Implementing Image Processing Algorithms for the Epiphany Many-Core Coprocessor with Threaded MPI

James A. Ross
U.S. Army Research Laboratory
Aberdeen Proving Ground, MD, USA
james.a.ross176.civ@mail.mil

David A. Richie
Brown Deer Technology
Forest Hill, MD, USA
drichie@browndeertechnology.com

Song J. Park
U.S. Army Research Laboratory
Aberdeen Proving Ground, MD, USA
song.j.park.civ@mail.mil

***Abstract*—The Adapteva Epiphany MIMD architecture is a scalable, two-dimensional (2D) array of RISC cores with minimal uncore functionality connected with a fast 2D mesh network on a chip (NoC). We apply a threaded MPI programming model for image processing kernels including a 2D Fast Fourier Transform (FFT) with high-pass filter for edge detection; local operators for Gaussian image smoothing; and a Sobel filter, Canny edge detection, and Harris corner detection operations. Conventional MPI parallelization is employed in the implementations, demonstrating the applicability of this parallel programming model for the Epiphany architecture. Benchmark performance is analyzed for understanding the relative performance of computation and communication. The impact of the results on performance projections is discussed for RISC arrays on the current Epiphany roadmap scaled to thousands of cores.**

Keywords—image processing; many-core coprocessor; network on a chip; threaded MPI; Adapteva Epiphany

I. INTRODUCTION

The Adapteva Epiphany MIMD architecture is a scalable 2D array of RISC cores with minimal uncore functionality connected with a fast 2D mesh network on a chip (NoC). Each mesh node contains a RISC CPU core, 32 KB of shared local memory (used for both instructions and data), a mesh network interface, and a dual-channel DMA engine [1] (shown in Fig. 1). The 16-core Epiphany III coprocessor has been integrated into the Parallella minicomputer platform where the RISC array is supported by a dual-core ARM CPU and asymmetric shared-memory access to off-chip global memory. Peak single-precision performance for the Epiphany III is 19.2 GFLOPS with an extremely low power consumption of approximately 0.6 watts. The raw performance of the Epiphany III is relatively low compared to modern high-performance CPUs and GPUs; however, the Epiphany architecture provides greater energy efficiency and is designed to be highly scalable. The published road map specifies a scale-out of the existing architecture to exceed 1,000 cores in the near future [2]. Within this context it is a competitive processor technology comparable to other emerging architectures. Given the low power consumption, it is also of interest to embedded system designers for signal processing and image processing tasks.

Processors based on this architecture exhibit good energy efficiency and scalability via the 2D mesh network, but require a suitable programming model to fully exploit the architecture. Key to performance with the Epiphany architecture is data re-use, requiring precise control of inter-core communication since the architecture does not provide a hardware cache at any level. The lack of a cache is one of the defining features of this processor (compared to other many-core processors) and contributes to the energy efficiency. Access to the shared local memory is very fast, permitting 32-bit or 64-bit loads or stores every clock cycle. Each core can access memory-mapped neighboring core memory quickly across the network, bypassing the network DMA engine, which is more efficient for larger transfers. The cores can access the larger pool of off-chip mapped memory with a significant performance penalty in both latency and bandwidth relative to accessing neighboring core memory.

In the previous work in [3] and [4], we have demonstrated an efficient parallel programming model for the Epiphany architecture based on the Message Passing Interface (MPI) standard. Using MPI exploits the similarities between the Epiphany architecture and a conventional parallel distributed cluster. Our approach enables MPI code to execute on the RISC array processor with little modification and achieve high performance. For the Epiphany architecture, the MPI programming model is a better choice than APIs designed for SMP processors, such as OpenMP and OpenCL, since the latter APIs lack good semantics for controlling inter-core data movement, which is critical to achieving high performance for anything but trivially parallel applications on this processor.

