Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing

Wajahat Qadeer, Rehan Hameed, Ofer Shacham,
Preethi Venkatesan, Christos Kozyrakis, Mark A. Horowitz
Stanford University, California
{wqadeer, rhameed, shacham, preethiv, kozyraki, horowitz}@stanford.edu

ABSTRACT

This paper focuses on the trade-off between flexibility and efficiency in specialized computing. We observe that specialized units achieve most of their efficiency gains by tuning data storage and compute structures and their connectivity to the data-flow and data-locality patterns in the kernels. Hence, by identifying key data-flow patterns used in a domain, we can create efficient engines that can be programmed and reused across a wide range of applications.

We present an example, the Convolution Engine (CE), specialized for the convolution-like data-flow that is common in computational photography, image processing, and video processing applications. CE achieves energy efficiency by capturing data reuse patterns, eliminating data transfer overheads, and enabling a large number of operations per memory access. We quantify the trade-offs in efficiency and flexibility and demonstrate that CE is within a factor of 2-3x of the energy and area efficiency of custom units optimized for a single kernel. CE improves energy and area efficiency by 8-15x over a SIMD engine for most applications.

Categories and Subject Descriptors

C.5.4 [**Computer Systems Implementation**]: VLSI Systems—*Customization, Heterogeneous CMP*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (Hybrid) Systems*

General Terms

Algorithms, Performance, Computational Photography

Keywords

Convolution, H.264, Demosaic, Specialized Computing, Energy Efficiency, Tensilica, Computational Photography

1. INTRODUCTION

The slowdown of voltage scaling has made all chips energy limited: the energy per transistor switch now scales slower than the number of transistors per chip. Paradoxically, we must use these additional transistors to reduce the number of transistors switched in each operation to improve energy efficiency. The primary way to achieve this goal is to create application specific accelerators to remove the overhead of predicting, fetching, decoding, scheduling, and committing instructions in a normal processor [7, 20, 31]. Accelerators provide as much as three orders of magnitude improvements in compute efficiency over general-purpose processors. Heterogeneous chips combining processors and accelerators already dominate mobile systems [4, 2] and are becoming increasingly common in server and desktop systems [17, 10]. Large specialized units perform hundreds of operations for each data and instruction fetch, reducing energy waste of programmable cores by two orders of magnitude [20]. Significant research is now focusing on automatic generation of specialized units from high-level descriptions or templates in order to reduce design costs [27, 13, 31, 19, 26].

This paper explores the energy cost of making a more general accelerator, one which can be user programmed. Current accelerators, whether designed manually or automatically generated, are typically optimized for a single kernel, and if configurable, are configured by experts in firmware. Clearly it would be better to create units that are specialized enough to reach close to ASIC compute efficiency, but retain some of the flexibility and reuse advantages of programmable cores.

An example of a programmable accelerator prevalent in embedded and desktop processors is the SIMD unit which targets data-parallel algorithms. However, SIMD units are still one to two orders of magnitude less efficient compared to algorithm-specific custom units [20]. This paper shows that it is possible to build more efficient programmable accelerators by exploiting the fact that *specialized units achieve most of their efficiency gains by tuning data storage structures to the data-flow and data-locality requirements of the kernel*. This tuning eliminates redundant data transfers and facilitates creation of closely coupled datapaths and storage structures allowing hundreds of low-energy operations to be performed for each instruction and data fetched. Hence, if we identify data-flow and data locality patterns that are common to a wide range of kernels within a domain, we can create specialized units that are highly energy efficient, but can be programmed and reused across a wide range of applications.

We concentrate on computational photography, image processing, and video processing applications that are popular on mobile systems. We find that a common motif is a convolution-like data flow: apply a function to a stencil of the data, then perform a reduction, then shift the stencil to include a small amount of new data, and repeat. Examples include demosaic, feature extraction and mapping in scale-invariant-feature-transform (SIFT), windowed histograms, median filtering, motion estimation for H.264 video processing and many more. In contrast to the current solu-

ISCA '13 Tel-Aviv, Israel

tions that create different accelerators for each of the applications [4, 2, 20], we describe the design of a flexible domain-specific *Convolution Engine (CE)* for all these applications. This means that our CE must handle differences in the size and dimensions of the stencil, the function applied on the stencil, the reduction function, and even the variance in shift across applications. We implement CE as an extension to a Tensilica processor and not as a stand alone accelerator, which means programmers can interact with it much like they use traditional SIMD extensions (e.g., SSE4 or NEON).

To better understand the context of this work, the next section introduces some image processing background, and the hardware currently used in this application domain. Section 3 then introduces the convolution abstraction and the five application kernels we target in this study. Section 4 describes the CE architecture focusing primarily on features that improve energy efficiency and/or allow for flexibility and reuse. It also presents how multiple CE engines aggregate into a larger engine if needed. We then compare this Convolution Engine to both general-purpose cores with SIMD extensions and highly customized solutions for individual kernels in terms of energy and area efficiency. Section 5 first introduces the evaluation methodology and Section 6 shows that the CE is within a factor of 2-3x of custom units and almost 10x better than the SIMD solution for most applications.

2. BACKGROUND AND RELATED WORK

Imaging and video systems are already deeply integrated into many of our devices. In addition to traditional cameras, cell phones, laptops and tablets all capture high-resolution images and high definition video is quickly becoming standard. These imaging systems push a large number of pixels through an image pipeline in real time and thus have very high computational requirements. Furthermore, compressing these individual frames into a video bit-stream requires complex video codecs such as H.264. Figure 1 shows a simplified imaging pipeline that processes still images and individual video frames, often implemented as an ASIC hardware accelerator to meet the tight time and energy budgets.

The field of computational photography [5, 25] employs additional digital image processing to enhance captured images using algorithms such as high dynamic range imaging [5, 16], digital image stabilization [23], flash no-flash imaging [24] and many more. Another class of applications augment reality by combining real-time vision algorithms with real time imaging systems. Most of these applications depend on feature detection and tracking algorithms such SIFT [22] and SURF [8] to find correspondences between multiple images or frames, perform object detection or other similar functions.

Interestingly, even though the range of applications is very broad, a large number of kernels in these applications "look" similar. These are kernels where the same computation is performed over and over on (overlapping) stencils within, and across frames. For example, demosaicing takes squares of *n* x *n* Bayer patterned pixels and interpolate the RGB values for each pixel. This is similar to the sum-of-absolute-differences (SAD) applied on *n* x *n* stencils used for motion estimation. We categorize this class of stencil operation as convolution-like. Unfortunately programmable platforms today do not handle convolution-like computation efficiently because their register files are not optimized for convolution. To address this issue, traditional camera and camcorder manufacturers typically use their own ASICs to implement camera pipelines such as Canon's DIGIC series [1], Nikon's Expeed processors [14] and Sony's Bionz processors[15]. Cellphone SoCs, including TI's OMAP [2], Nvidia's Tegra [4] and Qualcomm's Snapdragon [3] platform take a similar approach.

Some domains have been able to create user programmable application accelerators. GPUs are perhaps the most successful example. They are optimized for massively data parallel graphics applications and employ a large number of low control overhead cores, sacrificing single thread performance in favor of very high throughput. Large numbers of threads allow GPUs to efficiently hide memory latencies. By specializing for their workload characteristic, GPUs are able to get much higher throughput performance and lower energy than general purpose processors. Unfortunately, as we will show later, GPUs use more than 10x too much energy for most image processing tasks.

We explore creating a similar user programmable architecture, but geared for the image processing space. This architecture is optimized for convolutional data flow, and is described in Section 4. Before describing the architecture, the next section explains our notion of a generalized convolution, and the applications we will map onto our CE.

