Andrew Wallace, Institute for Sensors, Signals and Systems
Heriot-Watt University

... being a discussion of some of the activity on

EP/K009931/1: Programmable embedded platforms for remote and compute intensive image processing applications, 2013-2017 (‘RATHLIN’) 

Greg Michaelson, Rob Stewart, Deepayan Bhowmik, Nathanel Lemessa Baisa, HWU, Roger Woods, Fahad Siddiqui, Colm Kelly and Burak Bardak, QUB (with INSA, Clermont-Ferrand, EPFLausanne, Thales, Xilinx)
Objectives of the EPSRC Programme

- Use a “model of computation” dataflow process network (DPN) representation which will allow the processing and data organisation needs of image processing/analysis (IP/A) to be readily captured.
- Develop a domain specific Image Processing Processor (IPPro) processor architecture
- Develop code translation and transformation techniques that will allow efficient implementation on a variety of platforms (e.g. Multicore CPU, FPGA, GPU, IpPro)
- Develop a Domain Specific Language, RIPL, ‘above’ the DPN, for ease of use by practitioners
- Evaluate using a set of prototypical and novel IP/A algorithms expressed as application specific DPNs
The Target: a Distributed, Heterogeneous Architecture

Research Challenge

- Data intensive image processing applications e.g. Video Analytics, Surveillance, Smart Cameras and other sensors
- Option of distributed front-end image processing to reduce communication (and other costs) of backend processing

Research Methodology

- Distributed computing
- Data or control level parallelism (DLP)
- Programmability and Performance
- Single/Multiple Instruction Multiple Data (S/MIMD)

Processor based Architecture

- Programmability
- Scalability
- Flexibility
- Efficient Resource Utilization

e.g. multicore CPUs

Camera Node # 1
Location # 1
Video Detection and Tracking

Situational awareness (car), RFS filter, Mean Shift, Crowd Density/Flow, Wavelet Coding, LiDAR

Camera Node # 2
Location # 2

MB/s

1P1Pro

e.g. Xilinx-7: FPGA hardware plus ARM

Backend Processing

ECIT The Institute of Electronics, Communications and Information Technology

EPSRC Engineering and Physical Sciences Research Council

HERIOT WATT UNIVERSITY
Multi-target tracking, either from a CCTV network, or from a mobile vehicle or vehicles (Sensor – Region – Algorithm Utility)

What kind of Dynamic Imaging?

Multi-target tracking, either from a CCTV network, or from a mobile vehicle or vehicles

What’s the problem?

Code development for parallel or heterogeneous architectures ... e.g. it took several months to hand-craft GPU code to detect, track and associate 5-10 subjects with live video with two cameras at 10fps (40fps on recorded video)

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU time(ms)</th>
<th>GPU time(ms)</th>
<th>SpeedUp</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detection</td>
<td>48.81</td>
<td>8.43</td>
<td>5.8</td>
<td></td>
</tr>
<tr>
<td>Likelihood</td>
<td>30.95</td>
<td>13.60</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td>Fusion</td>
<td>-</td>
<td>0.09</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Resampling</td>
<td>0.44</td>
<td>0.56</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>Transition</td>
<td>2.11</td>
<td>0.37</td>
<td>5.7</td>
<td></td>
</tr>
<tr>
<td>Remove</td>
<td>-</td>
<td>0.02</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Add</td>
<td>0.02</td>
<td>0.31</td>
<td>0.1</td>
<td></td>
</tr>
<tr>
<td>TextureUpdate</td>
<td>-</td>
<td>0.30</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>82.3</td>
<td>23.2</td>
<td>3.5</td>
<td></td>
</tr>
</tbody>
</table>

What kind of Dynamic Imaging?
Sensing Forests: Multispectral LiDAR

Using Multi- or Hyper-spectral Lidar, it is possible to sense a single footprint, or build a 3D image of the scene below. This presents challenges to spectrally unmix pixels and images, such that structure, materials and material variation can be inferred.

What’s the problem?

Code development for parallel or heterogeneous architectures …. e.g. it took several months (and inevitable algorithmic changes) to hand-craft Beowulf code to analyse (RJMCMC) full waveform multispectral LiDAR for tree canopy data, to recover structure and physiology.


Fig. 17. Final fitting result of the real data (shown in Fig. 4) using DP SSD-RJCMC.

Single footprint data
What's the problem? Simple HOG on an IpPro

Direct transformation to a custom FPGA, or (better) direct FPGA Coding of HOG using the IpPro (QUB) is laborious, and not necessarily optimal.

Wouldn’t it be nice if ……..

