DOI:10.1145/2735841

Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing

By Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark Horowitz

Abstract

General-purpose processors, while tremendously versatile, pay a huge cost for their flexibility by wasting over 99% of the energy in programmability overheads. We observe that reducing this waste requires tuning data storage and compute structures and their connectivity to the data-flow and data-locality patterns in the algorithms. Hence, by backing off from full programmability and instead targeting 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 within that domain.

We present the Convolution Engine (CE)—a programmable processor specialized for the convolution-like data-flow prevalent in computational photography, computer vision, and video processing. The CE achieves energy efficiency by capturing data-reuse patterns, eliminating data transfer overheads, and enabling a large number of operations per memory access. We demonstrate that the CE is within a factor of 2–3× of the energy and area efficiency of custom units optimized for a single kernel. The CE improves energy and area efficiency by 8–15× over data-parallel Single Instruction Multiple Data (SIMD) engines for most image processing applications.

1. INTRODUCTION

Processors, whether they are the relatively simple RISC cores in embedded platforms, or the multibillion transistor CPU chips in Server/Desktop computers, are extremely versatile computing machines. They can handle virtually any type of workload ranging from web applications, personal spreadsheets, image processing workloads, and embedded control applications to database and financial applications. Moreover, they benefit from well-established programming abstractions and development tools, and decades of programming knowledge making it very easy to code new applications.

Processors, however, are also inefficient computing machines. The overheads of predicting, fetching, decoding, scheduling, and committing instructions account for most of the power consumption in a general-purpose processor core.[2, 7, 16] As a result they often consume up to 1000× more energy than a specialized hardware block designed to perform just that particular task. These specialized hardware blocks also typically offer hundreds of times higher performance using a smaller silicon area. Despite these large inefficiencies, processors form the core of most computing systems owing to their versatility and reuse.

Over decades, we have been able to scale up the performance of general-purpose processors without using excessive power, thanks to advances in semiconductor device technology. Each new technology generation exponentially reduced the switching energy of a logic gate enabling us to create bigger and more complex designs with modest increases in power. In recent years, however, the energy scaling has slowed down,[12] thus we can no longer scale processor performance as we used to do. Today we fundamentally need to reduce energy waste if we want to scale performance at constant power.

This paper presents a novel highly efficient processor architecture for computational photography, image processing, and video processing applications, which we call the Convolution Engine (CE). With the proliferation of cheap high quality imagers, computational photography and computer vision applications are expected to be critical consumer computation workloads in coming years. Some example applications include annotated reality, gesture-based control, see-in the dark capability, and pulse measurement.

Many of these applications, however, will require multiple TeraOps/s of computation which is far beyond the capability of general processor cores especially mobile processors on a constrained power budget of less than 1 Watt. The three orders of magnitude advantage in compute efficiency of hardware accelerators, means that current mobile systems use heterogeneous computing chips combining processors and accelerators.[11,15] An example accelerator is the video codec hardware employed in the mobile SOCs. However, these accelerators target either a single algorithm or small variations on an algorithm. Handling the diverse application set needed by these future smart devices requires a more programmable computing solution. Our CE provides that programmability while still keeping energy dissipation close to dedicated accelerators.

Our design approach is based on the observation that *specialized units achieve most of their efficiency gains by tuning data storage structures to the data-flow and data-locality requirements of the algorithm*. This tuning eliminates redundant data transfers and facilitates creation of closely coupled

The original version of this paper is entitled "Convolution Engine: Balancing Efficiency and Flexibility in Specialized Computing" and was published in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, June 2013.

data-paths and storage structures allowing hundreds of low-energy operations to be performed for each instruction and data fetched. Processors can enjoy similar energy gains if they target computational motifs and data-flow patterns common to a wide-range of kernels in a domain. Our CE implements a generalized map-reduce abstraction, which describes a surprisingly large number of operations in image processing domain. The resulting design achieves up to two orders of magnitude lower energy consumption compared to a general-purpose processor and comes within 2–3× of dedicated hardware accelerators.

The next section provides an overview of why general processors consume so much power and the limitations of existing optimization strategies. 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. We then compare this CE to both general-purpose cores with SIMD extensions and to highly customized solutions for individual kernels in terms of energy and area efficiency. Section 5 shows that the CE is within a factor of 2–3× of custom units and almost 10× better than the SIMD solution for most applications.

