FPGA or Cell for an Image Processing Application

Ryan N. Rakvic[a], Hau Ngo[a], Randy P. Broussard[b], Robert W. Ives[a]
[a]Department of Electrical and Computer Engineering, U.S. Naval Academy, Annapolis, MD;
[b]Department of Systems Engineering, U.S. Naval Academy, Annapolis, MD;

***Abstract*— 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 multi-core heterogeneous processor known as the Cell, which is designed to perform complex image processing algorithms at a high-performance level. All the while, image processing algorithms are still coded for off-the-shelf computers, such as the state-of-the-art Xeon-based computer systems. In this research project, we study the differences in performance of a modern image processing algorithm on three hardware platforms. We show that the heterogenous Cell based PS3 is able to outperform a state-of-the-art Xeon processor by 7.7 times. However, our results on an FPGA are 2.5 times better than the PS3. Although the CELL processor greatly outperforms the Xeon processor, the FPGA is the leader in performance.**

***Keywords*— Iris recognition algorithms, cell processing, reconfigurable computing, parallel computing, hamming distance, biometrics**

I. 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, multi-core 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 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 multi-core 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 [11].

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 co¬processors 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 [12].

According to IBM, the Cell BE is capable of achieving in many cases, 10 times the performance of the latest PC processor [13]. 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.

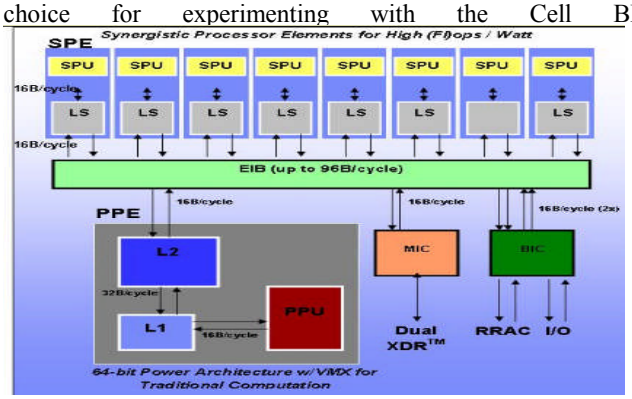

Figure 1. Cell BE High Level Architecture Diagram

Historically programmers have thought in sequential terms, and programming these multi-core 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"

978-0-7695-3915-7 2009

DOI 10.1109/ACT.2009.75

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 our image processing algorithm. In Section 3, we present an approach 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.

II. IRIS RECOGNITION BACKGROUND

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 [1]. An alternate iris recognition algorithm, referred to as the Ridge Energy Direction (RED) algorithm [2] will be the basis for this work.

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. 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 accomplishes this via the segmentation method described in prior art [3]. This establishes the central point of the iris within the image's x and y coordinates, allowing the computer to 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 x 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.

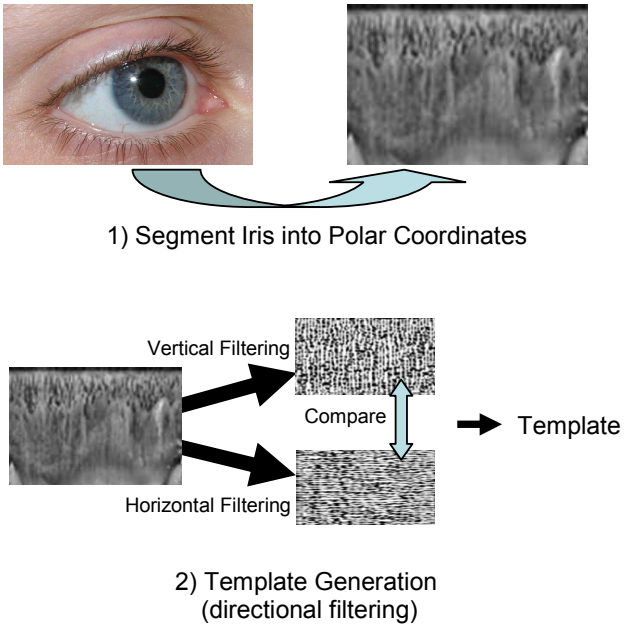

Figure 2. RED Iris Recognition Algorithm. Visible is the associated two dimensional encoding of the iris image into energy data [4].

The filter passes over this periodic array taking in 81 (9x9) values at a time (note, in [2], 11x11 is used). Finally, the template is generated by comparing the results of two different directional filters 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 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 Equation 1. This is illustrated in Figure 3.

$$(1)\quad HD = \frac{\left\|(\text{template A} \otimes \text{template B}) \cap \text{mask A} \cap \text{mask B}\right\|}{\left\|\text{mask A} \cap \text{mask B}\right\|}$$

The exclusive-or operation is used to detect disagreement between corresponding bit pairs in the two templates, 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 non-match declaration is made.

