Area-Energy Aware Dataflow Optimisation of Visual Tracking Systems

Paulo Garcia[1], Deepayan Bhowmik[2], Andrew Wallace[1], Robert Stewart[3], and Greg Michaelson[3]

[1] School of Engineering & Physical Sc., Heriot-Watt University, Edinburgh, EH14 4AS, U.K.
[2] Department of Computing, Sheffield Hallam University, Sheffield, S1 1WB, U.K.
[3] School of Mathematical & Computer Sc., Heriot-Watt University, Edinburgh EH14 4AS, U.K.
{p.garcia, a.m.wallace, r.stewart, g.michaelson}@hw.ac.uk.
deepayan.bhowmik@shu.ac.uk

Abstract. This paper presents an orderly dataflow-optimisation approach suitable for area-energy aware computer vision applications on FPGAs. Vision systems are increasingly being deployed in power constrained scenarios, where the dataflow model of computation has become popular for describing complex algorithms. Dataflow model allows processing datapaths comprised of several independent and well defined computations. However, compilers are often unsuccessful in identifying domain-specific optimisation opportunities resulting in wasted resources and power consumption. We present a methodology for the optimisation of dataflow networks, according to patterns often found in computer vision systems, focusing on identifying optimisations which are not discovered automatically by an optimising compiler. Code transformation using profiling and refactoring provides opportunities to optimise the design, targeting FPGA implementations and focusing on area and power abatement. Our refactoring methodology, applying transformations to a complex algorithm for visual tracking resulted in significant reduction in power consumption and resource usage.

1 Introduction

The dataflow model of computation has become popular in the image processing/computer vision domain for describing complex algorithms [14]. Currently, several different languages and compilers exist for myriad implementation platforms, *e.g.*, processor architectures, GPUs and FPGAs [6], as well as heterogeneous combinations. The computation model, *i.e.*, independent, parallel actors encapsulating computations, connected to form a dataflow network, is ideal for separation of concerns, computational composition and code re-use. In computer vision, where several low level image processing patterns are frequently re-used [13], a dataflow model allows expressing an algorithm as a processing datapath comprised of several independent and well defined computations.

When implementing *simple* algorithms, where *simple* means static complexity, stateless computations, predictable runtimes and no feedback loops, it is straightforward for a dataflow compiler to analyse the code and identify optimisation opportunities [16]; and refactoring the dataflow network in order to optimise particular metric(s), *e.g.*, area, power, performance. However, contemporary computer vision algorithms are, more often than not, dynamic in complexity, stateful, variable in runtime and result in dataflow networks with feedback loops [15]; thus, they exhibit properties which hinder compiler optimisations [11]: it is left to the designer to manually optimise for the given metrics, which is a non-trivial task in the lack of a formally defined methodology.

Literature suggests generic dataflow specific optimizations that are often non-domain specific and hence do not consider any common patterns in related algorithms. For example, Hueske *et al.* [5] leverage static code analysis to extract information from Map-Reduce-style user-defined functions where the approach is only applicable to Map-Reduce-style code. Based on the post-processing of dataflow execution traces, Brunet *et al.* [2] present a methodology that enables designers to make principled choices in the design space focusing solely on buffer sizes. Schulte *et al.* [12] have identified the problem of power optimizations in dataflow and researched the use of genetic optimization algorithms for software implementations, but did not consider hardware implementations. Kim *et al.* [7] presented a framework for algorithm acceleration from the dataflow to synthesized HDL design, but do not consider size or power.

In this paper, we describe a methodology for the optimisation of dataflow networks, according to patterns often found in computer vision systems. This methodology and associated techniques will be of use for computer vision algorithm designers who must optimise their implementations to meet some metric budget; particularly relevant in remote or mobile applications, where size and power are first class concerns. We demonstrate and evaluate our refinements using a popular complex dynamic computer vision algorithm, mean shift object tracking [3], targeting FPGAs, which are becoming increasingly ubiquitous deployment platforms for remote/mobile computer vision. To the best of our knowledge, such an approach towards area-energy aware domain specific dataflow optimisation is first of its kind.