• We could express any given algorithm as high level abstractions, drastically reducing code development time, yet easily translate that code into executable code for a variety of parallel architectures, and transform that code (using either analysis or profiling) to optimise “performance”, e.g.
  ✓ … for speed, memory use, power consumption, cost, and
• either use a single platform, or mix and match processors (e.g. CPU, FPGA, GPU, IpPro) to meet the desired objectives, yet
• match or even better “hand-crafted” code
Algorithm Development: the Rathlin Model

Domain Specific Language
Actor Language for a Dataflow Process Network (DPN)
Low level instruction set for dedicated Image Processor on Xilinx Hardware

Multi-threaded C for Multicore
Xilinx 7 + Arm
This talk

(p.s. Separate project targets RIPL -> SAC -> GPU)
Compiler Flow for FPGA route: RIPL to CAL

- Inline all function calls into the main function.
- Replace all RIPL type declarations to CAL array declarations.
- Generate stream-based actors for each use of a RIPL iterator.
- Derive dataflow wires from implicit data dependencies between RIPL variables.
- Generate CAL files for each actor, where there is one actor per RIPL iterator.
- Generate an XML/XDF file for the wire connections.
Compiler Flow: CAL to Verilog

The Orcc frontend parses CAL syntax into an abstract syntax tree (AST) which is then mapped to a dataflow IR.

The Xronos backend of the Orcc compiler generates Verilog for each actor, and a VHDL file that describes the network of wires between actors.

It does this by compiling the dataflow IR to a language independent model (LIM) IR which abstracts FPGA hardware.

OpenForge, open sourced by Xilinx, is used to compile LIM IR to Verilog.
Why DPNs?

- Image processing and analysis algorithms can be classified and developed in broad categories based on their algorithmic description:
  - point, local, global, temporal, adaptive or random.

- These classifications can be used to understand the hardware requirements and memory estimations.

- Early exposure to these requirements in dataflow representations can be used in optimisation, resource allocation and code synthesis.

- There is an established and active community working around the open source ORCC tools.
Mean Shift Algorithm – Exemplar

Originally using a fixed template and a colour model with an Epanechnikov Kernel, this algorithm has been used many times (>8000 citations) and has been adapted in many ways for image segmentation and Object Tracking. For our purposes, we have several previous language (Matlab, C, Hume, Renesas) codes, there are a number of published hand-crafted FPGA implementations we can compare against, and it is challenging, because of the optimisation loop and necessary precision, but not impossible for the IpPro.

Comaniciu and Meer, IEEE Trans PAMI 24(5), Mean Shift: a robust approach towards feature space analysis, pp603-618, 2002
The basic approach is to create a probability density function (PDF) in frame (n+1), based on the colour histogram in frame (n) or a fixed model, and use an iterative procedure to find the maximum in this PDF that defines the new position of the object being tracked.

Usually, the PDF is based on the similarity between centre-weighted colour histograms, using an Epanechnikov kernel; the similarity function is usually defined from the Bhattacharya distance.

\[
\rho[\hat{p}(y), q] = \sum_{u=1}^{m} \sqrt{\hat{p}_u(y)q_u}
\]

- **No. of bins in histogram**
- **Bhattacharya distance**
- **Model colour histogram**
- **Target colour histogram at position y**
DPN Exemplar: Mean Shift Algorithm for Tracking

- The kernel is recursively moved from the starting position in the previous frame \( \hat{y}_0 \) to a new position \( \hat{y}_1 \) until convergence.

\[
\hat{y}_1 = \frac{\sum_{i=1}^{n_h} x_i w_i g\left( \frac{\hat{y}_0 - x_i}{h} \right)}{\sum_{i=1}^{n_h} w_i g\left( \frac{\hat{y}_0 - x_i}{h} \right)}
\]

where \( g(x) = -k'(x) \).

\[
\text{and } w_i = \sum_{i=1}^{m} \sqrt{\frac{\hat{q}_u}{\hat{p}_u(\hat{y}_0)}} \delta[b(x_i) - u]
\]

Weights
Samples in PDF
Window size
Similarity function
E-kernel
Model
Candidate

Mean shift tracking algorithm

**Given** object position $y_0$ in frame $n$
Compute Epanechnikov kernel
Compute object colour model, $q_u(y_0)$

**Repeat**
Read next frame $(n+1)$
Compute object candidate model $p_u(y_0)$
Compute similarity function, $p(y)$

**Repeat**
Derive weights $w_i$ for each pixel in candidate window
Compute new candidate position, $y_1$
Evaluate similarity function, $p(y)$

**Until** $|y_1 - y_0| < \epsilon$ (near zero) or oscillatory or limit

**Until** end of sequence
What does the CAL Program look like (1)?

First, there is a Dataflow Process Network consisting of several actors – normally encoded with a XDF file that defines the connectivity and parameter passing between the several actors.

