Genet Program Evolvable Mach (2010) 11:147–184
DOI 10.1007/s10710-010-9102-5

ORIGINAL PAPER

Deployment of parallel linear genetic programming using GPUs on PC and video game console platforms

Garnett Wilson · Wolfgang Banzhaf

Received: 29 April 2009 / Revised: 25 January 2010 / Published online: 18 February 2010

Abstract We present a general method for deploying parallel linear genetic programming (LGP) to the PC and Xbox 360 video game console by using a publicly available common framework for the devices called XNA (for "XNA's Not Acronymed"). By constructing the LGP within this framework, we effectively produce an LGP "game" for PC and XBox 360 that displays results as they evolve. We use the GPU of each device to parallelize fitness evaluation and the mutation operator of the LGP algorithm, thus providing a general LGP implementation suitable for parallel computation on heterogeneous devices. While parallel GP implementations on PCs are now common, both the implementation of GP on a video game console using GPU and the construction of a GP around a framework for heterogeneous devices are novel contributions. The objective of this work is to describe how to implement the parallel execution of LGP in order to use the underlying hardware (especially GPU) on the different platforms while still maintaining loyalty to the general methodology of the LGP algorithm built for the common framework. We discuss the implementation of texture-based data structures and the sequential and parallel algorithms built for their use on both CPU and GPU. Following the description of the general algorithm, the particular tailoring of the implementations for each hardware platform is described. Sequential (CPU) and parallel (GPU-based) algorithm performance is compared on both PC and video game platforms using the metrics of GP operations per second, actual time elapsed,

This work is based on an earlier work: Deployment of CPU and GPU-based Genetic Programming on Heterogeneous Devices, in Proceedings of the 2009 Genetic and Evolutionary Computation Conference, © ACM, 2009. http://doi.acm.org/10.1145/1570256.1570356

G. Wilson (✉) · W. Banzhaf
Department of Computer Science, Memorial University of Newfoundland,
St. John's, NL A1B 3X5, Canada
e-mail: gwilson@cs.mun.ca

W. Banzhaf
e-mail: banzhaf@cs.mun.ca

speedup of parallel over sequential implementation, and percentage of execution time used by the GPU versus CPU.

Keywords Genetic programming · Parallel processing · SIMD · Graphics processing unit (GPU) · GPGPU · Xbox 360 · Heterogeneous devices

1 Introduction

An increasingly popular means of conducting massively parallel computing is general-purpose computing for graphics processing units (GPGPU). The adoption of graphics processing units (GPUs) for parallel computing is due to both the low price point of the GPU hardware compared to other options for parallel processing and the rate at which the computing power of the GPU hardware brought to market increases [1]. Evolutionary computation in general, including genetic programming (GP), can usually be readily adapted to parallel computing techniques. Thus, a growing number of GP practitioners have been using GPGPU to reduce the computing time required by their applications. This work describes an algorithm for implementation of linear genetic programming (LGP) using the GPU for fitness evaluation and mutation sections of the algorithm. Moreover, the authors describe how to implement LGP so that it can be deployed so that it uses the GPU on heterogeneous devices.

The toolset and runtime environment package used in this work to access the GPU, Microsoft's XNA (recursive acronym "XNA's Not Acronymed") framework, is designed to allow execution on heterogeneous devices including PCs (Windows XP or Vista), video game console (Xbox 360), and even a portable digital media device (Zune).[1] The XNA framework allows the creation of computer games by consumer designers across these hardware platforms. The XNA framework is used with Microsoft's C# (for CPU programming) and High Level Shader Language, or HLSL (for GPU shader-level programming). C# is a high level object-oriented language that is part of Microsoft's current Visual Studio development environment, while HLSL is a lower level C-style language specifically useful for programming GPU shader programs. While the authors were able to produce a general algorithm that could be applied across devices, particular elements of each algorithm were changed due to hardware considerations and the XNA framework. Despite the changes that were unique to each platform, we aimed to keep the high-level parallel methodology general across PC and video game console devices. By designing the algorithm within the XNA framework for all platforms, what is effectively developed is a Linear Genetic Programming game for PC and XBox 360 that runs a sequential or parallel GP and displays results numerically and within colored textures on screen as evolution takes place.

[1] CPU-based GP was implemented on this device by the authors, see [15] for further details.

This main objective of this paper is to describe both a general methodology, and the platform-dependent requirements, to implement GP using GPGPU on PCs and video game consoles. While the general methodology is the same across the two platforms, two separate versions of the implementation were used: One goal of this work was to attempt to best utilize the potential GPU shader programming on the different platforms used for the LGP algorithm, rather than achieve identical implementations (program instructions) across platforms at the unnecessary expense of performance on a given platform. This goal was a continual trade-off with the additional goal of maintaining uniformity of the general algorithm across the heterogeneous platforms. A description of the details of the modifications that the authors implemented to allow such deployment on the two heterogeneous platforms is the main contribution of the work. In addition, some empirical examination of performance of parallel (CPU and GPU-based) and sequential (only CPU-based) implementations on the two platforms is provided. Due to the differences in both PC and video game console hardware and underlying operating systems, these results only provide a means of comparing the combined algorithm and hardware implementations (where they are highly interrelated in this work). That is, no conclusions can be drawn regarding the speed of the underlying hardware and software used to deploy the algorithms. However, results can be gleaned in the comparison of parallel and non-parallel implementations using the combination of LGP algorithm and targeted hardware platform.

Section 2 discusses parallel implementations using general-purpose computation on graphics hardware programming (GPGPU) and previous, related work involving parallel programming of GP with GPUs. Section 3 provides a brief overview of linear genetic programming (LGP) and general purpose computation on graphics processing units (GPGPUs) and introduces the GPU-based data structures used for the LGP individuals. Section 4 discusses some hardware-related considerations for each platform, and describes the general methodology of programming LGP using the data structures introduced in the previous section. Section 5 details platform-dependant differences in the general algorithm for the PC and Xbox 360 deployments of the parallel and sequential fitness functions, with implementation of the parallel and sequential mutation operator described in Sect. 6. Section 7 examines performance results for the GP regression sextic polynomial problem for the parallel and sequential algorithms and hardware combinations. Discussion and conclusions follow in Sects. 8 and 9, respectively.

2 GPGPU programming and related work

GPUs have the ability to perform restricted parallel processing, while being mass produced and inexpensive; hence researchers are increasingly interested in their use for applications requiring intensive parallel computations. The type of parallel processing used by GPUs is referred to as single instruction multiple data (SIMD) processing, where all the processors on the graphics unit simultaneously execute the same set of program instructions 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."[2] The GPU processes the texture and outputs a vector of four floating point numbers for each pixel processed, corresponding to *rgba* (red, green, blue, and alpha, for transparency) components of a color or the four components of a position (x, y, z and w for an algebraic factor). The two parts 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 compute a triangle from which pixel (fragment) output is generated and sent to the pixel shaders. The shader program instructs the pixel shaders (processors) to produce the colors of each pixel in parallel and place the final pixel in a memory buffer prior for final output. 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 [2]. In contrast, vertex processors must send output through both the rasterizer and the pixel shader sections of the GPU. The general architecture of a GPU is shown in Fig. 1, showing a level of architecture detail matching the discussion above.

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 includes 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) that integrates with C++ and Microsoft Research's Accelerator [3] integrating with C#. This work uses the XNA framework, which uses C# with GPU programming done in HLSL. Both implementations described herein (PC and XBox 360) use only the classes and methods specified by the XNA framework (using C# and HLSL). Thus, the algorithms can be constructed with Visual Studio 2008 with XNA framework installed and video card with appropriate GPU. The only additional requirement to execute these algorithms on an XBox 360 is a membership in the XNA Creators Club, which has a recurring subscription fee.

A number of evolutionary computation practitioners have demonstrated significant speed-ups in computation for some time now using a number of distributed and parallel computing techniques, due to the fact that evolutionary computation-based algorithms are easily parallelizable [1, 4]. 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 [5]. Loviscach and Meyer-Spradow [6] used genetic programming to evolve pixel shaders in OpenGL and applied them to textures with user feedback required for determination of a fitness based on aesthetic; Ebner et al. [7] implemented a similar strategy with Cg. Lindblad et al. [8]

[2] Pixels of a texture are often called "texels" when considered as a portion of a texture. However, we use the terminology "pixels" throughout this paper.

