# A Comparison Study on Implementing Optical Flow and Digital Communications on FPGAs and GPUs

JOHN BODILY, BRENT NELSON, ZHAOYI WEI, DAH-JYE LEE, and JEFF CHASE
NSF Center for High Performance Reconfigurable Computing (CHREC),
Brigham Young University

FPGA devices have often found use as higher-performance alternatives to programmable processors for implementing computations. Applications successfully implemented on FPGAs typically contain high levels of parallelism and often use simple statically scheduled control and modest arithmetic. Recently introduced computing devices such as coarse-grain reconfigurable arrays, multi-core processors, and graphical processing units promise to significantly change the computational landscape and take advantage of many of the same application characteristics that fit well on FPGAs. One real-time computing task, optical flow, is difficult to apply in robotic vision applications because of its high computational and data rate requirements, and so is a good candidate for implementation on FPGAs and other custom computing architectures. This article reports on a series of experiments mapping a collection of different algorithms onto both an FPGA and a GPU. For two different optical flow algorithms the GPU had better performance, while for a set of digital comm MIMO computations, they had similar performance. In all cases the FPGA implementations required 10x the development time. Finally, a discussion of the two technology's characteristics is given to show they achieve high performance in different ways.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*Design studies*; I.4.9 [**Image Processing and Computer Vision**]: Applications

General Terms: Experimentation, Design, Performance, Measurement

Additional Key Words and Phrases: Digital communications, FPGA, GPU, optical flow, reconfigurable computing

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 0801876.

Authors' addresses: J. Bodily, NSF Center for High Performance Reconfigurable Computing (CHREC), Brigham Young University, 459 CB, Provo, UT 84602; B. Nelson (corresponding author), NSF Center for High Performance Reconfigurable Computing (CHREC), Department of Electrical and Computer Engineering, Brigham Young University, 459 CB, Provo UT, 84602; email: brent_nelson@byu.edu; Z. Wei, D.-J. Lee, J. Chase, NSF Center for High Performance Reconfigurable Computing (CHREC), Brigham Young University, 459 CB, Provo, UT 84602.

**ACM Reference Format:**
Bodily, J., Nelson, B., Wei, Z., Lee, D.-J., and Chase, J. 2010. A comparison study on implementing optical flow and digital communications on FPGAs and GPUs. ACM Trans. Reconfig. Technol. Syst. 3, 2, Article 6 (May 2010), 22 pages. DOI = 10.1145/1754386.1754387.
http://doi.acm.org/10.1145/1754386.1754387.

## 1. INTRODUCTION

Developers have traditionally had just a few options at their disposal when considering how to implement a given computation. When performance requirements have allowed, programming a processor to do the computation has often been the preferred approach due to its lower NRE costs and risk as compared to developing a custom hardware solution. In cases where higher performance is required, FPGAs have become a common alternative, providing a custom hardware solution with its associated higher performance, but without the added risk, longer development time, and higher NRE costs of ASIC technology.

Successful computing applications on FPGA technology have often possessed many, if not all, of the following five characteristics.

(1) The computations use relatively simple data objects (example: fixed-point format data objects rather than floating-point data objects).
(2) The computations require relatively modest arithmetic (example: multiply and add rather than divide and transcendental functions).
(3) The computations are readily computed using pipelined processing structures.
(4) The computations contain extensive data parallelism.
(5) The computations contain regular and simple control structures, which often can be statically scheduled at compile time.

As many research papers have detailed over the past decade (too many to cite), when these conditions are met by an application, FPGAs can outperform programmable processors by one to two orders of magnitude. Signal and image processing is an example application domain which often meets many of these requirements, making it a good match for FPGA technology.

Recently, the computational landscape has been altered by the appearance of a number of new technology choices, ranging from FPGA-like technologies such as coarse-grain reconfigurable architectures to more CPU-like technologies such as multicore chips and GPUs, and a wide variety of devices in between. Importantly, many of these new technologies can exploit the same application characteristics that FPGAs rely on for high performance, and thus could be viewed as competing alternatives to FPGAs for some applications. Some of these new technologies promise significant improvements in arithmetic capability, particularly with regard to floating point performance. Additionally, while some of these technologies retain the HDL-like development approach of hardware design, others are software programmable and therefore may provide improvements in user productivity.

In this article we compare and contrast our experience creating FPGA and GPU implementations of a number of different signal/image processing applications. Our purpose in doing so is to try to provide insight into the relative capabilities of the two technologies for these computationally demanding applications, both in terms of raw performance and in terms of the design/programming effort required for each. Our results thus speak to a number of issues.

(1) What is the readily extractable performance of these two technologies for the chosen algorithms?
(2) What were the design experiences of similarly prepared designers when working with both technologies?
(3) Do FPGAs and GPUs provide high performance using the same or different mechanisms (pipelining, parallelism, static scheduling, etc.) for this set of applications?

We feel that the characteristics of the chosen applications (image and signal processing pipelines for both computer vision and digital communication) are representative of a large enough number of algorithms that our results, while not applicable in every case, do provide interesting and useful insights into some of the differences between FPGA and GPU development.

The platforms chosen to compare were currently available FPGA and GPU platforms which represent what would be required to completely field each of the applications. Since FPGAs are able to run stand-alone, no host was included. In contrast, GPUs must be hosted and so a combination of GPU and host were included in the comparison.

Our initial focus is a pair of real-time optical flow algorithms which are the highest performing and most accurate FPGA-based optical flow implementations we are aware of to date. We then turn to modules selected from a MIMO digital communication systems design.

## 2. OPTICAL FLOW AND RELATED WORK

Optical flow aims to measure the motion field in an image sequence from the motion of the pixel brightness pattern in that image sequence. Optical flow is widely used in 3D vision tasks such as motion estimation, obstacle avoidance, Structure from Motion (SfM), etc. The basic assumption of many optical flow algorithms is called the brightness constancy constraint, which is that changes in image brightness between frames are due only to camera or object motion. In practice, if the time interval between frames is small enough, other effects causing brightness changes (such as changes in lighting conditions) will be negligible, thus satisfying the constraint.

Historically, the processing time for optical flow algorithms was on the order of seconds or tens of seconds per frame using general-purpose processors. These long processing times have thus prevented optical flow algorithms from being used for most real-time applications, such as autonomous vehicle navigation and control.

In recent years, a number of different schemes have been proposed to implement optical flow algorithms in real time using pipelined and/or parallel processing system architectures. Using a pipelined image processor, Correia and Campilho [2002] proposed an algorithm and corresponding design which can process a test set known as the "Yosemite" sequence of 316x252 images at a rate of a little over 20 frames per second. Alternatively, FPGAs have been used to process larger images at faster speeds [Zuloaga et al. 1998; Martin et al. 2005; Diaz et al. 2006; Arribas and Macia 2001; Niitsuma and Maruyama 2005]. Optical flow has recently been implemented on a GPU in Strzodka and Garbe [2004], Zach et al. [2007], and Mizukami and Tadamura [2007]. Strzodka and Garbe [2004] uses an older, nonunified GPU architecture to perform 25 fps (frames per second) optical flow calculations to identify moving objects in a scene. Mizukami and Tadamura [2007] use an iterative method to calculate optical flow based on the classic Horn and Schunck algorithm, and report a time of 2.806 seconds for 3000 iterations of the algorithm (also for the Yosemite sequence). It is unclear from the paper if the application has to be run with a large number of iterations to be effective. Zach et al. [2007], also based on Horn and Schunck, reports performance of 30 fps for 320x240 images. Like the algorithm described in this article, the Horn and Schunck algorithm is a dense optical flow algorithm. None of these papers makes comparisons between FPGA and GPU implementations of optical flow. While a number of prior authors have compared FPGA to GPU on a variety of other computations [Baker et al. 2007; Cope et al. 2005; Howes et al. 2006], we are not aware of any prior work which compares FPGAs and GPUs on the optical flow computation.