2 Background

Vision systems are increasingly being deployed in power constrained scenarios, such as automotive, robotics or remote sensing. Whilst deployment on CPU/GPU combinations has been the norm in the past few years, the constrained power budget has motivated the adoption of FPGAs as standard deployment platforms [9]. Softcore solutions for image processing have emerged and the research zeitgeist is the development of novel low-power techniques [18]. In this paper, we target power and size constraints in dataflow based design flows of image processing/computer vision systems.

2.1 Dataflow

A dataflow graph models a program as a directed graph. The model is depicted in Fig. 1. *Tokens* move between asychronously communicating stateful and well defined functional blocks called *actors*. They transform input streams into output streams via *ports*, connected with *wires*. Inside an actor is a series of fireable sequences of instructions. These instructions are encapsulated within *actions*, and the steps an actor takes determines which ports tokens are consumed and emitted and also which state-modifying instructions are executed. The conceptual dataflow model of explicit data streaming and functional units maps well onto FPGA design comprising explicit wires and basic building blocks [17].

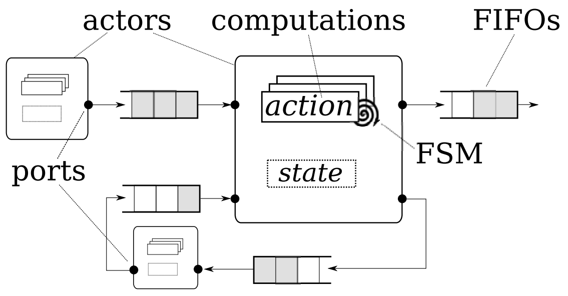

Fig. 1. The Dataflow Process Model

Our dataflow transformations are implemented into Orcc [19] that compiles CAL, a Dataflow Process Network (DPN) [8] language implementation with dynamic properties: *e.g.*, *guards* can be attached to an action to predicate its firing not only on the availability of a token on a given port, but also on its value; explicit *finite state machine* (FSM) transitions between actions, an implicit predication on firing actions as only actions reachable within one transition in the FSM declaration are fireable; *priority* statements declare an inequality between two actions.

2.2 Power on FPGAs

Power consumption on FPGAs consists of *a) static power*, which is directly proportional to the amount of used logic, technology & transistor types; and *b) dynamic power*, which is a weighted sum of several components (these include clock signal propagation power, proportional to clock frequency; signals power, proportional to signal switching rates, among others). Power consumption minimization techniques can be broadly classified in: *1)* computation independent and *2)* computation dependent. *Computation independent* techniques neither alter the behavior nor the results of a system which include clock/power/input gating of unused sub-systems [10], dynamic frequency scaling according to load [4] and different implementation strategies (*e.g.*, BRAMs or LUTs) of the given algorithmic.

Computation dependent techniques modify the behavior and/or the results of a system, minimizing power consumption at the expense of performance or accuracy. These include *1) stored data bit width*: by minimizing the number of bits stored, both static and dynamic power can be reduced as a consequence of smaller required data structures (BRAMs or LUTs). For example, it is possible to discard several least significant bits in image pixels either evenly across color channels; *2) computation bit width*: by minimizing bit widths of signals used for computations, smaller datapaths (eliminating least significant bits) or range (eliminating most significant bits using saturated arithmetic) can be realized; *3) arithmetic approximations*: performing calculations on integers rather than floating point numbers consumes less power. Numerical approximations, *e.g.*, square root, trigonometric functions, can be quantized to require less logic; *4) iterations*: upper limits on algorithm convergence criteria directly impact the number of computations perform (also relates to numerical approximations); and *5) data access* *ordering*: when using external memory, high spatial locality greatly contributes to reducing power consumption. All of these optimizations are realizable at the expense of an acceptable reduction in algorithmic accuracy.