Threaded MPI was developed to provide an extremely lightweight implementation of MPI appropriate for threads executing within the restricted context of the Epiphany RISC cores. It is not a port of another MPI implementation, but developed specifically for the Epiphany architecture. Threaded MPI is distinguished from conventional MPI implementations

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

by two critical differences: the device must be accessed as a coprocessor, and each core executes threads within a highly constrained set of resources. As a result, the cores are not capable of efficiently supporting a full process image or program in the conventional sense; therefore, the conventional MPI model of associating MPI processes to concurrently executing programs is not possible. Instead, coprocessor offload semantics must be used to launch concurrent threads that will then employ conventional MPI semantics for inter-thread communication. Threaded MPI has exhibited the highest performance reported to date for non-trivially parallel algorithms using a standard programming model for the Epiphany architecture.

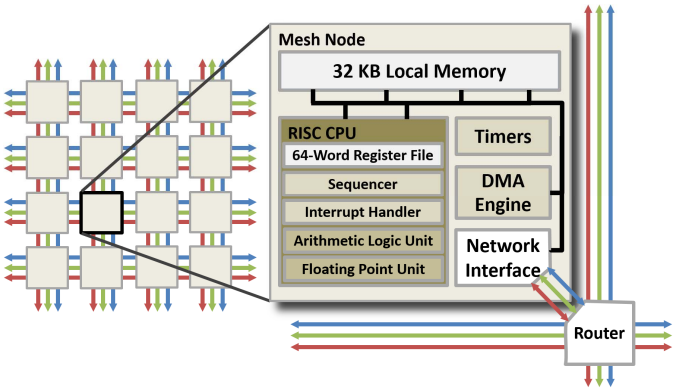

Fig. 1. Adapteva Epiphany III architecture diagram

We apply the threaded MPI programming model for image processing kernels including a 2D Fast Fourier Transform (FFT) with high-pass filter for edge detection. We demonstrate performance benchmarks of local operators for Gaussian blur, used for image noise reduction, and a Sobel filter, used for edge detection. We evaluate Canny edge detection and Harris corner detection operations with MPI communication between neighboring cores. Conventional MPI parallelization is employed in the implementations, demonstrating the applicability of this parallel programming model for the Epiphany architecture. Benchmark performance is analyzed for understanding the relative performance of computation and communication. The impact of the results on performance projections is discussed for RISC arrays on the current Epiphany roadmap scaled to thousands of cores.

II. Image Processing Algorithms on Epiphany

The 2D network topology of the Epiphany architecture suggests using the processor for tasks in which the data may be decomposed in two dimensions, although this is not a strict limitation. With the low-power floating point units and embedded nature of the Parallella platform, it makes sense to evaluate the platform for image processing tasks. Many image processing kernels have a natural domain decomposition where inter-process communication occurs with neighboring processes within the 2D computational domain. The threaded MPI programming model, developed for the Epiphany architecture, was used for coordinating the inter-core communication for the computational kernels in this work. With the exception of the 2D FFT with a high-pass filter application, the other algorithms exhibit relatively low $O(n)$ computational complexity. Increasing the number of pixels, n, or using more compact data types such as 8-bit unsigned integers (a single color channel or grayscale image) in place of 32-bit floating point data improves the arithmetic intensity (AI) of the application, defined as the ratio of floating point operations to the data movement in bytes. Using 8-bit unsigned integer data enables solving larger problems with little loss in precision.

There are performance, capability, and precision trade-offs when writing and optimizing image processing algorithms for the Epiphany architecture. As a consequence of the relatively low off-chip bandwidth, multiple image processing routines should be performed on the core after bringing data into the core-local memory. This increases the AI of the application but consequently increases the program size, reducing the available memory for image data. In most image processing applications, input data are read-only and separate storage is used for write-only output data. Due to the extremely limited available memory for image data, it makes sense to write the processed result image over the input data, effectively cutting storage requirements in half. However, there is some additional program overhead to handle the complexity; typically, a small partial buffer is required to temporarily store the input data as the image processing operation streams over the image. Moreover, reducing the input data's footprint by compressing the data type from 32-bit floating point data to 8-bit unsigned integer data enables effectively four times the pixel data, with some additional program overhead for converting between integer and floating point types while reading input and writing output. Here, additional math approximations are made when appropriate. This may not always be an option where a high degree of precision is required.