This article is organized as follows. In Section 2, a tensor-based optical flow algorithm and its adaptation to hardware are reviewed. In Sections 3 and 4, the FPGA and GPU implementations of this algorithm are described and Section 5 discusses the achieved results. Section 6 details a second, more accurate optical flow algorithm based on ridge regression and discusses its implementation results. Section 7 then describes an additional set of mapping experiments from a different application domain: MIMO digital communications systems and the results achieved. Section 8 summarizes all of the results, drawing conclusions about the relative capabilities of these two technologies for the selected applications, and Section 9 concludes.

### 2.1 A Tensor-Based Optical Flow Algorithm

A sequence of images can be treated as a volume of data where $x$ and $y$ are the spatial components and $t$ is the temporal component. Object movement in the spatial-temporal domain will generate brightness patterns with certain orientations according to the brightness constancy constraint. Local orientation can be represented compactly using a 3D tensor. There are different types of 3D tensors based on different formulations [Johansson and Farneback 2002; Haussecker and Spies 1999; Farneback 2000a; 2000b]. One of these, the gradient tensor, is representative of optical flow algorithms that map well onto FPGA hardware. We chose it for this design because it is similar to the classic

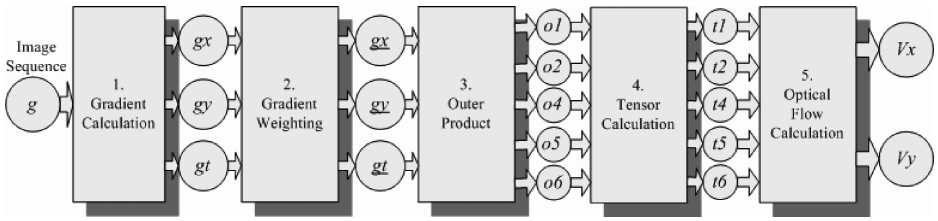


Fig. 1. Data flow of the design.

Lucas-Kanade algorithm and is a good fit for pipelined hardware. In addition, for simple signals such as those that occur in optical flow, the gradient tensor algorithm gives estimates similar to the more complex polynomial tensor [Johansson and Farneback 2002]. The specific tensor-based optical flow algorithm that we used is described in detail in Wei et al. [2007] in terms of its mathematical derivation and comparison to previously published optical flow algorithms. In this section, rather than repeat the derivation of the algorithm, we simply summarize the calculation.

The tensor-based optical flow algorithm is a five-stage computation as shown in Figure 1. The first stage performs 1D convolutions in the $x$, $y$, and $t$ dimensions using five-element masks. The mask coefficients were chosen to be powers of 2 to simplify the hardware implementation with little (and acceptable) image processing performance degradation. The results of these convolutions are gradient images $gx$, $gy$, and $gt$.

The second stage (Gradient Weighting in Figure 1) then performs seven-element row and column convolutions in succession on each of $gx$, $gy$, and $gt$ to produce weighted gradient images $\underline{gx}$, $\underline{gy}$, and $\underline{gt}$.

The third processing stage (Outer Product in Figure 1) produces five new matrices ($o1, o2, o4, o5, o6$) by multiplying various combinations of the weighted gradient matrices together element-wise ($o1 = \underline{gx} \times \underline{gx}$, $o2 = \underline{gy} \times \underline{gy}$, $o4 = \underline{gx} \times \underline{gy}$, $o5 = \underline{gx} \times \underline{gt}$, and $o6 = \underline{gy} \times \underline{gt}$).

The fourth stage (Tensor Calculation in Figure 1) performs a three-element row convolution followed by a three-element column convolution on each of these matrices to produce five tensor matrices ($t1, t2, t4, t5, t6$). The final stage (Optical Flow Calculation in Figure 1) then computes the $x$ and $y$ velocity flow fields $V_x$ and $V_y$ using the following formula.

$$V_x = \frac{(t_6 t_4 - t_5 t_2)}{(t_1 t_2 - {t_4}^2)} \tag{1}$$

$$V_y = \frac{(t_5 t_4 - t_6 t_1)}{(t_1 t_2 - {t_4}^2)} \tag{2}$$

The original design was simulated in Matlab. Since the first implementation of the design was to be done in an FPGA, the Matlab simulations were written to be bit-accurate models of the ultimate hardware design. This allowed them to be used to precisely evaluate the accuracy of the eventual fixed-point hardware solution. It also provided intermediate variable values to use in the hardware debug and verification process. Once the Matlab-based

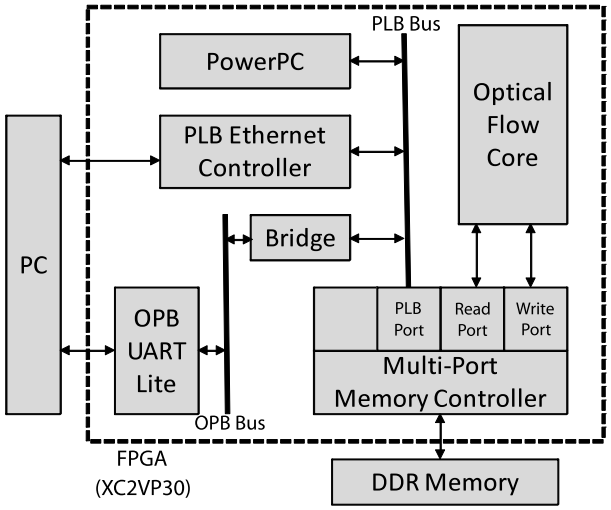


Fig. 2. System diagram.

accuracy/complexity trade-off studies were done to arrive at a final design, two implementations were created: a fixed-point FPGA implementation and a single-precision floating point GPU implementation.

## 3. FPGA IMPLEMENTATION OF OPTICAL FLOW

The design was first mapped to the Xilinx XUP V2P board. As shown in Figure 2, the on-board FPGA is a Virtex-II Pro XC2VP30 with 13,969 slices. The board also contains 256MB of off-chip DDR memory. The design was implemented using VHDL and the Xilinx EDK tools. As shown in the system diagram in Figure 2, the FPGA communicates with a host PC through both Ethernet and serial ports. These ports are used for high-speed data and control/debug information transfers, respectively. The design accepts 640x480 16-bit YUV images, which are read by the PPC into memory through the Ethernet interface. They are then read from memory into the optical flow module for processing where only the Y component is processed.