2. BACKGROUND

The low efficiency of general-purpose processors is explained in Figure 1, which compares the energy dissipation of various arithmetic operations with the overall instruction energy for an extremely simple RISC processor. The energy dissipation of arithmetic operations that perform the useful work in a computation remains much lower than the energy wasted in the instruction overheads such as instruction fetch, decode, pipeline management, program sequencing, etc. The overhead is even worse for media processing applications which typically operate on short data requiring just 0.2–0.5 pJ (90 nm) of energy per operation, with the result that over 99% energy goes into overheads.

These overheads must be drastically reduced to increase energy efficiency. That places two constraints on the processor design, (i) the processor must execute hundreds of operations per instruction to sufficiently amortize instruction cost and (ii) the processor must also fetch little data, since even a cache hit costs 25 pJ (90 nm) per memory fetch, compared to 0.2–0.5 pJ for the arithmetic operations.

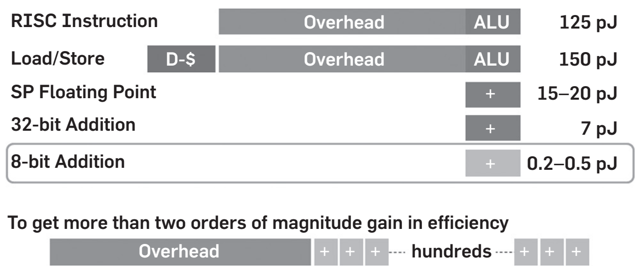
These two constraints seem contradictory as performing hundreds of operations per cycle would generally necessitate reading large amounts of data from the memory. These conditions could be reconciled, however, for algorithms where most instructions either operate on intermediate results produced by previous instructions, or reuse most of the input data used by the previous instructions. If an adequate storage structure is in place to retain this "past data" in the processor data-path, then large number of operations can be performed per instruction without needing frequent trips to the memory. Fortunately compute bound applications including most image processing and video processing algorithms are a good match for these constraints, providing large data-parallelism and data-reuse.

Most high-performance processors today already include SIMD units which are widely regarded as the most efficient general-purpose optimization for compute intensive applications. SIMD units typically achieve an order of magnitude energy reduction by simultaneously operating on many data operands in a single cycle (typically 8–16). However, as explained in Hameed et al.[7] the resulting efficiency remains two orders of magnitude less than specialized hardware accelerators, as the SIMD model does not scale well to larger degrees of parallelism.

To better understand the architectural limitations of traditional SIMD units, consider the two-dimensional sum of absolute difference operation (SAD) operating on a 16-bit 8 × 8 block as shown in Listing 1.[5] The 2D SAD operator is widely employed in multimedia applications such as H.264 video encoder to find the closest match for a 2D image sub-block in a reference image or video frame. Listing 1 carries out this search for every location in a *srchWinHeight* × *srchWinWidth* search window in the reference frame, resulting in four nested loops. All four loops are independent and can be simultaneously parallelized. At the same time, each SAD output substantially reuses the input data used to compute previous outputs, both in vertical and horizontal directions.

However, a typical SIMD unit with a register row size of 128 bits is only able to operate on elements that fit in one register row limiting the parallelism to the inner most loop. Trying to scale up the SIMD width to gain more parallelism requires either simultaneously reading multiple image rows from the register file (to parallelize across the 2nd most-inner loop), or simultaneously reading multiple overlapping rows of image data (to parallelize across multiple horizontal outputs). Neither support exists in the SIMD model.

Figure 1. Comparison of functional unit energy with that of a typical RISC instruction in 90nm. Strategy for amortizing processor overheads includes executing hundreds of low-power operations per instruction.



Listing 1. 2D 8 × 8 sum of absolute difference operation (SAD), commonly employed in H.264 motion estimation.

```
for (sWinY = 0; sWinY < srchWinHeight; sWinY++){
  for (sWinX = 0; sWinX < srchWinWidth; sWinX++){
    sad = 0;
    for (y = 0; y < 8; y++){
      for (x = 0; x < 8; x++){
        cY = y + sWinY; cX = x + sWinX;
        sad += abs(ref[cY][cX] - cur[y][x]);
      }
    }
    outSad[sWinY][sWinX] = sad;
  }
}
```

Exploiting the data-reuse requires storing the complete 8×8 block in eight rows of the SIMD register file, so that the data is locally available for computing subsequent output pixels. This is straightforward in vertical direction, as every new output just needs one new row and the seven old rows could be reused. To get reuse in the horizontal direction, however, each of the eight rows must be shifted by one pixel before computing each new output pixel, incurring substantial instruction overhead. At the same time, since the shifting process destroys the old pixels, we are limited to getting reuse either in the vertical or horizontal direction but not both, with the result that each data item gets fetched eight times from the memory, wasting too much memory energy.

