Compiling High-Level Image Processing using FPGS

BommarahyNarsing
Asst Professor
Department of ECE
Scient institute of engg and technology,Hyderabad,Telangana ,India

Abstract

Specialized image signal processors (ISPs) exploit the structure of image processing pipelines to minimize memory bandwidth using the architectural pattern of line-buffering, where all intermediate data between each stage is stored in small on-chip buffers. This provides high energy efficiency, allowing long pipelines with tera-op/sec. image processing in battery-powered devices, but traditionally requires painstaking manual design in hardware. Based on this pattern, we present Darkroom, a language and compiler for image processing. The semantics of the Darkroom language allow it to compile programs directly into line-buffered pipelines, with all intermediate values in local line-buffer storage, eliminating unnecessary communication with off-chip DRAM. We formulate the problem of optimally scheduling line-buffered pipelines to minimize buffering as an integer linear program. Finally, given an optimally scheduled pipeline, Darkroom synthesizes hardware descriptions for ASIC or FPGA, or fast CPU code. We evaluate Darkroom implementations of a range of applications, including a camera pipeline, low-level feature detection algorithms, and deblurring. For many applications, we demonstrate gigapixel/sec. performance in under 0.5mm2 of ASIC silicon at 250 mW (simulated on a 45nm foundry process), realtime 1080p/60 video processing using a fraction of the resources of a modern FPGA, and tens of megapixels/sec. of throughput on a quad-core x86 processor.

Introduction

The proliferation of cameras presents enormous opportunities for computational photography and computer vision. Researchers are developing ways to acquire better images, including high dynamic range imaging, motion deblurring, and burst-mode photography. Others are investigating new applications beyond photography. For example, augmented reality requires vision algorithms like optical flow for tracking, and stereo correspondence for depth extraction. However, real applications often require real-time throughput and are limited by energy efficiency and battery life. To process a single 16 megapixel sensor image, our implementation of the camera pipeline requires approximately 16 billion operations. In modern hardware, energy is dominated by storing and loading intermediate values in off-chip DRAM, which uses over 1,000⇥ more energy than performing an arithmetic operation [Hameed et al. 2010]. Simply sending data from mobile devices to servers for processing is not a solution, since wireless transmission uses 1,000,000⇥ more energy than a local arithmetic operation. Often the only option to implement these algorithms with the required performance and efficiency is to build specialized hardware. Image processing on smartphones is performed by hardware image signal processors (ISPs), implemented as deeply pipelined custom ASIC blocks. Intermediate values in the pipeline are fed directlybetween stages. This pattern is often called line-buffering. The combination of many arithmetic operations with low memory bandwidth leads to a power efficient design. Performing image processing in specialized hardware is at least 500⇥ more power efficient than performing the same calculations on a CPU [Hameed et al. 2010].

However, implementing new image processing algorithms in hardware is extremely challenging and expensive. In traditional hardware design languages, optimized designs must be expressed at an extremely low level, and are dramatically more complex than equivalent software. Worse, iterative development is hamstrung by slow synthesis tools: compile times of hours to days are common. Because of this complexity, designing specialized hardware, or even programming FPGAs, is out of reach to most developers. In practice, most new algorithms are only implemented on general purpose CPUs or GPUs, where they consume too much energy and deliver too little performance for real-time mobile applications.

In this paper, we present a new image processing language, Darkroom, that can be compiled into ISP-like hardware designs. Similar to Halide and other languages [Ragan-Kelley et al. 2012], Darkroom specifies image processing algorithms as functional DAGs of local image operations. However, while Halide's flexible programming model targets general-purpose CPUs and GPUs, in order to efficiently target FPGAs and ASICs, Darkroom restricts image operations to static, fixed size windows, or stencils. As we will show, this allows Darkroom to automatically schedule programs written in a clean, functional form into line-buffered pipelines using minimal buffering, and to compile into efficient ASIC and FPGA implementations and CPU code

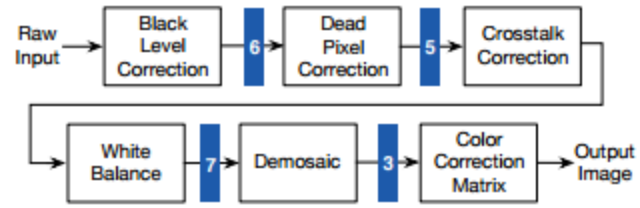

Figure 2: *Camera ISPs are pipelines of many image operations. Some stages access a single point, while others access a window around the output pixel, or* stencil *(indicated by blue boxes). No stages access arbitrary pixels from the image.*

