Iris Matching Algorithm on Many-Core Platforms

Chen Liu, Benjamin Petroski, Guthrie Cordone, Gildo Torres, Stephanie Schuckers
Department of Electrical and Computer Engineering
Clarkson University
Potsdam, New York 13699
Email: {cliu, petrosbg, cordonga, torresg, sschucke}@clarkson.edu

***Abstract*—Biometrics matching has been widely adopted as a secure way for identification and verification purpose. However, the computation demand associated with running this algorithm on a big data set poses great challenge on the underlying hardware platform. Even though modern processors are equipped with more cores and memory capacity, the software algorithm still requires careful design in order to utilize the hardware resource effectively. This research addresses this issue by investigating the biometric application on many-core platforms. Biometrics algorithm, specifically Daugman's iris matching algorithm, is used to benchmark and compare the performance of several many-core platforms. The results show the ability of the iris matching application to efficiently scale and fully exploit the capabilities offered by many-core platforms and provide insights in how to migrate the biometrics computation onto high-performance many-core architectures.**

Keywords*—*Iris matching, Daugman's algorithm, GPU, Xeon Phi, Single-Chip Cloud Computer, Many-core

I. Introduction

The utilization of biometrics in everyday life continues to increase. The latest example is Touch ID has become an integrated feature for fingerprint recognition in iPhone 5s, which also enables Apply Pay in iPhone 6. Associated with this, the amount of biometric data collected and examined has dramatically expanded. One major challenge is the operational capabilities of managing and analyzing large-scale biometric information in an effective and efficient manner [1]. In 2012, Sussman et al. suggested the foundation for the next-generation biometric systems being "expandable, scalable and flexible to accommodate new technologies and biometric standards, as well as interoperable with existing systems" [2]. Therefore, it is of great interest and importance to develop more efficient and effective large-scale biometric identity management systems.

The biometric application we will utilize in this study is the iris matching algorithm, one stage of Daugman's iris recognition algorithm [3]. This algorithm compares two samples represented by two 2D bit matrices by performing Hamming Distance calculation. The result in the form of matching score shows the similarity of the compared samples. Having real-time access and processing requirements, the iris recognition algorithm can be classified as a computation-intensive application. If purely implemented into software and executed on a general-purpose processor, the performance will be limited. In addition, the program structure of iris matching algorithms shows its code can be highly parallelized. However, the number of cores on a general-purpose processor is normally small and can achieve only a limited degree of parallel execution.

With the consistent advancement in VLSI technology, researchers can implement software algorithms into hardware which has strong computation capability and can process large amount of data in parallel. This approach can meet the real-time performance requirement of iris matching algorithm. We believe the future solution architectures should consider emerging many-core processors based on big data processing techniques. While operational systems have clearly demonstrated the capacity for large data processing, few related approaches have been proposed. Study of the tradeoffs in system design can be useful for future developers of large scale systems, particularly, when considering emerging technologies. The contribution of this work is to serve as a benchmark study on how iris recognition algorithm adapts to many-core platforms and how to explore the parallelism inside the algorithm on different platforms, as we anticipate that many-core platforms will be employed by the next-generation biometric systems.

The rest of the paper is organized as follows: in Section II we introduce the iris matching algorithm itself. In Section III we briefly go over some related work. Then in Section IV we discuss the iris matching on the Single-Chip Cloud Computer platform. Section V we presents the study of iris matching algorithm on Intel Xeon Phi coprocessor platform. Section VI we discuss the iris matching on the graphic processing unit platform. In Section VII we present the performance comparison across all the platforms. Finally in Section VIII conclusions and future work are described.

II. Iris Matching Algorithm

Daugman's iris matching algorithm [3] consists of a four-stage process that begins after an iris image has been captured. This process segments, normalizes, and then performs feature encoding to prepare the template for matching. The reason we choose Daugman's iris recognition algorithm is because the widely acceptance of Daugman's algorithm in the research community for iris recognition. And the availability of the implementation in Matlab from our previous research also aided this choice since we need to translate the previous implementation from Matlab to C to MPI/OpenMP/CUDA to conduct this research.

This work only focuses on the last stage of the iris recognition algorithm, which is the matching stage. If the application only needs to compare one iris template, then the matching stage may not be the most computationally intensive. However, when the application needs to cross-comparing hundreds of iris templates as we did in this study, the matching stage apparently becoming the most computationally intensive stage.