3. CONVOLUTION ABSTRACTION

Convolution is the fundamental building block of many scientific and image processing algorithms. Equation 1 and 2 provide the definition of standard discrete 1-D and 2-D convolutions. When dealing with images, *Img* is a function from image location to pixel value, while *f* is the filter applied to the image. Practical kernels reduce computation (at a small cost of accuracy) by making the filter size small, typically in the order of 3x3 to 8x8 for 2-D convolution.

$$(Img * f)[n] \stackrel{\text{def}}{=} \sum_{k=-\infty}^{\infty} Img[k] \cdot f[n-k] \quad (1)$$

$$(Img * f)[n,m] \stackrel{\text{def}}{=} \sum_{l=-\infty}^{\infty} \sum_{k=-\infty}^{\infty} Img[k] \cdot f[n-k, m-l] \quad (2)$$

We generalize the concept of convolution by identifying two components of the convolution: a *map* operation and a *reduce* operation. In Equation 1 and 2, the map operation is multiplication that is done on pairs of pixel and tap coefficient, and the reduce operation is the summation of all these pairs to a single value at location [n,m]. Replacing the map operation in Equation 2 from $x \cdot y$ to $|x - y|$ while leaving the reduce operation as summation, yields a sum of absolute numbers (SAD) function which is used for H264's motion estimation. Further replacing the reduce operation from $\sum$ to max will yield a max of absolute differences operation. Equation 3 generalizes the standard definition of convolution, to a programmable form. We refer to it as a convolution engine, where *f*, *Map* and *Reduce* ('*R*' in Equation 3) are the pseudo instructions, and c is the size of the convolution.

$$\begin{aligned}(Img \overset{CE}{*} f)[n,m] &\stackrel{\text{def}}{=} \\ & R_{|l|<c}\{R_{|k|<c}\{Map(Img[k], f[n-k, m-l])\}\}\end{aligned} \quad (3)$$

To increase the space of applications further, we extend the operations possible in the reduction stage to allow a non-commutative function, and provide a permutation network between the registers and the map function units. This allows us to perform more complex data combining in the *reduce* stage than true reductions. The following sections describe our test applications and how we map their kernels onto this generalized convolution framework.

3.1 Motion Estimation

Motion estimation is a key component of many video codecs including H.264. When the codec is implemented in software,

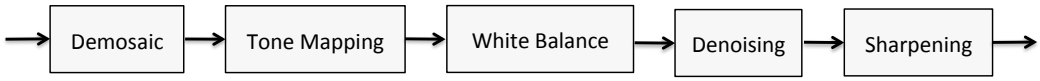

Figure 1: Typical (simplified) imaging pipeline.

motion estimation accounts for ∼90% of the execution time [11, 20]. The kernel operates on sub-blocks of a video frame, trying to find each sub-block's location in a previous and/or future reference frame of the video stream. In particular, in H.264, motion estimation is computed in two steps: IME and FME.

Integer Motion Estimation (IME): IME searches for an image-block's closest match from a reference image. It then computes a vector to represent the observed motion. The search is performed at each location within a two dimensional search window, using sum of absolute differences (SAD) as the cost function. IME operates on multiple scales with various blocks sizes from 4x4 to 16x16, though all of the larger block results can be derived from the 4x4 SAD results. Note how SAD fits quite naturally to a convolution engine abstraction: the *map* function is absolute difference and the *reduce* function is summation.

Fractional Motion Estimation: FME refines the initial match obtained at the IME step to a quarter-pixel resolution. FME first up-samples the block selected by IME, and then performs a slightly modified variant of the aforementioned SAD. Up-sampling also fits nicely to the convolution abstraction and actually includes two convolution operations: First the image block is up-sampled by two using a six-tap separable 2D filter. This part is purely convolution. The resulting image is up-sampled by another factor of two by interpolating adjacent pixels, which can be defined as a *map* operator (to generate the new pixels) with no *reduce*.

3.2 SIFT

Scale Invariant Feature Transform (SIFT) looks for distinctive features in an image [22]. Typical applications of SIFT use these features to find correspondences between images or video frames, performing object detection in scenes, etc. To ensure scale invariance, Gaussian blurring and down-sampling is performed on the image to create a pyramid of images at coarser and coarser scales. A Difference-of-Gaussian (DoG) pyramid is then created by computing the difference between every two adjacent image scales. Features of interest are then found by looking at the scale-space extrema in the DoG pyramid [22].

Even though finding scale-space extrema is a 3D stencil computation, we can convert the problem into a 2D stencil operation by interleaving rows from different images into a single buffer. The extrema operation is mapped to convolution using compare as a *map* operator and logical AND as the *reduce* operator.

3.3 Demosaic

Camera sensor output is typically a red, green, and blue (RGB) color mosaic laid out in Bayer pattern [9]. At each location, the two missing color values are then interpolated using the luminance and color values in surrounding cells. Because the color information is undersampled, the interpolation is tricky; any linear approach yields color fringes. We use an implementation of Demosaic that is based upon adaptive color plane interpolation (ACPI) [12], which computes image gradients and then uses a three-tap filter in the direction of smallest gradient. While this fits the generalize convolution flow, it requires a complex "reduction" tree to implement the gradient based selection. The data access pattern is also non-trivial since individual color values from the mosaic must be separated before performing interpolation.

$$Y_0 = x_0 * c_0 + x_1 * c_1 + x_2 * c_2 + x_3 * c_3 + ... + x_n * c_n$$
$$Y_1 = x_1 * c_0 + x_2 * c_1 + x_3 * c_2 + x_4 * c_3 + ... + x_{n+1} * c_n$$
$$Y_2 = x_2 * c_0 + x_3 * c_1 + x_4 * c_2 + x_5 * c_3 + ... + x_{n+2} * c_n$$
.....

Figure 2: We use the n-tap 1D convolution presented here to explain our SIMD implementation. For SIMD the equation is parallelized across outputs and executed one column at a time.

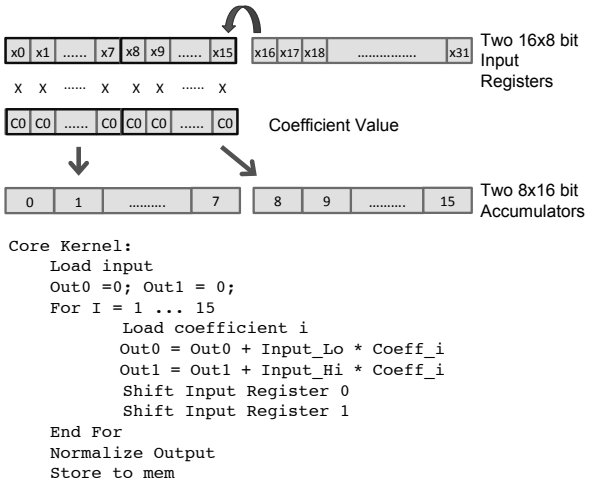

Figure 3: 1D Horizontal 16-tap convolution on a 128-bit SIMD machine, similar to optimized implementation described in [29]. 16 outputs are computed in parallel to maximize SIMD usage. Output is stored in two vector registers and two multiply-accumulate instruction are required at each step.

3.4 Mapping to Convolution Abstraction

Table 1 summarizes the kernels we use and how they map to the convolution abstraction. The table further describes how each algorithm is divided into the *map* and *reduce* operator and what is its data flow pattern such as 2D convolution or 1D vertical convolution. Although, two kernels could have identical *map* and *reduce* operators and data flow patterns, they may have differences in the way they handle the data. For example up-sampling in FME produces four times the data of its input image while Demosaic, also an interpolation algorithm, needs to separate out the different color channels from a single channel two-dimensional image before operating on the data. These requirements differentiate them from simple filtering operations and require additional support in hardware as we will see next.

4. CONVOLUTION ENGINE

Convolution operators are highly compute-intensive, particularly for large stencil sizes, and being data-parallel they lend themselves to vector processing. However, existing SIMD units are limited in

Table 1: Mapping kernels to convolution abstraction. Some kernels such as subtraction operate on single pixels and thus have no stencil size defined. We call these matrix operations. There is no reduce step for these operations.

