a partial result, the sums of which generate a local result corresponding to the centroid of the filter.

the Near Infrared light spectrum. Most iris capture systems have dedicated illumination and capture a 640 by 480 pixels image containing eight bits per pixel. Once a digital image of the iris is captured, the system begins processing the image to transform it from a two-dimensional array of pixels to a two dimensional encoded string of bits for comparison (see "Segment Iris into Polar Coordinates" in Figure 2). In this, the first step is to identify the iris among other facial elements such as the eyelids, sclera (white part of the eye), pupil (dark circle in the center of the eye), and eyelashes. The algorithm finds the pupil by thresholding the image and using basic features such as circularity to find the most circle-like object in the thresholded image. The outer boundary is found using local kurtosis which has near-zero values at the boundary. Details of this segmentation method are described in prior art [13]. Once these boundaries are located, the computer can now extract only the meaningful portions of the iris.

Once the iris is segmented, the algorithm takes the iris and divides it into m concentric annuli and n radial lines, which results in an $m \times n$ representation of the iris. This step is effectively a rectangular to polar coordinate conversion. The energy of each pixel is merely the square of the value of the infrared intensity within the pixel and is used to distinguish features within the iris. The next step is to encode the iris image from two dimensional brightness data down to a two dimensional binary signature, referred to as the template ("Template Generation" in Figure 2), to accomplish this, the energy data are passed into two directional filters to determine the existence of ridges and their orientation. The RED algorithm uses directional filtering to generate the iris template, a set of bits that meaningfully represents a person's iris.

To help perform this filtering, the energy data passed from the iris segmentation process is made periodic in the horizontal dimensions to account for edge effects when performing the rectangular to polar conversion. The filter passes over this periodic array taking in 81 (9×9) values at a time (note, in [5], 11×11 is used). More specifically, the result is computed by first multiplying each filter value by the corresponding energy 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 energy data, stepping right, column-by-column, and down, row-by-row, until the filtering is complete as shown in Figure 3. Finally, the template is generated by comparing the results of two different directional filters (horizontal and vertical, see Figure 3) 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. In this study, we assume that a template consists of 2048 bits, representing the uniqueness of the iris.

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.

Once encoded, the iris recognition system must be able to reliably match the newly created template with a database of previously enrolled templates. The newly encoded iris is compared to a database of previously created templates using a fractional Hamming Distance (HD) calculation, which is defined in (1). This is illustrated in Figure 4

$$\mathrm{HD} = \frac{\|(\text{template A} \otimes \text{template B}) \bigcap \text{mask A} \bigcap \text{mask B}\|}{\|\text{mask A} \bigcap \text{mask B}\|}. \quad (1)$$

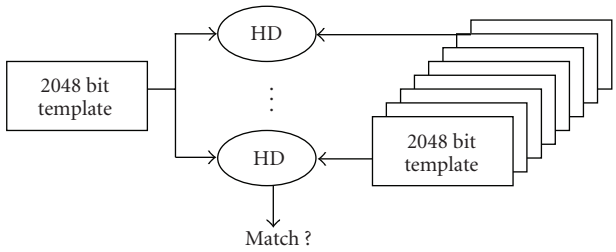

Figure 4: New template is compared with each template stored in a database.

The $\otimes$ operator is the exclusive-or operation used to detect disagreement between corresponding bit pairs 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 valid bits are included in the calculation, after artifacts are discounted. 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 HD calculation, or iris matching, is critical to the throughput performance of iris recognition since this task is repeated many times, seen in Figure 4. Traditional systems for HD calculation have been coded in sequential logic (software); databases have been spread across multiple processors to take advantage of the parallelism of the database search, but the inherent parallelism of the HD calculation has not been fully exploited.

3. Implementations

3.1. Sequential on a CPU. Currently, iris recognition algorithms are deployed globally in a variety of systems ranging from computer access to building security to national size databases. These systems typically use central processing unit- (CPU-) based computers. CPU based computers are general purpose machines, designed for all types of applications and are to first order programmed as sequential machines, though there are provisions for multiprocessing and multithreading. Recently, there has been an interest in exploring the parallel nature of this application [15]. It is challenging to exploit the inherent parallelism of many algorithms in such architectures.

In particular, the matching portion of the algorithm is important since it needs to be repeated many times (depending on the number of iris comparisons necessary). Illustrated in Figure 5 is optimized C++ code for computing the fractional HD between two templates. The optimizations in this code include the use of 32-bit logical operations and the use of a lookup table for bit counting.

