A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-Window Applications

ABSTRACT

With the emergence of accelerator devices such as multicores, graphics-processing units (GPUs), and field-programmable gate arrays (FPGAs), application designers are confronted with the problem of searching a huge design space that has been shown to have widely varying performance and energy metrics for different accelerators, different application domains, and different use cases. To address this problem, numerous studies have evaluated specific applications across different accelerators. In this paper, we analyze an important domain of applications, referred to as sliding-window applications, when executing on FPGAs, GPUs, and multicores. For each device, we present optimization strategies and analyze use cases where each device is most effective. The results show that FPGAs can achieve speedup of up to 11x and 57x compared to GPUs and multicores, respectively, while also using orders of magnitude less energy.

1. INTRODUCTION

Over the past decade, computing architectures have started on a clear trend towards increased parallelism and heterogeneity, with most mainstream microprocessors now including multiple cores, and system architectures commonly integrating accelerators such as graphics-processing units (GPUs) [2][4][24] and field-programmable gate arrays (FPGAs) [3][6][10][31] over PCIe and even on the same chip [22][30]. Numerous studies have shown that such architectures can accelerate applications by orders of magnitude compared to sequential software [2][3][4][5][29].

However, the multitude of accelerator options has significantly increased application design complexity due to the need for extensive design-space exploration to choose an appropriate device. Although GPUs have become a common accelerator due to widespread availability, low cost, and a simplified programming model compared to FPGAs, numerous device characterization [5][29] and application studies [2][3][4][23] have shown that metrics for different devices can vary significantly for different applications. Therefore, design-space exploration of different devices for different applications is critical to prevent designers from choosing inappropriate devices.

One challenge that makes such exploration difficult is that there is rarely a globally optimal device for a particular application. Instead, applications generally have a set of Pareto-optimal implementations that tradeoff numerous metrics such as performance, power, energy, cost, size, reconfigurability, application-design complexity, fault tolerance, etc. Furthermore, such exploration is complicated by numerous use cases. For example, an embedded system performing convolution may involve much smaller input sizes than convolution in high-performance computing, which would likely have different optimal or Pareto-optimal implementations.

In this paper, we perform an extensive analysis of sliding-window applications to determine the most effective devices for different use cases by considering performance and energy, different input sizes, different precisions, and different interconnects (e.g. PCIe, same chip). Sliding-window applications are a subdomain of digital signal processing that involve sliding a smaller signal (i.e., a window) across all positions in a larger signal (e.g., image), while generally performing a computationally intensive function at each window position. We evaluate sliding-window applications due to their frequent usage in digital signal processing, which is common on multicores, FPGAs, and GPUs.

The results show that an Altera Stratix III E260 FPGA is generally the fastest device for sliding-window applications compared to an NVIDIA GeForce 295 GTX GPU and quad-core Xeon W3520, with speedups of up to 11x and 57x, respectively. For an Information Theoretic Learning [27] based application, the FPGA was the only device capable of real-time usage. Furthermore, the FPGA used orders of magnitude less energy than other devices in many situations, providing the only realistic embedded system implementation for high-definition video.

The main contributions of the paper are summarized as follows:

- Highly optimized circuit architectures for FPGA implementations of sliding-window applications
- Optimization strategies for sliding-window applications on GPUs and multicores
- Analysis of performance and energy for different use cases such as different input sizes, precisions, and interconnect types, including estimations for emerging single-chip CPU/GPU devices.
- To our knowledge, we present the first sliding-window implementations demonstrated to achieve real-time usage with high-definition video, even while supporting larger window sizes than previous work.

The remainder of the paper is organized as follows. Section 2 discusses previous work. Section 3 describes the sliding-window applications that we evaluate. Section 4 describes the custom circuit architectures for our FPGA implementations, in addition to optimization strategies for the GPU and multicore implementations. Section 5 presents experimental results.

2. PREVIOUS WORK

Numerous studies have evaluated application performance for FPGAs [4][10] and GPUs [23][28]. Much work has focused specifically on image and video processing [4][17][26][28]. For example, Sinha et al. [28] evaluated the Kanade-Lucas-Tomasi (KLT) Feature Tracker algorithm on a GPU, which tracks specified features in a given image. Porter et al. [26] implemented several stereo matching algorithms on an FPGA, including sum of absolute differences (SAD). We also evaluate SAD, but using different use cases on different devices. [26] measured relative

Conference'10, Month 1–2, 2010, City, State, Country.

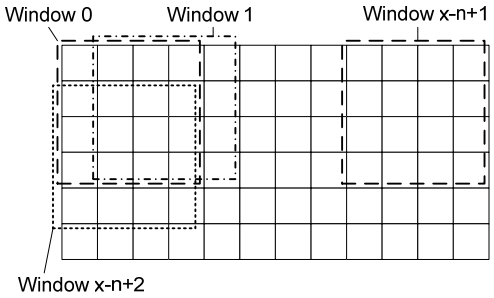

Figure 1: Input access patterns of sliding-window applications, where a "window" slides over all possible positions in the image.

cost to perform real-time computations using a custom technology-independent cost function.

Previous work has also compared performances of FPGAs, GPUs, and CPUs. Baker et al. [3] evaluated a matched filter algorithm on a Cell processor, FPGA, and GPU, concluding that the Cell provided the best performance and energy efficiency, but the GPU exhibited the best performance per dollar. Pauwels et al. [24] compared two complex vision-based algorithms requiring real-time throughput. The multi-stage algorithm involved a Gabor filter, stereo disparity estimates, local image features, and optical flow. That study found that although FPGAs were faster for certain single-stage algorithms, the GPU exhibited better performance when executing the entire multi-stage algorithm.

Several studies have considered different use cases of some of the same applications as this paper. The authors of [6] implemented 2D convolution and color correction on a GPU and FPGA to determine if GPUs can replace FPGAs in video processing. The authors optimized both implementations, and measured throughput using kernel sizes up to 11x11 for the 2D convolution. They concluded that the FPGA had better performance at higher kernel sizes than the GPU. Our study differs by evaluating multiple sliding-window applications, including 2D convolution, while also considering different precisions, larger image and kernel sizes that represent current use cases, and newer devices including multicore microprocessors.

Yu et al. [31] introduced an analytical approach to determine potential FPGA performance of sliding-window applications, while also creating on-chip buffers to exploit data reuse. In this paper, we evaluate custom FPGA circuits with similar buffering techniques for various common and emerging applications, while also comparing to GPUs and multicores.

In [2], Asano et al. studied image-processing techniques on a multicore CPU, GPU, and FPGA. The implementations included a 2D filter algorithm, SAD stereo vision disparity, and k-means clustering. The 2D filter's performance was measured up to a 15x15 kernel size, 241 SAD operations, and 48x4 distances in k-means clustering. In contrast to the SAD implementation in this paper, that previous study implemented a stereo-vision specific SAD algorithm. In that study, the FPGA had better performance for SAD and k-means, but was outperformed by the GPU for the 2D filter. The CPU outperformed the GPU in both SAD and k-means. Our study extends this previous work by providing a more in depth analysis of sliding-window applications. We present a generalized circuit architecture for sliding-window applications over a wider range of image and kernel sizes that apply to current and emerging use cases of sliding-window applications [12]. Additionally, we provide superior performance at significantly higher image and kernel sizes, and are the first to our knowledge

```
Input: image of size x×y, kernel of size n×m
for (row=0; row < x-n; row++) {
  for (col=0; col < y-m; col++) {
    // get n*m pixels (i.e., windows
    // starting from current row and col)
    window=image[row..row+n][col..col+m];
    output[row][col]=f(window,kernel);
  }
}
```

Figure 2: Pseudo-code for typical sliding window applications, assuming fully immersed windows, where the window function f() varies depending on the application.

to deliver real-time sliding-window processing of high-definition video on a single GPU or FPGA. We also evaluate a new application based on Information Theoretic Learning [27], which is an emerging area that is highly amenable to FPGA implementation.

