EFFEX: An Embedded Processor for Computer Vision Based Feature Extraction

Jason Clemons, Andrew Jones, Robert Perricone, Silvio Savarese, and Todd Austin
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109
{jclemons, andrewjj, rperric, silvio, austin}@umich.edu

ABSTRACT

The deployment of computer vision algorithms in mobile applications is growing at a rapid pace. A primary component of the computer vision software pipeline is feature extraction, which identifies and encodes relevant image features. We present an embedded heterogeneous multicore design named EFFEX that incorporates novel functional units and memory architecture support, making it capable of increasing mobile vision performance while balancing power and area. We demonstrate this architecture running three common feature extraction algorithms, and show that it is capable of providing significant speedups at low cost. Our simulations show a speedup of as much as 14× for feature extraction with a decrease in energy of 40x for memory accesses.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures—*mobile processors*; I.4.7 [**Image Processing and Computer Vision**]: Feature Measurement

General Terms

Design, Performance

Keywords

EFFEX, Heterogenous Architecture, Feature extraction

1. INTRODUCTION

The field of computer vision brings together image processing, feature extraction, and machine learning to enable computer systems to understand and act upon image data. The number of computer vision applications has been growing steadily as the underlying algorithms have become more efficient and robust. Examples of computer vision applications include augmented reality, vehicle lane detection systems, and vision-based video game controllers. Recently, the explosion of mobile applications has brought computer vision applications to the mobile space as well. Applications such as Google Goggles [7] and Layar [14] are starting to make use of computer vision algorithms on mobile platforms; however, the limited computational resources of mobile processors prohibits the use of many techniques found in higher performance machines. For example, the popular vision algorithm SIFT (Scale Invariant Feature Transform) [15] can take over 53s to process a 1024x768 image on an embedded processor (ARM A8) which is 34× longer than on a modern desktop machine (Intel Core 2 Quad).

A typical vision software pipeline, illustrated in Figure 1, takes an image or video and distills the data down to key relevant information components called *features*. The features are then processed, typically with machine learning algorithms, to gain semantic and geometric information about objects in the scene, while identifying the objects' type and location. Objects can be observed over time to gain understanding of the context of the scene. Typically the process is iterative, once enough object and context understanding is gained, such information can be used to refine knowledge of the scene.

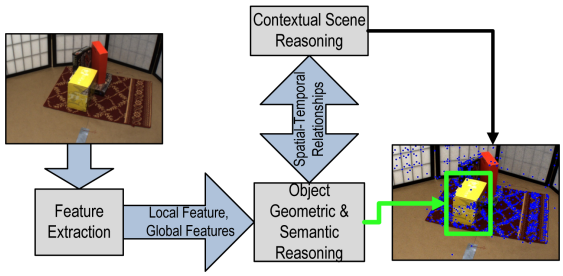

Figure 1: Overview of Computer Vision Processing This figure shows the typical flow of computer vision processing. The image on the left has features extracted (blue dots in the right image), and the features are used to understand the object (green box) and the scene in an iterative process. In the end, the yellow box is found on the carpet in the scene.

Features within an image are identified by the *feature extraction* algorithm, a principle component of the vision software pipeline. A capable feature extraction algorithm must distill important image information into scale, illumination, viewpoint, and rotation invariant signatures, called *feature descriptors*. Feature descriptors are vital to the algorithmic process of recognizing objects. For example to recognize a car, the feature extraction algorithm could enable the identification of the wheels, side-mirrors, and windshields. Given the relative location of these features in the image, a system could then recognize that the scene contains a car. The "quality" of any particular algorithm lies in its ability to consistently identify important features, regardless of changes in illumination, scale, viewpoint, or rotation. In general, the more capable an algorithm is at ignoring these changes, the more computationally expensive it becomes. To demonstrate this tradeoff, the unsophisticated FAST Corner Detector [20] executes in 13 ms for a 1024x768 image on a desktop machine, but provides no robustness to changes in illumination, scale, or rotation. In contrast, the highly capable SIFT algorithm, which is illumination, scale, viewpoint and rotation invariant, processes the same image in 1920 ms, which is 147× slower.

One method to address the performance issues of feature extraction on mobile embedded platforms is to utilize cloud computing resources to perform vision computation. However, this approach requires much more wireless bandwidth compared to a system with a capable feature detector. For example, transmitting compressed SIFT features would require about 84 kB for a large number of features (over 1000) compared to 327 kB to send a compressed image. Since existing wireless mediums are already straining to carry existing data [25], there is significant value to communication mediums to perform feature extraction locally, even if cloud resources are used to analyze the feature data.