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) 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. Summation of the bits is performed using lookup tables. Finally, the HD 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 Figure 6 is the associated assembly code created for the hamming distance calculation. The code is compiled for an Xeon Processor, and hence IA-32 assembly code is produced [16]. For each C++ computation, there are at least 5 assembly language instructions required. For example, the AND computation that is in C++ code generates 4 MOV instructions and one AND instruction. The MOV instructions are required to move data to and from memory. The AND instruction is a 32-bitwise computation performed by an ALU functional unit in the processor. As stated before, instruction execution bandwidth for a processor is limited by the number of functional units that it has. Loop instructions require overhead assembly instructions to again move the proper data to and from memory. For each iteration of the loop, there is required a total of 38 assembly instructions. Therefore, this code requires 64 loops $\times$ 38 assembly instructions to perform one template match.

3.2. Parallel on an FPGA. Field Programmable Gate Arrays (FPGAs) are complex programmable logic devices that are essentially a "blank slate" integrated circuit from the manufacturer and 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 VHDL (VHSIC Hardware Description Language). VHDL statements are inherently parallel, not sequential. VHDL allows the programmer to dictate the type of hardware that is synthesized on an FPGA. Ideally, if 2,048 matching elements could fit onto the FPGA, all 2048 bits of the template could be compared at once, with a corresponding increase in throughput. Here we perform the same function as the aforementioned C++ code. However, we are doing this computation completely in parallel. There are 2,048 XOR gates and 4,096 AND gates required for this computation. In addition, adders are required for summing and calculating the score.

This code is contained within a "process" statement. The process statement is only initiated when a signal in the sensitivity list changes values. The sensitivity list of the 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 FPGA board which contains a 50 Mhz clock. Therefore, every 20 ns, this hamming distance calculation is computed. This code is fully synthesizable and can be downloaded onto an FPGA for direct hardware execution.

```
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     <(unsigned int *)&matrix[row][ActualCols - 4];
    IntPtr1++,IntPtr2++,MaskPtr1++,MaskPtr2++)
    {
              // AND two Masks using 32 bit pointers
              Mask = *MaskPtr1 & *MaskPtr2;
              // XOR templates, AND with Masks using 32 bit
pointers
              XOR = (*IntPtr1 ^ *IntPtr2) & Mask;
              // Sum lower 16 bits of XOR using lookup table
              Sum += Value[XOR & 0x0000ffff];
              // Sum upper 16 bits of XOR
              Sum += Value[(XOR>>16) & 0x0000ffff];
              // Sum lower 16 of Mask
              MaskSum += Value[Mask & 0x0000ffff];
              // Sum upper 16 of Mask
              MaskSum += Value[(Mask>>16) & 0x0000ffff];
    };

Score->matrix[row][0] = (float)Sum/(float)MaskSum;
```

Figure 5: C++ code for fractional Hamming Distance Computation.

3.3. Parallel on a CELL. We have also parallelized the HD calculation on the Cell processor on the PlayStation3. As stated before, SPE management is left entirely to the programmer. We therefore have completely separate code and compilations for the PPE and the SPEs. The code on the PPE works as a slave master, spawning off threads of work to the 6 individual SPEs. The work is divided up on iris template matching boundaries, not within a template match. Therefore, each SPE is individually responsible for 1/6th of the HD comparisons. To maximize performance, the HD calculation is vectorized on the SPEs, taking advantage of the SIMD capabilities of the SPU's.

4. Results

The CPU experiment is executed on an Intel Xeon X5355 [17] workstation class machine. The processor is equipped with 8 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. We report the best-case per match execution time.

The PlayStation3 is used for our Cell experiments. Fedora Core 8 was chosen for installation onto the PlayStation3. Fedora Core 8 is not the most recent release of Fedora but was chosen because it is the most recent release that has been fully adapted to the PlayStation3. Additionally, the installation procedures available online for FC8 are the most detailed and complete of any Linux distribution. Furthermore, the IBM SDK, which is required for writing code that runs on the Cell's SPUs, is specifically only released for the commercial Red Hat Enterprise Edition Linux or the freely available Fedora Core.

The FPGA experiment is executed on a DE2 [18] board provided by Altera Corporation. The DE2 board includes a Cyclone-II EP2C35 FPGA chip, as well as the required programming interface. Although the DE2 board is utilized for this research, only the Cyclone-II chip is necessary to execute our algorithm. The Cyclone-II [19] family is designed for high-performance, low-power applications. It contains over 30,000 logic elements (LE) 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. The optimized C++ code time is actually faster than some of the times reported in the literature for commercial implementations [20]. We attribute this difference to improvements in CPU speed and efficiency between the time of our experiments and the previous reports. However, this indicates that our C++ code is a reasonable target for comparison and that we may reasonably expect similar improvements from application of FPGA technology to other HD-based algorithms.

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.

