Accelerating Evolutionary Computation through Graphics Processing Units

GECCO-2009 Tutorial

Montreal
July 2009

Wolfgang Banzhaf and Simon Harding
Computer Science
Memorial University of Newfoundland

Acknowledgement

- Joint Work with Simon Harding, Bill Langdon and Garnett Wilson
- Software from RapidMind Inc., Microsoft
- Hardware from NVidia

"Today, a new Renaissance in 3D graphics is under way, driven by fully programmable GPUs -- *graphics processing units* -- that deliver thousands of times the graphics power available just ten years ago. Combining incredible parallel computing power with modern, high-level programming languages, today's GPUs have unleashed a Cambrian Explosion of innovation and creativity. ... But the most important gain of programmability is that you can do *anything* with a GPU so long as you can find an algorithm to express your idea."

Foreword to GPU Gems 2
Tim Sweeney, Epic Games
Addison Wesley, 2005

Outline

- Part I - General Method
  - Genetic Programming Resource Demands
  - Sources of Speed-up
  - The GPU as a new platform for Genetic Programming
  - Methods for implementation
- Part II - Case Studies
  - Data Parallel Implementation
  - Population Parallel Implementation
- Summary

Genetic Programming Resource Demands

GP Resource Demands

- GP is notoriously resource consuming
  - CPU cycles
  - Memory
- Standard GP system, 1μs per node
  - Binary trees, depth 17: 131 ms per tree
  - Fitness cases: 1,000 Population size: 1,000 Generations: 1,000 Number of runs: 100 ⇒Runtime: 10 Gs ≈ 317 years
- Standard GP system, 1ns per node
  - ⇒ Runtime: 116 days
- Limits to what we can approach with GP

Sources of Speed-up

- Easy
  - Multiple fitness cases
  - Population
  - Statistically independent runs
- Hard
  - Multiple generations
  - Program (node) sequence
  - Evaluation of single nodes

Sources of Speed-up

- Single node evaluations
- Program/node sequence
- Multiple generations
- Independent runs
- Populations
- Fitness cases
- Proxy evaluations
- Random execution

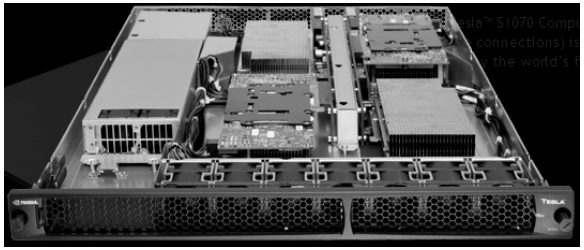
Sources of Speed-up

- Fast machines
- Vector Processors
- Parallel Machines (MIMD/SIMD)
- Clusters
- Loose Networks
- Multi-core
- Graphics Processing Units

A high-powered GPU: NVIDIA GEFORCE 8800 GTX

- 128 Stream Processors
- Max 520 GFlops at 575 / 1350 MHz clock
- Memory: 768 MB (128 x 6)
- Memory interface 384 bit
- Memory clock 900 MHz
- Max Memory bandwidth 86.4 GB per sec
- Power consumption 200-400 W

Current best GPU Setup



•4 GPUs , a total of 960 cores
•16Gb of memory
•Single Precision floating point performance (peak) : 3.73 to 4.14 Tflops
•Double Precision floating point performance (peak) : 311 to 345 GFlops

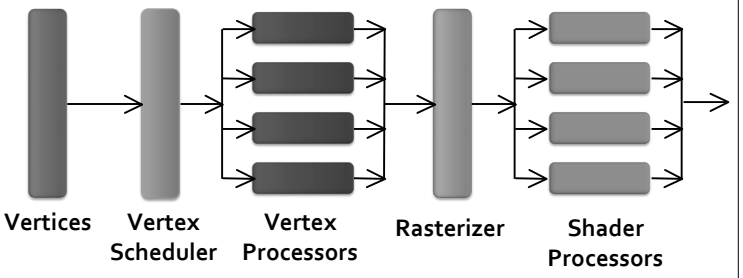
Future

- GPUs will continue to get more powerful
- Ability to add more cards to the system will increase
- Number of GPUs per board is also likely to increase
- Integration of the GPU into the CPU/motherboard?

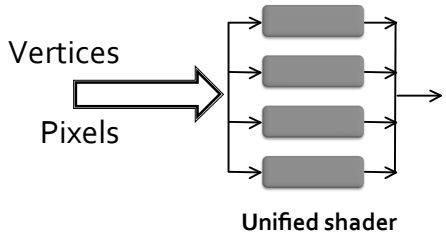
Graphics Processing Units as new Platforms for Genetic Programming

- Graphics cards are stream processors, highly specialized for graphics manipulation operations
- SIMD Parallelization
- Support for vector and matrix computations
- Rendering of textures, including shading
- Consist of programmable vertex and shading units

GPU Architecture



Unified Shader



General Purpose Computation on GPUs

- GPUs are not just for graphics operations
- High degree of programmability
- Fast floating point operations
- Useful for many numeric calculations
- Examples
  - Physical simulations (e.g. fluids and gases)
  - Protein Folding
  - Image Processing

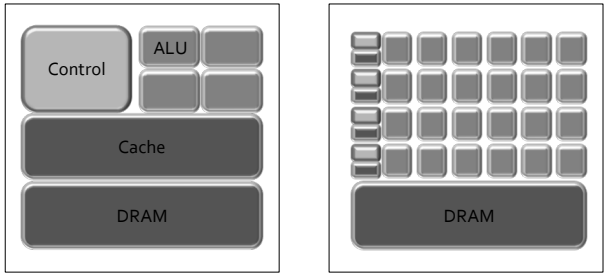
Nomenclature

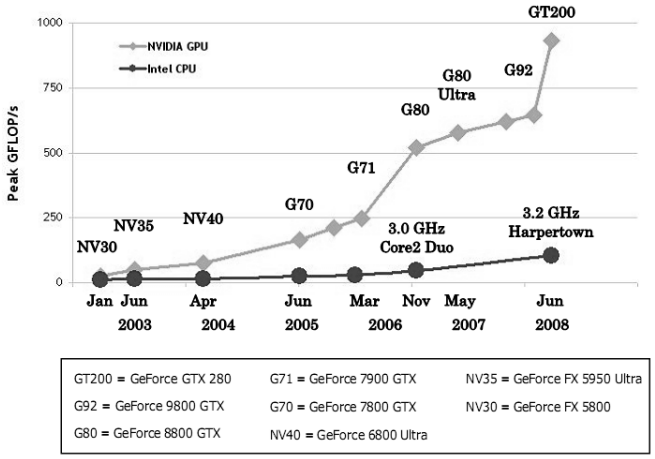
- Vertex
  - ⋆ An Object representing a point in a mesh
- Fragment
  - ⋆ A pixel in an image
- Shader
  - ⋆ Small program in the GPU pipeline

Why is this Architecture advantageous?

- Simplified pipeline architecture
- Highly parallel
- Memory is near the processor (reduces cache needs)
- Allows for scaling by hiding its true parallelism, i.e. scaling can increase number of processors
- Number of GPUs can be larger than 1

Why this Architecture is faster





Source: **Physics Simulation on NVIDIA GPUs,** Simon Green, Mark Harris, http://developer.nvidia.com/object/havok-fx-gdc-2006.html

Getting faster ...

- GPU development is driven by Gaming industry
- Speed increases faster than CPU speed increase (12 months vs 18 months doubling time)
- Provides an exponential advantage
- GPUs are inexpensive compared to other stream/parallel processors

Costs

Date	Cost Per GFLOP	Platform
1997	$30,000	32 X 16-Pentium-Pro processors
2000	$640	KLAT2 – 64 X 700Mhz AMD Athlon processors
2003	$82	KAYS0 – 128 X 2.6Ghz AMD Athlon processors
2005	$2.60	Xbox 360 (CPU only)
2006	$1	ATI X1900 GPU
2007.3	$0.42	Ambric AM2045 Processor
2007.10	$0.20	SONY PS3 GPU

Source: http://en.wikipedia.org/wiki/FLOPS

APIs

- There are a number of toolkits available for programming GPUs.
- So far, researchers in GP have not converged on one platform.
- Our case studies will illustrate several common APIs.

APIs

- CUDA
- MS Accelerator
- RapidMind
- Shader programming

CUDA

- http://www.nvidia.com/object/cuda_home.html

CUDA

ADVANTAGES

- Cross platform
- Programs are low(er) level
- High degree of flexibility
- Free
- Widely used
- Windows/Linux/Mac

DISADVANTAGES

- Works only on Nvidia hardware
- Programs are low(er) level
- Written in C

MS Accelerator

- http://research.microsoft.com/act/

MS Accelerator

ADVANTAGES

- Simple to use
- Highly abstracted
- Lazy evaluator

DISADVANTAGES

- Windows only
- May be too highly abstracted
- Only supports vectorised operations.

RapidMind

- http://www.rapidmind.net/

RapidMind

ADVANTAGES

- Cross platform:
  - GPU
  - Cell
  - CPU
- Multiple OS
- Easier C programming model than CUDA

DISADVANTAGES

- Not free
  - Academic license status is unclear.
  - It was 'free' ... But no longer

Shader Languages

ADVANTAGES

- Well documented, standard programming models.
- Cross hardware and software platform.

DISADVANTAGES

- Not really for GPGPU programming.

But more to come!

