EVA: An Efficient Vision Architecture for Mobile Systems

Jason Clemons, Andrea Pellegrini, Silvio Savarese, and Todd Austin

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan 48109
{jclemons, apellegrini, silvio, austin}@umich.edu

Abstract

The capabilities of mobile devices have been increasing at a momentous rate. As better processors have merged with capable cameras in mobile systems, the number of computer vision applications has grown rapidly. However, the computational and energy constraints of mobile devices have forced computer vision application developers to sacrifice accuracy for the sake of meeting timing demands. To increase the computational performance of mobile systems we present EVA. EVA is an application-specific heterogeneous multicore having a mix of computationally powerful cores with energy efficient cores. Each core of EVA has computation and memory architectural enhancements tailored to the application traits of vision codes. Using a computer vision benchmarking suite, we evaluate the efficiency and performance of a wide range of EVA designs. We show that EVA can provide speedups of over 9x that of an embedded processor while reducing energy demands by as much as 3x.

Categories and Subject Descriptors C.1.4 [*Parallel Architectures*]: Mobile Processors

General Terms Design, Performance

Keywords Computer Vision, Mobile, Architecture

1. Introduction

The technological growth and the commercial success of personal mobile devices – such as smart phones and tablets – is a remarkable success for computing history. Starting as devices which only provided basic features such as voice and text communications, mobile platforms have evolved into complex devices capable of running productivity applications and offering performance comparable to some desktop computers.

Along with increasing processing capability, mobile devices have seen improvements in peripherals that increase system capabilities. Most modern smartphones and tablet computers embed one or more cameras for taking photographs or HD videos [3, 19]. Such powerful cameras, coupled with high performance microprocessors, are giving rise to many new applications in mobile computing such as Google Glass, which features glasses with a camera and display in the lens that is connected to a mobile processor that can run a full OS [2]. Google Glass implements augmented reality, which allows users to point their smart devices at a scene or images in the real world and have useful information rendered with objects in the 3D scene [7]. Computer vision algorithms, once practical only on high performance workstations, are now becoming viable in the mobile space thanks to the technological improvements made in portable devices. As computer vision applications continue to make their way into mobile platforms, there is a growing demand for processors that can tackle such computationally intensive tasks. Unfortunately, mobile processor performance is limited by cost and energy constraints [15]. While desktop processors can consume 100 Watts over 250 mm^2 of silicon, typical mobile processors are limited to a few Watts with typically 5 mm^2 of silicon [4] [22].

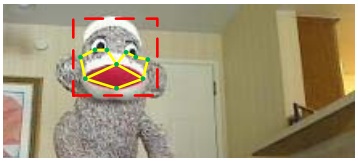


Figure 1: **Computer Vision Example** The figure shows a sock monkey where a computer vision application has recognized its face. The algorithm would utilize features such as corners and use their geometric relationship to accomplish this.

To meet the limited computation capability of mobile processors, computer vision application developers reluctantly sacrifice image resolution, computational precision or application capabilities for lower quality versions of vision algorithms. Thus there is an unsatiable demand for high-performance vision computing in the mobile space. To meet the performance, cost and energy demands of mobile vision, we propose EVA, a novel architecture that utilizes an application-specific design, tailored to the specific application characteristics of mobile vision software.

1.1 Our Contribution

We present EVA, our solution for efficient vision processing in the mobile domain. EVA is a heterogeneous multicore architecture with custom functional units designed to increase processing performance for a wide range of vision applications. Our design leverages a heterogenous multicore architecture, where more powerful cores coordinate the tasks of less powerful, but more energy efficient cores. Both types of cores are enhanced with specific custom functional units specially designed to increase the performance and energy efficiency of most vision algorithms. Furthermore, EVA develops a novel, flexible, memory organization which enables efficient access to the multidimensional image data utilized in many vision workloads. We also examine the thread-level parallelism available in vision workloads and evaluate the tradeoff between number of cores, energy and speedup. This work makes three primary contributions:

- The EVA application-specific architecture includes a selection of custom functional units tailored to mobile vision workloads including a novel accelerator for improved performance on decision tree based classification.
- The EVA memory system introduces the tile cache, a cache with a flexible prefetcher capable of handling both 1D and 2D memory access patterns efficiently.
- Using the MEVBench [15] mobile vision benchmark suite and full-system simulation, we explore various configurations of the EVA design and demonstrate that it can provide significant energy and performance improvements while being held to the tight cost constraints of mobile systems.

978-1-4799-1400-5/13/$31.00

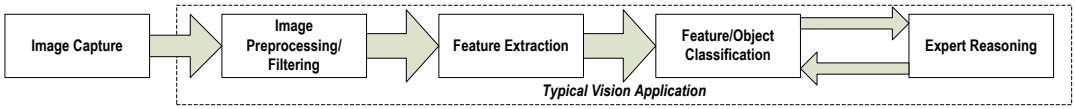


Figure 2: **Vision Software Pipeline** The figure shows a typical computer vision software pipeline. The image is captured using an imaging device such as a camera. The captured image is filtered to eliminate noise. Then, features are extracted from an image. The features are "classified" based on prior knowledge. Expert reasoning is utilized to generate knowledge about the scene.

We begin with a computer vision primer in Section 2. Section 3 discusses the computational traits of mobile vision applications. Section 4 details our architectural solution for efficient mobile vision computation. In Section 5 we discuss our experimental setup, and in Section 6 we present our experimental results. In Section 7 we look at related works. Finally in Section 8, we conclude and discuss future work.

2. Background On Mobile Vision Applications

The field of computer vision is a synergy between image processing, artificial intelligence, and machine learning. Computer vision algorithms analyze, process and understand the objects found in image data. For example, Figure 1 shows a picture where a computer vision application utilized a feature detection algorithm to locate the monkey's face. There is a wide variety of applications that can benefit from computer vision, such as defense, surveillance, and autonomous vehicles.

A typical computer vision software pipeline can be seen in Figure 2. This pipeline consists of five components: image capture, image preprocessing, feature extraction, feature/object classification, and expert reasoning. A challenge in optimizing for the vision pipeline's computation lies in the great variety of computational characteristics within the key kernels.

The first phase (*image capture*) involves capturing the imaging information from a sensor. In mobile vision applications the processor typically retrieves an image from the camera. This step is commonly an API call to the device driver although more capabilities are being exposed to user applications through software libraries such as FCAM [1]. The second phase (*image filtering*) involves applying filtering techniques to the imaging data to increase the discernibility of information in the data. Commonly, this phase is merged with the third phase (*feature extraction*). Feature extraction consists of the localization of salient image characteristics, such as corners or brightness contrast, and the generation of unique signatures for the located features. These signatures are commonly stored as one dimensional vectors referred to as feature vectors. Feature extraction has been shown to be highly compute-intensive [15].

The fourth phase (*classification*) utilizes machine learning algorithms to determine what objects could be represented by the features located in the image. Classification results are used for semantic and relational processing to better understand the components of the scene. For example, a feature may be classified as belonging to a known object. The final phase (*expert reasoning*) utilizes the information from the previous phases to generate specific knowledge about the original scene. This task may be as simple as recognizing a face in the scene, or as complex as a predicting where a group of people are headed. The expert reasoning phase can iterate with the classification component to refine its outcome, and this process is ultimately responsible for the output of the vision system. This iterative loop allows the developer to improve the quality of the result given sufficient computation capability.