What does the CAL Program look like (2)?

Second, there are CAL statements within each actor that define its function (e.g. Centre_XY).

This Actor has four actions scheduled by a FSM.

---

```
package main;
import std.header.Parameter.*;

actor updateCentreXY() int dx_i, int dy_i ==>

uint centre_x_out, uint centre_y_out, bool loop_status:

uint centre_x ;
uint centre_y ;
int dx;
int dy;
int loopcount := 0;

initialise: action ==>
   do
      centre_x := CENTRE_X;
      centre_y := CENTRE_Y;
   end
   loopcount := 0;

get_dx_dy: action dx_i:[val_x], dy_i:[val_y] ==>
   do
      dx := val_x;
      dy := val_y;
   end

while_loop_status := false;
updateCentreXY: action ==>
   do
      centre_x := centre_x + dx;
      centre_y := centre_y + dy;
      loopcount := loopcount + 1;
      if(((dx=0) & dy=0) || (loopcount>20)) then
         while_loop_status := false;
      else
         while_loop_status := true;
      end
   end

send: action ==> centre_x_out:[val_x],
         centre_y_out:[val_y], loop_status:[val_loop]

var
uint val_x,
uint val_y,
bool val_loop
   do
      val_x := centre_x;
      val_y := centre_y;
      val_loop := while_loop_status;
   end

schedule fsm s0 :
   s0 (initialise ) --> s1;
   s1 (get_dx_dy) --> s2;
   s2 (updateCentreXY) --> s3;
   s3 (send) --> s1;
```
<table>
<thead>
<tr>
<th>Actor</th>
<th>Slice LUT</th>
<th>Slice registers</th>
<th>Block RAM /FIFO</th>
<th>DSP48E</th>
<th>FMax (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>3664</td>
<td>8777</td>
<td>88</td>
<td>49</td>
<td>55.41</td>
</tr>
<tr>
<td>Final_XY</td>
<td>76</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>721.48</td>
</tr>
<tr>
<td>Centre_XY</td>
<td>182</td>
<td>199</td>
<td>0</td>
<td>0</td>
<td>530.81</td>
</tr>
<tr>
<td>Stream_to_YUV</td>
<td>90</td>
<td>287</td>
<td>24</td>
<td>0</td>
<td>420.07</td>
</tr>
<tr>
<td>update_model</td>
<td>1042</td>
<td>2399</td>
<td>30</td>
<td>0</td>
<td>148.74</td>
</tr>
<tr>
<td>YUV2RGB</td>
<td>300</td>
<td>957</td>
<td>7</td>
<td>0</td>
<td>126.71</td>
</tr>
<tr>
<td>displacement</td>
<td>545</td>
<td>1326</td>
<td>2</td>
<td>9</td>
<td>73.40</td>
</tr>
<tr>
<td>update_weight</td>
<td>556</td>
<td>1544</td>
<td>14</td>
<td>4</td>
<td>66.46</td>
</tr>
<tr>
<td>kArray_derv</td>
<td>437</td>
<td>1074</td>
<td>1</td>
<td>18</td>
<td>55.44</td>
</tr>
<tr>
<td>kArray_evaluation</td>
<td>460</td>
<td>1148</td>
<td>1</td>
<td>18</td>
<td>55.41</td>
</tr>
</tbody>
</table>
### DPN: Applying Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Actor fusion</td>
<td>Combines multiple actors into a single actor.</td>
</tr>
<tr>
<td>2 Actor fission</td>
<td>Load balance data between replicas of an actor.</td>
</tr>
<tr>
<td>3 Loop fission</td>
<td>Load balance data between replicas of a loop.</td>
</tr>
<tr>
<td>4 Actor pipelining</td>
<td>Pipelines an actor’s instructions into separate actors.</td>
</tr>
<tr>
<td>5 Task parallelism</td>
<td>Decomposes an expression into separate sub-expression.</td>
</tr>
<tr>
<td>6 Loop elimination</td>
<td>Replaces loops with on-the-fly streaming.</td>
</tr>
<tr>
<td>7 FSM simplification</td>
<td>Re-writes FSMs so that all states are always live.</td>
</tr>
<tr>
<td>8 Built-in constructs</td>
<td>Use constructs with optimised FPGA implementations.</td>
</tr>
</tbody>
</table>

Other computational transformations are considered for FPGAs, notably Floating vs Fixed point implementation and the use of LUTs.
Example 1: Data Parallelism & Actor Fission