	Map	Reduce	Stencil Sizes	Data Flow
IME SAD	Abs Diff	Add	4x4	2D Convolution
FME 1/2 Pixel Upsampling	Multiply	Add	6	1D Horizontal And Vertical Convolution
FME 1/4 Pixel Upsampling	Average	None	–	2D Matrix Operation
SIFT Gaussian Blur	Multiply	Add	9, 13, 15	1D Horizontal And Vertical Convolution
SIFT DoG	Subtract	None	–	2D Matrix Operation
SIFT Extrema	Compare	Logical AND	3	1D Horizontal And Vertical Convolution
Demosaic Interpolation	Multiply	Complex	3	1D Horizontal And Vertical Convolution

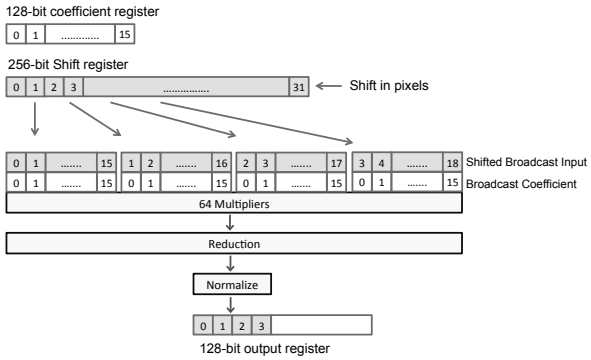

Figure 4: 1D Horizontal 16-tap convolution using a shifter register with shifted broadcast capability. Computes 4 output pixels per instruction.

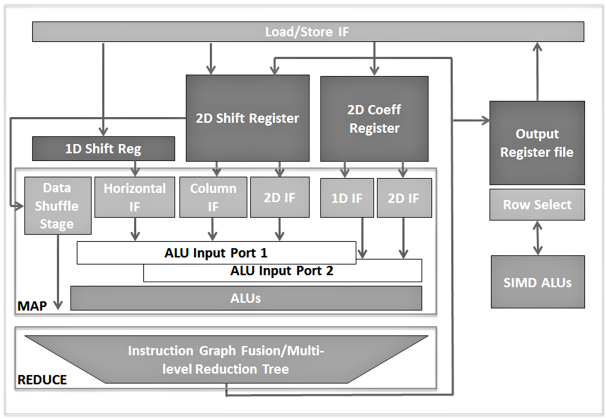

Figure 5: Block Diagram of Convolution Engine. The interface units (IF) connect the register files to the functional units and provide shifted broadcast to facilitate convolution. Data shuffle (DS) stage combined with Instruction Graph Fusion (IGF) stage form the Complex Graph Fusion Unit. IGF is integrated into the reduction stage for greater flexibility.

the extent to which they can exploit the inherent parallelism and locality of convolution due to the organization of their register files. Figure 2 presents equations for an n-tap 1D convolution that form the basis of a SIMD based convolution implementation presented in Figure 3. We demonstrate in Figure 3 the limitations of a SIMD based convolution implementation by executing a 16-tap convolution on a 128-bit SIMD datapath. This is a typical SIMD implementation similar to the one presented in [29], and the SIMD datapath is similar to ones found in many current processors. To enable the datapath to utilize the vector registers completely irrespective of the filter size, the convolution operation is vectorized across output locations allowing the datapath to compute eight output values in parallel.

Given the short integer computation that is required, one needs a large amount of parallelism per instruction to be energy efficient [20]. While this application has the needed parallelism, scaling the datapath by eight times to perform sixty four 16-bit operations per cycle would prove extremely costly. It would require an eight times increase in the register file size, inflating it to 1024-bits, greatly increasing its energy and area. To make matters worse, as shown in Figure 3, the energy efficiency of the SIMD datapath is further degraded by the fact that a substantial percentage of instructions are used to perform data shuffles which consume instruction and register energy without doing any operations. Alternatively, one can reload shifted versions of vectors from the memory to avoid data shuffles; however, that also results in substantial energy waste due to excessive memory fetches. These data motion overheads are worse for vertical and 2-D convolution.

GPUs target massively data parallel applications and can achieve much higher performance for convolution operations than SIMD. However, due to their large register file structures and 32-bit floating point units, we don't expect GPUs to have very low energy consumption. To evaluate this further we measure the performance and energy consumption of an optimized GPU implementation of H.264 SAD algorithm [28] using GPUGPUSim simulator [6] with GPUWattch energy model [21]. The GPU implementation runs forty times faster compared to an embedded 128-bit SIMD unit, but consumes thirty times higher energy. Even with a GPU customized for media applications we do not expect the energy consumption to be much better than the SIMD implementation as the GPU energy is dominated by register file, which is central to how GPUs achieve their high degree of parallelism.

CE reduces most of the register file overheads described earlier with the help of a shift register file or a FIFO like storage structure. As shown in Figure 4, when such a storage structure is augmented with an ability to generate multiple shifted versions of the input data, it can not only facilitate execution of multiple simultaneous stencils, but can also eliminate most of the shortcomings of traditional vector register files. Aided by the ability to broadcast data, these multiple shifted versions can fill sixty four ALUs from just a small 256-bit register file saving valuable register file access energy as well as area.

Our CE facilitates further reductions in energy overheads by supporting more complex operation in the reduction tree, allowing multiple "instructions" to be fused together. This fusion also offers the added benefit of eliminating temporary storage of intermediate data in big register files saving valuable register file energy. Fur-

thermore, by changing the shift register to have two dimensions, and by allowing column accesses and two dimensional shifts, these shift registers possess the potential to extensively improve the energy efficiency of vertical and two dimensional filtering. As Figure 5 shows, these 1D and 2D shift registers sit at the heart of our Convolution Engine.

CE is developed as a domain specific hardware extension to Tensilica's extensible RISC cores [18]. Augmented with user-defined hardware interfaces called TIE ports, developed using Tensilica's TIE language [30], these RISC cores control the program flow of CE by first decoding the instructions in their instruction fetch unit and then routing the appropriate control signals to CE using TIE ports. Since the number of cores that interface with CE can be more than one, the TIE ports are muxed. The cores are also responsible for memory address generation, but the data is sent/return directly from the register files within CE. The next sections discuss the key blocks depicted in Figure 5.

4.1 Load/Store Unit and Register Files

The load/store unit loads and stores data to and from the various register files. To improve efficiency, it supports multiple memory access widths with the maximum memory access width being 256-bits and can handle unaligned accesses.

The Convolution Engine uses a 1D shift register to supply data for horizontal convolution flow. New image pixels are shifted horizontally into the 1D register as the 1D stencil moves over an image row. 2D shift is used for vertical and 2D convolution flows and supports vertical row shift: one new row of pixel data is shifted in as the 2D stencil moves vertically down into the image. The 2D register provides simultaneous access to all of its elements enabling the interface unit to feed any data element into the ALUs as needed. A standard vector register file, due to its limited design, is incapable of providing all of this functionality.

The 2D Coefficient Register stores data that does not change as the stencil moves across the image. This can be filter coefficients, current image pixels in IME for performing SAD, or pixels at the center of Windowed Min/Max stencils. The results of filtering operations are either written back to the 2D Shift Register or the Output Register. The Output Register is designed to behave both as a 2D Shift register as well as a Vector Register file for the vector unit. The shift behavior is invoked when the output of the stencil operation is written. This shift simplifies the register write logic and reduces the energy. This is especially useful when the stencil operation produces the data for just a few locations and the newly produced data needs to be merged with the existing data which results in a read modify write operation. The Vector Register file behavior is invoked when the Output Register file is interfaced with the vector unit shown in the Figure 5.

4.2 MAP & Reduce Logic

As described earlier we abstract out convolution as a *map* step that transforms each input pixel into an output pixel. In our implementation interface units and ALUs work together to implement the *map* operation; the interface units arrange the data as needed for the particular map pattern and the functional units perform the arithmetic.