A. Threaded MPI Programming Model

The threaded MPI programming model differs from conventional MPI in two ways that reflect the platform integration of the Epiphany device as a coprocessor and the resource constraints found with the low-power RISC cores.

First, whereas conventional MPI initiates parallel execution from the command line by starting multiple processes, threaded MPI initiates parallel threads on a coprocessor from a single host program executing on the platform CPU. The *mpiexec* command is effectively replaced with an analogous function call: *coprthr_mpiexec()*. This programming model is necessary since the coprocessor cores have limited resources, preventing the practical execution of full process images, and must be used for parallel co-processing with lightweight threads. This programming model has the advantage of localizing the parallelism to a fork-join model within a larger application that may otherwise execute on the platform CPU, and multiple *coprthr_mpiexec()* calls may be made from within the same application. Direct coprocessor device control and distributed memory management tasks are handled from the platform CPU's host program using COPRTHR API routines and are separate from the MPI layer (which is used only for inter-thread message passing). Compared with conventional MPI code written for a distributed cluster, the only practical distinction with MPI code written for Epiphany is that the

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

main() routine of the MPI code must be transformed into a thread function and employ Pthread semantics for passing in arguments. Beyond this, no change in MPI syntax or semantics is required.

Second, threaded MPI and conventional MPI differ in the design of the communication layer itself. Whereas conventional MPI typically employs large internal buffers measured in megabytes tuned for optimal network and application performance, the inter-thread communication layer for Epiphany must utilize a portion of the 32 KB of local memory per core. This memory must be used for program instructions and local data structures in the application, leaving a very limited amount of memory for internal buffers supporting MPI calls. As a result, the MPI design must be extremely efficient in terms of memory use while at the same time exhibit a low-latency, high-performance communication mechanism.

The *MPI_Sendrecv_replace()* call, which provides a mechanism to exchange data between threads, was relied upon in all applications presented here. It exemplifies the overall approach to the design of the communication layer and protocol. This call cannot be implemented with a zero-copy design; rather, it requires a buffered transfer. A relatively small MPI internal buffer is used and large messages are transparently segmented into many small DMA transactions. The performance of the design has been measured and fit to a modified alpha–beta model to account for the transparent segmentation of larger messages. Results show a fixed latency per message of 1,216 ns with an additional cost of 309 ns per internal DMA transaction and a bandwidth of 1,250 MB/s, which is consistent with the expected bandwidth of the Epiphany NoC.

A subset of the MPI standard, shown in Table I, was available at the time of this work and used for the implementation of the various image processing algorithms. The threaded MPI communication layer is implemented within and supported by the COPRTHR SDK [5], which provides programming support for the Epiphany architecture. Threaded MPI exhibits the highest performance reported using a standard parallel API for the Epiphany architecture.

TABLE I. THREADED MPI CALLS USED IN IMAGE PROCESSING KERNELS

MPI Library Calls	
MPI_Init	MPI_Comm_free
MPI_Finalize	MPI_Cart_coords
MPI_Comm_size	MPI_Cart_shift
MPI_Comm_rank	MPI_Sendrecv_replace
MPI_Cart_create	

B. 2D FFT with High-Pass Filter

As an example of a frequency domain filter, we implement a conventional high-pass filter for edge enhancement. A 2D FFT developed in previous work is combined with a Gaussian filter in the frequency domain. A single kernel is constructed to perform the forward FFT, high-pass Gaussian filter, and inverse FFT. The CPU host program is used to read in an RGB image and convert the pixels to a grayscale, single-precision, floating point representation and subsequently write out the filtered image.