3. Application Traits

To build optimized computing platforms, it is imperative to fully understand the underlying algorithms and the traits of their computation. Through analysis of the MEVBench mobile vision benchmarks [15], we present three underlying characteristics that can be exploited in hardware to improve the performance of a mobile vision codes. These characteristics are: frequent vector reduction operations, diverse parallel workloads, and memory locality that exists in both one and two dimensions.

3.1 Frequent Vector Reduction Operations

Vector operations have been a common target for architectural optimization for decades. From vector processors to digital signal processors (DSP) to General Purpose Graphics Processing Units (GPGPUs), there is a lot of work on the processing of vector operations. This prior work has primarily focused on instructions that perform the same operation on a set of data or single instruction multiple data (SIMD) operations. While these operations are quite common within many applications, there is another class for vector operations that is not often considered.

During our investigation of mobile vision applications, we found that vector reduction operations occur frequently. While vector operations and vector reduction operations both take vectors as inputs, the former produces a vector result while the latter produces a scalar result. Current processors support SIMD instructions that allow computation on vectors [5] [20]. These solutions perform some functions similar to our accelerators, however they do not typically include the reduction step. Examples of two common vector reduction operations in vision codes are the dot product and max operations. These operations are used in preprocessing, image filtering, feature extraction, classification, application reasoning and result filtering. Despite the prevalence of vector reduction operations, most architectures primarily support the classical vector operations. Figure 3 shows the frequency of the dot product operations in the benchmarks. The figure also shows the size of the vectors these operations are operating upon and the number of floating point multiply-accumulates that result. We examined the run time of the benchmarks to find hot spots and instrumented the calls to these types of operations. We found dot products, monopoly compares, tree compares and max compares to be the most common vector operations.

The strict energy and performance constraints of mobile systems create the need to optimize vector reduction operations in hardware. In the mobile vision application space, these operations create opportunities to decrease execution runtime while also reducing energy. EVA provides first-class support to this architecturally underserved class of operations.

3.2 Diverse Parallelism

Computer vision workloads have been shown to contain thread-level parallelism by allowing multiple cores to work simultaneously to increase performance [15]. Unfortunately, the workloads thread are not typically well balanced.

Figure 4 shows the performance of the feature extraction and classification benchmarks from MEVBench when running with varied number of threads. The data for this figure was taken using an ARM A9 and timing each thread separately. The time to complete each benchmark is measured as the maximum time of all the threads in the workload to complete. The average speedup of both types of algorithms is plotted along with their geometric mean for a given number of cores. This figure demonstrates the duality of the workloads in vision applications. On one hand the feature classification workloads scale well with the number of cores, while on the other hand the feature extraction workloads quickly reach an asymptotic limit. We found that the feature extraction

workloads are limited by the performance of the coordination component more so than in the classification application. Most vision applications show similar behavior to feature extraction, resulting in applications having limited thread-level parallelism due to coordination bottlenecks.

3.3 One and Two Dimensional Locality

Most mobile vision workloads utilize imaging data as input. The initial phase of processing typically involves analyzing the image data in 2D pixel patches. We examined the source code of MEVBench and found that the feature extraction algorithms often access image data in two dimensions (e.g., when computing feature vectors), while the classification algorithms work on one dimensional feature vectors, confirming the result of [15]. Unfortunately, 2D locality does not transfer well into the typical raster scan order of pixel rows. When pixels are stored in raster scan order, two pixels that are vertically adjacent are separated in memory by a step of at least the width of the image in pixels times the size of a pixel. Thus, the typical linear approach to storage can can lead to inefficient access patterns, as they will often incur cache or DRAM row misses. However, the typical next phase of the vision pipeline, classification, utilizes linear vectors. In this phase 1D spatial locality is ample and readily exploited by current cache architectures. In order to optimize memory accesses in all phases of vision algorithms, the system needs to efficiently support both 1D and 2D locality.

4. EVA Hardware

We have developed architectural features that provide capabilities targeted to the characteristics of the mobile vision application space. We add in accelerators for vector reduction operations such as dot

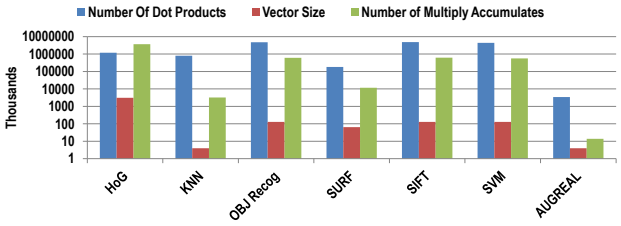


Figure 3: **Number of Dot Product Operations** The figure shows the number of dot product operations in the analyzed benchmarks for a small input size. The floating point multiply operations take a minimum of 5 cycles on ARM and thus each multiply accumulate operation has a large impact on execution time.

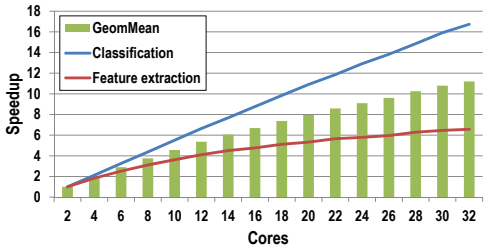


Figure 4: **Thread-Level Parallelism Within Vision Algorithms** The figures shows the idealized performance of the feature extraction and feature classification benchmarks from the MEVBench suite. The figure shows the average speedup for each type of benchmark. It also shows the geometric mean of the speedups. Speedups are closer to linear for the feature classification while the feature extraction speedups quickly begin to saturate. Thus a system needs to balance both types of workload.

product and tree compares. We introduce a software-enabled 2D locality prefetcher called the tile cache. Finally, we exploit diverse parallelism through the introduction of heterogeneous cores.

4.1 EVA System Architecture

EVA is designed to efficiently handle mobile vision workloads. An overview of a system with EVA can be seen in Figure 5. All cores in EVA contain a set of custom accelerators to handle the common vector reduction operations. In particular, EVA cores have units for performing the dot product, monopoly compare, max compare, and tree compare. These custom accelerators are designed to reduce both the effenergy and latency of their target operations. EVA's cache has been modified to support both 2D and 1D locality in memory accesses. This optimization permits improved memory performance when accessing image data.

Our solution takes into account that many mobile vision applications can extract thread-level parallelism, thus EVA is designed as a heterogenous multicore comprised of two types of ISA-compatible cores. The first type of core, called *coordinating core*, is a powerful 4-wide out-of-order core designed to efficiently handle sequential code that can not be parallelized effectively, such as that used to coordinate the work of a group of threads. The second type of core, called *supporting core*, is a low-power core that exploits thread-level parallelism by efficiently running the worker threads. The supporting cores eliminate many of the costly architectural features of the coordinating core. In particular, the supporting cores are 2-way superscalar cores as opposed to the 4-way issue that the coordinating core supports. They also have only 2 ALUs instead of the 4 found in the coordinating core. Their physical register file is reduced in size by 25% compared to the larger core. The ratio of coordinating to supporting cores and the total number of EVA cores can be configured based on the constraints of the system and the key application characteristics.

