Linear Genetic Programming GPGPU on Microsoft's Xbox 360

Garnett Wilson, *Member, IEEE*, and Wolfgang Banzhaf

We describe how to harness the graphics processing abilities of a consumer video game console (Xbox 360) for general programming on graphics processing unit (GPGPU) purposes. In particular, we implement a linear GP (LGP) system to solve classification and regression problems. We conduct inter- and intra-platform benchmarking of the Xbox 360 and PC, using GPU and CPU implementations on both architectures. Platform benchmarking confirms highly integrated CPU and GPU programming flexibility of the Xbox 360, having the potential to alleviate typical GPGPU decisions of allocating particular functionalities to CPU or GPU.

I. Introduction

Modern video game consoles are, in essence, graphics supercomputers [1], particularly at product launch. The XBox 360, at its launch on November 22, 2005, was the first PC or console to feature CPU chip multi-processing (CMP) with more than 2 cores (using 3 cores). It was also the first console to feature a graphics processing unit (GPU) with unified shader architecture (no distinct vertex and pixel shader engines). New generations of consoles feature large jumps in system performance, and occur in approximately five year intervals. Moreover, current GPUs in general provide considerable computational power. A survey of GFLOP performance of nVidia and ATI graphics cards compared to Intel processors from 2002 to late 2005 demonstrated that GPUs exceed Moore's Law, which predicts that general computing power doubles every 18-24 months [2]. In contrast, graphics hardware performance doubled every six months, whereas Intel PC CPUs did not meet predictions of Moore's Law. Similar results were reported for nVidia products up to late 2006 in [3].

In late 2006, Microsoft launched XNA's Not Acronymed (recursive acronym "XNA") Game Studio Express 1.0, which integrated with C# Studio Express. Microsoft's goal with the XNA framework was to empower academics and independent developers to create "homebrew" games for its commercial video game console Xbox 360 [4]. We describe how to perform genetic programming (GP) using XNA and the Xbox 360. Furthermore, we attempt to maximize the hardware resources of the system: the central processing unit (CPU) is used to run a GP tournament, while the GPU is used to perform parallel genetic operations. This work thus presents the first implementation of a research-based GP system on a commercial video game platform. It is also the first time that linear genetic programming (LGP) has been implemented in a general programming on graphics processing unit (GPGPU) application, and it is the first instance of the Xbox 360 being used for any GPGPU purpose. To show that it is possible to program the Xbox 360 for GP research is a step towards harnessing the power of future video game consoles.

The following Section provides a brief description of GPGPU programming and describes previous evolutionary computation work involving GPUs. Section III describes XNA programming and engineering considerations for the Xbox 360. Implementation details of the GPGPU GP algorithm in the XNA framework, separated into CPU and GPU functionality, are provided in Section IV. The regression and classification benchmarks are described in Section V. Comparison of GPU run time performance versus an identical CPU-only implementation on a Windows PC and the Xbox 360 are provided in Section VI.

II. GPGPU Overview and Evolutionary Computation with GPUs

A. *How GPGPU Programming Works*

GPUs have the ability to perform restricted parallel processing, hence the increasing interest among researchers in using them for applications requiring intensive parallel computations. The type of parallel processing used by GPUs is referred to as single instruction multiple data (SIMD), where all the processors on the graphics unit simultaneously execute the same code on different data. To be specific, a GPU is responsible for simultaneously rendering the pixels it is provided on an assembly of these pixels called a "texture." (Pixels of a texture are often called "texels" when considered as a portion of a texture.) The GPU processes the texture it is provided and outputs a vector of four floating point numbers for each texel processed, traditionally corresponding to RGBA (red, green, blue, and alpha, for transparency) channels of a color. The two components of GPU architecture that a user can control are the set of vertex processors and the set of pixel (or fragment) processors. An effect file, which is a program to control the GPU, is divided into two parts corresponding to the architecture: a pixel shader and a vertex shader. The vertex shader program transforms input vertices based on camera position, and then each set of three resulting vertices

This work was supported by a PRECARN Postdoctoral Fellowship.

G. Wilson is with the Department of Computer Science, Memorial University of Newfoundland, St. John's, NL, A1B 3X5, Canada (e-mail: gwilson@cs.mun.ca) and Verafin, Inc. St. John's, NL, Canada.

W. Banzhaf is head of the Department of Computer Science, Memorial University of Newfoundland, St. John's, NL, A1B 3X5, Canada (e-mail: banzhaf@cs.mun.ca).

compute a triangle from which pixel (fragment) output is generated and sent to the pixel processors. The shader program instructs the pixel shaders (processors) to "shade" each pixel in parallel and produce the final pixel with associated RGBA values for final output. Even though the latest GPUs (such as all those used this paper) use unified architecture, where the shader processors can handle vertex or pixel commands, the two functionalities are still separated when composing effect files.