Table 1 illustrates the execution times and acceleration achieved for our implemented FPGA version on the Cyclone-II EP2C35, a CELL-based version and an Xeon-based C++ version. The optimized C++ version takes 383 ns per match, the CELL version with 6 SPEs takes 50 ns, and the FPGA takes 20 ns per match. The main result in this research is that the

```
Mask = *MaskPtr1 & *MaskPtr2;// AND Masks with 32 bit pointers

00401D63   mov         ecx,dword ptr [ebp-24h]
00401D66   mov         edx,dword ptr [ebp-28h]
00401D69   mov         eax,dword ptr [ecx]
00401D6B   and         eax,dword ptr [edx]
00401D6D   mov         dword ptr [ebp-30h],eax

XOR = (*IntPtr1 ^ *IntPtr2) & Mask;

00401D70   mov         ecx,dword ptr [ebp-1Ch]
00401D73   mov         edx,dword ptr [ebp-20h]
00401D76   mov         eax,dword ptr [ecx]
00401D78   xor         eax,dword ptr [edx]
00401D7A   and         eax,dword ptr [ebp-30h]
00401D7D   mov         dword ptr [ebp-2Ch],eax

Sum += Value[XOR & 0x0000ffff]; // Sum lower 16 bits of XOR
using lookup table

00401D80   mov         ecx,dword ptr [ebp-2Ch]
00401D83   and         ecx,0FFFFh
00401D89   mov         edx,dword ptr [ebp-34h]
00401D8C   add         edx,dword ptr [ecx*4+4519E0h]
00401D93   mov         dword ptr [ebp-34h],edx

Sum += Value[(XOR>>16) & 0x0000ffff]; // Sum upper 16 bits XOR

00401D96   mov         eax,dword ptr [ebp-2Ch]
00401D99   shr         eax,10h
00401D9C   and         eax,0FFFFh
00401DA1   mov         ecx,dword ptr [ebp-34h]
00401DA4   add         ecx,dword ptr [eax*4+4519E0h]
00401DAB   mov         dword ptr [ebp-34h],ecx

MaskSum += Value[Mask & 0x0000ffff]; // Sum lower 16 bits of
Mask

00401DAE   mov         edx,dword ptr [ebp-30h]
00401DB1   and         edx,0FFFFh
00401DB7   mov         eax,dword ptr [ebp-38h]
00401DBA   add         eax,dword ptr [edx*4+4519E0h]
00401DC1   mov         dword ptr [ebp-38h],eax

MaskSum += Value[(Mask>>16) & 0x0000ffff]; // Sum upper 16
bits of Mask

00401DC4   mov         ecx,dword ptr [ebp-30h]
00401DC7   shr         ecx,10h
00401DCA   and         ecx,0FFFFh
00401DD0   mov         edx,dword ptr [ebp-38h]
00401DD3   add         edx,dword ptr [ecx*4+4519E0h]
```

Figure 6: C++ code (highlighted) and IA-32 Assembly Code for Hamming Distance Calculations.

Table 1: FPGA versus CPU comparison for iris match execution.

	Optimized Xeon Code on PS3	CELL (with 6 SPEs)	Cyclone-II EP2C35 (50 MHz)	Cyclone-II estimated @ 100 MHz	Stratix IV estimated @ 500 MHz
Time per match (ns)	383 ns	50 ns	20 ns	10 ns (est)	2 ns (est)
Speedup over Xeon	n/a	7.66	19.15	38.3	191.5
% usage of chip	n/a	n/a	73%	n/a	7.3% (est)

HD calculation on a modest sized FPGA is approximately 19 times faster than a state-of-the-art CPU design and 2.5 times faster than the image processing Cell processor. The Cell processor greatly outperforms the Xeon machine and scales really well across the cores, but still does not outperform a modestly sized FPGA.

In the Cyclone-II FPGA, there are over 400,000 memory bits available for on-chip storage. 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 Cyclone-II FPGA. Therefore, once per clock cycle, a 2048-bit vector is fetched from on-chip memory, and the HD calculation is performed. Again, therefore, the entire process can be executed in 20 ns. We have successfully implemented and tested the HD calculation with and without a memory device.

Also reported in Table 1 is the utilization of the FPGA resources. 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 the 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.

The Cyclone-II is not built for performance and is also not a state-of-the-art design. A projection of the performance of a faster Cyclone-II (100 MHz) and a state-of-the-art Stratix IV (500 MHz) FPGA is given in Table 1. A still modest Cyclone version clocked at 100 MHz is able to outperform the sequential version by a factor of 38. The faster Stratix IV is projected to perform approximately 190 times faster than the sequential version. Additionally, our implementation on the Stratix IV would only consume approximately 7.3% of the chip. On-chip memory for the Stratix-IV is also much larger with 22.4 Mbits of on-chip storage. For example, a database consisting of 10000 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 becoming 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. In addition, with a larger FPGA, it is possible to compute multiple matches in parallel. This available parallelism is also demonstrated in Table 1.