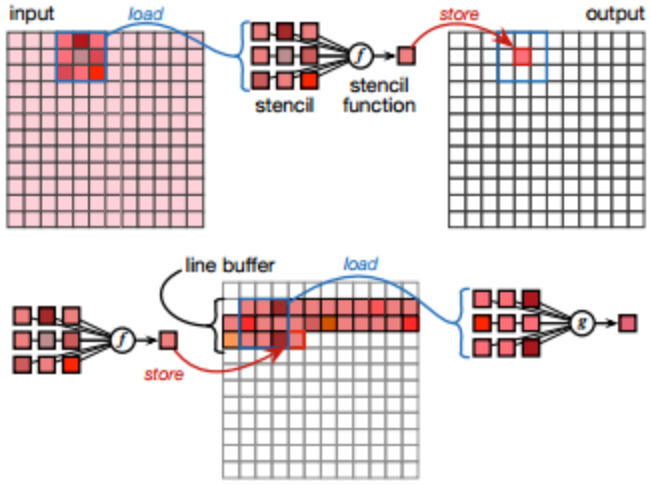

Figure 01:Each stage of the ISP is a pure function of its stencil input and (x, y) position (top). Stencil stages communicate through a line buffer (bottom), a local SRAM that holds a sliding window of the data produced by the previous stage. The required buffer size is determined by the size of the stencil consuming it. Here, it must be large enough to store all values needed for the next iteration of g.

In this paper, we present a new image processing language, Darkroom, that can be compiled into ISP-like hardware designs. Similar to Halide and other languages [Ragan-Kelley et al. 2012], Darkroom specifies image processing algorithms as functional DAGs of local image operations. However, while Halide's flexible programming model targets general-purpose CPUs and GPUs, in order to efficiently target FPGAs and ASICs, Darkroom restricts image operations to static, fixed size windows, or stencils.

As we will show, this allows Darkroom to automatically schedule programs written in a clean, functional form into line-buffered pipelines using minimal buffering, and to compile into efficient ASIC and FPGA implementations and CPU code.

This paper makes the following contributions:

• We demonstrate the feasibility of compiling high-level image processing code directly into efficient hardware designs.

• We formalize the optimization of ISP-like line-buffered pipelines to minimize buffer size as an integer linear program. It computes optimal buffering for real pipelines in < 1 sec.

• We demonstrate back-ends that automatically compile linebuffered pipelines into structural Verilog for ASICs and FPGAs. For our camera pipeline and most tested applications, the generated ASICs are extremely energy efficient, requiring < 250 pJ/pixel (simulated on a 45nm foundry process), while a mid-range FPGA runs them at 125-145 megapixels/sec. On our applications, the generated ASICs and FPGA designs use less than 3⇥ the optimal buffering.

• We also show how to compile efficiently to CPUs. Our results are competitive with optimized Halide, but require seconds to schedule instead of hours. We also show performance 7⇥ faster than a clean C implementation of similar complexity.

Background

Camera ISPs process raw data from a camera sensor to produce appealing images for display. Most camera sensors record only one color per pixel, requiring other channels at each pixel to be estimated from its neighbors ("demosaicing"). ISPs also correct noise, optical aberrations, white balance, and perform other enhancements (Fig. 2). Camera ISPs are typically implemented as fixed-function ASIC pipelines. Each clock cycle, the pipeline reads one pixel of input from the sensor or memory, and produces one pixel of output. ISPs are extremely deep pipelines: there are many stages, and a long delay between when a pixel enters the pipeline and when it leaves. ISP pipelines contain two types of operation. One type only operates on single pixels, such as gamma correction. We call these pointwise,because they only operate on a single point at a time. The second type needs a window of input to produce a single output pixel. For example, demosaicing needs multiple adjacent pixels to interpolate color information. We call these operations stencils, because they match the well-understood structure of stencil computations

Pointwise operations do not require buffering, because the input required (a single pixel) is exactly what is produced by the prior stage. In contrast, stencil operations require multiple pixels of input from the previous stage. For example, a stencil operation could require the pixel at (x, y 1) when producing the output for (x, y). However, the input stage would have produced (x, y 1) W clock cycles ago, where W is the width of the sensor. To provide this data, an ISP would buffer W pixels of the input stage's results on-chip. Using standard terminology from ISP design, we call this buffer a line-buffer, because it buffers lines of the input image (Fig. 1).