The iris matching algorithm compares two samples based on the matrix Hamming Distance calculation. The calculated

matching score represents the number of different bits between the samples. The lower the score is, the more similar the samples are. In this project we refer to the normalized score, the value of which varies between 0 and 1.

The size of the iris template is that each iris template consists of two matrices (template and mask) of the same size, 20 rows by 480 columns, and each entry is a single bit, so the size of matrix is 9600 bits.

The first matrix is the template representing the iris pattern. The second matrix is the mask matrix representing aspects that should not be considered in the matching, such as eyelid, eyelash, and noise (e.g., spectacular reflections) which may mask parts of the iris. The template matrix is computed by the following steps. First, iris region is segmented from a near-IR image of an eye and transformed from a circular image into a rectangular image. Next, Gabor filtering is used to extract the most discriminating information from iris pattern as a matrix.

The main body of the algorithm is a 17-round for-loop structure with the index shift from -8 to 8. Suppose the matrices of the first sample are named as template1 and mask1, and the matrices of the second sample are named as template2 and mask2, respectively. In each round, first, template1 and mask1 are left-rotated if shift is less than 0 or right-rotated if shift is larger than 0 for $2 \times abs(shift)$ columns to form two intermediate matrices, named template1s and mask1s. Next, we XOR template1s with template2 into temp and OR mask1s with mask2 into mask. Finally, the matrix temp will be ANDed with the matrix mask to form the matrix result. So the Hamming Distance for this round is calculated by dividing the number of 1s of the matrix result by the total number of the 2D matrix entries (which is 20×480) minus the number of 1s in matrix temp. The minimum-valued Hamming Distance of all 17 iterations is the calculated Hamming Distance, which is returned as the matching score of the two samples, showing their similarity.

III. Related Work

There are previous works that implemented the iris recognition algorithm on a single hardware platform, such as GPU, FPGA, DSP, and many-core processor.

There are many works for Field Programmable Gate Array (FPGA) prototyping of iris recognition. The straight forward way is to implement the time-consuming parts of the iris recognition algorithm into hardware for accelerating its execution. Rakvic et al. [4] did code parallelization for the iris segmentation, template creation, and template matching and implemented it on FPGA platform. They compared the performance of FPGA with that of the state-of-the-art CPU and showed speedups of 9.6, 324, and 19 folds, respectively, in the three iris matching parts. This was achieved with moderate hardware usage.

Profiling work helps to identify the time-consuming part of the iris recognition. Raida et al. [5] did computation workload profiling and identified Gabor filter, Gaussian masque, Canny, and Hough transform as the most time-consuming components of the iris recognition algorithm. These parts were then implemented directly into hardware IP.

Neural network mechanisms can be adopted in FPGA implementation to enhance the iris recognition accuracy. Mohd-Yasin et al. [6] employed neural network concepts to implement iris recognition on Altera FPGA board to improve the efficiency and accuracy of iris recognition.

Miyazawa et al. [7] applied the phase-based image matching based on technique using phase components in 2D DFT to two parts of the iris recognition algorithm: image alignment and matching score calculation. Then, they implemented the modified iris recognition algorithm into DSP as the prototype, while our work focus on many-core architecture. Their proposed implementation has low error rate and meets real-time requirement with low hardware cost.

Prior research has been performed on Graphical Processing Unit (GPU) but with different iris template properties, methods for template-mask data transfer from host to device, kernel launch characteristics, and bit-shifting while calculating hamming distances [8], [9].

Chang et al. [10] performed detailed profiling and binary instrumentation on the iris matching algorithm. But they focused on profiling the iris recognition algorithm and getting the generic workload characteristics for processor-in-memory (PIM) architecture, not focusing on many-core implementation.

Torres et al. [11] performed detailed analysis of iris matching algorithm, but only on a single hardware platform, Single-chip Cloud Computer (SCC), while our work implemented on multiple platforms and performed crossed-platform comparison, which complements the previous work on this subject.