GPUs achieve a much higher degree of parallelism by using a large number of simple SIMD cores, each with local register resources, and very large memory bandwidth to maintain high computational throughput. While that results in great compute throughput, the energy consumption is also very high thanks to large data access cost. Measuring the performance and energy consumption of an optimized GPU implementation of H.264 SAD algorithm[13] using GPUGPUSim simulator[1] with GPUWattch energy model,[9] we ascertain that the GPU implementation runs 40× faster compared to an embedded 128-bit SIMD unit, but also consumes 30× higher energy. That further illustrates the need to minimize memory accesses and provide low-cost local data-accesses.

As the next section shows, the SAD operator belongs to a large class of algorithms which can be described as convolution-like and have the desired parallelism and reuse characteristics for efficient execution. Next section discusses this computational abstraction and provides a number of example applications.

3. COMPUTATIONAL MODEL

Equations (1) and (2) provide the definition of standard discrete 1D and 2D 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 3×3 to 8×8 for 2D convolution:

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

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

We generalize the concept of convolution by identifying two components of the convolution: a *map* operation and a *reduce* operation. In Equations (1) and (2), the map operation is the multiplication that is done on pairs of pixel and tap coefficient, and the reduce operation is the summation of all these pairs to the output 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 H.264's motion estimation. Further replacing the reduce operation from Σ to *max* will yield a max of absolute differences operation. Equation (3) expresses the computation model of our CE, where *f*, *Map*, and *Reduce* ("*R*" in Equation 3) are the pseudo instructions, and *c* is the size of the convolution:

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

The convolution like data-flow works for many applications, but is sometimes limited by having a single associative operation in the reduction. There are a number of applications that have a data locality pattern similar to convolution, but need to combine results through a specific graph of operations rather than simple reduction. These applications can be handled by the CE by increasing the complexity of the *Reduce* operator to enable noncommutative functions, and allowing a different function to be used at each reduction stage. This generalized combining network extends the "reduction" stage to create a structure that can input a large number of values and then compute a small number of outputs through effectively a fused super instruction as shown in Figure 2.

The down side of this extension is that the placement of input into the combining tree is now significant; thus, to realize the full potential of the new generalized *reduce* operator, a high level of flexibility is required in the data supply network to move the needed data to the right position. This is achieved by extending the definition of the *map* operator to also support a data permutation network in addition to the already supported set of compute operators. These new enhancements to the *map* and *reduce* operators substantially boost their generalizability and applicability and thus increase the space of supported applications.

We now introduce a few kernels that we use in this paper and discuss how each of them maps into the computing abstractions we have defined above. Table 1 summarizes this information.

3.1. Motion estimation

Motion estimation is a key component of many video codecs including H.264, consuming about ~90% of the execution time for H.264 software implementations.[4, 7] The kernel operates on subblocks of a video frame, trying to find each subblock'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:

Integer Motion Estimation (IME): IME searches for the matching block in the reference image using the SAD

Figure 2. Generalized reduction unit fuses multiple operations into a super instruction.

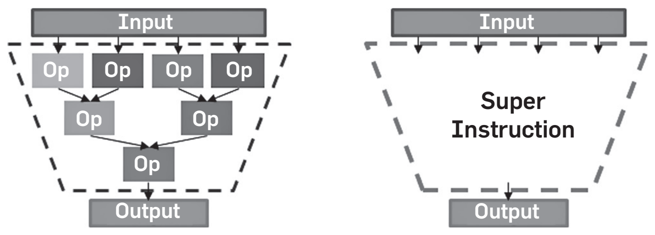


Table 1. Mapping kernels to convolution abstraction.

	Map	Reduce	Stencil sizes	Data flow
IME SAD	Abs diff	Add	4 × 4	2D convolution
FME 1/2 pixel up-sampling	Multiply	Add	6	1D horizontal and vertical convolution
FME 1/4 pixel up-sampling	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	9 × 3	2D convolution
Demosaic interpolation	Multiply	Complex	3	1D horizontal and vertical convolution

Some kernels such as subtraction operate on single pixels and thus have no stencil size defined. We call these as matrix operations. There is no reduce step for these operations.

operator previously described in Section 2. Note how SAD fits quite naturally to a CE abstraction: the *map* function is absolute difference and the *reduce* function is summation.