Interface Units: Interface Units (IF) arrange data from the shift register into a specific pattern needed by the map operation. Currently this includes providing shifted versions of 1D and 2D blocks, and column access to 2D register, though we are also exploring a more generalized permutation layer to support arbitrary maps. All of the functionality needed for generating multiple shifted versions of the data is encapsulated within the IFs. This allows us to shorten

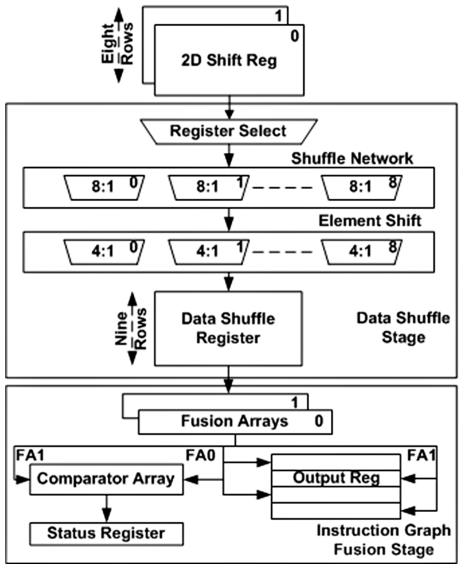

Figure 6: Complex Graph Fusion Unit.

the wires by efficiently generating the needed data within one block while keeping the rest of the datapath simple and relatively free of control logic. Since the IFs are tasked to facilitate stencil based operations, the multiplexing logic remains simple and prevents the IFs from becoming the bottleneck.

The Horizontal Interface generates multiple shifted versions of the 1D data and feeds them to the ALU units. The data arrangement changes depending on the size of the stencil so this unit supports multiple power of 2 stencil sizes and allows selecting between them. Column Interface simultaneously access the columns of the 2D Shift register to generate input data for multiple locations of a vertical 1D filtering kernel. The 2D interface behaves similarly to the Vertical interface and accesses multiple shifted 2D data blocks to generate data for multiple 2D stencil locations. Again multiple column sizes and 2D block sizes are supported and the appropriate one is selected by the convolution instruction.

Functional Units: Since all data re-arrangement is handled by the interface unit, the functional units are just an array of short fixed point two-input arithmetic ALUs. In addition to multipliers, we support difference of absolute to facilitate SAD and other typical arithmetic operations such as addition, subtraction, comparison. The output of the ALU is fed to the Reduce stage.

Reduce Unit: The *reduce* part of the *map-reduce* operation is handled by a general purpose reduce stage. Based upon the needs of our applications, we currently support arithmetic and logical reduction stages. The degree of reduction is dependent on the kernel size, for example a 4x4 2D kernel requires a 16 to 1 reduction whereas 8 to 1 reduction is needed for an 8-tap 1D kernel. The reduction stage is implemented as a tree and outputs can be tapped out from multiple stages of the tree. the function of the reduce unit can be increased by the Graph Fusion Unit which is described next.

4.3 Complex Graph Fusion Unit

To increase the domain of applications which can effectively use the CE, we allow a more complex combining tree than just implementing true reductions. This extra complexity enables us to merge many different convolution instructions into a single "super instruction" which allows a small program to be executed for each pixel in one convolution instruction. Demosaic is a good example, since

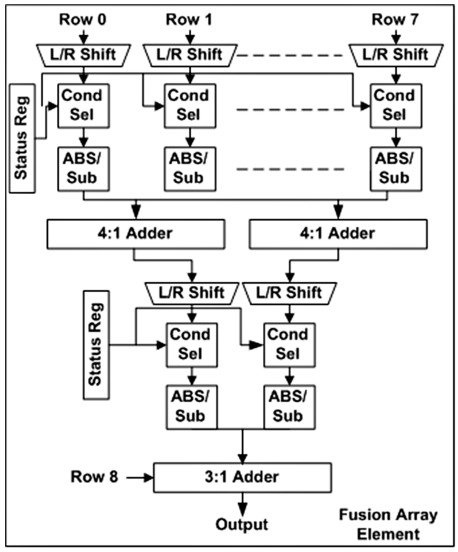

Figure 7: Details of Instruction Graph Fusion Stage.

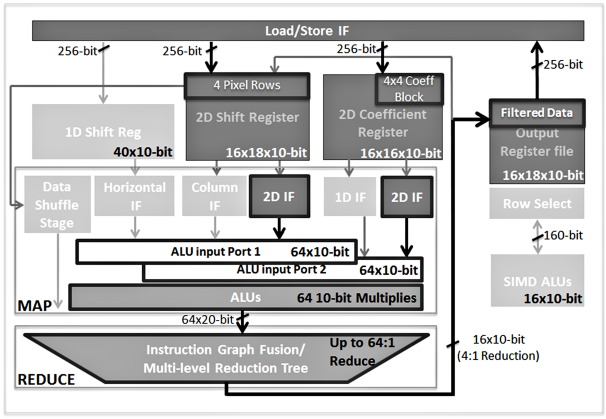

Figure 8: Executing a 4x4 2D Filter on CE. The grayed out boxes represent units not used in the example. The sizes of all of the resources are defined which will be explained in a later section.

it needs to adjust its operation depending on its local gradients. While Demosaic could use a conventional CE, it would first need to compute its gradients. Then it would need to compare the gradients to find out which direction was more stable, and finally using this information it could compute the needed output. Since all the information required is available from the original input data, and the total computation is not complex, a more complex combining tree can do the entire computation in one step. This increases the computational efficiency proportionally to the reduction in required instructions ($\sim$5x).

This extra capability is provided by the programmable complex graph fusion unit (CGFU) presented in Figure 6. The CGFU has the ability to fuse together up to nine arithmetic instructions. CGFU obtains its input from both the input and the output registers. Since this more complex data combination is not commutative, the right data (output of the map operation) must be placed on each input to the combining network. Thus the CGFU includes a very flexible swizzle network that provides permutations of the input data and sends it to a shifter unit which takes the shuffled data to perform element level shifts. These two units combine to form a highly flexible swizzle network that can reorder the data to support 1D horizontal, 1D vertical and even 2D window based fusions. This flexibility costs energy, so it is bypassed on standard convolution instructions. We also separate the data shuffle stage (DS) from the actual instruction graph fusion stage, since often many convolution instructions can use the data once it is shuffled.

While the DS stage is tasked with data reordering, the Instruction Graph Fusion (IGF) stage is responsible for executing the more complex data combining to implement the fused instruction subgraphs. The energy wasted in register file accesses is reduced by employing dedicated storage structure called Data Shuffle Register file to communicate between IGF and data shuffle stages. The most critical parts of the IGF stage are the two fusion arrays which are shown in Figure 7. Each array supports a variety of arithmetic operations and can implement data dependent data-flow by using predicated execution. These units are pipelined, so bits of the Status Register which are set from computation early in the combining tree can be used later in the computation to generate the desired output. Like the normal reduction network, the outputs of the two arrays are also fed to a two dimensional output register where they are stored in pairs. The absorption of IGF into the reduction stage does entail higher energy costs for convolution operations, but our experiments indicate that the overheads remain less than 15%.

4.4 SIMD & Custom Functional Units

To enable an algorithm to perform vector operations on the output data, we have added a 16-element SIMD unit that interfaces with the Output Register. This unit accesses the 2D Output Register as a Vector Register file to perform regular Vector operations. This is a lightweight unit which only supports basic vector add and subtract type operations and has no support for higher cost operations such as multiplications found in a typical SIMD engine.

An application may perform computation that conforms neither to the convolution block nor to the vector unit, or may otherwise benefit from a fixed function implementation. If the designer wishes to build a customized unit for such computation, the Convolution Engine allows the fixed function block access to its Output Register File. This model is similar to a GPU where custom blocks are employed for rasterization and such, and that work alongside the shader cores. For these applications, we created three custom functional blocks to compute motion vector costs in IME and FME and the Hadamard Transform in FME.

