Evaluating Programmable Architectures for Imaging and Vision Applications

Artem Vasilyev,* Nikhil Bhagdikar,* Ardavan Pedram,*†
Stephen Richardson,* Shahar Kvatinsky‡ and Mark Horowitz*
*Stanford University †Movidius ‡Technion

***Abstract*—Algorithms for computational imaging and computer vision are rapidly evolving, and hardware must follow suit: the next generation of image signal processors (ISPs) must be "programmable" to support new algorithms created with high-level frameworks. In this work, we compare flexible ISP architectures, using applications written in the Darkroom image processing language. We target two fundamental architecture classes: *programmable in time,* as represented by SIMD, and *programmable in space*, as typified by coarse grain reconfigurable array architectures (CGRA).**

We consider several optimizations on these two base architectures, such as register file partitioning for SIMD, bus based routing and pipelined wires for CGRA, and line buffer variations. After these optimizations on average, CGRA provides 1.6x better energy efficiency and 1.4x better compute density versus a SIMD solution, and 1.4x the energy efficiency and 3.1x the compute density of an FPGA. However the cost of providing general programmability is still high: compared to an ASIC, CGRA has 6x worse energy and area efficiency, and this ratio would be roughly 10x if memory dominated applications were excluded.

I. Introduction

Image data is exploding, driven by the availability of cheap imagers and recent advances in computational photography and image understanding. Unfortunately these cheap imagers have small pixels which yield noisy low-light performance and modest dynamic range, so creating good quality images from this data requires extensive processing. While a GPU can handle the computational demands, its power consumption greatly exceeds the 100mW-range power budget typical for these applications. Instead, a specialized image signal processor (ISP) is generally used, often expending energy of less than 1pJ/op in a 40nm technology.

To achieve this level of efficiency, most ISPs use a collection of fixed function, or at best microcoded, hardware blocks that implement the sequence of functions needed for processing the image. This fixed-function approach worked well in the past because algorithms changed slowly – traditionally, the algorithms were well-established and highly stable workhouse programs for converting raw image data to RGB pixels (the standard image pipe includes denoising, demosaicing, color correction, and sharpening), or various standards for image or video compression (JPEG, MPEG, H.264). Because the application base was well known and stable, the labor intensive design of both the functional units and their microcode could acceptably be amortized over a long time and multiple generations of product sales.

978-1-5090-3508-3/16/$31.00

The rapid growth in computational photography and computer vision is upsetting this stable application base. Today's cameras commonly include face detectors, panorama capture and high dynamic range imaging, while manufacturers are starting to explore light-field photography [1], digital zoom by merging multiple cameras [2], depth from stereo [3] and other techniques. Such features often get soft-coded for the less-efficient CPU/GPU pair on an SoC because building a new ISP block is too costly. Thus there is great interest in creating programmable ISPs and, indeed, several ISPs with programmable elements have been recently produced [4]–[6].

This growing need for programmability has also given rise to new languages/compilers for this domain. Halide [7] and Darkroom [8] are domain specific languages (DSL) that ease the task of creating high-performance imaging implementations for CPU and GPU, and can even target ASIC or FPGA hardware. This paper uses Darkroom applications to compare the energy and area costs for user-programmable ISPs. Like the RISC revolution of thirty years ago, we focus on the performance and energy cost of running compiler-generated rather than hand-optimized code, since this is the code that nearly all users will run.

We explore the two basic approaches to programmability: using instructions to configure in *time,* and using programmable wires to configure in *space.* A CPU, which is programmable in time, can execute an arbitrary compute graph by storing the live edges of the overall graph to registers/memory, and then using instructions to choose the right edge values and operations at each time step. To minimize the area and energy overhead of the instructions, we explore wide SIMD machines, leveraging the large amount of pixel parallelism in our target applications. FPGAs and coarse grain reconfigurable arrays (CGRAs), which are programmable in space, execute an arbitrary graph by configuring arithmetic/logical blocks and the wires between them to physically create that graph.

This paper makes the following contributions:

- We create and compare two image processors, a SIMD and a CGRA, and describe several locality optimizations important to our target application space.
- We use the same high-level applications, compiler, and hardware building blocks to compare SIMD and CGRA solutions, and show that the architectures are within 2x of each other in energy efficiency and compute density, with CGRA having an advantage in both metrics.

- We show that, in our best case solution, the (not unexpected) cost of programmability is a compute density one-tenth that of a comparable ASIC.

Section II discusses the high memory locality of image processing applications, and how different architectures exploit this for energy efficiency. Section III describes the line buffer memory architecture used in our CGRA design, while Sections IV and V explain our SIMD and CGRA architectures and optimizations for them. Section VI presents our evaluation framework, explaining how we generate area and energy numbers for our architectures, and how we compile code for them. The results of these architectures and the comparison with ASIC and FPGA are then given in Section VII.

II. Image Processing Architecture

Image and vision applications can be executed on custom hardware with high energy efficiency because they generally can be *blocked* for extreme memory locality with high compute intensity, and can use lower precision integer operations. With few memory references, the energy/op is set by the functional unit and local communication energy.

Minimizing energy often requires reading the input image directly from the sensor in scan line order, and then completely processing it before writing a result back to the memory. Hence, the only DRAM accesses happen when writing the finished picture. This means each pixel passes through a function unit only once. However, since functions often need to know the value of nearby pixels in the rows above and below the current position, each functional unit might need a small memory to hold enough lines of the image to provide this context. Since these buffers each hold a few scan lines of the image, they are called *line buffers*, and the resulting structure is a line buffered pipeline. These buffers also enable producer-consumer locality, since a function directly writes its output into the line buffer of its successor, preventing the need to access a more global memory.

The producer and consumer functions between successive line buffers are referred to as *kernels*, and the interconnection among kernels and line buffers mirrors the directed acyclic graph (DAG) of operations in an application [8]. The array of pixels used as input to each kernel is often called a *stencil*.

A. Existing Image Processors

Image processing pipelines have been implemented across many platforms. We classify these platforms into three main categories depending on how they are programmed: ASIC, or *non*-programmable (fixed function) architectures; multicore, vector and VLIW architectures which are programmable in *time*; and CGRA and FPGA which are programmable in *space.*

ASIC fixed-function solutions achieve the best efficiency and performance, and they are widely used to execute image processing and computer vision algorithms. These solutions implement the image processing pipeline directly in hardware. They have deep pipelines with line buffers in between compute nodes, where each node performs a specific operation in the application's dataflow graph. Well-known examples of ASIC solutions include image signal processor chips used in digital cameras, like Canon's DIGIC [9] and Fujitsu's Milbeaut [10]. These later appeared as fixed function blocks in mobile SoCs such as TI's OMAP [11], [12] and NVIDIA's Tegra [13], [14]

A similar approach was used in automotive computer vision systems for advanced driver assistance (ADAS), e.g. EyeQ from Mobileye [15] or Analog Devices' Blackfin [16]. More recently, such chips have been used for low power implementation of modern algorithms like HDR [17].

Efficiency, however, comes at the price of flexibility. Most ASICs typically only allow a few configurable parameters like filter coefficients. As a result, adding new features like face detection might require a new hardware implementation. As imaging and vision have become more dynamic, the delay required to create a new chip is no longer acceptable, which creates a strong need for an efficient programmable ISP.

Programmable-in-time approaches use a computing engine to execute the image application. Most of these solutions use some kind of SIMD or vector approach to amortize the energy and area overhead caused by the instruction fetch over many pixel-parallel data operations. VLIW solutions try to increase utilization of ALUs by allowing multiple sub-instructions to simultaneously use CPU resources after compile time reordering [18]. Some architectures such as Cadence Vision P5 [4], Myriad 1 [5], CEVA [19], and CogniVue [20] contain both SIMD and VLIW engines and support longer instructions to increase throughput, efficiency, and utilization.

GPUs add threads to SIMD by increasing the register file so as to allow multiple threads to be resident, which enables fast thread switching. This allows GPUs to have very high function unit utilization, since they context switch to hide memory and other long latency operations. As we will see, this decision increases energy cost, and is not energy efficient for this class of applications.