3 Area-Energy Aware Implementation Refinements

Our methodology consists of a profiling-refactoring loop, where profiling identifies optimization opportunities and refactoring applies code transformations to optimize the design. Profiling is performed in an orderly fashion considering three major domain specific power-area optimization criteria, observed in computer vision algorithms: *a)* streamlined memory usage, *b)* back-propagation of bit width requirements and *c)* dataflow actor fusion. Profiling is done through simulations, aided by automated tools, and focuses on identifying optimizations which are not discovered automatically by a general purpose non-domain specific optimizing compiler due to nuances in dataflow semantics. To show the effectiveness of our method, we present a case study (implemented in Orcc-CAL dataflow language) of a popular mean shift visual tracking algorithm. However, the proposed methods are portable across other vision algorithms. Because we target FPGA implementations, we focus our analysis on area and power optimizations, rather than performance which has been already addressed in details by related work [15].

3.1 Streamlined Memory Usage

Dataflow actors are independent computational units and every actor's internal memory requirements are eventually mapped to FPGA logic (*i.e.*, LUTs or BRAMs) and consume precious space and power. Hence, it is essential to minimize memory requirements without jeopardizing the algorithm. Our analysis of open-source code repositories[4] reveals several cases where actors can be refactored to minimize memory usage: a recurring design pattern is the use of unnecessary local arrays. Consider the following code example, which calculates the mean value of an array of pixels:

Listing 1.1. Unnecessary array for pixel mean calculation

```
int buffer[100];
int count := 0;
getValues: action Stream:[value] ==>
do
        buffer[count] := value;
        count := count + 1;
end
filter: action ==> mean:[val]
var int val
do
        val := buffer[0] + buffer[1] +...
        val := val/100;
end
```

[4] https://github.com/orcc/orc-apps

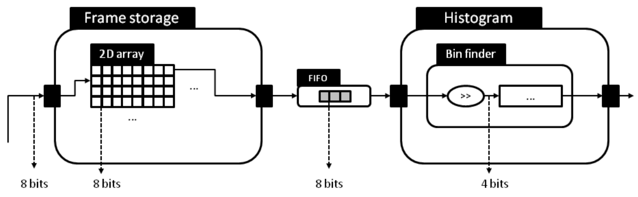

Fig. 2. Actor composition (frame storage and histogram) with bit width usage highlighted.

This is a typical design pattern, where required values are read from input sources, stored in local arrays, then processed (action scheduling logic is not depicted). Now consider the following refactored code:

Listing 1.2. Pixel mean calculation with streamlined memory usage

```
filter: action Stream:[values] repeat 100
            ==> mean:[(val[0]+val[1]=...)/100]
do
end
```

In this version, the number of required input data and the output data dependencies are explicit in the action declaration, removing the need for local arrays. Data is instead stored in the communication FIFOs used to link actors (*i.e.*, input stream). This optimisation is not automatically applied by optimising compilers because in the first version, the same data is used for read and write in two different actions. Automatic refactoring cannot be safely applied, unless the compiler is capable of dertemining that: (1) action firing order is temporaly consistent; (2) no other actions use this data; (3) the output calculation can be re-written in a (syntactically/semantically valid) single line. Precisely ensuring these conditions is still beyond compilers' static analysis capabilities, for languages with such varied semantics such as CAL.

3.2 Back-propagation of Bit Width Requirements

A pervasive pattern in several image processing operations is quantisation, where values (typically pixel color/luminosity components or processed data such as histograms) are scaled down for normalization or other purposes. When values are quantized, lower resolution components (*i.e.*, least significant bits) are not used for subsequent operations. Optimising compilers can reduce data dimensionality locally, *i.e.*, within a function or an actor, but are not capable of extrapolating these optimisations to broaden the range of optimised areas. This is especially prevalent in the dataflow paradigm, where quantization is performed in one actor, whilst the optimization opportunities are present in another; because communication between actors is performed through FIFO channels which establish data size, the compiler fails to infer the optimization. Figure 2 presents a depiction of this pattern, where the second actor (histogram) performs the following computation to determine the histogram bin:

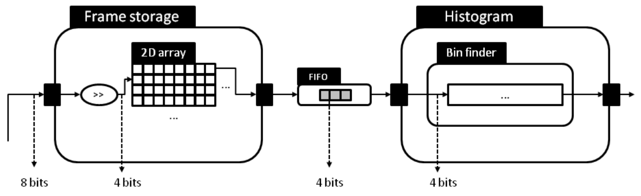

Fig. 3. Actor composition after bit width requirements back-propagation.