Transformation 2 Data parallelism with actor fission

\[
\begin{align*}
\text{\texttt{ACTOR\ A}} & \quad \text{In} \Rightarrow \text{Out} : \\
\text{\texttt{action}} & \quad \text{In}: [x] \Rightarrow \text{Out}: [x] \\
\text{\texttt{ACTOR\ B}} & \quad \text{In} \Rightarrow \text{Out} : \\
\text{\texttt{action}} & \quad \text{In}: [x] \Rightarrow \text{Out}: [\neg x] \\
\text{\texttt{ACTOR\ C}} & \quad \text{In} \Rightarrow \text{Out} : \\
\text{\texttt{action}} & \quad \text{In}: [x] \Rightarrow \text{Out}: [x]
\end{align*}
\]

\[
\begin{align*}
\text{\texttt{ACTOR\ A}} & \quad \text{A} \Rightarrow \text{01,02} : \\
\text{\texttt{action}} & \quad \text{In}: [x_1,x_2] \Rightarrow \text{01}: [x_1], \text{02}: [x_2] \\
\text{\texttt{ACTOR\ B}_1} & \quad \text{A} \Rightarrow \text{Out} : \\
\text{\texttt{action}} & \quad \text{In}: [x] \Rightarrow \text{Out}: [\neg x] \\
\text{\texttt{ACTOR\ B}_2} & \quad \text{A} \Rightarrow \text{Out} : \\
\text{\texttt{action}} & \quad \text{In}: [x] \Rightarrow \text{Out}: [\neg x] \\
\text{\texttt{ACTOR\ C}} & \quad \text{A} \Rightarrow \text{Out} : \\
\text{\texttt{action}} & \quad \text{In}: [x] \Rightarrow \text{Out}: [x]
\end{align*}
\]
Actor Fission: applied to update weights

Update weights: action →

Do

/* data parallelisable */
Foreach int i in 0 .... (NUMBINS) -1 do
    If (Pu_model_buffer[i] = 0) then
        R[i] := 0;
    Else
        sqrt ((Qu_model_buffer[i]/Pu_model_buffer[i]));
        R[i] := sqrtvalue;
    End
End

/* barrier necessary between two loops – not task parallelisable */
/* data parallelisable, but not cost-effective */
Foreach int x in 0 .... (X_SIZE-1) do
    Foreach int y in 0 .... (Y_SIZE-1) do
        weight_buffer[x][y] := R[bin_buffer{x}[y]];
    End
End
End
Actor Fission: Update weight (Mean Shift)

(a) before 
(b) after
Example 2: Task Parallelism

**Transformation 5** Decompose expression into task parallel actors

\[
\begin{align*}
\text{ACTOR A} & \rightarrow \text{Out} : \\
& \text{action} \rightarrow \text{Out} : [e_1] \\
& y := (e_1 \cdot e_2 \cdot e_3) \\
\end{align*}
\]

\[
\begin{align*}
\text{ACTOR B} & \rightarrow \text{Out} : \\
& \text{action} \rightarrow \text{Out} : [e_2 \cdot e_3] \\
\end{align*}
\]

\[
\begin{align*}
\text{ACTOR C} & \text{ In1, In2 } \rightarrow \text{Out} : \\
& \text{action In1}[x], \text{In2}[y] \rightarrow \text{Out} : [x \cdot y] \\
\end{align*}
\]
Task Parallelism: Displacement (Mean Shift)
### Applying Transformations to Mean Shift