GPGPU applications tend to take advantage of pixel shader programming rather than using the vertex shaders, mainly because there are typically more pixel than vertex shaders on a GPU and the output of the pixel shaders is fed directly to memory [5]. (In contrast, vertex processors must send output through both the rasterizer and the pixel shader sections of the GPU.) In terms of traditional data structures and execution, GPU textures are analogous to arrays, the shader program is like a Kernel program, and rendering effectively executes the program. The CPU runs the main program, and sends data in texture form to the GPU when parallel processing is required. The GPU renders to a texture in its memory (rather than to the screen), and the output texture data is consumed by the main (CPU-side) program. A summary of the elements of GPU hardware and flow of execution in GPU programming as they apply to our XNA implementation is given in Figure 1.

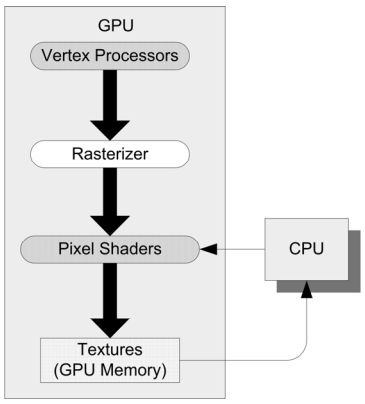

Fig. 1. GPU architecture and execution flow in GPGPU programming. Programmable elements of the GPU are shaded darkest. Thin arrows indicate passing of data in texture form between pixel shaders and CPU.

APIs for accessing the functionality of the GPU differ in level of abstraction. Lower level alternatives for GPU programming include DirectX and the Open Graphics Library (OpenGL). The next level of abstraction features C-style languages including C for Graphics (Cg), Microsoft's High Level Shader Language (HLSL), and nVidia's Compute Unified Device Architecture (CUDA). At the highest level are libraries that are integrated with object-oriented languages such as Sh (now RapidMind) with C++ and Microsoft Research's Accelerator [6] with C#.

B. *Evolutionary Algorithms on GPUs*

The first GPU-centered applications to use evolutionary algorithms in general naturally applied them to textures for use in image processing. The idea of applying genetic programming to evolve shaders was first suggested by Musgrave [7]. Loviscach and Meyer-Spradow used genetic programming to evolve pixel shaders in OpenGL, and applied them to textures with user feedback based on aesthetic required for determination of a fitness [8]. Ebner *et al.* [9] implement a similar strategy with Cg. Lindblad *et al.* [10] apply linear GP (LGP) with DirectX to the interpretation of 3D images.

Moving from GPU for traditional image analysis, general purpose computation (GPGPU) techniques were later tried using evolutionary algorithms. In [11], Yu *et al.* use Cg to implement a GA on a GPU using the fine-grained parallel model where each point of a 2D grid is an individual, which itself becomes a parent with its best neighbor. The chromosome of each individual is divided sequentially into several segments that are distributed across a number of textures with the same position. Each chromosome segment consists of four genes in each of a pixel's RGBA components, with a separate texture storing the fitness values of the pixel individuals. Unlike others, they implement fitness evaluation, selection, crossover, and mutation operators in shader programs on the GPU. For large populations, the GPU implementation was found to be faster than one on the CPU for the regression benchmark. Indeed, performance gains through GPU use for large populations have been found to be typical in EC-based GPGPU research.

Fok *et al.* implement EP (evolutionary programming) on the GPU in [12]. The individuals in a population are represented as textures on the GPU, as are fitness, random number, and indexing requirements. They determine empirically that it is most effective to implement mutation, reproduction, and fitness evaluation with the GPU while the CPU performs competition and selection (where GPU versions of those functionalities were also tried). Cartesian GP is implemented by Harding and Banzhaf in [13] using C# and Microsoft Accelerator, with Accelerator handling the compilation of CGP expressions into shader programs, execution of the programs, and the return of results as array data. Chitty [14] implements a tree-based GP implementation, using OpenGL to create data textures and converting tree GP individuals to Cg shader programs for evaluation on the GPU. Langdon and Banzhaf [15] created a GPU-based interpreter using RapidMind and C++ that operates on tree-based individuals. A modified implementation of the GPU-based interpreter approach is applied by Langdon and Harrison to represent extremely large populations for a bioinformatics application in [16].

III. Design considerations for Xbox 360 GPGPU

The C# XNA framework naturally provides the user with access to the Xbox 360 CPU. In addition, HLSL programs can also be loaded in an XNA program, allowing the user to perform vertex and shader programming with the Xbox 360 GPU. In early 2007, the updated XNA Game Studio Express 1.0 (Refresh) was released, which is used in this work. Given the accessibility that Microsoft has provided to the Xbox 360's CPU and GPU, and its continuing development of the XNA product, it is an obvious choice for implementing GPGPU applications. At the time of this writing, the authors are not aware of any other console manufacturers that have provided consumer access to programming of the GPU hardware. Microsoft Accelerator is not compatible with the XNA framework, so programming of shaders in HLSL is required to provide GPU access in XNA. The Xbox 360 hardware is ideal for graphics processing: each console features a custom built IBM PowerPC-based CPU with three 3.2GHz core processors sharing a 1Mb L2 cache. Each CPU core also has an associated complement of three SIMD vector processing units. The 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. The Xbox GPU by ATI houses 48 parallel shaders with unified architecture and 10 MB of embedded DRAM (EDRAM) [1], with 512 MB of DRAM as main memory.

