Implementing High-Performance, Low-Power FPGA-Based Optical Flow Accelerators in C

Josh Monson, Mike Wirthlin, Brad L Hutchings

NSF Center for High-Performance Reconfigurable Computing (CHREC)

Department of Electrical and Computer Engineering
Brigham Young University, Provo, Utah

jsmonson@gmail.com, wirthlin@ee.byu.edu, hutch@ee.byu.edu

***Abstract*—Recent developments in High-Level Synthesis (HLS) for FPGAs are making it possible to "run" C code on FPGAs thereby making modern programming environments available to FPGA developers. In this paper, C code for a complex optical-flow algorithm is optimized for both a desktop PC and for an FPGA-based system, the Xilinx Zynq-7000, a device containing both a programmable fabric and two ARM cores. The paper discusses how the code is optimized and restructured to execute effectively on the programmable fabric and the ARM cores. The resulting Zynq version of the C code is competitive with the desktop PC but only consumes 1/7th as much energy.**

***Keywords*-FPGA, configurable computing, ARM**

I. Introduction

Relatively recently, High-Level Synthesis (HLS) tools such as Vivado HLS (commercial) [1] and LegUp (academic) [2] have made it possible to program FPGAs using C code instead of VHDL/Verilog. HLS tools accept C syntax and generate a file format (typically VHDL/Verilog) that can be processed by the FPGA vendor software. HLS holds out the following promise: the ability to program in C while simultaneously achieving high performance and consuming low power.

This paper compares several highly-optimized versions of a complex optical-flow algorithm: one version optimized to run on a desktop microprocessor, one version operating on an embedded Arm processor, and one version optimized to run on an FPGA. The C language is used exclusively for all implementations. Vivado HLS is used to process the C code for the FPGA implementation. This paper demonstrates that it is feasible to create an FPGA accelerator for a complex algorithm using only C that: 1) achieves performance comparable to the optimized version running on a high-performance desktop machine, and 2) consumes approximately 7.4X less energy.

II. Related Work

There has been strong interest in creating hardware-based accelerators for optical flow within an FPGA [3]. FPGAs can provide significant improvements in both overall performance and power efficiency by customizing the datapath of the algorithm and optimizing memory accesses. There are many examples of FPGAs implementing an optical flow algorithm for improved performance or for a real-time, low-power implementation [4], [5], [6].

To improve the design productivity of implementing FPGA-based optical flow systems, several researchers have demonstrated optical flow implementations using high-level synthesis tools. In [7], the author develops a high performance circuit implementing the "Lucas" optical flow algorithm using the Mentor Graphics Catapult C high-level synthesis tool. This work identifies a number of tips for improving the quality of the synthesized circuit including a number of transforms, including memory access management, for exploring the design space and improving circuit throughput [8]. The Synfora PICO Extreme HLS tool was used to demonstrate the productivity of HLS for two machine vision algorithms including optical flow [9]. This work implemented a gradient optical flow algorithm based on Farneback's technique and demonstrate that HLS can provide a result close to the quality of RTL, hand-designed circuit. The Handel-C design language was used to implement a novel, bio-inspired optical flow algorithm on an FPGA [10] and a real-time optical flow vision system was developed using an FPGA with the optical flow algorithm programmed using the Celoxica [11] programming language.

This work differs from previous related work in two ways. First, this work implements *multiple* optical flow accelerators from a C specification using various HLS directives and code transformations. Second, this work targets a unique programmable SOC (Zynq) that includes both an embedded processor and a programmable logic fabric.

III. LK-Optical Flow summary

The goal of optical flow is to create a flow field that estimates the displacement of interesting features between successive frames. The optical flow algorithm used in this paper is based on the OpenCV function `calcOpticalFlowPyrLK` [12]. This function implements the sparse iterative optical flow algorithm described by

Bouguet [13]. The OpenCV implementation of the Bouguet algorithm (shown in Fig. 1) is accomplished using the Lucas and Kanade [14] method on multiple levels of an image pyramid. The optical flow algorithm, including the OpenCV implementation, is computationally demanding and requires significant computational resources to operate in real time. The OpenCV implementation has three major computational steps: Pyramid Creation, Calculate Gradients, and Feature Tracking.