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Transformation</th>
<th>Registers</th>
<th>Slice LUTs</th>
<th>BRAM</th>
<th>DSP</th>
<th>Clock (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream to YUV</td>
<td>None</td>
<td>90</td>
<td>287</td>
<td>24</td>
<td>0</td>
<td>420.0</td>
</tr>
<tr>
<td></td>
<td>Loop elimination</td>
<td>27</td>
<td>85</td>
<td>0</td>
<td>0</td>
<td>386.7</td>
</tr>
<tr>
<td>YUV to RGB</td>
<td>None</td>
<td>300</td>
<td>957</td>
<td>7</td>
<td>0</td>
<td>126.7</td>
</tr>
<tr>
<td></td>
<td>Actor fusion</td>
<td>99</td>
<td>353</td>
<td>0</td>
<td>0</td>
<td>182.8</td>
</tr>
<tr>
<td>Displacement</td>
<td>None</td>
<td>545</td>
<td>1326</td>
<td>2</td>
<td>9</td>
<td>73.4</td>
</tr>
<tr>
<td></td>
<td>Task parallelism</td>
<td>791</td>
<td>1210</td>
<td>7</td>
<td>9</td>
<td>110.0</td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>556</td>
<td>1544</td>
<td>14</td>
<td>4</td>
<td>66.5</td>
</tr>
<tr>
<td></td>
<td>Fission</td>
<td>12352</td>
<td>19878</td>
<td>55</td>
<td>128</td>
<td>72.5</td>
</tr>
<tr>
<td>Update weight</td>
<td>Just square root (none)</td>
<td>346</td>
<td>548</td>
<td>0</td>
<td>4</td>
<td>72.5</td>
</tr>
<tr>
<td></td>
<td>Square root Lookup</td>
<td>139</td>
<td>227</td>
<td>32</td>
<td>0</td>
<td>368.2</td>
</tr>
<tr>
<td></td>
<td>Combined</td>
<td>7907</td>
<td>38544</td>
<td>1028</td>
<td>0</td>
<td>225.9</td>
</tr>
<tr>
<td>k-array derive</td>
<td>None</td>
<td>437</td>
<td>1074</td>
<td>1</td>
<td>18</td>
<td>55.4</td>
</tr>
<tr>
<td></td>
<td>Loop promotion</td>
<td>4447</td>
<td>12484</td>
<td>5</td>
<td>144</td>
<td>52.7</td>
</tr>
</tbody>
</table>

But recall this is ‘once only’
Final results: Mean Shift

<table>
<thead>
<tr>
<th></th>
<th>Slice LUT</th>
<th>Registers</th>
<th>BRAM</th>
<th>DSP48E</th>
<th>Clock (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive version</td>
<td>2751</td>
<td>6582</td>
<td>86</td>
<td>13</td>
<td>66.5</td>
</tr>
<tr>
<td>Optimising HDL for speed</td>
<td>2751</td>
<td>6635</td>
<td>86</td>
<td>13</td>
<td>66.5</td>
</tr>
<tr>
<td>Optimising HDL for area</td>
<td>2748</td>
<td>6610</td>
<td>87</td>
<td>13</td>
<td>66.5</td>
</tr>
<tr>
<td>Dataflow optimisations</td>
<td>10786</td>
<td>51267</td>
<td>1026</td>
<td>9</td>
<td>110.0</td>
</tr>
</tbody>
</table>

Table 4: Comparison of Dataflow and HDL Level Optimisation Results

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Transformation</th>
<th>Runtime</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive version</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Updating the model</td>
<td>FSM simplification</td>
<td>2.06s</td>
<td>63.1</td>
</tr>
<tr>
<td>Displacement</td>
<td>Task parallelism</td>
<td>3.17s</td>
<td>41.0</td>
</tr>
<tr>
<td>Compute tracking window</td>
<td>Loop elimination</td>
<td>2.96s</td>
<td>43.9</td>
</tr>
<tr>
<td>RGB to YUV</td>
<td>Actor fusion</td>
<td>2.67s</td>
<td>48.7</td>
</tr>
<tr>
<td>k-array evaluation</td>
<td>Language use</td>
<td>2.34s</td>
<td>55.6</td>
</tr>
<tr>
<td><strong>Combined</strong></td>
<td></td>
<td><strong>1.70s</strong></td>
<td><strong>76.5</strong></td>
</tr>
</tbody>
</table>

Table 5: Transformation effects on CPU results
Conclusions (the story so far)

- We have applied DPN transformations to optimise algorithms expressed in the CAL dataflow language.
- This identifies transformations to target FPGAs; e.g. for Mean Shift, the overall clock frequency is increased from 66.5MHz to 110MHz.
- Applying all CPU targeting transformations increases mean throughput from 43fps to 77fps.
- In general, coding is (arguably) much simplified, e.g. a wavelet transformation is 4 lines of RIPL, 34 lines of CAL, and over 1000 lines of VHDL code.
- We have also developed an IpPro architecture, a partially reconfigurable soft core processor that will continue to evolve.
Future Work

- A key priority is to embed the dataflow transformations as compiler optimisations guided by FPGA simulation and CPU traced-based profiling.

- As the project develops, we hope to target the IpPro from both RIPL and Dataflow networks.

- We are developing concurrently new algorithms for dynamic video data analysis,
  - Random Finite Set approach to track multiple targets of two distinct types in clutter (e.g. pedestrians and vehicles, sheep and goats)
  - Crowd density and flow estimation techniques, that we hope to use to improve detection and tracking in sparse and dense populations

R. Stewart, D. Bhowmik, A Wallace, G Michaelson, Profile guided dataflow transformation for FPGAs & CPUs, IEEE Global Conference on Signal and Information Processing, December 2014 + Journal Submit.