1.1 Our Contribution

In this paper, we present EFFEX, an embedded heterogenous multicore processor specialized for fast and efficient feature extraction. The design is targeted for energy-constrained embedded

DAC 2011, June 5-10, 2011, San Diego, California, USA.

environments, in particular mobile vision applications. It is an application-specific design, but fully programmable, making it applicable to a wide variety of feature extraction algorithms. To demonstrate the effectiveness of our design, we evaluate its performance for three popular feature extraction algorithms: FAST [20], Histogram of Gradients (HoG) [6], and SIFT [15]. Specifically, this paper make three primary contributions:

- We perform a detailed analysis of the computational characteristics of three popular feature extraction algorithms: FAST, HoG, and SIFT. We identify key characteristics which motivate our application-specific hardware design.
- We develop a heterogenous multicore architecture specialized for fast efficient feature extraction. The architecture incorporates a number of novel functional units that accelerate common operations within feature extraction algorithms. We introduce memory enhancements that exploit two-dimensional spatial locality in feature extraction algorithms.
- Finally, we demonstrate the effectiveness of our design when running three different feature extraction algorithms. The EFFEX architecture is capable of high performance at low cost in terms of silicon area and power.

2. RELATED WORK

Previous works have attempted to improve computer vision processing performance with hardware optimizations. For example, work by Wu attempted to use GPGPUs to increase the speed of computer vision algorithms [26]. While GPGPUs provide large speedups, their power usage, at 100s of watts, is too high for the embedded computing space. Furthermore, GPGPUs, along with embedded GPUs, lose valuable performance gains when there is control divergence due to shared instruction fetch units. Our technique does not suffer from control divergence performance degradation. There are also many computations, such as the summation portion of the inner product, that do not map well onto a GPGPU architecture.

Silpa describes a patch memory optimization for use in texture memory in mobile GPUs to increase performance [22]. They evaluate using bulk texture transfer mechanisms, and supported these operations with texture caching. Our technique takes the design to a lower level and looks at increased hardware support.

Kodata developed an FPGA design focused on HoG [13]. Their design showed excellent speedup, but their approach was solely targeted at HoG. Our approach utilizes an application-specific processor applicable to many computer vision problems, thus, we can provide performance benefits to a wider range of algorithms. Other hardware design efforts have also worked to speed up various aspects of computer vision algorithms, such as Skribanowitz [23], Chang [5], and Qui [19], but similarly these efforts target a specific algorithm and do not offer the programmable benefits of EFFEX.

Prengler developed a SIMD engine targeted to vision algorithms with the capability to switch to MIMD processing [18]. Their technique reduces the large SIMD unit into many smaller SIMD units. While effective, it suffered from poor performance due to branch divergence, forcing expensive reconfigurations. In contrast, EFFEX does not require reconfiguration, and it benefits from special functional units along with memory optimizations.

The IBM Cell processor [9] is a heterogeneous architecture with some similarity to our proposed architecture. A major difference is that the Cell uses full vector processors and a conventional memory architecture. Our architecture focuses on specific vector reduction operations where vector data is reduced to smaller summary data, such as the inner product. This approach allows for smaller cores with specific vector instructions which take less area and power than a full vector processor. This also sets our approach apart from general vector processors. Furthermore, we use a memory system tuned specifically to vision algorithms.

3. FEATURE EXTRACTION ALGORITHMS

A typical feature extraction algorithm, as illustrated in Figure 2, is composed of five steps. The first step is to preprocess the image, an operation which typically serves to accentuate the intensity discontinuities (i.e., object boundaries) by, for example, eliminating the DC components (mean values) of the image. The second step scans the processed image for potential feature point locations; the specifics of this phase are highly dependent on the underlying algorithm. The third step of feature extraction works to filter out weak or poorly represented features through, for example, sorting the features found based on a key characteristic and then dropping the non-prominent results. The second and third steps implement a process typically called *feature point localization*. Once feature points are localized, the fourth step computes the feature descriptor. A *feature descriptor* is a compact representation of an image feature that encodes key algorithm-specific image characteristics, such as variations of pixel intensity values (gradients). The feature descriptor implements the illumination, scale, and rotation invariance supported by a particular algorithm. The fifth and final step of feature extraction performs another filter pass on the processed feature descriptors based on location constraints.

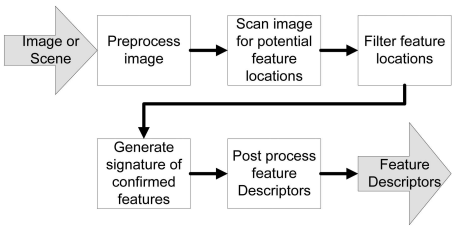

Figure 2: Overview of Feature Extraction This figure shows the general steps involved in feature extraction. The first three steps of the process locate feature points. The final two steps create feature vectors.