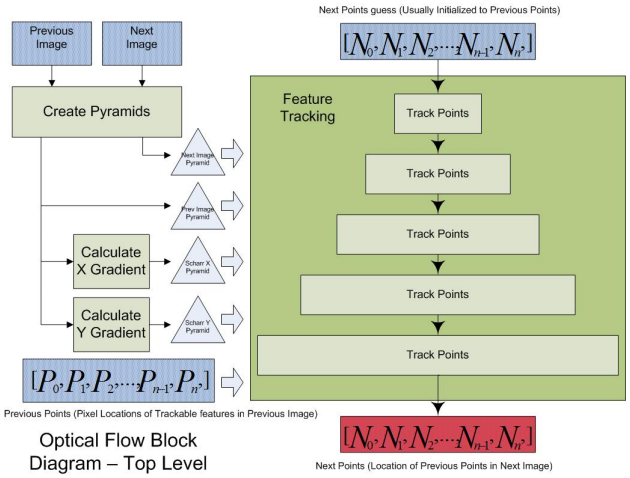

Figure 1. A Block Diagram showing the different steps of the LK-Optical Flow Algorithm.

A. Pyramid Creation

The Optical Flow algorithm takes two images as inputs. These are referred to as the "previous" and "next" images. The first step in calculating optical flow is to create image pyramids of both the Previous and Next images. Image pyramids are sets of images where the first image is the original image and successive images are down-sampled versions of the first. The first image or one at the "bottom" of the pyramid is the largest (highest resolution) and the image at the "top" of the pyramid is the smallest (lowest resolution). Starting with the original image as level L=0, level L+1 is created by applying a gaussian image filter to level L and removing the odd rows and columns from the filtered image. The gaussian filter is applied before down-sampling to prevent the aliasing of the high frequency components of the image.

After filtering and down-sampling, each level of the pyramid is set within a "reflected" border. Borders are used in image processing to handle cases where image data is needed from outside the image. The reflected border used in this algorithm does not repeat the edge pixel. For example, if pixel A is on the left edge of the image, followed by B, C, D, and E to the right, the reflected border from left to right would be E, D, C, and then B would be right next to the original A pixel on the edge of the image.

Bouguet proposed to use image pyramids to mitigate the trade-off between the accuracy and robustness of the algorithms. Small feature windows favor accuracy but are more sensitive to camera motion and changes in lighting. Large windows provide better tolerance to motion and changes in lighting but are less accurate. The use of image pyramids, therefore, allows the search to begin at a coarse resolution (providing a large feature window) and conclude at a fine resolution (providing a small window).

B. Gradient Calculation

The second step in the calculation of optical flow is to compute the gradient of each level of the previous image pyramid. The image gradient is an important piece of data used to estimate the displacement. The resulting image gradients (or derivatives) are set within a border like the image pyramids were. However, in this case constant borders are used rather than a reflected one.

The image gradient is computed for a given pixel by multiplying all pixels in a given neighborhood by an image kernel and then summing the results. The gradient in both the horizontal (x) and vertical (y) directions are required. Bouguet recommends that the Scharr kernels be used to calculate the gradient. As shown in Fig. 1, the gradient and the image pyramids are passed as input to the feature tracking phase.

C. Feature Tracking

Feature tracking is the heart of the optical flow algorithm. This is the phase where feature displacement is estimated. Features are represented by small neighborhoods of pixels known as windows. A window is represented by the coordinate of the pixel in the upper left corner.