- OpenCL

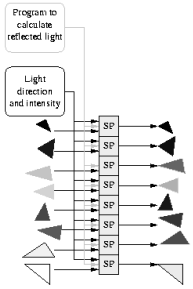
  http://khronos.org/opencl/
- Specification just released

  (8thDecember 2008)

Methods of Implementation I: The Shaders

SIMD on Shader Processors

- GPU designed for graphics
  - 32 bit floating point (2^{-23}) precision
  - Arrays of max 4 million elements
- Same operation done on many objects
  - Eg appearance of many triangles, different shapes, orientations, distances, surfaces
  - One program, many data → Simple (fast) parallel data streams
  - GPU does not allow random write access to large arrays (stack depth)



Power Consumption

- Another major cost factor is power consumption.
- In large deployments, energy costs are a major running cost.
- Not just for the raw processing, but for the air conditioning etc required for cooling

Programming a GPU

- General purpose computation is performed by implementing custom shader programs
- Lists of instructions to perform rendering operations
- GPUs have several types of shaders: Pixel, Vertex, Geometry

Shader Programs

- Pixel shaders as computational resource
- Textures are modified during computation
- Textures can be interpreted as any type of data
- Resulting images might be ignored or used for visualization
- Programming with OpenGL (Shader Language) or DirectX High-Level Shader Language
- APIs: Sh/RapidMind, Brook, Accelerator

Typical functions available

MATHEMATICAL

- Abs, add, ceiling, cos, divide, floor, log2, multiply, multiply add, pow, reciprocal, sqrt, rsqrt, subtract

BINARY

- And, Or, Not

LOGICAL

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

VECTOR

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

Caveats

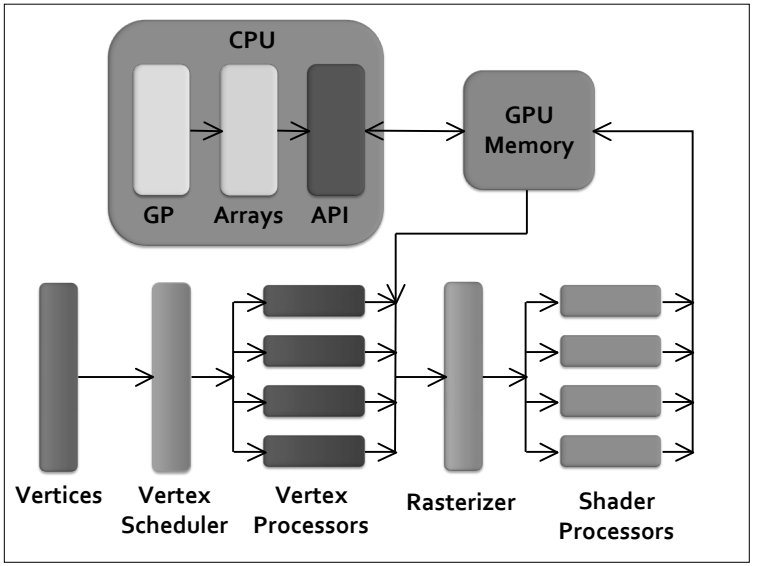
- Overhead of getting data onto the card
- Not hardware independent
- Length of shader programs might be limited
- Size of textures
- Mapping data into textures

Making GP go faster

- GPU shader processors are SIMD
- Fitness testing in Genetic Programming is also often SIMD
- Enables fitness case evaluation on shaders in parallel, scalable
- Added bonus: High processing speed for numeric computation
- Since 2007, papers on GP^{3}U

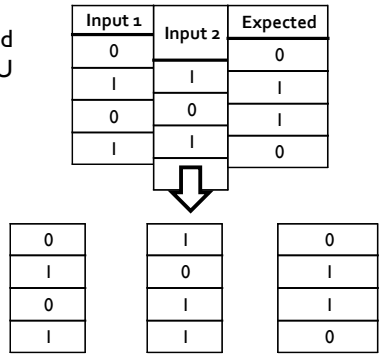
GP implementation

- The evolutionary algorithm and the population reside on the CPU
- Fitness evaluation is performed on the GPU
- Overheads:
  - Moving data to and from the GPU
  - Compiling genomes into shader programs

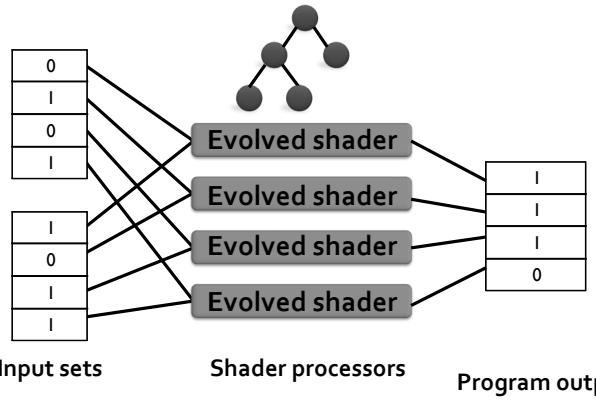


Fitness evaluation

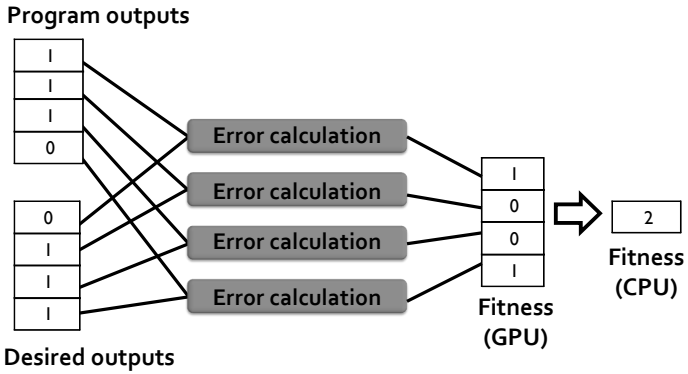
- Convert input table and expected output to GPU textures
- Split by 'column'
- Upload to GPU



Fitness evaluation (II)



Fitness evaluation (III)

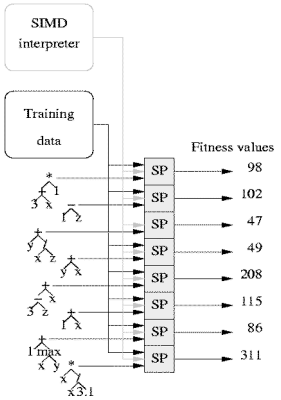


Other Implementations

- So far: Mapping of fitness cases onto GPU works well with a large number of fitness cases
- Use of vertex processors (not discussed)
- Next: Population-level parallelism

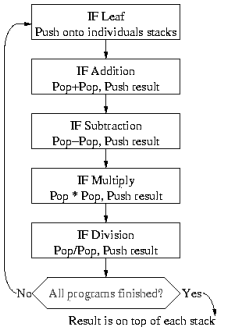
Interpreting many programs simultaneously

- Previous method used CPU to compile individuals to GPU code. Then one program run on multiple data (fitness cases)
- Now we are going to distribute the population among GPU PEs
- Trick: GP trees are considered data, running on a single interpreter on GPU (SIMD) for many trees
- Avoids compilation by interpreting GP trees

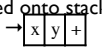


The GPU Genetic Programming Interpreter

- GP individuals wait for the interpreter to offer an instruction they need evaluating
- For example: Addition of numbers
  - When the interpreter wants to do an addition, everyone in the whole population who is waiting for addition is evaluated
  - The operation is ignored by everyone else
  - They then individually wait for their next instruction
- The interpreter moves on to its next operation
- The interpreter runs round its loop until the whole population has been interpreted (or is timed out)



Representing the Population

- Data is pushed onto stack before operations pop them (i.e. reverse polish notation x+y → 

)
- The tree is stored as linear expression in reverse polish notation
- Same structure on CPU as on GPU
- Genetic operations act on reverse polish notation:
  - random tree generation (eg ramped-half-and-half)
  - subtree crossover
  - 4 types of mutation
- Requires only one byte per leaf or function
  - Large populations (millions of individuals) are possible

Cost

- Interpreters avoid compilation but execution is slow
- SIMD has two main sources of waste
  - Synchronisation means short programs in the population take as long to execute as long programs
  - Most operations (80%) are not wanted and their results are thrown away
- Leafs access data and so are much more expensive than functions
  - A multiplication takes only 4 clock cycles = 3ns
  - Main memory read takes up to 300 clock cycles
  - Unfortunately, appr. 50% of trees are leafs
  - So cost is dominated by leafs, not function evaluation
- We accept other interpreter overheads (eg Lisp, Perl, Python, PHP), so why not SIMD overhead

GP Applications

1. Acceleration of fitness evaluation of an individual through distribution of fitness cases
2. Acceleration of population evaluation through distribution of individuals
3. Acceleration of single fitness case evaluation through distribution over GPU

Acceleration of Individual Fitness Evaluation

- Implementation used: Cartesian GP
- Benchmark: Random expressions of a given function size
- 'Typical' GP Runs: Regression and Classification

Suggested problems

- Image processing
- Systems involving physical simulation
  - Run the physics on the GPU
- Large developmental or multi-agent systems
- Revisit previously intractable problems

Accelerating Population Evaluation

- 5,000 to 5,242,000 individuals in a population
- Neighborhood structure of mutation and crossover operations
- Single Interpreter run on Multiple program trees (data)

Examples

- Approximating Pi
- Chaotic Time Series Prediction
- Mega population for Bioinformatics protein classification
- Predicting Breast Cancer fatalities
- Predicting problems with DNA GeneChips

Accelerating Single Fitness Case Evaluation

- Image processing tasks provide formidable problems for single fitness case evaluation
- Here, evolving programs have to manipulate entire images, with 256 x 256 or larger image size
- Here the GPU is used to accelerate image calculation

Questions?