A Multi-Port Memory Controller (MPMC) architecture was chosen which provides four separate memory ports to the FPGA fabric using lightweight interfaces. This is shown in Figure 2. As can be seen in the figure, one of the memory ports is connected to the PLB bus to provide PLB-based memory access, and the optical flow core is directly connected to two of the remaining ports. Bypassing the PLB bus for memory access in this way leaves the bus free for other communication tasks within the system. Interfacing with the MPMC is also simpler and more efficient than interfacing with the bus, resulting in a smaller and higher-performance optical flow core module.

The optical flow core was implemented in a fully pipelined fashion to maximize throughput. The fixed-point bit widths used were custom-selected at each stage of the processing pipeline and range from 13 bits to 32 bits (fixed point). Lookup table-based dividers were used instead of pipelined fixed-point dividers without any adverse impact on accuracy.

The system clock rate for this FPGA implementation was limited to 100 MHz by the use of IP cores from the Xilinx EDK. The implementation used 10,288 slices, or about 75% of the total slices available. It can process 640x480 frames at a little over 64 frames per second, equivalent to about 258 fps for 320x240 images. Additional details can be found in Wei et al. [2007].

A common measure of accuracy for optical flow computations is average angular error compared to ground truth, computed across the optical flow field. Our FPGA implementation results in an average angular error on the test sequence of 12.9 degrees. To the best of our knowledge, the only previous error analysis for optical flow implemented in an FPGA is given in Diaz et al. [2006], which was developed using Handel-C and achieved an average angular error of 18.3 degrees on the same test sequence. It processed 320x240 images almost an order of magnitude slower than our design (30 fps versus 258 fps) and used approximately 19,000 slices, almost twice as many as our design. This design used floating point instead of the fixed point of our implementation and spent nearly 80% of its area on its least squares matrix module.

### 3.1 Design Enhancements

Our analysis shows that a number of performance improvements would be possible for our design to provide additional performance beyond the 64 frames per second outlined before. Memory bandwidth was the limiting factor in this design. A number of steps could be taken to significantly improve memory bandwidth and therefore processing speed, including: (a) stripping out the unused YUV components from the image stream prior to storing the images in memory and (b) using a custom memory interface module in the design. Our analysis suggests that a processing rate of over 200 frames per second would have been achievable for this computation with the same architecture and platform that was originally used, but with these changes applied.

Further, as has been amply documented elsewhere, one of the key features of FPGA-based processing is its flexibility. This is evidenced by the variety of FPGA-based processing systems available, many of which greatly exceed the cost of the platforms used in this work. Additional performance increases would thus be possible using such a platform with multiple memories attached to multiple FPGAs. This could be either a custom-designed platform or a commercial platform.

## 4. GPU IMPLEMENTATION OF OPTICAL FLOW

The GPU is a massively parallel computing device, originally developed to operate on the many pixels of an image in parallel. GPU devices are being used more and more for applications outside the realm of computer graphics, something called general-purpose GPU (GPGPU) computing. Until recently, using a GPU for general-purpose computing required that the programmer use a graphics API to interact with the GPU computational elements. Newer GPUs, however, now provide a more generalized computing architecture and development environment where the GPU can be viewed as a somewhat general set of parallel processing elements. Newer GPUs, however, now provide a more

generalized computing architecture and development environment where the GPU can be viewed as a somewhat general set of parallel processing elements. The GPU implementation of this design was created for an NVIDIA GeForce 8800 GTX platform. The host for the GPU was an Intel Xeon 1.86 GHz machine with 1GB RAM.

The GPU used contains 128 processing elements, arranged into 16 multiprocessors with 8 SIMD elements each. The GPU is a coprocessor, meaning that a host program prepares data, copies it to the GPU memory, and then launches computational kernels which run in the GPU. The resulting data is then copied back to the PC memory from the GPU device memory for further analysis or display.

The GPU kernels are written in an extended version of C and compiled by the NVIDIA-provided cross compiler. A kernel's computation is subdivided into blocks, each of which executes entirely within one of the 16 multiprocessors and independently from other blocks in the kernel (no communication or synchronization between blocks is possible). Blocks are further subdivided into lightweight threads, groups of which are executed in a multiprocessor's SIMD elements. Threads within a single block communicate via shared memory and synchronization calls. Each thread can determine, at runtime, its block and thread index and thus which subset of the data to operate on.

The programmer, in general, is responsible for manually managing the GPU's memory hierarchy. The GPU platform's global memory (called device memory) is DRAM. It is large (100's of MB) but has access times measured in multiple 100's of clock cycles. It is optimized for block transfers; concurrent sequentially addressed memory accesses are *coalesced* by the hardware into larger block transfers, greatly improving performance. Closer to the processing fabric, each multiprocessor contains its own on-chip local store, called shared memory. Shared memory is small (10's of KB) but can provide single-cycle access. It is optimized for concurrent access; it can satisfy up to 16 concurrent memory accesses per cycle provided no bank conflicts occur (it is 16-ported). Coding an application properly to enable device memory coalescing and to avoid shared memory bank conflicts are thus crucial tasks to achieving performance on a GPU.

### 4.1 Our GPU Implementation of Optical Flow

Since we were already familiar with the gradient tensor algorithm from our work on the FPGA implementation, we chose to map it directly to the GPU. Using the same algorithm on both platforms allowed us to directly compare the implementations and results. While we did not explore alternative GPU optical flow algorithms directly, Diepold et al. [2006] suggest that algorithms similar to the classic Lucas-Kanade algorithm, like ours, may be the best fit for graphics hardware due to the large number of matrix-vector operations performed and inherent parallelism.

In our GPU implementation of optical flow, each input image is divided into tiles which are independently processed by the multiprocessors in the GPU. To keep each kernel simple, we wrote a different kernel for each operation in

Table I. Best Tile Sizes for Each Optical Flow Kernel with 640x480 Images

| Kernel Name | Best Tile Width | Best Tile Height |
|---|---|---|
| Gradient X | 320 | 1 |
| Gradient Y | 16 | 20 |
| Gradient Z | 16 | 4 |
| Weight X | 320 | 1 |
| Weight Y | 16 | 18 |
| Outer Product | 32 | 2 |
| Tensor X | 320 | 1 |
| Tensor Y | 16 | 20 |
| Flow | 32 | 4 |

the pipeline. Thus, the three convolutions in the Gradient Calculation stage of Figure 1 were each implemented as separate kernels, each of the six Gradient Weighting stage calculations were computed using different kernels, etc. A single frame computation thus requires 21 kernel calls; the intermediate data results are left in device memory between kernels.

Each of our kernels initially had a high ratio of memory accesses to computations. Therefore, we focused our optimization work on memory accesses. To enable alignment for coalescing of device memory accesses, the width of each image row was padded to be a multiple of 128 bytes; this seemingly simple modification provided a 1.5× speedup over the original implementation.

The second optimization related to how the image was broken up into tiles (blocks) for processing. Our initial (and naive) implementation used one block per row for the row convolutions and one block per column for the column convolutions. This used memory inefficiently since column memory accesses were always to a single pixel at a time and therefore could not benefit from coalescing. We then tested a family of tile sizes for each kernel and the best results for each are shown in Table I. As can be seen, different tile sizes (and therefore block and thread organizations) were found to be optimal for each of the various kernels. These tile sizes resulted in a 2× speedup for the column convolution kernels. Performing the tests to determine these optimal tile sizes was almost trivial, as were the final code changes required. Finally, the grid size can be computed at runtime, making the resulting code parameterizable for the size and shape of the input stream.

