Is There A Tradeoff Between Programmability and Performance?

Robert Halstead
University of California Riverside

Jason Villarreal
Jacquard Computing, Inc.

Roger Moussalli
University of California Riverside

Walid Najjar
University of California Riverside

***Abstract* — While the computational power of Field Programmable Gate Arrays (FPGA) makes them attractive as code accelerators, the lack of high-level language programming tools is a major obstacle to their wider use. Graphics Processing Units (GPUs), on the other hand, have benefitted from advanced and widely used high-level programming tools. This paper evaluates the performance, throughput and energy of both FPGAs and GPUs on image processing codes using high-level language programming tools for both.**

I. Introduction

FPGAs are commonly used as execution platforms for signal and image processing applications because they provide a good tradeoff between the programmability of CPUs and DSPs and the performance of ASICs. However, their programmability remains a major barrier to their wider acceptance by application code developers. These platforms are typically programmed in a low-level hardware description language, such as VHDL or Verilog, a skill not common among application developers and a process that is often tedious and error-prone. The Riverside Optimizing Compiler for Configurable Circuits (ROCCC) is a C-to-VHDL compilation toolset designed to raise the abstraction of FPGA programming.

GPUs, on the other hand, have seen the emergence of software tools (CUDA [5] and OpenCL [6]) that have bridged their programmability gap in recent years making them extremely attractive platforms for signal and image processing as well as high-performance computing. By extensively exploiting SIMD-style and multithreaded parallelisms they have demonstrated speed-ups over CPU that range in the two orders of magnitudes.

In this paper we evaluate the performance, programmability and energy consumption and the tradeoffs implied in using FPGAs and GPUs on a large set of image applications. We find FPGAs and GPUs offer comparable performance, and programmability in terms of pixels being output per cycles taken for computation, and lines of code written. We look at ideal executions on higher end platforms for both FPGAs and GPUs. We also find the real execution times of the GPU will be, on average, 46 times longer than the ideal due to memory accesses.

The rest of this paper is organized as follows. In Section II we explain the software we use to program our two platforms. Section 3 explains the benchmarks and the differences between their designs. Section 4 presents the data we collected, and some analysis explaining it. Section 5 provides conclusions.

II. Platforms and Languages

A. Programming FPGAs with ROCCC

FPGAs are commonly programmed using low-level hardware description languages such as VHDL and Verilog. Such languages require complete timing information and low-level details that software designers are traditionally unfamiliar with. Hardware implementations, however, may provide critical speedup to software applications [2], necessitating a way to overcome the long development times and programming overhead normally required to create FPGA implementations.

The Riverside Optimizing Compiler for Configurable Computing (ROCCC) [4] raises the level of abstraction for programming FPGAs to a subset of C, allowing developers to specify hardware designs in a familiar syntax. Through extensive optimizations, ROCCC transforms a subset of C into VHDL code that achieves similar throughput to handwritten VHDL with little overhead [3]. The ability to program FPGAs in a higher-level language not only enables software developers to utilize available hardware, but also increases user productivity and enables design space exploration from a single high-level source.

ROCCC supports the construction of hardware accelerators through a bottom-up design process where the user defines modular, reusable hardware blocks (referred to as modules) that can be combined and instantiated to create larger systems that process large amounts of streaming data. Each module may be a concrete computational block described in C, or imported into ROCCC from a preexisting source such as an IP core or netlist. All modules are stored in a database integrated with the compiler and managed by the Eclipse-based GUI. Module instantiations are available to be integrated directly into the high throughput hardware pipelines generated by