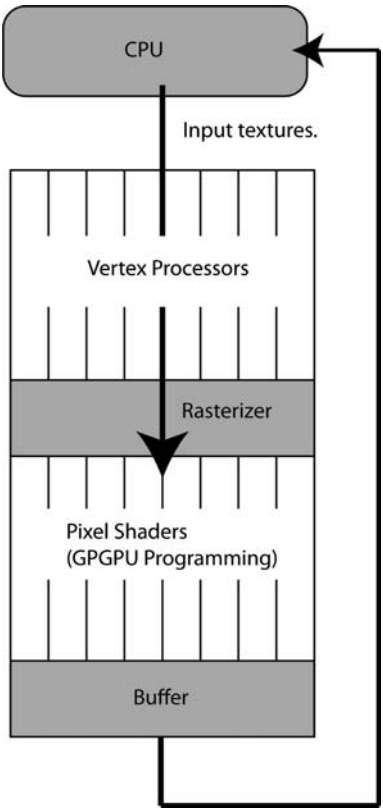

Fig. 1 Typical GPU architecture. The CPU sends information on textures and an Effect file containing pixel shader instructions to the GPU for parallel processing. Pixel shaders perform the GPGPU programming (instructions in the Effect file) with the texture as input. The *arrow* indicates the path of the texture/data from the CPU: Vertex processors are effectively bypassed (although they can be programmed), as is the rasterizer. Contents of the textures are manipulated by the pixel shaders, stored in a buffer, and then passed back to the CPU

applied linear GP (LGP) with DirectX to interpret 3D images, with fitness determining the difference between target and rendered images.

Moving from GP using the GPU for more traditional image analysis, general purpose computation on GPUs (GPGPU) techniques were later tried using evolutionary algorithms. Yu et al. [9] use Cg to implement a GA on a GPU using a 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 components, with a separate texture storing the fitness values of the pixel individuals. Their implementation incorporated 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 the one on the CPU for the regression benchmark (with this result being typical in EC-based GPGPU research). Given their hardware configuration (2005) of an AMD Athlon 2500+ CPU with 512 M RAM and an Nvidia GeForce 6800GT, the authors achieved speedups of 1.4× to 20.1× for genetic operators and 0.3× to 17.1× for

fitness evaluations for populations ranging from 1,024 to 262,144 for a regression benchmark.[3]

Fok et al. [4, 10] implemented EP (evolutionary programming) on the GPU. The individuals in a population are represented as textures on the GPU, as are fitness, random number, and indexing requirements. They determine 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). The authors achieved speedups of 1.25–5.02 for populations of 800–6,400 using five regression problems with a 2.4 GHz Pentium 4 with 512 MB of RAM and a GeForce 6800 Ultra video card. The lowest population size they tried, 400, did not improve in speed through the use of the GPU. GP (particularly Cartesian GP) is implemented by Harding and Banzhaf [11] using C# and Accelerator, with Accelerator handling the compilation of GP expressions into shader programs, execution of the shader programs, and the return of textures as array data. Fitness cases were evaluated in parallel on the GPU. Using sextic polynomial regression, the authors achieve speedups ranging from 0.02 to 95.37 testing combinations of maximum expression length {10, 100, 1,000, 10,000} and number of fitness cases {10, 100, 1,000, 2,000}. Chitty [12] implements a tree-based GP system, using OpenGL to create data textures and converting tree GP individuals to Cg shader programs for evaluation on the GPU. Langdon and Banzhaf [13] created a GPU-based interpreter using RapidMind and C++ that operates on stack-based GP trees. Their goal was to map a population of different individual programs to the GPU and evaluate the population in parallel.

Wilson and Banzhaf presented the first instance of parallel linear GP (LGP) using a graphics processing unit, deployed to PC and XBox 360, in [14]. In the previous work, fitness evaluation on the CPU maintained a loop over all fitness cases. Within that loop, the CPU iterated over n instructions, where n was the number of instructions in each individual. For each iteration of the fitness case loop, the shader on the GPU processed one instruction in all individuals in parallel, with the current values in each GP individual's registers passed in and out of the shader as single row texture. This method of evaluating the fitness allowed for the possibility of tracking the contents of registers, but was considerably more computationally expensive than the implementation described in detail in this work. The fitness function has since been optimized to include iteration over fitness cases and instructions in each individual for PC and iteration over all instructions (but not fitness cases) for the XBox 360. Furthermore, in [14] the authors did not implement fitness evaluation on the GPU of the XBox 360, but this is now accomplished in the current work (the previous work implemented only mutation using the XBox 360 GPU). Crossover is not implemented, as the authors felt that mutation provided sufficient variation to meet the goal of having implemented a working GP system on heterogeneous platforms. The authors wished to keep the algorithm as simple as possible in terms of computational time and memory overhead due to the programming of the unique

[3] While speedups in each study are strongly based on the hardware configuration, the results of the studies can provide a rough means of comparing the speedup of parallelization on the GPU over CPU only. Naturally, the proportion of the algorithm that is actually parallelized on the GPU will also affect these speedup measures.

hardware within the heterogeneous devices to which the implementation was to be ported.[4] This work describes the finalized algorithm, where the solution incorporates a new fitness function that allows for faster execution on the GPU. Once the fitness function was refined, differences in hardware prevented the authors from trivially porting the PC version of their GPU fitness function to the Xbox 360. To create a fitness function tailored to the Xbox 360 hardware, the authors implemented a number of changes to the fitness function component of the algorithm. Thus, both fitness and mutation can now be implemented on both the PC and XBox 360 platforms. This work represents a much more substantial explanation of the requirements for deployment on the PC and video game platforms than [15].

3 Linear genetic programming and its GPU parallel programming implementation

This first part of this section describes the general form of a linear GP (LGP) program. In particular, the instructions comprising a typical LGP individual and the genetic operator of mutation are discussed. The second part of this section provides details of how LGP is implemented for parallel processing using GPUs.

3.1 Linear genetic programming: brief overview

In Linear Genetic Programming (LGP), each individual is a sequence of instructions of an imperative programming language like C (or lower level languages like machine code). Usually, the instructions under evolution have a particular structure known to the evolutionary process: Each instruction consists of an operand, target, and two sources (each with an associated indicator component, or "flag"). A valid linear GP instruction takes the general form

$$\text{target} = src1\ op\ src2 \tag{1}$$

where it performs the operation *op* on values from either source registers or constant terminal inputs of the program represented by the sources (*src1* and *src2*), and places the result in the *target* register (see Fig. 1). Their respective flags determine whether the two source registers *src1* and *src2* refer to the source registers themselves or to the program inputs. Each instruction can therefore be represented by four integer values, *src1, src2, target, op*. The use of linear (bit) sequences in GP has been pioneered by N. Cramer with his JB language [16], and was later applied to machine language by Nordin et al. [17]. In recent years, a large number of LGP implementations have appeared [17–19]. Figure 2 shows a typical LGP program where the function set consists of arithmetic operators.