In the example system of Figure 5, EVA's cores are connected through a bus with fast snoop controllers. We chose this interconnect strategy due to its common usage in mobile designs such as Tegra 3 [31]. Additionally, this design lends itself to the workload characteristics of mobile vision where most of the time cores work on local data, allowing for efficient use of the bus. EVA's features do not rely on a specific interconnection topology. For more than eight cores, the bus would need to be replaced with a network-on-chip. The interconnect allows EVA to utilize a shared memory multicore architecture with a shared L2 cache that is non-inclusive. The cache coherency protocol is MOESI. The EVA cores communicate utilizing the Pthreads software library. Our system utilizes memory mapped I/O to access external devices. External subsystems can generate interrupts to the EVA cores and vice versa for coordination. For example, once an image is captured the image subsystem can produce an interrupt in the coordinator core alerting it of the new data. The image can then be retrieved from memory.

4.2 Custom Accelerators

The EVA accelerators take advantage of 64 32-bit floating point registers present in mobile SIMD units such as ARM NEON [5]. In typical modern SIMD units for mobile platforms these registers can be accessed individually or in groups of two or four single precision registers. EVA's accelerators require the extension of this ability to groups of up to sixteen registers. EVA assumes the ability to read the registers in groups of eight in one cycle, and that the register operands that are aligned in groups of eight i.e., 0, 8, 16 etc.

4.2.1 Dot Product Accelerator

As shown in Figure 3, the dot product occurs often in many vision codes. For example, the dot product is used to perform convolution for image filtering and also to normalize vectors during feature extraction. It is also a common operation in the classification phase for comparing various feature vectors. The operation performed by a dot product can be seen in Equation 1. The operation works by

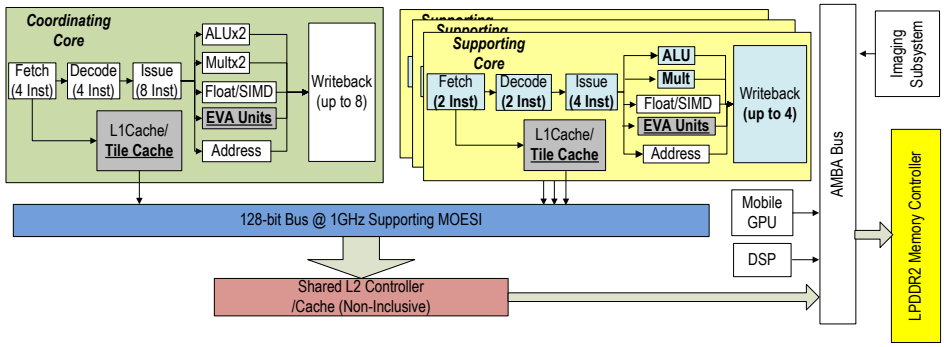


Figure 5: **An EVA-based System Overview** The figure shows an example of a mobile SoC with an EVA configuration as the primary processor. EVA contains a set of cores made up of two types: coordinating cores and supporting cores. Coordinating cores are powerful superscalar out-of-order processors who execute serial code, participate in parallel execution and coordinate the work of the supporting cores. The supporting cores are smaller and more energy efficient but ISA compatible with the coordinating core. Both types of cores contain accelerators for mobile vision applications. The EVA cores' 1GHz bus connects them to a shared L2 cache. The L2 cache connects to a to a LPDDR2 memory controller. EVA can communicate with external computing units using the AMBA bus. In the system configuration shown, the OS primarily runs on the coordinating core and schedules applications across the EVA cores based on resource availability. Another viable configuration is to execute the OS and non-vision applications on a separate mobile processor connected to EVA through the AMBA bus.

Table 1: **EVA Accelerator Instructions** The instructions added to utilize the EVA accelerators.

Instruction	Operand A	Operand B	Operand C	Result	Instruction
MONOCMP Monopoly Compare	F[m] Value	F[n] to F[n+15] Vector		R[k] Results	MONOCMP F[m], F[n], R[k] example: MONOCMP F[0], F[16], R[0]
TRECMP Tree Compare	F[m] to F[m+6] Feature Vector	F[n] to F[n+6] Tree Vector		R[k] Node Value	TRECMP F[m], F[n], R[k] example: TRECMP F[0], F[8], R[0]
MXCMP Maximum Compare	F[m] to F[m+7] Vector			R[k] Index of Maximum	MXCMP F[m], R[k] example: MXCMP F[0], R[0]
DOTPROD Dot Product	F[m] to F[m+15] Vector	F[n] to F[n+15] Vector		F[k] Result	DOTPROD F[m], F[n], F[k] example: DOTPROD F[0], F[16], F[32]
PATLOAD Patch Load	R[m] Address	R[n]<31:16> Patch Step	R[n]<15:8,7:0> <Width:Height>	R[k] Loaded Value	EVATCLD R[m], R[n], R[k] example: EVATCLD R[0], R[1], R[2]

multiplying corresponding vector entries and summing the result. Figure 6 shows both the dot product pseudocode and the operation of the accelerator.

$$result = \sum_{i=0}^{k-1} A_i * B_i \quad (1)$$

EVA supports the dot product with an the DOTPROD instruction seen in Table 1. The first operand F[m] indicates the first register in a sequence of 16 registers that will be used as the vector input. For example, F[0] sets floating point registers 0 through 15 as the input to the dot product unit. F[n] is the start index for the second set of sixteen floating point registers to be used as input. F[k] is the register to store the dot product's scalar result. The dot product example in Table 1 would result in registers 0 through 15 being the first vector, registers 16 to 31 being the second vector and the scalar result being placed in register 32.

In general, for the EVA accelerator instructions, the vector input sizes have been fixed to a length specified for the instruction in Table 1. If the output register overlaps with the input register, the output will overwrite the value in the input register upon instruction completion but the computation will be on the input. In the event of an floating exception, the floating point exception flag is set, and it is handled with the instruction in writeback.

4.2.2 Tree Accelerator

A commonly used data structure in the classification phase of computer vision is the tree. Trees are used in classification algorithms such as binary decision trees, Adaboost, and k-Nearest Neighbor Classification. They are used to classify feature vectors based on the feature vectors entries. Typically, tree data structures in computer vision are computed offline and are accessed but never modified by vision applications. It has been shown that collections of small trees can be used to produce high quality classification results [10, 11]. Based on this application behavior, we have designed the tree compare accelerator to accommodate a binary tree of depth three.

The tree compare accelerator is based on decision trees. Each node of a decision tree utilizes a value, called a *split*, and compares it to a specific feature entry to determine which child node should be used during the classification phase. The leaf node determines

a.
```
Original Code (450 Cycles):
dotProduct(A,B){
  sum = 0;
  for(i =0; i < 16; i++){
    sum += A[i] * B[i];}
  return sum;}
New Code (126 Cycles):
dotProduct(A,B){
  LOAD(A);
  LOAD(B);
  sum =DOTPROD(A+i,B+i);
  return sum;}
```

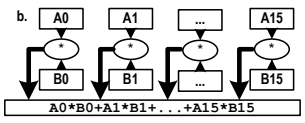
b.


Figure 6: **Dot Product Accelerator Overview** The figure shows the operation of the dot production unit. (a) shows the pseudocode of the dot product operation. (b) shows the operation it performs. The unit performs a dot product operation which is multiplying all the entries of two vectors and adding the results. The add component is done using a tree to add each adjacent result until the final result is computed.