The resulting GPU implementation processes 640x480 frames at 238 frames per second and 320x240 frames at 847 frames per second.[1]

## 5. OPTICAL FLOW RESULTS

Both the FPGA and GPU implementations were tested with the Yosemite sequence. Because the image sizes in this sequence are 316x252, they were padded to 640x480 before being processed since the FPGA could only process

[1]In our recent paper [Chase et al. 2008] *Real-Time Optical Flow Calculations on FPGA and GPU Architectures: A Comparison Study*, our reported processing rates for this calculation were lower than reported here. A recently updated memory allocation function in NVIDIA's SDK reduced memory copy times between the host and GPU, resulting in the increased performance numbers reported here.

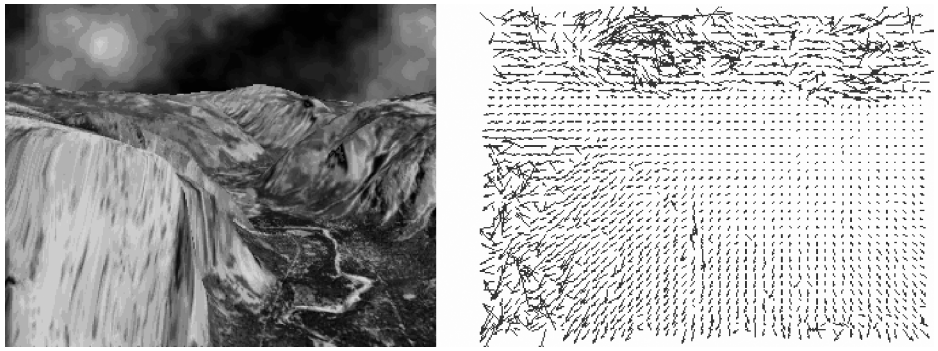

Fig. 3. The Yosemite sequence and the measured optical flow field: 8th frame of Yosemite sequence (left) optical flow field (right).

images of that size (the accuracy results quoted in what follows are for the original 316x252 image area, however).

Figure 3 shows one frame of the sequence and the resulting optical flow field computed by the FPGA implementation. A comparison of the accuracy between the FPGA and GPU shows that the GPU has a slightly better accuracy, with an average angular error of 12.1 degrees. This is due to the fact that the FPGA implementation is fixed point and the GPU implementation is single-precision floating point. The two implementations can be further compared in a number of ways including power, memory bandwidth, cost, flexibility, etc.

*Performance.* As described in previous sections, the reported FPGA design could be modified, resulting in an estimated performance of approximately 200 frames per second, but at increased cost (platform cost + development cost).

*Cost.* In this experiment, platform cost comparison was problematic due to the wide range of choices available for FPGA platforms, whether a standard or custom platform was used, how platform development costs versus volumes were computed, etc. The purchase cost of the FPGA platform used was $1,600 while the GPU cost was approximately $500. The FPGA platform was a prototype board with some unused I/O interfaces; a more custom and dedicated platform might cost less. However, upgrading that platform to technology comparable to the GPU's (Virtex-4 or 5, for example) would increase its cost since leading-edge FPGA's often cost $1,000's, with corresponding commercial platforms costing $10,000's. On the other hand, the GPU requires a host PC for operation while an FPGA platform might be stand-alone.

*Power and embedability.* The 8800 GTX GPU and its host together consume 200–300W of power and thus the GPU is unsuitable for many embedded applications. In contrast, our FPGA implementation of the optical flow algorithm consumes approximately 10W. This significantly lower power is typical of stand-alone FPGA solutions and allows them to be readily embedded into many size/weight/power constrained applications.

We also ran the optical flow code on a system with an NVIDIA GeForce 8400 GS video card. The 8400 GS has two multiprocessors (compared to the 16 in

the 8800 GTX). It provided a performance of 19 fps for the 640x480 image size (71 fps for the 320x240 size) with a power consumption of approximately 25% that of the 8800 GTX.

*Memory architecture.* The FPGA implementation must work out of off-chip memory for all image reads and writes due to insufficient on-chip memory. The bandwidth provided to this memory for the FPGA implementation described earlier is a few GB/sec at most. The GPU's DRAM is also off-chip. Host-to-GPU memory transfer rates were measured at just over 2GB/sec while the bandwidth between that GPU memory and the GPU fabric is more than 80GB/sec, but with very long access latencies.

*Flexibility.* FPGA memory interfaces, in contrast, have very low latency, providing algorithmic flexibility over a GPU implementation. Further, an FPGA platform can be a more flexible solution in terms of interfacing with the rest of an embedded system; an FPGA solution can include all required I/O interfaces and data processing in various and flexible organizations and form factors.

*Design productivity.* Part of the price paid for this FPGA flexibility is in design productivity. The FPGA implementation required more than 12× the design effort of the GPU implementation (an undergraduate for two weeks versus two graduate students for an entire summer). Further, the skill and experience level possessed on the part of the FPGA developers in this experiment was significantly higher for the FPGA implementation than for the GPU implementation.

We believe the main factor accounting for the design time differences between the two technologies had to do with the ease of performing design iterations. GPU design iterations are almost instantaneous, making it possible to iterate much more efficiently to arrive at an improved solution or to investigate radically different algorithms for an application. For example, searching for optimal block and thread size partitions for the various GPU kernels was done with simple parameter changes and was accomplished in a few hours. In contrast, the resulting FPGA designs were not nearly as parameterizable; alternative implementations were much more difficult to test out (simulate) as a part of *what-if* design activities. Further, hardware debug using logic analyzers and similar tools was significantly more cumbersome than the software debug model supported by the GPU platform. This difference in design time is perhaps the most striking result of this comparison.

## 6. A SECOND OPTICAL FLOW ALGORITHM BASED ON RIDGE REGRESSION

While the average angular error of 12.9 degrees for our optical flow implementation is more accurate than other FPGA implementations that we are aware of in the literature, it doesn't compare well with software-based algorithms in terms of accuracy, which can achieve an average angular error of around 1 degree [Farneback 2001]. The discrepancy in accuracy between software and our previous implementation motivated us to implement a second, more accurate

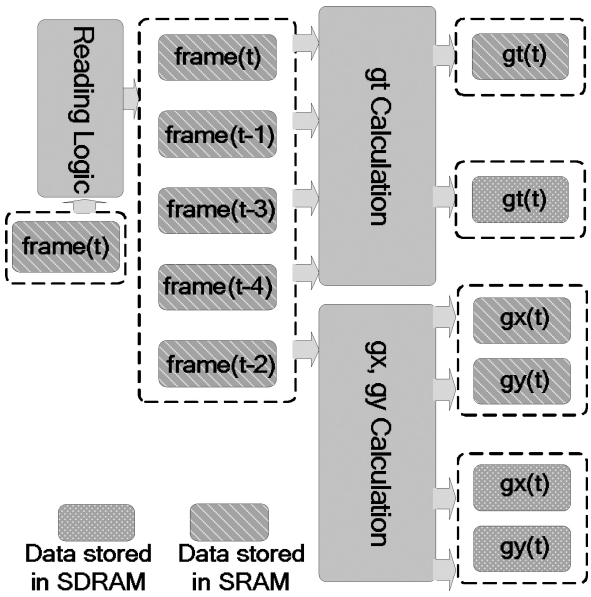


Fig. 4. Derivative calculation (DER) module.