Implementations created with the XNA framework can be deployed on the Xbox 360, Windows XP with SP2, and Vista variants. An XNA project requires that separate initialization (*Initialize*), update of program logic (*Update*), and rendering of graphics (*Draw*) methods be implemented. The program runs simply by repeatedly updating the *Update* and *Draw* methods—it is implemented as a video game that is constantly checking its logic and updating the graphics on the screen. The Draw method is the main component of a GPGPU implementation, as this is where the shader programs on the GPU will be called from. Rather than use a typical loop construct for GP tournament execution, the repeated execution of the Draw method is harnessed to conduct generational tournaments over trials.

The XNA framework provides a means of processing and compiling supported game assets such as textures and shaders called the "content pipeline." The content pipeline does not permit dynamic loading or switching of shader programs to the GPU. This precludes GP implementations (such as [13, 14]) that provide individuals to the GPU for processing, use the CPU to subject them to genetic operators, and reload them to the GPU. This allows XNA to provide faster loading of the GPU for rendering because all data is already pre-compiled to the correct format [17]. This decision certainly makes sense from the viewpoint of Xbox 360 console end users and speeds up genetic programming with GPGPU that repeatedly uses the shader (as we do). The XNA framework currently does not feature I/O to the Xbox hard drive or memory units, so all data must be output to the screen. An open source XNA keyboard component available from [18] allowed user input from the Xbox 360 control pad or USB keyboard connected directly to the console.

Pixel Shader version 3.0 (the most advanced shader profile supported by the Xbox 360 GPU) was used. Textures were rendered with XNA's surface format *HalfVector4* type, so that four 16 bit floats were placed per texel with one float per channel. A list of rendering options using the Xbox 360 GPU with XNA is available at [19], with HalfVector4 being the highest precision supported by the Xbox GPU while still allowing texture compactness of four channels per pixel. Some pertinent restrictions for the shader program are practically universal to all GPUs at the time of this writing. For instance, it is important to keep in mind when designing the GPGPU shaders that GPUs can only implement *gather*, but not *scatter*. (At least this has been the case until very recently, since nVidia has introduced that functionality in CUDA [3].) The only way to retrieve data from a shader program in XNA is to render results to a texture on a target buffer, and then read the texture's content back into the main calling program. One cannot load array or variable parameters into a shader program, alter their values in the shader code, and have their new values returned following execution. The GPU is optimized to only render textures, and that is its only means of returning values: any values required must thus be rendered by the GPU to internal targets. Furthermore, array data stored on textures passed to shaders as parameters must be referenced in the shader programs using texture coordinates appropriate to the mapping scheme of the GPU-dependent coordinate system.

In addition to XNA framework and general GPU restrictions, the Xbox GPU hardware (and Pixel Shader 3.0) have additional specifications that can be determined by querying the Xbox 360 with the XNA GraphicsDevice class or checking XNA documentation [20]. On the Xbox 360, a shader program can consist of 2048 instructions, with flow control depth of 4. (That is, a maximum of four instructions can be called from inside each other.) The Xbox 360 GPU can also support 16 simultaneous textures, with a maximum texture height and width of 8192. All engineering requirements of this section are met in the implementation now presented in Section IV.

IV. Implementing the GP algorithm using XNA

A. *GPU-side XNA GP Textures and Shaders*

Populations of individuals are, naturally, represented as textures to be processed by the GPU. Fitness cases are also represented as a texture that is passed to the GPU and referenced by the shader programs. Each individual's instructions are divided into eight chromosomes, separated into two sets of four, and each set is placed on a texel on two

separate textures. Collectively, the two textures perform an operation on the contents of either of two sources (fitness case or register content), and place the result in a target register according to the equation

$$target = src1\ op\ src2.$$

The variable integer *op*, *op* = [0, 3], indicates one of four operators ADD, SUB, MUL, or DIV. The integer *target*, *target* = [0, 3], indicates one of four target registers. The sources src_1 and src_2 can specify either fitness cases or registers based on flags in each instruction, and thus take values in [0, MAX(*classification features or regression inputs*, *registers*)]. An integer *id*, *id* = [0, *population size*] is also used in these textures to label the individual, and an integer *PC*, *PC* = [0, *instructions*] serves as a program counter 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. The texels of the first texture each possess the variables {*op*, *target*, *id*, *PC*} in their four color channels, and the texels of the second texture correspond to $\{f_1, src_1, f_2, src_2\}$. As the XNA *HalfVector4* surface format was used, each chromosome (channel) was a 16 bit float (interpreted as an integer as appropriate). The two textures represent a whole population, with each individual being a column of texels, and each texel in the column being an instruction. The width of the textures (in texels) is thus the number of individuals in the population, and its height is the number of instructions in an individual. As the shader interprets each dual-texel instruction across both textures at the same coordinate, the current state of an individual's four registers (following that instruction) are kept in a third texture's texel (at the same coordinates) as a set of four floats. With this representation, the fitness shader program can interpret an instruction across all individuals in the population at once, using the program counter to track which instruction is currently being executed. Textures representing individuals are depicted to the left in Figure 2.

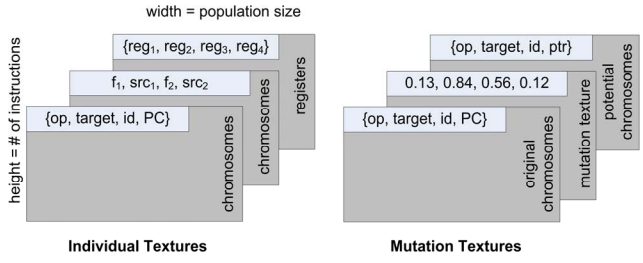

Fig. 2. Individual (left) and mutation (right) texture representations.

Textures are also associated with the mutation operator to allow its implementation on the GPU. To accomplish this, a texture with the same dimensions of the two chromosome textures is filled with randomly generated float values in the interval [0, 1] in the C# code (CPU-side). The value of each chromosome (texel color channel) is evaluated against the mutation threshold, and if it is less than or equal to the threshold, the corresponding original chromosome in the population is replaced with a chromosome at the same texture coordinate in a third replacement texture. The replacement texture is filled with randomly generated chromosome values meeting the specifications of the two original population textures. The textures to implement mutation are shown to the right in Figure 2.

Using the population textures just described, fitness evaluation in the shader uses the HLSL pseudocode:

```
float4 FitnessShader(float2 currentLocation : TEXCOORD0 ) : COLOR
    if (row in instruction textures == program counter)
        if (flag1 == 1) source1 = fitnessCases[src1]
        else  source1 = registers[src1]  // flag1 == 0
        if (flag2 == 1) source2 = fitnessCases[src2]
        else  source2 = registers[src2]  // flag2 == 0
        if (op == 0) register[target] = source1 + source2
        if (op == 1) register[target] = source1 - source2
        if (op == 2) register[target] = source1 * source2
        if (op == 3) register[target] = source1 / source2
    return registers;
```

The fitness shader above forces the GPU to process the population one instruction at a time (via the program counter check in the first IF statement). The shader thus runs for *k* passes, *k* = instructions per individual. As each instruction is interpreted, the register states for the previous instruction are retrieved via texture lookup. Depending on the source flags, data is retrieved via texture lookup from the fitness cases or from the registers. The operation is determined, and the result is placed in the appropriate target register. The shader program does *k* = 16 passes (number of instructions per individual) in its single technique definition. (Techniques define which functions of the shader Effect files are to be executed and what parameters they will be given.) As each pass is completed, the updated register texture is fed back to the shader on the CPU (C#) side where the loop over the passes is conducted. The shader thus interprets the correct instruction in each individual with updated register contents.

The mutation shader, *Mutate.fx*, is run with two techniques with identical logic, one with the mutation texture and original and replacement {*op*, *target*, *id*, *PC*} textures, and one with the mutation texture and the original and replacement $\{f_1, src_1, f_2, src_2\}$ textures. The mutation shader operation is relatively simple: the color channel of each texel in the original texture is replaced by the replacement channel value in the replacement texture if the mutation's texture channel value at the same coordinate exceeds the threshold. The mutation shader thus executes mutation on every instruction in every individual in the entire population simultaneously. The mutation HLSL shader code is:

```
float4 MutateShader1(float2 currentLocation : TEXCOORD0 ) : COLOR
    // default is to keep all chromosomes the same
    float4 outData = tex2D(originalPopSampler, currentLocation.xy);
    // look up mutation thresholds, potential replacement chromosomes
    float4 mutate = tex2D(mutatePopSampler, currentLocation.xy);
```

```
float4 replacement = tex2D(replacePopSampler, currentLocation.xy);
// if mutate pixel value exceeds threshold, replace chromosome
if (mutate.x <= mutationThreshold) outData.x = replacement.x;
if (mutate.y <= mutationThreshold) outData.y = replacement.y;
if (mutate.z <= mutationThreshold) outData.z = replacement.z;
if (mutate.w <= mutationThreshold) outData.w = replacement.w;
return outData;
```

B. *CPU-side XNA GP Algorithm Implementation*

Using the flow control of an XNA program (all relevant required methods to implement the XNA *Game* class are shown) and implementing arrays conforming to XNA texture objects where possible (rather than traditional C# data structures such as Collections or Lists), the CPU controls the overall operation of an LGP generational tournament:

```
GPGame {
   GPGame()  //constructor
      provide set of random seeds for trials
   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
```

The above pseudocode for the main program (C#) uses the continual frame refresh of an XNA program as the driving force behind the GP tournament. By using the Boolean checks throughout the Draw method, a GP tournament is implemented. The program begins with the on-screen keyboard, where the user specifies whether or not to include GPU functionality, population size, mutation rate, and number of rounds. In the Draw() method of the GP tournament, for each fitness case, the *Fitness.fx* shader interprets all instructions across all individuals, updating register information over effect passes as described in Section IV.A. Fitnesses are then updated and fitness-proportionate roulette wheel generational selection, followed by mutation, is performed on the population textures. Finally, textures and fitness information are displayed on the screen. As the population converges toward a solution, the population textures will move from random pixel colors throughout to colored bands across the texture (corresponding to greater uniformity of instructions across individuals). Upon publication, all XNA projects (including source code) will be available at *www.cs.mun.ca/~gwilson/XNA_LGP.html*. A screen shot of the implementation is provided in Figure 3.

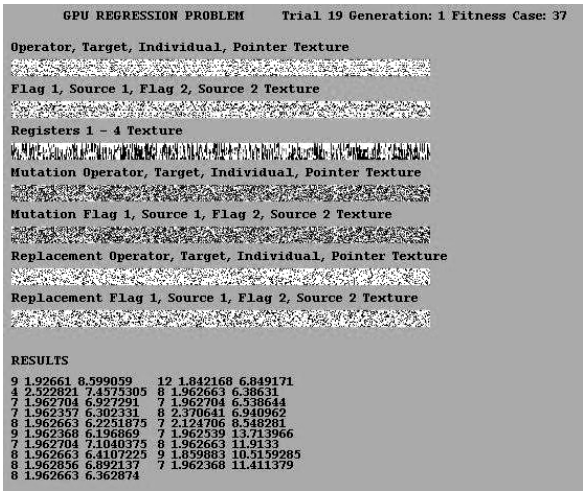

Fig. 3. XNA-based Linear GP GPGPU implementation screenshot. Each Result indicates the best hits, best raw error, and trial time (seconds).

From top to bottom of the screenshot in Figure 3, the first two textures display the relevant instruction components in each instruction (texture row) over the whole population (each individual is a texture column). It is these two textures that will form uniform horizontal bands as the population converges toward a solution. The state of each of the four registers at each instruction for the population is found in the third texture from the top. Mutation values for instruction segments in the first and second texture are in the fourth and fifth textures, respectively, with their potential replacement values located in the sixth and seventh textures, respectively. Results at the bottom of the screen display best hits, best raw error, and trial time in seconds.

V. EXPERIMENTS

Both a classification and regression benchmark were implemented. A CPU-only version of the implementation was also created, which simply implemented all shader functionality with appropriate C# code. The Ecoli problem from the UCI machine learning repository was chosen for classification [21], using 75% of the entire data set for training while retaining class distribution. Training was performed over 50 generations to benchmark processing times. Problem implementations were checked for correctness: CPU and GPU variants produced identical or similar results (given CPU and GPU float rounding differences). The CPU and GPU implementations are identical on both PC and Xbox 360, so the same code on both platforms is always compared, with respect to implementation. The sextic polynomial $x^6 - 2x^4 + x^2$ introduced by Koza [22] was implemented for regression, using float inputs in the range [0, 1] for 50 fitness cases. For

the experiments, the implementation was run on a Windows Vista Business PC, using an AMD Athlon 64 Processor 3500+ (2.21 GHz), 1023 MB of RAM, and a (at time of writing) state of the art ASUS EN8800GTX video card with an nVidia GeForce 8800 GTX GPU on board. The nVidia GPU features 128 parallel stream processors with unified shader architecture [23]. See Table 1 for parameterizations.

TABLE I
XNA LINEAR GP GPGPU PARAMETERS

Function Set	ADD, SUB, MUL, DIV (on floats)
Fitness	fitness-proportionate roulette wheel
Population	10, 1000, or 4000 individuals
Mutation	threshold = 0.1
Tournament	generational, 50 rounds
Fitness Cases	Classification: 251 training cases, 7 float features, 8 integer categories Regression: 50 cases, $x = [0, 1]$
Fitness Metric	Classification: correct classification, based on Reg[0] mapping to category Regression: 50 hits, where a hit is Absolute(Reg[0] – y) <= 0.01

VI. RESULTS

A. *Intra-Platform CPU and GPU Performance*

The fitness evaluation shader, while it does allow parallelization in that it processes every instruction in every individual at once, is relatively expensive for a shader. In order to process register subresults for LGP, it cannot process every instruction in every individual (all texels) at once (as is typically possible in EC and GA). Even interpreter approaches such as [15] process some instruction per pixel, rather than necessitating multiple passes that only operate on portions of the texture at a time. LGP requires such a process to store subresults in registers following each instruction. In contrast, the mutation shader operates on all instructions in all individuals at once, maximizing shader utility. Given these considerations, we compared execution times of both Fitness and Mutation shaders, the Mutation shader only, and the CPU on the Windows platform with nVidia GPU. Results are shown in Figure 4.

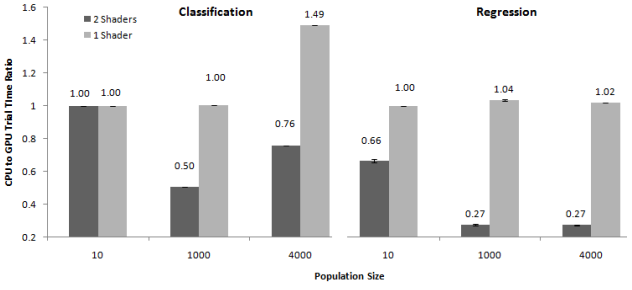

Fig. 4. PC CPU to GPU mean trial time ratios for both 1 (Mutation) and 2 (Mutation and Fitness) GPU shaders with standard error, based on 10 trials of 50 generations for classification (left) and regression (right) benchmarks. Ratios greater than 1 show GPU use is faster, less than 1 that CPU is faster.

GPU performance gains of 4% and 2% over CPU are seen for the two larger populations in regression, with a substantial gain of almost 50% in speed with the population of 4000 for classification. It is expected that the GPU algorithm performance increasingly exceeds CPU performance with larger population sizes, with this trend likely being amplified by the high number of training cases in the classification problem (251) as opposed to regression (50). It is evident from Figure 4 that LGP fitness evaluation is best left to the CPU: for neither benchmark does the use of *Fitness.fx* provide a performance gain. Standard errors reflect that times are quite consistent across all trials. Thus, the GPU functionality of the *Mutation.fx* shader only was compared to the CPU on the Xbox 360 (Figure 5).

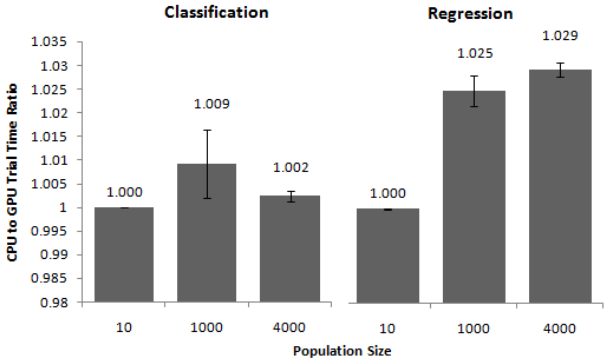

Fig. 5. Xbox 360 CPU to GPU mean trial time ratios with standard error, based on 10 trials of 50 generations. Ratios greater than 1 show GPU use is faster, less than 1 that CPU is faster.

For the lowest population level, no speed increase with GPU usage is seen. Speed increase with higher populations (1000 and 4000) for classification is very low (less than 1%), whereas the speed increases for those populations in regression is somewhat more substantial (2.5% and 2.9%, respectively). The smaller contributions of GPU processing to the Xbox 360 benchmarking may be due to the CPU being optimized for graphics workloads (Section III): the CPU even features SIMD vector processing units for each core and the GPU reads directly from the CPU L2 cache. Rather than the segregated CPU and GPU components in a PC, the Xbox provides highly integrated cooperation between CPU and GPU. Thus, while the GPU enhances performance over CPU-only implementations, the difference is just not as significant as on PC platforms. A benefit of this tight integration of CPU and GPU on the Xbox 360 is that the programmer need not be as concerned about performance consequences of opting to place functionality on the CPU as opposed to GPU, providing flexible design decisions on hardware optimized for GPGPU.

B. *Inter-Platform CPU and GPU Performance*

While benchmarking shows that Xbox 360 CPU and GPU are tightly integrated, and thus reduce the impact of CPU versus GPU design dilemmas, it is of interest to see how the Xbox 360 fairs in terms of speed performance against the PC. Both implementations again contain identical code, and

are created with the XNA framework. A comparison of PC and Xbox CPUs is shown in Figure 6.

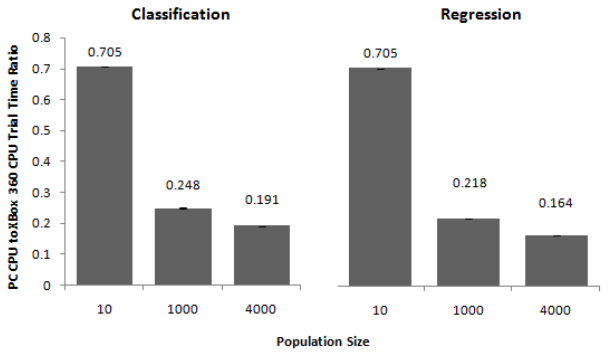

Fig. 6. PC CPU to Xbox 360 CPU mean trial time ratios with standard error, based on 10 trials of 50 generations. Ratios greater than 1 show the Xbox 360 CPU use is faster, less than 1 that the PC CPU is faster.

For both classification and regression problems, the PC CPU is faster than the Xbox 360 CPU. With increasing population, the performance of the Xbox 360 CPU further decreases in relation the PC CPU. This analysis must be interpreted with due consideration: the Xbox 360 CPU was designed for graphics processing. Furthermore, these metrics are obviously affected by the type of PC processor used (Athlon 64 Processor 3500+ 2.21 GHz), and the RAM available on the PC (1023 MB of RAM). The amount of RAM was the minium recommended system requirement for use of Vista Business, and the Xbox is put at a disadvantage with its 512 MB of RAM.

Furthermore, by coding the implementation to not use the GPU, even the Xbox 360 CPU is not being used at its full capacity. (As mentioned in the last section, the Xbox 360 CPU actually includes SIMD vector processors.) The CPU is custom made for the console, and while the graphics capabilities of the integrated CPU and GPU is enhanced, the graphics capabilities of the CPU are likely creating substantial additional overhead when traditional CPU implementations are run on the Xbox 360.

A more useful comparison, at least for practical purposes at time of publication, is that of the nVidia GeForce 8800 GTX GPU to the Xbox 360 GPU. The results for this comparison are given below in Figure 7.

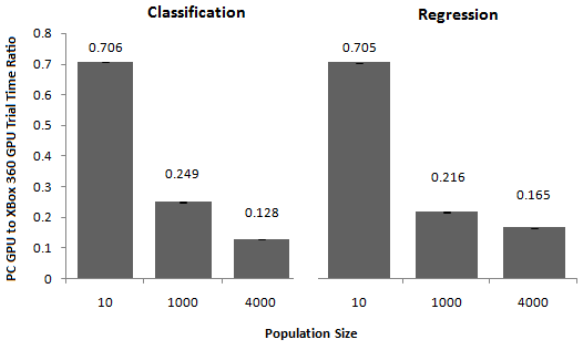

Fig. 7. PC GPU to Xbox 360 GPU mean trial time ratios with standard error, based on 10 trials of 50 generations. Ratios greater than 1 show the Xbox 360 GPU use is faster, less than 1 that the PC GPU is faster.

The GPU implementation on the Xbox 360 is approximately 25% and 22% of the PC GPU solution for populations of 1000 in both benchmarks, and 13% to 16% of the PC GPU performance for both benchmarks at a population of 4000. While these results demonstrate that the nVidia graphics card is faster than that of the Xbox 360 for every population level, they must be considered in the appropriate context. Recall that the Xbox 360 GPU incorporates 48 parallel shaders, while the nVidia card boasts 128. The Xbox technology is approximately two years older than the nVidia card: the nVidia GeForce 8800 GTX GPU was released November 2006 [24]. So if the Xbox was current with the PC GPU, would the integrated CPU and GPU provide a GPGPU performance advantage? To determine this approximately, we normalize the speed of Xbox 360 according to observed performance increase in GPUs over the past few years. According to [2, 3], the speed of GPUs ought to increase by 2^n where n is the number of 6-month periods between GPU products. The Xbox 360 benchmark is thus scaled to be $2^2 = 4$ times as fast; results are shown in Figure 8.

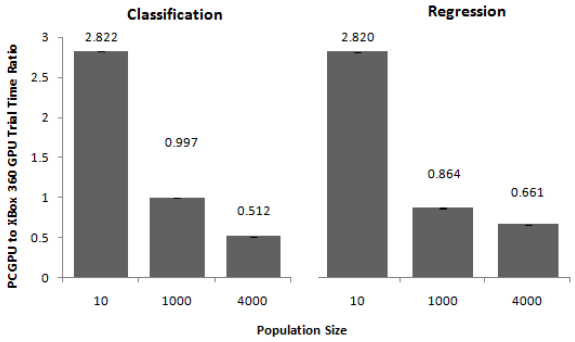

Fig. 8. PC GPU to Xbox 360 GPU mean trial time ratios with standard error, normalized to current generation of GPUs, based on 10 trials of 50 generations. Ratios greater than 1 show the Xbox 360 GPU use is faster, less than 1 that the PC GPU is faster.

Again, these results must be interpreted while keeping in mind that they are affected by PC CPU choice and the Xbox 360 possessing half the RAM of the PC platform. It is also likely that an integrated CPU will provide added performance to future iterations of the Xbox product. Even so, the Xbox 360 scaled performance surpasses (with approximately 2.8 times the speed) the CPU-intensive (lower population) graphics parameterizations. At the moderate population level of 1000, it is competitive with the nVidia with practically the same performance for classification and 86% of the regression performance. Only at the highest population level is the integrated architecture of the Xbox 360 surpassed by the nVidia GPU. It appears that if the Xbox 360 were to incorporate a current GPU, its architecture likely would provide superior or competitive performance for CPU-intensive applications that still

required moderate GPU use. For interested readers who wish to determine actual execution times for the experiments discussed, actual base CPU or GPU times are provided in Table 2. To determine execution times for any benchmark, simply multiply ratios in Figures 4 to 8 by the appropriate times.

TABLE 2
BASE TIME IN SECONDS FOR ALL IMPLEMENTATIONS

POPULATION	10	1000	4000
	REGRESSION		
PC, CPU VS. GPU (FIG. 4)	29.4	31.8	99.4
XBOX, CPU VS. GPU (FIG. 5)	41.7	145.5	608.2
PC CPU VS.XBOX CPU (FIG.6)	29.4	31.8	99.4
PC GPU VS. XBOX GPU (FIG. 7, 8)	29.4	30.7	97.7
	CLASSIFICATION		
PC, CPU VS. GPU (FIG. 4)	147.7	149.6	453.6
XBOX, CPU VS. GPU (FIG. 5)	209.4	603.7	2380.1
PC CPU VS.XBOX CPU (FIG. 6)	147.7	149.6	453.6
PC GPU VS. XBOX GPU (FIG. 7, 8)	147.7	149.1	304.0

VII. CONCLUSION

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. It is the first time that GPGPU, or genetic programming, has been implemented on a commercial video game console for research purposes. It also describes the first instance of a Linear GP implementation using GPGPU. We addressed performance considerations for the Xbox 360 for evolutionary computation in the GPGPU paradigm (and for GPGPU development in general), and found the Xbox 360 to offer tightly coupled CPU and GPU graphics performance.

