A Performance and Energy Comparison of Convolution on GPUs, FPGAs, and Multicore Processors

JEREMY FOWERS, GREG BROWN, JOHN WERNISING, and GREG STITT, University of Florida

Recent architectural trends have focused on increased parallelism via multicore processors and increased heterogeneity via accelerator devices (e.g., graphics-processing units, field-programmable gate arrays). Although these architectures have significant performance and energy potential, application designers face many device-specific challenges when choosing an appropriate accelerator or when customizing an algorithm for an accelerator. To help address this problem, in this article we thoroughly evaluate convolution, one of the most common operations in digital-signal processing, on multicores, graphics-processing units, and field-programmable gate arrays. Whereas many previous application studies evaluate a specific usage of an application, this article assists designers with design space exploration for numerous use cases by analyzing effects of different input sizes, different algorithms, and different devices, while also determining Pareto-optimal trade-offs between performance and energy.

Categories and Subject Descriptors: C.1.3 [Processor Architectures]: Other Architectural Styles—*Heterogeneous (hybrid) systems*; C.4 [Performance of Systems]: Design studies; D.1.3 [Programming techniques]: Concurrent Programming—*Parallel Programming*

General Terms: Performance, Design

ACM Reference Format:

Fowers, J., Brown, G., Wernsing, J., and Stitt, G. 2013. A performance and energy comparison of convolution on gpus, fpgas, and multicore processors. *ACM Trans. Architect. Code Optim.* 9, 4, Article 25 (January 2013), 21 pages.

DOI = 10.1145/2400682.2400684 <http://doi.acm.org/10.1145/2400682.2400684>

1. INTRODUCTION

Over the past decade, microprocessors have often been unable to meet the rapidly increasing performance and power requirements of many applications. As a result, designers are increasingly using accelerator devices such as multicores, Field-Programmable Gate Arrays (FPGAs), and Graphics Processing Units (GPUs), which have been widely shown to have significant performance advantages for various domains [Che et al. 2008; Owens et al. 2008; Kestur et al. 2010; Guo et al. 2004].

For designers using accelerator devices, one significant design challenge is choosing the appropriate device for a given application. Existing device characterization studies show that using the correct device is critical because metrics for different devices vary widely for different applications [Che et al. 2008; Underwood et al. 2004]. A significant amount of previous work has addressed this problem via application studies for different devices [Kestur et al. 2010; Baker et al. 2007; Underwood et al. 2004; Gac

This work is supported by the National Science Foundation grant CNS-0914474.

Authors' addresses: J. Fowers (corresponding author), G. Brown, J. Wernsing, G. Stitt, Electrical and Computer Engineering Department, University of Florida, 32611; email: jfowers@ufl.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART25 \$15.00

DOI 10.1145/2400682.2400684 <http://doi.acm.org/10.1145/2400682.2400684>

et al. 2008]. However, one common limitation of such studies is a focus on a particular usage scenario, such as a particular type of input, input size, or environment (e.g., high-performance computing or embedded systems). Although such analysis provides useful information, the results can cause designers to use inappropriate devices or algorithms for different usage cases.

One important observation for many application studies is that there is often no optimal device and algorithm for an application. Instead, the design space generally consists of a set of Pareto-optimal trade-offs between performance, power, energy, cost, etc. This problem is further complicated by the fact that many of these trade-offs are dependent on the particular use case (e.g., different input sizes and constraints). Given the multitude of accelerator devices, potential algorithms, and usage scenarios, instead of just considering speedup for a particular device, application studies should ideally determine Pareto-optimal devices and algorithms for common usage scenarios. Such analysis enables application designers to rapidly perform design space exploration to choose the best device and algorithm to meet the constraints for their intended usage of an application.

In this article, we perform such an analysis of 1D convolution for various systems consisting of multicores, FPGAs, and GPUs. Although convolution is generally a kernel used as part of larger applications, the presented analysis is critical due to the common usage of convolution in digital-signal processing on all types of accelerator devices in both high-performance computing and embedded systems. In addition to considering multiple systems, the analysis considers both time-domain and frequency-domain algorithms combined with novel optimization strategies, while also determining Pareto-optimal devices for numerous input sizes. This research enables a designer with a predetermined kernel, signal, performance, and energy use case to easily select the device and algorithm which best suits their application. Additionally, the presented Pareto-optimal analysis provides a basis for a more general comparison of CPUs, GPUs, and FPGAs in digital-signal processing.

2. PREVIOUS WORK

Numerous application case studies have been done for FPGAs [Bae et al. 2011; Guo et al. 2004; Le et al. 2011] and GPUs [Huang et al. 2009; Owens et al. 2009; Xiao et al. 2009]. To our knowledge, there has not been a thorough analysis of convolution across accelerator devices, although Koehler et al. [2012] performed a partial execution time analysis of time-domain convolution on both GPUs and FPGAs. This article extends that previous work by considering multiple algorithms, optimizations, and input sizes, while also evaluating energy.

Recent studies have also compared applications across CPUs, GPUs, and FPGAs. In Che et al. [2008], the authors evaluated Gaussian Elimination, Data Encryption Standard, and Needleman-Wunsch, and showed that FPGAs had the best performance for all input sizes, but with higher development costs. A matched filter algorithm was evaluated on a Cell processor, FPGA, and GPU in Baker et al. [2007], which concluded that the Cell provided the best performance and energy efficiency, but the GPU provided the best performance per dollar. Araya-Polo et al. [2011] implemented reverse time migration on a CPU, FPGA, GPU, and Cell and found that the accelerators all outperformed the CPU at the expense of significantly greater development effort. A comparison of BLAS algorithms was conducted in Kestur et al. [2010], which showed that FPGAs achieved comparable performance to other devices, but improved energy efficiency by 2.7x to 293x. Gac et al. [2008] presented a comparison of devices for high-speed 3D tomography where GPUs achieved the best performance. Li et al. [2011] compared device performance for Fourier-domain optical coherence tomography, for which FPGAs had significant performance advantages. Williams et al. [2010] performed

a device characterization analysis of numerous devices, showing that FPGAs were superior in terms of computational density for bit-level operations, in addition to 16-bit and 32-bit integer operations. GPUs provided better computational density for single-precision and double-precision floating-point operations, but FPGAs had better computational density per watt. Overall, these previous works show that GPUs and FPGAs often have significant speedup over CPUs, but that the best device depends heavily on the application and use case.