(and computationally demanding) optical flow algorithm. The details of this algorithm are described in Wei et al. [2008]; only a summary is provided here.

The algorithm is based on the observation that optical flow algorithms require strong smoothing to suppress noise in the images and to extract accurate motion information. This second optical flow algorithm uses a unique hardware structure to incorporate temporal smoothing and increase the accuracy of the solution. Additionally, ridge regression [Hoerl and Kennard 1970] is used in the algorithm to solve the collinear problem that arises in the traditional least squares method of calculating optical flow. This collinear problem occurs when two vectors used in the calculation are nearly linearly dependent, with the result that even small levels of noise lead to large and inaccurate motion vectors.

This new algorithm is similar in many ways to the original algorithm presented in the previous section, and is made up of multiple stages of processing. The first stage is a more complex derivative calculation as seen in Figure 4. In this stage, three distinct sets of derivative frames *gx*, *gy*, and *gt* are calculated from raw input images using 1D convolutions.

Figure 5 shows the remainder of the computational pipeline. After the derivative frames are calculated, each set of frames is smoothed temporally with a three-element convolution, resulting in a single smoothed frame each for *gx*, *gy*, and *gt*. Five-element row and column convolutions are then applied to these frames in a spatial smoothing step.

The smoothed derivatives are combined to build ridge regression model components in a stage similar to the outer product stage of the original algorithm. The result is six new matrices (*gxgx*, *gygy*, *gtgt*, *gxgy*, *gxgt*, and *gygt*). Another spatial smoothing is performed on these components by way of three-element

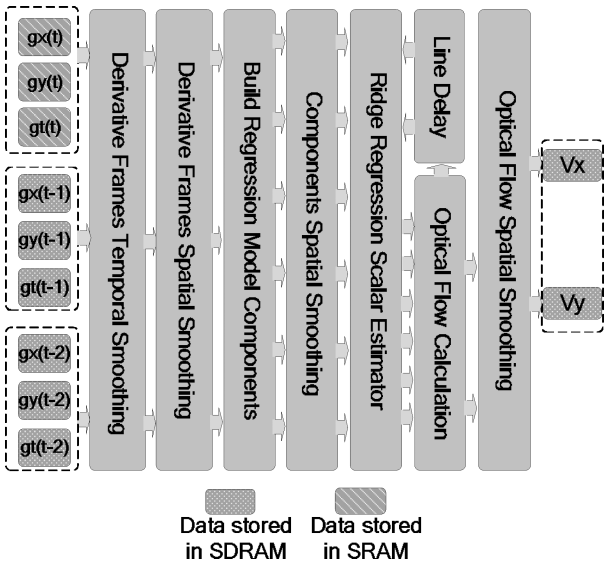


Fig. 5. Optical flow calculation (OFC) module.

row and column convolutions. The smoothed components are fed into the ridge regression scalar estimator (detailed in Farneback [2001] and Hoerl and Kennard [1970]) to calculate the scalar $k$, which is then used with the smoothed model components to calculate the optical flow fields as in Grob [2003]. The result is smoothed once more using seven-element spatial convolution to produce the final optical flow vectors, $V_x$ and $V_y$. As with the original algorithm, the design was simulated in Matlab and implemented on an FPGA and GPU.

### 6.1 Modified FPGA Implementation

The modified optical flow design was mapped to the BYU Helios FPGA platform, which consists of a Virtex-4 FX60 FPGA with two PowerPC processors, and is specially designed for embedded vision applications.

We used Xilinx's Base System Builder for the initial design of the system. The IP cores used limited the clock speed of the system to 100 MHz (our custom-designed optical flow module itself could be run stand-alone at 149MHz). In our system, various modules are connected to two buses, as shown in Figure 6. The PLB bus connects high-speed modules including the derivatives calculation (DER), optical flow calculation (OFC), and SDRAM. Lower-speed modules are connected to the OPB bus. The PLB and OPB buses are interconnected through a bridge.

Data flow in the system is directed by software running on the PowerPCs. The DER module is triggered when a new image frame is captured by the camera and has been stored in the SDRAM. After processing, results from the DER module are stored in SRAM and SDRAM. The DER and OFC modules share the high-speed SRAM through a multiport SRAM arbiter. The OFC module

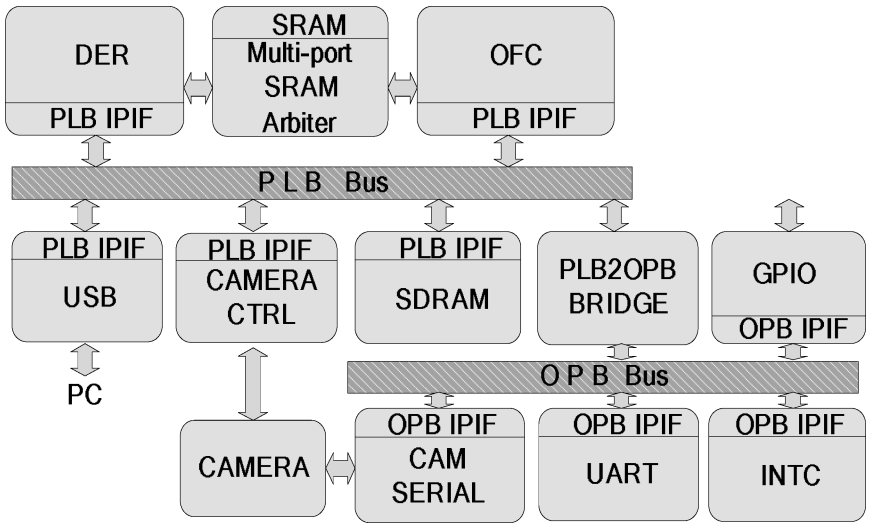


Fig. 6. System diagram of the modified optical flow algorithm.

then processes the intermediate results and stores the resulting motion vectors in SDRAM.

The DER module diagram, shown in Figure 4, calculates the derivative frames gx, gy, gt from raw images and stores the results in both SRAM and SDRAM. The reason these results are stored in both memories is that they are temporally smoothed in the OFC module and there is not enough space to store them in the hardware pipeline (SRAM). The OFC module then implements the processing pipeline shown in Figure 5. A drawback of this scheme is the increase in the memory throughput required and the resulting decrease in processing speed. However, as long as these modules are able to keep up with the camera frame rate, temporal smoothing can substantially improve the accuracy without impacting real-time performance.

### 6.2 Modified GPU Implementation

As with the original algorithm, the GPU implementation of the more accurate optical flow algorithm was implemented using a GeForce 8800 GTX. The implementation required one kernel call for each operation in the pipeline, for a total of 36 kernel calls. In the FPGA version, the calculation of the scalar $k$ is dependent on the value of the velocity of the pixel to the left of the pixel for which $k$ is being calculated. For the GPU version, this was modified so that the velocity of the current pixel was first calculated using $k = 0$. $k$ was then recomputed based on the resulting velocity instead of the velocity of the neighboring pixel. In this way, dependencies between pixels were removed and GPU parallelism was increased, since each pixel of the result could be calculated with a unique GPU thread. As the results show, the accuracy of the final optical flow field result was not adversely affected by this change.