The in-place complex 2D FFT is parallelized over stripes and performs a 1D, radix-2, decimation-in-time (DIT) FFT using the Cooley–Tukey algorithm. This requires performing a 1D FFT, followed by a transposition of the intermediate results; a second 1D FFT; and a final transposition to return the results to the correct data layout. The transposition of the data, conventionally known as a corner turn, requires threaded MPI for inter-thread data exchange. For efficiency, a single message is packed for exchange between thread pairs by reforming the data in each stripe, and the local transposition is combined with this step. The Gaussian high-pass filter is applied to the frequency components of the image using an exponential function to scale the frequency domain data. Initially, we used the exponential implementation from the open source PAL library, which contains various mathematical operations specifically implemented for the Epiphany. This exponential implementation was optimized with significant impact to the overall performance of the algorithm, producing a 230x speedup over the use of the original exponential function.

The performance results for the 2D FFT with high-pass filter appear in Table II. Approximately 45% of the execution time is spent during the corner turn MPI communication and about 5% of the execution time is spent on the Gaussian scaling. The balance of the time is spent calculating the forward and inverse FFT.

C. Gaussian Filter for Image Smoothing

The Gaussian filter used for this work is the standard 2D, symmetric, 3×3, local operator per pixel and normalized so that the scalar constants accumulate to unity. Operating on grayscale image data, the naïve implementation of the algorithm reads, scales, and accumulates nine pixel values with 15 floating point operations using nine instructions. However, a more optimal algorithm for the Epiphany architecture reads only three input pixels per result pixel, scales and accumulates a three-pixel column, then scales and accumulates the column values. Unrolling this by a factor of four and storing previously accumulated column values reduces the algorithm to eight floating point operations using five instructions per result pixel. Although this improved algorithm decreases the measured floating point performance, measured in MFLOPS, it increases overall throughput, measured in MPixels/s. Fig. 2 shows how the pixels are read, accumulated, and stored as columns for the next iteration within the unrolled inner loop of the algorithm. Additional code handles the remainder loop if the number of pixel columns are not a multiple of four.

The Gaussian filter was written to accept a read-only input image and write the result to a separate location in memory. This reduces the usable memory on each core by a factor of two because the resulting filtered image is about the same size as the input. With only 32KB of core-local memory, algorithms should maximize this by writing results over input data, in-place, when possible. By using a temporary buffer with a copy of the current and previous rows in the input image, the result can overwrite the input image, thus fully utilizing the usable

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

memory. Additionally, when maximizing the AI of the kernel, input data should be compressed to the smallest storage data type. For a single-channel, grayscale image, pixel data may be stored in 8-bit unsigned integers (u8) rather than 32-bit floating point (fp32). As shown in Table II, using these two techniques enables the processing of large images.

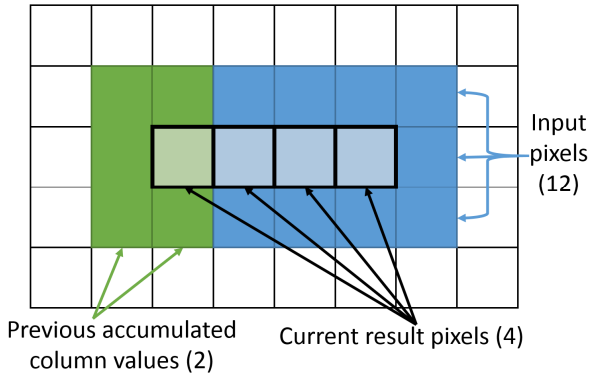

Fig. 2. Depiction of optimized 3×3 Gaussian filter for Epiphany architecture

For benchmarking purposes, only the computational portion is measured and communication (both inter-core and the reading of image data from global DRAM) is discounted. Typically, the Gaussian filter is used as part of a larger image processing application that has inter-core communication for sharing image edges between neighboring cores. Table II shows that the fp32 kernels exhibit a performance of more than a GigaPixel per second. The Gaussian in-place fp32 kernel has the extra overhead of copying a subset of the image to temporary local storage, although it does not significantly change the performance from the original kernel. This is a positive result that suggests similar in-place algorithms should be used when possible to best utilize the limited core-local memory. The Gaussian in-place u8 kernel suffered performance degradation from conversion between 32-bit floating point and 8-bit unsigned integer types—as well as the overhead of accessing unaligned data—requiring more load, store, mask, and bit-shift operations. This tradeoff must be considered during the application's development.