[4] In addition to the devices covered in this work, one of these devices was a portable media device (second generation 4 GB Zune), which involved only a sequential implementation and is not covered in this work since it was not a parallel implementation. See [15] for further details.

```
{ ...
  r[0] = r[5] + 11;
  // r[7] = r[0] - 62;
  r[4] = r[2] * r[0];
  // r[2] = r[3] + r[4];
  r[6] = r[4] * 21;
  r[1] = r[3] / 3;
  r[7] = r[6] - 5;
  r[0] = r[7] + r[1];
}
```

Fig. 2 Example of a typical linear GP individual. Instructions make use of registers r[0] to r[7] as sources and targets for operations. Operations here are arithmetic. *Double slashes* signal non-effective instructions if the final result is held in r[0]

Figure 2 shows an instantiated instruction set with each instruction consisting of a target, followed by an operation on two sources. In the first line of this instruction set, r[0] is the *target* of Eq. 1, "+" is the operator (*op*), r[5] is a value from the sixth internal register specified by *src1*, and 11 is a constant input value specified by *src2*. The value 11 is drawn from the program input for *src2* rather than drawing an input from an internal register due to the value of its accompanying flag. Below, we shall apply GP to a regression benchmark problem. In that example (a sextic polynomial), the integer variable *op*, $op = \{0, 1, 2, 3\}$, indicates one of four operators ADD, SUB, MUL, or DIV. Following Eq. 1, the integer variable *target*, $target = \{0, 1, 2, 3\}$, specifies one of four target registers. The integer variables *src1* and *src2* indicate either data from a fitness case or a register, based on the value of the respective flag variables *f1* and *f2* (not shown in generic instruction), which can have one of the values $\{0, 1\}$. The regression problem in this paper does not require control flow statements (conditionals and loops), and thus they are not encoded as possible instructions. However, general linear GP does allow the use of control flow statements simply through their inclusion in the set of operators (typically in addition to arithmetic operators). The control flow operators may also create general instruction forms in addition to Eq. 1.

The genetic operation applicable to LGP individuals in this work is mutation, where an instruction (single line of the LGP individual) is chosen using a random uniform distribution. The selected instruction then has each of its integer values changed to a random value within its acceptable range of values. There is no change in program size, as a single instruction is simply manipulated in place. Since all integer values within an instruction are mutated so they are within an acceptable range, no unfeasible instruction strings are generated.

3.2 GPGPU version of linear genetic programming

This work describes how to implement components of linear genetic programming in parallel on a graphics processing unit (GPU). To do this, a technique known as general purpose computation on GPUs (GPGPU) is used. GPUs typically consist of a number of processors that operate in parallel to perform "single instruction multiple data" (SIMD) processing where all processors on the GPU simultaneously execute the same program on different data. In particular, the GPU simultaneously

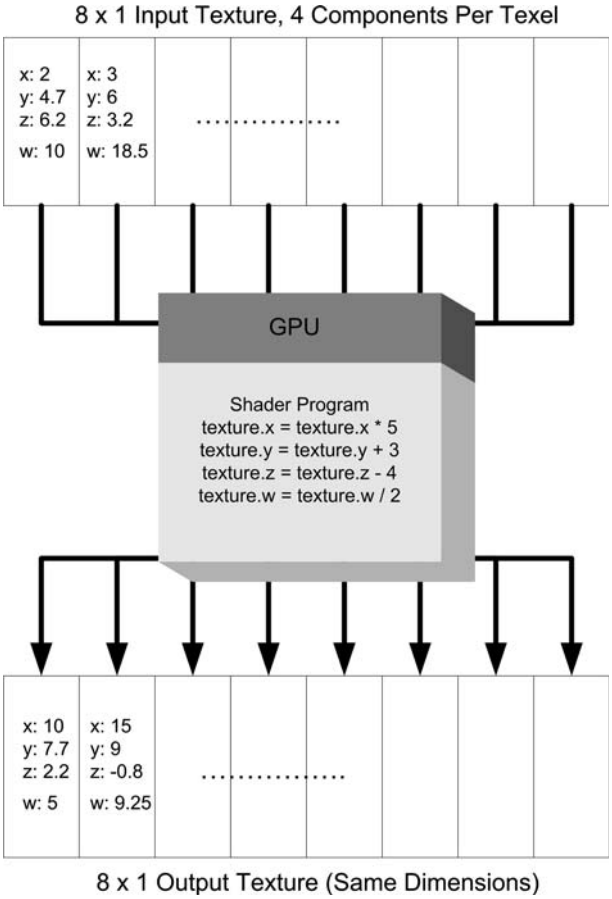

Fig. 3 Shader program for parallel execution using GPU. Each of the eight pixels of the input texture is processed in parallel by the GPU shader program. In the shader program, each particular channel across all pixels is adjusted using the same arithmetic evaluation. The results of the shader program on the input texture are shown as placed on the output texture

processes each pixel in the texture. Each pixel consists of four components (we use *xyzw* as the components, but *rgba* is equally as effective in GPGPU programming), see Sect. 2 for details on pixel components. The shader program on a GPU processes all components on every pixel at one time and outputs a vector of four floating point numbers for each pixel. Figure 3 summarizes the execution of a pixel shader program on the GPU.

In Fig. 3, each channel (*xyzw*) of each pixel in the input (top) 8×1 pixel texture are processed at the same time by the shader program on the GPU (darkened box, midde of Fig. 3). Thus, every x channel in every pixel is multiplied by 5, every y channel is increased by 3, and so on, at the end of the execution of the shader program on the GPU. The resulting texture that is output by the GPU is shown on the bottom of Fig. 3. In our LGP GPU-based implementation, the inputs to the GPU are either the individuals themselves or associated problem data to be represented as

textures. Numeric values corresponding to each segment of an instruction in an individual are stored as arrays in an XNA data type so they can be processed by the GPU as a texture. Our representation of an instruction uses four floats per pixel, one float for each of the 4 color components (*xyzw*) of the pixel. The fitness cases are stored on a separate (read only) texture object, and also use four floats per pixel. These texture-based data structures are described in greater detail in Sects. 5 and 6.

An instruction is encoded as two corresponding pixels. Two additional pieces of information are encoded on the pixels for convenience of reference when developing the program, but neither are required for execution: an integer *id* used to label the individual and an integer *PC (program counter)* to label the current instruction. There is no extra computational or space cost for adding this information—only 6 out of the 8 components across the 2 pixels used to represent an individual instruction are required for the instruction itself. Pixels of the first texture each contain the variables {*operator, target, id, PC*} corresponding to their four components and pixels of the second texture contain {*flag1, source1, flag2, source2*}.

The collection of pixel instructions make up an LGP individual, and the collection of individuals is the population. Accordingly, two textures can collectively represent the entire GP population: a particular column in both textures represents an individual, and the two pixels in the same location in both textures represent a unique instruction from the individual. The pixel-based width of these textures is the number of individuals in the population, and the pixel-based height of the textures is the number of instructions per individual. The length of all individuals (also the height of the population textures) is fixed at 16 for these experiments, where we found this length adequate to solve our chosen regression problem and did not wish to consume additional GPU texture memory. The representation of instruction, individual, and population is shown in Fig. 4.

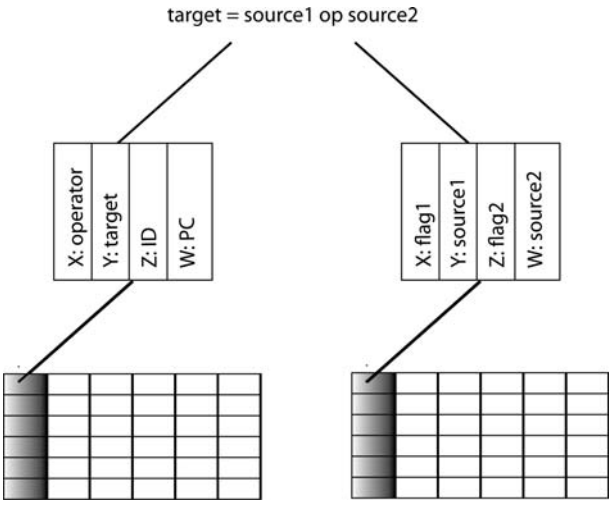

Fig. 4 Representation of pixel components as chromosomes, pixel pairs as instructions, column pairs as individuals, and two textures as the population

4 General linear genetic programming method for heterogeneous devices

The previous section described texture-based data structures suitable for use in parallel processing on a GPU, while the current section describes the general algorithm built to use these structures that is universal to the CPU and the GPU. Independent of CPU and GPU-related algorithm design considerations, there are practical hardware considerations when implementing the general GP algorithm on both PC and Xbox 360 platforms. These issues are discussed in Sect. 4.1. Section 4.2 then describes the general linear genetic programming algorithm that we aim to implement on both architectures (thus the algorithm is designed for the XNA framework) independent of CPU or GPU.