In addition to evaluating device performance for different applications, numerous studies have identified Pareto-optimal implementations via design space exploration including accelerator selection [Fowers et al. 2012], hardware/software partitioning [Eles et al. 1997], tuning configurable resources in a system-on-a-chip [Givargis et al. 2002], among others. This article complements previous work by performing a detailed analysis of convolution for different devices, different algorithms, different input sizes, and for usage scenarios including both high-performance computing and embedded systems.

3. CONVOLUTION

Convolution is the mathematical process of relating a linear, time-invariant system's output $y(t)$ to its input signal $x(t)$ and kernel $h(t)$, also known as the impulse response (not to be confused with a computational kernel). This article focuses on the one-dimensional convolution of discrete, finite sequences using two common algorithms: *time-domain*, also known as direct, convolution and *frequency-domain*, also known as Fast Fourier Transform (FFT) or fast, convolution. Both algorithms take signal x with length N_x and kernel h with length N_h and produce output y with length $N_y = N_x + N_h - 1$.

$$y(n) = \sum_{k=0}^n h(k) \times x(n-k) \quad (1)$$

Time-domain convolution produces each element $y(n)$ by summing the products of elements of h and x , as shown in Eq. (1), with a time complexity of $O(N_x * N_h)$.

$$y(n) = H(n) \times X(n) \quad (2)$$

Frequency-domain convolution takes advantage of the Convolution Theorem (depicted in Eq. (2)), which specifies that multiplication in the frequency domain is equivalent to convolution in the time domain. Frequency-domain convolution is accomplished by taking the FFT of both h and x , point-wise multiplying each element of the resulting functions H and X , and taking the IFFT of the result, Y , to produce output y . It should be noted that the size of the FFT, N_f , must be equal to the size of the IFFT and greater than or equal to output size N_y . The time complexity of frequency-domain convolution is $O(N_f \log N_f)$. For the remainder of the article, we assume that the signal size N_x is greater than the kernel size N_h . For situations where the kernel is larger, we simply treat the kernel as the signal, which is possible due to the commutativity of convolution.

The frequency-domain algorithm has a general performance advantage due to the $O(N \log N)$ time complexity. However, the time-domain algorithm can have an advantage at small kernel sizes by skipping operations known to result in 0 because of finite kernel size and because $A \times O(N^2)$ can be smaller than $B \times O(N \log N)$ for particular values of A , B , and N that vary for different devices.

A third algorithm, overlap-save (also known as overlap-discard) [Mansour et al. 1982], is used when an FFT size greater than or equal to N_y (typically driven by N_x) cannot be achieved due to resource limitations. Overlap-save takes advantage of circular convolution to produce one large convolution with many smaller FFTs of size

N_f on the signal. The last $N_h - 1$ input elements of each FFT are used (saved) as the first $N_h - 1$ inputs to the next FFT, followed by $N_f - N_h + 1$ new elements from the signal array. The first $N_h - 1$ outputs are also discarded from the IFFT output and the remaining outputs are concatenated to form the final convolution output. The number of FFTs required increases dramatically as N_h approaches N_f because only $N_f - N_h + 1$ signal elements can be processed per FFT, which results in poor performance for higher kernel sizes. Overlap-save also has the limitation that N_f must be greater than N_h .

4. DEVICE IMPLEMENTATIONS

To perform the evaluation of real-number convolution, we created a total of 7 convolution implementations for 3 devices: a multicore CPU, an FPGA, and a GPU. For each device, we created a highly optimized implementation of both the time- and frequency-domain convolution algorithms, in addition to a third CPU implementation which exploits multithreading for frequency-domain convolution. Each implementation uses single-precision floating point. The implementations and corresponding optimizations are discussed in the following subsections.

During the course of the study we discovered that our method for calculating FFTs on signals with no resource constraints—using the smallest power-of-two FFT size larger than the (signal size+kernel size-1)—could waste large amounts of computation. For example, for a signal size of 100 and a kernel size of 30, the smallest power-of-two FFT size would be 256 to process only 129 elements. We realized that using multiple FFTs with the overlap-save algorithm, for example, 3 FFTs of size 64, could process the same 129 elements with far fewer FFT inputs, in this case 192. We found that the savings in FFT size resulted in faster computations despite the overhead of overlap-save described in the previous section.

To exploit this performance improvement, we optimized all frequency-domain implementations to use overlap-save even when it wasn't mandated by resource requirements. To distinguish this optimization from the traditional overlap-save *algorithm*, we refer to this optimization as the *overlap-save optimization*. To perform this optimization appropriately, we used the cost function in Eq. (3) to estimate wasted computation for each implementation.

$$Cost(N_x, N_y, N_f) = Ceil\left(\frac{N_x + N_y - 1}{N_f - N_h + 1}\right) * N_f * \log_2 = \# \text{ of FFTs} * \frac{Cost}{FFT} \quad (3)$$

This cost function approximates the cost of a given implementation as the output size divided by the signal elements processed per FFT (i.e., the number of required FFTs) multiplied by the cost of a single FFT. Therefore, to optimize an implementation, we identify the FFT size with the smallest computational cost for convolution of a given signal and kernel by minimizing the cost function. Using a large number of FFTs can result in a significantly smaller number of floating-point operations than evaluating the entire convolution with a single FFT ($N_f = N_y$ rounded to the next power of two) for large values of N_y , because

$$N_f * \log_2 N_f \gg \# \text{ of FFTs} * \frac{N_f}{\# \text{ of FFTs}} * \log_2 \frac{N_f}{\# \text{ of FFTs}}. \quad (4)$$

This overlap-save optimization is most effective when kernel size is small because the FFT size, which must be higher than the kernel size, has more flexibility.