4.5 A 2-D Filter Example

Figure 8 shows how a 4x4 2D filtering operation maps onto the convolution engine. Filter coefficients reside in first four rows of the Coefficient Register. Four rows of image data are shifted into the first four rows of the 2D Shift register. In this example we have 64 functional units so we can perform filtering on up to four 4x4 2D locations in parallel. The 2D Interface Unit generates four shifted versions of 4x4 blocks, lays them out in 1D and feeds them to the ALUs. The Coefficient Register Interface Unit replicates the 4x4 input coefficients 4 times and send them to the other ALU port. The functional units perform an element-wise multiplication of each input pixel with corresponding coefficients and the output is fed to the Reduction stage. The degree of reduction to perform is determined by the filter size which in this case is 16:1. The four outputs of the reduction stage are normalized and written to the Output Register.

Since our registers contain data for sixteen filter locations, we continue to perform the same operation described above; however, the 2D Interface Unit now employs horizontal offset to skip over

Table 2: Sizes for various resources in CE.

	Resource Sizes
ALUs	64 10-bit ALUs
1D Shift Reg	40 x 10bit
2D Input Shift Reg	16 rows x 18 cols x 10bit
2D Output Shift Register	16 rows x 18 cols x 10bit
2D Coefficient Register	16 rows x 16 cols x 10bit
Horizontal Interface	4, 8, 16 kernel patterns
Vertical Interface	4, 8, 16 kernel patterns
2D Interface	4x4, 8x8 , 16x16 patterns
Reduction Tree	4:1, 8:1, 16:1, 32:1, 64:1

Table 3: Energy for filtering instructions implemented as processor extensions with 32, 64 or 128 ALUs. Overhead is the energy for instruction fetch, decode and sequencing.

	32 ALUs	64 ALUs	128 ALUs
Total Energy (pJ)	156	313	544
Overhead Energy (pJ)	37	39	44
Percent Overhead	24	12	8

already processed locations and to get the new data while the rest of the operations execute as above. Once we have filtered sixteen locations, the existing rows are shifted down and a new row of data is brought in and we continue processing the data in the vertical direction. Once all the rows have been processed we start over from the first image row, processing next vertical stripe and continue execution until the whole input data has been filtered.

For symmetric kernels the interface units combine the symmetric data before coefficient multiplication (since the taps are the same), allowing it to use adders in place of multipliers. Since adders take 2-3x lower energy, this further reduces wasted energy. The load/store unit also provides interleaved access where data from a memory load is split and stored into two registers. An example use is in demosaic, which needs to split the input data into multiple color channels.

4.6 Resource Sizing

Energy efficiency and resource requirements of target applications drive the sizes of various resources within CE. Energy overheads such as instruction fetch and decode affect the efficiency of programmable systems and can only be amortized by performing hundreds of arithmetic operations per instruction as shown in [20]. However, the authors in [20] studied small data such as 8-bit addition/subtraction, while convolution is typically dominated by multiplication that takes more energy per operation. To determine how to size the ALUs for CE with the goal of keeping overheads as low as possible, we present the energy dissipated in executing filtering instructions using 32, 64 and 128 10-bit ALUs (the precision required) in Table 3. In this table the total energy is comprised of the energy wasted in the processor overheads including fetch, decode and sequencing as well as the useful energy spent in performing the actual compute. As the number of ALUs increases, the overhead energy as a percentage of the total energy reduces. We choose 64 as the number of ALUs in CE as a compromise between efficiency and flexibility because it is easier to chain small units. The rest of the resources are sized to keep 64, 10-bit ALUs busy. The size and capability of each resource is presented in Table 2. These resources support filter sizes of 4, 8 and 16 for 1D filtering and 4x4, 8x8 and 16x16 for 2D filtering. Notice that that the register file sizes deviate from power of 2; this departure allows us to handle boundary

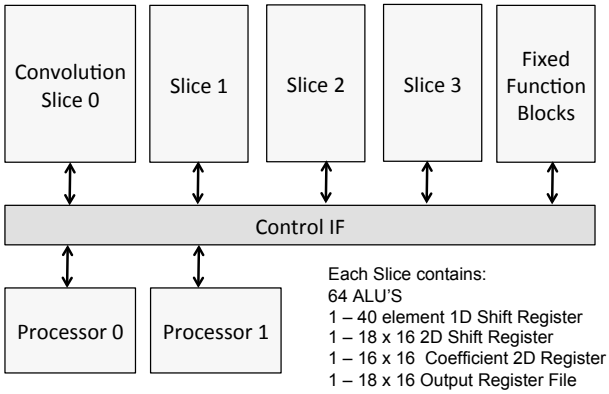

Figure 9: Convolution Engine CMP.

conditions common in convolution operations efficiently.

4.7 Convolution Engine CMP

To meet the diverse performance and energy requirements of different applications effectively, we have developed a CE chip multiprocessor (CMP) shown in Figure 9. The CMP consists of four CEs and two Tensilica's extensible RISC processors communicating with the CEs through muxed TIE ports as described earlier in this section. The decision to support two independent threads of control in the form of two processors is influenced largely by the requirements of the applications of interest, but also to a lesser extent by energy efficiency as smaller TIE port muxes keep energy wasted per instruction low. In the CMP, each instance of the CE is referred to as a slice and the slices possess the capability to operate completely independent of other slices and also in concatenation to perform an even larger number of operations per cycle. Dynamic concatenation of slices is especially desirable when the performance requirements of an algorithm cannot be satisfied by one slice or when the algorithm operates on small data requiring more than 64 operations per cycle to amortize overheads. When the slices are concatenated dynamically the register files and interface units of the interconnected slices are joined through short wires that run from one slice to another. Since the slices are laid out in close proximity to one another as shown in 9, these wires waste very little energy; therefore, not influencing the energy efficiency of connected slices. In addition to connecting multiple slices together to form a bigger slice with wide registers and ALU arrays, it is also possible to shut off the ALUs in the additional slices and use their registers as additional independent storage structures. Although, all the slices offer the same functionality, slices 0 and 1 are also equipped with complex graph fusion units integrated into their reduction blocks. The side effect of this integration is the additional 10-15% cost incurred by convolution operations executed on these slices. The processors and the slices are fed by dual-ported 16K instruction and 32K data caches. As has been discussed earlier, the processors are responsible for data address generation for the connected slices, but the flow of data into and out of the data cache is controlled by the slices themselves.

4.8 Programming the Convolution Engine

Convolution Engine is implemented as a processor extension and adds a small set of instructions to processor ISA. These CE instructions can be issued as needed in regular C code through compiler intrinsics. Table 4 lists the major instructions that CE adds to the ISA and Listing 1 presents a simplified example code which im-

Table 4: Major instructions added to processor ISA.

	Description
SET_CE_OPS	Set arithmetic functions for MAP and REDUCE steps
SET_CE_OPSIZE	Set convolution size
LD_COEFF_REG_n	Load n bits to specified row of 2D coeff register
LD_1D_REG_n	Load n bits to 1D shift register. Optional Shift left
LD_2D_REG_n	Load n bits to top row of 2D shift register. Optional shift row down
ST_OUT_REG_n	Store top row of 2D output register to memory
CONVOLVE_1D_HOR	1D convolution step - input from 1D shift register
CONVOLVE_1D_VER	1D convolution step - column access to 2D shift register
CONVOLVE_2D	2D Convolution step with 2D access to 2D shift register

plements 15-tap horizontal filtering for a single image row. There are mainly 3 types of instructions. Configuration instructions set options which are expected to stay fixed for a kernel such as convolution size, ALU operation to use etc. Other options which can change on a per instruction basis are specified as instruction operands. Then there are load and store operations to store data into appropriate registers as required. There is one load instruction for each input register type (1D input register, 2D input register, Coefficient register). Finally there are the compute instructions, one for each of the 3 supported convolution flows – 1D horizontal, 1D vertical and 2D. For example the CONVOLV_2D instruction reads one set of values from 2D and coefficient registers, performs the convolution and write the result into the row 0 of 2D output register. The load, store and compute instructions are issued repeatedly as needed to implement the required algorithm.

```
// Set MAP function = MULT, Reduce function = ADD
SET_CE_OPS (CE_MULT, CE_ADD);

// Set convolution size 16, mask out 16th element
SET_CE_OPSIZE(16, 0x7fff);

// Load 16 8-bit coefficients into Coeff Reg Row 0
LD_COEFF_REG_128(coeffPtr, 0);

// Load & shift 16 input pixels into 1D shift register
LD_1D_REG_128(inPtr, SHIFT_ENABLED);

// Filtering loop
for (x = 0; x < width - 16; x += 16) {
   // Load & Shift 16 more pixels
   LD_1D_REG_128(inPtr, SHIFT_ENABLED);

   // Filter first 8 locations
   CONVOLVE_1D_HOR(IN_OFFSET_0, OUT_OFFSET_0);

   // Filter next 8 locations
   CONVOLVE_1D_HOR(IN_OFFSET_8, OUT_OFFSET_8);

   // Add 2 to row 0 of output register
   SIMD_ADD_CONST (0, 2);

   // Store 16 output pixels
   ST_OUT_REG_128(outPtr);

   inPtr += 16;
   outPtr += 16;
}
```

Listing 1: Example C code implements a 15-tap filter for one image row and adds 2 to each output.