3. SLIDING-WINDOW APPLICATIONS

For all applications, the input is a 2D *image* with dimensions $x \times y$. Although sliding-windows applications also apply to 1D examples and signals other than images, this input is representative of many applications. Each application also takes as input a 2D *kernel* of size $n \times m$, whose purpose varies depending on the application (e.g., an image to search for, a set of constants, etc.). Each application slides a *window* of the same size as the kernel across all possible positions in the image, as shown in Figure 1, where the data associated with each window are the underlying image pixels. For each window, the application performs some application-specific *window function*, shown as *f()* in Figure 2, based on the current window and the kernel. Although the number of outputs is application specific, sliding-window applications often generate one output per window, as shown in the pseudo-code. In some cases, the exact ranges of sliding windows are application specific. In this paper, we consider use cases where the kernel is fully immersed in the image, meaning that the window never exceeds the image boundaries. We chose this use case because windows that extend past image boundaries generally require a padded image. Such padding commonly requires software pre-processing that is not relevant to the device comparison, which we therefore excluded.

Note that for the remainder of the paper we use these terms:

- n and m: kernel dimensions
- x and y: image dimensions

Sliding-window applications tend to be highly memory intensive due to the need to gather each window. For example, a 40×40 window consists of 1,600 pixels. For a 1000×1000 image, there are $(1000\text{-}40\text{+}1)^2 = 923{,}521$ windows. Therefore, the total amount of pixels an application must read from memory is approximately 1,477,633,600. For 16-bit images, these reads correspond to approximately 3 terabytes of data, much of which must be accessed from memory non-sequentially.

Similarly, sliding-window applications are often computationally intensive due to complex window functions. For example, 2D convolution multiplies each pixel of a window with a constant in the kernel and then accumulates the results. For a 40×40 window, each window requires 1,600 multiplications and 1,599 additions. For a 1000×1000 image, there are 923,521 different windows, thus requiring approximately 3 teraoperations.

Many sliding-window applications have a similar behavior as shown in Figure 2. Therefore, in the following sections, we

simply define the window function *f()* for each application, along with characteristics of the image, kernel, and output.

3.1 Sum of Absolute Differences (SAD)

Sum of absolute differences (SAD) is used in content-based image retrieval [8][32] and other image-processing applications as a measure of similarity between two images. For example, a security system may search a video stream for other images (i.e., kernels) from a database of criminals. For each kernel, the output with the lowest SAD value represents the closest match.

As the name suggests, the window function for SAD calculates the absolute difference between each window pixel and kernel pixel, and then accumulates the differences for the entire window. Therefore, each output is a measure of similarity between the corresponding window and the kernel image, where lower values represent a closer match. The output upon completion is a two-dimensional data structure of dimensions (x-n+1)×(y-m+1), which corresponds to the total number of windows. Although specific applications would post-process the output, we instead simply generate the output due to the large variation in SAD applications.

For all evaluations, we consider 16-bit grayscale images for both the image and the kernel image, while exploring numerous combinations of image and kernel sizes.

3.2 2D Convolution

2D convolution is a common operation in digital signal processing and scientific computing used in computing systems ranging from small embedded systems to high-performance embedded computing systems (e.g., satellites) to supercomputers.

For 2D convolution, the window function multiplies each image pixel with a constant in the kernel. The method for generating each output pixel is shown by the following formula:

$$output[a][b] = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} image[a+i][b+j] \times kernel[n-i][m-j]$$

This formula multiplies each image pixel with a constant at the same location in a "flipped" version of the kernel. For a 3×3 window, 2D convolution multiplies pixel (0,0) of the window with the constant at (2,2) in the kernel, followed by the multiplication of pixel (0,1) with constant (2,1), etc.

After multiplying the window with the flipped kernel, 2D convolution accumulates the products, which generates a single output. The entire output for fully immersed windows is an image of size (x-n+1)×(y-m+1).

For 2D convolution, we consider 16-bit grayscale images and kernels consisting of various precisions, including 32-bit floating point and 16-bit fixed point.

One optimization commonly implemented for 2D convolution consists of performing convolution twice with one-dimensional kernels, which is possible when the two-dimensional kernel is separable. In this paper, we evaluate non-separable kernels, due to their larger computational requirements.

Similarly, applications often perform large 2D convolutions using FFT convolution due to a lower time complexity. Although we do evaluate 2D FFT convolution for the GPU and multicore, we did not evaluate 2D FFT convolution on the FPGA analysis due to the lack of a 2D FFT core. Therefore, reported FPGA speedups represent a pessimistic lower bound. Additionally, we found that our 2D FFT multicore implementation, coded with the FFTW 3.2.2 2D FFT function [9], did not perform significantly better than the OpenCL sliding-window 2D convolution for the kernel sizes we tested. While we would expect significant speedup for larger kernels, we omitted the FFT CPU results from this paper because they are not applicable to our use cases.

3.3 Correntropy

Correntropy [16] is a measure of similarity based on Information Theoretic Learning (ITL) [27]. Correntropy can be used for many purposes [12][16], but we evaluate an application similar to Section 3.1 that searches an image to find the closest match of another image. Correntropy is defined as:

$$k(image_{i,j} - kernel_{a,b}) = \frac{1}{\sigma\sqrt{2\pi}} exp(-\frac{(image_{i,j} - kernel_{a,b})^2}{2\sigma^2})$$

For the application in this paper, function *k()* is a Gaussian. Based on these equations, the correntropy application performs the following computation for each window. First, correntropy finds the difference between each pixel in the window ($image_{i,j}$)and each corresponding pixel in the kernel image ($kernel_{a,b}$). However, instead of summing these differences, each difference is used as a parameter to the Gaussian function. For an exact match (i.e., a difference of zero), the Gaussian function will return 1, which corresponds to a perfect measure of similarity. For larger differences, the similarity measure will drop increasingly fast depending on the exact characteristics of the Gaussian curve. After computing the similarity for each pixel of the window, the application sums the similarities for all pixels to create a single value that represents the similarity for the entire window. Although different applications would process the similarity values in different ways, the application in this paper outputs the the two largest values and their corresponding window locations.

4. DEVICE IMPLEMENTATIONS

In this section, we present implementation strategies for the three sliding-window applications from the previous section. Section 4.1 describes custom circuit architectures for the FPGA. Section 4.2 describes GPU implementations and optimizations. Section 4.3 is similar for OpenCL on multicore processors.

For the FPGA analysis, we target a GiDEL ProcStar III board with a 65 nm Altera Stratix III E260 FPGA. The board has four FPGAs, in addition to 3 external memories per FPGA, although we only use one FPGA to keep device comparisons fair. The board is connected over PCIe x8 to a 2.26 GHz quad-core Xeon E5520 CPU with 6 GB of RAM.