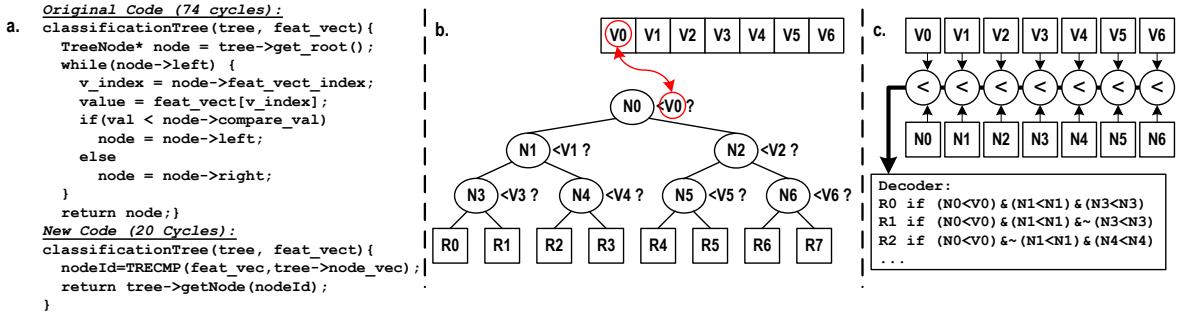


Figure 7: **Tree Compare Accelerator Overview** The figure shows the operation of the tree compare accelerator. (a) shows the typical tree comparison pseudocode used by vision algorithms, in particular classifiers. (b) shows how this looks relative to the tree. (c) shows how the accelerator unrolls the compares and performs them all at once and then decodes the binary bit vector to output the proper index.

the classification of the feature vector. Figure 7a shows the code for a tree compare operation of this type. We assume the feature vector has been compressed to only entries used by the tree. This compressed vector is passed into the tree comparison. The first entry of the vector is compared against the split value at the root node. Based on the result of the comparison, the left or right child node is then evaluated. The applications we considered only require less-than comparison. The feature vector index to compare to is stored with the tree node data structure along with the value to be compared. This traversal will continue until a leaf node is reached.

Figure 7b shows how the compressed feature vector and nodes comparisons take place relative to the tree. In this case the seven entries are placed in a vector based on which node they are to be compared with. The feature vector has the entries arranged such that the first entry is for the comparison to Node 0, the second entry is for comparison to Node 1, and so on. This can be achieved by taking the indices from the nodes and using them to put the entries in correct order before passing them to the tree compare operation. The node comparison values are also placed in an array in node order. The set of indices, the node values, and their order are computed offline.

Figure 7c shows how EVA computes a tree compare with the arrays. Values in the feature vector subset are compared in parallel with the tree nodes. The result is an 7-bit vector which is decoded to produce the index of the leaf node result. The leaf nodes are numbered left to right. The executing application uses the produced index to read the result value from an array which is computed offline and is part of the user program.

The instruction for the tree compare can be seen in Table 1. The first entry F[m] is the first register in a set of seven that will be used as the first index. For example, F[0] sets floating point registers 0 through 6 as the feature vector input to the tree compare accelerator. F[n] is the start index for the second set of seven floating point registers to be used as the tree values. R[k] is the register to store the result. For the tree compare example in Table 1, registers 0 through 7 contain the feature vector, registers 8 to 14 contain the tree vector and the leaf node index result is placed in register 0.

Since a binary tree can be decomposed into subtrees [16], the tree compare can be used for trees of arbitrary size by using the output value as an index to determine the next subtree to load.

4.2.3 Max Compare

Computing the maximum of a set of numbers is a common operation within mobile vision applications. This operation can be used for performing dilation filtering on an image in preprocessing, or in finding the largest histogram value in a feature extraction algorithm. It is commonly used in a function called non-maxima suppression to find the best scale/size or location of an object. Non-maxima suppression locates the maximal response within a region or set. This is a key operation in localizing features, objects, and responses. Thus, EVA provides a maximum operation to speed up this common computation.

The operation is utilized through a new instruction in the ISA named Max Compare which can be seen in Table 1. The max compare operates on small vectors with 8 or less entries and returns the index of the maximum value within the vector. The only input, F[m], is the first register in a set of eight that will be used as the vector. For example, F[0] sets floating point registers 0 through 7 as the input to the max compare accelerator. R[k] is the register to store the result. Table 1's maximum compare example would result in registers 0 through 7 being the vector and the index of maximum value in the vector being placed in register 0.

4.2.4 Monopoly Compare

A common operation that takes place in vision applications is the comparison of a single number to a large vector to determine if the scalar is smaller or larger than all the numbers in the vector. This operation is used in feature extraction to compare a single pixel value to its neighbors in an image or to find the corners in an image [28] [36]. This operation is quite frequent, and it is often a gating operation to performing more computation [15]. It can also be utilized during feature classification to track the top-N values. Thus, it can be used throughout the entire computer vision software pipeline. Given the potential benefits of speeding up this operation, EVA has an accelerator to support this vector reduction operation.

The monopoly compare accelerator can be seen in Figure 8 along with pseudocode of its operation. It supports both less-than and greater-than compares based on a bit in the opcode. The basic instruction for the monopoly compare can be seen in Table 1. The first entry F[m] is the value that will be compared to the vector. F[n] is the start index for the set of sixteen floating point registers to be compared against. R[k] is the register to store the binary results as a single word. The example for the monopoly compare in Table 1 would result in the value in register 0 being compared using less-than logic to the values in registers 16 to 31. The result would be placed in register 0.

4.3 Tile Memory Architecture

Many vision algorithms have been shown to have 2D spatial locality, in particular the feature extraction algorithms [40]. This characteristic is due primarily to the input of computer vision algorithms being images that are systematically scanned using small 2D windows. However, most image data is stored in raster scan order which causes vertically adjacent pixels to be stored at addresses that are far apart. Software solutions have been proposed to reorder the memory layout by Yang et al. [40]. Unfortunately, these solutions place a burden on the developer and can be done more energy efficiently in hardware [14]. The extra hardware costs of a entirely new memory controller may not be acceptable in mobile designs. Thus, we adopt a novel 2D prefetcher that warms up the cache when

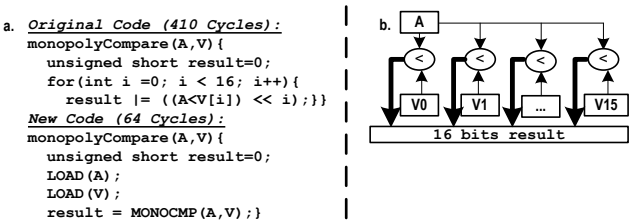


Figure 8: **Monopoly Compare Accelerator Overview** The figure shows the operation of the monopoly compare accelerator. (a) shows the pseudocode for the operation. (b) shows the operation itself. The accelerator compares a single value to the entries in a vector. The results are combined into a single 16-bit result.

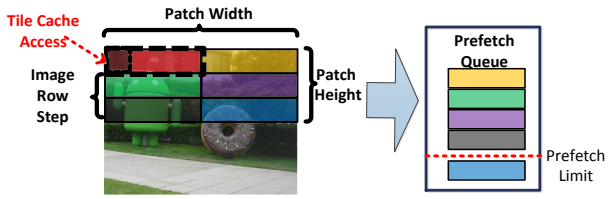


Figure 9: **Tile Cache Overview** The figure shows how the tile cache handles image data. In this example the patch size is 2 x 3 cache blocks. Each different colored segment is a cache block. When the upper left (red) cache block is accessed using a tile cache load the other cache blocks are added to the prefetch queue. The prefetch queue has a limit of 4 outstanding request thus the last cache block (blue) must wait until before the prefetch can be issued.

the application indicates it is touching 2D data. When the cache is operating with this prefetcher, we refer to this as the tile cache. Unlike other prefetchers, such as a stride prefetcher, that predict the access pattern, the tile cache receives the correct step amount from the application.