Listing 1.3. Histogram computation

```
//uint(size=8)
procedure findBin(uint R, uint G, uint B)
var int r, int g, int b
begin
  r := R >> 4;
  g := G >> 4;
  b := B >> 4;
  binValue := r + (g << 4) + (b << 8);
end
```

In our approach, we identify code sections where unnecessary resolution is used, and we trace the flow of data across the network to determine where to reduce bit widths. In the previous histogram bin finding function, the lowest 4 bits of each datum are not required for computation and can be removed. Back propagating new width requirements, the network can be refactored into the one depicted in Fig. 3.

Again this optimisation is not automatically applied by optimising compilers because data dependencies are not calculated outside actors' boundaries. Safely applying this optimisation would require a compiler to statically determine that: (1) data sources are not used in any other calculation; (2) the communication channel is not used for any other data, and; (3) the shifting operation could safely be applied before data storage.

3.3 Actor Fusion

Another pervasive pattern in image processing, especially when composing systems using third-party code components, is that separation of concerns (*i.e.*, dedicating actors to specific tasks) leads to over-optimizations. Consider the example depicted in Fig. 4(a), where pipeline consists of an actor performing a smoothing filter operation followed by an actor performing binary thresholding. This sequential composition of operations results in a processing pipeline which is not necessarily balanced in function of data throughput. Consider the smoothing filter requires S time units for operation and the binary threshold requires T time units for operation. If latency between sequential data arrival is greater than $S+T$ time units, temporal parallelism offered by the pipeline does not offer any performance improvement. Instead, both actors can be fused in one that takes $S + T$ time units to compute, decreasing space and power costs (refer Fig. 4(b)).

This optimisation is not automatically applied by optimising compilers because it requires some sort of profiling to determine execution times and throughputs, which cannot be performed automatically at compile time. Furthermore, even after profiling,

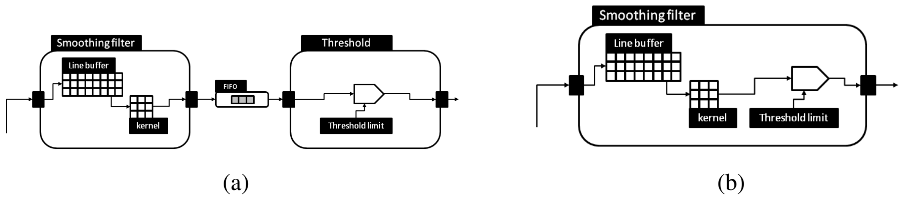

Fig. 4. (a) Over-optimised pipeline resulting in area/energy costs. (b) Optimised pipeline after actor fusion.

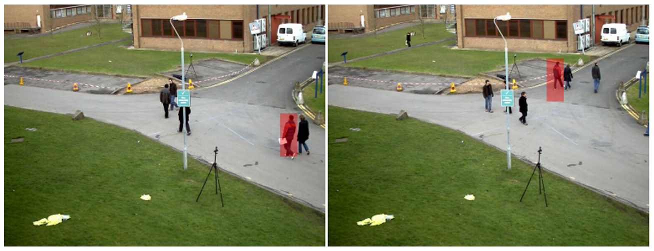

Fig. 5. Example of single target mean shift visual tracking.

a compiler would have to be able to ensure that: (1) profiled execution times would remain constant for any data input, and; (2) merging the two actors would not break the intended behaviour. If any of the two actors possessed additional ports, automatic optimisation would be further complicated, as the compiler would have to ensure that other functionalities remained unaffected by actor fusion.