4.1 Architecture issues

The design decisions surrounding the way the GPU is used by the algorithm was determined by the authors based on the nature of the underlying hardware. GPU-based GP implementations can use the GPU in one of two ways: the GPU instructions can represent the instructions within an individual of the population, or the GPU instructions can represent an interpreter that executes the instruction set from an individual that is passed to it over all fitness cases. The former approach requires dynamic compilation of shaders; that is, shader program must be compiled and re-loaded to the GPU whenever selection occurs. In the latter approach, the GPU program is compiled and loaded only once; thus the GPU program is static (fixed) throughout program execution. Practitioners choose either approach based on problem implementation or preference. In the case of XBox 360 implementation, however, the choice was obvious because the authors were only successful at compiling the shader components during initial compilation of the entire program. Thus, we use the interpreter approach for the GPU for both the PC and XBox 360. As the design considerations are largely hardware dependent, the limitations described in this section will likely change in the future with new devices. An issue that prompted two separate implementations for the platforms was that for the Xbox 360, there was an added restriction enforced by the microcode compiler upon compilation of the shader program: when a texture was used to hold the fitness cases, referencing it from within the inner of two nested loops did not allow the shader to compile. Thus, iteration over the instructions of an individual in a loop nested within a loop iterating over fitness cases was not possible for our application on the video game console, but was possible on the PC. To attempt to maximize the respective parallelization capabilities of the platforms' GPU hardware, the authors allowed the general methodology to diverge in this respect. That is, to preserve the optimized PC GPU shader technique, separate implementations were required because the XBox 360 would not compile the authors' PC shader technique that used the nested loops. While it is possible to create platform specific programs using the XNA framework to form a single class for both platforms, the inability to compile the shader prevented that possibility for the authors' implementation. Despite the two slightly modified shader techniques and associated CPU fitness

evaluation initiating the shader techniques, the procedure for deployment is still considerably general (see following Sect. 4.2 for details).

Hardware architecture considerations that we discovered also affected the allowable parameters across implementations. To keep experiments more consistent, the most restrictive setting determined by the authors between the two platforms was used for both platforms, as determined by querying the hardware devices. Parameterization related to population size is affected in our implementation by the size of the video card backbuffer (memory area where the textures are drawn instead of the screen so they can be retrieved) limiting the texture size corresponding to the population. In particular, the width of the result texture is the number of individuals in the population. The XBox 360 used textures with pixel dimensions of up to 8,192 x 8,192. Our implementation's preliminary trials found that 400 individuals ran acceptably on the XBox 360, so the maximum population tested is thus 400. In addition to population-related GP parameters, the number of possible fitness cases is also impacted. The Xbox 360 features a maximum shader constant limitation of 256, so far the Xbox 360 implementation we restricted the number of fitness cases to 200 or under. Fitness cases were loaded into memory as an array rather than as a texture as on PC (due to microcode compilation issues associated with nested loops mentioned earlier in this section). The number of fitness cases on the PC is limited by the number of instructions that the shader is composed of following the unrolling of the nested fitness case and individual instruction loops. Unrolling of the loops is done by the shader program (HLSL) compiler and translates the instructions within all nested loops into a completely sequential version of those instructions to be used by the GPU. Even using shader options (such as preferring dynamic control flow) and directives to dynamically compile the shader, the authors found that the unrolling of the loops on the PC still occurred. Regardless, the preliminary tests using the XBox 360 prompted a limit of 200 fitness cases for our implementations.

4.2 General methodology

We implement linear GP on two different hardware platforms using a general methodology while adhering to the architectural requirements of both platforms mentioned in the preceding section. The framework used for programming both platforms is Microsoft's publicly available XNA Game Studio, where this work uses version 3.0. The authors were unable to get Microsoft Research's Accelerator, a tool for general purpose programming of the GPU, to operate with the XNA framework. The only other means of programming both platforms that the authors are aware of is Microsoft's XBox 360 Development Kits professional developer tools, which are generally restricted to established video game development companies [20]. These professional tools require special approval and licenses. Thus, we did not use these tools. However, a recent publication described the use of these professional tools to perform scientific computing using GPGPU programming on the XBox 360 in a medical application [21].

Programming of shaders using the HLSL shader language is used to provide GPU access using the XNA framework. CPU-side programming is with C# in Microsoft's Visual Studio 2008 development environment, where GPU shader programs are invoked using C# commands from the XNA framework. The shader programs themselves are programmed with Microsoft's HLSL. Implementation of LGP begins by creating a project using an XNA Framework Windows (or XBox 360) Game template. Upon creating the project with either template, the user will see two C# files containing a class: *Game1.cs* and *Program.cs*. *Program.cs* is a wrapper class for our purposes, containing only a *Main* method that begins a program (a game in the context of an XNA project). The Main method begins by creating an instance of the game class and invoking the *Run* method. The second file is called *Game1.cs* and contains the *Game1* class by default, which contains the methods of greatest interest to our implementation (and most other standard games). The class contains the methods *Initialize*, *LoadContent*, *UnloadContent*, *Update*, and *Draw*. The *Initialize* method is used to query services and handle non-graphics related content. The *LoadContent* method is automatically called only once per run and is used to initialize objects necessary to draw graphics, load effect files (files containing HLSL shader programs to be run), and load textures that will be drawn to the screen. *UnloadContent* is also automatically called once, and removes all loaded content. The *Update* method is used to check if a particular state is true during game play, including whether or not the user has pressed a certain key. The *Draw* method draws the current state of the game to the screen. The linear GP algorithm is programmed using the methods and functionality of the default *Game* class in a typical XNA framework project. Thus, in actuality, we are creating a Genetic Programming game that runs an example GP program and displays results on screen. The general GP method that we wish to implement in this framework is described in Table 1.

The first line of Table 1 simply reflects that we need to generate an initial population of LGP individuals. Lines 2–10 perform the entire GP algorithm for the desired number of generations. During each generation (iteration of the loop), each individual in the population has its fitness evaluated (loop of lines 3–9). For each iteration of the loop, a nested loop (line 4) iterates over the fitness cases (sets of one or more input values and one or more corresponding outputs) for the given problem. For each fitness case, the instructions in each individual are processed (loop of lines

Table 1 General linear genetic programming (LGP) algorihtm

```
1  generate population of individuals
2  do
3    for each individual in population
4      for each fitness case
5        for each instruction in instruction set
6          process instruction
7        accumulate raw fitness and hits
8    select individuals for next generation
9    apply genetic operators to unselected individuals
10 until maximum generations reached
```

5–6) and a measure of raw fitness (cumulative error difference between actual and provided answer for regression) and hits (number of acceptable errors) are tallied (line 7). Individuals to be copied for the next generation are selected using fitness-proportionate roulette wheel selection (line 8). Genetic operators (in this work, mutation only) are applied to individuals in the new population based on a threshold that is specified *a priori* (line 9).

To port the generic GP algorithm to XNA, each frame refresh by the automatic call of the Draw method processes a GP generation. The general method to accomplish this is the same for both platforms, and the generic method of GP implementation will be described here with differences described in the next section. Upon first executing the GP algorithm for either platform, the main method in the Program class mentioned previously creates an instance of the *Game* class and calls the internal method *Run()* for that class. The pseudocode C# commands for our use of the *Game* class to perform LGP are shown in Table 2.

When the constructor is called from the *Program* class, the program begins with the initialization of variables in the constructor (lines 3–4). In particular, any variables required to record data over multiple trials are created: seeds for each trial, and the arrays to record the best fitness, hits, and actual clock time elapsed for each trial. The *Initialize* method provides an on-screen keyboard for the user to input parameters. In the implementation discussed here, the user can specify: whether or

Table 2 C# pseudocode for the use of the XNA Game class to run LGP

```
1  class Game1
2    declare variables
3    GPGame()  //constructor
4      initialize trial-related variables
5    Initialize()
6      prompt user for required parameters
7    LoadContent()
8      initialize variables, arrays, textures
9    UnloadContent()
10   Update(GameTime)
11     Process initial program parameters from user
12     Check for exit key pressed on control pad
13   Draw(GameTime)  // evaluate individuals
14     *FITNESS EVALUATION (CPU or GPU)
15     if not at the end of a trial
16       fitness-proportionate selection
17     *MUTATION (CPU or GPU)
18     if at the end of a trial
19       trial++;
20       generation = 0;
21       add best fitness for output
22     if all trials are not yet done
23       display data, population texture
```

Instructions in all capitals indicate sections of the algorithm that can be run sequentially or in parallel using a GPU

not the sections of the algorithm that can be performed in parallel are to be sequential or in parallel (Boolean), population size (integer), mutation threshold (float) and number of generations in each trial (integer). The parameters accepted from the user are not central to the implementation of the algorithm, and additional flexibility could be implemented.[5] The *LoadContent* (lines 7–8) method is used to initialize declared variables, arrays for storing results of trials during execution, and textures that store graphical information to potentially be passed to the GPU for parallel processing (GPU processing chosen by user) or at least displayed to screen (CPU only processing chosen by user). It is this method that instantiates textures and associated XNA data structures that store the population (content of individuals), fitness cases, and thresholds for mutation. Thus, line 1 of Table 1 is performed in this method. *UnloadContent* (line 9) simply removes these variables from memory following completion of all trials. The *Update* method (lines 10–12), which takes the current game time as an argument, simply checks whether or not the user has pressed a key to exit the program and takes the initial parameters from the user.

The *Draw* method performs the creation of new generations; that is, the do loop (lines 2–10 of Table 1) begins iteration when the *Draw* method is implicitly called. The fitness evaluation (Table 2, line 14) incorporates the three nested loops in lines 3–7 of the general GP algorithm (Table 1). The task in Table 2, line 14 can be implemented sequentially on the CPU or in parallel on the GPU. In the CPU implementation, three nested loops (Table 1, lines 3–7) accomplish the evaluation of the fitness over the entire population. In the GPU implementation, the first loop over all individuals (Table 1, line 3) is avoided by processing all individuals at once as a texture using the GPU (detailed in Sect. 5). At Table 2, lines 15–16, if a particular trial has not yet ended, then fitness-proportionate generational selection proceeds using the CPU in the case of both sequential and parallel implementations. On line 16, mutation is applied (with an associated threshold). The mutation operator is described in detail in Sect. 6 and can be applied to each individual sequentially (on CPU only) or to all individuals in parallel (on GPU). Line 18 of the Draw method checks whether or not a particular trial has ended. If so, the current trial number is incremented, the current generation variable is reset, and the best fitness for the trial that has just ended is recorded in a growing list to be displayed on screen (lines 19–21). If all trials are not yet done, the list of best fitnesses of all trials elapsed so far, the number of the current trial, the current generation, and population textures are displayed to the user (lines 22–23).

5 Implementation of fitness function

In the previous section, the general LGP algorithm was described independent of deployment on the CPU or GPU. This section details the different CPU (sequential) and GPU (parallel) versions of the fitness function component of the general

[5] For instance, we always run 50 trials. Since the number of trials is controlled by the Draw method in the Game class, however, this is a natural addition to the user parameters.

algorithm (Table 2, line 14). A description of the sequential CPU fitness function is provided in Sect. 5.1. In particular, the structure and use of texture-type data structures available using XNA are explained as they pertain to the sequential fitness function. Moving the data in the relevant data structures of the CPU to textures, a parallel version of the fitness function can be implemented on the GPU of the PC or XBox 360. The parallel version of the fitness function for the PC is described in Sect. 5.2, with the particular implementation differences for the XBox 360 described in Sect. 5.3.

5.1 Sequential CPU fitness function

This section describes the sequential implementation of the fitness function; that is, the fitness function as it is implemented in a CPU-only execution of the implementation. The sequential implementation of the fitness function is the same for both the PC and XBox 360 platforms. The CPU version of the fitness function is the standard generally implemented in LGP, and only differs from the GPU implementation insofar as an array of fitness cases is used rather than referencing pixels on a texture in GPU memory. Typically, instructions would simply be binary strings if no GPU was used. For increased consistency across GPU and CPU implementations, the data structures are kept as similar as possible. Thus, the individual in the CPU-side implementation is still stored in two arrays of XNA *Vector4* types, which store appropriate float/integer values for each chromosome in groups of four (as they would be stored in pixels of four components). In addition, the fitness cases are stored in an array of the *Vector4* data type. The arrays are still placed on textures for display on-screen; however, they are simply not processed by the GPU during execution of the fitness function. The instructions for the CPU version of the fitness function just described are shown in Table 3, with arrays that could be placed on textures underlined.

The LGP fitness function, when applied to the whole population, consists of three nested loops: the outermost loop iterates over the individuals in the population, the middle loop iterates over fitness cases, and an inner loop iterates over the sequence of instructions comprising the individual. Line 1 of Table 3 is the loop over all individuals in the population (Table 1, line 3). Before executing the two inner loops, the hits and cumulative fitness variables are initialized (Table 3, lines 2–3). A loop over all fitness cases is then conducted (line 4) wherein the four registers of an individual that are used to store sub results are initialized (line 5). The inner most loop (line 6) then executes the instructions within each individual for each fitness case. For each instruction, the operator and target register for the instruction are retrieved from an XNA data structure that holds what would be transferred to the first of two textures representing the population in a parallel GPU implementation (line 7). Similarly, the flags and source locations are retrieved from the data structure that would be transferred to the second of two textures representing a

Table 3 C# CPU fitness function instructions for PC and XBox 360

```
1  for (int x = 0; x < populationSize; x++)
2    hits[x] = 0;
3    cumRawFitness[x] = 0;
4    for (int fitCase = 0; fitCase < fitnessHeight; fitCase++)
5      float[] registers = { 1.0, 1.0, 1.0, 1.0 };
6      for (int y = 0; y < instructions; y++)
7        read operator, target from popTexture1[x][y];
8        read flag1, flag2 source1, source2
9          from popTexture2[x][y];
10       if (flag1 == 1)
11         source1Value = fitnessCaseTexture[source1];
12       else   // flag1 == 0.0
13         source1Value = registers[source1];
14       if (flag2 == 1)
15         source2Value = fitnessCaseTexture [source2];
16       else   // flag2 == 0.0
17         source2Value = registers[source2];
18       if (operator == 0)
19         registers[target] = source1 + source2;
20       if (operator == 1)
21         registers[target] = source1 - source2;
22       if (operator == 2)
23         registers[target] = source1 * source2;
24       if (operator == 3)
25         if (source2 == 0) // avoid divide by 0
26           registers[target] = 1;
27         else
28           registers[target] = source1 / source2;
29     error = fitnessCaseTexture[result] - registers[0];
30     if (Math.Abs(error) < 0.01)
31       hits[x]++;
32     cumRawFitness[x] = cumRawFitness[x] + error;
33   record hits and cumRawFitness;
```

Fitness case and population arrays are underlined

population in the parallel implementation (lines 8–9).[6] If the value of a flag is 1, data is taken from the fitness case referred to by the first source location, denoted *source1* (line 10–11), else the value of the flag is 0 and data is taken from the individual's internal register denoted by *source1* (line 12–13). A similar conditional performs the same procedure for the second flag and *source2* data location (lines 14–17). The operator retrieved in line 7 can have the value of 1, 2, 3, or 4, which corresponds to the operations of addition, subtraction, multiplication, and division, respectively. The conditionals on lines 18–28 take the values in the two source locations, perform the specified operation, and place the result in the register corresponding to the

[6] Lines 7–9 indicate that a two-dimensional array is being accessed for clarity. In actuality, a one-dimensional array is treated conceptually as a two-dimensional array and is accessed using offsets such that indices (x, y) are the index $(x + (y * \text{populationWidth}))$. A one-dimensional array must be used in order to place data on an XNA Texture2D object to be passed to the GPU.

target location from line 7. The division operator is protected in the case of divide by 0, where the function will place a value of 1 in the target register. Exiting the loop after processing the individual's instructions, the difference (error) between the desired value for the particular fitness case and the current value in register 0 is calculated (line 29). If the error is within an acceptable threshold (0.01), the number of hits is incremented by one (lines 30–31). The current error is then added to the cumulative error (line 32). Exiting the loop over all fitness cases, the hits and cumulative raw fitness are recorded for each individual (line 33) within the loop over all individuals (line 1).