The tile cache generates prefetches of cache blocks in both the *x* direction and *y* direction in the image whenever a special load instruction is encountered. The amount of blocks to fetch in each direction, patch width and patch height, are passed to the tile cache through a single register along with the step or stride between image rows in a second register as seen in Table 1. This allows the tile cache to prefetch the entire patch that will be worked on. In the example in Table 1, R[1] would contain an address to load. R[2] would contain the patch step in the upper 16 bits. The lower 16 bits would contain patch width in bits 8 to 15 and the patch height in bits 0 to 7. The width is number of 64 bytes chunks in the tile. The height is the number of steps in the tile. This information is provided directly to the prefetcher. The result would be placed in R[0].

The prefetcher attempts to prefetch all the cache blocks in a patch. However, the hardware has a limit on the number of outstanding prefetch requests to avoid overburdening the memory system. We empirically found a maximum prefetch count of four outstanding 64-byte cache lines to be sufficient. Once a slot opens up for prefetching, the line that has been requested the most will be issued first.

Figure 9 shows an example with the tile cache. The pixel values are stored one after the other, and pixels in a tile can span on multiple cache lines (shown in different colors in the figure). The cache will fetch the required data while the tile cache will generate requests for the rest of the tile. The cost of this mechanism is minimal, it requires minor changes to state machine of the prefetcher and the register operands can be general purpose registers. This gives the benefits of 2D locality without modifying the rest of the memory system.

4.4 Heterogenous Chip Architecture

As indicated in Section 3.2, diverse parallelism is a characteristic of many vision workloads. In particular, there are points in many vision algorithms where threads coordinate their efforts and share their results with other threads [15]. Furthermore, some components of vision algorithms do not benefit from thread-level parallelism. For example, when the amount of data being processed is small the coordination cost may hinder the overall performance. While utilizing more cores can ensure support for thread-level parallelism, the constraints of mobile systems require low energy usage to preserve battery life. Thus, a balance must be struck. EVA provides heterogenous cores to deal with this situation. In particular, the EVA architecture is separated into two sets of cores. Each EVA system has at least one high performance core called a coordinating core (CC) and one or more lower performance cores called supporting cores (SC). All the cores contain the EVA accelerators to improve their energy efficiency and performance on mobile vision workloads. The EVA architecture can support any processor interconnect; however, for the remainder of this work we assume a bus interconnect. All the cores share an L2 cache while they each have a private L1 cache and tile cache support.

Figure 5 shows the relationship between the coordinating and supporting cores performance capabilities. In particular, the coordinating core has a wider pipeline than the supporting core. The coordinating core also has more integer computation units, an L1 data cache with double the associativity and a wider writeback stage when compared to a supporting core. They both have the same floating point/SIMD engine design and EVA units. Furthermore, the supporting cores focus more on conserving energy than improving performance.

In our research we found that supporting threads take approximately 20% to 40% less time to complete. Based on this information, the supporting cores in EVA can be as much as forty percent less powerful as the coordinating cores on vision workloads with minimal impact on the overall execution time.

Given these two types of cores, a key design question is what is the proper number of each type. The answer to this design decision lies in the demands of the target applications, combined with the area and cost constraints of the target market. We shall examine this key design decision in the experiments section. We investigate a set of similar-sized configurations and evaluate their performance and energy demands. These configurations can be seen in Figure 10.

5. Experimental Setup

5.1 EVA Model

We simulated our system using the gem5 simulator [9] in full system mode. The simulated system ran Ubuntu for ARM Linux with kernel version 3.3. We utilized Linux because it is a common

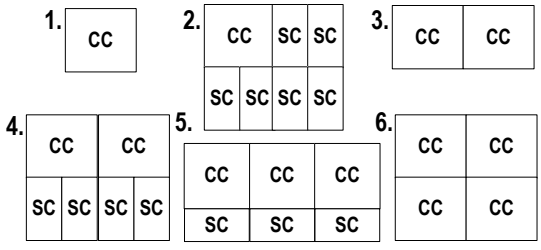


Figure 10: **EVA Coordinating (CC) and Supporting (SC) Cores Configurations With Area Constraint** The figure shows the possible EVA configurations given the maximum area of a four coordinating core without the EVA accelerators. Configurations (2),(4) and (5) have roughly the same area as a 4 coordinating cores without EVA features. Configuration (6) is slightly larger due to EVA features.

mobile operating system used in Android and Ubuntu for ARM. We utilized the Ubuntu image for compatibility with MEVBench. The Ubuntu for ARM was stripped down to give minimal services. We modified the gem5 ARM model ISA to support the EVA accelerator instructions. We modified the memory system to support the tile cache functionality through a special load instruction. The performance parameters of the gem5 model for both the coordinating and supporting cores are listed in Table 2. The mobile baseline is a coordinating core without EVA accelerators and mobile GPU is a SGX 54x series.

Table 2: **EVA Configuration**

Feature	Configuration
Core Clocks:	1 GHz
Coordinating Core:	32 bit RISC out-of-order, 4-way superscalar
Coordinating Core Pipeline:	8-Stage
Coordinating FUs:	4 integer units, 1 floating point units, 1 SIMD unit, 1 set of EVA accelerators
Vector Registers:	64 32bit Single Precision Registers
Coordinating Core L1 Caches:	32k 4-way assoc. instr. and data (2ns)
Supporting Core:	32 bit RISC out-of-order
Supporting Core Pipeline:	8-Stage, 2-way superscalar
Supporting Core FUs:	2 integer units, 1 floating point units, 1 SIMD unit, 1 set of EVA accelerators
Vector Registers:	64 32bit Single Precision Registers
Supporting Core L1 Cache:	32k 2-way assoc. instr. and data (1ns)
L2 Cache:	1MB unified non-inclusive (12ns)
Cache Coherency:	MOESI
Processor Interconnect:	128-bit Bus@ 1GHz with fast snoop unit
System Memory:	2GB LPDDR2
Instruction Set:	ARM-v7
Technology Node:	45nm

The EVA dot product accelerator is based on the efficient floating point unit designed by Galal and Horowitz [18]. The unit is pipelined and provides its result after 7 cycles. The EVA monopoly compare, tree compare and max compare are based on the work of Kim and Yoo [25]. The compare based units are pipelined and provide their results after a 5 cycle delay. The area for each functional unit can be seen in Table 3 along with core area estimates. The base estimates for the area of the cores are based on information from [8, 26] and [6]. We estimated the energy of the base cores using McPAT [27] along with energy models based on the accelerator designs.

Table 3: **Area estimates for the EVA Cores** These estimates assume a 45 nm silicon process.

Module	depth	latency	Area (mm^2)
Monopoly Compare	6	5 cycles	0.0489
Tree Compare	6	5 cycles	0.0215
Max Compare	6	5 cycles	0.0244
Dot Product	8	7 cycles	0.3290
Total for accelerators per core			0.4240
Coordinating Core			7.1200
Supporting Core			1.5839
Baseline Mobile Core w/SIMD + Embedded GPU			15.400

6. Experiments

6.1 Benchmarks

We utilized the MEVBench mobile vision benchmark suite [15] for our evaluation of the EVA system. We modified the benchmarks to insert the EVA operations where appropriate by finding loops with acceleration opportunities and inserting the new instructions into the code with inline assembly. We used the Code Sourcery ARM cross compiler suite version 4.6.1 [29] to generate static executables. We limited the compiler optimizations to -O1 for program correctness; however, we did enable ARM Neon instructions and their usage by the compiler with auto vectorization. In our simulations, the coordinator core was running the coordinating thread of the benchmark.