Fig. 2 is a block diagram of the feature tracking process. First the windows are created from the previous, next, and derivative windows located at the pixel coordinate representing the feature. Each of these windows are interpolated. The interpolated pixels of the derivative window are sent to a multiply-accumulate block where 3 sums are computed. The first and third sums are the squares of the x and y components of the derivative. The second sum is the sum of the product of the x and y components. A pixel-by-pixel difference of the interpolated previous and next windows is taken. These results are sent to a multiply accumulate block where 2 sums are computed. The first sum is the pixel-by-pixel difference multiplied by the x-component of the derivative. The second sum is the pixel-by-pixel difference multiplied by the y-component of the derivative.

These five summations are then used to calculate the displacement estimate. If the magnitude of the displacement is small (below a specified threshold) the feature is assumed to have been found and the computation is complete. However, if the magnitude of the displacement is large another iteration is required. The second iteration starts by shifting

the next window by the value of the displacement and repeating the process. The results from the interpolation and summation of the previous and derivative windows can be reused in the successive iteration. However, since the next window has been shifted, interpolation and summation must be re-calculated. The resulting displacement estimate is used as input to the feature tracking computation on lower levels of the pyramid (as shown in Fig. 1). The displacement estimate is refined as the results travel down the pyramid.

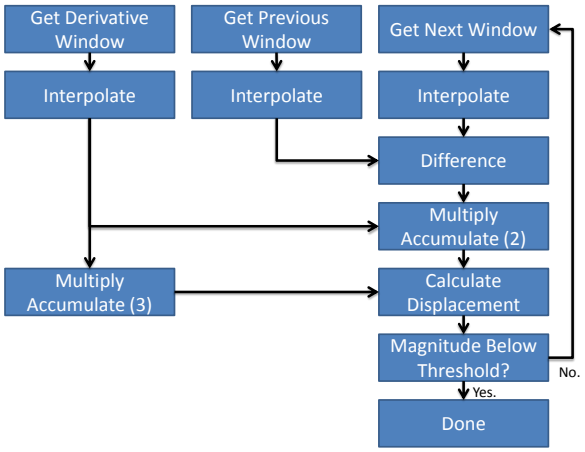

Figure 2. A Block Diagram of the feature tracking computation.

IV. Processor Implementations of Optical Flow

The baseline source code for the optical flow algorithm described above was obtained from the `calcOpticalFlowPyrLK` function of the OpenCV library. This function was written in C and can be compiled to a number of different processor architectures. For this paper, this function was compiled and run on two different processor architectures: an Intel Core i7 processor and the ARM Cortex A9 embedded processor. Each processor offers a different trade-off between performance, power, and system complexity.

A. Core i7 Processor

The i7 implementation was mapped to a Intel Core i7 860 2.8 GHz desktop running the Windows 7 operating system. The Microsoft Visual Studio 2010 compiler was used to generate the the i7 executable. The Intel Thread Building Blocks library was enabled to maximize performance by spreading the feature tracking operation across all four i7 cores. The i7 implementation used hand-optimized SIMD instructions to accelerate the application.

Using these optimizations, the Core i7 was able to achieve a performance of 80 frames per second (FPS) on 720x480 images, using 15x15 integration windows, and tracking between 350-400 features. This frame rate allows the optical flow algorithm to process images in real time. The relative execution time of the three major phases of the algorithm are summarized in Table I. As expected, the feature tracking portion of this algorithm requires the most execution time (48.0%).

Although this implementation will run in real-time, it requires a relatively large amount of power and a complex system infrastructure. While we do not have power measurements of the processor running the application, a review [15] reported that the core i7 consumed 124 Watts while running a full load on all four of its cores. With each frame completed in 12.5 ms, the computation for each frame requires 1.6 J of energy.

B. ARM Cortex A9

The OpenCV optical flow algorithm described above was also ported to the ARM Cortex A9 embedded processor. The ARM A9 implementation was mapped to one of the embedded Cortex A9 cores within the Xilinx Zynq 7020 Extensible Processing Platform (EPP). The code was compiled using the Xilinx version of the Code Sourcery compiler and verified on the Xilinx "ZedBoard" development board. This single core implementation does not utilize the NEON SIMD instructions or multi-threading.