4.1. Multicore CPU Implementations

We evaluated the CPU implementations on a 64-bit Red Hat Enterprise 5 Linux system with 12GB of RAM and a 2.67GHz 45nm Intel Xeon 4-core W3520 with 8 processing

threads via Hyper-Threading. We created each implementation in C++ and compiled using g++ version 4.1.2 with high (-O3) optimization.

We created the time-domain implementation for the CPU based on a straightforward C++ implementation of the functionality defined by Eq. (1). The code iterates through the input arrays, compiling the summed product of signal and kernel items without any explicitly coded parallelism. We could have potentially parallelized the time-domain implementation across multiple cores, but our analysis showed that the frequency-domain CPU implementation was almost always significantly faster. Therefore, we focused on parallelizing all other implementations.

For the frequency-domain implementation, we used the Fastest Fourier Transform in the West (FFTW) [Frigo et al. 2005] to provide a highly optimized FFT. FFTW is a C library that analyzes the target hardware and creates a plan for an efficient execution of the requested FFT operation by automatically considering multiple FFT algorithms and load balancing the selected algorithm across available CPU cores while optimizing for vector units, different cache sizes, etc.

To further optimize the frequency-domain implementation, we applied the previously discussed overlap-save optimization to select a power-of-two FFT size that minimizes computational cost. Our experiments showed that the overlap-save optimization improved the execution time by as much as 8.6x and an average of 2.2x versus a single power-of-two FFT for the input parameters tested.

Once an FFT size is selected by our cost function, FFTW plans highly optimized FFT and IFFT functions. The implementation uses the FFTW FFT to transform the kernel, and then iterates through the signal to transform windows of size $N_f - N_h + 1$. Next, the implementation takes the dot product of the frequency-domain signal window with the frequency-domain kernel, and calculates the IFFT to produce a chunk of the result according to the overlap-save algorithm. The chunks are concatenated in the output vector while discarding extra circular convolution values.

If the overlap-save cost function determines that a single FFT is ideal, then the implementation uses the standard frequency-domain algorithm without overlap-save, where N_f is the nearest power-of-two above $N_x + N_h - 1$.

We created single-threaded and 8-threaded frequency-domain implementations by setting the number of threads available to FFTW to minimize communication overhead and take full advantage of the target CPU, respectively. The dot product operation was not parallelized because analysis showed it represented less than 5% of computation. We limited the number of overlap-save FFTs available to the 8-threaded implementation for a given convolution to 5000 to avoid the significant communication overhead of coordinating many thousands of small FFTs.

We also considered a 4-threaded implementation to match the number of physical cores on the CPU instead of the number of Hyper-Threading cores. However, preliminary experiments showed an average of 13% better performance for 8 threads. The 4-threaded implementation had significantly faster performance than 8 threads for very small input sizes, due to the lower overhead of coordinating fewer threads (single-threaded performance was even higher for those inputs). The speedup of 8-threaded over 4-threaded performance increased substantially as input size increased, because the FFTW planner was able to map the computation to the number of Hyper-Threads available. We decided to use the single-threaded and 8-threaded implementations in the experiments because they had superior performance at small and large input sizes, respectively.

4.2. FPGA Accelerator Board Implementations

Because FPGA implementations are generally specific to a board, we targeted the implementations to the Gidel ProcSTAR III board, which is used in both embedded

Fig. 1. Datapath architecture for FPGA frequency-domain convolution.

systems and in the Novo-G supercomputer [George et al. 2011]. This board contains four Altera Stratix III E260 FPGAs, fabricated at 65nm, with three external DDR2 memories per FPGA. To keep the device comparisons fair, we only use one FPGA. The board connects to a host microprocessor over PCIe x8, which causes the FPGA to have higher communication overhead than the other devices. We designed both implementations using VHDL, which we synthesized using Quartus 9.1.

4.2.1. Time-Domain FPGA Implementation. For the time-domain implementation, the circuit stores the signal, kernel, and output arrays in the three separate off-chip DDR2 memories. The datapath consists of a fully pipelined floating-point multiply-accumulate tree, which is capable of doing 128 multiplications and 127 additions every cycle at 125 MHz. For the floating-point operations, we used Altera IP cores. The entire circuit implements time-domain convolution by first loading the kernel into a set of registers connected to one of the inputs of each multiplier. Next, the circuit streams the signal from the off-chip memory into a specialized buffer [Dong et al. 2007] that generates kernel-sized windows of data each cycle. For example, for a 128-element kernel, the buffer would output signal elements 0-127, 1-128, 2-129, etc. This buffering is possible because time-domain convolution is a sliding-window algorithm, where the window is the size of the kernel. The buffer provides the multiply-accumulate tree with 128 inputs every cycle, which is critical for avoiding datapath stalls.

Although the FPGA cannot implement more than 128 floating-point multipliers, which limits the maximum kernel size to 128, we extended the implementation to support arbitrary kernel sizes using the common overlap-add method (not to be confused with overlap-save). The overlap-add circuit iteratively generates partial results by performing multiple convolutions of a signal with nonoverlapping 128-element chunks of the kernel. During each iteration, the circuit overlaps the previous partial results appropriately, and then adds these partial results to the current convolution output. After the circuit has processed the entire kernel (i.e., kernel size 128 iterations), the convolution output is correct.

This implementation used 79% of logic (93k ALUTs, 158k registers), 6% of BRAM (847k bits), and 67% of DSP blocks (256) on the Stratix III E260.

4.2.2. Frequency-Domain FPGA Implementation. The frequency-domain implementation for the FPGA uses a fully pipelined circuit for computing the result at 175 MHz, as depicted in Figure 1. Unlike the time-domain implementation, this circuit stores the signal and kernel input arrays into a single off-chip RAM (shown as *Off-chip Input RAM*) to reduce transfer overhead. This storage is possible because the entire contents of the single *Off-chip Input RAM* are continuously and sequentially streamed into the datapath. Initially, the circuit streams the kernel into the datapath and directs the kernel elements through Altera’s FFT v9.1 IP Core (variable streaming mode) [Altera

2011b]. The circuit then streams the results of that FFT operation to an on-chip RAM block (shown as *On-chip Kernel RAM*) for later use.