4 Case Study: Mean Shift Visual Tracking

We use a popular Mean shift tracking algorithm [3] as a use case for applying our methodologies. Mean shift tracking is an object tracking algorithm; given an object's initial position is the first frame, it tracks the object's position in subsequent frames. We use a benchmark data sets (*http://www.cvg.reading.ac.uk/PETS2009/a.html*) in our experiments, and have implemented the complete algorithm on a Xilinx Zedboard, connected to an external camera.

Mean shift [3] is a feature-space analysis technique for locating the maxima of a density function. An example of applying mean shift to image processing for visual tracking is shown in Fig. 5. The target is successfully tracked from the initial frame on the left, to the final frame on the right. The algorithm is a kernel based method normally applied using a symmetric Epanechnikov kernel within a pre-defined elliptical or rectangular window. The target region of an initial image is modelled with a probability density function (a colour histogram) and identifies a candidate position in the next image by finding the minimum distance between models using an iterative procedure. A summary is given in Algorithm 1.

Mean shift tracking exhibits several of the properties of computer vision that hinder automatic optimisations. It is a dynamic algorithm, *i.e.*, only worst case estimations of the time require for execution per frame can be performed, due to iterative

Algorithm 1: Summary of Mean-shift visual tracking

```
   Input: Target position $y_0$ on $1^{st}$ frame;
1  Compute Epanechnikov kernel;
2  Calculate target color model $q_u(y_0)$ (e.g., using RGB color histogram);
3  repeat
       Input: Receive next frame;
4      Calculate target candidate color model: $p_u(y_0)$;
5      Compute similarity function $\rho(y)$ between $q_u(y_0)$ & $p_u(y_0)$;
6      repeat
7          Derive the weights $\omega_i$ for each pixel in target candidate window;
8          Compute new target displacement $y_1$;
9          Compute new candidate colour model $q_u(y_1)$;
10         Evaluate similarity function $\rho(y)$ between $q_u(y_0)$ & $p_u(y_1)$;
11         while $\rho(y_1) < \rho(y_0)$ do
12             Do $y_1 \leftarrow 0.5(y_0 + y_1)$;
13             Evaluate $\rho(y)$ between $q_u(y_0)$ & $p_u(y_1)$;
14         end
15     until $|y_1 - y_0| < \epsilon$ (near zero displacement);
       Output: $y_1$ (Target position for current frame);
16     Set $y_0 \leftarrow y_1$ for next frame;
17 until end of sequence;
```

loops with non-trivial termination conditions. It consists of several different components, each with very different levels of complexity. It is implemented as a network with feedback loops for recursion. Figure 6 depicts the full algorithm, implemented in CAL (*https://goo.gl/TKpN7e*).

Our implementation targets low-power FPGA implementations, and has been prototyped on a Xilinx Zedboard (*https://goo.gl/tsqg1a*). Only integer calculations are performed, and our prototype uses 320x240 frames supplied by an external camera. Software on the attached processor, which is used to feed the video from the FPGA to a remote computer over Ethernet, supplies the initial position (*i.e.*, which object to track) to the mean shift implementation.

4.1 Meanshift transformations

Our initial implementation of Meanshift did not consider any premature optimisations; rather, we attempted to be as faithful to the algorithmic description in Algorithm 1 as possible, following the dataflow paradigm; *i.e.*, each aspect of computations is performed by parallel actors. The *CAL_bin* actor depicted in Fig. 6 is the largest (both in code length and in FPGA resource usage) for two reasons: apart from computations, it also stores the current frame (hence, has the biggest memory requirements in the network) and is responsible for managing network state (*e.g.*, is it processing the first or subsequent frames).