- Our website: www.gpgpgpu.com run from beautiful Newfoundland

PART II

GP on GPUs - Case Studies

- Darren M. Chitty

A data parallel approach to genetic programming using programmable graphics hardware

Approach

- Cg

  http://developer.nvidia.com/page/cg_main.html

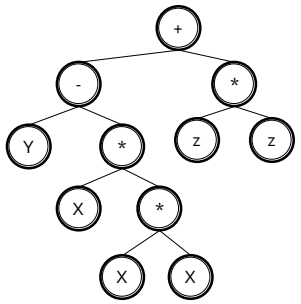
- Low end GPU

  Nvidia Geforce 6400 Go

Approach

1. Initialise OpenGL and setup graphics window
2. Copy data inputs into separate textures
3. Create a cgContext object
4. Generate the initial population
5. For each individual:
   - Convert chromosome to Cg source code using recursive functions
   - Compile and load program using Cg Toolkit
   - Bind program and textures to the cgContext object
   - Render the Cg program on the data
   - Get output from the GPU
   - Compare result with desired output and assign a fitness to the individual
6. For each generation:
   - Generate a new population using crossover and mutation
   - Evaluate population by performing step 5
7. Cleanup and output best result

Example of Cg individual



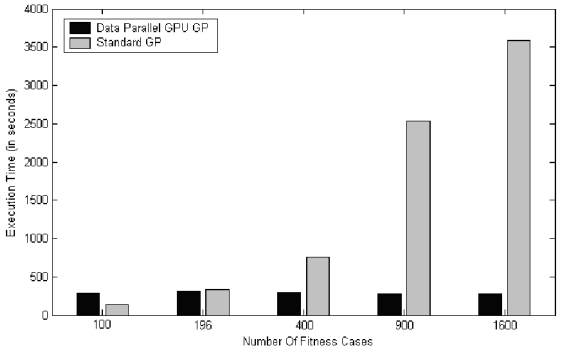

```
float4 FragmentProgram (in float2 coords : TEXCOORD0,
    uniform samplerRECT InputTextureX,
    uniform samplerRECT InputTextureY,
    uniform samplerRECT InputTextureZ)  : COLOR
{
    float4 X = texRECT (InputTextureX, coords);
    float4 Y = texRECT (InputTextureY, coords);
    float4 Z = texRECT (InputTextureZ, coords);

    float4 V1 = X;
    float4 V2 = X;
    float4 V3 = V1*V2;
    float4 V4 = X;
    float4 V5 = V3*V4;
    float4 V6 = Y;
    float4 V7 = V5-V6;
    float4 V8 = Z;
    float4 V9 = Z;
    float4 V10 = V8*V9;
    float4 V11 = V10+V7;

    return(V11);
}
```

Results

Execution times in seconds:

Problem	Standard GP	GPU GP
$x^4 + x^3 + x^2 + x$ (100 cases)	122	300
$2.76x^2 + 3.14x$ (400 cases)	2531	277
Iris Classification (150 cases)	226	252
11-way Multiplexer (2048 cases)	9115	304

Results II



- W. B. Langdon and W. Banzhaf

A SIMD Interpreter for Genetic Programming on Graphics Cards

RapidMind

- Grew out of Sh meta-programming (Waterloo)

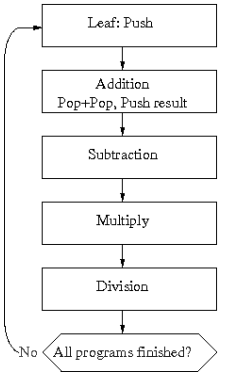
  Not source compatible with Sh but very similar concepts
- High level, C++ very heavy use of templates
- Compatible with free GNU C++ (GCC)
- Templates/GDB on occasion produce huge incomprehensible error messages leading to a difficult learning path
- Very active, new releases, targeting new hardware. Suggests RapidMind will be a viable option in the future as well as now
- Integrated compiler for GPU works almost without problem

The Approach

- Execute the entire population in parallel
- Implemented an interpreter on the GPU

The Interpreter

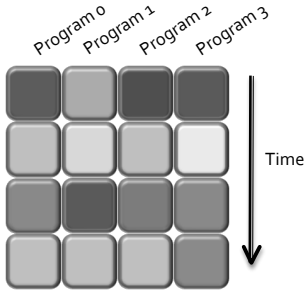
- The interpreter program is a number of IF statements
- Texture is interpreted as instructions
- Remember SIMD
- Each processor only does something when the right instruction comes past



Representation

- Representation is Linear expression in Reverse Polish i.e. Tree based GP
- But programs represented as textures (linearised)
- Short programs padded with NOPs

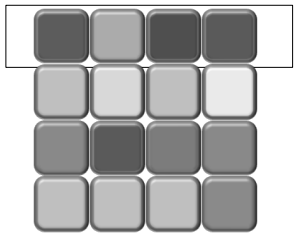
The Interpreter



•Each program runs as a different thread

•Here there are 4 threads (i.e. 4 shader programs)

The Interpreter

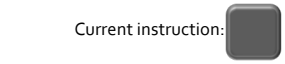


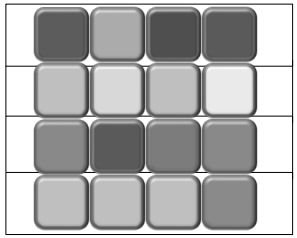
FOR Each Row
    FOR Each instruction in function set
        IF Instruction at this row is the current instruction type
            THEN perform instruction

The Interpreter

Current instruction:

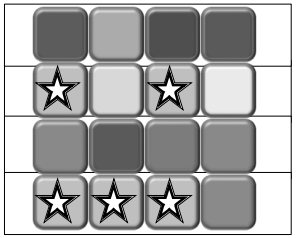
The Interpreter

Current instruction: 



The Interpreter

Current instruction: 



The Interpreter

- Major problem:
  - Some threads are doing nothing as they are not currently acting on their instruction type
- However:
  - No compiler overhead
  - Single, simple shader program
  - Supports VERY large populations

Evolutionary Algorithm

- Spatial Algorithm
- Implemented on GPU

Demo Approximating Pi

- Spatial population 512x400 204800 individual programs
- Two randomly chosen constants and + - * /
- Programs of up to 8 leafs and 7 functions (total 15).
- Binary tournament selection
- Subtree crossover and subtree mutation.
- Both selection and crossover act between neighbours.
- GPU takes about 0.5 per generation.

Protein Prediction - using a Mega Population

- 1213 proteins used for training
  - Randomly selected 200 at each fitness evaluation
- Spatial population 1024x1024 = 1,048,576 individual programs
- 1000 generation
- 6 hours 46 mins

Speed

Experiment	Population	Mean program size	Program runs	Speed
Laser 9-8	5000	49.6	376640	190
Protein	106	56.9	200	595
Laser	18225	55.4	151360	656
Mackey-Glass	204800	11	1200	895
Mackey-Glass	204800	13	1200	1056

Speed is in Millions of GP operations per second

CPU Speed:
• Mackey-Glass : 2.2Ghz CPU only 71M GPop/Second

Data Parallel GP using RapidMind

```
#define RMTYPE Value4f

Array<1,RMTYPE>* Input0;   //The first input i.e. i
Array<1,RMTYPE>* Input1;   //The second input i.e. j
Array<1,RMTYPE>* ExpectedOutput;  //The expected output i.e. what the function should be

 GPU_ADD = RM_BEGIN {
     In<RMTYPE> a;  // first input
     In<RMTYPE> b;  // second input
     Out<RMTYPE> c; // output

     c = a + b; // operation on the data
 } RM_END;
```

GP with RapidMind

```
//Construct a suitable GraphRunner
GPUCGPGraphRunner<Array<1,RMTYPE>>* Runner = new GPUCGPGraphRunner<Array<1,RMTYPE>>(
    ind->Graph, //The graph
    2,//Inputs
    1); //Outputs

//Apply the inputs
Runner->Inputs[0] = *Input0;
Runner->Inputs[1] = *Input1;

//Work out which nodes in the graph we're going to execute
Runner->FindNodesToEvaluate();

//Then execute these nodes
Runner->Iterate();

//The output is store in the last entry of the ResultsCache
//We take this output,and compare it to the expected value
Array<1,RMTYPE> Difference = GPU_DIFF(Runner->ResultsCache[Runner->AssociatedGraph->Length()-1],
    *this->ExpectedOutput);

//Calculate the sum of the differences i.e. the total error
Value4f result = sum(Difference);

//Set the fitness of the individual
Candidate->SetFitness(result.get_value(0)+result.get_value(1)+result.get_value(2));
```

GP with RapidMind II

Inside the iterate function:

```
switch(Node->Function)
{
    case(ADD):
        this->InstructionsCount+=CGPNode_NumberOfInputs-1;
        OutputValues[0] = GPU_ADD(InputValues[0], InputValues[1]);
        #ifdef DEBUGTRACE
            cout << " ADD ";
        #endif
        break;

    case(SUB):
        this->InstructionsCount+=CGPNode_NumberOfInputs-1;
        OutputValues[0] = GPU_SUB(InputValues[0], InputValues[1]);
        #ifdef DEBUGTRACE
            cout << " SUB ";
        #endif
        break;
```

Examples Results

Experiment	Number of Terminals	|F|	Population	Program size	Stack depth	Test cases	Speed (million OPs/sec)
Mackey-Glass	8+128	4	204 800	11.0	4	1200	895
Mackey-Glass	8+128	4	204 800	13.0	4	1200	1056
Protein	20+128	4	1 048 576	56.9	8	200	504
$Laser_a$	3+128	4	18 225	55.4	8	151 360	656
$Laser_b$	9+128	4	5 000	49.6	8	376 640	190
Cancer	1 013 888+1001	4	5 242 880	≤15.0	4	128	535
GeneChip	47+1001	6	16 384	≤ 63.0	8	½M, sample 200	314

Lessons learnt

- Interpreting GP trees on the GPU is dominated by leafs:
  - since there are lots of them and typically they require data transfers across the GPU
  - adding more functions will slow interpreter less than might have been expected
- Actual speed 0.2 - 1.0 billion GP ops /second
- Speed up 7 – 12 times faster than CPU

Fast Genetic Programming & Developmental Systems using GPUs

- Simon Harding and Wolfgang Banzhaf
- Memorial University, St John's, NL, Canada