For the GPU analysis, we target a 55 nm EVGA GeForce GTX 295 PCIe x16 board with Compute Capability 1.3. This board also has multiple devices, but we limit analysis to a single device for fair comparisons. All implementations use CUDA Version 3.2. All GPU examples were tested using a Red Hat Enterprise 5 Linux 64-bit server with 12 GB of RAM and a 45 nm 2.67 GHz Intel Xeon 4-core W3520 with 8 threads via Hyper-Threading

The OpenCL multicore implementations use the same system as the GPU, but with Windows 7 Enterprise 64-bit instead of Linux in order to use the latest OpenCL Intel SDK Version 1.1.

Although evaluating an older 65 nm FPGA results in a slightly unfair comparison, the FPGA is still generally the most effective device for most use cases, as shown in Section 5. Also, although the slower processor used with the ProcStar III potentially makes

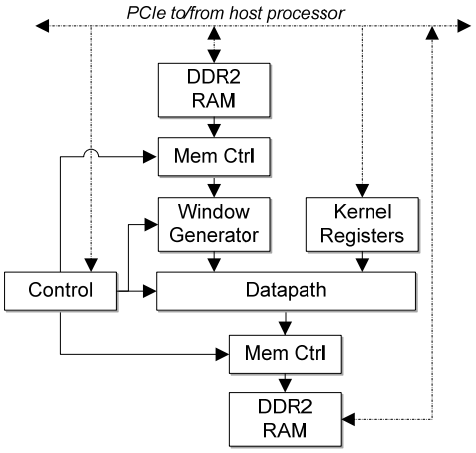

Figure 3: Circuit architecture for sliding window applications.

the FPGA results pessimistic, the CPU was responsible for less than 1% of execution time.

For the SAD and correntropy applications, we limit window sizes to 45×45 due to shared memory limitations on the GPU. Interestingly, the FPGA implementations have resource restrictions that support only slightly larger windows. For 2D convolution, we restrict window sizes to 25×25 due to limited FPGA multipliers, as discussed later. The FPGA circuits can support arbitrary kernel sizes with trivial extensions, but we limit the analysis to sizes that the FPGA can execute in parallel.

4.1 FPGA Circuit Architecture

Because much of the sliding-window functionality is shared across multiple applications, all custom circuits used for the FPGA evaluation use the architecture shown in Figure 3. This architecture consists of a controller and pipelined datapath that takes as input a window and the kernel for the application.

To keep the pipelined datapath from stalling, the circuit must provide a new window every cycle, which requires very high bandwidth. For example, a 40×40 window of 16-bit data requires 3,200 bytes per cycle, or 320 GB/s for a 100 MHz clock, which cannot be provided by external memory. However, the *window generator* buffers the overlap between consecutive windows inside of the FPGA, significantly reducing bandwidth requirements. Although previous studies have introduced various window generators [31][33], we use a buffer similar to [7] that aims to maximize performance at the cost of extra area.

The window generator buffers n-1 complete rows of the image using on-chip RAMs that act as specialized FIFOs. Like all FIFOs, these specialized FIFOs pop from the front and push to the back. However, pop operations do not actually delete the data and simply move the front pointer to the next element.

To use the window generator, the controller initially starts a sequential read from an external DDR2 memory that stores the image. The window generator stores arriving pixels in a FIFO corresponding to the current row. When the current FIFO is full (i.e., the entire row is buffered), the window generator starts storing pixels in the next FIFO. After the nth FIFO has received pixels, the window generator begins to create windows. Specifically, whenever there are pixels in all n FIFOs, the window generator pops a pixel from each FIFO into an $n \times m$ set of shift registers used to store the current window. After the window generator pops m pixels from each FIFO, the shift registers contain a valid window. The window generator continues to pop pixels, producing a new window each cycle, until all the FIFOs are empty, which corresponds to the end of one row of windows. At this point, the window generator adjusts internal pointers to move each FIFO up one row, while moving the first FIFO to the back and discarding the contents because the remaining windows will not require data from the first row. Because the other FIFOs already contain the buffered data for a row of the image, the window generator resets the front pointer for each FIFO to the first pixel, effectively making the FIFOs full again without having to reread data from memory. After resetting the front pointers, the window generator repeats this process for the remaining windows.

To buffer n rows of the image, the window generator requires $n*y$ words of memory. For example, for a 1920×1080 image with 40×40 windows, the window generator requires 1920*40 memory words. For 16-bit grayscale images, these words require only 154 kilobytes of on-chip memory, which is a small amount for current FPGAs. Although other window-generation techniques use less area [33], those techniques trade off performance. Because on-chip memory was not a bottleneck for the evaluated applications, we used the described approach to maximize performance.

For each window, the datapath performs the application-specific window function using the window and the kernel, which the circuit also stores in $n \times m$ registers. After some latency, the datapath produces an output each cycle, which a memory controller writes to a second external DDR2 memory.

The circuit connects to a PCIe bus that allows the host microprocessor to read and write data into the external memories, the kernel registers, and the controller. For all applications, the host software transfers the image into the input DDR2 memory and then initializes the kernel. Next, the software enables the controller to start the computation and then polls the controller until the datapath has produced all outputs. Finally, the software reads back all outputs from the second DDR2 memory.

4.1.1 SAD

For the SAD application, we created the datapath shown in Figure 4. The datapath takes $2*n*m$ inputs, where half of the inputs are the window pixels (shown as *w[]*), and the other half are kernel pixels (*k[]*). The datapath initially subtracts every corresponding pair of window and kernel pixels, and then takes the absolute value, which is stored in a register. The datapath then passes all $n*m$ absolute differences to a pipelined adder tree that contains

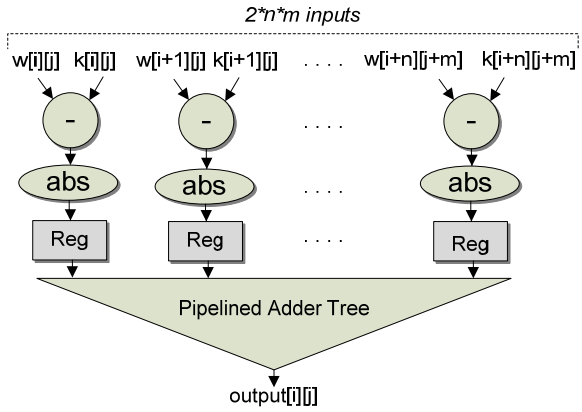

Figure 4: Datapath for sum of absolute differences (SAD).

registers at each level. The datapath then outputs the result from the adder tree. For all SAD evaluations, we use an image and kernel consisting of 16-bit grayscale images of varying sizes. The adder tree generates carry bits at each level of the tree to ensure that overflow cannot occur. Therefore, for 16-bit inputs, each output is 16+log2(*n***m*) bits. We could have potentially reduced area requirements by using 16-bit adders in the adder tree, but preventing overflow is important for many use cases and also provides a lower bound on FPGA performance.

The total number of parallel operations for the SAD datapath is *n***m* subtractions, *n***m* absolute values, and *n***m*-1 additions. For a 40×40 window, the datapath executes 1,600 subtractions, 1,600 absolute values, and 1,599 additions every cycle after the initial pipeline latency of 1+log2(*n***m*).