D. Sobel Operator for Edge Detection

The Sobel operator used for this work is the standard 3×3 convolution that approximates vertical and horizontal derivatives and then calculates the magnitude of the gradient vector. The magnitude is approximated with a fast hypotenuse calculation and two Newton iterations for improved precision, although a single iteration might be sufficient in some cases. Both the Sobel operator and the Gaussian filter use the same code optimization scheme described in Section II.C. However, the AI for the Sobel operator is three times greater. There are 24 arithmetic operations for each pixel, five of which are fused multiply–add (FMA) operations.

The performance results of the Sobel operator are presented in Table II. Like the Gaussian filter, the results show the performance for three different methods in which the output is written to a separate location in memory, the output overwrites the input data in-place, and a final method using 8-bit unsigned integer data for input. There is a necessary performance tradeoff for using 8-bit unsigned integers, but it is less significant with the Sobel algorithm due to the increased AI. The Epiphany architecture achieves excellent performance with the Sobel kernel as it did with the Gaussian operator.

E. Canny Edge Detector

The Canny edge detector accepts a noisy grayscale image, smooths the noise, estimates the edge strength of image features, estimates the orientation of the edge normal, performs non-maximum suppression for collapsing the edge to a single pixel, and performs a hysteresis threshold for pruning edges and connecting chains of edges. The hysteresis threshold following a chain of connected edges has not been used here because it introduces additional data-dependent performance, a dynamically growing list of pixels, and very challenging communication patterns. Instead, we use a double threshold for weak edge pixels connected to a strong edge pixel in the 8-connected neighborhood pixels.

Two of the computationally complex portions of the algorithm, the Gaussian filter and Sobel operator, are used here. The Sobel operator is modified to store the direction of the edge normal (0º, 45º, 90º, or 135º) efficiently in the last two bits of the floating point value, creating a composite data type. Employing this technique does not compromise precision since the bits used in the floating point representation overrepresent the real precision of the original data. The use of an 8-bit unsigned integer for storing the result is ruled out since the algorithm would potentially lead to overflow errors. The algorithm typically requires calculating the edge angle with the arctangent of the x- and y-gradient magnitudes. However, because the algorithm only requires computing one of four rough directions, a fast approximation can be made by scaling and comparing the gradient magnitudes, essentially looking at the slope formed by the ratio of the x- and y-gradient magnitudes without performing a division or trigonometric operation.

Sharing the boundaries of the image result occurs after each stencil operation with the exception of the last operation before internal image results are copied to global memory. These edge exchanges are small, involving only a few hundred bytes, and comprise approximately 6% of the total execution time. The performance data of the Canny edge detector appear in Table II.

F. Harris Corner Detector

The Harris corner detector accepts a noisy grayscale image, calculates the image gradient, and applies a Gaussian operation over a tensor structure created from the image derivatives; then, it applies Nobel's corner measure, which approximates the harmonic mean of the eigenvalues of the tensor structure. We use a modified Sobel filter to calculate the image gradients. Unlike the previous applications, the operation requires storing—at a minimum—the values of the gradients in each of the two dimensions for the Gaussian over the tensor. Thus, the algorithm requires using more than twice the memory than is required for the input image. Additionally, because the program is relatively large, the available memory for the image is already low.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

Similar storage and computational techniques described in previous sections were used. The 2D gradient data overwrite the input image. The edges are shared in a similar manner as described in the Canny edge detector, except that two values are sent for each pixel point. For the division operation found in Nobel's corner measure, a Newton–Raphson division algorithm is used with three iterations and a fast initial approximation that exploits the floating point data's format by manipulating it with a precalculated integer constant to approximate a floating point division operation. The performance data of the corner detector appear in Table II.

G. Image Processing Performance Results

Table II (below) includes a summary of the performance results for the image processing algorithms described in the previous sections. The performance numbers exclude the off-chip memory access time but include inter-core communication time for the 2D FFT filter, the Canny edge detector, and Harris corner detector.