MS Accelerator

- http://research.microsoft.com/act/

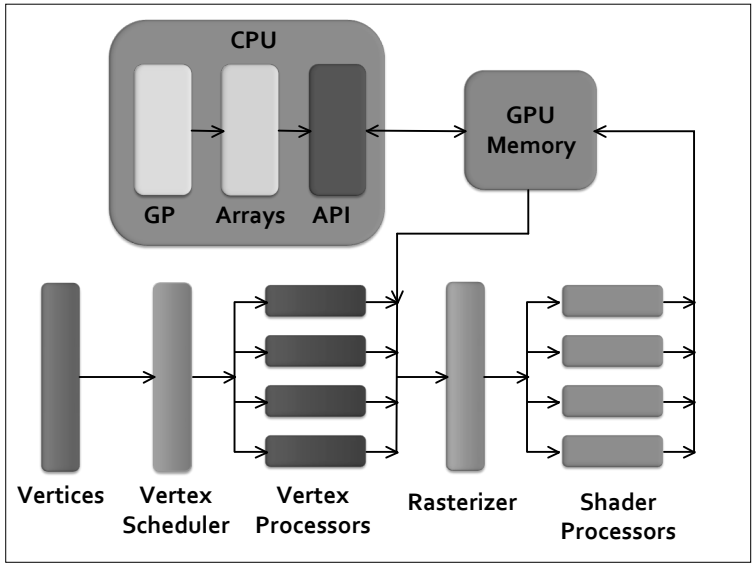
Microsoft Research

MS Accelerator - Example Code

```
ParallelArrays.InitGPU();

float[,] CPU_Array1 = new float[4096,4096];
float[,] CPU_Array2 = new float[4096, 4096];

//Populate CPU arrays here

FloatParallelArray GPU_Array1 = new DisposableFloatParallelArray(CPU_Array1);
FloatParallelArray GPU_Array2 = new DisposableFloatParallelArray(CPU_Array2);

FloatParallelArray GPU_Array3 = ParallelArrays.Add(GPU_Array2, GPU_Array2);

FloatParallelArray GPU_Array4 = ParallelArrays.Divide(0.1234f, GPU_Array3);

float[,] CPU_Result = new float[4096, 4096];

ParallelArrays.ToArray(GPU_Array4, out CPU_Result);

// Process CPU_Result array here

ParallelArrays.UnInit();
```

Available Functions

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

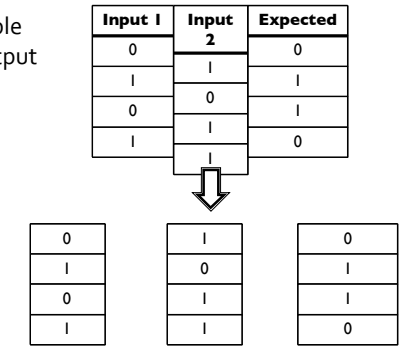


Overheads

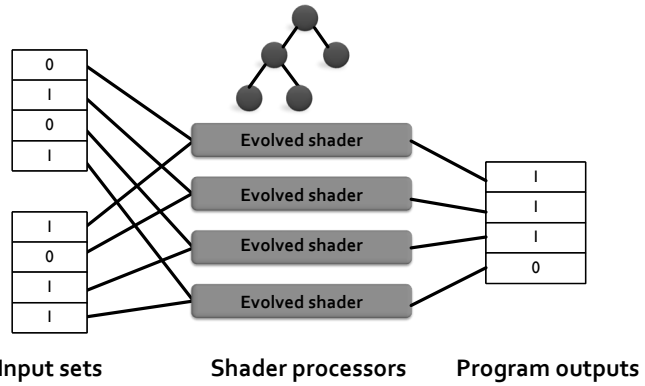
- Moving data to and from GPU
- Compiling the genome to a shader
- The results shown later in this talk give an indication where GPU use is appropriate.

Evaluation of GP

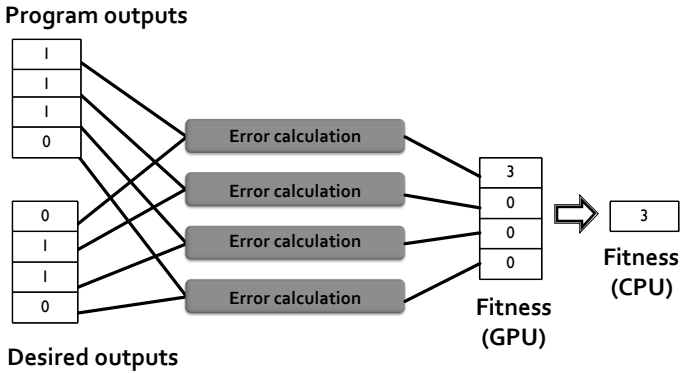
- Convert input table and expected output to GPU textures
- Split by 'column'
- Upload to GPU



Evaluation of GP



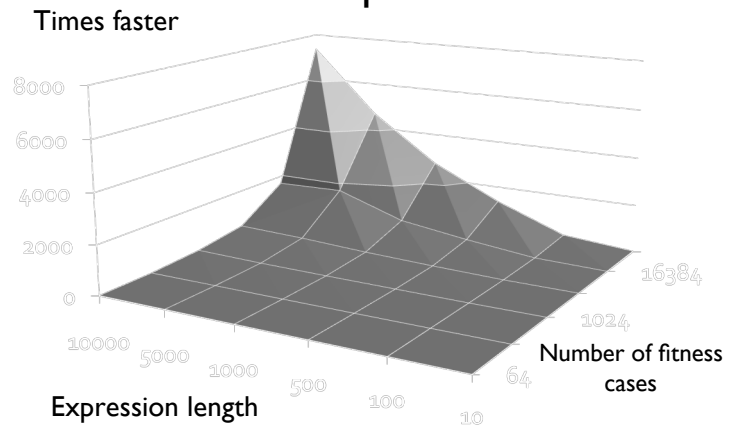
Evaluation of GP



Experiments

- Three types of experiment used for benchmarking
  - Random expressions of a given function size.
  - 'Typical' genetic programming runs.
  - GP in a developmental system.
- We used CGP
  - Expect similar results from other GP types.

Benchmark: Random Floating Point Expressions

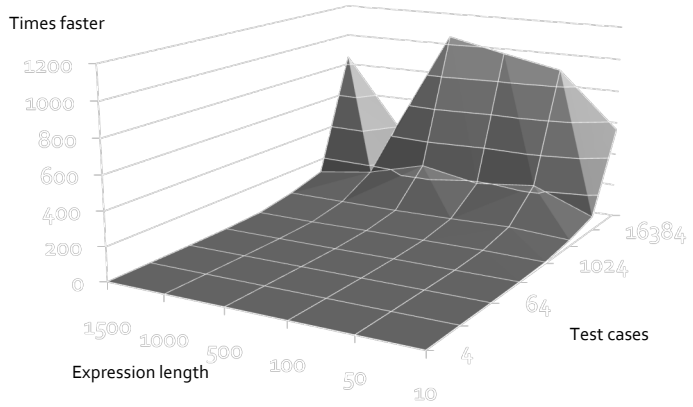


Benchmark: Random Floating Point Expressions

Fitness cases

Expression length	64	256	1024	4096	16384	65536
10	0.04	0.16	0.6	2.39	8.94	28
100	0.4	1.38	5.55	23.03	84.23	271
500	1.82	7.04	27.84	101.13	407	1349
1000	3.47	13.78	52	204.35	803	2694
5000	10.02	26.35	87	349.73	1736	4642
10000	13.01	36.5	157.	442	1678	7351

Benchmark: Binary

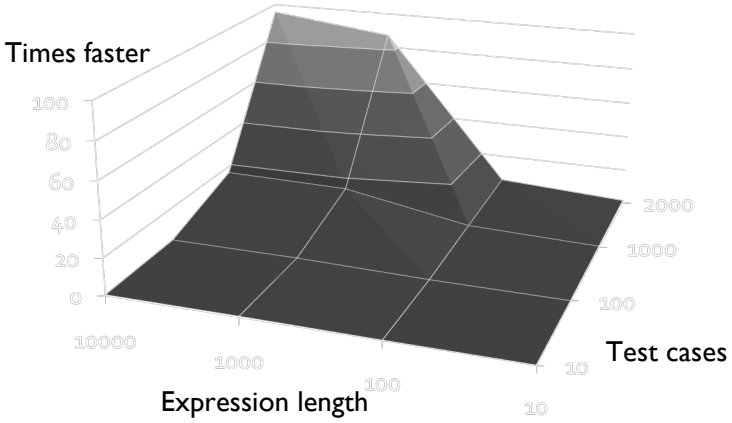


Benchmark: Binary II

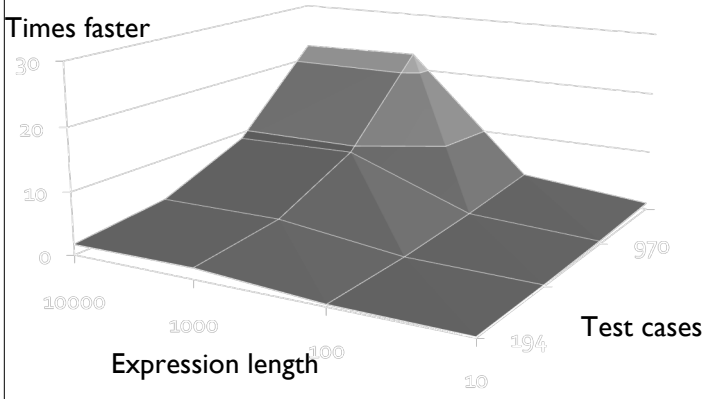
Test cases

Expression length	4	16	64	256	1024	4096	16384	65536
10	0.22	1.04	1.05	2.77	7.79	36.53	84.08	556
50	0.44	0.57	1.43	3.02	14.75	58.17	228	896
100	0.39	0.62	1.17	4.36	14.49	51.51	189	969
500	0.35	0.43	0.75	2.64	14.11	48.01	256	1048
1000	0.23	0.39	0.86	3.01	10.79	50.39	162	408
1500	0.4	0.55	1.15	4.19	13.69	539	113	848

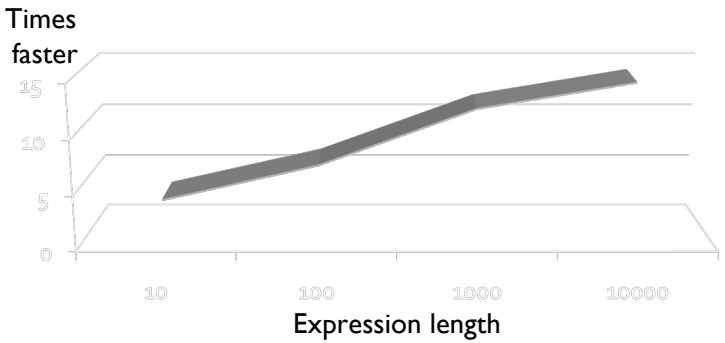
Regression



Classification: Spirals



Classification: Proteins



Developmental Systems

- Inspired by the properties of development and growth in biological systems.
  - Fault tolerance, recovery
  - Compression
  - Evolvability
- Concerned with cellular developmental systems
  - These are typically a cellular automata