The code example in Listing 1 brings it all together. First CE is set to perform multiplication at MAP stage and addition at reduce stage which are the required setting for filtering. The the convolution size is set which controls the pattern in which data is fed from the registers to the ALUs. Filter tap coefficients are then loaded into the coefficient register. Finally the main processing loop repeatedly loads new input pixels into the 1D register and issues 1D_CONVOLVE operations to perform filtering. While 16 new pixels are read with every load, our 128-ALU CE configuration can only process eight 16-tap filters per operation. Therefore two 1D_CONVOLVE operations are performed per iteration, where the second operation reads the input from an offset of 8 and writes its output at an offset of 8 in the output register. For illustration purposes we have added a SIMD instruction which adds 2 to the filtering output in row 0 of 2D output register. The results from output register are written back to memory.

It is important to note that unlike a stand-alone accelerator the sequence of operations in CE is completely controlled by the software which gives complete flexibility over the algorithm. Also CE code is freely mixed into the C code which gives added flexibility. For example in the filtering code above it is possible for the algorithm to produce one CE output to memory and then perform a number of non-CE operations on that output before invoking CE to produce another output.

4.9 Generating Instances with Varying Degrees of Flexibility

The programmable convolution engine as described has full flexibility in silicon to perform any of the supported operations, and the desired operation is selected at run time through a combination of configuration registers and instruction operands. However, we also want to study the individual impact of various programmability options present in CE. To facilitate that, we have designed the CE in a highly parameterized way such that we can generate instances with varying degrees of flexibility ranging from fixed kernel as shown in Figure 4, to fully programmable. When the fixed kernel instance is generated in Figure 4 the whole 2D register with its associated interface unit goes away. The 1D interface also goes away, replaced by the hardwired access pattern required for the particular kernel. The remaining registers are sized just large enough to handle the particular kernel, the flexible reduction tree is replaced by a fixed reduction and the ALU only supports the single arithmetic operation needed.

The efficiency of this fixed kernel datapath matches the custom cores. The programmability options that convolution engine has over this fixed kernel datapath can be grouped into three classes which build on top of each other:

Multiple kernel sizes: This includes adding all hardware resources to support multiple kernel sizes, such that we still support only a single kernel, but have more flexibility. The support for that primarily goes in interface units which become configurable. Register files have to be sized to efficiently support all supported kernel sizes instead of one. The reduction stage also becomes flexible.

Multiple flows: This step adds the remaining data access pat-

terns not covered in previous step, such that all algorithm flows based on the same arithmetic operations and reduction type can be implemented. For example for a core supporting only 2D convolutions, this step will add vertical and 1D interfaces with full flexibility and also add any special access patterns not all already supported including offset accesses, interleaved writes and so on.

Multiple arithmetic operations: This class adds multiple arithmetic and logical operations in the functional units, as well as multiple reduction types (summation versus logical reduction).

The next section describes how we map different applications to a Convolution Engine based CMP and the experiments we perform to determine the impact of programmability on efficiency. By incrementally enabling these options on top of a fixed kernel core we can approach the fully programmable CE in small steps and assess the energy and area cost of each addition.

5. EVALUATION METHODOLOGY

To evaluate the Convolution Engine approach, we map each target application on CE based CMP described in 4.7. As already discussed this system is fairly flexible and can easily accommodate algorithmic changes such as changes in motion estimation block size, changes in down-sampling technique etc. Moreover, it can be used for other related algorithms such as a different feature detection scheme like SURF, or other common operations like sharpening or denoising etc.

To quantify the performance and energy cost such a programmable unit, we also build custom heterogeneous chip multiprocessors (CMPs) for each of the three applications. These custom CMPs are based around application-specific cores, each of which is highly specialized and only has resources to do a specific kernel required by the application. Both the CE and application-specific cores are built as a datapath extension to the processor cores using Tensilica's TIE language [30]. Tensilica's TIE compiler uses this description to generate simulation models and RTL as well as area estimates for each extended processor configuration. To quickly simulate and evaluate the CMP configurations, we created a multiprocessor simulation framework that employs Tensilica's Xtensa Modeling Platform (XTMP) to perform cycle accurate simulation of the processors and caches. For energy estimation we use Tensilica's energy explorer tool, which uses a program execution trace to give a detailed analysis of energy consumption in the processor core as well as the memory system. The estimated energy consumption is within 30% of actual energy dissipation. To account for interconnection energy, we created a floor plan for the CMP and estimated the wire energies from that. That interconnection energy was then added to energy estimates from tensilica tools. The simulation results employ 45nm technology at 0.9V operating voltage with a target frequency of 800MHz. All units are pipelined appropriately to achieve the frequency target.

To further extend the analysis, we quantify the individual cost of different programmability options discussed in Section 4.9. Starting from a fixed kernel datapath closely matching the custom hardware, we add the programmability options in steps. That way we can identify the incremental cost of each programmability class and understand if some types of programmability options are costlier than others.

Figure 10 presents how each application is mapped to our CE based CMP. This mapping is influenced by the application's performance requirements. In this study, like most video systems these days, we support HD 1080P video at 30FPS. This translates to an input data rate of around 60 MPixels/s. For still images we want to support a similar data rate of around 80-100 MPixels/s which can be translated for example to processing 10MP images at 8-10FPS

Figure 10: Mapping of applications to CE CMP.

or 5MP images at a higher rate of 16-20FPS etc. H.264 motion estimation only deals with video data, whereas SIFT and Demosaic can be applied to both video and still images. However, when SIFT and Demosaic are applied to full HD video streams the resolution drops from 5MP to 2.1MP increasing the frame rate substantially. Now, we describe the mapping in detail for each application.

H.264 Motion Estimation: Our mapping allocates one processor to the task of H.264 integer motion estimation. The 4x4 SAD computation is mapped to the convolution engine block, and the SIMD unit handles the task of combining these to form the larger SAD results. This requires a 16x32 2D shift register and 128 ABS-DIFF ALU units, so 2 slices are allocated to this processor. In addition a fixed function block is used to compute motion vector cost, which is a lookup-table based operation. Fractional motion estimation uses up only 64 ALU units, but requires multiple register files to handle the large amount of data produced by up-sampling, so it takes up 2 slices. The convolution engine handles up-sampling and SAD computation. A custom fixed function block handles the Hadamard transform.

SIFT: Each level in the SIFT Gaussian pyramid requires five 2D Gaussian blur filtering operations, and then down-sampling is performed to go to the next level. The various Gaussian blurs, the difference operation and the down-sampling are all mapped to one of the processors, which uses one convolution engine slice. The Gaussian filtering kernel is a separable 2D filtering kernel so it is implemented as a horizontal filter followed by a vertical filter. The second processor handles extrema detection, which is a windowed min/max operation followed by thresholding to drop weak candidates. This processor uses 2 slices to implement the windowed min across 3 difference images and SIMD operations to perform the thresholding. SIFT generates a large amount of intermediate pyramid data, therefore 64x64 image blocking is used to minimize the intermediate data footprint in memory. The minima operation crosses block boundaries so buffering of some filtered image rows is required. Moreover, the processing is done in multiple passes, with each pass handling each level of the pyramid.