The quality of a feature extraction algorithm is evaluated on four major invariance characteristics: illumination, scale, viewpoint, and rotation. A very capable feature extraction algorithm will produce feature sets for an image that are nearly identical, despite changes in lighting, object position or camera position. In this work we focus on FAST [20], HoG [6], and SIFT [15]. These algorithms represent a wide trade-off of quality and performance, ranging from the high-speed low-quality FAST algorithm to the very high-quality and expensive SIFT algorithm. In addition, these algorithms are widely representative of the type of operations that are typically found in feature extraction algorithms.

3.1 Algorithm Performance Analysis

We analyzed the execution of the FAST, HoG and SIFT feature extraction algorithms to determine how their execution might benefit from hardware support. We instrumented a single-threaded version of each algorithm and profiled their execution.

FAST corner detection is designed to quickly locate corners in an image for position tracking [20]. It is the least computationally intensive and the least robust of the algorithms we examine. The FAST feature matching degrades when the scene is subject to changes in illumination, object position or camera location and image noise. As seen in Figure 3, the majority of time in the FAST algorithm is spent performing feature point localization. The algorithm locates corners by comparing a single pixel to the 16 pixels around it. To perform this comparison, the target pixel and surrounding pixels must be fetched from memory, and then the target pixel must be compared to all the pixels in the enclosing circle. The descriptor is made by concatenating the pixel intensities of the 16 surrounding pixels. Speeding up these comparisons, through a combination of functional unit and thread-level parallelism, greatly improves the performance of FAST.

HoG is commonly used for human or object detection [6]. The HoG algorithm is more computationally intensive than FAST, because it provides some illumination and rotation invariance. Figure 3 shows that HoG spends a significant amount of time performing feature localization and descriptor building. The descriptor is built using the histogram of the gradients of pixel intensities within a region, which are subsequently normalized. The major opera-

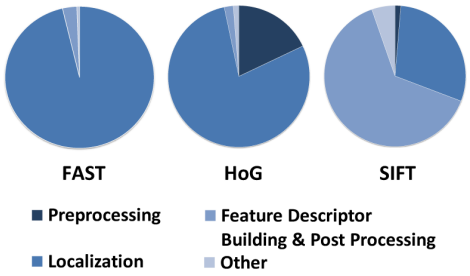

Figure 3: Feature Extraction Execution Time Distribution FAST spends most of the time locating the features using a comparison of a center pixel value to other pixels that form a circle around it. HoG spends most of the time localizing which involves building the descriptor for comparison. SIFT spends a large amount of time building the descriptor which involves gradient computations and binning.

tions for this phase of HoG computation are the fetching of image data from memory and the calculation of histograms using integral images [24]. This phase of the HoG algorithm utilizes a sliding window of computation in which each window is independent. Consequently, much parallelism is available to exploit. The second major time component is preprocessing, which for HoG is computation of the integral image. This is comprised mainly of memory operations, the computation of the image gradient, and finally the histogram binning of gradient values based on direction. More efficient computation of these components, through functional unit support and thread-level parallelism, significantly speeds up processing.

SIFT is a feature extraction algorithm widely used for object recognition [15]. It is the most computationally expensive and algorithmically complex of the feature extraction algorithms we examine, but it provides a high level of invariance to most scene changes. Figure 3 shows that the largest component of time is spent in feature descriptor building. This portion of the algorithm involves computing and binning the gradient directions in a region around the feature point, normalizing the feature descriptor, and accessing pixel memory. The operations in this phase are performed on each feature point and benefit from specialized hardware. The second largest component of SIFT is the feature point localization. This portion is dominated by compare and memory operations to locate the feature points. There are also 3D curve fitting and gradient operations to provide sub-pixel accuracy and filter weaker responses, respectively. This phase of SIFT provides ample thread-level parallelism. The preprocessing step, the third most expensive component in SIFT, involves iterative blurring of the image which is a convolution operation. The convolution requires multiplying a region of the image by coefficients and summing the result, operations which can benefit from specialized functional unit support.

4. PROPOSED ARCHITECTURE

To meet the needs of feature extraction in the embedded space, a number of design criteria exist. First, the design must run feature extraction algorithms efficiently, as close to real-time analysis of high-resolution images as possible. To this end, we seek to exploit the ample explicit parallelism in these algorithms, plus employ the latency reduction capabilities of specialized functional units and memory architectures. Second, efficient execution of feature extraction algorithms must be possible at low power and area cost, as the mobile space predominantly relies on low-cost untethered platforms using batteries. To meet these challenging criteria, we employ many computational resources with a simple energy-frugal design, to lower overall power demands. Finally, the design must be able to run a variety of feature extraction algorithms to accommodate the fast-moving pace of feature extraction algorithm development. To serve this demand, we employ an application-specific processor design, aimed at providing efficient execution of feature extraction algorithms across a wide range of applications.