Next, the circuit streams the signal into the same *FFT Core* and directs the FFT results to a complex dot product block, while simultaneously streaming the corresponding kernel elements from the *On-chip Kernel RAM*. The output of the dot product streams into the *IFFT Core*, also based on Altera FFT v9.1, which produces the final result that the circuit stores into *Off-chip Output RAM*. The controller for this datapath automatically handles any zero padding necessary to fit the data into a power-of-two FFT, avoiding unnecessary data transfer.

The area-intensive pipelined FFT core allows for a maximum FFT size of 32,768 on the targeted FPGA. To support convolutions with more than 32,768 total elements, we extended the initial implementation to use overlap-save. The “save” portion of the algorithm is handled by another on-chip RAM (shown as *On-chip Signal RAM*), which caches relevant signal elements as they stream in from the *Off-chip Input RAM*. The controller (not shown) operates a multiplexor (*MUX*) that can alternate between sending zeros, data from the *On-chip Signal RAM*, or data from the *Off-chip Input RAM* to the *FFT Core*. The controller handles the “discard” portion of the algorithm by only sending valid IFFT outputs to *Off-chip Output RAM*. Overlap-save is well suited to this pipelined architecture because caching and selective data discarding are simple tasks for an FPGA, adding only two cycles of computation to each FFT iteration and requiring an insignificant amount of BRAM for caching the signal.

To further optimize the implementation, we used the overlap-save optimization to select FFT sizes between 128 and 32,768 for the runtime-configurable FFT and IFFT cores. The circuit automatically implements overlap-save whenever output size exceeds FFT size. The computation savings from the cost function optimization are limited by the low flexibility of an FFT core with a maximum size of 32,768.

The maximum FFT size of 32,768 also limits the kernel size of the implementation even when using overlap-save. Kernel size must be below 32,768, and the number of FFTs required to complete the convolution becomes

$$\# \text{ of FFTs} = \text{Signal Size} / (N - \text{Kernel Size} + 1). \quad (5)$$

As kernel size approaches FFT size, the number of required FFTs in Eq. (5) grows rapidly and quickly becomes prohibitive, which is a limitation for this implementation. For example, with a signal size of 1,000,000 and FFT size of 32,768, the computation on a size 50 kernel needs 31 FFTs. At a kernel size of 16,000, sixty FFTs are required, roughly doubling execution time. In contrast, a computation with size 32,000 kernel requires 1300 FFTs and takes 640x longer than the size 50 kernel.

This implementation used 89% of logic (102k ALUTs, 147k registers), 54% of BRAM (8M bits), and 33% of DSP blocks (256) on the Stratix III E260.

4.3. GPU Implementations

To evaluate the GPU, we targeted an EVGA GeForce GTX 295 PCIe x16 card on the same machine as the multicore implementations. The GPU in this evaluation uses a 55nm process, which is a newer process than the FPGA (65nm) and an older process than the CPU (45nm). The GeForce GTX 295 has two GPUs on the board, but like the FPGA implementation, we only considered a single GPU chip to keep device comparisons fair. Each GPU chip has 240 CUDA cores, a graphics clock of 576 MHz, processor clock of 1242 MHz, and 896MB of GDDR3 memory. We created both GPU implementations using C++ and CUDA SDK 2.3 with high (-O3) optimization.

The time-domain GPU implementation focuses on optimizing GPU memory, which consists of a large, slow “global” memory, and a fast, small “shared” memory. We arrange the convolution input as a 2D table where rows correspond to signal elements, columns

Fig. 2. Overview of the GPU time-domain convolution implementation.

correspond to kernel elements, and cells are the product of the corresponding signal and kernel elements. Outputs are generated by accumulating diagonal strips of the table, as visualized in Figure 2(a).

The shared memory of current GPUs is too small to store $N_x \times N_h$ elements, which requires the problem to be broken up into regions to optimize memory access, as shown in Figure 2(b). Each problem region consists of 512 rows of the table to capitalize on the 512 threads available per thread block in the GTX 295. Each region includes all of the columns, however, to save on shared memory the implementation only caches small groups of kernel elements from global memory as needed.

Each thread calculates the products for one column of the region in parallel, and then passes the result down and to the right for accumulation every iteration. The thread at the bottom of the region sends its products to a *partial sum buffer* that will be used in the next region, and the thread at the top of the region receives products for accumulation from the partial sum buffer from the previous region. This process is visualized in Figure 2(c).

The GTX 295 supports 30 thread blocks, which we use simultaneously to process 30 of these regions in parallel. We further optimized performance by aligning memory accesses and utilizing the CUDA “broadcast” function for distributing the kernel value for the current column to all 512 threads in a given region.

The GPU frequency-domain implementation uses CUFFT (CUDA FFT) [NVIDIA 2011a], which is the CUDA equivalent of the FFTW library used in the multicore implementation. The design is similar to the multicore version, except that CUFFT calls replace the FFTW calls and all I/O must be passed over the PCIe bus.

To optimize the implementation, we used the overlap-save optimization to select an ideal FFT size and number of FFTs. Experimental evidence showed that communication overhead between the various GPU threads prevented the overlap-save optimization from improving performance for signal sizes under 1 million elements, so we used a single power-of-two FFT for those computations. The overlap-save optimization

achieved a maximum speedup of 2.4x compared to the GPU without the optimization. The average speedup from this optimization was 1.4x for signal sizes above 1 million.

After selecting an FFT size, the implementation allocates the signal, kernel, and output on the GPU and transfers inputs over the PCIe bus. The CUFFT library [NVIDIA 2011a] plans an FFT and IFFT to maximize performance, and then executes the FFT on the kernel. If the cost function selected a single FFT then the entire signal is transformed to the frequency domain. Otherwise, the signal is broken into overlapping chunks which are transformed in sequence using the FFT. The implementation then executes a parallelized dot product between the kernel and signal, followed by an IFFT on the resulting data to produce the final output. The output is sent back to the CPU over the PCIe bus.

To overcome the limitation of 896MB of GDDR3 RAM on the targeted board, the frequency-domain implementation also uses the overlap-save algorithm for signal sizes greater than 8 million elements.