Using the same algorithm parameters as the i7 implementation (i.e., 720x480 images, etc.), the ARM implementation achieves a performance of 10.1 frames per second. As shown in Table I, the relative execution times of each phase are similar to those of the core i7.

As an embedded, low-power processor, the ARM implementation consumes far less power than the core i7. Using actual measurements on the computing platform, the average power consumed by the ZedBoard was measured at 6.5 Watts. This is 19× less power than the core i7. Although this implementation is 8× slower than the core i7, it consumes 2.5× less energy to perform the computation as the core i7.

Table I
Summary of Implementation and Execution Results.

	ARM	%	core i7	%
pyramid	14.3 ms	14.4%	2.8 ms	22.4%
scharr	17.4 ms	17.6%	3.7 ms	29.6%
tracker	66.7 ms	67.3%	6.0 ms	48.0%
other	.7 ms	.7%	0.0 ms	0.0%
Total	99.1 ms	100%	12.5 ms	100%
FPS	10.1		80.0	
Power	6.5W		124W	
Energy	.644J		1.6J	

C. Optical Flow Acceleration with FPGA Resources

As discussed earlier, application circuits implemented within FPGA resources have been used to accelerate the Optical Flow algorithm and reduce the energy required to perform the computation. Our goal is to achieve real-time

performance of the algorithm within the low-power ARM platform by exploiting the integrated FPGA resources of the Zynq platform. This work seeks to identify portions of the optical flow algorithm that can be moved from the ARM processor and onto the FPGA resources such that the combined hardware and software implement the algorithm in real time.

To achieve the real-time performance of the core i7, the ARM implementation must be sped up by a factor of 3×. According to Amdahl's law, achieving a 3× speed-up requires that at least 66.7% of the runtime of the sequential implementation of the algorithm needs to be "infinitely" accelerated. As shown in Table I, the runtime of the algorithm is split between the three algorithm parts – *all* parts of algorithm will be accelerated to achieve the 3× speedup.

Fortunately, each of the three algorithm parts are amenable to FPGA acceleration. In particular, image filtering is used in pyramid creation and in the Scharr derivative. Image filtering accounts for 79.2% of pyramid creation and nearly 100% of the Scharr derivative. We estimate that by implementing image filtering in an FPGA, the runtime of the pyramid can be improved from 14.3 ms to 3.7 ms (3.9×). Additionally, we estimate that the runtime of the Scharr derivative can be improved from 17.4 ms to 3.1 ms (5.6×). These improvements, however, are not sufficient for achieving real-time performance and additional improvements must come by accelerating the feature tracker.

Like the pyramid creation and Scharr derivative, the feature tracking portion of optical flow is amenable to acceleration with custom hardware. However, estimating the performance improvements of an accelerator for the feature-tracking portion is difficult because the algorithm is irregular and alternates between sequential and parallel code. High-level synthesis tools will be used to rapidly explore various strategies for accelerating the feature tracking portion of the algorithm. The C code used to define these functions can be quickly synthesized into hardware and evaluated in terms of hardware cost and potential performance improvement.

V. Zynq Architecture

The target architecture for the optical flow algorithm is the Xilinx ZYNQ-7000 Embedded Processor Platform. The Zynq-7000 is a programmable device, fabricated at 28 nm that includes both a programmable fabric and a dual-core ARM processor. The ARM core runs up to 1 GHz and includes level-1 and level-2 caches, supports single and double-precision floating point, and provides a variety of peripherals: DMA and memory controllers, 256 KB of dual-ported on-chip RAM, along with an interrupt controller, various timers, etc. Several other fixed-function peripherals are also included: two Gigabit Ethernet ports, two USB ports, two SPI ports, UARTs, etc. These fixed-functions are connected to a conventional programmable fabric (FPGA). The fabric contains 53,200 LUTs, 106,400 flip-flops, 220 DSPs and 280 block RAM (BRAM). Communication between the fabric and the ARM core is achieved via a bus-oriented master-slave interface using the AMBA AXI bus standard.