4.1 Support for Heterogenous Parallelism

All of the feature extraction algorithms that we studied demonstrated a dichotomy of internal algorithmic styles. At a high level, the algorithms are quite complex, with significant control-oriented code that displayed many irregular branching patterns, but with a large amount of the workhorse computation that exhibited repetitive patterns for feature point localization. Given these contrasting code styles, we employ a heterogeneous multicore architecture. The EFFEX high-level architecture is illustrated in Figure 4. Our architecture has a single complex superscalar core, coupled with a variable number of simple single-issue cores. The complex core is used to perform high-level tasks that require complex decision making while the simple cores plow through the vast amount of simple repetitive tasks that often require unpredictable control flow using specialized vector instruction units. Since the simple cores require only a fraction of the area of the complex core, this heterogeneous multicore design approach yields significant performance per unit area for feature extraction algorithms.

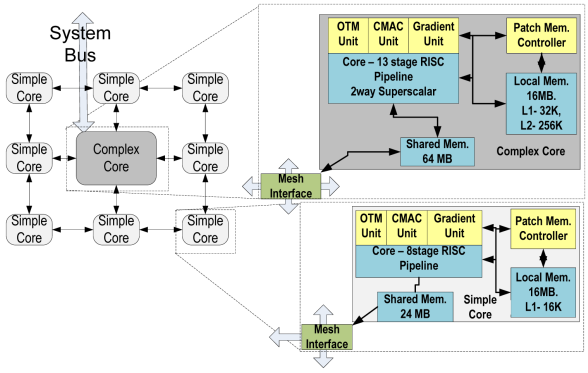

Figure 4: EFFEX Architecture This shows the overall EFFEX embedded architecture. There is one complex core surrounded by multiple simple cores on a mesh network. Each core has application-specific functional units and a patch memory architecture.

The image under analysis is shared among all of the processing units, and each simple core must have local memory to store intermediate results. For our design we use an uncached distributed shared memory in a NUMA mesh [10]. The feature extraction algorithms have a large amount of parallelism and little communication demands between the worker threads: the shared memory is primarily used to hold the image under analysis, plus work requests sent by the complex core to the simple core. Additionally, the simple cores return computed feature descriptors to the complex core through the shared memory. As such, we rely on explicit inter-processor communication through shared memory. Synchronization between the cores is implemented with barriers that reside in the shared memory.

4.2 Functional Units for Feature Extraction

Our analysis of the three feature extraction algorithms revealed three application-specific functional units that could be used to accelerate their execution. These functional units are useful across a broad array of feature extraction algorithms. In general, the functional units provide vector reduction instructions for the cores. Every core has one of the following functional units.

One-to-many Compare Unit Searching for the feature point locations typically involves a large number of compares between a pixel location and its neighbors. For example, SIFT compares a image pyramid pixel value to its immediate neighbors' values while FAST compares a center pixel value to surrounding pixels. The one-to-many compare (OTM) unit, illustrated in Figure 5, significantly speeds up this processing by performing the comparisons in parallel through the use of an instruction extension. The one-to-many compare unit takes a primary operand and compares it against 16 other values in a single operation. The unit returns the total number of values that are less than (or greater than) the instruction operand along with a basic result that is one if all compares are true, zero otherwise. The unit can also be used for histogram binning by comparing a value to the limits of the histogram bins. This makes the output equal to the number of the histogram bin.

The one-to-many compare unit is loaded with image data using a single bulk load operation from patch memory (see Section 4.3), which takes approximately 9 cycles. The comparisons complete in 32 additional cycles. All together, a one-to-many comparison is 3.5 times faster with the specialized functional unit support, compared to executing the necessary operations on an typical ARM A8 core running at the same clock speed.

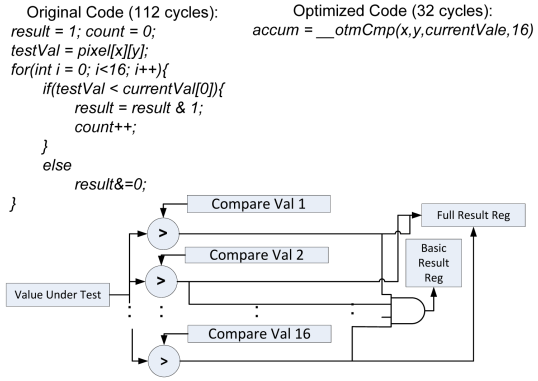

Figure 5: One-to-Many Compare and Binning Unit This functional unit is used frequently in SIFT and FAST. It can be used for comparing one pixel to many others all at once. It has fast access to patch memory, allowing multiple values to be retrieved and compared quickly. It can also be used for efficiently computing histogram bins.