Demosaic: Demosaic generates a lot of new pixels and intermediate data and thus needs multiple 2D shift register files. It uses register resources from two slices and further uses register blocking to get multiple virtual registers from the same physical register. Demosaicing is the first step in a typical camera pipeline. In our current mapping, the second processor and remaining two slices are idle when demosaicing is in operation. However, these resources can be used to implement next steps in the imaging pipeline such as white balance, denoising, and sharpening which are also based on convolution-based kernels.

6. RESULTS

Figures 11 and 12 compare the performance and energy dissipation of the proposed Convolution Engine against a 128-bit data-parallel (SIMD) engine and an application specific accelerator implementation for each of the five algorithms of interest. In most cases we used the SIMD engine as a 16-way 8-bit datapath, but in a few examples we created 8-way 16-bit datapaths. For our algo-

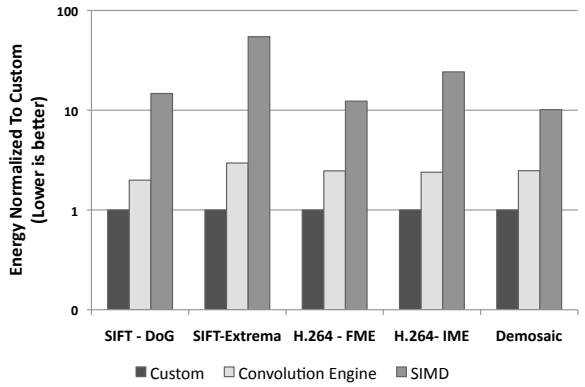

Figure 11: Energy Consumption normalized to Custom implementation: Convolution Engine vs Custom Cores and SIMD.

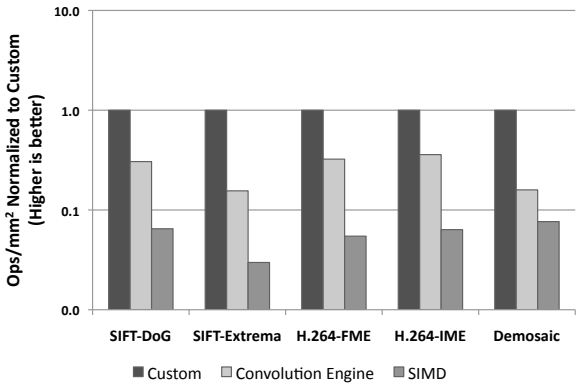

Figure 12: Ops/mm2 normalized to Custom implementation: Number of image blocks each core processes in one second, divided by the area of the core. For H.264 an image block is a 16x16 macroblock and for SIFT and demosaic it is a 64x64 image block.

rithms, making this unit wider did not change the energy efficiency appreciably.

The fixed function data points truly highlight the power of customization: for each application a customized accelerator required 8x-50x less energy compared to an optimized data-parallel engine. Performance per unit area improves a similar amount, 8x-30x higher than the SIMD implementation. Demosaic achieves the smallest improvement (8x) because it generates two news pixel values for every pixel that it loads from the memory. Therefore, after the customization of compute operations, loads/stores and address manipulation operations become the bottleneck and account for approximately 70% of the total instructions.

Note the biggest gains were in IME and SIFT extrema calculations. Both kernels rely on short integer *add*/*subtract* operations that are very low energy (relative to the *multiply* used in filtering and up-sampling). To be efficient when the cost of compute is low, either the data movement and control overhead should be very low, or more operations must be performed to amortize these costs. In a SIMD implementation these overheads are still large relative to the amount of computation done. These kernels also use a 2D data flow which requires constant accesses and fetches from the register file. Custom hardware, on the other hand, achieves better performance at lower energy by supporting custom 2D data access patterns. Rather than a vector, it works on a matrix which is shifted every cycle. Having more data in flight enables a larger number arithmetic units to work in parallel, better amortizing instruction and data fetch.

With this analysis in mind, we can now better understand where a Convolution Engine stands. The architecture of the Convolution Engine is closely matched to the data-flow of convolution based algorithms, therefore the instruction stream difference between fixed function units and the Convolution Engine is very small. Compared to a SIMD implementation, the convolution engine requires 8x-15x less energy with the exception of Demosaic that shows an improvement of 4x while the performance to area ratio of CE is 5-6x better. Again Demosaic is at the low end of the gain as a consequence of the abundance of loads and stores. If we discount the effect of memory operations from Demosaic, assuming its output is pipelined into another convolution like stage in the image pipeline, the CGFU based Demosaic implementation is approximately 7x better than SIMD and within 6x of custom accelerator. The higher energy ratio compared to a custom implementation points up the costs of the more flexible communication in CGFU compared to CE's blocks optimized for convolution.

The energy overhead of the CE implementation over application specific accelerator is modest (2-3) for the other applications, and requires only twice the area. While these overheads are small, we were interested to better understand which programmability option discussed in Section 4.9 added the most overhead compared to custom accelerators. To generate these results, For each convolution algorithm we start with an accelerator specialized to do only the specific convolution kernels required for that algorithm, and then gradually add flexibility. These results are shown in Figures 13 and 14.

For SIFT's filtering stage, the first programmability class entails an increase in energy dissipation of just 25% which is relatively small. The fixed function hardware for SIFT already has a large enough 1D shift register to support a 16-tap 1D horizontal filter so adding support for smaller 4 and 8 tap 1D filters only requires adding a mall number of multiplexing options in 1D horizontal IF unit and support for tapping the reduction tree at intermediate levels. However, the second programmability class incurs a bigger penalty because now a 2D shift register is added for vertical and 2D flows. The coefficient and output registers are also upgraded from 1D to 2D structures, and the ALU is now shared between Horizontal, Vertical and 2D operations. The result is a substantial increase in register access energy and ALU access energy. Moreover, the 2D register comes with support for multiple vertical and 2D kernel sizes as well as support for horizontal and vertical offsets and register blocking, so the area gets a big jump shown in 14 and consequently the leakage energy increases as well. The final step of adding multiple compute units has a relatively negligible impact of 10%.

For SIFT extrema the cost of adding multiple kernel sizes is again only 1.3x. However, supporting additional access patterns adds another 2x on top of that bringing the total cost to roughly 2.5x over the fixed kernel version. Unlike filtering stage, SIFT extrema starts with 2D structures so the additional cost of adding the 1D horizontal operations is relatively low. However, the 2D and vertical IF units also become more complex to support various horizontal and vertical offsets into the 2D register. The cost of multiplexing to support these is very significant compared to the low energy map and reduce operations used in this algorithm. The result is a big relative jump in energy. The last step of supporting more arithmetic operations again has a relatively small incremen-

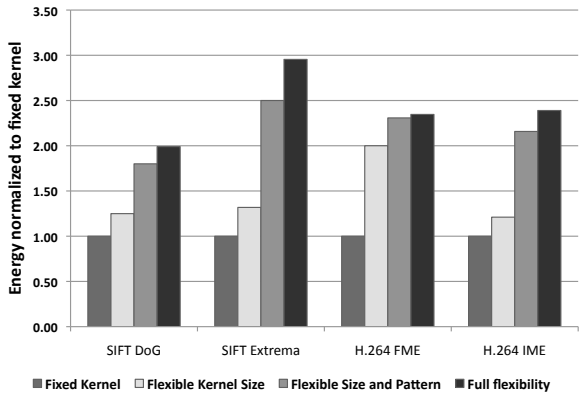

Figure 13: Change in energy consumption as programmability is incrementally added to the core.