6.2 Single Core Results

Figure 11 shows the speedup gained through utilization of the EVA features while running benchmarks compared to a coordinating core without the EVA features. The tile cache benefits are primarily seen in the feature extraction benchmarks (e.g., HoG, SIFT, and SURF). These benchmarks all access image data and benefit from the 2D data locality provided by our design. The tile cache had little impact on benchmarks without 2D locality such as, SVM and BOOST. There is a small amount of improvement with FACEDETECT as well. Overall, the tile cache provides moderate performance improvements (2% to 40%) for programs that exhibit 2D localities. Since use of the tile cache is software controlled, its use can be avoided for programs with out 2D locality, thereby preventing tile cache prefetcher from negatively impacting program performance by saturating the memory system.

Our accelerators provide a speedup in all the benchmarks except FACEDETECT and AUGREAL. The benefits of the EVA accelerators was not seen fully due to limited use of the accelerators in these benchmarks. Overall the complete EVA design provides an average speedup of 1.8x. It peaks near 4x for SVM due to its very heavy use of the dot product operation.

Figure 12 shows the normalized energy usage of EVA while running the benchmarks. The energy is calculated using the simulation statistics to generate an input model for McPAT [27]. Mcpat models the common microarchitectural components such as caches and ALUs at our given technology node (45nm) based on runtime activity. A framework similar to CACTI [39] is used to model memory components in this framework. We combine these results with models for the energy consumption for each custom accelerator to compute the total energy. The graph in Figure 12 plots the energy of three EVA designs, normalized to the energy of a coordinating core without EVA enhancements. It is interesting to note that, in some cases, the tile cache provides a small amount of energy savings. This is due to not having the wait for the data to be returned before resuming execution, thus reducing idleness of the accelerators. The tile cache is designed to allow for memory accesses in both a 1D and 2D fashion. It is primarily beneficial in the feature extraction

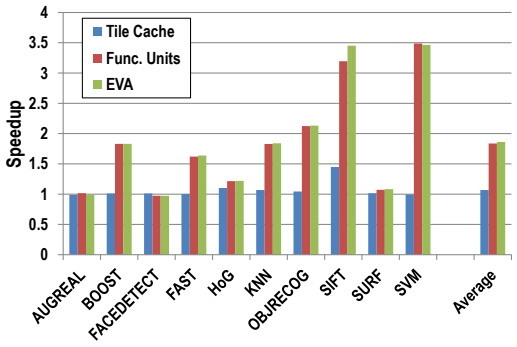


Figure 11: **EVA Single Coordinating Core Speedup** The figure shows the speedup of the an EVA single coordinating core versus a single coordinating core speedup that does not have EVA. The plot shows how both the tile cache and EVA accelerators both contribute to the performance increase.

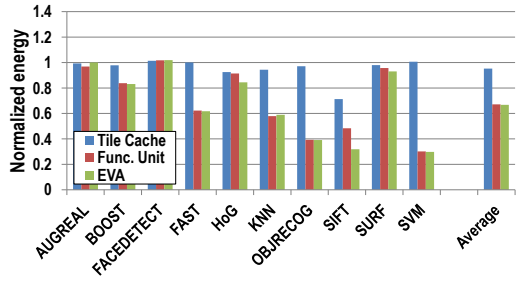


Figure 12: **EVA Single Coordinating Core Normalized Energy** The figure shows the energy per image frame for a single EVA enhanced coordinating core normalized to a coordinating core without the EVA enhancements. The energy savings using EVA in general are quite good. In general the savings come from the use of the EVA accelerators that are specifically designed for the given operation, and the decrease in the committed instructions which decreases the work of the pipeline as a whole.

benchmarks as a result. The accelerators show a decrease in energy in all the benchmarks except FACEDETECT. This is due to the same effects that caused the slight slowdown. In the case of SVM, the use of the efficient vector reduction dot product accelerator provides a large amount of energy savings. In terms of energy, EVA provides savings of close to 30% and peaks at 3x decrease in energy while also providing an average speedup. While the majority of energy savings are due to the more efficient hardware utilization, modest savings are also thanks to the tile cache.

Figure 13 shows the usage of the accelerators. The figure demonstrates how the various benchmarks utilize different accelerators. Overall, the most exercised accelerators are the dot product accelerator followed by the monopoly compare accelerator. The tree compare accelerator is heavily utilized in the tree-based benchmarks.

Figure 14 shows the accuracy of the tile prefetcher, and shows its improved performance compared to a traditional stride prefetcher [12]. The tile cache performs at least as well as the stride prefetcher in all benchmarks. The tile cache outperforms the stride prefetcher in the feature extraction algorithms due to the accessing of patches to build feature descriptors for feature points. The locations of the feature points are random and thus the stride prefetcher is unable to detect a consistent stride. This condition is also present when performing filtering operations with small 2D kernels.

Overall the single-core EVA provides an average speedup of approximately 1.8x while reducing the energy usage of a core by over 30%.

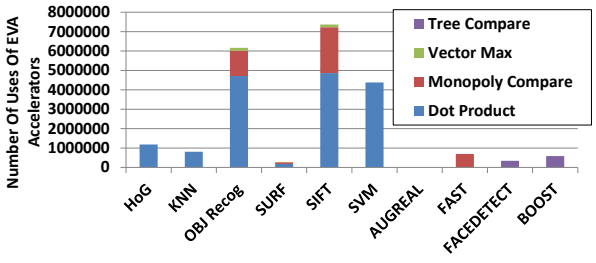


Figure 13: **EVA Single Coordinating Core Accelerator Usage** The figure shows the usage of the EVA single coordinating core accelerators. The plot shows how often the a given accelerator contributes to the performance increase.

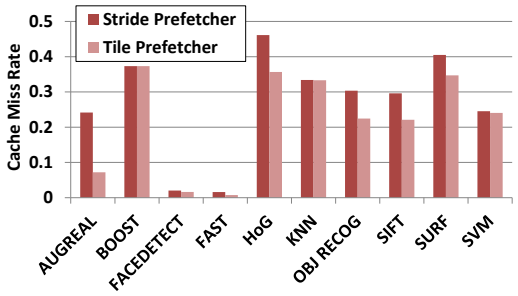


Figure 14: **Comparison of Tile to Stride Prefetcher** The figure shows the cache miss rates for accessing image data using the tile cache versus a cache with a stride prefetcher. The tile cache performs at least as well as the stride prefetcher in all benchmarks. It outperforms the stride prefetcher in cases where there are small image patches or random patch access patterns. Random patch access patterns typically occur during feature descriptor building phase of feature extraction.

6.3 Multicore Results

We ran the configurations shown in Figure 10. We show the average speed up for the benchmark suite for each configuration in Figure 15. We constrained our configurations to approximately the area of an embedded quad-core processor utilizing four coordinating cores without EVA features. This represents a modern class of embedded machines and considers its area constraints. The EVA accelerators were used on all the benchmarks in the figure. The tile cache, which can be configured in software, was only active on benchmarks that showed a performance benefit during single-threaded execution. Overall, the use of more cores is beneficial although the performance is limited by the last thread to complete it's workload. In some cases, the workload of a supporting core causes slowdown for the entire system. The overall performance shows that having a single coordinator and six supporting core seems to be the best performing configuration.