Convolution MAC The second specialized functional unit is a convolution multiply accumulate (CMAC) unit. The CMAC unit takes two floating vectors and performs an inner product operation on them, as shown in Figure 6. This operation is performed in SIFT and HoG for image preprocessing and for vector normalization. The vectors are loaded using a bulk load from patch memory which takes 36 cycles for two 32-entry vectors. Once the vectors are loaded the CMAC unit can complete a convolution step in 6 cycles. This is up to 96 times faster than an ARM A8 running at the same clock speed as EFFEX.

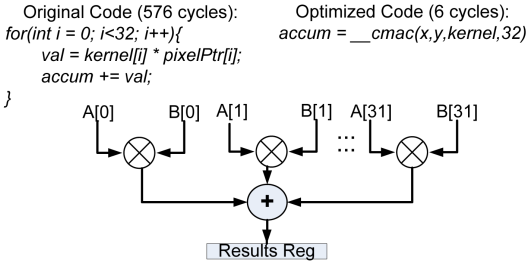

Figure 6: Convolution MAC Unit This unit is a MAC that takes vectors from patch memory and implements fast convolutions. This unit is used in SIFT for preprocessing and HoG for vector normalization.

Gradient Unit Both SIFT and HoG have many image gradient computations while building the feature descriptors. We have included the gradient functional unit, illustrated in Figure 7, to speed up this computation. This functional unit computes the gradient of an image patch using a Prewitt version of the Sobel convolution kernel [3]. The unit operates on single-precision floating point data, and it is able to compute the gradient in both the x and y directions at the same time. The gradient unit uses patch memory to quickly load pixel operands from memory based on a pixel address operand for the gradient instruction. The gradient unit is able to perform both the x and y gradient operation for a $3x3$ pixel patch in 33 cycles which is a speedup of 3.57 over an execution on an ARM A8 processor running at the same clock speed.

4.3 Patch Memory Architecture for Fast Pixel Access

Feature extraction algorithms generally inspect 2D regions of an image, leading to high spatial locality in accesses. However, this

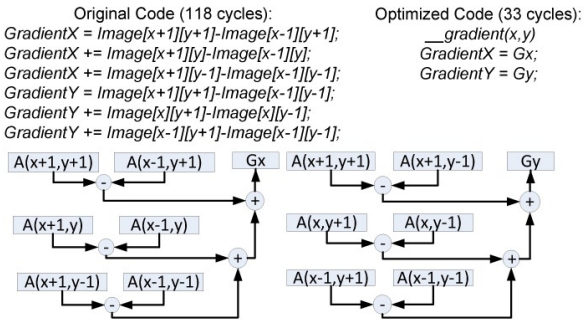

Figure 7: Gradient Unit This unit uses access to patch memory to quickly compute the gradient at a pixel location. This unit computes the x and y gradient at the same time. This unit is used frequently in SIFT and HoG. It is a small SIMD engine interfaced to patch memory architecture.

spatial locality is relative to the 2D space of pixel locations. The traditional approach of a scan-line ordered layout of pixels in memory, as illustrated in the left image of Figure 8, is an inefficient way to store pixel data. Access to pixel data in scan line order results in image data residing in multiple DRAM rows which leads to long latencies and wasted energy.

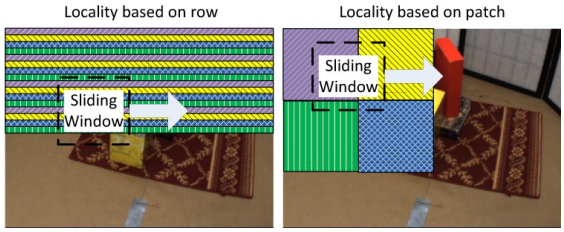

Figure 8: Patch Memory Access Pattern This figure illustrates the storage of image data in traditional DRAM (left) and our patch based DRAM (right). Each color/pattern is a different DRAM row containing data. In traditional DRAM storage, a single sliding window will access a large number of different DRAM rows, while in patch-based memory the window only accesses at most four DRAM rows for a reasonably sized window.

The *patch memory architecture* is a memory system design that combines hardware and software optimizations to significantly speed up access to pixel data. Software is optimized to store pixel data in region-order. In our design regions are defined as small two-dimensional patches of pixels that are n pixels wide by m pixels high which fit evenly within a single DRAM row, resulting in the memory layout shown in the right image of Figure 8. Any access to a single memory patch typically only requires reading one DRAM row, and subsequent reads of nearby patches can often be serviced out of the DRAM row buffer. When the region accessed spans the boundaries of the region-ordered memory, multiple DRAM rows may need to be accessed. As such, multiple DRAM row buffers (four in our design) are desired to ensure that data is quickly available.

Image data access requires translation of the pixel location to a memory address, an operation typically performed in software. Calculating addresses in patch memory requires a similar amount of computation. Given the high frequency and irregularity of pixel accesses in feature extraction, we provide special patch memory instructions and hardware support to convert an integer (x, y) pixel address quickly into its corresponding patch memory address, as illustrated in Figure 9. The specialized address generation unit checks if subsequent accesses are in the same memory patch, and in this case omits an unnecessary recomputation of the pixel address.