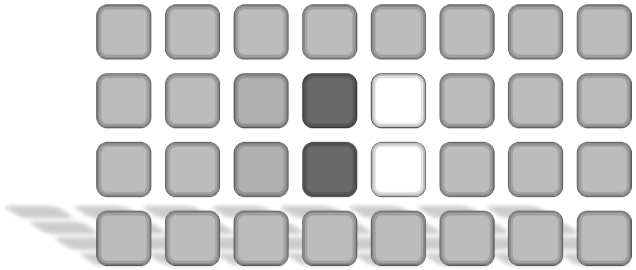
Developmental Systems II

- Concerned with cellular developmental systems
  - These are typically a cellular automata
- Update rules are typically evolved
  - Here we use CGP
- Often have simulated chemical environment
  - With additional CAs used to model diffusion of signaling chemicals

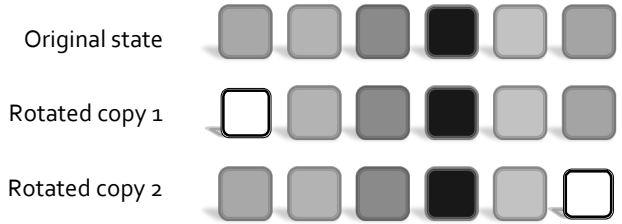
Developmental Systems III

- Each cell contains the same program
- With similar inputs:
  - Current state
  - Current state of neighbours
  - State of chemicals

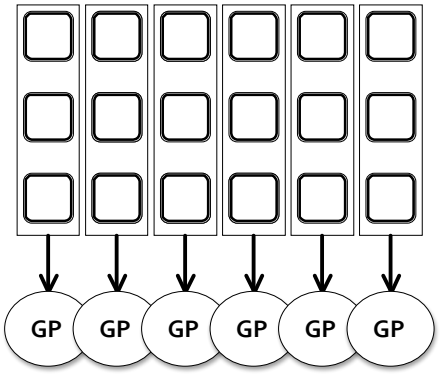
Developmental Systems IV



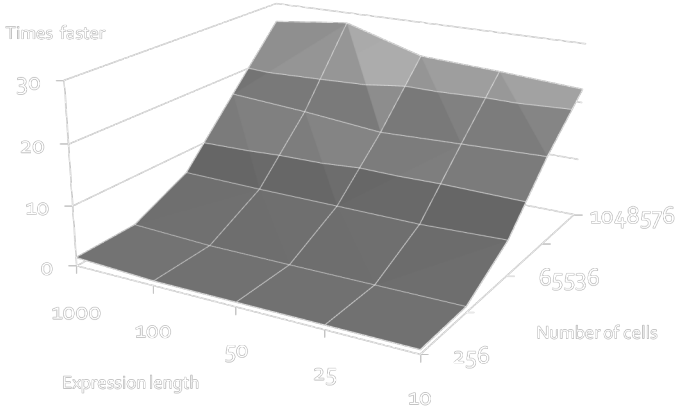
Finding the Neighbors



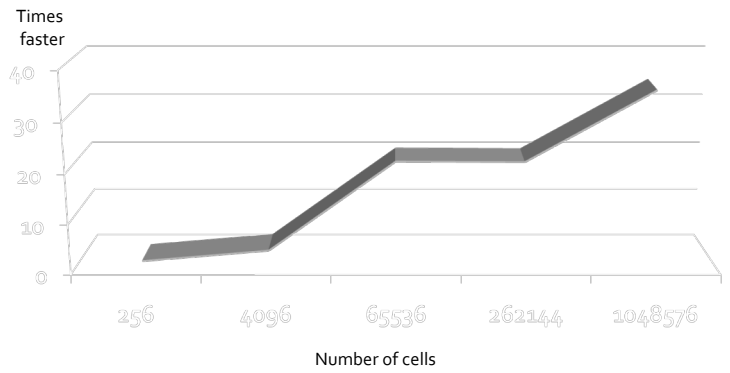
Finding the Neighbors II



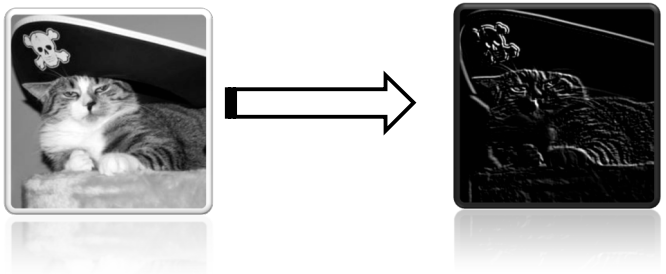
Developmental System



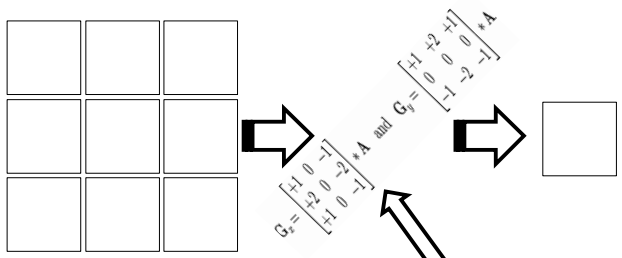
Diffusion simulation



Evolving Image Filters



Evolving Image Filters (II)



We can evolve this!

Evolving Image Filters

- Evolves a program/expression that will take a neighbourhood of pixels, apply a convolution and output a new value for the center pixel.
- 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.

Evolving Image Filters

- Very little research exists on evolving filters in software
- Reason: It takes too long on a CPU to test an individual
- Most research looks into evolving hardware (e.g. FPGA) implementations

Evolving Image Filters

- Applications restricted to single 256x256 pixel images
- But how do we know if this evolved program would work on other images, with different characteristics?

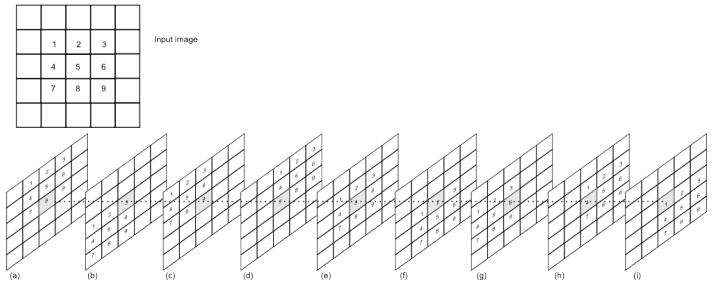
Evolving Image Filters

- Ideally, we need to train on a number of pictures
- Better still, validation should be done on another set of images
- There might be issues with overfitting & generalisation as elsewhere in machine learning

Evolving Image Filters on the GPU

- We need to map to the SIMD architecture
- Aim: One pixel per processor
- But how do we get the neighbourhoods of pixels?
  - Shifting!

Evolving Image Filters on the GPU



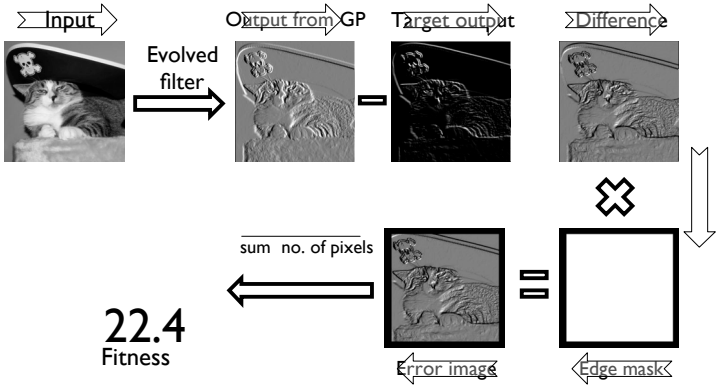
Multiple Images at the same time

- We present multiple images simultaneously
- In effect, we process one large image
- However, we must be careful of the edges where different images touch

Multiple Images at the same time

- 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



Evolving Image Filters on the GPU

- Using the parallelism of the GPU we can effectively evaluate all the pixels simultaneously
- It is substantially faster than a CPU based approach

Fitness Function

- Find the average error of each pixel
- Error = difference between the output of the evolved program and the target image
- Although computationally expensive, this can be processed in parallel on the GPU

Noise Removal

- Here we train on 4 different images to help prevent overfitting
- We corrupt the images with noise, and then find a filter that produces an image that is close to the original image

Removing Salt and Pepper Noise

Original image | Corrupted with 5% Salt and Pepper noise | Evolved filter | Conventional filter

Reverse Engineering Filters

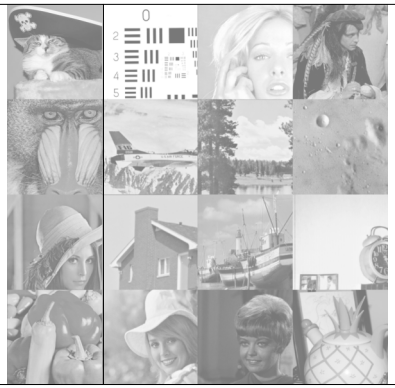
- The same approach can be used to reverse engineer an existing filter
- A filter from a an image manipulation package (Photoshop, GIMP) is applied, and a program is evolved that can reproduce the effect

Reverse Engineering Filters

- 16 images
- 256 x 256 pixels per image

Reverse Engineering Filters

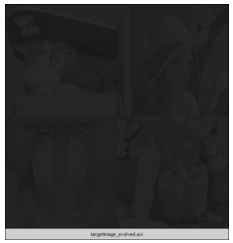
Training set



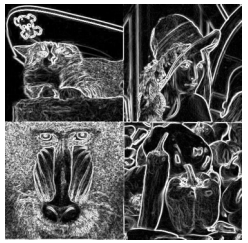
Validation set

Evolution of a Filter

Input



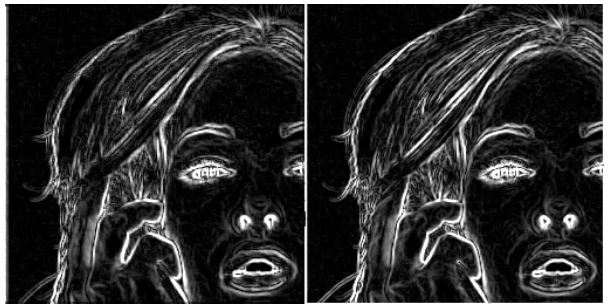
Best evolved individual



Target

Examples of Evolved Filters

Evolved filter Target filter



Examples of Evolved Filters (II)

Evolved filter Target filter

Examples of Evolved Filters (III)

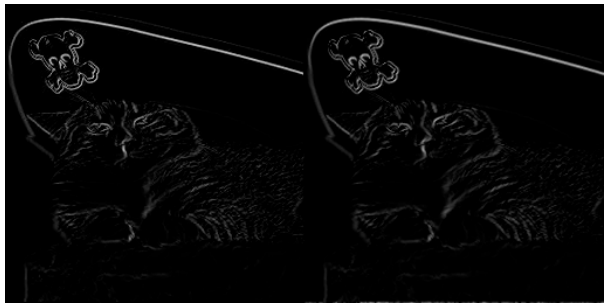
Evolved filter Target filter

Examples of Evolved Filters (IV)

Evolved filter Target filter

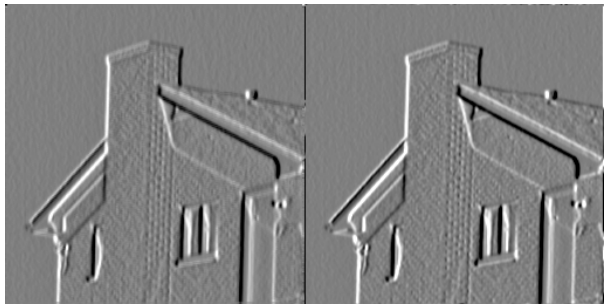
Examples of Evolved Filters (V)

Evolved filter Target filter



Examples of Evolved Filters (VI)

Evolved filter Target filter



GPU Speed

- Measured speed on filter evolution in Millions of Genetic Programming Operations Per Second

Filter	Peak	Average
dilate	116	62
dilate2	281	133
emboss	254	129
erode	230	79
erode2	307	195
motion	280	177
neon	266	185
sobel	292	194
sobel2	280	166
unsharp	324	139

GPU Speed

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

- Garnett Wilson and Wolfgang Banzhaf

Linear Genetic Programming GPGPU on Microsoft's Xbox 360

Why do GPGPU on Consoles?

- Current generation video game consoles have considerable GPU and CPU power, which can be harnessed for research.
- At launch, they are basically graphics supercomputers with cutting edge hardware.
- E.g. Xbox 360, launched on November, 2005, was the first PC (or console) to feature:
- CPU multi-processing with 3 cores
- GPU-unified shader architecture

Implementation

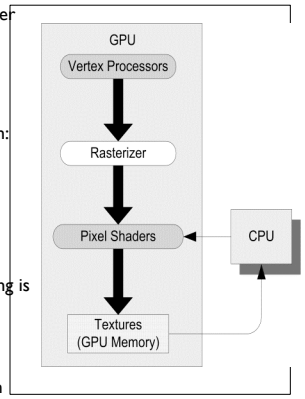
- First implementation of a research-based GP system on a commercial video game platform.
- First time linear GP (LGP) has been implemented in a GPGPU application.
- First instance of XBox 360 being used for any GPGPU purpose.

Xbox 360: Under the Hood

- Custom built IBM PowerPC-based CPU with three 3.2GHz core processors sharing a 1Mb L2 cache.
- CPU core also has an associated complement of SIMD vector processing units.
- CPU cache, cores, and vector units are customized for graphics-intensive computation, and the GPU is able to read directly from the CPU L2 cache.
- ATI GPU with 48 parallel shaders
- 512 MB of DRAM

GPGPU Summary

- GPGPU applications tend to use pixel shaders (rather than vertex shaders):
  - typically more pixel shaders
  - pixel shader output fed directly to memory
- In terms of traditional data structures and execution:
  - GPU textures are analogous to arrays.
  - the shader program is like a Kernel program.
  - rendering effectively executes the program.
  - CPU runs the main program, and sends data in texture form to the GPU when parallel processing is required.
  - GPU renders to a texture in its memory (rather than to the screen).
  - the output texture data is consumed by the main (CPU-side) program.



The XNA Framework

- In 2006, Microsoft launched XNA's Not Acronymed (recursive acronym "XNA") Game Studio Express 1.0
- Integrated with C# in Visual Studio variants.
- Game Studio 2.0 and 3 CTP have now been released.
- XNA allowed, for the first time, access to the GPU on a video game console.

Tools for Homebrew Development on the Xbox 360

The following are required for GPGPU on the Xbox 360:

- C# Studio Express (Game Studio Express 1.0 and Refresh) or Visual Studio 2005 product (Game Studio 2.0)
- XNA Game Studio (XNA Framework)
- nVidia's FX Composer (not absolutely required)
- Xbox 360 with hard drive and XNA Game Launcher installed.
- Membership in Creator's Club and internet access to Xbox Live.
- Windows PC with XP SP2 or Vista variant installed.
- To maximize texture representations, a graphics card capable of supporting at least Pixel Shader v. 3.0.
- LAN connection between PC and Xbox 360.

Design Considerations

- Microsoft is currently the only console vendor allowing access to GPUs.
- XNA's "content pipeline" does not permit dynamic loading or switching of shader programs to the GPU (so treating shader programs as individuals to be subject to operators is not possible).
- Hard drive I/O was not possible as of XNA 1.0 Refresh, so data must be output to screen. Means of input for the Xbox 360 include controller and USB keyboard.

Design Considerations II

- With XNA, GPU cannot implement scatter, thus:
  - Results must be rendered to a texture on an internal target buffer (rather than the screen).
  - Content is read back to the calling program from the internal target.
  - Array data stored on textures must be referenced using texture coordinates with an appropriate mapping.
- Xbox 360 GPU and Pixel Shader 3.0 have additional specifications (available by querying Xbox 360 with XNA GraphicsDevice class):
  - Shader program can consist of 2048 instructions.
  - Flow control of depth 4 (maximum of 4 instructions can be called within one another).
  - Supports 16 simultaneous textures.
  - Maximum texture height and width of 8192.

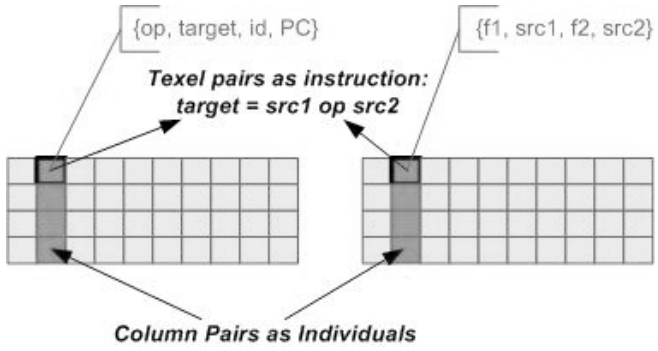
GP Individual Representation (Textures)

- Eight chromosomes in an instruction, each set of 4 placed on different texture.
- First Texture holds {*op*, *target*, *id*, *ptr*}.
- Second Texture holds {f_1, src_1, f_2, src_2}.
- Each instruction perform an operation on the contents of two sources (fitness case or register content), placing result in target:
- *target = src1 op src2*
  - op = [0, 3] corresponding to ADD, SUB, MUL, or DIV
  - src_1 and src_2 can specify either fitness cases or registers, and thus take values in [0, MAX(*classification features or regression inputs*, *registers*)]
  - *id*, *id* = [0, *population size*] labels the individual
  - *ptr*, *ptr* = [0, *instructions*] serves as a pointer to the current instruction
  - Boolean flags f_1, f_2, indicate whether to load from fitness cases or registers for src_1 and src_2, respectively.

GP Individual Textures

- XNA *HalfVector4* surface format was used, each chromosome (channel) was a 16 bit float.
- The two textures represent a whole population, with each individual being a column of texels, and each texel in the column being an instruction.
- Width of the textures (in texels) is the number of individuals .
- Height is the number of instructions in an individual.
- Current state of an individual's four registers (following an instruction) are kept in a **third** texture's texels (at the same coordinates) as 4 floats.

GP Individual Textures



GPU-side Shader Program: Mutation

For every channel (all 4) of each pixel of the instructions (2 textures)

- a "mask" texture, with channels containing values [0.0 … 1.0], is applied.
- If the mutation threshold is > mask texture amount for a particular channel
  - an appropriate replacement value for the channel is given by randomly generated replacement textures (2 textures corresponding to instruction textures)

GPU Mutation Code

```
if (threshold.x <= mutationThreshold)
    current.x = replacement.x
if (threshold.y <= mutationThreshold)
    current.y = replacement.y
if (threshold.z <= mutationThreshold)
    current.z = replacement.z
if (threshold.w <= mutationThreshold)
    current.w = replacement.w
return current
```

GPU-side Shader Program: Fitness

- There are considerations for running the fitness shader on the XBox 360 vs. nVidia GeForce 8800:
  - XBox microcode compiler issues with loops inside other loops relying on instructions of the outer loop.
  - Prevents looping over instructions within loop over fitness cases, for instance.
  - Fitness implemented CPU-side on XBox 360.
  - Note: Mutation can still be implemented no problem GPU side on XBox 360 with this restriction.

GPU Fitness Code (PC only)

```
for each fitness case f
re-initialize registers
for each instruction i
    // determine src1 and src2
    if (flag1 == 1)
        x1 = fitnessCase(src1, f)
    if (flag1 == 0)
        x1 = register[src1]
    if (flag2 == 1)
        x2 = fitnessCases(src2, f)
    if (flag2 == 0)
        x2 = register[src2]
    // determine operator, result in target register
    if (op == 0) register[target] = x1 + x2
    if (op == 1) register[target] = x1 - x2
    if (op == 2) register[target] = x1 * x2
    if (op == 3) register[target] = x1 / x2
    // determine raw fitness and hits
    fitness = abs(register[0] – fitnessCaset(result, f))
    if (fitness < 0.01)  hits++
    totalFitness = totalFitness + fitness
return totalFitness, hits
```

Thanks to Simon Harding

CPU-side GP Program

```
GPGame {
        GPGame()  //constructor
                provide seedings for each trial
        Initialize()
                prompt for user input using on-screen keyboard
                declare and populate HalfVector4[] data arrays for all textures
Update(GameTime)
                check for exit key pressed on control pad
                parse user keyboard input until completed
Draw(GameTime)  // evaluates fitness case over population
          // each pass evaluates an instruction over all individuals
          for passes in fitnessEffect
                    run Fitness.fx  HLSL program (see above)
                    resolve render target to texture, get array data from texture
          // do for each fitness case
          adjust all individual's fitnesses; fitCase++
          if at the end of a generation
                fitness-proportionate generational selection
                run Mutate.fx HLSL program (on two texture sets)
          if at the end of a trial
                    trial++;  round = 0;
                    add best fitness to growing List for output
        if all trials are not yet done
                display fitness, timer, and population texture output
}
```

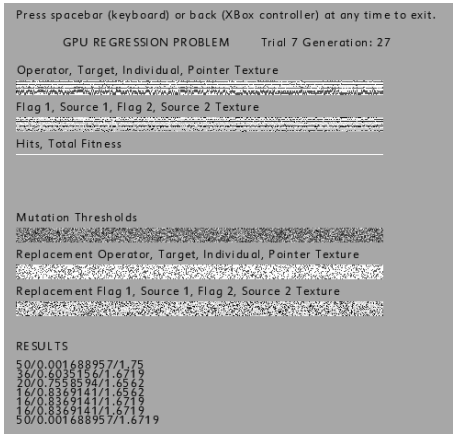
Input of Parameters with Xbox 360

Options:

•USB keyboard
•connected to
•XBox 360

•XBox 360 chatpad

XNA-based Linear GP GPGPU in Action



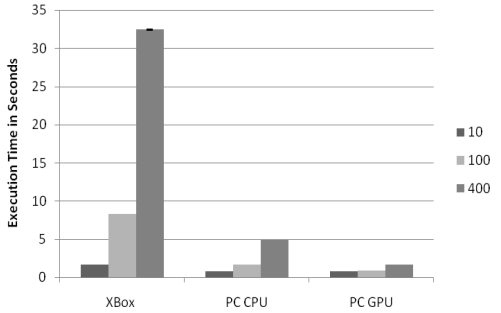
Experimental Set-up

- CPU-only version of the implementation was also created, implementing all shader functionality with appropriate C# code.
- The sextic polynomial $x^6 - 2x^4 + x^2$ introduced by Koza was implemented for regression, using float inputs in the range [0, 1] for 50 fitness cases.
- Windows PC specifications:
  - OS: Windows Vista Business PC
  - IDE: Visual C# 2005 Express with XNA Game Studio Express 1.0 (Refresh)
  - CPU: AMD Athlon 64 Processor 3500+ (2.21 GHz),
  - Memory: 1023 MB of RAM
  - Graphics Card: ASUS EN8800GTX video card with nVidia GeForce 8800 GTX GPU on board (using128 parallel stream processors with unified shader architecture)

Parameters

Function Set	ADD, SUB, MUL, DIV (on floats)
Fitness	fitness-proportionate roulette wheel
Population	10, 100, or 400 individuals
Mutation	threshold = 0.1
Tournament	generational, 50 rounds
Fitness Cases	Regression: 50 cases, x = [0, 1]
Fitness Metric	Regression: 50 hits, where a hit is Absolute(Reg[0] − y) <= 0.01

Results



Xbox 360, PC CPU, and PC GPU execution times with standard error, based on 10 trials of 50 generations for sextic polynomial. Fitness and mutation were implemented on the GPU for PC GPU, and mutation was implemented on the GPU for Xbox.

Interpretation of Results

- It should be noted that the Xbox 360 (released 2005) has to complete with a PC with twice as much RAM and housing a GPU with 128 processors for parallel processing as opposed to its 48 processors.
- For low population (10), the execution time in seconds is similar across all platforms.
- For population of 100
  - execution time does not noticeably increase for the GPU-dominated PC implementation
  - approximately doubles for the PC CPU implementation
  - increases over 5-fold for the Xbox 360.
- For the highest population considered, 400
  - there is little increase in PC GPU over population sizes due to the parallel processing of the GPU (doubling of time with 40-fold increase in population)
  - PC CPU sees a 5-fold increase with 40-fold increase in execution time
  - Xbox 360 has an increase in processing time of 20-fold for a 40-fold increase in population.
- As is common in genetic programming GPU literature, a performance benefit is evident by using the GPU for parallel processing.

Conclusions

- The main goal of this work was to show how to implement a GP system on a commercial video game console (Xbox 360) using GPGPU.
- First time (to our knowledge) that GPGPU, or genetic programming, has been implemented on a commercial video game console for research purposes.
- First instance of a Linear GP implementation using GPGPU.
- Xbox 360 appears to offer tightly coupled CPU and GPU graphics performance.

Source

Wilson, Garnett, and Banzhaf, Wolfgang, "Linear Genetic Programming GPGPU on Microsoft's Xbox 360," Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2008), June 1-6, 2008, Hong Kong, China, IEEE Press, ISBN 978-1-4244-1823-7, pp. 378-385.

Genetic Programming With CUDA

Programming with CUDA

- High number of threads.
- Threads are grouped into thread blocks.
- Thread blocks are grouped into a grid.
- Grids are executed on a device (i.e. GPU).

Thread Block

- Group of threads that can efficiently access some shared memory.
- Each thread has a unique *thread ID*
  - Blocks contain either a 2D or 3D array of threads
- Thread blocks can be synchronised.
- Blocks contain a limited number of threads

Thread Blocks

- ID number comes from the position in the array
- For 2d (x,y):
  - ID = x + y .Dx
- For 3d (x,y,z):
  - ID = x + y. Dx + z.Dx.Dy

(where Dx and Dy are the width and height of the array)

Grid of Thread Blocks

- Blocks have a limited number of threads.
- So, if we want to execute more threads than we are allowed we could either:
  - Run sets of threads sequentially and lose things such as synchronisation.
  - Run as a grid of thread blocks.
- The GPU will decide how best to execute the thread blocks.

Grid of Thread Blocks II

- Blocks have a *block ID*
- Blocks are arranged in a 2D grid (x,y)
  - Block ID = x + y. Dx
- More than one grid can be scheduled for execution on the device.

Example I

```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Kernel invocation
    vecAdd<<<1, N>>>(A, B, C);
}
```

Example II

```
__global__ void matAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Kernel invocation
    dim3 dimBlock(16, 16);
    dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
                 (N + dimBlock.y - 1) / dimBlock.y);
    matAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

- Denis Robilliard, Virginie Marion-Poty and Cyril Fonlupt

Population Parallel GP on the G80 GPU

The Approach

- They argue that evaluating the population in parallel is better.
  - Most GP problems have few test cases.
  - This results in under utilisation.
  - Running the population in parallel would be preferable.
- Can we get the benefit of both data parallel and population parallel?

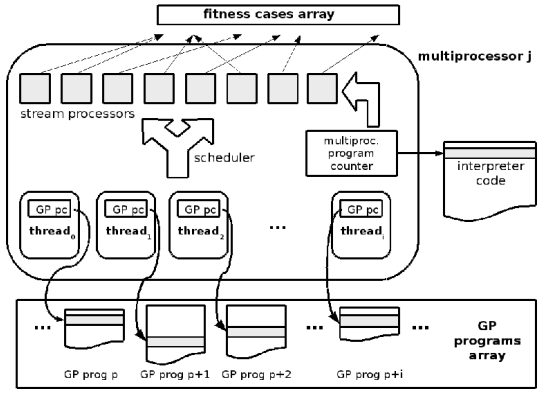
The Approach II

- SPMD
  - Single Program Multiple Data
- Some Nvidia processors (G80 etc) effectively have multiple processors – and therefore can run several copies of the same program at once.
  - Note the subtle difference to SIMD!
- 16 processors, 8 steam processors each

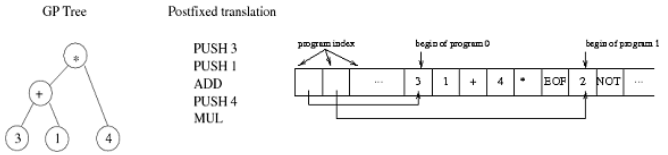
The Approach III

- To clarify:
  - A number of processors
  - They all run the same program
  - BUT they all have different program counters
- What this means:
  - Fitness cases can be divided up into blocks, and these blocks run in parallel

The Approach IV



Interpreter

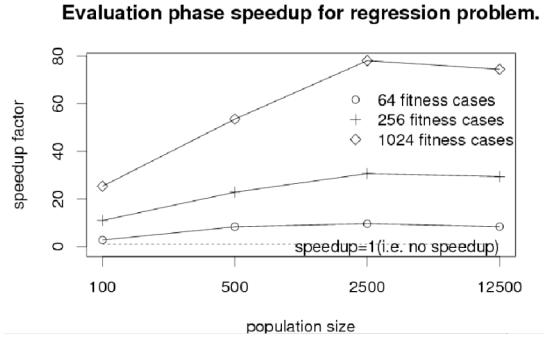