Spatially programmable architectures reconfigure their connections to support programmability and include FPGAs, which inspired the creation of a more specialized architecture known as the coarse grain reconfigurable array (CGRA). Past and present CGRA chips targeted for image processing include RaPID [21], PiCoGA [22], RICA [23], MorphoSys [24] and ADRES [25]. Others go beyond image processing to support other high-performance computing applications, including PipeRench [26], Dyser [27], triggered instruction architectures [28] and others. Recently, deep learning applications for imaging have inspired CGRA accelerators for convolutional neural networks including NeuFlow [29], NeuroCGRA [30], Origami [31] and Diannao [32]. A detailed overview of reconfigurable architectures and their applications can be found in Tessier's survey [33].

III. Line Buffer

As mentioned earlier, a key to achieving low energy for image processing is to prevent intermediate data from being read/written to DRAM, or even to a large on-chip cache. This section discusses a local buffering mechanism for achieving

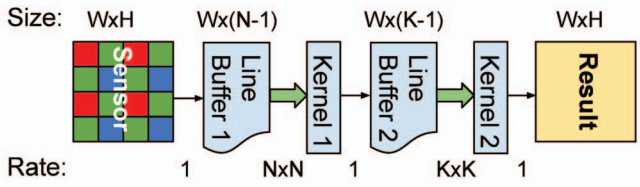

Fig. 1. Data flow for an ISP with two kernels and line buffers

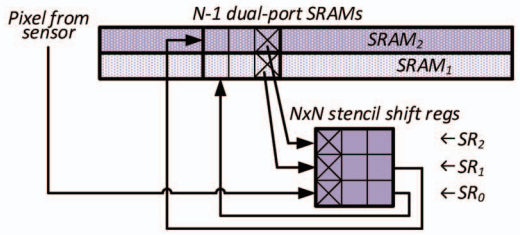

Fig. 2. Multi-SRAM line buffer architecture and operation for 3x3 kernel. Every cycle, one pixel goes from the sensor to shift register SR_0; from $SRAM_1$ to SR_1; and from $SRAM_2$ to SR_2. These operations load a new column into the shift registers. During this same cycle the output of SR_0 is written to $SRAM_1$ and SR_1 is written to $SRAM_2$ to move the data into the right SRAM for the next line.

this goal, and explains the memory system we used to evaluate different architectures.

An imaging system typically reads data directly from the image sensor in raster order (row-wise) into a chain of "kernels" that operate on a sliding data window. In one cycle, a kernel reads an N×N window of data and generates an output pixel. In the next cycle, the data window shifts one column to the right and a new output pixel is generated in raster order. When the data window reaches the right end of the image, it shifts one row down and back to the left end of the image, and the operation repeats. Thus, an image row is read N times for an N×N kernel. We can store N-1 image lines in a local buffer (called the *line buffer* or LB) and exploit this data reuse. Similarly, each column read from a line buffer is used N times by the N×N kernel. We store these columns in even smaller shift registers to exploit this data reuse.

For example, consider the operation of a two-kernel pipeline, similar to that shown in Figure 1, where Kernel 1 is a 5×5 blur operation followed by Kernel 2, a 3×3 median filter. We need to store at least four image lines (plus an additional 5 pixels) from the sensor in a "line buffer" before the 5x5 blur kernel can begin processing the data. Similarly, the output of the 5x5 filter is stored (in raster order) to another line buffer for use by the median kernel. Since the median kernel uses a 3×3 window, we need to store two image lines (and an additional 3 pixels) in this line buffer.

Note that, because image sensors produce data in raster (row-wise) order, the width of each line buffer must be equal to the image width, while its height is equal to the height of that buffer's kernel window minus one. Alternatively, a full frame of sensor output can be buffered in DRAM, and then processed in vertical strips, reducing the width of the required line buffers. This extra DRAM read and write usually makes the stripped system less energy efficient, although it does

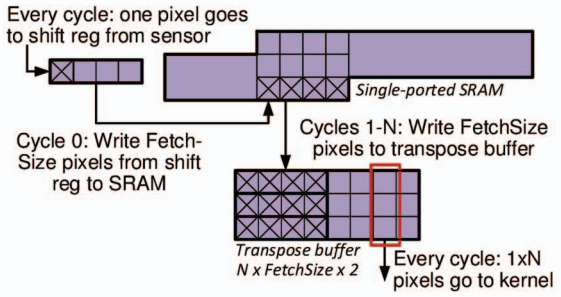

Fig. 3. Single-SRAM line buffer architecture and operation showing how a wide access singled ported SRAM can be used as a line buffer. By fetching M pixels each access, we only need to fetch each row every M cycles. If $M > N$, we have enough cycles to fetch all the rows needed and write the new data into the memory using a single port.

reduce the line buffer area. A comprehensive treatment of the tradeoffs involved with stripping can be found in Brunhaver's thesis [34].

Figure 2 shows the detailed operation and construction of a line buffer feeding an N×N kernel. This simple line buffer consists of N-1 dual-port SRAMs, $SRAM_1$ to $SRAM_{N-1}$, each containing an image line; and N shift registers, SR_0 to SR_{N-1} holding N pixels each. During each cycle, the line buffer receives one input pixel. The incoming pixel goes to SR_0, which represents the newest row of the image, while the remaining shift registers SR_i $(0 < i < N)$ are fed by reading one pixel each from $SRAM_i$. This feeds a new column of pixels into the shift registers, so the needed N×N data window for the kernel is provided by these shift registers. As the new column of pixels is stored in the shift registers, the column of pixels shifted out of the registers is written back to the SRAM, but shifted by one row: each shift register SR_i is written into $SRAM_{i+1}$. This row shift moves the data into the right location for computation of the next line in the image (the data from the top line, SR_{N-1}, is not needed any more and is dropped).[1]

While simple in construction, the multi-SRAM line buffer of Figure 2 has high area (dual-ported SRAM) and energy overhead (narrow access width). A more compact and efficient line buffer can be constructed from one wide single-ported SRAM and a "transpose buffer," as shown in Figure 3. The SRAM in this line buffer always reads or writes M pixels per cycle, where M ≥ N. The incoming pixel is stored in an M pixel shift register. Out of every M cycles, a) data is read from the SRAM for N-1 cycles, one access for each row that is stored in SRAM; b) the incoming shift register is written into the SRAM during 1 cycle; and c) in the remaining (M - N) cycles, the SRAM is idle and gated off to save power. The transpose buffer is a cyclic buffer that writes data into it row wise, but reads data out column wise. Each cycle, a column

[1]If you are willing to put a crossbar between the SRAM and the shift registers, you don't need to move the data between the SRAMs. Instead, since the oldest pixel (top-right in the stencil in the diagram) is no longer needed, the newest pixel (bottom-right) goes from SR_0 to $SRAM_{N-1}$ for later usage, over-writing the dropped pixel. After completing the line, use an N×N crossbar to "relabel" the SRAMs ($SRAM_i$ becomes $SRAM_{i+1 \bmod N}$), so there is only one pixel written independent of the size of the input stencil.

of pixels is read from the transpose buffer to the stencil shift registers (not shown in the figure).

For the CGRA, we created a small line buffer building block using this efficient approach, and can combine these blocks to form larger LBs. Each LB block contains a 16kB SRAM that can hold two rows of 16 bit single-channel data from a 4K-wide image, natively supporting stencil heights up to 3 pixels. Multiple-channel pixel support is handled by using additional LB blocks in parallel, operating in lock step with each other. To implement an LB with larger height we chain the LB blocks together like we chained the SRAMs in the simple line buffer model. The input data is fed into the first LB block. The pixel of the output column that is going to be dropped is fed as the input of the next LB block, in addition to being sent to the kernel. With this arrangement, the newest two lines come from the first LB block, and the next two lines come from the next LB block, etc. Chains can be of arbitrary length. For example, a 5×5 kernel would chain two such blocks.

The base architecture for our CGRA implementation replaced some columns of block RAMs with LB blocks. To estimate the area/energy of these blocks, we used an ASIC flow to implement the control logic and transpose buffer that are needed in addition to the SRAM. As expected, SRAMs dominate the area and energy of the LB blocks; the added logic adds just 10% area and 15% energy overhead over the SRAM cost.