We inspected compilation logs (both for CPU and FPGA backends) to ensure that no optimisations could be applied by the dataflow compiler. Subsequently, we applied

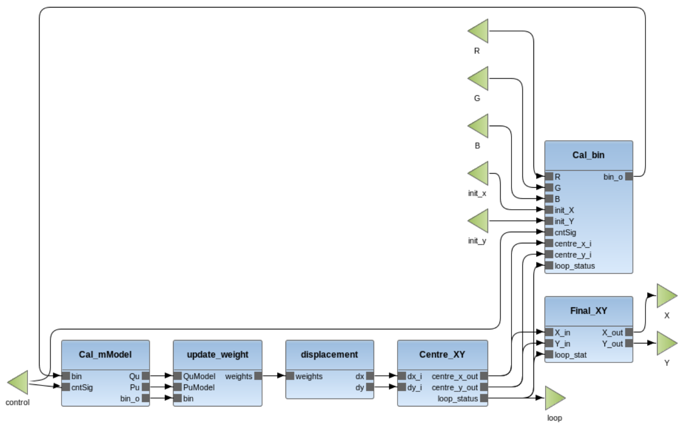

Fig. 6. Meanshift Tracking CAL dataflow process network (final optimised version).

our optimisation methodology and iteratively refined the implementation. At each iteration, we measured power consumption and performance (details are described in Section 5). We do not show detailed code/block diagram examples from Mean shift in this section, as the examples depicted in Section 3 are either identical or sufficiently similar to provide the reader with the necessary understanding.

The first optimisation was the actor fusion transformation (Section 3.3). We observed that the actor responsible for storing the Epanechnikov kernel and providing the results to *CAL_mModel* actor, *K_Array* (no longer depicted in the final version in Fig. 6) performed with the same speed and latency as the following actor (*CAL_mModel*), at a processing speed superior to the rate of data availability. Hence, we clearly identified an over-optimised pipeline which could be fused to reduce size and power.

The second optimisation applied was the transformation described in Section 3.1: streamlined memory usage. We observed that one of the computations in the *update_weight* actor read data from input sources, stored them in local arrays, then processed outputs. After refinement, the number of required input data and the output data dependencies became explicit in the action declaration (*i.e.*, performing a ”pure” computation), removing the need for local arrays.

The final optimisation applied was the transformation described in Section 3.2: back-propagation of bit width requirements, and the one with the most substantial power/size gains. In the original un-optimised version, the *CAL_bin* actor stored the complete current frame; 3 times 320x240 8 bit values (one per RGB colour channel). These values were passed to a *histogram* actor which binned them according to value, performing the “Calculate *target candidate* color model: $p_u(y_0)$” step in the algorithm. After the transformation, previously depicted in Fig. 3, current frame storage required only 4 bits per value. Actors were subsequently fused.

5 Experimental Results and Discussions

At each iteration in our optimisation methodology, we characterized performance at both actor and network level using RTL simulation in Xilinx Vivado Design suite. We

Table 1. Micro benchmarks results

	Power (W)		Usage	
Refinement	Original	Refined	Original	Refined
Streamlined memory usage	0.008	0.006	783/2012 (FF/LUTs)	648/1593 (FF/LUTs)
Bit width back-propagation	0.125	0.096	84 BRAMs	62 BRAMs
Actor fusion	0.017	0.016	190/690 (FF/LUTs)	170/511 (FF/LUTs)

Table 2. Meanshift Power consumption. All reported numbers are in Watt.

	Total	Static	Dynamic	Clocks	Signals	Logic	BRAMS	DSP	I/O
V1	0.461	0.129	0.331	0.112	0.028	0.015	0.172	0.001	0.002
V2	0.356	0.128	0.228	0.070	0.022	0.011	0.123	0.000	0.002
V3	0.321	0.127	0.194	0.070	0.020	0.011	0.091	0.000	0.002

also calculated power consumption, at actor granularity, using Xilinx Power Analyzer embedded in the Vivado suite, reporting high confidence level. Simulation results were verified through physical implementation on a Xilinx Zedboard. Table 1 depicts approximate power and resource usage results for the examples described in Section 3. Because the impact of optimisations is highly dependent on their coverage, *i.e.*, what percentage of an actor is affected by the transformation, it is hard to accurately quantify transformation impact without applying them to a large collection of benchmark programs, which is not feasible without automating refactoring. However, the results in Table 1 should suffice as proof of concept of the proposed transformations' impact.