### 6.3 Modified Optical Flow Results

*Accuracy.* The FPGA version of this new algorithm results in an angular error of 6.8 degrees on the Yosemite image sequence compared to 12.9 degrees for the original FPGA algorithm (which, to the best of our knowledge, was the most accurate version of optical flow on an FPGA at the time of its implementation).

The GPU version achieves an even higher level of accuracy of 5.3 degrees, due to the use of single-precision floating point data for the storage of intermediate results in the computation. In comparison, the FPGA version uses fixed-point data and adjusted bit widths throughout the algorithm to optimize resource usage.

*Speed.* While the FPGA implementation of the original algorithm can process 640x480 images at 64 frames per second, this second algorithm calculates only 15 frames per second for that image size. The additional smoothing required in the new algorithm significantly increases the required memory bandwidth. Our analysis shows that a larger SRAM on the Helios board could likely provide an additional 2× performance increase to about 30 frames per second for 640x480 images.

The GPU implementation of the modified optical flow does not suffer from these same problems. It processes frames at 158 frames per second for 640x480 images, compared to 238 frames per second for the original GPU optical flow algorithm. The slowdown is due to the increased computation required in the modified algorithm, which uses 36 kernel calls for each image compared to 21 in the original.

*Cost and development time.* Cost and development time results for the more accurate algorithm closely matched those of the original implementation. The Helios platform costs $2,000 per board when purchased at low quantities, compared to the $1,600 system used for the original implementation. It could cost as low as $1,000 per board if purchased in bulk.

For both FPGA implementations the designers were experienced (graduate student) FPGA designers. The GPU programmers were similarly graduate students, but their GPU programming capabilities were less well-developed than the design abilities of the FPGA designers. For this second optical flow algorithm, the FPGA version took approximately 10× the design effort of the GPU version, similar to the 12× difference in effort for the original algorithm. It is likely that the development time for the FPGA version would have been much higher had we created an entirely original design rather than using IP cores.

*Performance differential for ridge regression optical flow.* The GPU implementation of the original optical flow algorithm is about 3.8× faster than the FPGA implementation for the 640x480 image size. For the modified algorithm, the GPU processes images nearly 10.5× faster than the FPGA (5.3× faster if the Helios system had a larger SRAM).

Since both GPU implementations were based on the corresponding FPGA designs, we expected a similar performance differential between GPU and

FPGA for the two algorithms. However, the differential for the second algorithm is larger than that of the original algorithm due to at least two factors. First, the GPU performs better on computations which have higher compute-to-I/O ratios, and the ridge regression algorithm has such a higher ratio compared to the original optical flow algorithm. Second, the ridge regression FPGA implementation was designed for an embedded, low-power computer vision platform with limited resources. We estimate that additional resources on the Helios board (larger memories in particular) would increase its performance, perhaps to the same performance differential compared to the GPU as was observed for the original algorithm.

## 7. MIMO COMMUNICATION SYSTEM RESULTS

Since doing the work described earlier, we have completed additional comparisons between FPGA and GPU technology using three computations taken from a MIMO digital communication system design which we are currently completing. This system is intended for use in aircraft telemetry applications and transmits two independent (but mathematically related) data streams from two source antennas which are then received by a single ground station antenna and demodulated in an FPGA platform. In each case, the GPU implementation was completed with significantly less design effort than the FPGA implementation, similar to the results obtained before. However, the performance results were mixed and help to further illustrate differences between the capabilities of the two technologies.

### 7.1 Viterbi Decoder

The Viterbi trellis decoder is the final stage of the demodulation process and is used to decode a sequence of bits from heavily processed input samples. The trellis is an 8-stage multistage graph where each stage consists of between 16–128 nodes (states). In a conventional trellis the number of states per stage and the connection pattern between stages remains constant. However, in this system the combination of space-time code with the modulation scheme used requires that the trellis structure be time-varying and that each stage have a different number of states and interstage connection pattern, as shown in Figure 7. Full details on the reasons for the need for a time-varying trellis can be found in Alamouti [1998]. The characteristics of the trellis made a straightforward implementation difficult for both technologies.

The trellis inputs eight complex values at a time and then passes them through the trellis structure of Figure 7. The result of one pass through the pipeline is then used as the starting state for the next pass when a new set of eight complex samples is also input. This dependency prevented the trellis from being implemented in a high-performance hardware pipeline. The input data is organized into *frames*, each of which requires 800 such passes to complete, at which point the trellis can be reset and the process starts over. The design was implemented on a Xilinx Virtex II Pro clocked at 206 MHz using 8,790 slices and just meets the timing requirement of 320$\mu$s per frame.

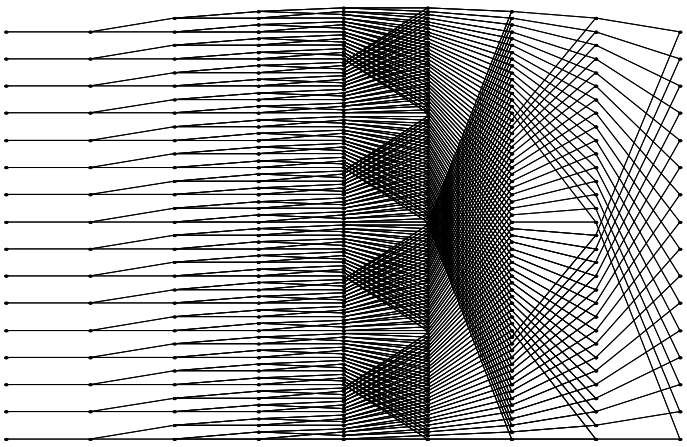

Fig. 7. The eight-stage trellis detector.

The initial GPU implementation processed one stage at a time, computing all of the state values in that stage in parallel. The irregular interconnections between stages did not pose a particular problem because intermediate results between stages were stored in shared memory and so only bank conflicts were a concern. On the other hand, there is only a modest amount of parallelism in each stage computation. Further, it could only use one of the 16 multiprocessors in the GPU architecture due to data dependencies. The initial implementation thus required almost 32× the allowed time for the computation.

Since input frames are processed independently, however, the unused GPU resources could be applied to process multiple frames in parallel. Thus, in a second implementation, 32 frames of samples were buffered and then operated on by the GPU in parallel. This provided enough total computation and latency for the GPU to improve its performance. The result surpassed the FPGA throughput by approximately 20%, but increased processing latency by 32×. Had each frame not been independent, this would not have been a feasible solution.