This structure was not used for the SIMD machine. Since the SIMD processes a kernel over multiple cycles, the SIMD line buffer does not need to read or write pixels every clock cycle. This reduced memory bandwidth, combined with intrinsic wide fetches, made scratchpad memory an efficient line buffer, and the register file served as the transpose buffer. Similarly, this approach has no LB-specific control logic, so the energy overhead associated with this "line buffer" is simply the energy used by the SIMD's load and store instructions. Thus for SIMD, the line buffer energy is easy to estimate from the memory energy. Because the scratchpad is part of the SIMD base architecture, it has no extra LB-specific area overhead.

IV. SIMD Architecture Overview

Like most compute engines for highly data parallel applications, we use a SIMD architecture to amortize the instruction overhead over multiple data elements processed in parallel, and choose a SIMD width to make the resulting instruction overhead small. We parallelize over the image data, so if we represent the imaging application as a directed graph of operations, each SIMD lane processes this graph for its pixel. Like many others we use VLIW techniques to increase performance by enabling multiple ops to be processed each cycle, which also enables overlap of data movement (memory load/store) and computation.

Our SIMD fetches an instruction from the instruction RAM, decodes it, and broadcasts control signals to multiple execution lanes (see Figure 4). Each lane contains some local storage (register file), a processing element (PE), and a port to the memory hierarchy. There are no inter-lane communication ports. A load-store unit for the SIMD engine fetches wide data from the system memory, splits it into the size for each lane, and loads it into the register file, or else it takes the data from the lanes, aggregates it and stores it into the memory. This unit contains a double wide register allowing data from two memory fetches to be concatenated together, and then a funnel shifter which can extract any contiguous vector from this register to present the needed data to the right SIMD lane. This shifter is used to correctly align memory data for the SIMD engine.

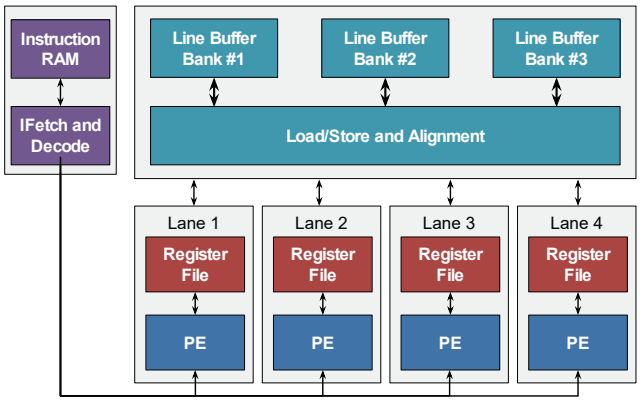

Fig. 4. Programming-in-time (SIMD) architecture: data and control flow for a single core in a multicore system. Line buffer banks are used to 'strip' the image across multiple cores.

To explore the energy/area efficiency of this type of architecture we use a SIMD engine with 32 16-bit-wide independent lanes which, as we will show later, is enough to reduce the effect of instruction overhead to 14%, while not causing too much inefficiency with the block size. It has a conventional seven stage execution pipeline with variable length VLIW instructions. The base architecture has a limited VLIW extension, allowing the overlap of load and compute ops, and optimally encoding instructions that do not use both VLIW slots. Our functional unit supports 16-bit integer arithmetic (including multiply), logical and shift operations. It has two inputs and one output. The register file is initially sized to accommodate the largest active working set among all applications, while the instruction RAM is sized to accommodate the largest application. (Application and sizing details can be found in Sections VI and VII respectively.) The application DAG is scheduled using a depth-first scheduler [35].

We use fine grain vectorization, i.e. adjacent lanes operate on adjacent pixels. This scheme can handle many different image widths, which is critical when we need to use multiple SIMD cores. The down side of fine grain vectorization is that the data needed by adjacent lanes overlaps for filter convolution. We can supply overlapping input data by storing multiple copies of the data in the register files or by sharing data across lanes. While, as we will see later, the number of registers is an issue, the shareable input pixel values form a small subset of the active data set. Thus removing duplicated values would not significantly decrease register access area or

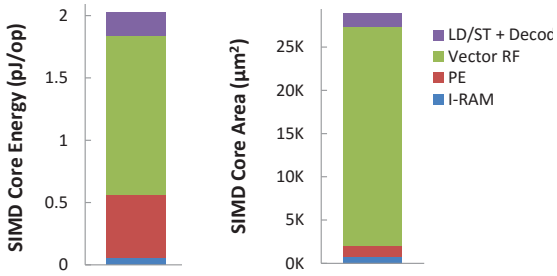

Fig. 5. SIMD area and energy components before architectural optimizations. The vector register file dominates both metrics.

energy. Since our LD/ST unit allows us to duplicate the data without increasing the number of memory fetches, and the cost of even adjacent inter-lane sharing is greater than the cost of a register file fetch, we duplicated data in the register file and do not support inter-lane communication. This decision also simplifies the compiler and the physical design of the hardware.

We implement the line buffers as blocks in a large scratchpad memory. The access width of the scratchpad is equal to the SIMD vector width. We read the data that the kernel will operate on, one SIMD vector at a time, from the scratchpad memory into the load store unit. The load store unit funnels the incoming data to create multiple shifted vectors, which are then loaded in the register file, one at a time. The loading of K vectors along the row of a K×N window triggers only two LB reads and K shifts in the load store unit. A K×N stencil gets stored in the register file as KN entries.

We implemented the SIMD engine described above using TSMC's 40nm technology. Figure 5 shows the area and energy breakdown for this base SIMD architecture. Next, we describe some optimizations used to improve the base architecture.

A. Hierarchical Register File

Figure 5 shows that, in our SIMD base architecture, the energy and area were dominated by the register file, which was sized to accommodate the largest active data set. Clearly some memory hierarchy is needed. We split the register file into two levels, RF and L0. RF, the lower level of the hierarchy, has multiple read-write ports, connects to the PE, and stores data to be utilized in the near future. This RF is built using flip-flops and muxes. The L0 is implemented using single ported SRAM to reduce area, increasing compute density. Given the static nature of the compute graph for these applications, the compiler can determine the data movement between the L0 and the RF during compile time. Because of this static scheduling, we can use SRAMs with wide I/O to amortize access cost across all lanes.

Our statically scheduled VLIW machine handles data movement without any performance overhead, overlapping the data movement with computation. The extra slot in the instruction does increase the code size and thereby the instruction fetch energy cost, however this impact is minimal because it is amortized across all SIMD lanes. To get the greatest area saving, we want to reduce RF size as much as possible. The lower bound for RF size is a function of the L0 bandwidth. As we reduce the size of the RF, the data movement to/from the L0 level increases rapidly and eventually overwhelms the L0. We chose the smallest RF size that was within the L0 bandwidth limit. The final design is discussed in the results section.

B. Other Optimizations

Introducing a single level of memory hierarchy reduces the total SIMD area considerably. Still, L0 area dominates (see results section). With this large fixed memory area, the only way to improve compute density is to put additional functional units in each lane and try to schedule multiple operations per cycle. These functional units can be controlled either by adding extra VLIW computation slots and scheduling multiple independent operations per cycle, or by fusing statistically prevalent dependent operations into a single new operation. Of course the higher computation rate will increase the required L0 bandwidth. The VLIW approach also necessitates an increase in the number of RF ports and requires availability of independent operations to schedule in each cycle. If the independent operations are temporally far apart, this approach increases the active working set of the application, which either increases the RF size or the L0 bandwidth. On the other hand, the fused-operations approach does not affect the working set of the code. While supporting fused instructions requires additional register ports, it generally needs fewer additional RF ports than VLIW.

We implemented and scheduled ISPs using both approaches to measure their energy and area efficiencies. For the VLIW approach, we extended our baseline two-slot VLIW by adding a third functional slot. This required a doubling of the register file ports, and the addition of another PE. For the fused operations approach, based on our analysis of the application graphs, we added an extra fused-add-subtract unit at the tail of the existing PE. This unit, when enabled in the instruction, takes one of its operands from the result of the PE and the other operand from a newly added register file port, and overrides the PE output. The results are shown in Section VII.

C. Multicore SIMD Implementation

The optimizations so far have focused on improving SIMD compute density and energy efficiency. However, the throughput of a single SIMD core is much less than the throughput of our CGRA and ASIC architectures. To process demanding applications at full frame rate, and for a fair architectural comparison, we need to increase the SIMD throughput.