This SAD circuit, for a 1920×1080 image and 45×45 kernel, complete with all IP for the GiDEL board, uses 137,260 LUTs (67%), 156,377 registers (76%), 2,256,464 block memory bits (15%), and zero DSP blocks on the Stratix III E260. Resource utilization increases linearly with kernel size and the limiting resource is logic elements. The circuit was operated at frequencies between 100 and 115 MHz depending on the feature size.

4.1.2 2D Convolution

The datapath for 2D convolution is similar to Figure 4, with the difference that the subtraction and absolute value operations are replaced by a multiplication. In addition, convolution flips the order of the kernel in the inputs as described in Section 3.2. The pipelined adder tree is identical to the description in the previous section. For the convolution examples, we evaluate 16-bit grayscale images, and kernels consisting of both 32-bit floating-point values and 16-bit fixed-point values. To reduce resource usage from fixed-to-float conversions, we pre-process the 16-bit image in software and transfer 32-bit float pixels to the FPGA.

The total number of parallel operations for this datapath is *n***m* multiplications and *n***m*-1 additions every cycle. For the floating-point kernels, the output is also floating point. For the fixed-point kernels, the output is 16+log2(n*m) bits due to the adder tree accounting for overflow.

When using a 1920×1080 image and a 16-bit fixed-point 25×25 kernel, this circuit with all GiDEL IP uses 33,547 LUTs (17%), 57,122 registers (28%), 1,601,104 block memory bits (11%), and 738 DSP blocks (96%) on the Stratix III E260. For this circuit, the limiting resource is multipliers in the DSP blocks. The circuit was operated at frequencies between 104 and 115 MHz depending on the feature size.

The floating-point version uses significantly more DSP blocks to achieve the same kernel size. The resources available allow up to a 13×13 kernel, and the circuit uses 129,024 LUTS (63%), 126,821 registers (62%), 1,633,872 block memory bits (11%), and 676 DSP blocks (88%). The limiting resource is DSP blocks and the circuit used frequencies between 103 and 114 MHz.

4.1.3 Correntropy

Figure 5 illustrates the correntropy datapath. The initial stages of the pipeline calculate the absolute difference of each pair of window and kernel pixels, which is identical to SAD. The datapath then connects the absolute difference to a lookup table that implements a Gaussian curve. The datapath uses the absolute difference to take advantage of the symmetry of the Gaussian curve. By ignoring negative values, we reduce the size of the lookup table by 50%. Choosing the exact size of the lookup tables

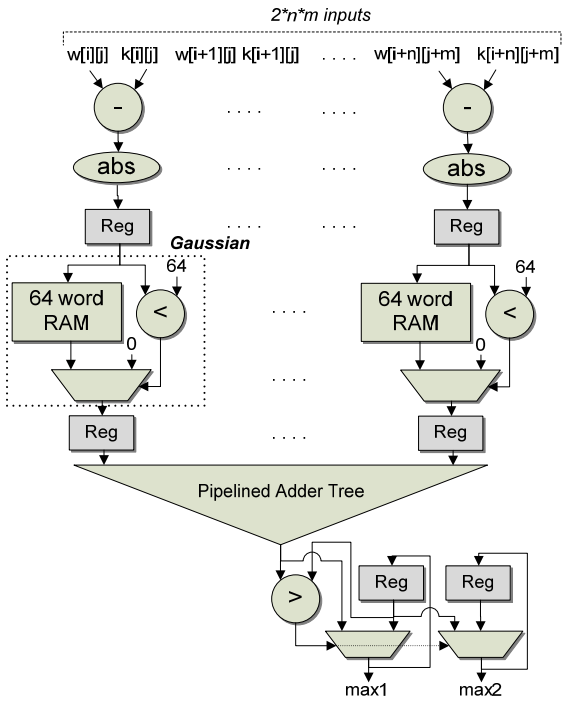

Figure 5: Datapath for correntropy.

is highly application dependent, but we chose a size of 64 words based on the curves required by a correntropy-based optical flow application [12]. In addition to the lookup table, the datapath uses a comparator and mux that treats points on the Gaussian for differences of greater than 64 as zero. The output of the Gaussian lookup is an 8-bit fixed-point value between 0 and 1 that represents the similarity between each pair of pixels. We chose the 8-bit precision based on the requirements of [12] and point out that other applications may require different precisions. These similarity values are then summed using the same pipelined adder tree as the previous examples. Finally, the datapath monitors the output of the adder tree and saves the two largest similarity values for all possible windows, along with the corresponding window positions (not shown). Each output is 8+log2(*n***m*) bits.

The correntropy datapath performs *n***m* subtracts, *n***m* absolute values, *n***m* Gaussian lookups, and *n***m*-1 additions every cycle.

For a 1920×1080 image and 45×45 kernel, the correntropy circuit, with all GiDEL IP, uses 141,633 LUTs (69%), 143,137 registers (70%), 2,256,464 block memory bits (15%), and zero DSP blocks on the Stratix III E260. Resource utilization increases linearly with kernel size, and the limiting factor is logic elements. The circuit was operated at frequencies between 101 and 111 MHz depending on the feature size.

4.2 GPU

A graphics processing unit (GPU) is a highly parallel architecture which can run thousands of threads. An overview of GPU functionality is described in [23]. We use the CUDA framework [19] to implement applications for GPUs.

A complete discussion of the CUDA code is omitted for brevity. Instead, we focus on optimizations for the CUDA memory hierarchy, which previous work has shown can significantly improve performance [23]. The CUDA memory hierarchy consists of local memory, shared memory, texture memory,

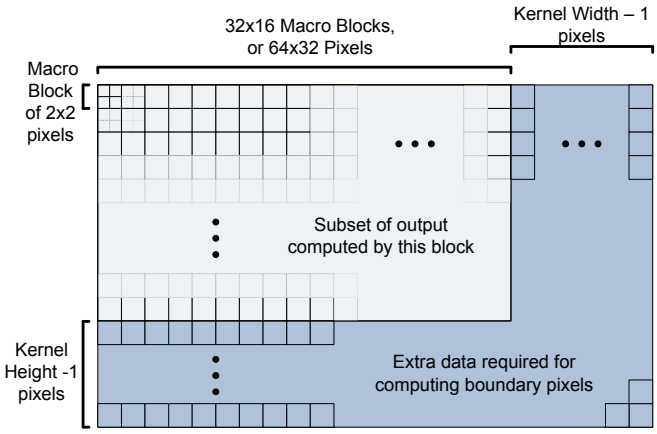

Figure 6. Organization of shared memory for each thread block.

constant memory, and global memory. Ideally, threads should use local or shared memory, which have the lowest latency, but their limited size and their restriction of only sharing data within a thread block requires applications to also use the other memories. By contrast, global memory has the largest size due to the use of external memory, and can be accessed by all thread blocks, but also has the highest latency. Texture memory is a cached version of global memory, which is more suitable for 2D locality.

The presented GPU implementations use a specialized memory organization that maximizes the usage of shared memory for the numerous repeated accesses in sliding-window applications. This organization stores the entire kernel in shared memory, as well as a subset of the image needed by the corresponding thread block, while storing the entire image in texture memory rather than global memory due to a lower penalty for uncoalesced reads and improved access times from cache hits.