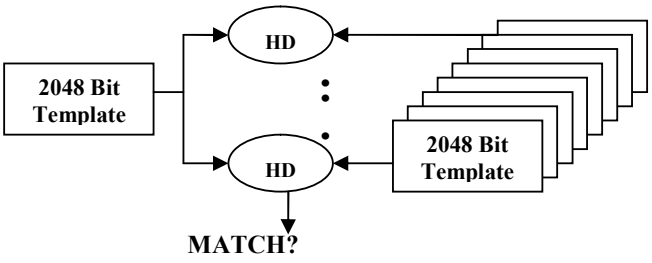

Figure 3. New template is compared with each template stored in a database.

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 3. 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.

III. IMPLEMENTATIONS

A. 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 multi-processing and multi-threading. Recently, there has been interest in exploring the parallel nature of this application [9]. 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 4 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 look-up 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.

The code is compiled for a Xeon Processor, and hence IA-32 assembly code is produced [8]. 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 x 38 assembly instructions to perform one template match.

```
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 4. C++ code for fractional Hamming Distance Computation.

B. Parallel on a 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. FPGAs are commonly programmed via VHDL (VHSIC Hardware Description Language). VHDL statements are inherently parallel, not sequential. 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, but now in parallel. There are 2,048 XOR gates and 4,096 AND gates required for this computation. This code is contained within a “process” statement. In this code, the clock signal is drawn from our FPGA board which contains a 50 Mhz clock. Therefore, every 20ns, this hamming distance calculation is computed.

C. 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.

IV. RESULTS

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 CPU experiment is executed on an Intel Xeon X5355 [5] workstation class machine. The processor is equipped with 8 cores, 2.66 GHz clock and an 8 MB L2 cache. 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 optimized C++ code time is actually faster than some of the times reported in the literature for commercial implementations [10]. 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.

The FPGA experiment is executed on a DE2 [6] 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 [7] 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. We are able to determine the size required of our program on the FPGA, and the resulting execution time.

Figure 5 illustrates the execution times and acceleration achieved for our implemented CELL based version on the PS3, FPGA version on the Cyclone-II EP2C35, and a 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 Cell processor greatly outperforms the Xeon machine by 7.7 times, scaling really well across the cores, but still does not out-perform a modestly-sized FPGA. Iris matching 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.

	Optimized Xeon Code	CELL (with 6 SPEs)	Cyclone-II EP2C35 (50 MHz)
Time per match (ns)	383 ns	50 ns	20 ns
Speedup over Xeon	n/a	7.66	19.15

Figure 5. FPGA vs. CPU comparison for iris match execution.

V. CONCLUSION

In this research, we demonstrate a crucial portion of an image processing algorithm can be parallelized on both a PS3 and an FPGA. The heterogenous CELL based PS3 is able to outperform a state-of-the-art XEON processor by 7.7 times. However, our results on an FPGA are 2.5 times better than the CELL processor on the PS3. Although the CELL processor greatly outperforms the Xeon processor, the FPGA is the leader in performance. We plan to study the different advantages of FPGAs, including energy consumption. An FPGA-based option presents an exciting parallel alternative for future parallal image-based algorithms.

REFERENCES

[1] 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.

[2] Robert W. Ives, Randy Broussard, Lauren Kennell, Ryan Rakvic and Delores Etter, "Iris Recognition Using the Ridge Energy Direction (RED) Algorithm," Proceedings of the 42nd Annual Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, Nov. 2008.

[3] L. Kennell, R.W. Ives, R.M. Gaunt, "Binary Morphology and Local Statistics Applied to Iris Segmentation for Recognition," presented at the 2006 IEEE International Conference on Image Processing, Atlanta, GA, Oct. 2006.

[4] J. Daugman, "Statistical richness of visual phase information." International Journal of Computer Vision, 45(1), pp 25-38.

[5] Intel Corporation, 16 June 2008, <http://processorfinder.intel.com/details.aspx?sSpec=SL9YM>.

[6] Altera Corporation, 16 June 2008, <http://www.altera.com/education/univ/materials/boards/unv-de2-board.html>.

[7] Altera Corporation, 16 June 2008, < http://www.altera.com/products/devices/cyclone2/cy2-index.jsp >.

[8] Intel Corporation, 25 June 2008, < http://www.intel.com/products/processor/manuals/index.htm>.

[9] R.P. Broussard, R.N. Rakvic, R.W.Ives, "Accelerating Iris Template Matching using Commodity Video Graphics Adapters," 2008 IEEE International Conference on Biometrics: Theory, Applications and Systems, Crystal City, VA, Sep. 2008.

[10] J.Daugmann. "How iris recognition works," IEEE Transactions on Circuits and Systems for Video Technology vol. 14, January 2004.

[11] "Synergistic Processing in Cell's Multicore Architecture". http://www.research.ibm.com/people/m/mikeg/papers/2006_ieeemicro.pdf. IEEE.

[12] Cell Broadband Engine Programming Handbook. IBM Developer Works. http://www¬01.ibm.com/chips/techlib/techlib.nsf/techdocs/1741C509C5F64B3300257460006FD68D.

[13] Cell Broadband Engine. http://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine.