Evolving programs to manipulate images
- Noise removal
- Arbitrary filters (current and future work).

The real novelty here is the use of the GPU
- But, we do make some additions to the technique that become feasible with additional computational resources.

We will evolve a program/expression that will take a neighbourhood of pixels, apply a convolution and output a new value for the centre pixel.
**GP as an image filter**

- Form of GP based on acyclic directed graphs
- Implicit re-use of nodes in the graph
- Fixed length genotype
  - List of integers encoding the nodes in the graph
- Bounded variable length phenotype
  - Not all nodes in the graph are connected

**Cartesian Genetic Programming**

- A form of neutrality present in CGP
  - Shown to be beneficial to the evolutionary process

**Function set**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>SUM</td>
</tr>
<tr>
<td>SUB</td>
<td>MIN</td>
</tr>
<tr>
<td>MULT</td>
<td>PROD</td>
</tr>
<tr>
<td>DIV</td>
<td>QUOT</td>
</tr>
<tr>
<td>LOG2</td>
<td>LOG2</td>
</tr>
<tr>
<td>RECIPRICAL</td>
<td>RECIPRICAL</td>
</tr>
<tr>
<td>RSQRT</td>
<td>RSQRT</td>
</tr>
</tbody>
</table>

**The evolutionary algorithm**

- Population of size 50
- 5% of genes mutated
- No crossover
- 5 best individuals promoted to next generation
- Graph size of 50 nodes
- Maximum of 50,000 evaluations

**MS Accelerator**

- We chose to use Microsoft Research's Accelerator
- [http://research.microsoft.com/act/](http://research.microsoft.com/act/)
MS Accelerator

ADVANTAGES
- Simple to use
- Highly abstracted
- Lazy evaluator
- CPU mode

DISADVANTAGES
- Windows only
- May be too highly abstracted

Available functions

- GPUs have a full range of mathematical operations
- Some are specific to vector manipulation
- Others are more useful in parallel uses

Evolving image filters

- Computational problem:
  - We need to apply this evolved program to every pixel in an image.
  - We also need to compute some sort of fitness score.
  - We need to do this for every individual in our population in every generation.

MS Accelerator – example code

```c
#include <stdio.h>

void ParallelArray() {
    float ParallelArray[4][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    float CPU_Result[4][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

    // Process CPU_Result array here
}
```

Available functions

- MATHEMATICAL
  - Abs, add, ceiling, cos, divide, floor, log, multiply, multiply add, pow, reciprocal, rsqrt, sqrt, subtract

- BINARY
  - And, or, not,

- VECTOR
  - Sum, rotate, add/drop dimension, gather, inner product, product, stretch, select, section

- LOGICAL
  - ==, <=, >=, !=, <, >

Evolving image filters

- There is actually relatively little research on evolving filters using GP.
- Why? It takes too long on a CPU to test an individual.
- A lot of the research looks into evolving hardware (e.g. FPGA) implementations.
Evolving image filters

- Most research looks at applying filters to a single 256x256 pixel image.
- But how do we know if this evolved program would work on other images, with different characteristics?

Fitness function

- Find the average error of each pixel
- Where the error is the difference between the output of the evolved program and the target image.
- This is also computationally expensive, but can be processed in parallel on the GPU.

Noise removal

- Salt and pepper
  - 5% of pixels set to either black or white
- Random noise
  - 5% of pixels set to random grey value

Evolving image filters on the GPU

- We need to map to the SIMD architecture and the MS Accelerator way of working.
- Accelerator is purely element wise.
  - Simple API leads to some minor issues.
- But how do we get the neighbourhoods of pixels?
  - Shifting!
  - Would be much better to do some sort of direct access – but this is not available in this API.

Ideally, we need to train on a bunch of pictures.
- Better still, we should be able to validate these filters on another set of images.
  - In the WCCI paper I did not do this, but will later briefly describe it as part of our ongoing work.
Here we train on 4 different images (to help prevent over-fitting).

We also tried compared using 4 copies of each image.

We corrupt the images with noise, and then find a filter that produces an image that is close to the original image.

We present multiple images simultaneously.

In effect, we process one large image.

However, we must be careful of the edges where different images touch.

We apply a mask to the fitness function.

The mask is another image consisting of pixels of either 0 or 1.

When we find the difference between two images, we then multiply this by the mask.

In effect, we process one large image.

Different images touch.

We corrupt the images with noise, and then find a filter that produces an image that is close to the original image.

We also tried compared using 4 copies of each image.

We apply a mask to the fitness function.

The mask is another image consisting of pixels of either 0 or 1.

When we find the difference between two images, we then multiply this by the mask.