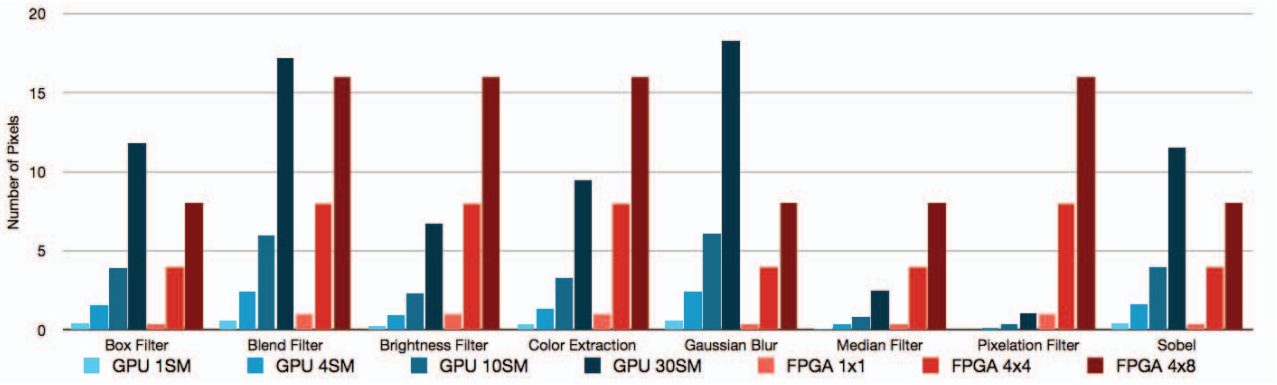

Fig. 1. The number of pixels that are output per cycle when varying the number of Streaming Multiprocessors on a GPU, and the data streamed in per cycle for the FPGA.

ROCCC as appropriate.

ROCCC performs extensive optimizations and program transformations targeting hardware on the C code. The purpose of these transformations is to generate circuits that maximize the throughput, minimize the number of off-chip memory accesses, and minimize the area used. The user is given fine-grained control over the extent of these transformations, which allows for design space exploration from a singular C source.

Many applications in signal, image and video processing consist of a sequence of well-defined and often standard operators applied to a data set or a stream of data. While it is possible to apply these compiler transformations to each operator separately, studies have shown that applying them to the whole application yields dramatically better results in terms of speed, area and throughput.

By allowing the user to import modules under three forms, ROCCC not only improves productivity by supporting code reuse, it also reduces design variability by allowing the use of tested cores in the forms of netlists. However, importing a module as a netlist or a VHDL code prevents the compiler from applying the above mentioned program transformations to the whole application.

ROCCC is not designed as a hardware description language and cannot be used to describe arbitrary circuits in C. Instead, the focus of ROCCC is on generating high performance hardware accelerators that perform many computations on streams of data.

Different platforms have different characteristics, including available area and bandwidth. Applications must be tuned to take advantage of the platform specifics. One of the most influential controls that ROCCC supports is through the control of loop unrolling. Users may specify exactly which loops are unrolled by what amount, unlike in a traditional compiler where all of these decisions are made across the whole program. Unrolling will adjust the number of data elements required per loop iteration, which in turn adjusts the necessary bandwidth for maximium throughput. Also, the user can control the number of incoming channels on a stream-by-stream basis, ensuring that incoming and outgoing data rates are maximized. The memory interface address generation and reuse of fetched data is handled by ROCCC.

The examples used in this paper operate on images that are treated by ROCCC as two-dimensional streams. Processing a sliding window on a two-dimensional stream is coded as a doubly nested for loop, with both the outer and inner loop's unroll factor separately controllable by the user. By unrolling either the outer or inner loop we can control the amount of data fetched each clock cycle in both the horizontal and vertical dimensions. In Section IV, a design specified 4x8 has been unrolled 4 times in the horizontal direction and 8 times in the vertical direction, which will require a 4x8 window of data each cycle to achieve maximium throughput.

B. Programming GPUs with CUDA

CUDA is a C API that facilitates the programming of GPUs, and helps incorporate their computational power into software applications. GPUs are targeted toward designs that are highly parallel, and require the same set of instructions to be executed over large quantities of data, i.e the single instruction multiple data (SIMD) paradigm. CUDA has become and valid way for developers to program GPUs, and speedup their designs [1].

Using CUDA a developer writes a computational kernel. This kernel is a set of instructions that should be executed on all necessary data. Each kernel reads data from global memory, executes the instructions and stores data back to global memory. The data is offloaded from the host's main memory to the GPU's global memory. The GPU is evoked to process the data, and writes its output again to its own global memory. When done the host can then read the data back.

A GPU will contain a number of Streaming Multiprocessors (SMs). These SMs are responsible for working on blocks of data. A GPU can have a different number of SMs. The Nvidia Tesla C1060 used in this work has 30 SMs. Each SM has a number of Streaming Processors (SP) that process threads of data inside each block, and unique caches for data and instructions. Each SM also has some shared memory that allows communication and sharing of data between each SP.

When writing a kernel for a GPU the developer must take

into consideration the number of SMs their GPU has, and how many SPs each SM has. The developer will then specify a number of blocks the data should be broken into, knowing that each of these blocks will go to one SM. They must also decide how many threads should exist in each block based on the capabilities of the GPU. These parameters determine how efficiently the developer is utilizing the GPU.

III. BENCHMARKS

In this paper, we make use of eight image-processing benchmarks (kernels) that we implement for both CUDA and ROCCC.

The benchmarks fall in two categories. In the first, the algorithm operates on a sliding window (typically 3x3) over the image, therefore resulting with an image two columns and two rows smaller than the original input. Box Filter, Gaussian Blur, Median Filter, Pixelation, and Sobel are examples of this first category. In the second category, the code operates on a single pixel at a time. Blend Filter, Brightness Filter and Color Extraction are the benchmarks in this second category.

Median Filter requires finding the median value on a window of pixels, to do this we used a modified version of the Even-Odd sort implementation as described in [9].

Pixelation lends itself easily to the explicit employment of the GPU's shared memory (on each SM) through making use of the *__shared__* CUDA construct. This resulted in considerable speedup over the base implementation that relies solely on global memory for shared content. On the other hand, on FPGAs, ROCCC detects data re-use and places common data in what is referred to as a smart buffer [3].

Finally, the C-like code is highly similar for both platforms (C for ROCCC and CUDA for GPUs), with minor exceptions such as the explicit instantiation of shared memory.

IV. EXPERIMENTS, RESULTS, AND DISCUSSION

In this section, we evaluate the computational performance and energy efficiency of several GPU and FPGA setups for each of the aforementioned benchmarks.

A. Experimental Setup

The input to each kernel is a 512x512 pixel image, with the exception of Blend Filter, where two 512x512 images are used as input. The output of the kernels is a single 512x512 image. Images are stored in the Netpbm [11] format, which helps reducing the computation overhead involved with locating pixels.

A total of 24 bits were used to represent RGB values, and 8 bits for grayscale. In all of the Blend, Brightness, and Color Extraction kernels, each of the red, green, and blue pixel values were treated as independent sets of data; in other words, three data sets were streamed to the FPGA circuitry, and three arrays were stored in global memory for the GPU. On the other hand, we were able to treat the RGB pixel as one 24-bit number in the Pixelation kernel, where the latter computes the average of red, green and blue values.

We offer the evaluation of four GPU setups, having 1, 4, 10 and 30 SMs respectively. The 30 SM GPU represents higher end GPUs, such as the Tesla C1060. Each SM consists of 8 SPs (as with the Tesla C1060), where each SP can ideally execute 4 instructions per cycle. A cycle accurate simulation of such GPUs was achieved using GPGPU-Sim [7]. Here, we configured the simulator to make use of ideal memories, having zero access latency; this applies to both global memory and local cache memories.

We compare the GPU setups to three FPGA implementations of the kernels, where the computation loop unrolling is varied (1x1, 4x4, 4x8). We made use of Xilinx's ISE 11.1 toolset to assess the area utilization and maximum operational frequency on a Xilinx Virtex 6 LX760 FPGA. We also utilized the built in cycle-by-cycle Xilinx simulator to evaluate the performance.

B. Pixels Output per Cycle

We show in Figure 1 the pixels that are output per hardware cycle, on each of the GPU and FPGA setups. The results assume ideal memory latencies for both GPUs and FPGAs; therefore, the resulting metric depicts solely the computational efficiency each the device. This comparison helps determine the performance of each device regardless of the implications of the memory hierarchy (latency, cache misses, etc…) specific to each platform.

As shown in Figure 1, as expected, the computational performance is following a linear progression for both GPUs and FPGAs while increasing the computational power. We can thus extrapolate the power of an FPGA if we were to use a larger design to account for the unused area. These extrapolations will be discussed later in the paper.

Note that a GPU with 4 SMs will fully utilize 32 cores executing on the image, which equates to 128 data elements being requested in a pseudo-parallel fashion. Unrolling the loops on the FPGA increases the parallelism similarly: as the loop is unrolled in the FPGA, we increased the number of data channels.

The largest GPU we simulated has 240 cores, each able to execute 4 instructions per cycle, thus processing 960 data elements in parallel. The FPGA is bringing in at most 32 data elements each cycle on the furthest unrolled design (4x8). This is far from the 960 data elements that the largest GPU is executing on, and is much closer to the GPU simulations with only 1 SM.

The largest designs for the FPGA approximately utilize 20% of the total resources available on a Virtex 6. Hence, unlike the GPU, the FPGA is not being used to capacity. The results in Figure 1 show GPUs and FPGAs are similar in the number of pixels output per cycle, and in some cases even out perform the GPU.

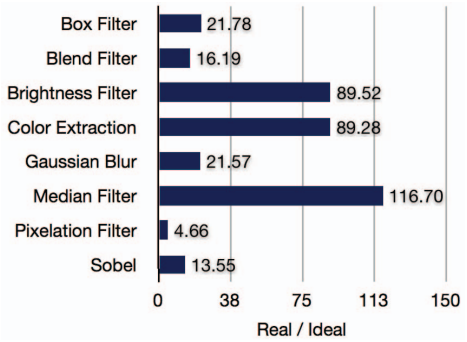

Fig. 2. The ratio of real execution time to ideal execution time for a GPU. The real execution time takes on average 46 times longer

C. Real versus Ideal Execution Time

Assuming the GPU can execute on 960 unique data elements every cycle is an unrealistic assumption that factors out cache misses, bus contention, and the global memory access latency. On the other hand, the simulations ran for the FPGA do take these into consideration since the FPGA does not utilize caches.

To compute the real execution time for the GPU we ran our designs on the Tesla C1060. To warm up the caches, we ran the designs multiple times; moreover, we show the average of several execution times for each kernel. To be consistent with the FPGA simulations, the measured execution time does not account for the offloading of data to and from the GPU to the host machine.

The ideal execution time is computed as the number of cycles a kernel executes, divided by the operational frequency of the underlying platform. We assumed the Tesla GPU operational frequency of 1.3GHz for all out GPU setups. The average clock frequency for our FPGA 4x8 designs was 109 MHz, and is higher for smaller (less unrolled) implementations.

Figure 2 shows the ratio of read to ideal execution time for the 30 SM GPU. On average, the real execution time is 46 times longer than its ideal counterpart. Closer inspection reveals that the ratio is larger in designs operating on color images where the colors are stored in separate arrays. While the ration is smaller in other designs, it still reveals considerable slowdown when compared to the ideal execution times. Since the CUDA implementation of the median filter kernel was highly similar to that of the FPGA design, both the execution time and ratio could be improved if the design targeted strictly GPU architectures.

D. Resource Utilization

When computation is done on a GPU all SMs are used. It is up to the developer to effectively partition the data knowing the underlying architecture of their GPU.

The FPGA developer has to create their design to utilize as much area as possible, keeping in mind that as designs for FPGAs increase in size and complexity, the maximum operational frequency will drop.

The total area of each kernel circuitry implementation on an FPGA averaged less than 20% of the overall resources available on the target Virtex 6. Therefore, we can extract more parallelism in most of these benchmarks through further unrolling, as the memory bandwidth permits.

E. ROCCC and CUDA Code Comparison

When looking at lines of code we only consider lines contributing to the execution of the kernel. We do not account overhead such as declaration of variables, library instantiation, and others. We also do not count lines of code used to fill in arrays and streams with data. In CUDA we would count lines computing the thread index, and for ROCCC we count lines initializing temporary variables.

Table 1 shows the code sizes. As we can see, both ROCCC and CUDA code are of highly similar volumes and structures.

TABLE I
LINES OF CODE

Kernel	ROCCC	CUDA
Box Filter	14	14
Blend Filter	11	11
Brightness Filter	16	16
Color Extraction	16	13
Gaussian Blur	14	15
Median Filter	84	79
Pixelation Filter	25	26
Sobel	29	20

The average lines of code, used to do computation, are similar in overall size. The ratio of ROCCC code to CUDA is 1.08.

The ratio for all ROCCC benchmarks to CUDA benchmarks is 1.08:1.

The ROCCC code is two to three orders of magnitude less than the generated VHDL code, which is used to program the FPGA. This VHDL code is generally invisible to the user and is comparable to the amount of VHDL code that would have had to be written by hand. Additionally, optimizations on the ROCCC code such as loop unrolling would have required complete rewrites of the VHDL code.