Figure 15 shows the average energy usage to process a 352x288 frame for each configuration for the benchmark suite. The energy is normalized to a single coordinating core without the EVA enhancements. The energy usage drops as the number of cores increases due to the reduced runtime as thread-level parallelism is exploited. Once a core has completed their portion of work, they sit idle. The supporting cores use less energy making the idle time less costly.

Figure 16 shows the scalability of the EVA coordinating cores given a fixed power budget of 5 Watts as the voltage is scaled

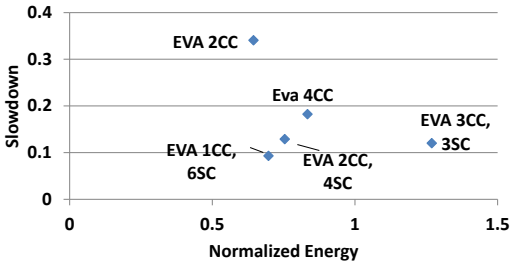


Figure 15: **Multicore Configuration performance of EVA** This figure shows the multicore performance of EVA. The x axis is energy is normalized to a single coordinating core without EVA enhancements. The y axis is 1/speedup such that closer to the origin is better. It shows that utilizing 1 coordinator core and 6 supporting cores is pareto optimal given the quad core area constraint. In other configurations, the coordinating cores waste energy waiting. If the supporting cores wait, they use less energy than a coordinating core.

down. The plot shows that for a 5W budget, the number of cores with highest performance is 12 cores with a peak speedup of 3.2x on the MEVBench benchmarking suite with each core running at approximately 800MHz clock. The 24 core configuration is above the power budget but with a slight increase in the power budget would also be a good platform for vision applications. Thus with a fixed power budget, the performance versus energy tradeoff can be managed to provide a balanced solution. For example, if energy is more important than speed then 14 cores would be a better solution, given this power budget.

6.4 Comparisons To Other Approaches

Figure 17 compares the performance of EVA against other solutions. For each experiment, the platform is running the SIFT algorithm optimized for that particular platform. The graph plots the performance of Qualcomm Snapdragon S2 with Adreno [13, 24], Intel i7, GTX 260, EFFEX [14], and EVA on a pareto chart, which indicates the energy demands and performance capabilities of each design solution. The EVA solution is closest to the origin making it a pareto optimal solution, i.e., there are no other analyzed solutions with better energy (at this performance), or better performance (at this energy). EVA's accelerators increase the computation performance and decrease the energy usage. The use of heterogeneous multicore takes advantage of thread-level parallelism in an efficient manner. While GPUs are efficient cores for graphics, the comparison of the mobile GPU with the desktop GPGPU shows the large gap in performance in utilizing GPUs on mobile devices [13].

7. Related Work

Companies such as Qualcomm and Texas Instruments (TI) have released closed-source computer vision libraries that are optimized for better performance on their SoCs [34, 38]. These libraries typically use computational resources not available to end developers such as small DSPs and image filtering accelerators. Thus, newer vision algorithms may not be supported immediately or the API may not match the developer's needs. With EVA, the user software has access to the accelerators, thus new or modified algorithms can be readily ported to the EVA platform.

Yao et al. [41] propose the use of specific hardware for the SIFT algorithm. Javaid et al. [23] proposed optimized architectures for video encoding. Others have proposed similar hardware for specific computer vision algorithms such as BOOST [21] or applications such as robotics [42]. Such solutions are capable but inflexible. For example, although ORB [37] was developed based on FAST [36] the algorithms are different enough that any hardware system built for the FAST algorithm would be need to be completely redesigned. EVA's programmability provides flexibility for the developer to evolve the platform's vision software capabilities, while still maintaining an efficient solution.

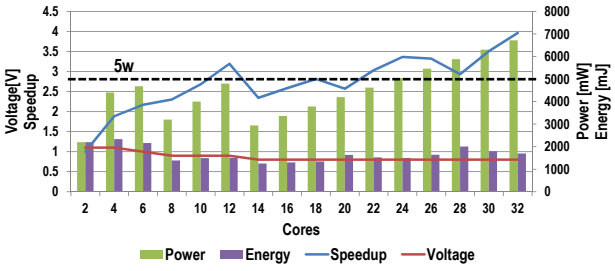


Figure 16: **EVA Coordinator Cores Scaling** The figure shows the energy and speed up for a fixed power budget of 5W as voltage is scaled and the number of cores is increased. This analysis assumes an interconnect that can scale such as for a NoC. 12 cores seems to give a good performance with a high speed up and low energy.

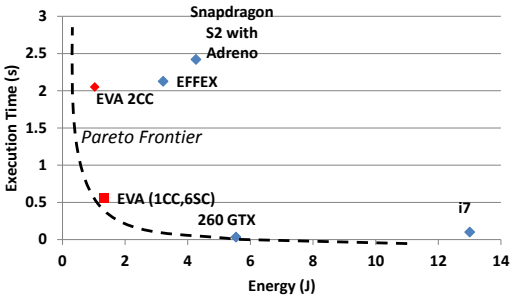


Figure 17: **Comparison Of EVA with Other Solutions** This figure shows a comparison of EVA against other solutions based on [13, 24, 30]. It demonstrates that EVA provides a highly effective energy-performance tradeoff for the SIFT algorithm.

Clemons et al. [14] developed EFFEX for the feature extraction phase of mobile vision. EVA provides capabilities for the entire mobile vision pipeline as opposed to the single phase focus of EFFEX. EVA avoids using a special memory controller to exploit 2D locality and instead utilizes a lower cost tile cache prefetcher. EVA utilizes less energy and has less latency than EFFEX.

Raghavan et al. has proposed powering up cores for fractions of a second and then powering them down to cool to handle high computational loads such as mobile vision [35]. This technique is very effective for applications that have pauses between computations, however, many vision applications such as person tracking and augmented reality require continuous computation. EVA is able to maintain high performance in mobile vision for extended periods.

ARM has released the big.LITTLE platform for energy efficient heterogeneous computing [6]. This is a general purpose system with two sets of multi-core processors of varying performance capability and energy usage. They are comparable in how they deal with diversely parallel workloads, however the big.LITTLE cores have no special accelerators for vision. EVA contains accelerators for mobile vision and a tile cache to increase performance.

GPGPUs have been used to increase the performance of computer vision on the desktop with impressive speedups by Prisacariu [33], Fung [17] and OpenCV [32] among others. However, mobile GPUs have been shown to be inefficient at mobile vision due to the energy and cost constraints of mobile systems [24]. Figure 17 shows that EVA is a pareto optimal solution when compared to GPGPUs and mobile GPUs. There are other solutions within the SoC space, such as the DSP. EVA can be utilized to improve the performance of these solutions as well. EVA is orthogonal to these solutions as it can share the processing with other devices allowing even more heterogeneity in the design.

Chen and Baer proposed the stride prefetcher to accommodate predictable data loads [12]. When accessing data within a small 2D window, the stride prefetcher has issues stabilizing due to the access pattern in the 2D window having a stride moving in the x direction of one pixel and a stride moving in the y direction of the image step or vice versa. Algorithms such as BRIEF can have random steps within a patch causing further issue with stride prefetchers. The EVA tile cache has no stabilization issues because it receives information from the program and is designed for multiple strides.