Fractional Motion Estimation (FME): FME refines the initial match obtained at the IME step to a quarter-pixel resolution. It first up-samples the block selected by IME, and then performs a slightly modified variant of the SAD operation. 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. Second, 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.[10] 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.[10]

Gaussian blurring and down-sampling are naturally 2D convolution operations. Finding scale-space extrema is a 3D stencil computation, but we can convert it 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.[3] 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),[8] 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 nontrivial since individual color values from the mosaic must be separated before performing interpolation.

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, as explained earlier, existing SIMD units are limited in the extent to which they can exploit the inherent parallelism and locality of convolution due to the organization of their register files. The CE overcomes these limitations with the help of shift register structures. As shown in Figure 3 for the 2D convolution case, when such a storage structure is augmented with an ability to generate multiple shifted versions of the input data, it can fill 128 ALUs from just a small 16×8 2D register with low access energy as well as area. Similar gains are possible for 1D horizontal and 1D vertical convolutions. As we will see shortly, the CE facilitates further reductions in energy overheads by creating fused super-instructions introduced in Section 3.

The CE is developed as a domain specific hardware extension to Tensilica's extensible RISC cores.[6] The extension hardware is developed using Tensilica's TIE language.[14] The next sections discuss the key blocks in the CE extension hardware, depicted in Figure 4.

4.1. Register files

The 2D shift register 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 shift register provides simultaneous access to all of its elements enabling the interface unit to feed any data element to the ALUs. 1D shift register is used 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.

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 convolution operations are either written back to the 2D Shift Register or the Output Register. A later

Figure 3. Implementation of 8 × 8 2D SAD operation that exploits parallelism in all four loops of Listing 1. The reference block resides in a 2D shift register while the current block is stored in a 2D register. Because both registers allow 2D access of the 8 × 8 block, 64 ALUs can operate in parallel. To enable an even larger degree of parallelism and to exploit data-reuse in the horizontal direction, the shift register generates pairs of multiple overlapping 8 × 8 blocks which are then fed to the ALU through a multiplexer. These pairs allow parallel execution of 128 ALUs generating two outputs in parallel. After the generation of four pairs of horizontal outputs, the shift register shifts up by one to make room for a new row of search window achieving vertical data-reuse.

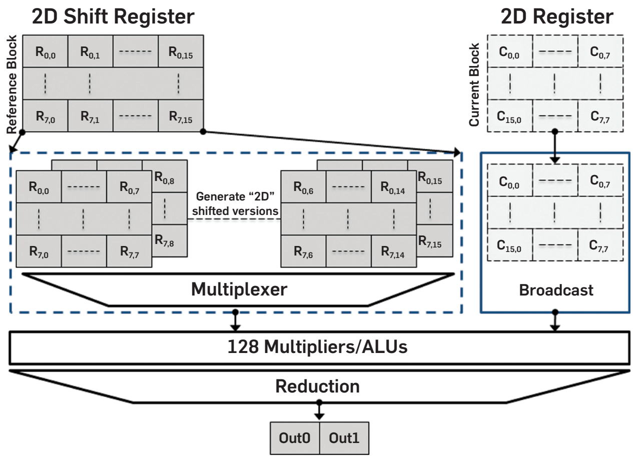
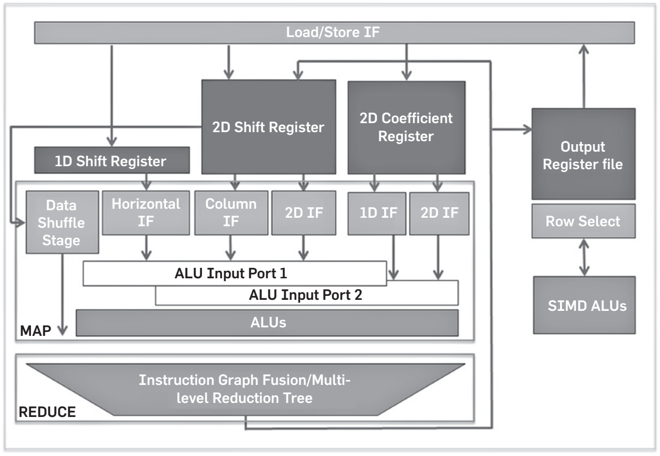


Figure 4. Block diagram of convolution engine. The interface units (IF) connect the register files to the functional units and provide shifted broadcast to facilitate convolution. Data shuffle (DS) stage combined with instruction graph fusion (IGF) stage create the generalized reduction unit, and is called the complex graph fusion unit.



section shows how the output register file also works as the vector register file for the vector unit shown in Figure 4.