TABLE II. IMAGE PROCESSING PERFORMANCE

Algorithm (data type)	Performance Metric			
	Pixels	*Size (KB)*	*MPixels/s*	*AI*
2D FFT filter (fp32) [a]	128×128	128	13.8	8.4
Gaussian (fp32)	224×168	147	1112	1
Gaussian in-place (fp32)	304×228	271	1076	1
Gaussian in-place (u8)	608×456	271	473	4
Sobel (fp32)	224×168	147	361.1	3
Sobel in-place (fp32)	304×228	271	360.2	3
Sobel in-place (u8)	608×456	271	207.4	12
Canny in-place (fp32) [a]	224×168	147	35.3	8.38
Harris in-place (fp32) [a]	160×120	75	141.5	7.13

[a]. Performance includes inter-core communication of image boundary data with MPI.

The arithmetic intensity gives a good measure for which algorithms are best suited for offload to the Epiphany coprocessor. It may also be used to calculate the achieved floating point performance by multiplying the performance (in pixels per second) by the AI and the total data movement in bytes (input and output images). For example, the Sobel fp32 kernel achieves 8.7 GFLOPS.

III. RELATED WORK

There have been several efforts to implement image processing algorithms on low-power devices. In one such effort [6], binary image thresholding, Gaussian blur, Sobel filter, and edge detection were benchmarked on Intel platforms, ARM platforms, and smartphones. In another [7], computer vision routines such as Canny, median blur, optical flow, color conversion, morphology, Gaussian blur, FAST detector, Sobel, pryDown, and image resize were accelerated with NEON instructions on a Tegra 3 platform. As reported in 2014 [8], an extended Canny edge detection algorithm was parallelized on an embedded many-core platform (STHORM) designed by STMicroelectronics. In the area of license plate detection [9], real-time edge detection techniques were compared on an embedded DSP. Elsewhere [10], Canny edge detection parallelization methods were investigated for the tiled multicore Tile64 processor. Real-time Harris corner detector implementation was presented [11] using a low-power, highly parallel ClearSpeed architecture. Finally, the feature detection, Canny edge detector, and Harris corner detector were designed on a flexible and parameterizable FPGA development board [12]. The work described in this paper extends the performance analysis of core image processing algorithms to an Epiphany platform.

MPI is the de facto standard for parallel computing systems. The concept of threaded MPI execution on shared memory systems was proposed several years ago [13]; however, this previous study does not consider the constraints imposed by the low-power Epiphany architecture and its accelerator configuration.

Energy efficiency is one of the primary technical challenges in the march toward exascale computing [14,15]. There are several processing technologies designed to maximize the performance-per-watt ratio. The MPPA family by Kalray comprises massively parallel architectures designed to provide high power efficiency [16]. Arria 10 FPGA by Altera features a compelling GFLOPS-per-watt as reported this year [17]. TILE-Gx processors by EZchip are tile-based many-core SoCs designed to deliver high performance per watt [18]. Similarly, Epiphany architecture is among the leaders in the power efficiency metric [19]. At this time, it is uncertain which technology will meet the requirements of future generations of HPC systems.

IV. CONCLUSION

The Epiphany architecture can perform basic 2D stencil operations like a Gaussian filter or the Sobel operator with remarkable efficiency. Methods for overwriting input data were found to perform well—where they were possible—effectively doubling the size of the input data set that can be processed. For a small performance penalty, stencil operations can operate on compressed data types like 8-bit integers instead of 32-bit floating point data. Novel solutions are needed for this architecture to carefully utilize the available core memory, such as storing an edge gradient direction in the last two bits of a floating point number. Where appropriate, fast approximations for the division operations, and the exponential and arctangent functions, enable high performance image processing codes. Even with the modifications described above it is also very challenging to fit some applications within the local memory and still have sufficient free memory for the image data. The Canny edge detection and Harris corner detection codes push at the limits of the usable space.