Different from previous work, in our research, we employ three innovative many-core architecture platforms (GPU, SCC and Xeon Phi) with abundant hardware resources for parallel computation. The iris samples to be compared are distributed to many processing elements and processed in a parallel fashion. In addition, we directly observe and compare the performance differences across these many-core platforms, which provide more thorough insights than the previous works. To the best of our knowledge, our work is also the first to implement the iris matching algorithm on Xeon Phi coprocessor.

IV. Iris Matching on Single-chip Cloud Computer

In the first experiment, we use Intel's Single-chip Cloud Computer (SCC) as the many-core platform for executing the iris matching algorithm. The Single-chip Cloud Computer experimental processor [12] is a 48-core concept vehicle created by Intel Labs as a platform for many-core software research. It consists of 48 Pentium class IA-32 cores put on a 6×4 2D-mesh network of tiles, with each tile housing 2 cores. Containing four memory controllers, the SCC divides the 6×4 2D-mesh of tiles into four memory domains, where each controller serves to the cores inside its own domain.

SCC integrates advanced power management technologies. Containing a configurable voltage regulator controller (VRC), the programmer has the ability of independently changing the voltage across the entire chip [12]. In addition to the physical platform, SCC also comes with the RCCE library. It is a many-core communication environment, very similar to Message Passing Interface (MPI), specially developed for the SCC. Please note SCC is the predecessor of Intel's Many

Integrated Core (MIC) architecture, which we will employ in next section.

A. Experimental methodology

Although iris recognition is typically a computation performed in real-time, a data set of iris templates, consisting of left and right irises, was prepared for this experiment to reveal speedup on the computationally intensive matching step. All the iris data we use throughout this study are from the Q-FIRE database [13]. The results presented in this section correspond to executing the iris matching application over an iris data set consisted of 573 templates and 573 masks, giving 573 unique iris samples. Each sample is compared against the entire dataset following an N vs N approach, resulting in a total of $163,878$ comparisons. To compile the code, we use icc version 8.1 and RCCE version $1.0.13.x$, with no optimization flag.

Knowing that the iris matching application is an embarrassingly parallel algorithm, where each comparison can be completed independently and in parallel, we ran two experiments with different configurations (Frequency/Voltage settings) on the SCC. For both of these experiments we completed the comparison of all the iris samples using a varying number of cores from 1 to 48, in order to show how such application would benefit from an increasing usage of hardware resources (cores).

For the first experiment we configured the cores on the SCC with voltage and frequency of [533MHz - 0.8V] in what we called the "Low-Gear" setting. In the second experiment, which we refer as "High-Gear" setting, we configured the SCC cores with voltage and frequency of [800MHz - 1.2V].

In order to avoid a bottleneck when several cores try accessing memory for loading their data files through the same memory controller at the same time, we evenly distributed the cores involved on each run from 1 to 48 cores amongst the four memory controllers. Experimental results showed no performance degradation for different core allocation policies, meaning that our specific application does not saturate the bandwidth offered by the SCC memory controllers.

B. Experimental results

Figure 1 presents the performance results obtained from running both the High-Gear and Low-Gear experiments on the SCC platform. It shows how effectively the iris matching algorithm scales on the SCC, resulting in an almost linear speedup as the number of cores increases from 1 to 48. Both cases appear to have very similar behaviors when varying the number of cores, having some fluctuation when reaching the maximum number of SCC cores. The setting running at higher frequency of 800MHz shows a better performance for all cases. The fastest computation time was 290 seconds with 46 cores.

V. Iris Matching on Xeon Phi Coprocessor

In this experiment, we use Intel Xeon Phi coprocessor as the many-core platform to execute the iris matching algorithm. The Intel Xeon Phi coprocessor is Intel's latest implementation of its Many Integrated Core (MIC) architecture. The Xeon Phi coprocessor we used in this study is Model 5110P running

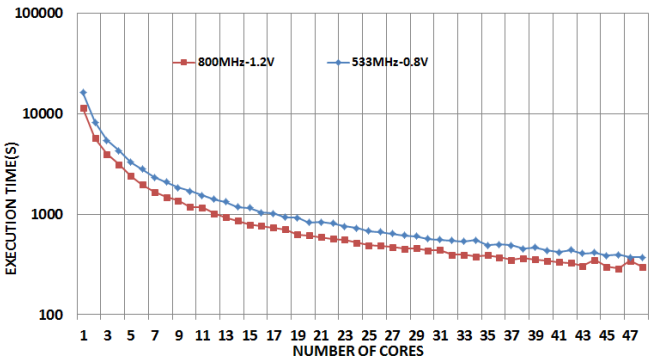

Fig. 1. Performance on SCC

at 1.053GHz. This model contains 60 cores that each has four hardware threads, giving a total of 240 threads across a single device [14]. We use the high level of hardware parallelism within the Xeon Phi coprocessor to exploit the inherent parallelism presented in the iris matching algorithm. Here, we port the iris matching algorithm onto the Xeon Phi coprocessor and explore the speedup of the computation as we increase the number of parallel threads.