The most obvious way to improve performance is to implement a multi-core SIMD system. We can divide the image into multiple "strips" and allow each core to process a strip. The downside of this approach is that the overlapping region at the strip boundaries must be read multiple times. However, with this approach, all cores follow the same schedule, and there are no inter-core dependencies. This makes the scheduling and implementation easier. The line buffer is divided into banks, with each bank storing data for a single core. The number of pixels stored in a bank is equal to the strip width. A core can

access up to K (K=3 in Figure 4) banks to its right, in order to support kernels with width greater than the strip width and to allow the kernel to access its neighbor's strip for data in the overlap region.

Since the schedule for all cores is identical, the read and write addresses of all banks are identical and change in lock-step, avoiding any memory contention. The area overhead in this arrangement comes from additional muxes, and increased line buffer area due to higher bandwidth requirement. The energy overhead is mainly due to re-fetching pixels from the overlap region across multiple SIMD machines. For example, as an extreme case, if the kernel width is equal to the width of two strips, the data in each bank will be fetched twice: once for its own core, and once for the core to the left. This might seem excessive, but if the line buffer energy is not a significant component of the total pixel energy, the overhead is small. In our experiments the per-pixel energy increased by a maximum of 10% due to multi-core implementation. The area overhead was less than 5%.

V. CGRA Architecture Overview

Our CGRA architecture is based on a standard FPGA design, allowing us to use VPR [36] for our research. VPR is a set of tools that performs mapping, placement and routing for FPGA-like designs. It also provides a fair way to compare our results to a reference FPGA (based on Stratix IV) in terms of performance, area and energy/power [37].

Like an FPGA, our CGRA uses an island style [38], [39] organization where a chip consists of a certain number of tiles and each tile has three kinds of blocks:

- Processing Element (PE), also known as the logic box, responsible for performing functions on the inputs.
- Switch Box (SB), responsible for implementing connections between any two tiles and connecting the PE output to other tiles. Such connections are done using wire segments, and the SB enables building a longer segment out of two shorter ones.
- Connection Box (CB), responsible for connecting the PE inputs to the wires passing through the SB.

Tiles live on a 2D grid and connect to each other with a certain number of tracks of wire segments. Like most FPGAs we use a simple mesh connection topology.

Our architecture consists of two kinds of tiles, memory and compute. Compute tiles (PEs), shown in Figure 6, are homogeneous and consist of ALU, MULT and LUT blocks, with two 16-bit bus inputs and one 1-bit input. All inputs are connected to the functional elements through an optional register. This register can be used for pipelining or storing constant inputs to the PEs. A PE has one 16-bit output and one 1-bit output. Additionally, there are a few global control signals which are not shown in the figure, like *clock*, *clock enable* and *reset*. The line buffer blocks are used for the memory tiles. Like most FPGAs, the memory and compute tiles are placed in columns in the final layout.

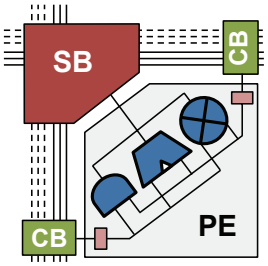

Fig. 6. How PEs connect to programmable wires in a CGRA tile. The PE consists of registers on the CB outputs, LUTs, adder and multiplier. It also contains internal muxes and some other logic. The solid lines represent 16 bit buses and the dashed wires represent individual wires.

A. Interconnect architecture

All connections in the application's DAG can be separated into two groups of signals: 16-bit buses and 1-bit signals. For both groups we use identical routing architecture composed of switch boxes, connection boxes and wire segments. We use heterogeneous length wire segments in the wiring tracks, and measure segments in terms of the number of tiles that they span. In our architecture roughly 50% are length 1 wires, 20% are length 4 and 30% are length 8. We explored routability using a VPR-based method [40] and found that this combination of lengths works slightly better for our benchmarks.

A key parameter for CGRA is the number of tracks that pass through a tile. The larger this number, the more area (and energy) the SB and CB will take, but in return it allows greater flexibility in connections. If the number of tracks is too low, an application can become unroutable.

We set the number of tracks at 12 for both bus and bit-wide connections in both horizontal and vertical directions. We use unidirectional wire segments (a common practice in FPGAs) meaning that there are six 16-bit buses plus six 1-bit wires entering each tile from four sides and six buses plus six wires exiting from each side. We picked 12 because it was enough to route any application from our benchmark set (most of them need less).

Our switch boxes use a Wilton pattern [41] with Fs=3 to change track direction, and they consist of 4:1 MUXes, one per output wire segment. This means each incoming bus/wire to a switch box can only leave on one track in each output direction. The MUX is 4:1 because each output can come from one track in three possible input directions (this parameter is commonly called *Fs*) or from the PE's output. Remember that not all wire segments end at every switch box. Long segments route through switch boxes and don't require this MUX. For our wire length distribution, around 60% of a tile's outputs have MUXes and around 40% just pass through the switch-box. Additionally, each SB has four registers per group of signals, such that each wire (but not all wires at once!) can be optionally registered for better timing. To support our 12 16-bit track system, the SB has 14, 16-bit 4:1 MUXes (7 for the vertical and 7 for the horizontal), and 4 16-bit registers. The SB for our 12, 1-bit signals has the same architecture.

A CB allows an input of a PE to be connected to nearby

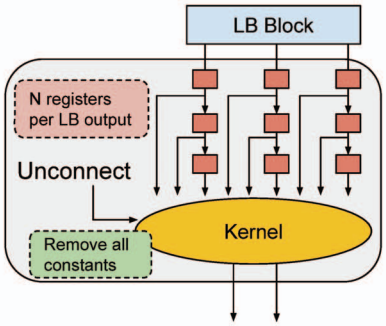

Fig. 7. Application mapping. The compiler builds kernels and line buffers, which we process and map onto the CGRA hardware as shown.

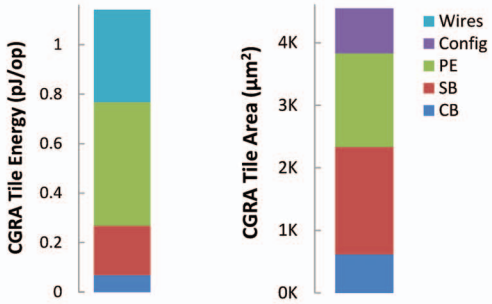

Fig. 8. CGRA area and energy components before optimization.

tracks. Only tracks that are next to a tile connect to the CB. As shown in Figure 6, the CB on the right of the tile connects to the vertical tracks, while the CB on the top of the tile connects to the horizontal tracks. In order to reduce the area of a CB in FPGAs, CBs connect only to a subset of nearby tracks, a technique called *depopulation.* Depopulation exploits inherent redundancy in connectivity but it also makes routing harder; also, the right degree of depopulation is not obvious. In our case, we get the benefit of depopulation for free by using buses and doing bus based routing instead of routing each bit individually. This technique allows us to have each CB connected to all tracks. Our PE has two bus inputs and one 1-bit input; this translates to two bus-based CBs each having a 12:1 16-bit MUX, and one single-bit CB with a 12:1 MUX.

B. Extra elements

Besides the MUXes, the SB in each tile will have some number of signal repeaters for signal regeneration of long wires that span multiple tiles and buffers. We account for these buffers in energy and area calculations without explicitly mentioning it.

Another "hidden" element is the *configuration*; this storage holds the programmed select values for all the CB and SB MUXes, plus opcodes, MUX selects, and register enable bits etc. for the PEs. Overall, each tile needs 12 bytes of storage for configuration, which we model with latches. Note that this is additional storage needed for configuration; the storage for the LUTs has already been included in the PE area.

C. Mapping DSL application to CGRA

To map an application from DSL to CGRA, our compiler first converts it to a DAG of LBs and kernels, where each kernel is in turn represented by a DAG of operations, as shown back in Figure 1. Next we map the application's line buffers into our LB blocks, grouping/chaining them to create buffers of the correct size. Since our LB blocks output a column of pixels each cycle, the final pixel shift registers are built from the registers in the PE blocks. We pulled the final shift registers out of the LB blocks to increase the locality of this highly used data as shown in Figure 7. The kernel's operations are then mapped to the CGRA PE blocks, building the configuration of blocks and connections needed to create the desired kernel DAGs.