5. PERFORMANCE ANALYSIS

5.1 EFFEX Performance Model

We simulated our system using a technique similar to Graphite [17]. We modeled the EFFEX architecture using an ARM A8 like custom model for the complex cores, and an ARM A5 (with floating point)

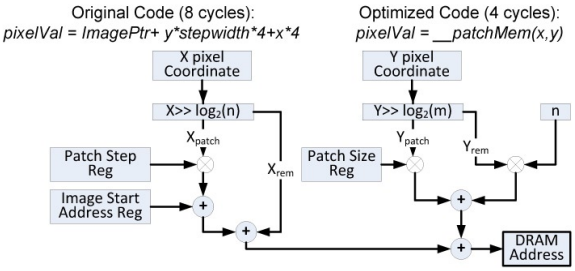

Figure 9: Patch Memory Address Computation This figure shows the logic required to compute the patch memory DRAM address based on the pixel x and y coordinates. Using this address computation method the memory can automatically place the data in the optimal access pattern for many of the feature extraction algorithms.

model for the simple cores, all running at 1 GHz. Each simulated processor contained a CMAC, OTM compare unit, and Gradient Unit along with a patch memory controller. The memory system was modeled such that each core had a local cached private memory and uncached access to the main shared memory. The interconnect between the cores was modeled as a mesh network. Table 1 lists the basic configuration parameters.

We modeled shared and local memory using a custom memory simulator with support for 16 pixel by 16 pixel patch memory in the local memories, and a traditional row-oriented DRAM architecture for the main shared memory. Power estimates for the memory system were implemented using activity counters, similar to the approach in Wattch [4]. To gauge power and area estimates for the application-specific functional units, they were synthesized from Verilog using Synopsys synthesis tools for a 65 nm process node. The CMAC unit is based on the work in [12]. The area overhead of the EFFEX cores can be seen in Table 2. The base complex and simple core areas and power were based on [2] and [1], respectively. For comparison, we modeled an embedded GPGPU similar to the PowerVR SGX family [16] with an 8-thread embedded GPGPU (2 sets of 4 threads) running at 110 MHz.

Table 1: EFFEX Configuration

Feature	Configuration
Core Clocks:	1 GHz
Complex Core:	32 bit RISC in-order 2-way superscalar
Complex Pipeline:	13-Stage
Complex Local Cache:	32k instr. and data 256k unified L2
Simple Core:	32 bit RISC in-order
Simple Pipeline:	8-Stage, single issue
Simple Local Cache:	16k instr. and data
Local Memory Clock:	1GHz
Local Memory Size:	16MB
Shared Memory Clock:	500MHz
Complex Core Shared Mem. Size:	64MB
Simple Core Shared Mem. Size:	24MB
Total Shared Memory Size:	256MB
Memory Bus Width:	128 bits
Processor Interconnect:	Mesh

Table 2: Area estimates for the EFFEX 9-core processor These estimates assume a 65 nm silicon process.

Module	Area (mm^2)
One-to-many Compare	0.0023
Gradient Unit	0.0034
Convolution MAC	0.5400
Total for functional units per core	0.5457
Complex Core	4.0000
Simple Core	0.8600
Total for 9 Core EFFEX	15.8000
SIMD Complex core w/o EFFEX	6.0000
Normal Complex core w/Embedded GPGPU	16.5000

5.2 Benchmarks

The computer vision algorithms analyzed were the FAST, HoG, and SIFT feature extraction algorithms. The base implementations for SIFT, HoG and FAST are from [11], [8], and [21], respectively. We modified the algorithms to expose explicit parallelism for our heterogeneous multicore. For test inputs, we randomly selected fifteen 1024x768 images as a data set. The algorithms were compiled for Linux using the GNU GCC compiler set to maximum optimization. All GPGPUs used the CUDA SIFT implementation from [26].

The algorithms were transformed into multithreaded versions by splitting the algorithms into phases at logical serialization points. Within a phase the image processing was split into regions, and each thread performed computation on a separate region. Each thread was assigned to a different core with synchronization performed using barriers in shared memory.

We simulated all the algorithms running on: i) a single complex core without EFFEX functional units, ii) a complex core with EFFEX functional units, and iii) an EFFEX with a complex core and a variable number of simple cores. We also ran the SIFT algorithm on i) a 65 nm 1.2 GHz NVIDIA GTX260, ii) a 45 nm 2.67 GHz Intel Core 2 Quad Q8300 with 8 GB of RAM and iii) a simulated 110 MHz embedded GPGPU. We simulated the embedded GPGPU with the aid of an NVIDIA GT210.