A. Programming model

We use the offload programming model to port the iris matching algorithm onto the Xeon Phi coprocessor. With the offload programming model, code is executed from a traditional 'host' processor which 'offloads' work to the coprocessor if specified. The host processor we use for our offloading model is an Intel Xeon E5-2670 processor running at 2.60 GHz. The region of the code for the Xeon to offload to the Xeon Phi is specified via specific compiler directives. Any code within the offload region is sent to the coprocessor to execute. The benefit of the offload model is that the Xeon processor can perform the serial computations while the Xeon Phi coprocessor can perform the parallel computations, which capitalizes on the strengths of each processor [15].

For our experiment, we read and sort the iris data with the Xeon host and offload the sorted data to the Xeon Phi. Using the sorted data, the Xeon Phi performs comparisons in parallel to determine the hamming distance between every iris sample in the data set. The individual hamming distances for each iris-to-iris comparison are stored in a vector on Xeon Phi, which is sent back to the Xeon host once all of the comparisons are completed. The host then compares each of the individual hamming distances to determine which one is the smallest, effectively identifying the two irises within the data set that are closest to being identical.

B. Parallelization

To fully exploit the power of the Xeon Phi, we need to reach a level of parallelism within our code that can equal the parallel capabilities of the platform. To parallelize our code, we use the OpenMP API for C. OpenMP offers a simple way to create and run a large amount of parallel threads within our code. To use OpenMP, we specify a region of the code to parallelize with the compiler directive "#pragma omp

parallel", which spawns a number of threads determined by the environment variable "OMP_NUM_THREADS". Each thread spawned will execute the code presented within the parallel region simultaneously. To divide the work between threads we specify private variables for each thread using the 'private' clause and divide the work within a 'for' loop using the "#pragma omp parallel for" compiler directive. The 'parallel for' compiler directive divides the work load of a 'for' loop between all of the parallel threads [15]. To compile the code, we use icc version 14.0.2 and OpenMP version 4.0, with no optimization flag.

For our experiment, we initialize the parallel region within our code immediately after the offload region, causing the parallel threads to spawn once the code is offloaded to the Xeon Phi. Each iris-to-iris comparison is handled by a single thread, meaning that a number of comparisons are performed simultaneously on the coprocessor depending on how many threads are spawned. The scheduling of the comparisons between the threads is handled by a 'parallel for' compiler directive. Each thread performs the iris matching algorithm and finds a hamming distance value for each unique comparison. The parallel region ends once every possible comparison between the irises in the data set is performed. The lowest hamming distance value found among all the comparisons within each thread is stored in a public vector at a location unique to the thread number. This vector is then sent back to the Xeon host for a final comparison and determination of the two iris templates with the lowest hamming distance between them.

C. Experiment results

The workload is the same as in Section IV, with 573 iris samples and a total of 163,878 comparisons. For each result, we ran the code with a specific thread number three times and recorded the average time taken to perform the computation. We varied the number of threads of the computation from one thread, using a base 2 increase (1, 2, 4, 8, etc.), until we reached 1024 threads. We also recorded data with thread numbers that were of interest due to the specific hardware features of the Xeon Phi coprocessor: such as 60, 120, 180, and 240 threads. These numbers were of interest because they are multiples of the number of logical cores on the platform, with the maximum simultaneous thread capability of the Xeon Phi being 240 hardware threads. We expected the execution time to decrease as the number of threads is increased up to 240, because until that point all execution threads can be mapped 1-to-1 to the coprocessor hardware threads. Beyond 240 threads we expected the execution time to increase with the number of threads due to the bottleneck of 240 simultaneous threads on the Xeon Phi, where eventually more than one execution thread would be mapped to some or all of the existing hardware threads.