In this mapping process we move constant storage (parameters that don't change during invocation, like filter parameters) close to where it is used. When mapping a kernel, we trim the connections which route constants, and place the constant in a local register to reduce the energy and area of the resulting implementation.

Now we have a complete representation of the application in terms of elements that have to be placed in the CRGA architecture. We use VPR for placement and routing, optimizing for the delay between connected nodes in the design, which is identical to minimizing total wire length. VPR doesn't support bus based routing, so we represent all 16-bit buses with specially marked single bit wires. Even though buses and bits become indistinguishable, the routing result is still valid because we use identical interconnect resources for these two groups.

D. CGRA Optimization

Running VPR provided the energy and area of our base CGRA architecture, shown in Figure 8. While the energy/op is good, the architecture suffered from long critical paths, which limited maximum operating frequency and affected area efficiency. The easiest way to solve this is through pipelining, since each kernel is a DAG, which can be pipelined to any depth without breaking functionality. Unfortunately, pipelining the PEs didn't help (it was in the base design) because the long delay was caused by long wires. Like some recent FPGAs, we added registers to the SBs to allow us to pipeline the wires and create a better design. The challenge was to fit it into the existing design flow.

To pipeline the design we take the routed output from VPR, which has the X,Y coordinates of each CGRA element, along with a list of the SBs used to make each connection in the design. From this data we extract the original compute graph, annotated with the SBs used to form each connection along with the length of the wires between these switch boxes. Based on this information, we use static timing analysis to find where pipeline registers are required to achieve our target clock frequency. This analysis includes the delays of all SBs along the path, plus CB and source PE; these delays are taken from the synthesis report of corresponding blocks implemented in

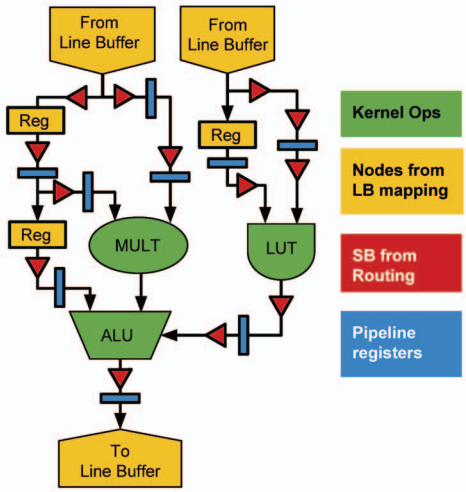

Fig. 9. Data flow for a kernel after it has been placed and routed on the CGRA architecture and then pipelined. The registers are the architectural delays from the application, and the thin blue boxes are the registers added from pipelining. Notice that the total pipeline delay added to each input of a functional block must be the same.

	Shift	Mux	Logic	Div	ALU	Mult	Complexity
Harris	14	1	4	0	51	20	86
Fast	1	1	351	0	50	3	77
ISP	129	198	41	0	584	61	975
FCAM	289	1285	198	4	3313	302	5205
Stereo	0	65	0	1	9683	1	9750

TABLE I
APPLICATION COMPLEXITY AS MEASURED IN BASIC OPERATIONS. "COMPLEXITY" IS THE SUM OF ALL OPERATIONS WEIGHTED BY BIT WIDTH AND NORMALIZED TO 16 BITS.

Synopsys PT. Lastly we add additional registers to balance the cycle delay of each input and guarantee that functionality is not broken, as illustrated in Figure 9. The pipeline registers are implemented from the registers that were added to the SBs, so adding them doesn't change the routing information. After pipeline insertion, we have a complete view of the implemented application with all the blocks required to estimate its energy.

We found that pipelined wires greatly improve area efficiency of the CGRA by increasing its clock frequency 2-11x and making it the same as the SIMD machine. This resulted in about 10% increase in tile area and between 2% and 10% decrease in the energy efficiency.

VI. Evaluation Framework

The previous sections presented the SIMD and CGRA architectures and discussed the optimizations we used to improve their energy efficiency (pJ/operation) and compute density (operations/mm^2). To derive these metrics, we ran the same applications on each architecture and tracked the activity of the hardware blocks during execution, e.g. RF, L0, PE and memory accesses for SIMD, and bus transitions for CGRA. We then combined this activity data with a table of the actual energy costs of the operations extracted from simulating block implementations.

Table I lists the applications used and their complexity in terms of how many operations produce one pixel of result. Harris [42] and Fast [43] are corner detectors, where Fast uses a much simpler algorithm and hence has faster runtimes. ISP and FCAM are both complete image processing pipelines. ISP is a generic imaging pipeline that performs demosaicing, white balance, color correction, crosstalk correction, dead pixel suppression, and black level correction. FCAM is a more sophisticated version of this pipeline that does various levels of noise reduction, as well as higher quality (and more computationally intensive) versions of the other steps. Stereo [3] implements maximum similarity detection within a linear search distance. A prior paper describes some of these applications in more detail [8]

The applications were chosen to stress different architectural components. For example, the Harris corner detector is dominated by line buffer accesses while Stereo is computation dominated. Fast contains many single bit operations and benefits from an FPGA architecture, which can pack many such operations in a CLB. The large operation count of Stereo (two orders of magnitude larger than Harris) stresses the VPR mapper and the depth-first scheduler. Stereo also demands the highest amount of architectural resources like PEs, routing tracks and register file entries, and it lets us measure metrics at the upper utilization bound. ISP and FCAM are examples of "typical" applications that run on ISPs.

These benchmarks, written in the Darkroom DSL [8], were compiled to an intermediate representation (IR) which consisted of multiple kernels (filters) connected to produce a final image, plus descriptions of the DAG for each kernel as was mentioned earlier. The DAG description consists of nodes doing simple operations like ADD, AND, MUX and MULT that can be scheduled on PEs.

On the SIMD engine, we scheduled each kernel's DAG in a depth-first-search manner (keeping any intra-kernel loops intact), and counted the number of clock cycles, register file accesses, instruction memory accesses, and load/stores needed to generate an output pixel. For CGRA, we used VPR to map the completely unrolled DAGs onto a homogeneous CGRA-tile array with configurable routing. We counted the number of tracks being used in each channel, the number of live buses in each CGRA tile, and the minimum number of CGRA tiles needed to fully route the DAG. In addition to predicting usage for computing energy, these metrics also enabled us to design the architectural components.

We implemented the architectural components as Verilog models and placed and routed them using industry standard tools for the TSMC 40G (40nm) technology node, to determine performance and area. To estimate energy consumed, we simulated the routed netlists by applying uniformly random data to the data inputs while keeping the control signals constant at their expected values. This gave us toggle counts and, using the parasitics from the routed netlists, Synopsys' Primetime-PX tool estimated gate and wire energy. We compiled the memory instances used in the architectures with ARM's single port SRAM memory compiler and derived their performance, area, and access energy from the resulting compiler-generated datasheets.

SIMD energy and compute density

For SIMD, we assume each IR instruction requires an instruction fetch i, plus one or more data fetches r from the register file, plus execution of an *op* in the PEs, and possibly a line buffer fetch L. We counted each of these events while running the scheduled code to get totals n_i, n_r, n_{op} and n_L respectively. Then, using the per-event energy found by simulation, and given time t_{app} for the app to run, we calculated energy E and compute density C

$$E = (n_i e_i + n_r e_r + n_{op} e_{op} + n_L e_L)/(n_{op})$$
$$C = (n_{op}/t_{app})/(\textit{total area})$$

CGRA energy and compute density

For CGRA, each IR instruction is mapped to a tile. The tiles are then placed and routed using VPR to minimize tile array size while ensuring complete connectivity. We obtained the number of live buses in every tile and the number of active and passive tiles from VPR logs (tiles that have been mapped to an IR operation are active tiles, while tiles that are used only for routing are passive tiles). We simulated tiles with different numbers of live buses and different PE mappings. We thereby got the energy of CGRA tile components for different activity levels. Using these energy numbers and VPR statistics, we derived the energy and compute density of each CGRA application in the same manner as with SIMD.