5.2 Parallel GPU fitness function

This section describes the parallel implementation of the fitness function for the GPU on the PC, where changes to the technique discussed here for the XBox 360 are provided in the next section. We use textures on the GPU to represent fitness cases and the population, where each pixel (the name of a pixel when placed on a texture) has four components. In the case of the fitness texture, we model a regression problem (detailed in Sect. 7) and actually only require two components of a four component pixel: In this work, a fitness case corresponds to a single input (x variable) and a corresponding output (y variable) for the equation $y = x^6 - 2x^4 + x^2$ used as the regression problem. A texture of 200 fitness cases was used, so a 2×200 array was required. Since two values can fit within two components of a four component pixel, a texture of 1×200 pixels can be used. The two population textures took the form shown in Fig. 4, where each texture had a width of the population size and a height of the number of instructions per individual. In each pixel of both population textures, all four components are used to store information as detailed in Sect. 3: In the first texture each pixel contains target register, operator, individual ID number, and program counter (only the former two components are segments of an instruction). In the second texture, each pixel contains two flag and source pairs, namely (*flag1*, *source1*) and (*flag2*, *source2*). To place the data on each texture, the XNA *HalfVector4* surface format was used. In the *HalfVector4* format, each of the four components was a 16 bit float (interpreted as an integer where appropriate). This choice of surface format attempted to maximize the Xbox 360 GPU hardware capabilities while increasing the speed of execution through placing more information on single textures.

The GPU version of the fitness function is written in High Level Shader Language (HLSL). In order to execute the GPU fitness function, the 1×200 fitness case texture and two population textures are loaded into GPU memory by the CPU-side classes. Raw fitness and hits are returned by the GPU each time the shader program fitness function is executed. In HLSL, pixel components are stored in *float4*s, which store four floats as components of a single *float4* variable, and are here specified using *xyzw*. The only output is the cumulative fitness (cumulative raw error) and associated hits (fitness cases with acceptable error) across all fitness functions at the end of execution; no intermediate results for individual instructions or even individual fitness cases are available to the calling program during its execution. The final output of the shader program is a pixel (on-screen pixel) for

each individual, with a component taken for hits and another component for raw cumulative fitness over all cases. Thus, a texture is produced by the shader with dimensions *populationSize* × *1* where each pixel is the result of program execution over all fitness cases stored as {*hits*, *raw fitness*, *n/a*, *n/a*}. Pseudocode of the shader fitness function is shown below in Table 4.

Table 4 HLSL GPU fitness function instructions for PC

```
1   float4 fitness (float2 location : TEXCOORD0) : COLOR
2     int hits = 0;
3     float cumulativeFitness = 0;
4     for (int fitCase = 0; fitCase < 200; fitCase++)
5       fitnessData = fitnessCaseTexture pixel at fitCase;
6       registers = float4(1, 1, 1, 1);
7       for (int pointer = 0; pointer < instructions; pointer++) {
8         opTargetData = popTexture1 at (location, pointer);
9         flagSourceData = popTexture2 at (location, pointer);
10        if (flagSourceData.x == 1.0)
11          src1Index = fitnessData.x;
12        else     // flagSourceData.x == 0, fetch from register
13          src1Index = flagSourceData.y;
14          if (src1Index == 0) src1 = registers.x;
15          if (src1Index == 1) src1 = registers.y;
16          if (src1Index == 2) src1 = registers.z;
17          if (src1Index == 3) src1 = registers.w;
18        if (flagSourceData.z == 1.0)
19          src2Index = fitnessData.x;
20        else // flagSourceData.z == 0, fetch from register
21          src2Index = flagSourceData.w;
22          if (src2Index == 0) src2 = registers.x;
23          if (src2Index == 1) src2 = registers.y;
24          if (src2Index == 2) src2 = registers.z;
25          if (src2Index == 3) src2 = registers.w;
26        float result;
27        if (opTargetData.x == 0) result = src1 + src2;
28        if (opTargetData.x == 1) result = src1 - src2;
29        if (opTargetData.x == 2) result = src1 * src2;
30        if (opTargetData.x == 3)
31          if (src2 == 0) result = 1;
32          else result = src1 / src2;
33        if (opTargetData.y == 0) registers.x = result;
34        if (opTargetData.y == 1) registers.y = result;
35        if (opTargetData.y == 2) registers.z = result;
36        if (opTargetData.y == 3) registers.w = result;
37      if (abs(registers.x - fitnessData.y) < 0.01) hits++;
38      cumulativeFitness = cumulativeFitness +
39        abs(registers.x - fitnessData.y);
40    registers.x = hits;
41    registers.y = cumulativeFitness;
42    return registers;
```

Fitness case and population arrays are underlined and are loaded in GPU memory prior to execution

The HLSL shader for fitness evaluation in Table 4 begins with the assumption that the fitness case and population textures are present in GPU memory after being loaded from CPU-side execution. The underlined texture names of the fitness case texture and two population textures in Table 4 correspond to the identically named array structures in Table 3, and serve the same function for both implementations of the fitness function. (The shader commands associated with loading and reading data from particular sections of the textures, called "sampling," has been removed for brevity.) The *fitnessShader* program then involves the declaration of hits and cumulative fitness variables. Rather than requiring a loop over all individuals in the population, the parallel fitness function processes every individual (every column) on the population textures at once. The outermost loop in the sequential fitness function (Table 3, line 1) is thus not applicable. The first for loop of the parallel fitness function (Table 4, line 4) iterates over the fitness cases, resetting the value of the registers each time. The appropriate location for the current fitness case on the fitness texture in memory is retrieved in the first step of the loop body (line 5, HLSL texture sampling commands removed for brevity). For each fitness case, prior to iteration over an individual's instructions, registers are re-initialized (line 6). The HLSL variable consisting of four components (*float4* using components *xyzw*) is used to store the four register values.

The inner loop, using the pointer variable, then executes the individual's instructions sequentially (line 7). The appropriate location of the pixels on both population textures for the current instruction are read to retrieve four components (*xyzw* in HLSL) per pixel. For each instruction, the first flag is checked (*flagSourceData.x* from the *flagSourceData* texture) prior to retrieving the data for the first source (*src1*) of the general instruction form in Eq. 1 (*target* = *src1 op src2*). If the value of the first flag (*flagSourceData.x*) is 1, then fitness information is fetched from the input value for the current fitness case on the fitness texture (*fitnessData.x*) in lines 10–11, else a subresult is fetched from one of the four registers (contained in the four *xyzw* components of *registers*) indexed by the value in *flagSourceData.y* (lines 12–17). Following that, the second source data (*src2*) is loaded in a similar manner by checking the second flag from *flagSourceData.z*, and then loading data from the register index indicated by *flagSourceData.w* or from fitness cases, based on the second flag (*flagSourceData.z*) in lines 18–25. The operation specified by the component *opTargetData.x* (*op* in *target* = *src1 op src2*) is mapped to the appropriate mathematical operator (lines 27–32), with the result being placed in the *target* register specified by the component *opTargetData.x* (lines 33–36). The inner loop is then complete, and the raw (absolute) cumulative fitness over all fitness cases is computed, as is the number of hits, prior to the end of the outer loop over all fitness cases (lines 37–39). The hits and cumulative fitness results are transferred to the first two components (floats) of the *float4* variable registers (no longer used to store subresults at this point), which is the variable returned by the shader program to the CPU-side program as a texture with height of one and width of the population size.