8. Conclusion and Future Work

Mobile vision is a large and complex application space that has an insatiable need for computational power. Some vision applications can utilize large amounts of parallelism while others benefit from serial performance. There are some vision algorithms that have 2D data locality while others only contain mostly 1D locality. Most vision algorithms contain vector reduction operations. As such, there is a need for a system that can take advantage of all these traits to increase performance in the mobile space.

In this work we presented EVA, an efficient architecture for mobile vision applications. EVA is a heterogeneous multicore with

custom accelerators and a 2D locality caching system to increase the performance of mobile vision systems. We have shown that EVA's vector reduction accelerators and ability to exploit 2D locality improves both the energy and performance when executing mobile vision workloads.

We have explored performance-optimal configurations of EVA given a mobile quadcore area constraint. The single coordinating core with 6 supporting core is the best performing design under this constraint due to having the lowest energy and execution time on the benchmarks. Additionally, the energy-optimal number of cores given a fixed power budget has also been shown. For a 5W power constraint, we found that the most effective design was a 12-core configuration based running at 680 MHz and 0.9 volts.

There are several ways in which we intend on extending EVA. The first is increasing the number of custom accelerators to include more functionality. We would also like to study the possible usage of EVA in applications outside of computer vision such as server-level recommendation systems. Finally, we want to investigate the application of EVA optimizations to other mobile compute platforms, such as mobile GPUs and DSPs, to increase their effectiveness.

9. Acknowledgment

This work is supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA.

References

[1] A. Adams, E.-V. Talvala, S. H. Park, D. E. Jacobs, B. Ajdin, N. Gelfand, J. Dolson, D. Vaquero, J. Baek, M. Tico, H. P. A. Lensch, W. Matusik, K. Pulli, M. Horowitz, and M. Levoy. The Frankencamera: an experimental platform for computational photography. In *SIGGRAPH*, 2010.

[2] C. Albanesius. Google 'Project Glass' Replaces the Smartphone With Glasses. http://www.pcmag.com/article2/0,2817,2402613,00.asp, 2012.

[3] Apple. Apple. http://www.apple.com/, 2011.

[4] ARM. 2GHz Capable Cortex-A9 Dual Core Processor Implementation. http://www.arm.com/files/downloads/Osprey_Analyst_Presentation_v2a.pdf, 2011.

[5] ARM. ARM NEON. http://www.arm.com/products/processors/ technologies/neon.php, 2011.

[6] ARM. ARM big.Little. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf, 2012.

[7] R. Baldwin. Ikea's Augmented Reality Catalog Will Let You Peek Inside Furniture. http://www.wired.com/gadgetlab/2012/07/, 2012.

[8] Berkeley Design Technology Inc. ARM Announces 2GHz Dual Core Cortex A9. http://www.bdti.com/InsideDSP/2009/09/23/Arm, 2011.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[10] D. G. R. Bradski and A. Kaehler. *Learning OpenCV*. O'Reilly Media, Inc., 2008.

[11] L. Breiman. Random forests. *Mach. Learn.*, 45(1), Oct. 2001.

[12] T.-f. Chen and J.-l. Baer. Effective Hardware-based Data Prefetching for High-performance Processors. *IEEE Transactions on Computers*, 1995.

[13] K.-T. Cheng and Y.-C. Wang. Using mobile GPU for general-purpose computing: A case study of face recognition on smartphones. In *VLSI-DAT*, 2011.

[14] J. Clemons, A. Jones, R. Perricone, S. Savarese, and T. Austin. EFFEX: An embedded processor for computer vision-based feature extraction. In *DAC*, 2011.

[15] J. Clemons, H. Zhu, S. Savarese, and T. Austin. MEVBench: A mobile computer vision benchmarking suite. In *IISWC*, 2011.

[16] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[17] J. Fung and S. Mann. OpenVIDIA: Parallel GPU computer vision. In *Proceedings of the 13th annual ACM international conference on Multimedia*, MULTIMEDIA '05, New York, NY, USA, 2005. ACM.

[18] S. Galal and M. Horowitz. Energy-efficient floating-point unit design. *IEEE Trans. Comput.*, 60(7):913–922, July 2011.

[19] Google. Nexus Galaxy Tech Specs. http://www.google.com/nexus/#/tech-specs, 2011.

[20] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2003.

[21] M. Ibarra-Manzano and D. Almanza-Ojeda. Design and Optimization of Real-Time Boosting for Image Interpretation Based on FPGA Architecture. In *CERMA*, 2011.

[22] Intel. Intel Core i7-4770K Processor. http://ark.intel.com/products/75123/, 2013.

[23] H. Javaid, M. Shafique, S. Parameswaran, and J. Henkel. Low-power adaptive pipelined mpsocs for multimedia: an h.264 video encoder case study. In *DAC*, 2011.

[24] G.-R. Kayombya. SIFT feature extraction on a Smartphone GPU using OpenGL ES2.0. Master's thesis, Massachusetts Institute of Technology, 2010.

[25] J.-Y. Kim and H.-J. Yoo. Bitwise competition logic for compact digital comparator. In *Proceedings of the IEEE Asian Solid States Circuits Conference*, 2007.

[26] J. Koppanalil, G. Yeung, D. O'Driscoll, S. Householder, and C. Hawkins. A 1.6 GHz dual-core ARM Cortex A9 implementation on a low power high-K metal gate 32nm process. In *VLSI-DAT*, 2011.

[27] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[28] D. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 2004.

[29] Mentor Graphics. Sourcery CodeBench. http://www.mentor.com/embedded-software/codesourcery, 2011.

[30] M. Murphy, K. Keutzer, and H. Wang. Image feature extraction for mobile processors. In *IISWC*, oct. 2009.

[31] NVIDIA. Variable SMP A Multi Core CPU Architecture for Low Power and High Performance. http://www.nvidia.com/object/white-papers.html.

[32] OpenCV.org. OpenCV Platforms: CUDA, November 2012. http://opencv.org/platforms/cuda.html.

[33] V. Prisacariu and I. Reid. fastHOG - A real-time GPU implementation of HOG. Technical Report 2310/09, Department of Engineering Science, Oxford University, 2009.

[34] Qualcomm. Fastcv. https://developer.qualcomm.com/mobile-development/mobile-technologies/ computer-vision-fastcv.

[35] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Computational sprinting. In *HCPA*, 2012.

[36] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *ECCV*, May 2006.

[37] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. ORB: An Efficient Alternative to SIFT or SURF. In *ICCV*, 2011.

[38] Texas Intruments. VLIB 2.0: Video Analytics And Vision Library, December 2008. http://www.ti.com/lit/ml/sprt502a/sprt502a.pdf.

[39] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *ISCA*, 2008.

[40] X. Yang and K. Cheng. Accelerating surf detector on mobile devices. In *ACM Multimedia Conference*, 2012.

[41] L. Yao, H. Feng, Y. Zhu, Z. Jiang, D. Zhao, and W. Feng. An architecture of optimised sift feature detection for an fpga implementation of an image matcher. In *FPT*, 2009.

[42] J. Yudi Mori, D. Muñoz Arboleda, J. Arias Garcia, C. Llanos Quintero, and J. Motta. Fpga-based image processing for omnidirectional vision on mobile robots. In *SBCCI*, 2011.