F. Joules per Pixel

We also examined the amount of energy the kernels used when processing their respective data sets. The Tesla C1060 GPU is rated at 187.8 watts of maximum power. As noted earlier, it also has an operational frequency of 1.3GHz. We take these as constant across all kernels designed for the GPU.

To measure the power consumption on the FPGA we used Xilinx's Power Estimator (XPE) tool [10]. Given a specific board, a clock frequency and the number of slices, BRAMs, and DSP blocks used XPE can estimate the power consumption. The tools are able to provide typical power consumption for a design, and maximum power consumption for a design. We used the Virtex 6 LX760 as our target FPGA.

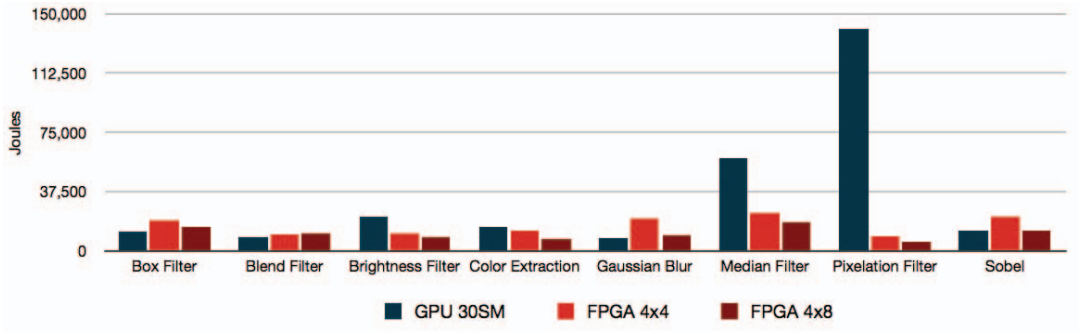

Fig. 3. The Joules per Pixel consumed by Nvidia Tesla C1060 GPU and Xilinx Virtex 6 LX760.

$$\frac{Watts}{(Pixels/Cycle)(Cycles/\sec)} = \frac{Joules}{Pixels} \quad (1)$$

We compared the Joules consumed per pixel for each device. Using Equation (1), we show the maximum Joules consumed per output pixel in Figure 3. As shown, the GPU will usually use more power than a 4x8 FPGA design; it is in fact more comparable to the 4x4 FPGA design in most cases.

V. Conclusion

In this paper we analyzed the computational efficiency of both FPGAs, and GPUs on 8 image processing kernels programmed from the same level of abstraction using ROCCC and CUDA respectively. While the kernels on GPUs with perfect memory accesses offer comparable performance to FPGAs, we found that the measured execution time of GPUs was 46 times longer than the ideal, leading to better performance on the FPGA. Additionally, the FPGA was not fully utilized, indicating that a fully utilized FPGA would achieve significant gains when compared to a fully utilized GPU.

Acknowledgment

This work was supported in part by NSF Awards CCR 0905509 and CCR 0811416.

References

[1] S. Che, J. Le, W. Sheaffer, K. Skadron, J. Lach. "Accelerating Compute-Intensive Applications with GPUs and FPGAs," Application Specific Processors, 2008. SASP 2008.

[2] Z. Guo, W. Najjar, F. Vahid and K. Vissers. "A Quantitative Analysis of the Speedup Factors of FPGAs over Processors," In. Symp. on Field-Programmable gate Arrays (FPGA), Monterrey, CA, February 2004.

[3] J. Villarreal, A. Park, W. Najjar and R. Halstead. "Designing Modular Hardware Accelerators in C With ROCCC 2.0," in The 18th An. Int. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), Charlotte, NC, May 2010.

[4] http://rocce.cs.ucr.edu/

[5] http://www.nvidia.com/object/cuda_home_new.html

[6] http://www.khronos.org/opencl/

[7] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, T.M. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator," Performance Analysis of Systems and Software, 2009. ISPASS 2009.

[8] Erik Lindholm, John Nickolls, Stuart Oberman, John Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro, pp. 39-55, March/April, 2008

[9] R. Mueller, J. Teubner, G. Alonso. "Data Processing on FPGAs" in VLDB August 2009.

[10] http://www.xilinx.com/products/design_resources/power_central

[11] http://netpbm.sourceforge.net/