FPGA energy and compute density

For FPGA comparison we used the *k6-frac-N10-frac-chain-mem32K-40nm* architecture from VPR's distribution. This allowed us to use the same set of tools for placement and routing as we used for CGRA. Like CGRA, this is a 40nm part, and the model comes annotated with timing and energy information, providing a good comparison. It has 6-input, 2-output LUTs with carry chains for fast arithmetic and hard DSP and RAM blocks, all of which we use in our designs.

However, our reference FGPA doesn't support pipelined wires. This was a very important feature for getting good area efficiency in our CGRA, so for fair comparison we estimate its effect on the baseline FPGA. To do this, we generate FPGA area and energy numbers using VPR, then increase the SB area by the area of the pipelining flip-flop assuming 1 pipeline register per track. For energy, we assume that the clock is running at peak frequency and adjust dynamic power from VPR's report by the ratio of peak frequency over frequency achieved by VPR. Finally we increase the energy by the same amount as "pipelining" in CGRA. While this energy adjustment is a crude approximation, given the small energy cost of pipelining, the resulting error will be small.

For each application, we ran a design with no initial pipeline registers, and one that was pipelined. These designs' dynamic energy varied by up to 30% even after accounting for the energy of the flops, so we took the lower energy as our starting point and added the pipeline energy to that.

ASIC energy and compute density

For the ASIC comparison we translated each application's graph directly to Verilog and used standard synthesis and back-end tools to generate the physical implementation. Area, performance and energy numbers were extracted from this implementation. The ASIC designs used the same SRAM library as the other implementations, but could use memory sizes that were optimized for that particular application. These designs did not worry about being compatible with a range of applications.

	CGRA	SIMD	FPGA+pw	ASIC
Harris	1.38	2.08	3.69	0.53
Fast	3.91	6.27	3.19	0.52
ISP	0.91	1.68	1.12	0.20
FCAM	1.04	1.41	1.42	0.15
Stereo	1.30	1.94	3.19	0.13
Average	1.71	2.68	2.35	0.31
Peak	0.38	0.76	1.35	0.03

TABLE II
ENERGY PER OP FOR THE SELECTED IMAGING APPLICATIONS ON EACH ARCHITECTURE, IN PJ/OP.

VII. RESULTS

We implemented the SIMD, CGRA, ASIC, and FPGA architectures, as explained in the methodology section, using our initial estimates of architectural parameters. Based on the feedback from these initial experiments, we optimized the architectures using techniques mentioned earlier in the paper. In this section, we present our analysis and results for efficiencies of various architectures and architectural improvements.

A. Energy/Op

Table II shows the energy per op for each architecture. To separate out structural overhead, versus the overhead caused by communication, we include a "peak" energy efficiency number in this table. The peak number gives the energy cost of a single local operation for each of the different architectures, i.e. a not-to-be-exceeded number that would happen only if the application consisted only of 16-bit adds with locally-generated inputs. So the ASIC peak number is simply a 16-bit adder running at 1 GHz. Peak CGRA efficiency reflects one tile doing 16-bit adds at 800 MHz with data coming from adjacent units. SIMD peak efficiency is calculated for a single core with 32 lanes performing 16-bit adds at 800 MHz where two vectors are read while one is written to the register file. FPGA peak energy efficiency is modeled as a single CLB doing 16-bit add at 741 MHz counting the energy of CLB, SB and CB.

The peak numbers show the large energy overhead of creating a programmable system with flexible communication, even with locality. The lowest possible energy/op for all programmable platforms uses over 10x more energy than ASIC, which only contains the energy of the adder. The buses, muxes, registers, etc. that you include to make the adder part of a PE all add overhead to its fundamental operation. While this overhead energy is not large, the add energy is very small, so the overhead energy dominates it.

Since the CGRA and SIMD machines use the same basic function-unit design, their energy difference is caused by the cost of the register file relative to the connection box cost. This

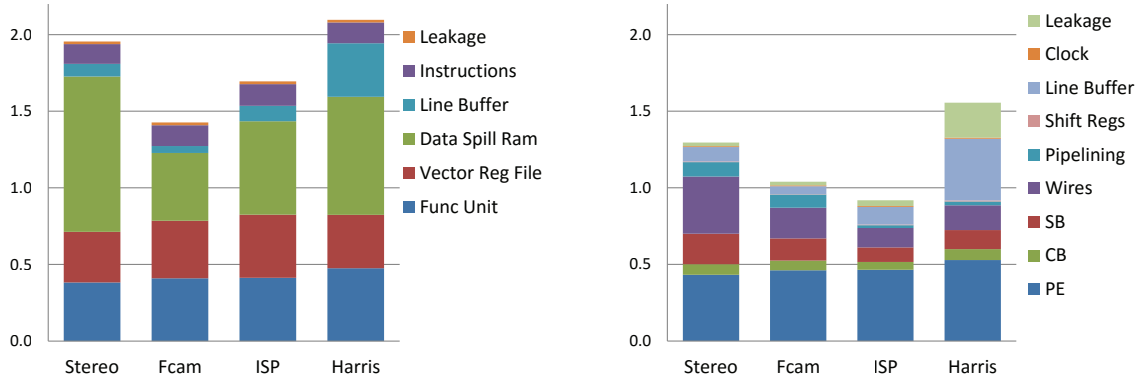

Fig. 10. Energy breakdown in pJ/op for the selected imaging applications (SIMD on the left, CGRA on the right).

	CGRA	SIMD	FPGA+pw	ASIC
Harris	.013	.013	.039	.0046
Fast	.034	.046	.032	.0038
ISP	.007	.010	.016	.0011
Fcam	.006	.009	.017	.0006
Stereo	.006	.008	.024	.0004
Average	.013	.017	.026	.0021
Peak	.005	.008	.012	.00009

TABLE III
AREA EFFICIENCY FOR SELECTED IMAGING APPLICATIONS ON EACH ARCHITECTURE, IN MM^2/GOPS, INCLUDING LINE BUFFER AREA.

difference makes sense, as the SIMD register file explicitly does some amount of communication that the CGRA must do with longer with wires. Thus while the peak SIMD:CGRA ratio is 2:1, on average SIMD takes 1.6 times the energy of a CGRA implementation. The energy difference between an FPGA with pipelined wires and the CGRA is mostly caused by slightly longer wires in the FPGA and less efficient functional units, so FPGA takes 1.4 times more energy than CGRA.

While the line buffer energy is not a significant factor in the programmable solutions, it is important in ASIC implementations. For applications like FCAM and Stereo whose kernels have significant computation between line buffers, computation energy dominates even in the ASIC implementation and the gap between CGRA and ASIC is around 10x in energy. However, smaller applications like Harris, Fast and ISP that are dominated by LB memory accesses have only around 3x gap in energy efficiency, since the line buffer energy in both implementations is similar.

Figure 10 breaks down energy consumption in SIMD and CGRA architectures. Since we use the same functional units in both designs, it is not surprising that the energy of the function units are nearly identical for the two architectures. ("func unit" for SIMD, "PE" for CGRA). The difference in total energy comes from the different cost of communication. In SIMD this is the combination of the register file and spill RAM energy and averages over 1 pJ/op, while the combined cost of the wires, switch box and connection box in the CGRA is around 0.5 pJ/op. Note that in both architectures the "overhead energy" is generally low.

B. Area per Op/s

Table III tells the required area/GOPS for the different implementation approaches, with the peak line again giving the minimum possible mm^2/GOPS. We use the same base configuration for calculating the peak compute density as energy for all the targets except for FPGAs, where we now include multipliers with the LUTs. When viewing these numbers, it is important to remember that the presence of the line buffers makes calculating compute density tricky for two reasons. First, since the LB area doesn't scale with application performance, lower performance (and hence smaller area) solutions have worse compute densities. So, since a single-core SIMD machine has lower performance and lower area than the competing architectures—e.g. for FCAM it requires 2.6mm^2 compared to CGRA's 25mm^2—to fairly compare their compute density, we used multiple SIMD cores to create solutions with matching performance.