5. EXPERIMENTS

The experiments section is organized as follows. We first define the hardware used in the experiments and the tests conducted (Section 5.1). We then evaluate each implementation individually and present the performance as speedup over a common baseline, highlighting particular areas of strength for each (Section 5.2). Next, we compare the execution times (Section 5.3) and energy consumptions (Section 5.4) of all implementations. Lastly, we discuss the results and determine Pareto-optimal trade-offs for a variety of use cases (Section 5.5).

5.1. Experimental Setup

All multicore and GPU implementations were tested using a Red Hat Enterprise 5 Linux 64-bit server with 12GB of RAM, a 2.67 GHz Intel Xeon 4-core W3520 with 8 processing threads via Hyper-Threading, and a GeForce 295 PCIe x16. FPGA implementations were tested on Node 5 of the Novo-G reconfigurable supercomputer [George et al. 2011], which uses a GiDEL ProcStar III accelerator board with Stratix III E260 and a PCIe x8 interface connected to a 2.27 GHz quad-core Xeon E5520 CPU and 6GB of RAM. All results include data transfer times between devices. Section 4 provides additional information for each setup. Although the slower Novo-G processor potentially makes the FPGA results pessimistic, the CPU was responsible for less than 1% of execution time.

In addition to the PCIe FPGA experiments, we also evaluated the same FPGA frequency-domain implementation for standalone FPGAs where the data source was directly connected to the device, which is common in embedded systems. The performance for a standalone FPGA system was approximated by subtracting the bus-transfer times from the total execution times. We confirmed that such an approximation had a worst-case 10% error via VHDL simulations, although most input sizes had smaller error. Although ideally we would have also evaluated single-chip CPU/GPU processors, at the time of writing, GPUs capable of performing convolution without an external CPU do not come close to the performance for the GeForce GTX 295. Conversely, the Stratix III E260 used in these experiments is used commercially for performing standalone convolutions. Future work will extend the evaluation for emerging devices, such as single-chip CPU/GPUs [Brookwood 2010].

All implementations were tested using signal sizes between 128 and 64M and kernel sizes between 50 and 32550. The signal cap was established due to memory constraints for the FPGA board (2GB) and the kernel cap was established due to the FFT size limit in the FPGA. For clarity of discussion we will refer to small signal sizes as the range 128–16K, medium as 16K – 512K, and large as 512K – 64M; small, medium, and large

kernel sizes correspond to ranges of 50–10K, 10K–20K, and 20K–32550, respectively. Such kernel sizes are representative of common convolution usage. For example, FIR filters commonly use small kernels with 32–128 elements. Audio applications may use kernels corresponding to a second-long impulse response, which at 44.1 kHz represents a kernel size of 44100. Based on our analysis of applications, smaller kernels tend to be more common, but different applications use kernels across this entire range.

The baseline for all speedup comparison was the single-threaded, frequency-domain CPU implementation, which we believe represents the most common implementation that achieves useful performance across all signal and kernel sizes. Although we compare this baseline to different algorithms and devices, such a comparison is fair because Pareto-optimal trade-offs are affected by both device and algorithm. Therefore, to allow application designers to select the most appropriate Pareto-optimal design for their intended usage scenario, we evaluate all combinations of devices and algorithms.

5.2. Device Performance Evaluations

5.2.1. Multicore Performance. Our analysis for the time-domain CPU implementation showed that this implementation was always the slowest for any input size and that execution times became unreasonably long for larger input sizes, and hence it is not included in the results.

The execution-time graph for the single-threaded, frequency-domain CPU implementation is shown in Figure 3(a). The results show the characteristics of the frequency-domain algorithm discussed in Section 4.1. Interestingly, execution time is independent of kernel size for most input combinations because the algorithm chooses the FFT size based on the output size, which is largely driven by signal size.

Speedup for the 8-threaded frequency-domain multicore implementation versus the single-threaded baseline is depicted in Figure 3(b). Speedup is achieved for all inputs except for where both kernel and signal size are small, in which case overhead for synchronizing threads exceeds the benefits of multithreading. The spikes in the speedup graph are caused by minor inconsistencies due to FFTW planning efficiency for various input sizes. The maximum speedup was 3.2x.

5.2.2. FPGA PCIe Accelerator Board Performance. The time-domain results for the FPGA, shown in Figure 4(a), show a maximum speedup of 4.3x found in the large signal size and very small kernel size region, while a maximum slowdown of 39x occurred when kernel size was large.

The frequency-domain FPGA implementation, depicted in Figure 4(b), achieved speedup over the baseline for most input parameters and had a maximum speedup of 5.5x. As explained in Section 4, execution time increased rapidly as kernel size approached the FPGA's FFT size limit of 32768, reaching 17 seconds at 32050 elements and 57 seconds at 32550 elements, compared to 1.31 seconds at 16050 elements and 0.94 seconds at 50 elements, for the maximum input size of 64M.

5.2.3. Standalone FPGA Performance. The speedup results for the standalone FPGA frequency-domain implementation compared to the single-threaded frequency-domain CPU implementation are shown in Figure 5(a). Peak speedup increased from 5.5x to 13x and average standalone execution was 2.8x faster overall. Figure 5(b) shows that communication accounts for an average of 55% of execution time with a peak of 90% at the smallest input sizes. These results argue strongly for the need for faster interconnect on FPGA boards, which typically achieve half the bandwidth, at most, of GPU boards.

5.2.4. GPU Performance. As shown in Figure 6(a), the GPU time-domain implementation outperformed the CPU frequency-domain baseline when one input parameter was small and the other was medium to large. Performance was far higher in the large

Fig. 3. (a) Execution time in seconds for single-threaded, frequency-domain CPU implementation over a variety of signal and kernel sizes; (b) speedup results for 8-threaded, frequency-domain CPU convolution vs. 1 thread. A logarithmic base-2 scale is used on the *Signal Size* axis.

signal size case, reaching a peak speedup of 19x. The large kernel size case achieved speedup of 16.9x. Performance for all other kernel and signal sizes was significantly slower than the baseline, with a maximum slowdown of 154x at the top kernel size of 32550.