Each operation in the ISP is a pure function of its input stencil and the (x, y) position of the pixel it is computing. This means that they can be pipelined and parallelized to an arbitrary performance target using standard techniques [Leiserson and Saxe 1991].

Camera ISPs perform an enormous amount of computation on a large amount of data. (Our camera pipeline performs roughly 1000 arithmetic operations/pixel, or 120 Gigaops/sec. for 1080p/60 video.) Combined with the fact that ISPs are used in battery-constrained mobile devices, this means that energy efficiency is crucial.

Recent work has shown that a general purpose CPU uses 500⇥ the energy of a custom ASIC for video decoding [Hameed et al. 2010]. While the CPU can be improved, the majority of the ASIC advantage comes from using a long pipeline that performs more arithmetic per byte loaded. On a modern process, loading one byte from off-chip DRAM uses 6400⇥ the energy of a 1 byte add; even a large cache uses 50⇥ the energy of the add [Malladi et al. 2012; Muralimanohar and Balasubramonian 2009]. ISPs are ideally tuned to these constraints: they achieve high energy efficiency by performing a large number of operations per input loaded, and exploiting locality to minimize off-chip DRAM bandwidth. Existing commercial ISPs use less than 200mW to process HD video [Aptina]

Programming Model

Based on the patterns exploited in ISPs, we define the Darkroom programming language. Its programming model is similar to prior work on image processing, such as Popi, Pan, and Halide [Holzmann 1988; Elliott 2001; Ragan-Kelley et al. 2012]. Images at each stage of computation are specified as pure functions from 2D coordinates to the values at those coordinates, which we call image functions. Image functions are defined over all integer coordinates (x, y), though they can be explicitly cropped to a finite region using one of several boundary conditions

In our notation, image functions are declared using a lambda-like syntax, im(x,y). For example, a simple brightening operation applied to the input image I can be written as the function

It cannot be collapsed any further without changing the stencils of the individual computations. Notice that this is not a linear pipeline, but a general DAG of operations communicating through stencils. In this example, the final sharpened result is composed of stencils over both the horizontally-blurred intermediate, and the original input image

Generating Line-buffered Pipelines

Given a high-level program written in Darkroom, we first transform it into a line-buffered pipeline. This pipeline processes input in time steps, one pixel at a time. During each time step, it consumes one pixel of input, and produces one pixel of output. The pipeline can contain both combinational nodes that perform arithmetic, and line buffers that store intermediate values from the previous time step.

Despite being correct, this pipeline is not optimal: there is an unnecessary line buffer after Obs which would disappear if we choose to shift Obs by 1. To create an optimal pipeline, we must choose shifts which both ensure causality and minimize line buffer size.

The general case is complicated by the fact the program may have multiple inputs and multiple outputs (e.g., an RGB image and a separately-calculated depth map). Furthermore, individual line buffers are not always the same size. For instance, some values may be 1-byte greyscale while others might be 3-byte RGB triples. Fig. 6 shows an example of where different sized outputs can produce different scheduling decisions.

We can formulate this optimization as an integer linear programming problem. Let F be the set of image functions. For each value p(x + d) evaluated in the process of evaluating c(x), we generate a

Implementation

After generating an optimized line-buffered pipeline, our compiler instantiates concrete versions of the pipeline as ASIC or FPGA hardware designs, or code for CPUs (Fig. 7). The Darkroom compiler is implemented as a library in the Terra language [DeVito et al. 2013] that provides the im operator. When compiled, Darkroom programs are first converted into an intermediate representation (IR) that forms a DAG of high-level stencil operations. We perform standard compiler optimizations such as common sub-

expression elimination and constant propagation on this IR. A program analysis is done on this IR to generate the ILP formulation of line buffer optimization, described in the previous section. We solve for the optimal shifts using an off-the-shelf ILP solver (lpsolve), and use them to construct the optimized pipeline [Berkelaar et al. 2004]. It converges to a global optimum in less than a second on all of our test applications. The optimized pipeline is then fed as input to either the hardware generator, which creates ASIC designs and FPGA code, or the software compiler, which creates CPU code

Many Darkroom programs we have written contain a DAG of dependencies, where image functions have multiple inputs and multiple outputs. In order to support these programs in our hardware implementation, we first translate the Darkroom program into an equivalent Darkroom program that is a straight pipeline. This process is described in Fig. 9. While this process always produces semantically correct results, the merging of nodes in the programs can create larger line buffers than what could be achieved with a hardware implementation that supported DAG pipelines. In the future, we plan to multi-port the buffers to eliminate this restriction.