The basic functionality of the GPU implementations is described as follows, which is based on [17] and illustrated in Figure 6. Each thread block initially loads a subset of the image from texture memory into shared memory, then generates the corresponding subset of the output pixels. Individual threads generate small groups of output pixels that we refer to as macroblocks, which are stored to global memory. Shared memory stores all image pixels used by a thread block, in addition to the kernel, which requires $(a+n-1)*(b+m-1)+n*m$ words, where $a \times b$ are the dimensions of the output pixels generated by the thread block. For this paper, we determined output groups of 64×32 performed well on the targeted device. Therefore, each thread block uses $(64+n-1)*(32+m-1)+n*m$ words of shared memory.

Another consideration for the GPU is macroblock size. Smaller macroblocks increase threads per block, but increased thread count may also increase shared memory bank conflicts. We empirically determined that 2×2 macroblocks performed well on the targeted device for sliding-window applications, which differed from previous work [17] that used 8×8 macroblocks. Based on this macroblock size, each thread block consists of 32×16=512 threads, which is the maximum amount.

One limitation of this memory organization is that shared memory limits the maximum window size. For example, for the evaluated GPU, the shared memory supports window sizes up to 45x45. For larger windows, implementations must use other memories, which would likely significantly reduce performance.

The SAD application's threads compute the sum of absolute differences between the kernel and the four windows in the corresponding macroblock.

The 2D convolution threads are similar, but with each thread performing multiply accumulates as described in Section 3.2. For the GPU, we also evaluate a frequency-domain implementation of 2D convolution using a 2D FFT as described in [25], which used the CUFFT library [20]. The frequency-domain implementation computes the 2D FFT of the kernel and of the image, then performs a point-wise multiplication of the frequency-domain signals. The implementation then performs an inverse FFT on the resulting products to produce the output. Pre and post-processing is required to account for small kernel sizes and to extract the output desired. Note that we refer to the original time-domain implementation as sliding-window convolution and the frequency-domain version as FFT convolution.

The correntropy implementation for the GPU extends the SAD implementation by adding the intermediate step of taking the Gaussian of each absolute difference before accumulating. We optimized performance by storing the Gaussian function in a lookup table stored in shared memory, using the same size lookup table as the FPGA implementation. Absolute differences cannot be predicted here, so shared bank conflicts likely cannot be prevented, but we still expect shared memory to provide the best performance due to low latency.

One challenge with the GPU implementation of correntropy is locating maximum similarity values. On the FPGA this functionality only required two registers, a comparator, and muxes. However, for the GPU implementation, there is no way to communicate between thread blocks other than global memory, and there is no way to synchronize or guarantee the order in which thread blocks execute. Therefore, finding the maximum value from multiple thread blocks must take place after the sliding-window outputs have been produced. We implement this maximum function as a reduction problem, where each thread compares a subset of the outputs simultaneously. Each thread temporarily stores the results, which the implementation then uses for a smaller reduction. This reduction process continues until it computes the two maximum values. We implemented this reduction by altering a highly optimized reduction adder from NVIDIA [11].

4.3 Multicore

We used the OpenCL parallel programming standard [18] for the multicore implementations. Similar to CUDA's thread organization, OpenCL organizes threads into a 1, 2, or 3 dimensional grid called an NDRange. This NDRange is divided into work-groups, which are further divided into work-items. The work-items are the threads that run on a device, and each work-item has access to three types of memory (listed from greatest to smallest latency): global, local, and private. Global memory is available to all threads. Each work-group has local memory that is shared among threads in the group. Private memory stores individual thread data.

Like the CUDA implementations, leveraging the NDRange and memory hierarchy effectively are vital for optimizing OpenCL applications. To ensure good performance, we followed all guidelines specified in [15]. Since caching OpenCL memory objects on a CPU is managed automatically by hardware, managing the memory hierarchy is limited to coalescing memory accesses. As a result, we focused our optimizations on minimizing

communication between threads. Each implementation uses the same following structure. The NDRange is a 2D grid with the same dimensions as the output. As recommended in [15], we stored the image, kernel, and output as buffers in global memory. Each work-item computes the result of one window. Unlike the GPU implementations, the OpenCL compiler automatically groups the work-items into work-groups as well as unrolls loops and vectorizes operations when applicable [15].

The implementation of each work-item was a straightforward specification of the window function for each application. The correntropy implementation used a global lookup table buffer for the Gaussian calculations. This lookup table was the same size as the one used for the GPU and FPGA. Like the GPU correntropy implementation, locating the maximum similarity values required a two-phase reduction where each work-item locates the maximum values for a section of the output.

5. EXPERIMENTAL RESULTS

The experiments section is organized as follows. We first define the experimental setup (Section 5.1). We then evaluate application performance individually for each device in terms of frames per second (Section 5.2), while also providing a speedup analysis. Next, we estimate speedup for emerging single-chip CPU/GPU systems and standalone FPGAs (Section 5.3). We then discuss the energy efficiency and performance in embedded systems (Section 5.4) of all implementations.

5.1 Experimental Setup

Details of the targeted systems are given in Section 4. We also estimated performance for emerging single-chip systems that integrate CPUs and GPUs, in addition to standalone FPGAs not requiring PCIe and a host processor. For simplicity, we refer to these GPU and FPGA systems as *single-chip systems*. We obtained upper-bound performance estimates for these systems by removing PCIe transfer times.

In addition to the implementations in Section 4, we also evaluate a sequential C++ implementation on the same microprocessor as the multicore examples, which we use as a baseline for speedup comparisons. These baseline implementations were compiled using g++ 4.1.2 with -O3 optimizations.

All implementations were evaluated for image sizes of 640×480 (480p), 1280×720 (720p), and 1920×1080 (1080p), which are common video resolutions. The SAD and correntropy implementations were evaluated at kernel sizes of 4×4, 9×9, 16×16, 25×25, 36×36, and 45×45. 2D convolution was evaluated at kernel sizes of 4×4, 9×9, 16×16, 25×25. Section 4 explains the maximum kernel sizes for each example.

5.2 Application Case Studies

In this section, we evaluate the performance of each application on each device in terms of frames per second (FPS). The frame rate is derived by inverting the execution time for one frame.

5.2.1 Sum of Absolute Differences

Figure 7 shows the frame rates for each implementation of SAD. All of the implementations were able to achieve real-time frame rates of 30 FPS or greater at small image and kernel sizes. However, the CPU rapidly decreased in performance. The GPU supported real-time usage when either the image or kernel size was small, but fell below 30 FPS for kernels larger than 25×25 in

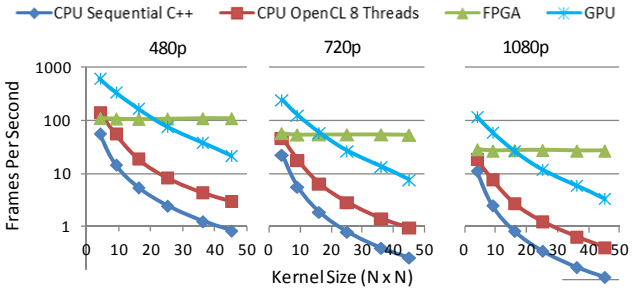

Figure 7. Performance of the SAD implementations measured in frames per second (1/execution time). Each chart corresponds to the results for all kernel sizes across one image size. The y-axis uses a log 10 scale for clarity.