4.2. Map and reduce logic

As described earlier we abstract convolution as a *map* and *reduce* 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. The Interface Units (IF) arrange data from the register files into a specific pattern needed by the map operation. For 2D convolutions, multiple shifted 2D sub-blocks can be simultaneously accessed from the 2D register. Multiple block sizes such as 2 × 2, 4 × 4, 8 × 8, etc. are supported and the appropriate size is selected based on convolution kernel size. Similarly for vertical convolution, multiple 2D register columns can be accessed in parallel, with support for multiple column access sizes. Finally, the 1D IF supports accessing multiple shifted 1D blocks from the 1D shift register for horizontal convolution. We are also exploring a more generalized permutation layer to support arbitrary maps.

Functional units. Since all data rearrangement 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 absolute difference to facilitate SAD and other typical arithmetic operations such as addition, subtraction, and 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 programmable 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 4 × 4 2D kernel requires a 16 to 1 reduction whereas 8 to 1 reduction is needed for an 8-tap 1D kernel. Thus, the reduction stage is implemented as a combining tree and outputs can be tapped out from multiple stages of the tree.

To enable the creation of "super instructions" described in Section 3, we augment the combining tree to enable handle noncommutative operations by adding support for diverse arithmetic operations at different levels of the tree. This fusion increases the computational efficiency by reducing the number of required instructions and by eliminating temporary storage of intermediate data in register files. Because this more complex data combination need not be commutative, the right data (output of the map operation) must be placed on each input to the combining network. Thus, a "Data Shuffle Stage" is also added to the CE in the form of a very flexible swizzle network that provides permutations of the input data.

4.3. Other hardware

To facilitate vector operations on the convolution output, we have added a 32-element SIMD unit. This unit interfaces with the 2D Output Register and uses it as a Vector Register file. This unit is wider than typical SIMD units, as it operates on intermediate data generated by convolution data path and thus is not constrained by data memory accesses. Despite being wider, the vector unit is still lightweight as it only supports basic vector add and subtract type operations and has no support for higher cost operations such as multiplications.

Because 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 CE 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.

4.4. Resource sizing

Energy efficiency considerations and resource requirements of target applications drive the sizes of various resources within CE. As shown in Hameed et al.,[7] amortizing the instruction cost requires performing hundreds of ALU operations per instruction for media processing applications based on short 8-bit Addition/Subtraction operations. Many convolution flow applications are, however, based on higher energy multiplication operations. Our analysis shows that for multiplication-based algorithms, 50–100 operations per instruction are enough to provide sufficient amortization. Increasing the number of ALUs much further than that gives diminishing returns and increases the size of storage required to keep these units busy, thus increasing storage area and data-access energy. Thus for this study we choose an ALU array size of 128 ALUs, and size the rest of the resources accordingly to keep these ALUs busy. To provide further flexibility we allow powering off half of the ALU array and compute structures. 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 4 × 4, 8 × 8, and 16 × 16 for 2D filtering. Notice that that the register file sizes deviate from power of 2 to efficiently handle boundary conditions common in convolution operations.

4.5. Programming the convolution engine

CE is implemented as a processor extension and adds a small set of instructions to the processor ISA. These CE instructions can be issued as needed in regular C code through compiler intrinsics. Table 3 lists the major CE instructions. Configuration instructions set kernel parameters such as convolution size, ALU operation to use at Map and Reduction stages, etc. Then, there are load and store operations for each register resource. Finally, there are the compute instructions, one for each of the three 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 code example in Listing 2 brings it all together and implements a 2D 8 × 8 Filter. First the CE is set to perform multiplication at MAP stage and addition at reduce stage which are the required setting for filtering. Then the convolution size is set which controls the pattern in which data is fed from the registers to the ALUs. Filter tap coefficients

Table 2. Sizes for various resources in CE.

	Resource sizes
ALUs	128 × 10 bit ALUs
1D shift register	80 × 10 bit
2D input shift register	16 rows × 36 cols × 10 bit
2D output shift register	16 rows × 36 cols × 10 bit
2D coefficient register	16 rows × 16 cols × 10 bit
Horizontal interface	4, 8, 16 kernel patterns
Vertical interface	4, 8, 16 kernel patterns
2D interface	4 × 4, 8 × 8, and 16 × 16 patterns
Reduction tree	4:1, 8:1, …, 128:1

Table 3. 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

Listing 2. Example C code implements 8 × 8 2D filter for a vertical image stripe and adds 2 to each output.