5. Conclusion

The trend in modern computing is toward a multicore design. In this research, we are interested in the performance of a modern multicore, Cell processor, compared to an FPGA for an image processing algorithm. We demonstrate that a vital portion of an iris recognition algorithm can be parallelized on both systems, and our results on an FPGA are 2.5 times better than the CELL processor. FPGAs have been on an impressive scaling trend over the last 10 years. We expect this scaling trend to continue in the short term and we even believe that an FPGA could potentially be a part of the General Purpose Computer of tomorrow.

References

[1] "Synergistic processing in cell's multicore architecture," http://www.research.ibm.com/people/m/mikeg/papers/2006_ieeemicro.pdf.

[2] *Cell Broadband Engine Programming*, IBM Developer Works, https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D.

[3] "Cell Broadband Engine," https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine.

[4] J. Daugman, "Probing the uniqueness and randomness of iriscodes: results from 200 billion iris pair comparisons," *Proceedings of the IEEE*, vol. 94, no. 11, pp. 1927–1935, 2006.

[5] R. W. Ives, R. P. Broussard, L. R. Kennell, R. N. Rakvic, and D. M. Etter, "Iris recognition using the ridge energy direction (RED) algorithm," in *Proceedings of the 42nd Annual Asilomar Conference on Signals, Systems and Computers*, pp. 1219–1223, Pacific Grove, Calif, USA, November 2008.

[6] C.-H. Park, J.-J. Lee, M. J. T. Smith, and K.-H. Park, "Iris-based personal authentication using a normalized directional energy feature," in *Proceedings of Audio and Video Based Biometric Person Authentication Conference*, vol. 2688, pp. 224–232, 2003.

[7] Y. Chen, S. C. Dass, and A. K. Jain, "Localized iris image quality using 2-D wavelets," in *Proceedings of the International Conference on Biometrics (ICB '06)*, pp. 373–381, Hong Kong, January 2006.

[8] S. Shao and M. Xie, "Iris recognition based on feature extraction in kernel space," in *Proceedings of the IEEE Biometrics Symposium*, Baltimore, Md, USA, September 2006.

[9] R. P. Broussard, L. R. Kennell, and R. W. Ives, "Identifying discriminatory information content within the iris," in *Biometric Technology for Human Identification V*, Proceedings of SPIE, Orlando, Fla, USA, March 2008.

[10] G. Gupta and M. Agarwal, "Iris recognition using non filter-based technique," in *Proceedings of the Biometrics Symposium*, pp. 45–47, Arlington, Va, USA, September 2005.

[11] R. W. Ives, L. Kennell, R. Broussard, and D. Soldan, "Iris recognition using directional energy," in *Proceedings of the IEEE International Conference on Image Processing (ICIP '08)*, San Diego, Calif, USA, October 2008.

[12] L. Masek, *Recognition of human iris patterns for biometric identification*, M.S. thesis, The University of Western Australia, Perth Crawley, Australia, 2003, http://www.csse.uwa.edu.au/~pk/studentprojects/libor/LiborMasekThesis.pdf.

[13] L. Kennell, R. W. Ives, and R. M. Gaunt, "Binary morphology and local statistics applied to iris segmentation for recognition," in *Proceedings of the IEEE International Conference on Image Processing (ICIP '06)*, Atlanta, Ga, USA, October 2006.

[14] J. Daugman, "Statistical richness of visual phase information: update on recognizing persons by iris patterns," *International Journal of Computer Vision*, vol. 45, no. 1, pp. 25–38, 2001.

[15] R. P. Broussard, R. N. Rakvic, and R. W. Ives, "Accelerating iris template matching using commodity video graphics adapters," in *Proceedings of the 2nd IEEE International Conference on Biometrics: Theory, Applications and Systems (BTAS '08)*, Crystal City, Va, USA, September 2008.

[16] Intel Corporation, June 2008, http://www.intel.com/products/processor/manuals/index.htm.

[17] Intel Corporation, June 2008, http://processorfinder.intel.com/details.aspx?sSpec=SL9YM.

[18] Altera Corporation, June 2008, http://www.altera.com/education/univ/materials/boards/unv-de2-board.html.

[19] Altera Corporation, June 2008, http://www.altera.com/products/devices/cyclone2/cy2-index.jsp.

[20] J. Daugman, "How iris recognition works," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 14, no. 1, pp. 21–30, 2004.