The GPU frequency-domain implementation, depicted in Figure 6(b), outperformed the baseline in all situations except when both input parameters were small. Speedup increased in intervals as either input parameter increased, with a maximum speedup of 15.7x near the maximum signal and kernel size.

5.3. Performance Comparisons

Execution-time comparisons for each of the implementations are shown for small (50), medium (16050), and large (32550) kernel sizes in Figure 7 (a), (b), and (c), respectively. These kernel sizes represent corner-cases and clearly show how substantially

Fig. 4. Speedup results for PCIe-FPGA (a) time-domain convolution and (b) frequency-domain convolution vs. CPU frequency domain-convolution with 1 thread.

performance changes in different kernel-size regions. The small size is indicative of the region where time-domain algorithms are most practical. The medium size is outside the region of practicality for time-domain and also represents the region just before the frequency-domain FPGA implementation becomes significantly saturated. Lastly, the large size shows an almost completely saturated frequency-domain FPGA implementation and displays the utility of kernel-independent frequency-domain implementations. Standalone FPGA results were not included because the standalone FPGA is unlikely to be used in the same context as the other devices, but those results are discussed later in this section.

Comparing the performance of all implementations shows that the CPU dominates when the signal and kernel are both small. CPU performance slides compared to other implementations as either parameter increases as the increased parallelism of the accelerator devices begins to justify their overhead. The time-domain implementations are fastest when one parameter is small and the other is large. The GPU time-domain

Fig. 5. (a) Speedup results for standalone-FPGA frequency-domain convolution vs. CPU frequency-domain convolution with 1 thread. Note that maximum speedups are more than 2x higher than the PCIe FPGA; (b) percentage of execution time spent on communication for the frequency-domain PCIe FPGA.

implementation is always faster than the FPGA time-domain implementation in these regions, with up to 8x better performance.

The frequency-domain implementations outperform the time-domain implementations when both the kernel and signal sizes are above 50 and 2050, respectively. When the standalone FPGA is excluded, the GPU implementation is the fastest in this region. The 8-threaded CPU is the next best option to the GPU when both signal and kernel are medium sized, although it is 3.2x slower on average. When signal size is medium to high and kernel size is medium the PCIe FPGA is the most competitive to the GPU, although the GPU is an average of 2.1x faster. Lastly, for all large kernel sizes, the GPU has a large advantage of 2–5x over its nearest competitor, the 8-threaded CPU.

The standalone frequency-domain FPGA implementation had higher performance than all other implementations for many inputs in the region defined by signal sizes above 8192 and kernel sizes below 13050, a region which would otherwise be held by the frequency-domain GPU implementation. The optimal implementation for each input size is depicted in the boundary chart in Figure 8.

Fig. 6. Speedup results for GPU (a) time-domain convolution and (b) frequency-domain convolution vs. CPU frequency domain-convolution with 1 thread.

It should be noted that in Figure 7 there are many input parameters for which multiple implementations have close to best execution time. For example, at small kernel sizes and signal sizes between 4096 and 32678, the CPU frequency-domain single-thread, GPU frequency-domain, and GPU time-domain implementations are comparable. As a general rule, the time-domain implementations all perform well when one input parameter is very small, the frequency-domain implementations have steady performance over most input sizes, and both the FPGA and GPU provide speedup over the CPU except when both input parameters are very small.

5.4. Energy Comparisons

We measured the power consumption for each implementation by monitoring a watt meter while running the convolution code in a loop. Each measurement includes all system components including the power supply, hard drives, etc., but only the relevant acceleration hardware. For example, the FPGA measurement was taken with no

Fig. 7. Execution time—lower is better—of each implementation for (a) small kernel size of 50; (b) medium size of 16050; and (c) large size of 32550 at a variety of signal sizes. Both axes use a base-2 log scale.

Fig. 8. Boundary chart depicting the device and algorithm with the best performance for each input size tested.

GPU, and vice versa. We measured FPGA power by temporarily adding a single-FPGA GiDEL board, similar to the one in Novo-G, to the same system as the other configurations to keep comparisons fair, and because we are unable to measure the power of individual Novo-G nodes. The standalone FPGA power consumption was estimated as the difference between the FPGA FFT power and the idle system with no FPGA ($170 - 130 = 40$ watts).

The three charts in Figure 9 show energy consumption (excluding standalone FPGA), with parts (a), (b), and (c) showing small (50), medium (16050), and large (32550) kernel sizes, respectively. Energy is calculated by multiplying the execution time and measured wattage of the corresponding system. A similar methodology can be found in previous work (e.g., Baker et al. [2007]).

The results show that both GPU and both FPGA implementations were competitive when the signal size was medium to large and kernel size was small. The single-threaded CPU implementation had superior energy efficiency when both input parameters were small.

The frequency-domain CPU and GPU implementations were the most energy efficient for medium kernel sizes and small signal sizes, and the FPGA frequency-domain and GPU frequency-domain were most efficient for medium to large signal sizes. The GPU had a strong advantage for large kernel sizes with as much as 6x better efficiency than its closest competitor, the 8-threaded CPU.

The standalone FPGA implementation had superior energy efficiency for virtually all input parameters, excluding only kernel sizes above 28550 and the region defined by signal size less than 1024 and kernel size less than 550. The GPU had lower energy use in the former region and the CPU had lower energy use in the latter.

5.5. Discussion

This section includes a more detailed analysis and explanation of all performance comparisons, in addition to a discussion of limitations and future work.

5.5.1. Performance Analysis. After analyzing all results, we determined that the performance of a convolution implementation for a given input size can be explained by:

Fig. 9. Energy consumption—lower is better—for each implementation for (a) small kernel size of 50; (b) medium size of 16050; and (c) large size of 32550 at a variety of signal sizes. Both axes use a base-2 log scale.

Fig. 10. The importance of various hardware/software characteristics changes as a function of both kernel and signal size.

(a) the impact of kernel size on the efficiency of the chosen algorithm, (b) setup cost for initiation such as thread synchronization and bus transfers, (c) raw clock speed, and (d) parallelism.