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

// Set convolution size 8
SET_CE_OPSIZE(8);

// Load eight rows of eight 8-bit coefficients
// into Coeff Reg's rows 0 to 7
for(i = 0; i < 8; i++){
    LD_COEFF_REG_128(coeffPtr, 0);
    coeffPtr += coeffWidth;
}

// Load & shift seven rows of sixteen input pixels
// into 2D shift register
for(i = 0; i < 7; i++){
    LD_2D_REG_128(inPtr, SHIFT_ENABLED);
    inPtr += width;
}

// Filtering loop
for (y = 0; y < height; y++) {
   // Load & Shift 16 more pixels
   LD_2D_REG_128(inPtr, SHIFT_ENABLED);

   // Filter first 8 locations. Because we have
   // access to 128-ALUS, we can filter two 8x8
   // blocks in parallel
   for(RW_OFFSET = 0; RW_OFFSET < 8; RW_OFFSET+=2){
      CONVOLVE_2D(RW_OFFSET, RW_OFFSET);
   }

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

   // Store 8 output pixels
   ST_OUT_REG_64(outPtr);

   inPtr += width;
   outPtr += width;
}
```

are then loaded into the coefficient register. Finally, the main processing loop repeatedly loads new input pixels into the 2D shift register and issues 2D_CONVOLVE operations to perform filtering. While 16 new pixels are read with every load, our 128-ALU CE configuration can only process two 8×8 filtering operation per operation. Therefore four 2D_CONVOLVE operations are performed per iteration. For illustration purposes we have added a SIMD instruction which adds 2 to each output value produced by the convolution operation. The results from output register are written back to memory.

Figure 5 maps this execution onto the CE hardware. The 8×8 coefficients are stored in the first eight rows of the Coefficient Register File. Eight rows of image data are shifted into the first eight rows of the 2D Shift register. Since we have 128 functional units, we can filter two 8×8 2D locations at a time. Thus the 2D Interface Unit generates two shifted versions of 8×8 blocks, which are rearranged into 1D data and fed to the ALUs. The functional units performs an element-wise multiplication of each input pixel with the corresponding coefficient and the output is fed to the Reduction stage. The degree of reduction is determined by the filter size, which in this case is 8×8; therefore, 64:1 reduction is chosen. The two outputs of the reduction stage are normalized and written to the Output Register.

It is important to note that unlike a stand-alone accelerator the sequence of operations in CE is completely controlled by the C code which gives complete flexibility over the algorithm. 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.

5. EVALUATION

To evaluate the efficiency of the CE, we map each target application described in Section 3 on a chip multiprocessor (CMP) comprised of two CEs. To quantify the performance and energy cost of such a programmable unit, we also built custom heterogeneous chip multiprocessors (CMP) for each of the three applications. These custom CMPs are based around application-specific cores, each of which is highly specialized for 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.[14] Tensilica's TIE compiler uses this description to generate simulation models and RTL 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

Figure 5. Executing a 8 × 8 2D filter on CE. The grayed out boxes represent units not used in the example.