The second issue is the size of LB that is included in the area cost. In an ASIC solution, you build the size LB that its single application requires. For a programmable solution, you provide sufficient resources to solve the entire range of problems it might see. Thus for most applications, the programmable engine has surplus memory that it doesn't use. This extra memory further decreases the compute density. So, while the actual hardware contains a maximally-sized LB, Table III calculates a per-application area efficiency using only that portion of the LB that was required for each application, with the assumption that a user generally tries to put as much of the computation on the accelerator as will fit: it is unlikely that only Harris would use the accelerator, so when the accelerator is actually deployed in a real system, other applications would be using the part of the LB that that the Harris application didn't need.

To quantitatively address these issues, Table IV provides the area of an LB block in each of our programmable architectures, and the size LB needed for each application. It also provides the GOP rate of each application. Together this data allows one to compute the area/GOPS overhead that is caused by the LB (for the size LB you choose to use). In addition we give the mm^2 per GOPS for the hardware excluding the LB, so adding the two terms together generates the data in

	Area efficiency, mm²/GOPS			*LB size, rows*	*Perf, GOPS*
	CGRA	SIMD	FPGA+pw		
Harris	.007	.008	.022	8	69
Fast	.029	.041	.018	6	62
ISP	.005	.009	.012	20	780
Fcam	.005	.008	.015	50	4164
Stereo	.006	.008	.022	70	7800

	Size of 16kB LB in mm²			*16kB LB, # rows*	-
	CGRA	SIMD	FPGA+pw		
	.104	.094	.266	2	-

TABLE IV

1) AREA EFFICIENCY WITHOUT LINE BUFFER, MM²/GOPS. 2) LB SIZE REQUIRED PER APPLICATION, IN NUMBER OF ROWS. 3) APPLICATION PERFORMANCE AT 800MHZ, IN GOPS. 4) PER-ARCHITECTURE LB SIZE (BECAUSE OF PER-ARCHITECTURE DIFFERENCES IN CONTROL LOGIC AND OVERHEAD, LB HAS DIFFERENT AREA IN EACH ARCHITECTURE). 5) A 16KB LB HOLDS TWO ROWS OF PIXELS, ASSUMING 4K IMAGE WIDTH.

SIMD lane		**CGRA tile**	
Func Unit	1313	PE	1327
Vector Reg File	1405	SB	1517
Data Spill Ram	2681	CB	545
Iram	875	Configuration	633

TABLE V

ABSOLUTE AREA BREAKDOWN COMPARISON FOR AN OPTIMIZED SIMD LANE AND AN OPTIMIZED CGRA TILE, μm^2.

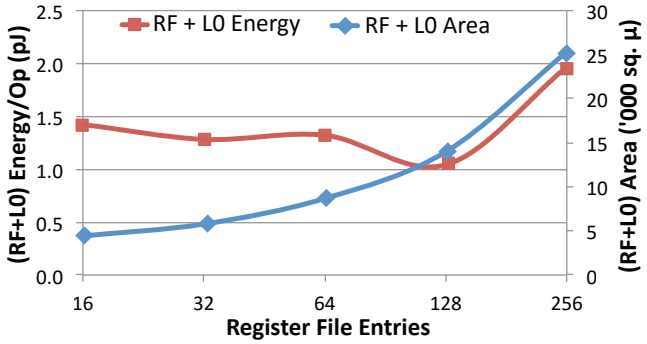

Fig. 11. Total reg file (RF+L0) area and access energy for different RF sizes.

App	Peak op/cy	L0 bandwidth	Feasible op/cy	Area eff. improve
Harris	1.82	0	1.82	29%
ISP	1.96	99%	1.96	38%
FCam	1.98	138%	1.43	1%
Stereo	1.99	238%	1.10	-23%

TABLE VI

FEASIBILITY OF CO-SCHEDULING OPS CONSIDERING LIMITED L0 BANDWIDTH AND THE CORRESPONDING COMPUTE DENSITY IMPROVEMENT.

Table III.

For example, in Table IV we see that Harris uses about one tenth of the LB that Stereo requires. CGRA compute hardware for Harris uses 0.007 mm²/GOPS, so adding just the LB it needs (8 rows or 4 LB blocks) adds 0.42 mm² / 69 GOPS or 0.006 mm²/GOPS yielding 0.013 mm²/GOPS, which is the approximately the number in Table III. Yet if you add the area for the full line buffer, the LB area increases by almost 10x, and the area/GOPS increases to 0.6 mm²/GOPS. This means for Harris the programmable solution is either around 3x or 12x the area of the ASIC solution depending on how you account for the LB area. Since the LB is only important for the simpler applications and depends on interpretation, the rest of this discussion will focus on the compute density excluding the LB.

Table IV shows that Fast is an outlier since it is dominated by binary operations. Neither SIMD nor CGRA was optimized for this type of operation (even though they could be) so their compute density falls by about 5x. The table shows that CGRA and SIMD compute density are not far apart, being on average only about 40% different (excluding Fast), i.e. SIMD is taking about 40% more resources than CGRA and thus has about 40% lower compute density. The reason for this difference can be seen in Table V. Again, the area of the functional units is nearly the same in the two designs, but the area used for the programmable wires in the CGRA is less than the area required for the memory in the SIMD, even after the register file has been optimized.

Both SIMD and CGRA initally had much better compute density than FPGA, largely because of the difference in clock rate. Pipelining the CGRA wires and functional units was essential to reach 800 MHz. Adding wire pipelines to the FPGA greatly improved its area efficiency, but it is still 3.1 x worse if Fast is excluded (see FPGA+pw in Table IV), because LUTs are less efficient than ALUs for addition and because its hard macro blocks are not as well tuned to the application domain. The FPGA memory is dual ported and not as wide as our LB blocks, and the DSP block is a multi-precision 36-bit unit.

C. SIMD Compute Density Optimization

To optimize the SIMD architecture, we partitioned the register file into an RF and an L0, as explained earlier. For our set of imaging applications, the RF needed a minimum of 16 entries to keep the L0 traffic below one word/cycle bandwidth, so we used that as the base RF size. The corresponding L0 size is around 200 entries. The largest active data set belongs to Stereo, which set the total (RF+L0) register file size. The partitioning reduced energy per register-file fetch by 25%, and total register file area by 80% (Figure 11). Although the total register file area reduced considerably, the L0 portion still dominated with 42% of the total SIMD area ("Data Spill RAM" in Table V). So, to improve the compute density further, we tried extended-VLIW and operation-fusion to better amortize the L0 area, techniques that were discussed in the SIMD section.

The extended VLIW approach, i.e. extending VLIW width from two to three, increased total SIMD lane area by 42% because of the additional register file ports and PE. We modified our scheduler to put independent operations in the newly added functional slot, and it found independent operations to co-schedule almost for all cycles. However, because of the nature of the application graph, the distance between such independent operations was large. Co-scheduling such operations increased traffic to the L0 level, overwhelming its

bandwidth for certain applications. In the best case, we could increase the number of operations scheduled per cycle from 1.0 to 1.96 without overwhelming the L0 bandwidth, increasing the effective compute density by 38%. However, the VLIW extension decreased the effective compute density by 23% for Stereo because the additional slot area could not be offset, as the L0 bandwidth limited the slot utilization. Table VI lists the applications, the peak operations per cycle possible without considering L0 bandwidth, the highest feasible operations per cycles possible without saturating the L0 bandwidth, and the effective computation density improvement.

Meanwhile, the fused-operation approach increased total area by 11%. This overhead is significantly less than that for the VLIW approach because only one RF port was added for fused operations, as compared to four for VLIW. Also, the tail unit added for fusion supports only add and subtract operations, as opposed to the fully functional PE needed for VLIW. By suitably enabling the unit, we could increase the number of operations executed per cycle by a maximum factor of 2x for Stereo, giving a compute density improvement of 80%. Improvements for other applications were smaller. By combining the VLIW and operation fusion techniques area efficiency got better, on average, by 30%.

Referring again to Table III, it is clear that the resulting compute density of CGRA and SIMD units is still quite small compared to ASIC solutions. Adding operand fusion and more complex functional units can improve computational density, but our preliminary results indicate that these approaches will only make modest (30%) changes in overall compute density.