The chart in Figure 10 shows how these four factors vary in importance with input parameters (lower numbers have a higher impact). Setup cost is most relevant for small input sizes and is amortized by computation as either input parameter increases. Kernel cost increases in importance for time-domain and overlap-save implementations as kernel size increases. Parallelism increases in importance as the signal size increases, giving an advantage to massively parallel systems that can process much of a signal simultaneously through either a pipeline or a multitude of threads. Clock speed is of greatest importance for medium kernel and signal sizes when the other factors are not at peak importance.

The success of the multicore implementations where both signal and kernel size are small can be explained by the minimal setup cost due to a lack of off-chip bus transfers. The 8-threaded implementation was slower than the single-threaded implementation for very small input sizes because of the setup cost of thread synchronization, and was faster in all other cases because of the increased available parallelism. Performance for both frequency-domain CPU implementations was highest when kernel sizes were small due to the overlap-save optimization.

The FPGA implementations had the highest startup costs due to the slow PCIe x8 bus, and the lowest clock speed at 175 MHz, but the highest parallelism. Therefore, the frequency-domain implementation saw its best performance for small to medium kernel sizes and medium to large signal sizes and the time-domain implementation was best for small kernel sizes and medium to large signal sizes.

The standalone FPGA execution of the frequency-domain implementation had far higher performance for all inputs, but especially so for small signal sizes because the setup cost represented a large percentage of performance in that region.

The GPU implementations saw high performance in all cases except when both inputs were small, due to the GPU's startup cost from the PCIe x16 bus. High performance in the remaining regions is due to the combination of relatively medium clock speed and parallelism.

The time-domain GPU implementation had its best performance when either input parameter was small due to the high cost of large kernel sizes on the time-domain algorithm. The frequency-domain implementation on the GPU performed best at medium to large signal sizes and all kernel sizes because of the strength of the GPU at executing FFTs. Performance was slightly higher for the small kernel sizes due to the computation savings of the overlap-save optimization.

The overlap-save optimization provided significant performance improvements for the frequency-domain implementations, particularly the CPU and GPU implementations that were able to select FFT sizes over 32,768. The optimization was most effective for small kernels and large signals. For example, a single-FFT approach with a 64M signal and 50-element kernel requires an 128M FFT, or 3.6 billion operations. In contrast, the optimization on the CPU selects an FFT size of 16K, which would only require 1.9 billion operations, for a speedup of 1.9x. The optimization becomes less effective as kernel size increases. For example, for a kernel size of 19550, the optimized approach would select a 256k FFT for a savings of only 1.4x at 2.6 billion operations.

5.5.2. Limitations and Future Work. Although this study evaluates numerous usage scenarios of convolution, it is not feasible to evaluate all scenarios in a single paper. The presented results are still useful to designers by simply eliminating devices that exceed constraints that we did not evaluate.

One issue that we did not consider was device cost. The largest FPGAs cost around \$10,000 [Altera 2011a], which is significantly higher than most GPUs and multicores, which are typically under \$1,000. Specialized GPUs can cost more than \$2,000 [NVIDIA 2011c]. It is difficult to come to generalized conclusions regarding performance per dollar as FPGA device costs vary significantly based on the size.

Another issue that we did not consider was application design complexity and design time. Although it is difficult to determine the exact design time of each implementation, we estimate that the CPU and GPU examples were created and optimized in approximately two weeks per implementation. The FPGA implementations required several months due to the need for custom register-transfer-level VHDL code, in addition to learning the FPGA board's interface. Although a designer familiar with an FPGA board could create such implementations significantly faster, in our experience the productivity for the FPGA was significantly lower than the other devices, which is supported by numerous previous studies [Che et al. 2008; Li et al. 2011; Merchant et al. 2008; Nelson et al. 2008].

An additional limitation was that the evaluated Stratix III E260 FPGA used an older 65nm process compared to a 55nm GPU and 45 nm CPU. Although ideally we would have analyzed identical technology nodes, it is difficult to select devices that use the same technology node due to different devices having different process schedules. For example, a GPU using the same process as a microprocessor would be commercially available years after the microprocessor, at which point designers would likely be using newer microprocessors. Therefore, we focused on the fastest devices available at the same time, which represents practical equivalency. The selected devices represent the state-of-the-art for 2009.

We also did not evaluate integrated, single-chip CPUs and GPUs because at the time of this study, such architectures were not common and were targeted only towards low-power embedded systems [NVIDIA 2011b]. Performance improvements for standalone FPGAs compared to PCIe FPGAs were significant and in many cases superior to all other devices, which could indicate that similar improvements are possible for integrated GPUs. Similarly, we did not consider ASICs, although a performance bound for the same circuits could be roughly estimated using faster clock frequencies.

Furthermore, we only considered floating-point operations. FPGA studies have shown order of magnitude improvements for integer operations [Williams et al. 2010]. Therefore, for cases where floating-point precision is not required, FPGAs may have significantly better results, which we will investigate as future work.

6. CONCLUSIONS

This article presented a detailed exploration of floating-point 1D convolution implementations, which included analysis of different devices, input sizes, optimizations, and algorithms. As expected, frequency-domain implementations generally outperformed time-domain implementations, except for when either the signal or kernel size was small. The frequency-domain multicore implementation was fastest when signal and kernel sizes were small and accelerator devices could not amortize high bus-transfer costs. For large signal sizes and kernel sizes below 12050, the standalone (i.e., not PCIe) frequency-domain FPGA implementation was the fastest implementation, although the GPU achieved comparable performance. For large kernel and signal sizes, GPU frequency-domain performance was up to 15.7x faster than a CPU baseline. For energy efficiency, the best implementations for each input size were similar to the performance results, with the FPGA frequency-domain implementation trailing the GPU by a small margin for many inputs. Interestingly, the standalone FPGA achieved the best energy consumption for almost all input sizes. Such results motivate emerging architectures that integrate GPUs and CPUs on a single chip, which could improve energy and data transfer. CPU-GPU processors will be evaluated in future work along with larger FPGAs and integer operations.