To provide context to transformation results, Table 2 depicts power consumption for Mean shift versions after successive refinements applications and Table 3 depicts FPGA resource usage. In both tables V1, V2, V3 signifies *(V1)* original un-optimised version (baseline), *(V2)* after actor fusion and streamlined memory usage & *(V3)* after actor fusion, streamlined memory usage and back-propagation of bit width requirements, respectively. Peak performance (maximum clock frequency and achievable frames per second) were unaltered by transformations, at 81MHz and 145fps, respectively. These results provide designers with a quantitative view of how the aforementioned transformations can contribute to decrease resource usage and power consumption.

Our results show that several optimisations which affect power consumption and resource usage can be applied, without compromising functionality or performance: this is certainly desirable for the design of power/size constrained vision systems. However, these cannot be automatically applied by optimising compilers. Two main insights are gained from our experiments: firstly, concerned designers must be aware of manual or semi-automated refactoring methodologies, beyond what is freely given by the compiler. Secondly, compiler optimisation technology, despite great advances in recent years, must still benefit from improvements.

Domain-specific refactoring can be applied by simulation-refinement loops, where performance estimation can expose over-optimised sub-systems, consuming unnecessary power and resources (*e.g.*, actor fusion example). Manual code inspection can re-

Table 3. Meanshift FPGA usage on Xilinx Zedboard (Zynq 7020).

	Registers	LUTs	BRAM	DSP
V1	3792 (3.00%)	9603 (18.00%)	111 (79.00%)	22 (10.00%)
V2	2189 (2.00%)	4679 (8.00%)	124 (88.00%)	8 (3.60%)
V3	2490 (2.34%)	4066 (7.64%)	67 (48.00%)	8 (3.60%)

veal refactoring opportunities which minimize resource usage, either through local code optimisation (*e.g.*, streamlined memory usage example) or through cross sub-system code optimisation (*e.g.*, back propagation example). Both of these can be aided by automated profilers (*e.g.*, the CAL-Orcc framework supplies Turnus [1]) and by static code analysers. The methodologies and transformations we have presented should act as a guide for image processing engineers to perform such optimisations on their systems.

Optimisation passes in contemporary compilers are limited by language semantics. Static dataflow (*i.e.*, without loops, stateless) is far simpler to analyse and subsequently optimise than dynamic dataflow (*e.g.*, CAL). However, most real-world code relies heavily on dynamic features and is typically composed from third-party sub-systems. Hence, it is necessary to extend optimisation passes with more sophisticated static analysis capabilities that can infer memory waste and cross sub-system optimisations. The methodologies and transformations we have presented should aid compiler designers in identifying optimisation bottlenecks and possible solutions.

On Mean shift tracking, a complex algorithm exhibiting several design properties which inhibit automated compiler optimisations, our transformations resulted in 31% power consumption reduction, from 0.461 to 0.321W, and a reduction of 10.36, 31 and 6.4 percentage points in LUTs, BRAMs and DSPs resources usage, respectively.

6 Conclusions

We have described a methodology for the optimisation of dataflow networks, according to patterns often found in computer vision systems. The proposed methodology will be of use for computer vision algorithm designers who must optimise their implementations to meet some metric budget and for compiler designers in identifying optimisation bottlenecks and possible solutions. Our refactoring methodology, applying transformations to a complex algorithm, resulted in 31% power consumption reduction and a reduction of 10.36%, 31% & 6.4% in LUTs, BRAMs and DSPs resources usage, respectively. We also identified which design/language features were responsible for hindering automated optimisation. Our current work focuses on developing new static analysis technologies and refactoring tools; in future work, we hope to integrate these static refactoring tools into the Orcc framework and extend the optimisation methodologies so they are applicable to other compilers and language paradigms.

Acknowledgement: We acknowledge the support of the Engineering and Physical Research Council, grant references EP/K009931/1 (Programmable embedded platforms for remote and compute intensive image processing applications), EP/K014277/1 (MOD University Defence Research Collaboration in Signal Processing).