720p and 1080p images. The FPGA was the only device able to maintain real-time performance over all of the input sizes tested.

The frame rates of the FPGA were constant across all kernel sizes for the same image size because the circuit computed one window each cycle regardless of kernel size. For larger kernel sizes that the circuit cannot compute in parallel, FPGA performance would decrease linearly as kernel size increased. FPGA frame rate decreased linearly with larger images, due to larger PCIe transfers and the increased number of windows.

CPU and GPU frame rates decreased linearly with kernel size (width×height) and image size (width×height). The kernel and image sizes in Figure 7 each increase quadratically, causing the CPU and GPU graphs to decrease quadratically for each image size and $O(n^4)$ overall. This trend occurs because these implementations calculate every subtraction and addition using a limited pool of parallel resources that quickly becomes saturated as kernel size increases. The GPU always delivers a faster frame rate than the CPU running OpenCL, which in turn is always faster than the CPU sequential C++ baseline.

5.2.2 2D Convolution

The frame rates for each implementation of 2D Convolution, using 16-bit fixed-point kernels, are given in Figure 8.

The trends for 2D convolution were similar to SAD, except on a smaller scale due to the more limited set of kernel sizes. As mentioned in Section 4.1.2, the FPGA supports a maximum window of 25x25 due to a shortage of multipliers. The GPU-FFT implementation had the same kernel-size independence as the FPGA but at significantly higher speeds.

The GPU-FFT and FPGA implementations were able to maintain frame rates over 30 across all input sizes tested. The two CPU

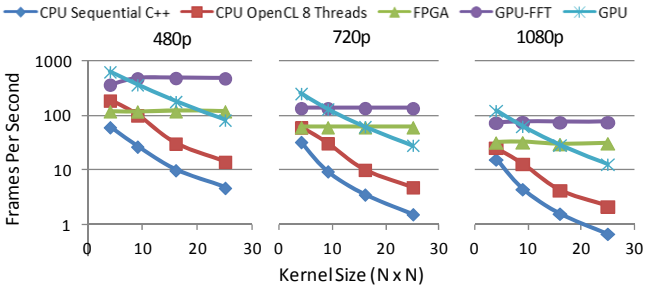

Figure 8. Performance of the 2D Convolution implementations measured in frames per second (1/execution time). Each chart corresponds to the results for all kernel sizes across one image size. The y-axis uses a log 10 scale for clarity.

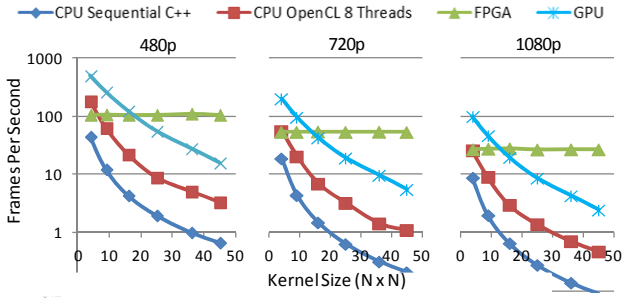

Figure 9. Performance of the correntropy implementations measured in frames per second (1/execution time). Each chart corresponds to the results for all kernel sizes across one image size. The y-axis uses a log 10 scale for clarity.

implementations were only able to provide real-time performance for 4×4 and 9×9 kernels and had low frame rates overall for 1080p images. The GPU sliding-window (i.e., time domain) implementation provided the highest frame rates for 4×4 kernels and was able to deliver 30 FPS for all kernel and image size combinations except the maximum of 25×25 and 1080p.

It should be noted that the GPU-FFT implementation performs independently of kernel size when the kernel fits within the FFT size (i.e. the 1080p version could operate with a 128×128 kernel in the same amount of time as the 25×25 kernel).

2D convolution using floating-point kernels was also evaluated. The sequential C++ baseline took an average of 2x longer to use floating point. The OpenCL and GPU implementations performed within 5% of their execution times for 16-bit fixed-point kernels. The FPGA used an average of 20% more time for the same kernel sizes, due entirely to the additional cost of moving a 32-bit image over the PCIe bus as described in Section 4.1.2.

5.2.3 Correntropy

The frame rates for each implementation of correntropy are given in Figure 8. The trends for correntropy were extremely similar to those from SAD.

The FPGA delivered real-time performance across all feature sizes at 480p and 720p, and 27 FPS for all kernel sizes at 1080p. The GPU provided more than 30 FPS for 25×25 and lower at 480p, 16×16 and lower for 720p, and 9×9 and lower for 1080p. The CPU implementations only provided real-time frame rates at the smallest kernel sizes in 480p and 720p.

5.2.4 Discussion

Performance, as indicated by speedup over the CPU sequential C++ implementation for each application, is shown for 720p images in Figure 10. The trends for 480p, 720p, and 1080p were similar enough that it is only necessary to display one image size.

The data shows that, for each application, the sliding-window GPU implementation (i.e., time domain) was faster than the FPGA for 4×4 and 9×9 kernel sizes and roughly equivalent in the 16×16 case. The FPGA gained significantly over the GPU at 36×36 and larger kernels, reaching a maximum speedup over the baseline of 240x, 45x, and 298x for SAD, 2D convolution, and correntropy, respectively. While the GPU had nearly constant speedup, the FPGA increased its speedup linearly with kernel size due to its kernel-size independent performance.

CPU OpenCL implementations provided steady speedup over the baseline, with a maximum of 3.9x, 3.7x, and 5.3x for SAD, 2D convolution, and correntropy, respectively. This consistency was

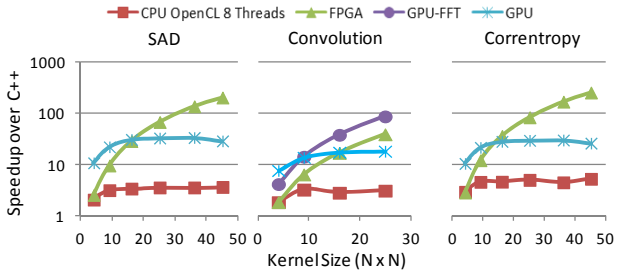

Figure 10. Speedup of all implementations over the sequential C++ baseline for SAD, 2D Convolution, and correntropy at all kernel sizes tested on 720p images. A log 10 scale is used on the y-axis for improved clarity.

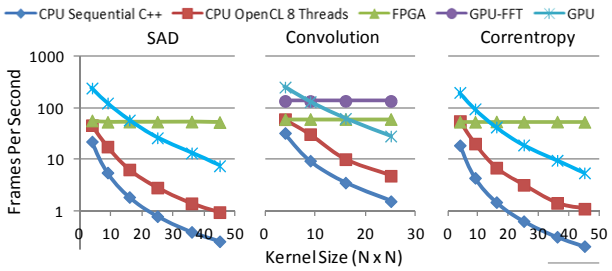

Figure 11. Performance of all implementations in frames per second for SAD, 2D Convolution, and correntropy at all kernel sizes tested on 720p images. A log 10 scale is used on the y-axis for improved clarity.

supplied by performing similar operations spread out over the 4 CPU cores. OpenCL was marginally faster than the FPGA at 4x4 kernels and significantly slower at all other sizes.