The CPU-side C# pseudocode used to call the GPU fitness function is given in Table 5 below. The first line creates an Effect object that is used as the intermediary to set parameters in the GPU shader program prior to its execution (line 1). Line 2 changes the render target so that information will be written to the GPU memory

Table 5 C# CPU fitness function instructions for PC to invoke GPU shader

```
1  create effect object
2  switch render target
3  set effect population textures (x2) parameters
4  set effect population size integer parameter
5  set effect fitness case texture parameter
6  create spriteRenderer object
7  spriteRenderer.Begin();
8  effect.Begin();
9  spriteRenderer.Draw(resultTexture, location, Color.White);
10 spriteRenderer.End();
11 effect.End();
12 switch render target
13 resultTexture = renderTarget.GetTexture();
14 resultTexture.GetData<HalfVector4>(resultTextureData);
```

rather than the screen. The required textures (two population textures, an integer reflecting population size, and the texture containing fitness cases) are passed to the shader program as parameters (lines 3–5). An object called a *spriteRenderer* (line 6) is used to draw the textures. In combination with the effect object that allows access to the shader, the spriteRenderer is used to execute the shader program by drawing the specified textures (lines 7–11). Once the shader program has finished execution (*effect.End*(), line 11), the render target is resolved (by switching the render target again, line 12). The data is retrieved from the texture written to by the shader by first retrieving the texture from GPU memory and storing it in a Texture object (line 13). The data stored in this texture is then retrieved using the *GetData* function (line 14) and placed in a data structure. The hits and cumulative raw fitness are then read from the appropriate components of the retrieved texture.

5.3 Parallel GPU fitness function for XBox 360

For implementation on the Xbox 360, the authors encountered an added restriction enforced by the microcode compiler upon compilation of the shader program: when a texture is used to hold the fitness cases, referencing it from within the inner loop of two nested loops did not allow the shader to compile (Table 4, lines 11 and 19). To avoid this compiler issue, the fitness cases were passed to the shader program as an array of *float2*s in GPU memory. Each *float2* variable consists of x and y components, which are naturally used for the input and output variables for each fitness case. For other fitness case scenarios involving more variables, a number of XNA *float2* or *float4* arrays could be passed to the shader. Upon implementing a usable shader technique otherwise similar to the PC, the authors found that the Xbox 360 GPU did not persist texture and/or parameter contents in the same way as the PC. The authors found that for their implementation of the fitness shader, the GPU memory would flush texture content before it could be read back by the CPU.

A means of keeping the information written to the GPU memory so that it could be read back by the CPU-side program led to the authors implementing a number of changes to the shader file (where the suitable changes were found to be documented

in the MSDN online documentation [22]). One of the changes was to the variable that handles drawing to the current location: it can be specified as a SPRITETEXCOORD to allow the drawing of point sprites through the *DrawPrimitives* method rather than sampling a texture using the standard TEXCOORD semantic as on the PC. The *DrawPrimitives* method was used to handle particular aspects of writing to the GPU memory on the XBox 360, which was handled by the *SpriteBatch.Draw*() method for the PC: The data, once written to the GPU memory using the latter method on the PC, was resolved and read back by the CPU-side (C#) program. On the Xbox 360 (unlike the PC), when an attempt was made to resolve the texture produced, the intended data written was not retrieved. To expand on the implementation of the fitness function for the XBox 360, the means of persisting the data written to the Xbox 360 GPU memory that the authors chose was to use a *DrawUserPrimitives* method called directly from the XNA object representing the graphics device rather than using *SpriteBatch.Draw*(). The *DrawUserPrimitives* method draws the data as scaled point sprites rather than a texture of pixels. That is, the data in *HalfVector4* format was drawn as a list of points. The version of the *DrawUserPrimitives* method used accepts as arguments the array of *HalfVector4*s to be drawn, a starting pixel on the backbuffer at which to draw (0), and the number of data points to be drawn (length of the array) in the point sprite. When initializing the drawing method, the *PointSize* (in pixels) is to be initialized to the size of the population so there is a pixel drawn for each member of the population. In addition, what is passed as x, y points to the shader get moved to points z, w and may become negative [23]. To correct for this, the absolute values of the z, w coordinates on the GPU are taken as the intended x, y coordinates. The resulting Xbox 360 HLSL shader fitness function pseudocode is shown in Table 6.

The HLSL shader program for the PC (Table 4) differs from the shader for the XBox 360 (Table 6) due to the hardware changes noted in this section. There are only two textures declared to be sampled, with the fitness texture passed directly as a variable parameter in the form of an array of 200 *float2*s (line 1). This array of *float2*s for the fitness cases consists of one input and one output for the sextic polynomial equation we chose. The fitness function declaration is similar to that of the PC, only using SPRITETEXCOORRD as mentioned previously (line 2). The first line of the fitness function corrects for the peculiar change in z, w components and possible change in sign (line 3). Inside the outer loop over fitness cases (line 4), the *fitnessData* information is now retrieved from the components of the elements of

Table 6 HLSL GPU fitness function instructions for XBox 360

```
1  float2 fitnessArray[200];
2  float4 fitness(float4 inLocation : SPRITETEXCOORD0 ) : COLOR
3  float2 location = abs(inLocation.zw);
4  for (int fitCase = 0; fitCase < 200; fitCase++) {
5    registers = float4(1, 1, 1, 1);
6    fitnessInput = fitnessArray[fitCase].x;
7    for (int pointer = 0; pointer < instructions; pointer++) {
8      opTargetData = popTexture1 at (location, pointer);
9      flagSourceData = popTexture2 at (location, pointer);
10     . . . as in Table 4
```

Table 7 C# fitness function instructions for XBox 360 to invoke GPU shader

```
1  create effect object
2  switch render target
3  set effect population textures (x2) parameters
4  set effect population size integer parameter
5  set effect float2[] fitnessArray parameter
6  create spriteRenderer object
7  spriteRenderer.Begin();
8  effect.Begin();
9  spriteRenderer.GraphicsDevice.
10   DrawUserPrimitives<HalfVector4>(PrimitiveType.PointList,
11   resultTextureData, 0,
12   resultTextureData.Length);
13 spriteRenderer.End();
14 effect.End();
15 switch render target
16 trasfer texture from video card to texture object;
17 texture.GetData();
```

Differences from PC instructions to initiate GPU shader are italicized

the *float2* array *fitnessArray* rather than from a texture (line 6). The CPU-side C# instructions used to call the HLSL shader in Table 6 are shown in Table 7.

In contrast to Table 5 (PC shader program invocation), the XBox 360 version of the C# instructions in Table 7 must populate an array of *float2*s with the fitness cases prior to executing the shader. Other than that, the C# instructions differ from the PC in that fitness cases are passed to the shader as the array parameter *fitnessArray* (Table 7, line 5) rather than by passing a texture (Table 5, line 5). Finally, the *DrawUserPrimitives* method must be used to draw the data itself (stored in *resultTextureData*) directly to the GPU memory buffer rather than drawing a texture (Table 7, lines 9–12) to the GPU memory using the *Draw* method of *SpriteRenderer* as for the PC (Table 5, line 9). An important parameter to note for the *DrawUserPrimitives* method (Table 7, line 10) is the last one, *resultPopulationTextureData.Length* (Table 7, line 12), which allows the proper scaling of the output to the size of the population so there is a four component pixel result drawn for each individual in the population. If the scaling is not correct, the result values were found to be either nonsense or blank (default color).

6 Implementation of the mutation operator

Following the execution of the fitness function on either the CPU or GPU, roulette wheel fitness-proportionate selection is performed on the population using the CPU (see Table 2, lines 15–16). The mutation operator is then implemented (Table 2, line 17), which is described in this section. In particular, linear GP micro mutation (mutation occurring within an instruction) is implemented where any of the particular chromosomes of an instruction can be replaced with a new acceptable value for that chromosome. In other words, mutation occurs at the level of symbols rather than at the lower level of binary encoding. The GPU

implementation of mutation in LGP is much more straightforward than GPU-based fitness evaluation: there are no nested loops to complicate compilation of the shader, and particular areas of textures are not evaluated at different times. All pixels on the three mutation textures are evaluated at the same time by the GPU, and both CPU and GPU mutation implementations can be accomplished in the same way for PC and XBox 360. Mutation involves three new textures generated by the CPU, all of which have width corresponding to individuals in the population and height corresponding to the number of instructions per individual. These first two of the three textures contain acceptable potential replacement values for each chromosome for the two textures that currently make up the population (as described in Sect. 3). The first of the three textures contains pixel information of $\{op, target, id, PC\}$ and a second texture contains pixels of $\{f_1, src_1, f_2, src_2\}$ using values chosen from a normal random distribution. The third texture used for mutation has the four components of each pixel initialized to a set of four floats chosen with a random uniform distribution in the range [0.0, 1.0]. This texture contains values that correspond to a single float mutation threshold that is determined in each tournament round. If the mutation threshold exceeds the value in a pixel's component of this threshold texture, the chromosome on both of the two original population textures in that component's position will be replaced. Five textures altogether are involved in mutation: the two textures representing the original (pre-mutation) population of individuals, a texture of values corresponding to a mutation threshold, and two textures containing potential replacement chromosomes for the two original population textures. For mutation using only the CPU, the data were held in an array rather than a texture. The textures/arrays used for mutation and the general method are shown in Fig. 5.