As shown in Fig. 3, Zynq provides fully-shared access to the DDR between the ARM core and the programmable fabric via a hardened DDR interface that provides one dedicated 64-bit port per ARM core (via the L2) and four dedicated 64-bit high performance interfaces to the programmable fabric. Shared, high-performance access to DDR from both the ARM core and the programmable fabric is an extremely important feature as it makes it feasible to build accelerators that use both the ARM core and the programmable fabric for computation. The application described in this paper makes extensive use of the high-performance DDR interface.

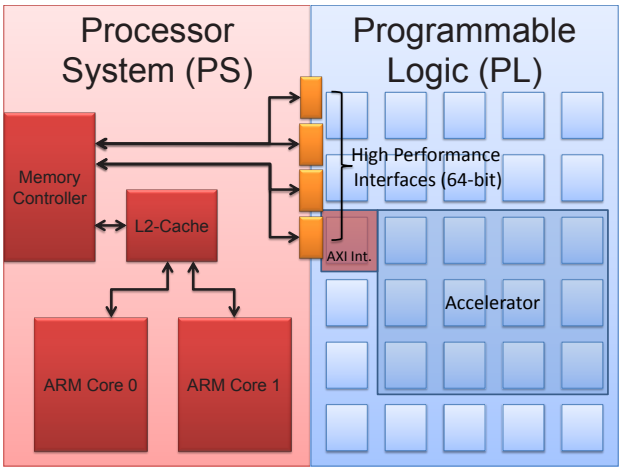

Figure 3. A Block Diagram of the ZYNQ Architecture.

VI. Optical Flow Accelerator using High-Level Synthesis

A variety of optical flow accelerators were created using the C-language and synthesized using the Vivado HDL synthesis tool. The use of the C-language allowed us to quickly explore a variety of accelerators and identify an accelerator that provides real-time processing on the embedded Zynq platform. Each of these accelerators was targeted to the Zynq-7020 SoC FPGA and downloaded onto the Xilinx ZedBoard reference board. Table II reports the following for each accelerator: the size of the accelerator (in terms of hardware resources), the execution time of the accelerator, and the average power of the accelerator.

The resource results in Table II were obtained from the FPGA vendor synthesis reports and include all surrounding hardware (interconnect, measurement hardware, etc.). The runtime measurements of the accelerator were taken by actual measurements in the FPGA. The number of clock cycles between the time the accelerator starts and the time it finishes as it executed in the FPGA was measured with

a dedicated hardware monitor. The throughput of the accelerator, measured in frames per second (FPS), is estimated by taking the reciprocal of the execution time. The power consumption of the accelerator was obtained by measuring the voltage across a shunt resister (and thus the current) for the entire system, e.g., all components on the ZedBoard. The reported power values are averages over the entire execution of the accelerator. The energy consumed by the accelerator is estimated by multiplying the average power by the execution time.

A. Baseline

The first accelerator is the "baseline" accelerator and it was created by taking the original C code specification and modifying it so that it is synthesizable by the Vivado HLS tool. Before any C code can be synthesized into hardware using the Vivado HLS tool, the algorithm must be expressed using a synthesizable subset of the C language. For the Vivado HLS tool, this means that all memory must be statically allocated, the code must be free of recursive functions, and pointers and references are not used as class members. The original OpenCV C code of the feature tracker was modified to remove non-synthesizable code and to include support for burst memory transactions. The `memcpy` function was used to replace indirect image memory accesses. This allows the use of burst transfers and avoids large overhead penalties associated with handshaking for memory transfers. A bus bridge was synthesized that performs aligned memory transfers. Code was added to perform proper modulo pointer arithmetic on aligned memory transfers to retrieve unaligned data.

The baseline accelerator computed a single frame in 54.4 ms and can operate at 18.4 frames per second. Without any architectural modifications or optimizations, this baseline accelerator provides an improvement over the ARM software-only version (1.2×). This implementation is a starting point and suggests that additional speedup can be obtained by implementing well-known code transformations or synthesis directives. This baseline design uses less than half the available FPGA resources suggesting that there are sufficient resources to implement other optimization techniques.