References

1. Brunei, S.C., Mattavelli, M., Janneck, J.W.: Turnus: a design exploration framework for dataflow system design. In: Int'l Symp. on Circuits & Systems (ISCAS). pp. 654–654 (2013)
2. Brunet, S.C., Mattavelli, M., Janneck, J.W.: Buffer optimization based on critical path analysis of a dataflow program design. In: Int'l Symp. on Circuits and Systems (ISCAS). pp. 1384–1387 (2013)
3. Comaniciu, D., Ramesh, V., Meer, P.: Kernel-based object tracking. IEEE Trans. on Pattern Analysis and Machine Intelligence 25(5), 564–577 (2003)
4. Ge, R., Vogt, R., Majumder, J., Alam, A., Burtscher, M., Zong, Z.: Effects of dynamic voltage and frequency scaling on a k20 GPU. In: Int'l Conf. on Parallel Processing. pp. 826–833 (2013)
5. Hueske, F., Peters, M., Krettek, A., Ringwald, M., Tzoumas, K., Markl, V., Freytag, J.C.: Peeking into the optimization of data flow programs with MapReduce-style UDFs. In: Int'l Conf. on Data Engineering (ICDE). pp. 1292–1295 (2013)
6. Janneck, J.W., Miller, I.D., Parlour, D.B., Roquier, G., Wipliez, M., Raulet, M.: Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study. In: IEEE Workshop on Signal Processing Systems (SiPS). pp. 287–292 (2008)
7. Kim, Y., Jadhav, S., Gloster, C.S.: Dataflow to Hardware Synthesis Framework on FPGAs. In: Int'l Symp. on Computer Architecture and High Performance Computing Workshops (SBAC-PADW). pp. 91–96 (2016)
8. Lee, E.A., Parks, T.M.: Dataflow process networks. Proc. of the IEEE 83(5), 773–801 (1995)
9. Malik, M., Farahmand, F., Otto, P., Akhlaghi, N., Mohsenin, T., Sikdar, S., Homayoun, H.: Architecture exploration for energy-efficient embedded vision applications: From general purpose processor to domain specific accelerator. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI). pp. 559–564 (2016)
10. Pandey, B., Yadav, J., Pattanaik, M., Rajoria, N.: Clock gating based energy efficient ALU design and implementation on fpga. In: Int'l Conf. on Energy Efficient Technologies for Sustainability (ICEETS). pp. 93–97 (2013)
11. Rheinländer, A., Leser, U., Graefe, G.: Optimization of complex dataflows with user-defined functions. ACM Computing Surveys 50(3), 38:1–38:39 (May 2017)
12. Schulte, E., Dorn, J., Harding, S., Forrest, S., Weimer, W.: Post-compiler software optimization for reducing energy. SIGARCH Comput. Archit. News 42(1), 639–652 (Feb 2014)
13. Seinstra, F.J., Koelma, D.: The lazy programmer's approach to building a parallel image processing library. In: Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS). pp. 1169–1176 (2001)
14. Sérot, J., Berry, F., Bourrasset, C.: High-level dataflow programming for real-time image processing on smart cameras. Journal of Real-Time Image Processing 12(4), 635–647 (2016)
15. Stewart, R., Bhowmik, D., Wallace, A., Michaelson, G.: Profile guided dataflow transformation for FPGAs and CPUs. Journal of Signal Processing Systems 87(1), 3–20 (2017)
16. Stewart, R., Michaelson, G., Bhowmik, D., Garcia, P., Wallace, A.: A dataflow IR for memory efficient RIPL compilation to FPGAs. In: Int'l Conf. on Algorithms and Architectures for Parallel Processing. pp. 174–188 (2016)
17. Teifel, J., Manohar, R.: An asynchronous dataflow FPGA architecture. IEEE Trans. on Computers 53(11), 1376–1392 (2004)
18. Turcza, P., Duplaga, M.: Hardware-efficient low-power image processing system for wireless capsule endoscopy. IEEE J Biomedical and Health informatics 17(6), 1046–1056 (2013)
19. Yviquel, H., Lorence, A., Jerbi, K., Cocherel, G., Sanchez, A., Raulet, M.: Orcc: Multimedia development made easy. In: Proc. ACM int'l conf. on Multimedia. pp. 863–866 (2013)