The results of our experiment are shown in Figure 2, where we see that the platform behaved as hypothesized: execution time decreased as the threads increased up to 240 threads, and increased beyond 240 threads. The fastest computation time was 252 seconds with 240 threads (maximum parallel capability of the Xeon Phi). The slowest computation time was 3843 seconds with one thread (serially). These results show that there was a 15X speedup of the execution time from the computation with one thread to the computation with 240 threads.

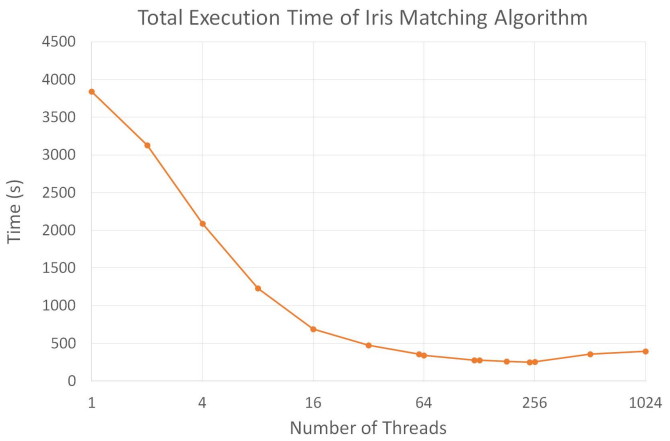

Fig. 2. Performance on Xeon Phi

VI. Iris Matching on Graphic Processing Unit

The use of Graphic Processing Units (GPUs) to perform general purpose processing has become much more prevalent in the recent past [16]. GPUs are designed to perform computations in parallel using many compute cores. Each core contains a floating point unit and an integer unit for performing generic computations. The GPU we used in this study is an Nvida GeForce GTS 450 (192 cores with 783 MHz Graphics Clock and 1.566 GHz Processor Clock). The CPU used in association with the GPU is Intel Core i7-2600 running 2.8GHz.

A. Experimental methodology

Through using Nvidia's CUDA framework, algorithms may be parallelized by writing a dedicated kernel to be launched on the GPU. CUDA is useful for quickly accelerating computations, especially for computations that are inherently parallel. Many computations involved with biometrics share this characteristic due to the need to perform comparisons between data sets. The comparison of iris samples using Daugman's iris matching algorithm to calculate a hamming distance is one such example and will be used to understand and compare the performance speedup when using CUDA. To compile the code, we use gcc version 4.6.3 and nvcc release 4.0, V0.2.1221, with no optimization flag.

In order to launch a kernel that is able to reference specific templates and masks, the template files and the mask files were each converted to a 1-D array and copied into global memory on the GPU. The characteristics of the kernel, specifically the number of thread blocks and the number of threads were then used to locate a specific template or mask within the 1-D global memory array to be used in each comparison. Once the templates and masks had been stored in local memory, the kernel performed Daugman's iris matching algorithm. However, due to the limitation on our GPU implementation, mainly due to the size of the scratchpad memory of the GPU, we were not able to load all 573 iris samples to the GPU

to perform the comparison. The results we presented in this section are from up to 128 iris samples with 8128 comparisons.

To identify the fastest speed up and to understand the process by which kernels are launched on the GPU when performing Daugman's iris matching algorithm, the characteristics of the kernels were varied across several thread sizes and template counts. The execution time was the primary basis of comparison to verify an improved performance as the techniques for customizing GPU computations beyond kernel characteristics are limited. Despite this, varying the kernel characteristics and measuring the execution time revealed a significant difference between CPU and GPU runtime, as well as between varying CUDA kernel thread-level granularities.

B. Experimental results

After running several kernels with varying threads on multiple data sets, the execution time and hamming distance results were compared to a serialized CPU version of Daugman's iris matching algorithm. The hamming distance was used to confirm that the GPU computation was returning the same result as the CPU. Once this was confirmed, the execution time for each computation was compared, as shown in Figure 3.