B. Dataflow Directive

The processes in Fig. 2, can be split into two groups. Group 1 is the first two columns of processes (derivative and previous window columns) and Group 2 is the third column (next column). In the baseline accelerator design, these two groups of processes are executed sequentially as shown in the schedule in Fig. 4. However, there is nothing to prevent these two groups from operating in parallel. In Vivado HLS, the dataflow directive can be used to specify that two (or more) sequential groups of computation be implemented to execute in parallel. When the dataflow directive is used, Vivado HLS creates independent processes and connects the first to the second using FIFOs or ping-pong buffers. This allows Group 1 to start on the second feature while Group 2 finishes the first (see Fig. 5).

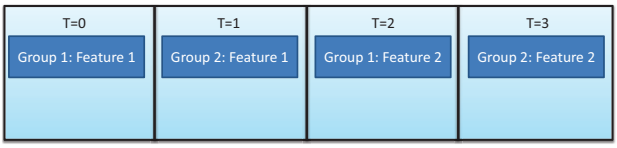

Figure 4. Schedule showing at a high-level how the baseline design is scheduled.

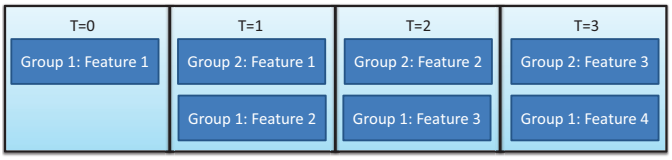

Figure 5. Schedule showing how the dataflow allowed the groups to be scheduled in parallel.

To use the dataflow directive, the main loop of baseline code had to be restructured to split the code for Group 1 & Group 2. This code transformation and code refactoring resulted in an accelerator that required significantly more resources (see Table II). The increase in DSPs, LUTs, and FFs may largely be due to the fact that the code structure of the primary data paths were modified to perform more operations in parallel. FIFOs added as channels between the Group 1 & Group 2 processes resulted in the large increase in BRAMs. This increase in hardware, however, provides more than 2× improvement in throughput over the baseline accelerator.

C. Integrating the Scharr Derivative into the Feature Tracker

Another way of accelerating the application is by reducing the computational load and the required bandwidth. This optimization does both by integrating the Scharr derivative into the feature tracking accelerator as shown in Fig. 6. This can be done because the feature tracking accelerator is the only consumer of the Scharr derivative computation. The computational load is reduced because the Scharr derivative is only taken on those parts of the previous pyramid that are used by the accelerator. The required memory bandwidth is reduced because 1) the Scharr derivative is no longer a separate step, this removes a read of the entire previous pyramid and a write of the entire derivative pyramid (which is 4x larger than the previous pyramid), and 2) it means that the accelerator no longer has to read derivative integration windows from memory. The fact that a constant border is used (rather than a reflected border) helps the data flow of the function. This allows the Scharr derivative function to output a constant value if the integration window overlaps with the

Table II
SUMMARY OF IMPLEMENTATION AND EXECUTION RESULTS.

Version	LUTS	FF	DSPs	BRAMs	Latency (cycles)	Clock Rate	Time	FPS	Avg. Power
baseline	22,647 (42.6%)	16,171(15.2%)	110(50.0%)	23 (8.2%)	5,171,563	95 MHz	54.4 ms	18.4	6.01 W
dataflow	34,007(63.9%)	32,844(30.9%)	196(89.1%)	201 (71.7%)	2,590,972	100 MHz	25.9 ms	38.6	6.86 W
scharr	37,266 (70.0%)	33,601 (31.6%)	194 (88.2%)	185(66.1%)	2,518,141	100 MHz	25.2 ms	39.7	6.92 W
max. clock	37,055 (69.7%)	33,626 (31.6%)	194 (88.2%)	185 (66.1%)	2,538,290	110 MHz	23.1 ms	41.8	7.08 W

border. Coding the Scharr derivative in C that accounted for this border issue was simple.