5.3 Simulation Results

Figure 10 shows the speedup that EFFEX achieves over a single complex core lacking SIMD and the EFFEX enhancements. The plot shows that EFFEX is capable of achieving a significant speedup over a basic embedded processor. It also shows that for SIFT and FAST, adding 4-way 32 bit floating point SIMD instructions to EFFEX has negligible impact on the speedup. The speedup in HoG increases more with SIMD because SIMD instructions speed up a portion of the algorithm that the EFFEX enhancements do not. This shows that EFFEX can extract enough parallelism that there is typically not much need for SIMD. In the remaining experiments EFFEX is configured without SIMD support.

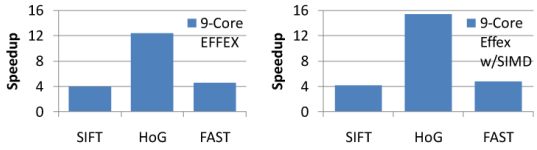

Figure 10: 9-Core EFFEX Speedup The left graph shows the speedup of a 9-Core EFFEX configuration without general SIMD support for three algorithms versus a single complex core without EFFEX enhancements. The graph on the right shows the same comparison but for a 9-Core EFFEX with general SIMD instruction support as well.

Figure 11 demonstrates that as the number of simple cores increases, so does the performance of EFFEX. It can be seen that at around 4 total cores the EFFEX solution begins to outperform the embedded GPGPU. This is due primarily to the efficiency of the vector reduction operations that EFFEX performs, which run much less efficiently on the embedded GPGPU.

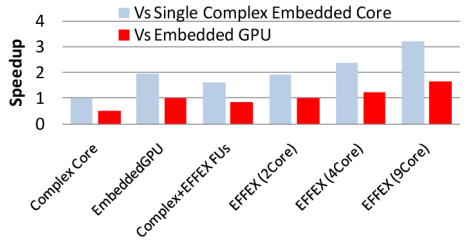

Figure 11: Scalability for SIFT Running on EFFEX This graph shows the scalability of the EFFEX cores running SIFT when compared to a complex embedded core (without EFFEX enhancements) and an embedded GPGPU.

The 9-Core EFFEX solution has slightly less area than a typ-

ical embedded core combined with an embedded GPGPU; however, Figure 11 confirms that the EFFEX solution has higher performance. EFFEX maintains higher performance due to the ability that each core can take a divergent control path without hindering the performance of another core and due to the tighter interprocessor integration afforded by EFFEX.

The performance-cost benefits of EFFEX versus other computing solutions can be seen in Figure 12. This plot shows the performance of the various designs (in frames processed per second) per unit of cost (either silicon area or power). Clearly the EFFEX solution is capable of providing a higher performance per unit cost than other solutions, making the EFFEX design particularly attractive for cost sensitive embedded targets. This result is due primarily to the efficiency of the vector reduction instructions and the patch based memory. We found that the patch memory architecture allows for a decrease in the total number of cycles while also decreasing the total energy. For example for a run of FAST the number of memory cycles decreased from 39 million to 937 thousand while the energy went from 6 J to 12 mJ. This provides a significant performance boost at very low cost.

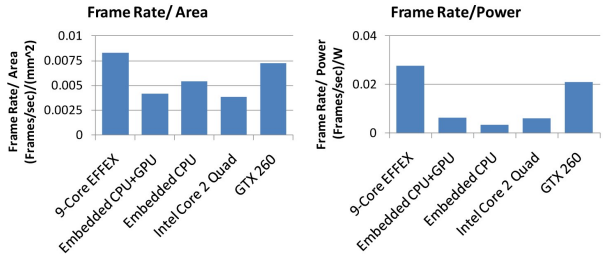

Figure 12: EFFEX Normalized Performance Running SIFT This graph shows the normalized performance of EFFEX when compared to other computing solutions. EFFEX outperforms embedded single cores, embedded GPGPUs, desktop CPUs, and desktop GPUs when the performance is normalized by cost. The performance here is measured by the number of 1024x768 frames processed per second.

Figure 13 shows a Pareto chart comparison of various computing solutions using the metrics of execution time per frame and overall cost ($area * power$). In the Pareto chart, better designs are in the lower left of the chart, as these designs have faster frame rates and lower overall cost. $Area * power$ is used for the x axis because it captures two key cost factors into a single metric. The figure shows that while the embedded solutions are not as fast as the desktop GPU and CPU, they are far less expensive in terms of cost. Furthermore, a 9-core EFFEX is cost effective and has the fastest frame rate for the embedded solutions.