The GPU-FFT implementation for 2D Convolution was faster than the FPGA for all kernel sizes tested, with an average of 3x better performance than the FPGA. As mentioned previously, a FFT implementation on the FPGA may reduce this speedup.

The data in Figure 11 shows that performance can vary by application for each device, despite each application sharing the same basic structure and memory access pattern. The sequential C++ CPU took an average of 1.7x longer to execute SAD than 2D convolution because of the extra steps for calculating the absolute value. Correntropy took significantly longer for the sequential C++ than either SAD or 2D Convolution because of the additional step of accessing the Gaussian lookup table on top of absolute value and subtraction. Additionally, tracking the maximum value required extra comparisons. The same trends apply to the CPU OpenCL implementation.

The FPGA implementations took nearly the same amount of time to execute regardless of the sliding-window function because the pipelined architecture amortizes any extra steps as latency without affecting throughput. This behavior is a strong example of the usefulness of FPGAs when a large number of computations must be performed on a small group of data. The GPU implementation for SAD executed slightly faster than the correntropy implementation because the Gaussian lookups and the comparisons for establishing the maximum output became an expensive reduction operation, as mentioned in Section 4.2.

Overall, the three applications showed performance in the same order of magnitude for each image and kernel size.

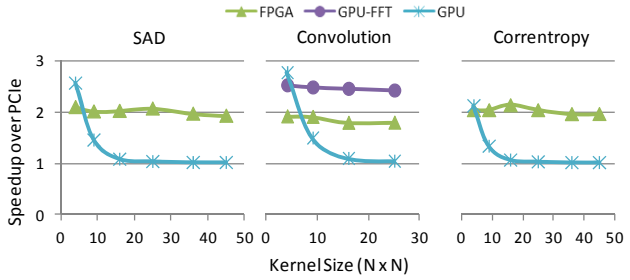

Figure 12. Speedup of single-chip implementations over their PCIe equivalents for SAD, 2D Convolution, and correntropy at all kernel sizes tested on 720p images.

5.3 Single-Chip Systems

Figure 12 presents speedup of emerging single-chip CPU/GPU devices in addition to standalone FPGAs over traditional PCIe accelerators, which we collectively refer to as single-chip systems. The results show significant improvements compared to accelerator boards due to the elimination of PCIe transfer times, which accounted for as much as 65% of execution time for the GPU and 64% for the FPGA.

The single-chip, sliding-window GPU implementations experienced the greatest speedup at low kernel sizes, which resulted from low computation compared to data set size. The speedup decreased quickly as data transferred over the PCIe x16 bus was amortized against quadratically larger computations.

The standalone FPGA implementations had a nearly constant speedup averaging 2x over PCIe versions. Speedup was constant because the execution time did not change as kernel size changed, leading to no amortization of data-transfer times. The GPU-FFT convolution implementation followed the same trend.

It should be noted that as of this writing, the available CPU/GPU chips on the market, such as the AMD Fusion APU and NVIDIA Tegra 2, do not come close to the performance of the NVIDIA GeForce GTX 295 used in these experiments. Still, the severity of the PCIe bottleneck points to huge potential for devices that hardware can accelerate without bus transfers.

The graphs in Figure 12 are limited to 720p images only because the trends for all image sizes were similar.

5.4 Energy Comparisons

To evaluate energy consumption, we calculate the energy for a given implementation by multiplying the execution time of each implementation by the worst-case power consumption of the corresponding device. Although such an analysis may be pessimistic, sliding-window applications are likely to reach these worst-case power levels due to their memory-intensive and computation-intensive behavior. The CPU implementations have a worst case power of 130 watts [14], the GPU implementations use 274.5 watts (130 watt CPU + 144.5 watt GPU) [14][21], and the FPGA uses 100 (80 watt CPU + 20 watt FPGA) [1][13]. When used as a standalone device the FPGA consumes 20 watts [1].

The data in Figure 13 shows the stratification between the energy efficiency of each device. The FPGA was clearly the most energy-efficient implementation, with one and two orders of magnitude lower energy than the sliding window GPU and CPU, respectively, at the 45×45 kernel size. The GPU-FFT implementation was able to obtain comparable energy efficiency to the FPGA for convolution because of its better performance.

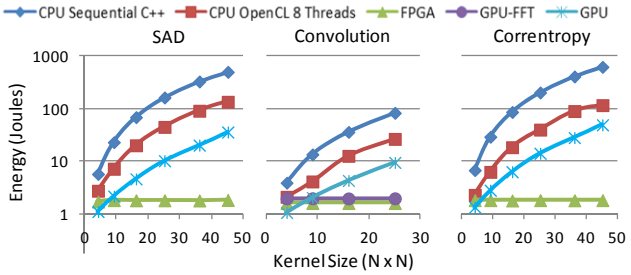

Figure 13. Energy consumed to process one frame for SAD, 2D Convolution, and correntropy at all kernel sizes tested on 720p images. A log 10 scale is used on the y-axis.

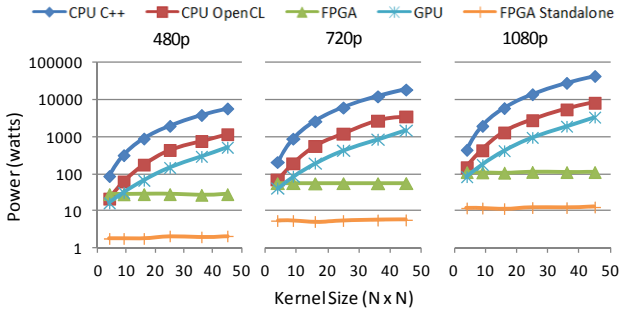

Figure 14. Theoretical wattage required for calculating 30 frames per second for correntropy. A log 10 scale is used on the y-axis.

Next, we evaluate the amenability of each device for real-time embedded systems usage by determining the theoretical power required to provide 30 frames per second. Note that many of the devices where not capable of providing such performance, causing the resulting power to exceed the worst-case power of the device. We calculate this data by multiplying the energy for 1 frame by 30 frames per second (FPS). Figure 14 presents the power analysis for correntropy, which was selected for its applications in resource-limited embedded systems [12]. The results show that an embedded system using correntropy for target tracking under a realistic power budget can only be achieved using an FPGA, as the other devices required orders of magnitude more power for larger kernels sizes. In addition, the wattage for non-FPGA systems was optimistic because those implementations were not capable of providing 30 FPS without parallelizing across multiple devices, for example 2 GPUs in an SLI configuration.

The FPGA was able to produce 30 FPS correntropy results for 2, 5.5, and 12 watts for 480p, 720p, and 1080p, respectively. The CPU and GPU required several orders of magnitude higher wattage, using a theoretical 8 kW and 3 kW, respectively, for the 1080p 45×45 case. A smaller embedded FPGA, such as an Altera Cyclone series, could perform the same correntropy calculations at an even lower wattage at the cost of lower maximum image and/or kernel size.

A system consisting of a standalone FPGA is practical for this application because the correntropy architecture described in Section 4.1.3 is capable of receiving data directly from a camera with the same Stratix III E260 used in these experiments. By contrast, the single-chip GPU estimation is excluded from this comparison because current state-of-the-art embedded GPU solutions, such as the NVIDIA Tegra 2, do not come close to the GeForce GTX 295 in performance and do not support CUDA as of this writing. We plan such analysis as future work.