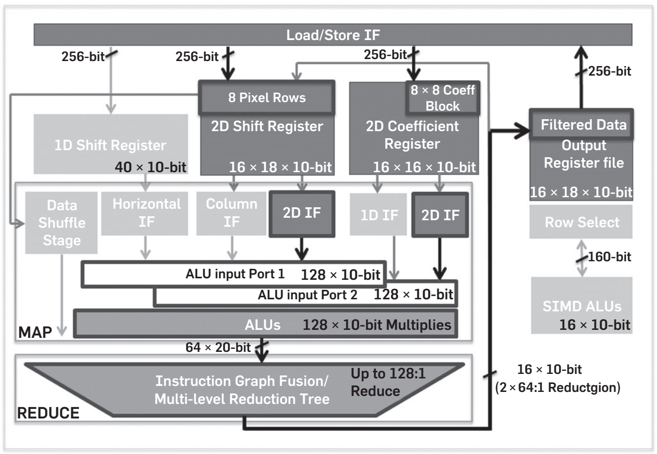


Figure 6. Energy consumption normalized to custom implementation.

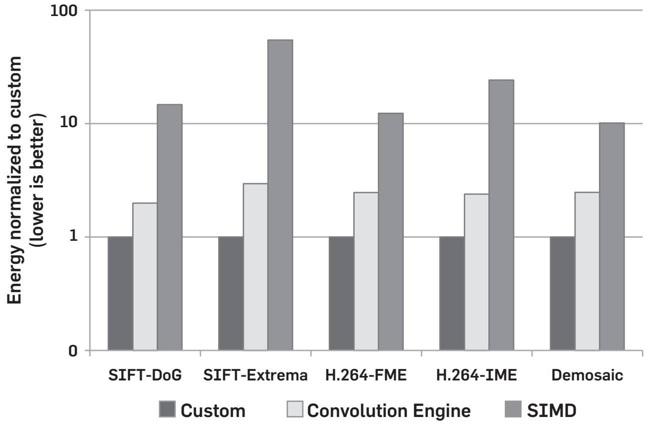
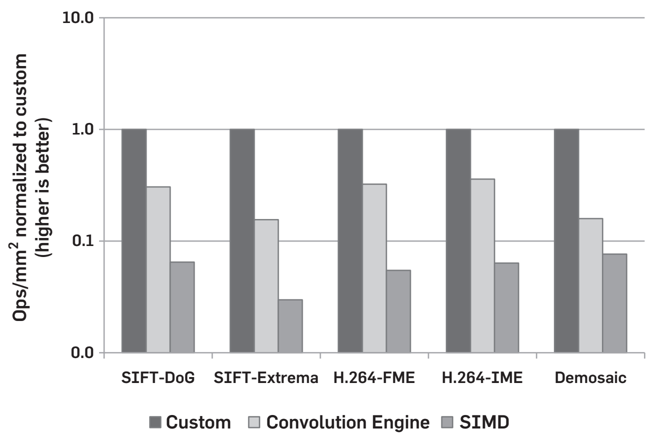


Figure 7. Ops/mm² normalized to custom implementation: number of image blocks each core processes in 1 second, divided by the area of the core. For H.264 an image block is a 16 × 16 macroblock and for SIFT and Demosaic it is a 64 × 64 image block.



interconnection energy was then added to energy estimates from Tensilica tools. The simulation results employ 90 nm technology at 1.1 V operating voltage with a target frequency of 450 MHz. All units are pipelined appropriately to achieve the frequency target.

Figures 6 and 7 compare the performance and energy dissipation of the proposed CE against a 128-bit data-parallel (SIMD) engine and the custom accelerator implementation for each of the five algorithms of interest. In most cases we used the SIMD engine as a 16-way 8-bit data-path, but in a few examples we created 8-way 16-bit data-paths. For our algorithms, 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 requires 8–50× less energy compared to SIMD engine. Performance per unit area is also 8–30× higher than the SIMD implementation. Demosaic achieves the smallest improvement (8×) because it generates two new 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). As previously discussed, for SIMD implementation the instruction overheads and data access energy are still large relative to the amount of computation done. Custom accelerators, on the other hand, are able to exploit the parallelism and data-reuse in their respective kernels, fully amortizing instruction and data fetch.

We can now better understand where the CE stands. The architecture of the CE is closely matched to the data-flow of convolution-based algorithms, therefore the instruction stream difference between fixed function units and the CE is very small. Compared to a SIMD implementation, the CE requires 8–15× less energy with the exception of Demosaic that shows an improvement of 4× while the performance to area ratio of CE is 5–6× 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 CE-based Demosaic is approximately 7× better than SIMD and within 6× of custom accelerator. The higher energy ratio compared to a custom implementation points up the costs of the more flexible communication in the generalized reduction.

However, for all the other applications the energy overhead of the CE compared to fixed function units stands at a modest 2–3×, while the area overhead is just 2×. While these over-heads are small, to better understand the sources of these inefficiencies, Figures 8 and 9 create three different implementations of each application, moving from a custom implementation, to one with flexible registers, but fixed computation, and to the fully flexible CE.

Figure 8. Change in energy consumption as programmability is incrementally added to the core.

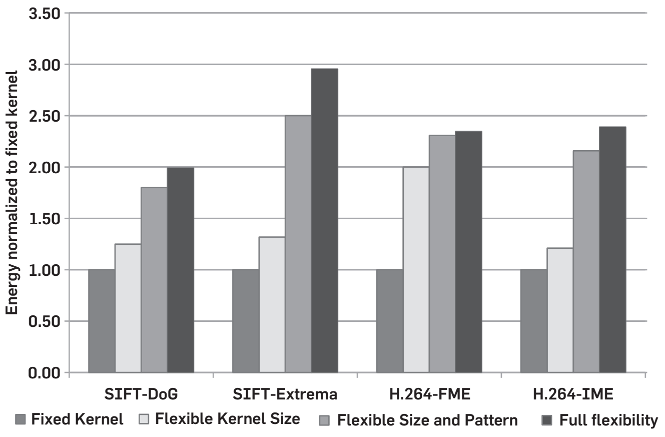
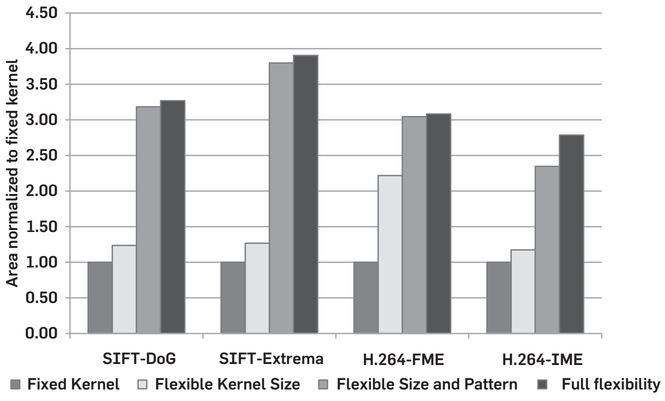


Figure 9. Increase in area as programmability is incrementally added to the core.



The figures 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 often 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. On the other hand, more flexible functional units have small overheads, which provides flexibility at low cost.