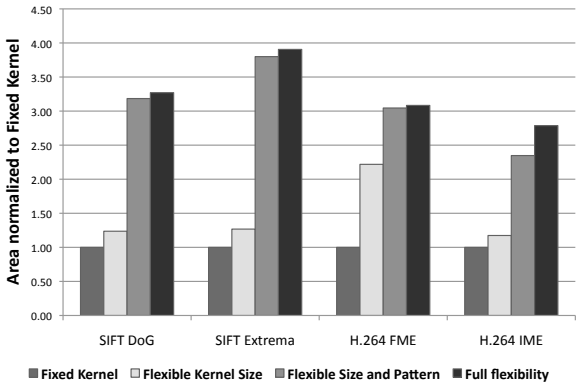

Figure 14: Increase in area as programmability is incrementally added to the core.

tal cost of around 1.2x. The final programmable version still takes roughly 12x less energy compared to the SIMD version.

Like SIFT extrema, IME also has a lightweight map step (absolute difference), however, it has a more substantial reduction step (summation). So the relative cost of muxing needed to support multiple 2D access patterns is in between the high-energy-cost filtering operations and low-energy-cost extrema operations. The cost of supporting multiple kernel sizes and multiple arithmetic operations is still relatively small.

FME differs slightly from other algorithms in that it takes a big hit when going to multiple kernel sizes. The fixed function core supports 1D-Horizontal and 1D-Vertical filtering for a relatively small filter size of 8 taps. The storage structures are sized accordingly and consist of two small 2D input and two even smaller 2D output shift registers. Adding support for multiple kernel sizes requires making each of these registers larger. Thus multiple stencil sizes not only require additional area in the interface units, but the bigger storage structures also make the muxes substantially bigger, increasing the register access cost. This is further exacerbated by the increase in the leakage energy brought about by the bigger storage structures which is fairly significant at such small feature sizes. Thus the first programmability class has the most impact on the energy efficiency of FME. The impact of the second programmability class is relatively modest as it only adds a 2D interface unit – most of the hardware has already been added by the first programmability class. The cost of supporting multiple arithmetic operations is once again small suggesting that this programmability class is the least expensive to add across all algorithms.

Our results show that the biggest impact on energy efficiency takes place when the needed communication paths become more complex. This overhead is more serious when the fundamental computation energy is small. In general the communication path complexity grows with the size of the storage structures, so over provisioning registers as is needed in a programmable unit hurts efficiency. This energy overhead is made worse since such structures not only require more logic in terms of routing and muxing, but also have a direct impact on the leakage energy which is significant at such small feature sizes. On the other hand, more flexible function units have small overheads, which provides flexibility at low cost.

7. CONCLUSION

As specialization emerges as the main approach to addressing the energy limitations of current architectures, there is a strong desire to make maximal use of these specialized engines. This in turn argues for making them more flexible, and user accessible. While flexible specialized engines might sound like an oxymoron, we have found that focusing on the key data-flow and data locality patterns within broad domains allows one to build a highly energy efficient engine, that is still user programmable. We presented the Convolution Engine which supports a number of different algorithms from computational photography, image processing and video processing, all based on convolution-like patterns. A single CE design supports applications with convolutions of various size, dimensions, and type of computation. To achieve energy efficiency, CE captures data reuse patterns, eliminates data transfer overheads, and enables a large number of operations per cycle. CE is within a factor of 2-3x of the energy and area efficiency of single-kernel accelerators and still provides an improvement of 8-15x over general-purpose cores with SIMD extensions for most applications. While the CE is a single example, we hope that similar studies in other application domains will lead to other efficient, programmable, specialized accelerators.

8. ACKNOWLEDGEMENT

This material is based upon work supported by the Defense Advanced Research Projects Agency under Contract No. HR0011-11-C-0007. Any opinions, findings and conclusion or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency.

9. REFERENCES

[1] Digic Processors, Canon Inc. http://learn.usa.canon.com/resources-/articles/2012/digic_processors.htmlp.

[2] Omap 5 platform, texas instruments. www.ti.com/omap.

[3] Snapdragon Processors, Qualcomm Inc. http://www.qualcomm.com/snapdragon/processors.

[4] Tegra processors. NVIDIA Corporation.

[5] A. Adams, D. Jacobs, J. Dolson, M. Tico, K. Pulli, E. Talvala, B. Ajdin, D. Vaquero, H. Lensch, M. Horowitz, et al. The frankencamera: an experimental platform for computational photography. *ACM Transactions on Graphics (TOG)*, 2010.

[6] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[7] J. Balfour, W. Dally, D. Black-Schaffer, V. Parikh, and J. Park. An energy-efficient processor architecture for embedded systems. *Computer Architecture Letters*, 7(1):29–32, 2007.

[8] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Computer Vision–ECCV 2006*, pages 404–417, 2006.

[9] B. Bayer. Color imaging array, 1976. US Patent 3,971,065.

[10] J. D. Brown. The ibm power edge of network processor. In *The Technical Record of the 22nd Hot Chips Conference*, Aug. 2010.

[11] T. C. Chen. Analysis and architecture design of an HDTV720p 30 frames/s H.264/AVC encoder. *IEEE Transactions on Circuits and Systems for Video Technology*, 16(6):673–688, 2006.

[12] Y. Cheng, K. Xie, Y. Zhou, and Y. Liu. An adaptive color plane interpolation method based on edge detection. *Journal of Electronics (China)*, 2007.

[13] J. Cong, V. Sarkar, G. Reinman, and A. Bui. Customizable domain-specific computing. *IEEE Des. Test*, 28(2):6–15, Mar. 2011.

[14] N. Corporation. *Expeed Digital Image Processors*. Nikon Corporation., http://imaging.nikon.com/lineup/microsite/d300.

[15] S. Corporation. *BIONZ Image Processing Engine*. Sony Corporation., http://www.sony-mea.com/microsite/dslr/10/tech/bionz.html.

[16] P. Debevec, E. Reinhard, G. Ward, and S. Pattanaik. High dynamic range imaging. In *ACM SIGGRAPH 2004 Course Notes*, page 14. ACM, 2004.

[17] R. Golla and P. Jordan. T4: A highly threaded server-on-a-chip with native support for heterogeneous computing. In *The Technical Record of the 23rd Hot Chips Conference*, Aug. 2011.

[18] R. Gonzalez. Xtensa: a configurable and extensible processor. *Micro, IEEE*, 20(2):60–70, Mar/Apr 2000.

[19] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *Micro, IEEE*, 2012.

[20] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding Sources of Inefficiency in General-Purpose Chips. In *ISCA '10: Proc. 37th Annual International Symposium on Computer Architecture*. ACM, 2010.

[21] J. Leng, S. Gilani, T. Hetherington, A. ElTantawy, N. S. Kim, T. M. Aamodt, and V. J. Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *ISCA 2013: International Symposium on Computer Architecture*, 2013.

[22] D. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[23] Y. Matsushita, E. Ofek, X. Tang, and H. Shum. Full-frame video stabilization. In *Computer Vision and Pattern Recognition (CVPR), 2005. IEEE Computer Society Conference on*.

[24] G. Petschnigg, R. Szeliski, M. Agrawala, M. Cohen, H. Hoppe, and K. Toyama. Digital photography with flash and no-flash image pairs. In *ACM Transactions on Graphics (TOG)*.

[25] R. Raskar. Computational photography. In *Computational Optical Sensing and Imaging*. Optical Society of America, 2009.

[26] O. Shacham, S. Galal, S. Sankaranarayanan, M. Wachs, J. Brunhaver, A. Vassiliev, M. Horowitz, A. Danowitz, W. Qadeer, and S. Richardson. Avoiding game over: Bringing design to the next level. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, june 2012.

[27] A. Solomatnikov, A. Firoozshahian, W. Qadeer, O. Shacham, K. Kelley, Z. Asgar, M. Wachs, R. Hameed, and M. Horowitz. Chip Multi-Processor Generator. In *DAC '07: Proceedings of the 44th Annual Design Automation Conference*, 2007.

[28] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, vLi Wen Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu. Impact technical report. In *IMPACT-12-01*, 2012.

[29] Tensilica Inc. ConnX Vectra LX DSP Engine Guide.

[30] Tensilica Inc. Tensilica Instruction Extension (TIE) Language Reference Manual.

[31] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. ASPLOS '10. ACM, 2010.