6. CONCLUSIONS

In this paper, we compared performance and energy of sliding-window applications when implemented on FPGAs, GPUs, and multicore devices, under a variety of different use cases. For most cases, the FPGA provided significantly faster performance, except for small inputs sizes, with speedups up to 11x and 57x compared to GPUs and multicores, respectively. GPUs provided the best performance when the basic sliding-window functionality could be replaced by frequency-domain algorithms. FPGAs provided the best energy efficiency in almost all situations, and were in some cases orders of magnitude better than other devices. For large input sizes, FPGAs were the only device capable of realistic embedded system usage. The consistency of the results across the 3 applications studied suggests that the trends described in this paper can be applied to other sliding-window applications, with only minor differences caused by the operation applied to the sliding window. To our knowledge, we also demonstrated the first real-time sliding-window implementations to operate on high definition video with kernels up to 45×45.

7. REFERENCES

[1] Altera, Inc. 2011 Stratix III Early Power Estimator. http://www.altera.com/support/devices/estimator/st3-estimator/st3-power-estimator.html.

[2] Asano, S., Maruyama, T., and Yamaguchi, Y. 2009. Performance comparison of FPGA, GPU and CPU in image processing. In *Proc. of Int. Conf. on Field Prog, Logic and App.* FPL '09. 126-131.

[3] Baker, Z.K., Gokhale, M.B., and Tripp, J.L. 2007. Matched filter computation on FPGA, Cell and GPU. In *Proc. of the IEEE Symp. on Field-Prog. Custom Computing Machines.* FCCM'07. 207-218.

[4] Chase, J., Nelson, B., Bodily, J., Zhaoyi W., and Dah-Jye, L. 2008. Real-time optical flow calculations on FPGA and GPU architectures: a comparison study. In *Proc. of the Int. Symp. on Field-Prog. Custom Computing Machines.* FCCM '08. 173-182.

[5] Che, S., Li, J., Sheaffer, J.W., Skadron, K., and Lach, J. 2008. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proc. of the Symp. on Application Specific Processors*. SASP'08. 101-107.

[6] Cope, B., Cheung, P.Y.K., Luk, W., and Witt, S. 2005. Have GPUs made FPGAs redundant in the field of video processing? In *Proc. of the IEEE Int. Conf. on Field-Prog. Technology*. 111-118.

[7] Dong, Y., Dou, Y., and Zhou, J. 2007. Optimized generation of memory structure in compiling window operations onto reconfigurable hardware," in *Proc. of the Int. Symp. on Applied Reconfigurable Computing*, ARC '07. 110–121.

[8] Friemel, B.H., Bohs, L.N., and Trahey, G.E. 1995. Relative performance of two-dimensional speckle-tracking techniques: normalized correlation, non-normalized correlation and sum-absolute-difference. In *Proc. of the IEEE Ultrasonics Symp.*. 2, 1481-1484.

[9] Frigo, M., and Johnson, S. 2009. FFTW Library. http://www.fftw.org

[10] Guo, Z., Najjar, W., Vahid, F., and Vissers, K. 2004. A quantitative analysis of the speedup factors of FPGAs over processors. In *Proc. of the ACM/SIGDA Int. Symp. on Field Prog. gate arrays*. FPGA '04. 162-170.

[11] Harris, M. 2007. "Optimizing Parallel Reduction in CUDA," NVIDIA Developer Technology.

[12] Hunt, L. 2009. Fault-aware machine vision in small unmanned systems. In *Proc. of the Florida Conf. on Recent Advances in Robotics*. FCRAR'09.

[13] Intel. 2009. Intel Xeon Processor E5520. http://ark.intel.com/Product.aspx?id=40200

[14] Intel. 2009. Intel Xeon Processor W3520. http://ark.intel.com/Product.aspx?id=39718

[15] Intel. 2010. *Writing Optimal OpenCL Code with Intel OpenCL SDK: Performance Guide*. http://software.intel.com/file/37171/.

[16] Liu, W., Pokharel, P., and Principe, J. 2007. Correntropy: Properties and applications in non-Gaussian signal processing. *IEEE Tranactions on. Signal Process*ing, 55, 11 (Nov. 2007), 5286–5298.

[17] Mehta, S., Misra, A., Singhal, A., Kumar, P., and Mittal, A. 2010. A high-performance parallel implementation of sum of absolute differences algorithm for motion estimation using CUDA. *HiPC Conf. 2010*.

[18] Munshi, A. *The OpenCL Specification*. http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

[19] NVIDIA. 2001. CUDA. http://developer.nvidia.com/object/cuda.html.

[20] NVIDIA. 2011. CUDA CUFFT Library. http://developer.nvidia.com/cuda-toolkit-40.

[21] NVIDIA. 2009. NVIDIA GeForce GTX 295. http://www.nvidia.com/object/product_geforce_gtx_295_us.html

[22] NVIDIA. 2011. NVIDIA Tegra 2. http://www.nvidia.com/object/tegra-2.html.

[23] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., and Phillips, J.C. 2008. GPU computing. *Proc. of the IEEE*. 96, 5, 879-899.

[24] Pauwels, K., Tomasi, M., Diaz Alonso, J., Ros, E., and Van Hulle, M. 2011. A comparison of FPGA and GPU for real-time phase-based optical flow, stereo, and local image features. *IEEE Transactions on Computers.* 99.

[25] Podlozhnyuk, V. 2007. *FFT-based 2D convolution*. White Paper. NVIDIA Corporation.

[26] Porter, R.B. and Bergmann, N.W. A generic implementation framework for FPGA based stereo matching. In *Proc. of the IEEE Speech and Image Technologies for Computing and Telecommunications,* TENCON '97. 461-464.

[27] Principe, J., Fisher III, J., Xu, D. 2000. Information theoretic learning. In S. Haykin (Ed.), *Unsupervised adaptive filtering.* New York, NY: Wiley.

[28] Sinha, S., Frahm, J.M., and Pollefeys M. 2006. *GPU-based Video Feature Tracking and Matching*. Technical Report TR06-012, University of North Carolina at Chapel Hill.

[29] Underwood, K.D. and Hemmert, K.S. 2004. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *Proc. of the IEEE Symp. on Field-Prog. Custom Computing Machines*, FCCM'04. 219-228.

[30] Xilinx. 2010. *Virtex-4 Family Overview v3.1*. (Aug 30, 2010). http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf

[31] Yu, H. and Leeser, M. 2006. Automatic sliding window operation optimization for FPGA-based computing boards. In *Proc. of the IEEE Symp. on Field-Prog. Custom Computing Machines*. FCCM '06. 76-88.

[32] Zhang, J., He, Y., Yang S., and Zhong, Y. 2003. Performance and complexity joint optimization for H.264 video coding. In *Proc. of the Int. Symp. on Circuits and Systems*. ISCAS '03. 2, 888-891.

[33] Zhi G., Betul B., and Walid N. 2004. Input data reuse in compiling window operations onto reconfigurable hardware. In *Proc. of the ACM SIGPLAN/SIGBED Conf. on Languages, compilers, and tools for embedded systems.* LCTES '04. 249-256.