6. 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

CE 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 2–3× of the energy and area efficiency of single-kernel accelerators and still provides an improvement of 8–15× 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, and specialized accelerators.

Acknowledgments
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. C

References
1. Bakhoda, A., Yuan, G., Fung, W.W.L., Wong, H., Aamodt, T.M. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS: IEEE International Symposium on Performance Analysis of Systems and Software* (2009).
2. Balfour, J., Dally, W., Black-Schaffer, D., Parikh, V., Park, J. An energy-efficient processor architecture for embedded systems. *Comput. Architect. Lett. 7*, 1 (2007), 29–32.
3. Bayer, B. *Color Imaging Array*. US Patent Application No. 3971065 (1976).
4. Chen, T.-C., Chien, S.-Y., Huang, Y.-W., Tsai, C.-H., Chen, C.-Y., Chen, T.-W., Chen, L.-G. Analysis and architecture design of an HDTV720p 30 frames/sec H.264/AVC encoder. *IEEE Trans. Circuits Syst. Video Technol. 16*, 6 (2006), 673–688.
5. Corbal, J., Valero, M., Espasa, R. Exploiting a new level of DLP in multimedia applications. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture* (Nov. 1999), 72–79.
6. Gonzalez, R. Xtensa: A configurable and extensible processor. *Micro IEEE 20*, 2 (Mar. 2000), 60–70.
7. Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B.C., Richardson, S., Kozyrakis, C., Horowitz, M. Understanding sources of inefficiency in general-purpose chips. In *ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010), ACM.
8. Hamilton, J.F., Adams, J.E. *Adaptive Color Plane Interpolation in Single Sensor Color Electronic Camera*. US Patent Application No. 5629734 (1997).
9. Leng, J., Gilani, S., Hetherington, T., Tantawy, A.E., Kim, N.S., Aamodt, T.M., Reddi, V.J. GPUWattch: Enabling energy optimizations in GPGPUs. In *ISCA 2013: International Symposium on Computer Architecture* (2013).
10. Lowe, D. Distinctive image features from scale-invariant keypoints. *Int. J.Comput. Vision 60*, 2 (2004), 91–110.
11. NVIDIA Inc. Tegra mobile processors. http://www.nvidia.com/object/tegra-4-processor.html.
12. Shacham, O., Azizi, O., Wachs, M., Qadeer, W., Asgar, Z., Kelley, K., Stevenson, J., Solomatnikov A., Firoozshahian, A., Lee, B., Richardson, S., Horowitz, M. Rethinking digital design: Why design must change. *IEEE Micro 30*, 6 (Nov. 2010), 9–24.
13. Stratton, J.A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.W., Anssari, N., Liu, G.D., Hwu, W.-M.W. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. IMPACT Technical Report. In *IMPACT-12-01*, 2012.
14. Tensilica Inc. Tensilica Instruction Extension (TIE) Language Reference Manual.
15. Texas Instruments Inc. OMAP 5 platform. www.ti.com/omap.
16. Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., Taylor, M.B. Conservation cores: Reducing the energy of mature computations. In *ASPLOS'10* (2010), ACM.

Wajahat Qadeer and Rehan Hameed ({wqadeer, rhameed}@gmail.com), Palo Alto, CA.

Ofer Shacham (shacham@alumni.stanford.edu), Google, Mountain View, CA.

Preethi Venkatesan (preethiv@stanford.edu), Intel Corporation, Santa Clara, CA.

Christos Kozyrakis and Mark Horowitz ({kozyraki, horowitz}@stanford.edu), Stanford University, Stanford, CA.