### 7.2 Timing and Channel Estimator

The timing and channel estimator block precedes the Viterbi module in the processing chain and its job is to estimate the timing and channel effects present in the two received signals. The most computationally demanding part of this requires evaluating an error surface and locating the minimum point on that surface. It uses a classic 2D simplex algorithm for its search. The simplex algorithm begins by choosing three points, each of which represents a vertex of a triangle on the surface. It calculates the error values at each point, compares them, then flips or rotates the triangle away from the largest of the three values. It continues to flip the triangle across the surface in this way until the global minimum of the surface is found (in our case the surface is a grid of 4096 discrete points). Due to the convex shape of the error surface, the

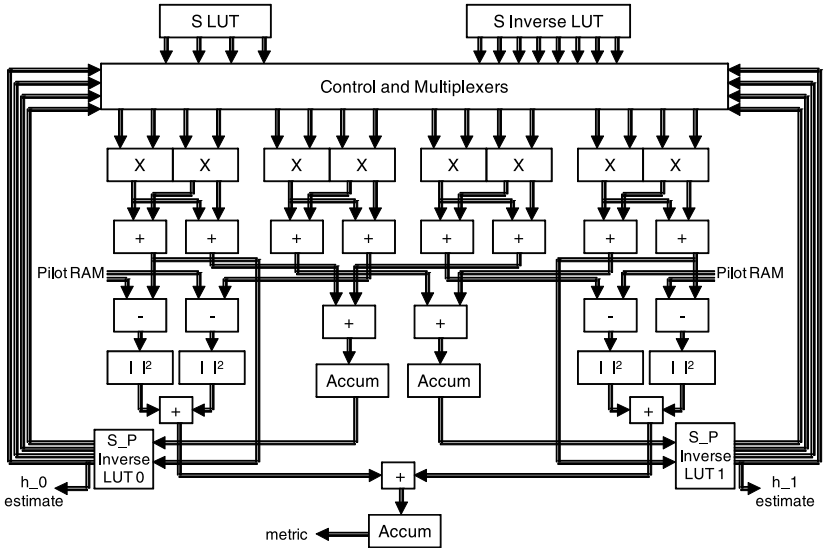


Fig. 8. Datapath for FPGA timing and channel estimator.

simplex approach can find the global minimum after calculating only a small subset (64) of the total number of points.

The computational pipeline in the FPGA implementation consists of 8 parallel complex multiplies followed by 8 complex adds. These are followed by a pipelined series of complex adds, subtracts, accumulators, and magnitude squared operations. The incoming data is 18-bit fixed point with bit growth as needed to account for growth. The datapath is shown in Figure 8. In that figure, double lines represent complex data.

The entire estimator consumes about 7,000 slices on a Xilinx XC2VP50-7 part and runs at 187 MHz. It is able to evaluate about 70 surface points in the requisite $320\mu$ seconds and then passes its results on to other portions of the timing and channel estimator that run in parallel with it. The entire FPGA-based timing and channel estimator is thus capable of performing the needed calculations in the $320\mu$ second frame time.

The sequential nature of the simplex search makes it a poor match for a GPU. Further, the computation of each point on the error surface requires multiple vector data sum reductions, each of which acts as a bottleneck to parallelism. A sequential point-by-point search, similar to that performed by the FPGA, was thus infeasible. An alternate approach tried was to calculate the entire surface in parallel (4096 points), eliminating the serial search. The total calculation time for this exceeds the allowed frame time of $320\mu$s by more than $20\times$. However, the fully convex shape of the error surface in this case allows the use of a two-step search. A coarse sampling of the error surface is first done to identify a tentative minimum point on the surface. A fine $7 \times 7$ search in the neighborhood of this tentative minimum is then used to find the true minimum. Using this strategy the GPU is just able to meet the $320\mu$s frame time requirement.

### 7.3 Pilot Detector

The last comparison made was for a correlator used in the system to locate frame boundary markers in the sampled data stream. In addition to calculations for complex weighting, overlap and add, magnitude, and peak finding, the system also requires seven 1024-point complex valued FFT's be performed every $12\mu$ seconds. The FPGA design consumes the input samples as they are produced (one per clock cycle) and operates in a roughly pipelined fashion at $186MHz$. Its FFT blocks are based on the radix $2^2$ pipelined FFT structure described in He and Torkelson [1996] and Palmer and Nelson [2004] and are easily able to keep up with the 41.6M sample data rate using only four FFT modules, a control unit, and a few scalar multipliers and adders. The FPGA implementation of the pilot detector requires about 16K slices on a Xilinx XC2VP50-7 part.

The FFT's were the limiting factor for the GPU version of this computation. A total of 182 FFT's are required per $320\mu$s frame time. Our tests using the NVIDIA-supplied CUFFT library indicate the GPU computes a single 1024-point complex valued FFT in a little over $60\mu$ seconds on our GPU but that a batch of 32 such FFT's can also be completed in about the same time. The solution was to process many frames in parallel to take advantage of the GPU's parallel resources as with the trellis computation given before. In this case, the application can tolerate the increased latency, whereas for other applications this may not be the case.

## 8. FPGA AND GPU PERFORMANCE

To summarize, the 8800 GTX GPU significantly outperformed the FPGA on both optical flow algorithms, although Sections 3.1 and 6.3 detail FPGA enhancements that could decrease the performance differential. For the MIMO communications system, the FPGA and GPU had similar performance for the three computations compared but the GPU required processing large amounts of data in a block fashion to meet the real time constraints, leading to greatly increased latencies compared to the FPGA pipelined implementations. Both FPGAs and GPUs are capable of high-performance computing, but their relative performance on various applications is not yet well understood. While a full analysis of the ways in which each of these two technologies obtains high performance is beyond the scope of this initial work, some comparisons can be made based on these mixed results.

An early FCCM paper [Graham and Nelson 1996] reported on a detailed performance analysis which attempted to quantify the precise source of speedup for an FPGA-based traveling salesman computation compared to an equivalent workstation implementation. The results of that work seem relevant still today, even though FPGA devices have grown from 100's of LUTs to 10,000's of LUTs and clock rates have increased from 10 MHz to over 300 MHz. In that paper, the three most significant factors identified in the performance of an FPGA design compared to an equivalent C-based workstation computation were: fine-grained parallelism (pipelining), hard-wired control, and coarse-grained parallelism (data parallelism). The relative speedups contributed by

these for the application studied in Graham and Nelson [1996] were 11×, 2.7×, and 1.5×, respectively.

Another key issue for FPGA applications is how they often hide memory latency by statically scheduling and pipelining all memory accesses to keep their computational pipelines full. Finally, total memory bandwidth, as measured by the number of independent memory accesses supported per cycle, has also been shown to be a limiting factor to FPGA performance [Graham and Nelson 1998].

Like the FPGA, the GPU is able to support multiple concurrent memory accesses each cycle. Also like the FPGA, the GPU is able to hide much of its main memory (device memory) latency, but it does so in a very different way. In contrast to the static memory scheduling of many FPGA designs, the GPU uses dynamic thread scheduling for this purpose; when a group of threads issues requests to device memory, the GPU hardware deschedules these threads and swaps in another group of threads. This effectively hides GPU device memory latency, provided there are sufficiently many threads waiting to be scheduled at any given time. For large, regular calculations this can be the case. In the case of the MIMO computations we studied before, finding sufficient parallel threads to hide memory latency was problematic, leading us to attempt to buffer up 10,000's of data samples in memory at a time to provide enough parallel work for the GPU.

Interestingly, the levels of data parallelism achieved on a GPU may be an order of magnitude higher than the data parallelism present in many FPGA implementations. In our experience, pipelining is often the more important type of parallelism in FPGA designs. In Graham and Nelson [1996] pipelining in the FPGA accounted for 10× more speedup than data parallelism did. In the MIMO modules we investigated, our sense is that was the case as well.