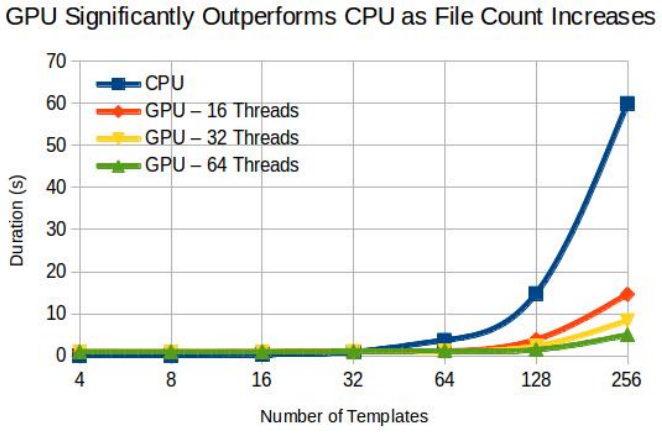

Fig. 3. Performance on GPU

Several conclusions were made from this comparison. The first was that there is clearly an overhead when using GPUs for computation relative to CPUs, which is seen when performing the computation on small biometric data sets. The execution time of the iris matching is slightly faster when performed serially on the CPU-only side for biometric data sets of 32 or fewer template files (16 or fewer samples as each sample contains one template file and one mask file). But for larger numbers of templates, namely 64 or more (32 samples or more), the GPU begins to significantly outperform the CPU. Additionally, there appeared to be a decrease in runtime as the number of threads in the kernel on the GPU side doubled. Despite this trend, the largest speedup occurred with a kernel thread size of 64. At 256 template files (128 samples), we achieved a speedup of 15X compared with that of the CPU only case. The execution time of kernels with 128 and 256 threads was very close to the results from 64 threads hence the results are omitted here. The execution results reveal that the exponential growth of the iris matching runtime is much more significant on CPUs than on GPUs. Furthermore, by varying the thread-level granularity of kernels launched on the GPU, additional speedup may be achieved.

VII. Performance Comparison across Platforms

In this section we present a head-to-head comparison of the performance obtained from running the same workload on all four different platforms (CPU, SCC, Xeon Phi, and GPU). With each platform running the same workload, this experiment consists of comparing 128 samples, where each sample contains a template and a mask. It follows an all-to-all comparison pattern, completing a total of 8128 comparisons. Please note this workload is different from the workload in the SCC and Xeon Phi sections, which consists of 573 iris templates with a total of 163,878 comparisons. This mainly due to the limitation of our GPU implementation, as we mentioned in the previous section.

Each platform was configured as follows:

For the CPU-only case, we use Intel Core i7-2600. Even though it has 4 cores and supports up to 8 threads, we only ran the single-thread case on CPU.

For the case of the SCC, it was configured using 46 cores running at 800 MHz and 1.2 Volts. Basically it is the fastest configuration for the iris matching algorithm on SCC, following the results from Section IV. SCC follows a message passing programming model, with each core comminutes with another core through messages. It is different from Xeon Phi and GPU, which are both thread-based.

Following the results from Section V, the Xeon Phi platform was configured to use 240 threads (one-to-one software to hardware thread mapping) for the iris matching algorithm, when it offered the best performance.

Following the results from Section VI, the NVIDIA's GPU-based architecture is configured using 64 threads. when the best results were obtained. Basically the total 8128 comparisons are divided into 127 thread blocks, with each thread block configured to 64 threads.

Clearly GPU achieves the best performance overall, as shown in Figure 4. The performance results show the GPU architecture being 12.2 times faster than CPU, 3.2 times faster than SCC, and 2.8 times faster than the Xeon Phi platform, respectively; while the performance of Xeon Phi is 13% faster than that of SCC.

All the execution time we presented in this study includes the entire execution time, from loading the files to identifying the matching. When reporting the performance of a parallel system, one can exclude the time to reading/writing files, putting it away as I/O time while only report the pure execution time; or one can report the entire time as a whole, e.g., including the file I/O. We took the latter approach. And we did not employ any special memory optimization techniques such as cache striping, etc.