Now that Microsoft's XNA framework has provided GPU access on a commercial video game console, it seems likely that this trend will continue to the next iteration of the Xbox console as Microsoft currently continues development of the XNA product. Programming for the Xbox 360 will likely place GPGPU developers in a position to take advantage of future console hardware advancements, while providing greater design flexibility with regard to CPU and GPU functionality. From a practical programming and future hardware viewpoint, it is worthwhile to use the Xbox 360 as an evolutionary computation GPGPU development platform.

REFERENCES

[1] J. Andrews and N. Baker, "XBox 360 System Architecture," *IEEE Micro*, vol. 26, no. 2, pp. 25-37, 2006.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," in Eurographics 2005, State of the Art Reports, 2005.

[3] nVidia, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," nVidia Corp., Santa Clara, USA, Version 0.8.2, 2007.

[4] Microsoft, (2007, Nov.). "XNA Game Studio Express." [Online]. Available: *http://msdn2.microsoft.com/en-us/directx/aa937795.aspx*

[5] M. Harris, "Mapping Computational Concepts to GPUs," in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* M. Pharr and R. Fernando, Eds. Boston, MA: Addison-Wesley Professional, 2005.

[6] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses " in Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS '06), San Jose, CA, 2006, pp. 325-335.

[7] F. K. Musgrave, "Genetic Textures," in *Texturing and Modeling: A Procedural Approach, 2nd Ed.*, D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, Eds. Cambridge, USA: AP Professional, 1998, pp. 373-385.

[8] J. Loviscach and J. Meyer-Spradow, "Genetic Programming of vertex shaders," in Proceedings of EuroMedia 2003, Plymouth, UK, 2003, pp. 29-31.

[9] M. Ebner, M. Reinhardt, and J. Albert, "Evolution of Vertex and Pixel Shaders," in Proceedings of the 8th European Conference on Genetic Programming, Lausanne, Switzerland, 2005, pp. 261-270.

[10] F. Lindblad, P. Nordin, and K. Wolff, "Evolving 3D model interpretation of images using graphics hardware," in Proceedings of the 2002 Congress on Evolutionary Computation (CEC 2002), Honolulu, Hawaii 2002, pp. 225-230.

[11] Q. Yu, C. Chen, and Z. Pan, "Parallel Genetic Algorithms on Programmable Graphics Hardware," *Proceedings of the First International Conference on Natural Computation, ICNC 2005*, vol. LNCS 3612, pp. 1051-1059, 2005.

[12] K.-L. Fok and T.-T. Wong, "Evolutionary Computing on Consumer Graphics Hardware," *IEEE Intelligent Systems*, pp. 69-78, 2007.

[13] S. Harding and W. Banzhaf, "Fast Genetic Programming on GPUs," in Proceedings of the 10th European Conference on Genetic Programming, Valencia, Spain, 2007, pp. 90-101.

[14] D. M. Chitty, "A Data Parallel Approach to Genetic Programming Using Programmable Graphics Hardware," in Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2007), London, England, 2007, pp. 1566-1573.

[15] W. B. Langdon and W. Banzhaf, "A SIMD interpreter for genetic programming on GPU graphics cards (in preparation)," 2007.

[16] W. B. Langdon and A. P. Harrison, "GP on SPMD parallel Graphics Hardware for mega Bioinformatics Data Mining (submitted)," *Soft Computing Journal* 2007.

[17] B. Nitschke, *Professional XNA Game Programming: For Xbox 360 and Windows, 1st Ed.* Hoboken, USA: Wrox, 2007.

[18] K. Jaegers and J. Jaegers, (2007, Nov.). "XNA Resources: Resources for XNA Game Developers." [Online]. Available: *http://xnaresources.com/pages.asp?pageid=27*

[19] Microsoft, (2007, Nov.). "Xbox 360 Surface Formats." [Online]. Available: *http://msdn2.microsoft.com/en-us/library/bb447675.aspx*

[20] Microsoft, (2007, Nov.). "XBox 360 Device Capabilities." [Online]. Available: *http://msdn2.microsoft.com/en-us/library/bb313967.aspx*

[21] D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz, (2007, Nov.). "UCI Repository of Machine Learning Databases. University of California, Department of Information and Computer Science." [Online]. Available: *http://www.ics.uci.edu/~mlearn/MLRepository.html*

[22] J. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge: MIT Press, 1998.

[23] nVidia, "Technical Brief: NVIDIA GeFOrce 8800 GPU Architecture Overview," nVidia Corp., Santa Clara, USA, TB-02787-001_v01, 2007.

[24] nVidia, (2007, Nov.). "New NVIDIA Products Transform the PC Into the Definitive Gaming Platform: New NVIDIA GeForce 8800 and NVIDIA nForce 680 Redefine Reality on the PC " [Online]. Available: *http://www.nvidia.com/object/IO_37234.html*