There was a 10× to 12× difference in development effort required in this work between the FPGA implementation and the GPU implementation. We attribute this productivity difference principally to the fact that programming a GPU is a software development activity while developing for an FPGA is a hardware design activity. Software debug iterations in this work involved fast edit/compile/execute cycles compared to the much more time-consuming process of edit/compile/simulate followed by edit/synthesize/place-and-route/execute required for hardware development. Performance optimizations were also simpler to implement and test in software than in hardware for the same reason.

## 9. CONCLUSIONS AND FUTURE WORK

In this work, a pair of optical flow algorithms originally developed for implementation in an FPGA were also implemented on a GPU, providing an opportunity for comparing the two technologies for this computation. Additionally, three modules from a MIMO digital communication system were implemented using both technologies and the results discussed. A number of conclusions emerge from our experience. The first is that FPGAs possess unrivaled flexibility for combining custom I/O with computation, and that they can support a

variety of platform structures. The second is that GPUs seem to be more sensitive to compute-to-I/O ratio than FPGAs. The third is that one way to achieve high performance on a GPU for the MIMO computations was to buffer up large amounts of input data to provide sufficient computations to utilize the GPU's resources (turning streamed computations into block computations). In contrast, the FPGA implementations of the MIMO modules easily consumed the input samples as their pipelined computations progressed, leading to lower-latency solutions. Finally, the required development infrastructure, skill level, and time was significantly higher for the FPGA implementations than for the GPU implementations.

A number of research questions pose themselves in the context of these comparisons, as well as with regards to the large number of other new computing devices (CGRA, Cell, multicore) becoming available for use. Our future work will focus on two different questions. The first is a more comprehensive performance comparison between FPGA and GPU technology, specifically focusing on the relative capabilities of the two, the sources of their performance, and the characteristics of the applications spaces for which each is well suited. This work will be extended to include other architectures as well. The second question will focus on how these various technologies might best be combined into new types of field-programmable CCM architectures for both embedded and nonembedded computing.

## REFERENCES


ALAMOUTI, S. 1998. A simple transmit diversity technique for wireless communication. *IEEE J. Selected Areas Comm. 16*, 1451–1458.

ARRIBAS, P. C. AND MACIA, F. M. H. 2001. FPGA implementation of camus correlation optical flow algorithm for real time images. In *Proceedings of the 14th International Conference on Vision Interface*. 32–38.

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.

CHASE, J., NELSON, B., BODILY, J., Z., W., AND D.J., L. 2008. Real-Time optical flow calculations on FPGA and GPU architectures: A comparison study. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'08)*. IEEE Computer Society Press.

COPE, B., CHEUNG, P., LUK, W., AND WITT, S. 2005. Have GPUs made FPGAs redundant in the field of video processing? In *Proceedings of the IEEE International Conference on Field-Programmable Technology*. 111–118.

CORREIA, M. AND CAMPILHO, A. 2002. Real-Time implementation of an optical flow algorithm. In *Proceedings of the IEEE International Conference on Image Processing (ICIP'02)*. Vol. 4. 247–250.

DIAZ, J., ROS, E., PELAYO, F., ORTIGOSA, E. M., AND MOTA, S. 2006. FPGA-Based real-time optical-flow system. *IEEE Trans. Circ. Syst. Video Technol. 16*, 2, 274–279.

DIEPOLD, K., DURKOVIC, M., OBERMEIER, F., AND ZWICK, M. 2006. Performance of optical flow techniques on graphics hardware. In *Proceedings of the International Congress on Mathematical Education (ICME'06)*. 241–244.

FARNEBACK, G. 2000a. Fast and accurate motion estimation using orientation tensors and parametric motion models. In *Proceedings of the International Conference on Pattern Recognition (ICPR'00)*. Vol. 1. 135–139.

FARNEBACK, G. 2000b. Orientation estimation based on weighted projection onto quadratic polynomials. In *Proceedings of the Conference on Vision, Modeling, and Visualization*. 89–96.

FARNEBACK, G. 2001. Very high accuracy velocity estimation using orientation tensors, parametric motion, and simultaneous segmentation of the motion field. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV'01)*. Vol. 1. 77–80.

GRAHAM, P. AND NELSON, B. 1996. Genetic algorithms in software and in hardware—A performance analysis of workstation and custom computing machine implementations. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*. J. Arnold and K. Pocek, Eds. 216–225.

GRAHAM, P. AND NELSON, B. 1998. FPGA-Based sonar processing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. J. Cong and S. Kaptanoglu, Eds. ACM Press, 201–208.

GROB, J. 2003. Linear regression. Lecture Notes in Statistics.

HAUSSECKER, H. AND SPIES, H. 1999. *Handbook of Computer Vision and Application. Vol. 2*. Academic Press, New York.

HE, S. AND TORKELSON, M. 1996. A new approach to pipeline fft processor. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96)*. 766–770.

HOERL, A. AND KENNARD, R. 1970. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics 12*, 1, 55–67.

HOWES, L., PRICE, P., MENCER, O., BECKMANN, O., AND PELL, O. 2006. Comparing FPGAs to graphics accelerators and the playstation 2 using a unified source description. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'06)*. 1–6.

JOHANSSON, B. AND FARNEBACK, G. 2002. A theoretical comparison of different orientation tensors. In *Proceedings of the Symposium on Image Analysis (SSAB'02)*. 69–73.

MARTIN, J. L., ZULOAGA, A., CUADRADO, C., LAZARO, J., AND BIDARTE, U. 2005. Hardware implementation of optical flow constraint equation using FPGAs. *Comput. Vis. Image Understand. 98*, 462–490.

MIZUKAMI, Y. AND TADAMURA, K. 2007. Optical flow computation on compute unified device architecture. In *Proceedings of the 14th International Conference on Image Analysis and Processing (ICIAP'07)*. 179–184.

NIITSUMA, H. AND MARUYAMA, T. 2005. High speed computation of the optical flow. Lecture Notes in Computer Science, vol. 3617. Springer, 287–295.

PALMER, J. AND NELSON, B. 2004. A parallel FFT architecture for FPGAs. In *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL'04)*. 948–953.

STRZODKA, R. AND GARBE, C. 2004. Real-Time motion estimation and visualization on graphics cards. In *Proceedings of the Conference on Visualization (VIS'04)*. IEEE Computer Society, 545–552.

WEI, Z., LEE, D., NELSON, B., AND ARCHIBALD, J. 2008. Real-Time accurate optical flow sensor. In *Proceedings of the International Conference on Pattern Recognition (ICPR'08)*.

WEI, Z., LEE, D. J., NELSON, B., AND MARTINEAU, M. 2007. A fast and accurate tensor-based optical flow algorithm implemented in FPGA. In *Proceedings of the IEEE Workshop on Application of Computer Vision (WACV'07)*. 18.

ZACH, C., POCK, T., AND BISCHOF, H. 2007. A duality based approach for realtime TV-L1 optical flow. In *Proceedings of the DAGM Symposium on Pattern Recognition*. 214–223.

ZULOAGA, A., MARTIN, J. L., AND EZQUERRA, J. 1998. Hardware architecture for optical flow estimation in real time. In *Proceedings of the IEEE International Conference on Image Processing (ICIP'98)*. Vol. 3. 972–976.

Received July 2008; revised November 2008; accepted April 2009