Fitness function with mask

22.4

Fitness

Number of pixels

Difference image

Edge mask

Results

<table>
<thead>
<tr>
<th>Noise Type</th>
<th>No. of test images</th>
<th>Average Fitness</th>
<th>Best Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>4</td>
<td>1.3</td>
<td>0.56</td>
</tr>
<tr>
<td>Random</td>
<td>16</td>
<td>1.06</td>
<td>0.53</td>
</tr>
<tr>
<td>Salt and Pepper</td>
<td>4</td>
<td>0.76</td>
<td>0.39</td>
</tr>
<tr>
<td>Salt and Pepper</td>
<td>16</td>
<td>1.04</td>
<td>0.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Noise Type</th>
<th>Median filter fitness</th>
<th>Unfiltered image fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>3.56</td>
<td>3.77</td>
</tr>
<tr>
<td>Salt and Pepper</td>
<td>3.62</td>
<td>6.30</td>
</tr>
</tbody>
</table>
Results

<table>
<thead>
<tr>
<th>Noise Type</th>
<th>No. of test images</th>
<th>Avg. evals.</th>
<th>Min. evals.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>4</td>
<td>42539</td>
<td>14712</td>
</tr>
<tr>
<td>Random</td>
<td>16</td>
<td>43296</td>
<td>1812</td>
</tr>
<tr>
<td>Salt and Pepper</td>
<td>4</td>
<td>42851</td>
<td>8732</td>
</tr>
<tr>
<td>Salt and Pepper</td>
<td>16</td>
<td>39795</td>
<td>59403</td>
</tr>
</tbody>
</table>

Removing random noise

Removing salt and pepper noise

GPU Speed Up

- MS 'broke' the CPU side version of Accelerator in their last update!
- But, you can switch it to use the reference driver (CPU based).
- This has unknown overheads and is unclear what the performance penalty may be.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Image Size</th>
<th>Million GP Op/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>512x512</td>
<td>300</td>
</tr>
<tr>
<td>GPU</td>
<td>1024x1024</td>
<td>620</td>
</tr>
<tr>
<td>CPU</td>
<td>1024x1024</td>
<td>1.82</td>
</tr>
</tbody>
</table>

Current and Future Work

- nVidia GeForce 7300, Intel 6400, Windows XP
**Reverse engineering filters**

- We can use the same approach to reverse engineer an existing filter.
- i.e. We take an image, apply a filter in an image manipulation package (Photoshop, GIMP) and then evolve a program that can reproduce the effect.

**Evolution of a filter**

- 16 images
- 256 x 256 pixels per image

**Examples of evolved filters**

- Sobel
- Erode
Examples of evolved filters

- Emboss

GPU Speed

- We measured the Millions of Genetic Programming Operations Per Second (Nvidia 8800)

<table>
<thead>
<tr>
<th>Filter</th>
<th>Peak</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>dilation</td>
<td>111</td>
<td>113</td>
</tr>
<tr>
<td>emboss</td>
<td>104</td>
<td>129</td>
</tr>
<tr>
<td>erode</td>
<td>120</td>
<td>75</td>
</tr>
<tr>
<td>erodea</td>
<td>327</td>
<td>99</td>
</tr>
<tr>
<td>motion</td>
<td>280</td>
<td>277</td>
</tr>
<tr>
<td>neon</td>
<td>166</td>
<td>185</td>
</tr>
<tr>
<td>sobel</td>
<td>293</td>
<td>194</td>
</tr>
<tr>
<td>sobela</td>
<td>280</td>
<td>166</td>
</tr>
<tr>
<td>unsharp</td>
<td>324</td>
<td>139</td>
</tr>
</tbody>
</table>

GPU Speed

- Using the CPU bound reference driver, we found we could achieve an average of 1.2 million GPOpks.
- The CPU appears to be 100 times slower.
- The GPU has 128 processors....

Conclusion

- GPUs are effective platforms to use for image processing operations. We can exploit this computational power when researching GP applications involving image sets.
- We can feasibly tackle much harder (more interesting?) problems.
- We can also increase the reliability of our algorithms by greater testing during evolution.

Conclusion

- We have inconsistent results in timing Operations Per Second.
  - The reason is unclear. Perhaps it is due to the MS API being "efficient"?
- We do however see big increase in performance over running on the CPU reference driver
  - we have already measured true CPU performance on other problems.

Thank you!

- www.gpgpgpu.com
GPU Programming
Thoughts & opinions on the practical aspects

Timeline

- MS Accelerator
  - Early 2007
- RapidMind
  - Mid 2007...
- CUDA
  - Early 2008
- HLSL
  - Two weeks ago

API Hell

- Numerous APIs exist
  - Each with their own benefits and drawbacks.
- It is still unclear which is the best approach

Documentation

High Level Programming

- C or C++
  - RapidMind
  - CUDA
  - Cg, Etc
- Java
  - Via the JNI to the C ones?
- .net
  - Accelerator
  - XNA/HLSL
  - RapidMind fails to link due to CRT issues
**Debugging**

- GPGPU development tools are immature.
- Debugger support is weak
  - Unable to inspect memory on GPU during execution
  - Unable to step through shader programs
  - (Conditional) Breakpoints

**Future**

- Will my favourite API still be useful next month? Will I be forced to change platform when a new GPU technology comes out (e.g. Multi-GPU)?
- Several APIs have already died
  - Sh, PyGPU
- Several are ‘underground’
  - MS Accelerator, Peakstream