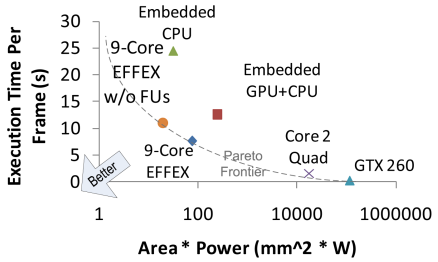

Figure 13: Performance versus Overall Cost Running SIFT This graph shows the performance and area and power cost for various computing solutions. The cost is measured in $area * power$ on the x axis while execution time is along the y axis. In this study the desktop CPU and GPU are faster but their power and area make them unattractive for the embedded space. In the embedded space, the 9-core EFFEX has the highest frame rate, making it a very a cost-effective solution.

6. CONCLUSIONS

We have analyzed the execution characteristics of three common feature extraction algorithms: FAST, HoG, and SIFT. Using this analysis, we presented EFFEX, an efficient heterogeneous multicore architecture that utilizes specialized functional units and an optimized memory architecture to significantly speedup feature extraction algorithms on embedded mobile platforms. We presented three specialized functional units in our design, including a one-to-many compare, convolution and gradient functional units that can take advantage of our patch memory architecture. Our application-specific heterogenous multicore is capable of improving the performance of embedded feature extraction by over 14x in some cases. We have shown that our architecture is more cost effective than a wide range of alternative design solutions, and it effectively executes a wide variety of feature extraction algorithms.

We are currently pursuing multiple future directions for this work. We are working to expand EFFEX's coverage of computer vision algorithms to include the machine learning based algorithms used to analyze extracted features. We are also investigating multiple techniques to further improve pixel memory performance.

7. ACKNOWLEDGEMENTS

The authors acknowledge the support of the National Science Foundation and the Gigascale Systems Research Center.

8. REFERENCES

[1] ARM. Cortex-A5 Processor, 2010. http://www.arm.com/products/processors/cortex-a/.
[2] ARM. Cortex-A8 Processor, 2010. http://www.arm.com/products/processors/cortex-a/.
[3] D. G. R. Bradski and A. Kaehler. *Learning OpenCV*. O'Reilly Media, Inc., 2008.
[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.
[5] L. Chang and J. Hernández-Palancar. A Hardware Architecture for SIFT Candidate Keypoints Detection. In *CIARP*, 2009.
[6] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *CVPR*, 2005.
[7] Google. Google Goggles For Android, September 2010. http://www.google.com/mobile/goggles/#text.
[8] S. Goyal. Object Detection using OpenCV II - Calculation of HoG Features, October 2009. http://smsoftdev-solutions.blogspot.com/2009/10/object-detection-using-opencv-ii.html.
[9] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *MICRO*, 26(2), 2006.
[10] D. E. Guller, J. P. Singh, and A. Gputa. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.
[11] R. Hess. SIFT feature detector for OpenCV, February 2009. http://web.engr.oregonstate.edu/~hess.
[12] S. Jain, V. Erraguntla, S. Vangal, Y. Hoskote, N. Borkar, T. Mandepudi, and V. Karthik. A 90mW/GFlop 3.4GHz Reconfigurable Fused/Continuous Multiply-Accumulator for Floating-Point and Integer Operands in 65nm. In *VLSID*, Jan. 2010.
[13] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura. Hardware Architecture for HOG Feature Extraction. In *IIH-MSP*, sep. 2009.
[14] Layar. Layar Reality Browser, September 2010. http://www.layar.com/.
[15] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 2004.
[16] I. T. Ltd. SGX Graphics IP Core Family, 2010.
[17] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA*, January 2010.
[18] A. Prengler and K. Adi. A Reconfigurable SIMD-MIMD Processor Architecture for Embedded Vision Processing Applications. *SAE World Congress*, 2009.
[19] J. Qiu, Y. Lu, T. Huang, and T. Ikenaga. An FPGA-Based Real-Time Hardware Accelerator for Orientation Calculation Part in SIFT. *IIH-MSP*, 2009.
[20] E. Rosten and T. Drummond. Fusing points and lines for high performance tracking. In *ICCV*, volume 2, October 2005.
[21] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *ECCV*, May 2006.
[22] B. Silpa, A. Patney, T. Krishna, P. Panda, and G. Visweswaran. Texture filter memory; a power-efficient and scalable texture memory architecture for mobile graphics processors. In *ICCAD*, 2008.
[23] J. Skribanowitz, T. Knobloch, J. Schreiter, and A. Konig. VLSI implementation of an application-specific vision chip for overtake monitoring, real time eye tracking, and automated visual inspection. In *MicroNeuro '99*, 1999.
[24] P. Viola and M. Jones. Robust real-time object detection. *IJCV*, # 2002.
[25] J. Wortham. Customers Angered as iPhones Overload ATT. *New York Times*, September 2009.
[26] C. Wu. SIFTGPU, September 2010. http://www.cs.unc.edu/~ccwu/siftgpu/.