Following DAG linearization, we use Genesis2, a Verilog metaprogramming language [Shacham et al. 2012] to elaborate the topology into a SystemVerilog hardware description for synthesis. We verify functionality of the hardware description using Synopsys VCS G-2012.09, which also produces the activity factor required for ASIC power analysis. The ASIC design is synthesized and analyzed using Synopsys Design Compiler Topographical G-2012.06-SP5-1 for a 45nm cell library. The FPGA design uses Synopsys Synplify G-2012.09-SP1 to synthesize the design and Xilinx Vivado 2013.3 to place and route the design for the Zynq XC7Z045 system-on-a-chip on the Zynq 706 demo board. The FPGA performance is measured on the demo board using a custom Linux kernel module

CONCLUSION

We presented Darkroom, a compiler which takes a high-level definition of image processing code and maps it efficiently to ASICs, FPGAs and modern CPUs. Minimizing working set is crucial on CPU, FPGA, and ASIC, because each has a hard limit on the amount of local fast storage available: CPUs have a limited cache, FPGAs have a limited number of BRAMs, and ASICs are often limited by area. We showed that, thanks to Darkroom's carefully restricted programming model, an ILP scheduling algorithm is able to quickly schedule image processing programs into line-buffered pipelines with minimal intermediate storage. Darkroom synthesizes these optimized pipelines into efficient ASIC designs and FPGA implementations capable of real-time processing of high resolution images at video rates, and CPU code competitive with state of the art image processing compilers. Our initial ASIC and FPGA results are especially exciting, because they fit within a modest hardware budget on a 45nm process, or a small fraction of a mid-range FPGA, suggesting that there is opportunity to prototype much larger real-time image processing pipelines, given the right tools. We have shown that Darkroom's programming model has enough generality to be useful, but we also believe that some of its restrictions could be reduced without eliminating its fast, predictable scheduling. We are interested in extending Darkroom to support image pyramids and serial operations, both of which would allow it to support operations that can propagate information further than the stencil size. This would enable applications like optical flow with larger search windows, or region labeling. In addition, there is significant interest in further investigating how to best map our programming model to existing architectures like CPUs, GPUs and DSPs, with an eye towards understanding how these architectures could be modified to support these image processing workloads with better energy efficiency. We are excited about new areas of research Darkroom enables for the graphics and imaging

community. First, extending prior work on the Frankencamera, mobile camera platforms that include FPGAs programmed by Darkroom would allow researchers to quickly experiment with new applications in real cameras, with real-time performance [Adams et al. 2010]. Second, we believe our approach has the potential to accelerate the development of commercial ISP ASICs, eventually enabling new image processing and computer vision applications on future cameras.

REFERENCES

[1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, "Gpu computing", Proceedings of the IEEE 96 (5) (2008) 879–899,2008.

[2] H. Zhu, Y. Cao, Z. Zhou, M. Gong, "Parallel multi-temporal remote sensing image change detection on gpu", in: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, IEEE, pp. 1898–1904, 2012.

[3] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips,W.-m. Hwu, "Gpuclusters for high-performance computing", in: Cluster Computing and Workshops, 2009. CLUSTER'09.IEEE International Conference on, IEEE, pp. 1–8, 2009.

[4] H. Takizawa, H. Kobayashi, " Hierarchical parallel processing of large scale data clustering on a pc cluster with gpu co-processing", The Journal of Supercomputing 36 (3) 219–234, 2006.

[5] J. Sanders, E. Kandrot, "CUDA by example: an introduction to general purpose GPU programming", AddisonWesley Professional, 2010.

[6] M. O. Guld, C. Thies, B. Fischer, T. M. Lehmann, "A generic concept for the implementation of medical image retrieval systems", international journal of medical informatics 76 (2) 252–259, 2007. 372