VIII. Conclusion

Iris matching has been widely adopted as an effective biometrics for secure identification and verification. This research

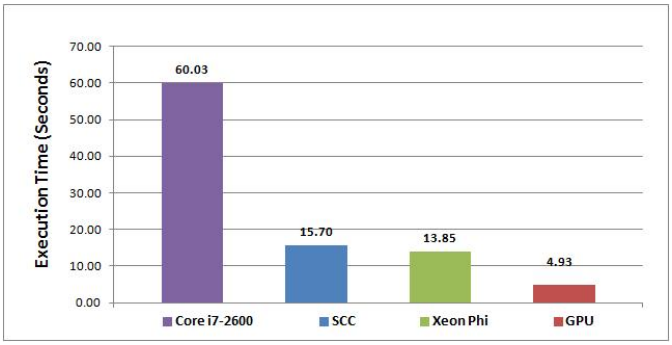

Fig. 4. Cross-platform performance results

focused on studying the performance aspect of iris matching on several different many-core platforms. We modified and ported this application onto Intel's Single-chip Cloud Computer, Intel's Xeon Phi coprocessor, and Nvidia's Graphic Processing Unit. The high core count of the hardware platforms and the internal parallelism existed in the software application allowed us to achieve solid performance enhancements across the platforms. The results as a whole showed the ability of the iris matching application to efficiently scale and fully exploit the capabilities offered by future many-core platforms.

As for future work, we will apply our methodology to other stages of the iris recognition algorithm. In addition, we are also planning to incorporate other biometric applications, such as fingerprint and facial recognition to further take advantage of the computing capability that many-core platform can provide.

ACKNOWLEDGMENT

This work is supported by the National Science Foundation under Grant Numbers IIP-1332046, IIP-1068055, ECCS-1301953. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] "The national biometrics challenge," September 2011, National Science and Technology Council, Subcommittee on Biometrics and Identity Management.

[2] A. Sussman, "Biometrics and cloud computing," in *The Biometrics Consortium Conference*, September 2012.

[3] J. Daugman, "High confidence visual recognition of persons by a test of statistical independence," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 15, no. 11, pp. 1148–1161, Nov 1993.

[4] R. Rakvic, B. Ulis, R. Broussard, R. Ives, and N. Steiner, "Parallelizing iris recognition," *Information Forensics and Security, IEEE Transactions on*, vol. 4, no. 4, pp. 812–823, Dec 2009.

[5] H. Raida and M. A. YassineAoudni, "Hw\ sw implementation of iris recognition algorithm in the fpga," *International Journal of Engineering Science*, vol. 4, 2012.

[6] F. Mohd-Yasin, A. Tan, and M. Reaz, "The fpga prototyping of iris recognition for biometric identification employing neural network," in *Microelectronics, 2004. ICM 2004 Proceedings. The 16th International Conference on*, Dec 2004, pp. 458–461.

[7] K. Miyazawa, K. Ito, T. Aoki, K. Kobayashi, and H. Nakajima, "A phase-based iris recognition algorithm," in *Proceedings of the 2006 International Conference on Advances in Biometrics*, ser. ICB'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 356–365. [Online]. Available: http://dx.doi.org/10.1007/11608288_48

[8] F. Sakr, M. Taher, and A. Wahba, "High performance iris recognition system on gpu," in *Computer Engineering Systems (ICCES), 2011 International Conference on*, Nov 2011, pp. 237–242.

[9] N. Vandal and M. Savvides, "Cuda accelerated iris template matching on graphics processing units (gpus)," in *Biometrics: Theory Applications and Systems (BTAS), 2010 Fourth IEEE International Conference on*, Sept 2010, pp. 1–7.

[10] J.-T. Chang, F. Hua, G. Torres, C. Liu, and S. Schuckers, "Workload characteristics for iris matching algorithm: A case study," in *Technologies for Homeland Security (HST), 2013 IEEE International Conference on*, Nov 2013, pp. 633–638.

[11] G. Torres, J.-T. Chang, F. Hua, C. Liu, and S. Schuckers, "A power-aware study of iris matching algorithms on intel's scc," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct 2013, pp. 1028–1037.

[12] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: the programmer's view," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov 2010, pp. 1–11.

[13] "Biometric dataset collections." [Online]. Available: http://www.citer.wvu.edu/biometric_dataset_collections

[14] J. Reinders, "An overview of programming for intel xeon processors and intel xeon phi coprocessors," 2012, intel Developer Zone.

[15] J. Jeffers and J. Reinders, *Intel Xeon Phi High-Performance Programming*. Morgan Kaufmann, 2013.

[16] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.