This optimization actually increased the performance of the feature tracker accelerator. This is due to the combination of not having to read the derivative integration window and improvements in read latency. We have now implemented the Scharr derivative function for only the cost of a few resources (3,189 LUTs and 757 flip flops). Additionally, we estimate that including the Scharr derivative into the feature tracker reduced bandwidth requirements for the feature tracker by 57.5% and reduced bandwidth requirements for the entire system by 64.3%.

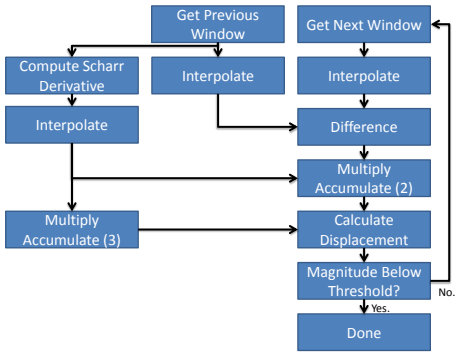

Figure 6. A Block Diagram showing the addition of the Scharr derivative.

D. Increasing the Clock Rate

Our final optimization is to increase the clock rate of the feature tracking accelerator. The high performance DDR interfaces used for the feature tracker can operate at clock rates up to 150 MHz. The dataflow and Scharr optimizations were clocked at 100 MHz; however, this was not necessarily their maximum clock rate. By experimenting with clock constraints in EDK we found that the feature tracker with the Scharr optimization could operate at speeds up to 110 MHz. The performance of this accelerator is shown in Table II under the name max. clock.

VII. PERFORMANCE ANALYSIS

Table III shows the estimated performance and average energy usage of the application under each version of the feature tracking accelerator. These runtimes were calculated by summing the estimated runtimes of FPGA accelerated versions of pyramid, Scharr, and feature tracking (i.e., $T_{est} = T_{pyramid} + T_{sharr} + T_{ft}$). For example, the runtime estimate of the application using the baseline feature tracking accelerator is 3.7 ms + 3.1 ms + 54.4 ms = 61.2 ms (1/.0612 S = 16.3 FPS). The runtimes for pyramid (3.7 ms) and Scharr (3.1 ms) were estimated in Section IV-C and the runtime for the baseline (54.4 ms) is found in Table II. Since the "Scharr" and "max clock" accelerators include the Scharr derivative computation within the accelerator circuit, the estimated $T_{scharr} = 3.1ms$ is not used within the runtime estimation of these accelerators. Energy usage was estimated using the average power consumption of the feature tracking accelerator from Table II. This seems to provide a reasonable estimate given that most of the application is now executed in the reconfigurable fabric.

Table III
PERFORMANCE OF OPTICAL FLOW

Accelerator	Runtime	FPS	Avg. Energy
baseline	61.2 ms	16.3	.369 J
dataflow	32.7 ms	30.6	.224 J
scharr	28.9 ms	34.6	.200 J
max. clock	26.8 ms	37.3	.190 J

Comparing the hardware accelerated optical flow and the original software implementations must be done carefully. The original software versions were written to create an image pyramid for both of the input images each time the function was called. On the other hand, the hardware version reuses previous pyramid creation effort and only needs to create 1 new pyramid over the same duration. In other words, the software implementations are doing more work. This can be accounted for by subtracting half of the pyramid creation (reported in Table I) from the total runtimes reported (also reported in Table I). After accounting for pyramid creation time, the runtime of the ARM implementation is estimated as 92.0 ms (10.9 FPS). The reduction in execution time reduces the energy consumption to .598 J (from .644 J). The runtime of the core i7 improves from 12.5 ms (80 FPS) to 11.1 ms (89.7 FPS) while estimated energy is reduced from 1.6 J to 1.4 J. The hardware version out-performed the ARM implementation by 3.6× while consuming 3.2× less energy. We also see that the hardware version was competitive with the Core i7 while consuming 7.4× less energy.