REFERENCES

- ALTERA, INC. 2011a. Buy FPGA and CPLD Devices, Stratix IV EP4SE820. 2010. <http://www.buyaltera.com/scripts/partsearch.dll/multisearch?site=ALTERA&lang=EN&keywords=EP4SE820>.
- ALTERA, INC., 2011. FFT MegaCore Function User Guide. http://www.altera.com/literature/ug/ug_fft.pdf.
- ARAYA-POLO, M., CABEZAS, J., HANZICH, M., PARICAS, M., RUBIO, F., GELADO, I., SHAFIQ, M., MORANCHO, E., NAVARRO, N., AYGUADE, E., CELA, J., AND VALERO, M. 2011. Assessing accelerator-based HPC reverse time migration. *IEEE Trans. Parallel and Distrib. Syst.* 147–162.
- BAE, S., CHO, Y. C. P., PARK, S., IRICK, K., JIN, Y., AND NARAYANA, V. 2011. An FPGA implementation of information theoretic visual-saliency system and its optimization. In *Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'11)*. 41–48.
- BAKER, Z. K., GOKHALE, M. B., AND TRIPP, J. L. 2007. Matched filter computation on FPGA, cell and GPU. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*. 207–218.
- BROOKWOOD, N., March 2010. AMD fusion family of APUs: Enabling a superior, immersive PC experience. http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf
- CHE, S., LI, J., SHEAFFER, J.W., SKADRON, K., LACH, J. 2008. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proceedings of the Symposium on Application Specific Processors (SASP'08)*. 101–107.
- DONG, Y., DOU, Y., AND ZHOU, J. 2007. Optimized generation of memory structure in compiling window operations onto reconfigurable hardware. In *Proceedings of the International Symposium on Applied Reconfigurable Computing*. 110–121.
- ELES, P., PENG, Z., KUNCHINSINSKI, K., AND DOBOLI, A. 1997. System level hardware/software partitioning based on simulated annealing and tabu search. In *Design Automation for Embedded Systems*. 5–32.
- FOWERS, J., BROWN, G., COOKE, P., AND STITT, G. 2012. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *FPGA'12: Proceedings of the ACM. SIGDA International Symposium on Field Programmable Gate Arrays* 47–56.
- FRIGO, M. AND JOHNSON, S. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2, 216–231.
- GAC, N., MANCINI, S., DESVIGNES, M., AND HOUZET, D. 2008. High speed 3D tomography on CPU, GPU, and FPGA. *EURASIP J. Embed. Syst.* 5, 1–12.
- GEORGE, A., LAM, H., AND STITT, G. 2011. Novo-g: At the forefront of scalable reconfigurable supercomputing. *Comput. Sci. Engin.* 13, 1, 82–86.

- GIVARGIS, T., VAHID, F., AND HENKEL, J. 2002. System-level exploration for pareto-optimal configurations in parameterized system-on-a-chip. In *IEEE Trans. VLSI Syst.* 416–422.
- GUO, Z., NAJJAR, W., VAHID, F., AND VISSERS, K. 2004. A quantitative analysis of the speedup factors of FPGAs over processors. In *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA'04)*. 162–170.
- HUANG, S., XIAO, S., AND FENG, W. 2009. On the energy efficiency of graphics processing units for scientific computing. In *Proceedings of the 5th IEEE Workshop on High-Performance, Power-Aware Computing*. 1–8.
- KESTUR, S., DAVIS, J. D., AND WILLIAMS, O. 2010. BLAS comparison on FPGA, CPU and GPU. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 288–293.
- KOEHLER, S., STITT, G., AND GEORGE, A. 2012. (to appear). Platform-Aware bottleneck detection for reconfigurable computing applications. *ACM Trans. Reconfig. Technol. Syst.*
- LE, H., JIANG, W., AND PRASANNA, V. 2011. Memory-efficient IPv4/v6 lookup on FPGAs using distance-bounded path compression. In *Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'2011)*. 242–249.
- LI, J., SARUNIC, M. V., AND SHANNON, L. 2011. Scalable, high performance fourier domain optical coherence tomography: Why FPGAs and not GPGPUs. In *Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'2011)*. 49–56.
- MANSOUR, D. AND GRAY, A., JR. 1982. Unconstrained frequency-domain adaptive filter. *IEEE Trans. Acoust. Speech Signal Process.* 30, 5, 726–734.
- MERCHANT, S., HOLLAND, B., REARDON, C., GEORGE, A., LAM, H., STITT, G., SMITH, M., ALAM, N., GONZALEZ, I., EL-ARABY, E., SAHA, P., EL-GHAZAWI, T., AND SIMMLER, H. 2008. Strategic challenges for application development productivity in reconfigurable computing. In *Proceedings of the IEEE National Aerospace and Electronics Conference (NAECON'08)*.
- NELSON, B. E., WIRTHLIN, M. J., HUTCHINGS, B. L., ATHANAS, P. M., AND BOHNER, S. 2008. Design productivity for configurable computing. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'08)*. 57–66.
- NVIDIA CORP. 2011a. CUDA CUFFT library. <http://developer.nvidia.com/cuda-toolkit-40>
- NVIDIA CORP. 2011b. NVIDIA tegra 2. <http://www.nvidia.com/object/tegra-2.html>
- NVIDIA CORP., 2011c. NVIDIA tesla workstations. <http://www.nvidia.com/object/personal-supercomputing.html>
- OWENS, J. D., HOUSTON, M., LUEBKE, D., GREEN, S.; STONE, J. E., AND PHILLIPS, J. C. 2008. GPU computing. *Proc. IEEE* 96, 5, 879–899.
- UNDERWOOD, K. D. AND HEMMERT, K. S. 2004. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *(FCCM'04). Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 219–228.
- WILLIAMS, J., GEORGE, A., RICHARDSON, J., GOSRANI, K., MASSIE, C., AND LAM, H. 2010. Characterization of fixed and reconfigurable multi-core devices for application acceleration, *ACM Trans. Reconfig. Technol. Syst.* 3, 4, 1–29.
- XIAO, S., AJI, A., AND FENG, W. 2009. On the robust mapping of dynamic programming onto a graphics-processing unit. In *(ICPADS'09). Proceedings of the 15th International Conference on Parallel and Distributed Systems*. 26–33.

Received July 2011; revised May 2012; accepted August 2012