V. FUTURE WORK

Zero-copy and one-sided communication algorithms will be explored, which should have a positive performance impact on threaded MPI codes on the Epiphany architecture. The prospect of building larger applications for the Epiphany cores remains challenging within the 32 KB core memory constraint. The code and program size of the Canny edge detector approached

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

the limit of what was possible with the current hardware and software stack.

ACKNOWLEDGMENT

The authors wish to acknowledge the U.S. Army Research Laboratory-hosted Department of Defense Supercomputing Resource Center for its support of this work.

REFERENCES

[1] "Epiphany architecture reference," Adapteva, Rev. 14.03.11.

[2] A. Olofsson, "A 1024-core 70 GFLOP/W floating point manycore microprocessor," Adapteva, Oct. 2011.

[3] D. Richie, J. Ross, S. Park, and D. Shires, "Threaded MPI programming model for the Epiphany RISC array processor," *J. Comput. Sci.*, vol. 9, pp. 94–100, July 2015.

[4] J. A. Ross, D. A. Richie, S. J. Park, and D. R. Shires, "Parallel programming model for the Epiphany many-core coprocessor using threaded MPI," in *Proceedings of the 3rd International Workshop on Many-core Embedded Systems*, 2015, pp. 41–47.

[5] "GitHub - The CO-PRocessing THReads (COPRTHR) SDK." [Online]. Available: https://github.com/browndeer/coprthr. [Accessed: 08-Jan-2015].

[6] G. Mitra, B. Johnston, A. P. Rendell, E. McCreath, and J. Zhou, "Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2013, pp. 1107–1116.

[7] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Realtime computer vision with OpenCV," *Commun. ACM*, vol. 55, no. 6, pp. 61–69, 2012.

[8] B. Cheikh, T. Lamine, G. Nicolescu, J. Trajkovic, Y. Bouchebaba, and P. Paulin, "Fast and accurate implementation of Canny edge detector on embedded many-core platform," in *New Circuits and Systems Conference (NEWCAS), 2014 IEEE 12th International*, 2014, pp. 401–404.

[9] Z. Musoromy, F. Bensaali, S. Ramalingam, and G. Pissanidis, "Comparison of real-time DSP-based edge detection techniques for license plate detection," in *Information Assurance and Security (IAS), 2010 Sixth International Conference on*, 2010, pp. 323–328.

[10] A. Z. Brethorst, N. Desai, D. P. Enright, and R. Scrofano, "Performance evaluation of canny edge detection on a tiled multicore architecture," in *Proc. SPIE7872, Parallel Processing for Imaging Applications*, 2011, p. 78720F–78720F–8.

[11] F. Hosseini, A. Fijany, and J.-G. Fontaine, "Highly parallel implementation of Harris corner detector on CSX SIMD architecture," in *Euro-Par 2010 Parallel Processing Workshops*, 2011, pp. 137–144.

[12] P. R. Possa, S. A. Mahmoudi, N. Harb, C. Valderrama, and P. Manneback, "A multi-resolution FPGA-based architecture for real-time edge and corner detection," *IEEE Trans. Comput.*, vol. 63, no. 10, pp. 2376–2388, Oct. 2014.

[13] H. Tang, K. Shen, and T. Yang, "Program transformation and runtime support for threaded MPI execution on shared-memory machines," *ACM Trans. Program. Lang. Syst. TOPLAS*, vol. 22, no. 4, pp. 673–700, 2000.

[14] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, and others, "Exascale computing study: Technology challenges in achieving exascale systems," *Def. Adv. Res. Proj. Agency Inf. Process. Tech. Off. DARPA IPTO Tech Rep*, vol. 15, 2008.

[15] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Commun. ACM*, vol. 58, no. 7, pp. 56–68, 2015.

[16] "Flyer: MPPA MANYCORE." Kalray, Feb-2014.

[17] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," 2015.

[18] "EZchip Multicore Processors." [Online]. Available: http://www.tilera.com/products/?ezchip=585. [Accessed: 23-July-2015].

[19] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with Epiphany," *ArXiv Prepr. ArXiv14125538*, 2014.