```
sp = 0 ;
for each instruction {
  switch (instruction) {
    case OPERAND: push operand;
    case ADD: stack[sp-2]=stack[sp-1]
                          +stack[sp-2];
              sp--;
    ... } }
```

http://www.gpgpgpu.com/EuroGPo8.pdf

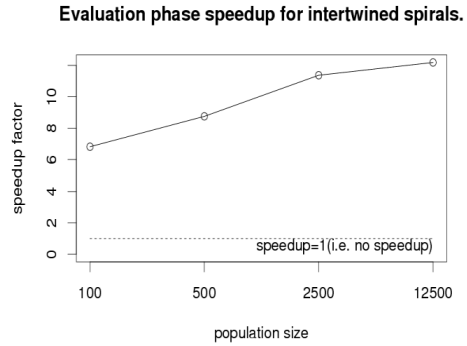
Results

- Example 1 : Regression on $x^6 - 2x^4 + x^2$



Results II

- Example 2: Spiral classification



Implementation

- Available here:

http://www-lil.univ-littoral.fr/

~robillia/EuroGP08/gpuregression.tgz

- Compatible with ECJ
  - http://cs.gmu.edu/~eclab/projects/ecj/

My CUDA GP

- Currently developing new CUDA implementation
- Hardware
  - 4 GPUs (2 x 9800 cards)
    - 512 shader processors!
  - 4 core CPU

Software

- CUDA for low level
  - Evolved programs executed as CUDA programs
- C# top end
  - Reuse existing software library
- GASS Cuda.Net middleware

CUDA.Net

- http://www.gass-ltd.co.il/en/products/cuda.net/

GASS

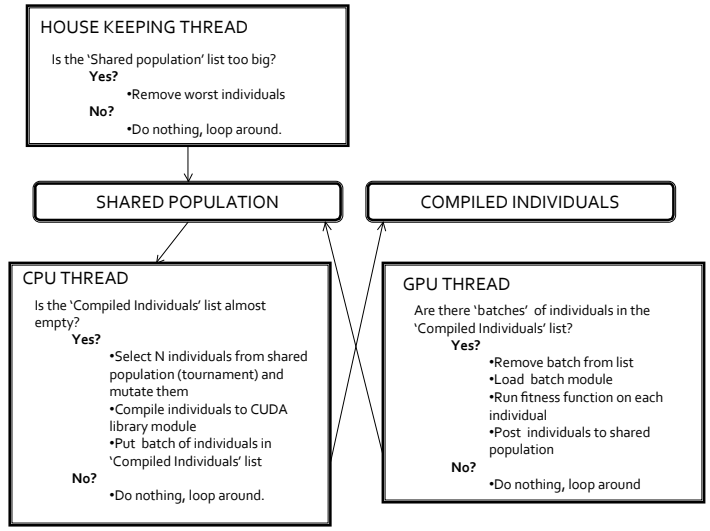
Allows ANY .Net language to communicate with CUDA.

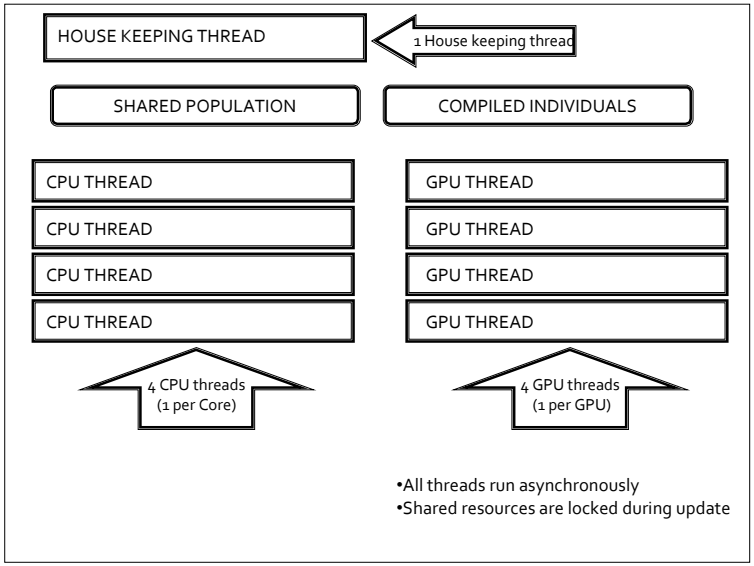
CUDA.Net

- An effort to allow .NET programmers work with GPU hardware
- A wrapper over CUDA
- Using same semantics as CUDA - making easy to port applications
- Fully complaint with CUDA 1.1 and CUDA 2.0

GP with CUDA.Net

- Similar approach to D. Chitty's Cg implementation:
  - Individuals are converted to C programs (shader fragments in Cg).
  - Then compiled.
  - Then executed and evaluated.
  - Data parallel approach





Example Individual

```
/* ------------------------------------- */
//START INDIVIDUAL3
extern "C" __global__ void Individual3(float* ffOutput, float* ffInput, int width, int height)
{
    //set up indexes of where to read from
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    unsigned int DataIndex = yIndex * width + xIndex;
    //
    //Number of operations : 7
    //
    ffOutput[DataIndex] =
    (ffInput[DataIndex]) - ((((ffInput[DataIndex]) / (ffInput[DataIndex]))
    + (ffInput[DataIndex])) / (((ffInput[DataIndex]) + (ffInput[DataIndex]))
    + ((ffInput[DataIndex]) / (ffInput[DataIndex]))))
    ;
}

//END INDIVIDUAL3
/* ------------------------------------- */
```

Fitness Evaluation

First we have to set up our data:

```
public static float[] cpuInputData = null;
public CUdeviceptr gpuInputData;

public static float[] cpuExcpectedData = null;
public CUdeviceptr gpuExpectedData;

public float[] cpuOutputData = null;
public CUdeviceptr gpuOutputData;
```

Fitness Evaluation II

```
cpuInputData = new float[TestSetHeight * TestSetWidth];
cpuExcpectedData = new float[TestSetHeight * TestSetWidth];

uint TotalTestCases = TestSetWidth * TestSetHeight;
for (uint i = 0; i < TotalTestCases; i++)
{
    cpuInputData[i] = (float)((float)(i) / 100000);
    cpuExcpectedData[i] = (float)Math.Sin(cpuInputData[i]);
}
```

- Populate the data

```
cpuOutputData = new float[TestSetHeight * TestSetWidth];
```

- Create room to get some output back

- Push data on GPU

```
this.gpuInputData = this.CudaDevice.CopyHostToDevice<float>(cpuInputData);
this.gpuExpectedData = this.CudaDevice.CopyHostToDevice<float>(cpuExcpectedData);
this.gpuOutput = this.CudaDevice.Allocate<float>(this.cpuOutputData);
this.gpuDifference = this.CudaDevice.Allocate((uint)(4 * this.cpuOutputData.Length));
this.gpuSum = this.CudaDevice.Allocate((uint)(4 * this.cpuOutputData.Length));
```

Fitness Evaluation III

- We can then load a library and call a function
- This executes 1 of the compiled individuals

```
CUfunction p = this.CudaDevice.GetModuleFunction(CudaModule, Ind.FunctionName);
this.Routines.SetParameters(p,
    gpuOutput,
    gpuInputData,
    (uint)TestSetHeight,
    (uint)TestSetWidth);
this.CudaDevice.SetFunctionBlockShape(p, BLOCK_DIM, BLOCK_DIM, 1);
this.CudaDevice.Launch(p,
    (int)TestSetHeight / BLOCK_DIM,
    (int)TestSetWidth / BLOCK_DIM);
```

Fitness Evaluation IV

- We can then compute the error (difference between output and expected)

```
this.Routines.AbsDifference(ref gpuOutput,
    ref gpuExpectedData,
    ref gpuDifference,
    TestSetWidth, TestSetHeight);

this.Routines.Sum(ref gpuDifference,
    ref gpuSum,
    (int)(TestSetHeight * TestSetWidth));

float[] SumOutput = new float[1];
this.CudaDevice.SynchronizeContext();
this.Routines.CopyGPUArrayToCpu(ref SumOutput, ref gpuSum);
```

This finds the difference for each element in the arrays.

We sum all the elements in the difference array.

And bring this single number back from the GPU – it's the fitness score!

(note that I've wrapped up the Sum and AbsDifference calls in a library!)

Results

Length of genotype

Length of test set	16	32	64	128	256	512	1024
65536	0.01	0.01	0.01	0.01	0.01	0.01	0.01
262144	0.04	0.04	0.05	0.05	0.05	0.04	0.04
1048576	0.16	0.20	0.17	0.17	0.21	0.20	0.17
4194304	0.55	0.49	0.63	0.72	0.63	0.63	0.65
16777216	1.36	1.41	1.71	1.56	1.69	1.67	1.62

Average Giga GP Op/s
(including all overheads)

Results II

Length of genotype

Length of test set	16	32	64	128	256	512	1024
65536	0.02	0.02	0.02	0.02	0.02	0.02	0.02
262144	0.07	0.08	0.08	0.08	0.08	0.08	0.07
1048576	0.28	0.30	0.31	0.30	0.30	0.30	0.28
4194304	0.92	0.94	0.99	1.02	1.01	1.00	0.92
16777216	2.07	2.22	2.28	2.32	2.34	2.34	2.29

Peak Giga GP Op/s
(including all overheads)

Conclusion

- This approach looks fast!
  - We've got some big data sets (images) to process, so data wide is best for our approach.
- Suitable for multiple GPU/CPU environments
  - We will even try this on a cluster of GPU machines
- Early days. But look forward to seeing the first publications next year.

Summary

- Different implementation methods for GP on GPUs
  - Fitness case distribution
  - Population distribution
  - Single Fitness case acceleration
  - Full implementation on Xbox CPU/GPU
- Different GP representations used
  - Cartesian GP (Harding)
  - Tree GP (Langdon)
  - Linear GP (Wilson)

Summary (II)

- Accelerations from <1 to 8,000, typical 10...50
- Advantage will exponentiate over coming years
- Requires new thinking about GP data structures
- High-level tools need some further improvement
- Potential to connect to visualization

Future Work

- GPU technology is moving very quickly therefore newest hardware is necessary (e.g. multiple cards)
- Further evaluation of new software tools
- Other parallel architectures, Cell and Multicores
- Technology can continue to drive our applications, provided we stay on top of the wave

Questions?

- Our website: www.gpgpgpu.com run from beautiful Newfoundland