VIII. Conclusions

As use of image data grows explosively, it is critical to create systems that allow programmers to quickly develop and deploy efficient image applications. Using a modern image DSL (Darkroom), we were able to fairly compare different approaches for building programmable image processors, evaluating SIMD, CGRA and FPGA approaches, and comparing them to a custom ASIC solution. To optimize the performance of our implementations, it was critical to exploit the locality inherent in imaging applications. For SIMD this meant creating a small, low energy register file, and for CGRA it meant bus based routing and pipelining the programmable wires. With these changes the SIMD and CGRA results were similar, with the CGRA able to achieve 1.0 pJ/op and 0.006 mm^2/GOPS, which is state-of-the-art for a programmable ISP. We also discovered that memory and register file overheads in the SIMD machine made it worse than the CGRA equivalent by around 60% in energy efficiency and by about 40% in area efficiency.

On the other hand, when we compare these programmable solutions to ASIC blocks for computationally intensive kernels, we see that programmability still has significant energy and area costs. This is troubling, since these kernels can dominate the total computation performed. These results explain why current programmable ISPs are often hybrid designs that contain both programmable hardware and fixed function unit, and raise the challenge of compiling code for these machines. We believe that these energy and area gaps can be reduced by making coarser grained processing elements and by implementing operation fusion, and we plan to address this issue in future work.

Acknowledgment

This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. S. Kvatinsky is partially supported by the Viterbi Fellowship in the Technion Computer Engineering Center.

References

[1] T. Georgiev, Z. Yu, A. Lumsdaine, and S. Goma, "Lytro camera technology: theory, algorithms, performance analysis," in *Proc. SPIE*, vol. 8667, 2013, p. 86671J.

[2] S. A. Shroff and K. Berkner, "Image formation analysis and high resolution image reconstruction for plenoptic imaging systems," *Applied optics*, vol. 52, no. 10, pp. D22–D31, 2013.

[3] Z. Lu, Y.-W. Tai, F. Deng, M. Ben-Ezra, and M. S. Brown, "A 3d imaging framework based on high-resolution photo- metric-stereo and low-resolution depth," *International journal of computer vision*, vol. 102, no. 1-3, pp. 18–32, 2013.

[4] P. Desai, "Choosing the right DSP for high-resolution imaging in mobile and wearable applications," http://ip.cadence.com/uploads/899/Tensilica_Vision_P5_WP_Final_100515-pdf.

[5] M. H. Ionica and D. Gregg, "The Movidius Myriad architecture's potential for scientific computing," *IEEE Micro*, vol. 35, no. 1, pp. 6–14, 2015.

[6] T. R. Halfhill, "Silicon Hive breaks out," *Microprocessor Report, Dec*, vol. 1, 2003.

[7] J. R. Kelley, C. Barnes, A. Adams, S. Paris, S. Amarasinghe *et al.*, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *SIGPLAN Not.*, vol. 48, no. 6, pp. 519–530, 2013.

[8] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144:1–11, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2601097.2601174

[9] E. Angel, "DIGIC processors explained," *Canon, Inc., http://www.learn.usa.canon.com/resources/articles/2012 /digic_processors.shtml*, January 2012.

[10] S. Komatsu, M. Kimura, A. Okawa, and H. Miyashita, "Milbeaut image signal processing LSI chip for mobile phones," *Fujitsu Sci. Tech. J*, vol. 49, no. 1, pp. 17–22, 2013.

[11] G. Martin and H. Chang, Eds., *The TI OMAP platform approach to SoC*. Springer, 2003, ch. 5.

[12] D. Witt, "OMAP4430 architecture and development," in *Technical Record of the 21st Hot Chips Conference*, 2009.

[13] M. Ditty, J. Montrym, and C. Wittenbrink, "NVIDIA's Tegra K1 system-on-chip," in *Technical Record of the 26th Hot Chips Conference*, 2014.

[14] D. Boggs, G. Brown, N. Tuck, and K. Venkatraman, "Denver: NVIDIA's first 64-bit ARM processor," *Micro, IEEE*, vol. 35, no. 2, pp. 46–55, Mar 2015.

[15] G. Stein, E. Rushinek, G. Hayun, and A. Shashua, "A computer vision system on a chip: a case study from the automotive domain," in *Computer Vision and Pattern Recognition - Workshops, 2005. CVPR Workshops. IEEE Computer Society Conference on*, June 2005, pp. 130–130.

[16] R. Bushey, H. Tabkhi, and G. Schirner, "Flexible function-level acceleration of embedded vision applications using the pipelined vision processor," in *Signals, Systems and Computers, 2013 Asilomar Conference on*, Nov 2013, pp. 1447–1452.

[17] R. Rithe, P. Raina, N. Ickes, S. V. Tenneti, and A. P. Chandrakasan, "Reconfigurable processor for energy- efficient computational photography," *Solid-State Circuits, IEEE Journal of*, vol. 48, no. 11, pp. 2908–2919, 2013.

[18] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule, "Hexagon DSP: An architecture optimized for mobile multimedia and communications," *Micro, IEEE*, vol. 34, no. 2, pp. 34–43, 2014.

[19] R. Merritt, "CES: Ceva recognizes gestures with new core," *EE Times*, January 2012, http://www.eetimes.com/document.asp?doc_id=1260890.

[20] B. Hariri, S. Abtahi, S. Shirmohammadi, and L. Martel, "Demo: Vision based smart in-car camera system for driver yawning detection," in *Distributed Smart Cameras (ICDSC), 2011 Fifth ACM/IEEE International Conference on*. IEEE, 2011, pp. 1–2.

[21] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD–reconfigurable pipelined datapath," in *Field-programmable logic smart applications, new paradigms and compilers*. Springer, 1996, pp. 126–135.

[22] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW processor with reconfigurable instruction set for embedded applications," *Solid-State Circuits, IEEE Journal of*, vol. 38, no. 11, pp. 1876–1886, 2003.

[23] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The reconfigurable instruction cell array," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 1, pp. 75–85, 2008.

[24] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and M. C. Eliseu Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Computers, IEEE Transactions on*, vol. 49, no. 5, pp. 465–481, 2000.

[25] F.-J. Veredas, M. Scheppler, W. Moffat, and B. Mei, "Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes," in *Field Programmable Logic and Applications, 2005. International Conference on*. IEEE, 2005, pp. 106–111.

[26] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "Piperench: A reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[27] 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," *IEEE Micro*, no. 5, pp. 38–51, 2012.

[28] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel *et al.*, "Triggered instructions: A control paradigm for spatially-programmed architectures," *ACM SIGARCH Comp. Arch. News*, vol. 41, no. 3, pp. 142–153, 2013.

[29] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 2011, pp. 109–116.

[30] S. M. A. H. Jafri, T. N. Gia, S. Dytckov, M. Daneshtalab, A. Hemani, J. Plosila, and H. Tenhunen, "NeuroCGRA: A CGRA with support for neural networks," in *High Performance Computing & Simulation (HPCS), 2014 International Conference on*. IEEE, 2014, pp. 506–511.

[31] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. ACM, 2015, pp. 199–204.

[32] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[33] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proc. of the IEEE*, vol. 103, no. 3, pp. 332–354, 2015.

[34] J. S. Brunhaver, "Design and optimization of a stencil engine," Ph.D. dissertation, Stanford University, 2015.

[35] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. Jacobs, and K. Langendoen, *Modern Compiler Design*, 2nd ed. Springer Science & Business Media, 2012.

[36] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Field-Programmable Logic and Applications*. Springer, 1997, pp. 213–222.

[37] K. K. Poon, S. J. Wilton, and A. Yan, "A detailed power model for field-programmable gate arrays," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 10, no. 2, pp. 279–302, 2005.

[38] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-programmable gate arrays*. Springer Science & Business Media, 2012, vol. 180.

[39] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for deep-submicron FPGAs*. Springer Science & Business Media, 2012, vol. 497.

[40] V. Betz and J. Rose, "FPGA routing architecture: Segmentation and buffering to optimize speed and density," in *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*. ACM, 1999, pp. 59–68.

[41] S. J. Wilton, "Architectures and algorithms for field-programmable gate arrays with embedded memory," Ph.D. dissertation, University of Toronto, 1997.

[42] C. Harris and M. Stephens, "A combined corner and edge detector," in *Alvey vision conference*, vol. 15. Manchester, UK, 1988, p. 50.

[43] E. Rosten, R. Porter, and T. Drummond, "Faster and better: A machine learning approach to corner detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 32, no. 1, pp. 105–119, 2010.