VIII. Conclusion

HLS synthesis was used to develop an implementation of the optical flow algorithm in C that has performance competitive with desktop processor and operates at 1/7th the energy. HLS significantly reduced the amount of effort and time to develop this low-power implementation.

- *You start immediately with the original C source code.* It was not necessary to convert the original C description into RTL as is typically done for FPGA implementations. The original C source code was easily modified so that it was compatible with the HLS software.
- *You can rapidly explore the design space to achieve sufficient performance and low-power operation.* Different versions of the algorithm were quickly created and analyzed to determine their impact on performance and energy consumption.
- *You can quickly compare and verify different versions of the algorithm in C.* It is much easier to compare different versions of the algorithm in C and determine execution speeds and estimate power consumption. Verification is also simpler because it is much easier to co-execute and compare different versions of the software in a C development environment than is possible with RTL simulation.

Some may argue that further energy reductions may be possible if the accelerator were implemented in RTL, because it *may* be possible to reduce energy consumption and increase clock rate with RTL. Indeed, it may be possible to achieve more efficient implementations in RTL. However, it is unlikely, given the much longer development cycle of RTL, that the developers would ever arrive at the same optimizations discovered while rapidly exploring the design space in C. As is often the case in real situations, it is usually more important to quickly arrive at an implementation that meets the requirements than it is to try to discover the most efficient implementation of an application.

Acknowledgment

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

References

[1] Xilinx, *Vivado Design Suite User Guide: High-Level Synthesis (UG902 (v2012.4))*, PDF File, Xilinx, San Jose, California, December 2012.

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 33–36. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950423

[3] J. Bodily, B. Nelson, Z. Wei, D.-J. Lee, and J. Chase, "A comparison study on implementing optical flow and digital communications on FPGAs and GPUs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 2, pp. 6:1–6:22, May 2010. [Online]. Available: http://doi.acm.org/10.1145/1754386.1754387

[4] N. Roudel, F. Berry, J. Serot, L. Eck, and C. List, "Hardware implementation of a real time Lucas and Kanade optical flow," in *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2009.

[5] A. Browne, T. McGinnity, G. Prasad, and J. Condell, "FPGA based high accuracy optical flow algorithm," in *Signals and Systems Conference (ISSC 2010), IET Irish*, june 2010, pp. 112 –117.

[6] D. Honegger, P. Greisen, L. Meier, P. Tanskanen, and M. Pollefeys, "Real-time velocity estimation based on optical flow and disparity matching," *To appear, Intelligent Robots and Systems (IROS)*, 2012.

[7] X. Ren, "RTL Implementation of an Optical Flow Algorithm (Lucas) Using the Catapult C High-Level Synthesis tool," Master's thesis, Delft University of Technology, 2011.

[8] C. Feenstra, "A Memory Access and Operator Usage Profiler Framework for HLS Optimization," Master's thesis, Delft University of Technology, 2011.

[9] K. Denolf, S. Neuendorffer, and K. Vissers, "Using C-to-gates to program streaming image processing kernels efficiently on FPGAs," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 31 2009-sept. 2 2009, pp. 626 –630.

[10] G. Botella, A. Garcia, M. Rodriguez-Alvarez, E. Ros, U. Meyer-Baese, and M. Molina, "Robust bioinspired architecture for optical-flow computation," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, no. 4, pp. 616 –629, april 2010.

[11] J. Diaz, E. Ros, F. Pelayo, E. Ortigosa, and S. Mota, "FPGA-based real-time optical-flow system," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 16, no. 2, pp. 274 – 279, feb. 2006.

[12] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[13] J. Bouguet, "Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm," *Intel Corporation*, 2001.

[14] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," 1981, pp. 674–679.

[15] D. Graham-Smith. (2009, September) Intel core i7-860 review. [Online]. Available: http://www.pcpro.co.uk/reviews/processors/351388/intel-core-i7-860