The GPU-side mutation shader operates by using the two textures representing the current population (bottom two textures in Fig. 5). For each component (*xyzw*) of each pixel on the current population textures, the shader program checks the corresponding component from the same pixel location on the threshold texture (containing a value in the range [0.0, 1.0]). If the threshold texture (middle black texture, Fig. 5) component value does not exceed the mutation threshold, the value in the corresponding pixel component in the potential replacement texture (middle grey texture pair, Fig. 5) replaces the value in the component of the current population, otherwise the current population component value is preserved. In other words, the threshold texture serves as a mask, where values in the threshold texture not exceeding the mutation threshold value allow new values for instruction chromosomes, otherwise the current chromosome values are preserved. The resulting mutated population is then the new GP generation (top textures, Fig. 5). The mutation technique as implemented on CPU and GPU is described in Sects. 6.1 and 6.2, respectively.

6.1 CPU mutation for PC and XBox 360

The CPU version of the mutation operation, to keep the CPU and GPU versions as similar as possible, uses the same data structures as GPU (current, replacement, and

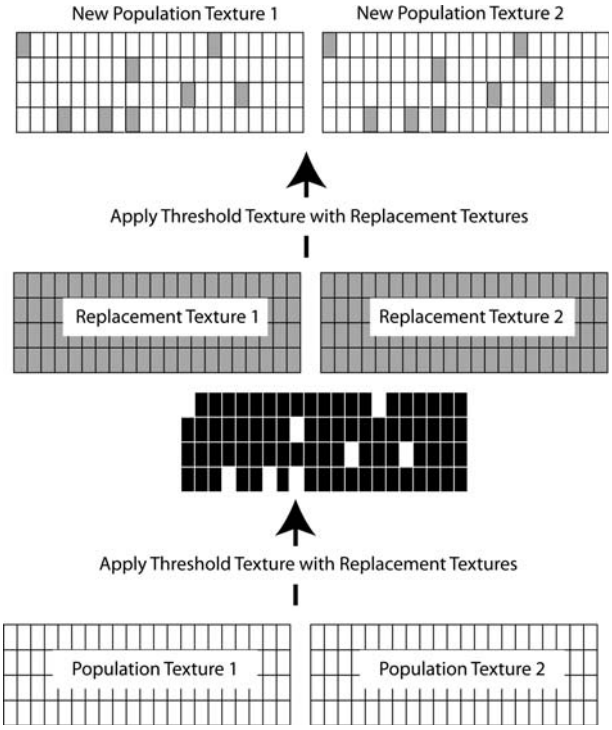

Fig. 5 Representation and process of mutation

threshold arrays prior to being placed on textures). However, the elements of the arrays cannot obviously be processed in parallel as on a GPU. Therefore, a nested loop is required to iterate over the arrays to accomplish mutation, using the CPU-side C# instructions in Table 8.

The above instructions use the set of four components in each data array element {*x, y, z, w*} to hold the values for the three textures required for mutation. Two-dimensional arrays hold the components of each instruction in each array element: {*op*, *target*, *id*, *PC*} in each element of *populationTexture1Data* (and its

Table 8 C# CPU mutation instructions for PC and XBox 360

```
1  for (int x = 0; x < populationSize; x++)
2    for (int y = 0; y < totalInstructions; y++)
3      if (thresholdTextureData[x][y].X <= mutationThreshold)
4        populationTexture1Data[x][y].X
5           = replacementTexture1Data[x][y].X;
6      if (thresholdTextureData[x][y].X <= mutationThreshold)
7        populationTexture2Data[x][y].X
8           = replacementTexture2Data[x][y].X;
9      repeat conditionals on lines 4 - 8 for Y, Z, W
```

corresponding potential replacment texture *replacementTexture1Data*) and $\{f_1, src_1, f_2, src_2\}$ in each element of *populationTexture2Data* (and its corresponding potential replacement texture *replacementTexture2Data*). The instructions above iterate over all individuals (line 1), and all instructions in those individuals (line 2), mutating each of the eight chromosomes of the instruction (four chromosomes on each texture for every instruction) if the matching component in the replacement array is not over the user-specified mutation threshold (lines 3–9).

6.2 Parallel GPU mutation for PC and XBox 360

The GPU implementation of the mutation operator allows the mutation of all chosen chromosomes at once by processing all pixels in parallel. The shader program is applied to every pixel of the current population texture at the same time, checking the four component values on the corresponding threshold texture and replacing each component of every pixel of the current population texture with the corresponding replacement texture component value based on the threshold texture values. The current population texture is passed to the shader for processing, so the information in a component of the original population is preserved by default—if the shader does not alter the component information, it is preserved. The HLSL instructions for the mutation operator on the GPU are shown in Table 9, with corresponding CPU-side C# instructions used to invoke the shader shown in Table 10 for completeness.

Table 9, line 1 is the declaration of the shader program for the mutation operator. The texture that is passed to the shader program is the current population texture, where the parallel mutation operator in Table 9 is applied to both population textures separately. The shader function is passed the current population texture (line 9, Table 10), so all pixels in that texture are handled simultaneously by the shader program. In particular, *currentLocation* individually refers to all pixels at once (Table 9, line 1). The corresponding pixel at the identical location on both the threshold texture and the replacement population texture are retrieved in Table 9, lines 2–3. For the threshold pixel, all components are checked against the user-defined mutation threshold (Table 9, lines 4, 6, 8, 10). If the threshold has not been exceeded, the component is subject to mutation and the component being checked on the pixel of the current population is replaced with that component on the pixel

Table 9 HLSL GPU mutation instructions for PC and XBox 360

```
1  float4 mutateShader(float2 currentLocation : TEXCOORD0): COLOR
2    threshold = thresholdTexture at currentLocation;
3    potentialReplacement = replacePopTexture at currentLocation
4    if (threshold.x <= mutationThreshold)
5      currentPop.x = potentialReplacement.x;
6    if (threshold.y <= mutationThreshold)
7      currentPop.y = potentialReplacement.y;
8    if (threshold.z <= mutationThreshold)
9      currentPop.z = potentialReplacement.z;
10   if (threshold.w <= mutationThreshold)
11     currentPop.w = potentialReplacement.w;
12   return currentPop;
```

Table 10 C# GPU mutation instructions for PC and XBox 360

```
1  create effect object
2  switch render target
3  set effect mutation threshold float parameter
4  set effect mutation threshold texture parameter
5  set effect replacement population texture parameter
6  create spriteRenderer object
7  spriteRenderer.Begin();
8  effect.Begin();
9  spriteRenderer.Draw(currentPopTexture, location, Color.White);
10 spriteRenderer.End();
11 effect.End();
12 switch render target
13 currentPopTexture = renderTarget.GetTexture();
14 currentPopTexture.GetData<HalfVector4>(currentPopTextureData);
```

of the replacement population (Table 9, lines 5, 7, 9, 11). The texture returned by the GPU shader program thus retains all components on all pixels of the original population where a threshold component did not fall within the user mutation threshold, otherwise a mutated value replaces it from the corresponding component of the replacement texture. The completely mutated texture is returned at the end of the shader program (Table 9, line 12).

Table 10 shows the CPU-side instructions used to execute the shader. The implementation is largely the same as described for the PC version of the C# instructions for GPU fitness in Table 5. Initially, the render target is switched from the screen to the backbuffer (line 2), and values are provided to the shader in terms of float mutation threshold (line 3), the texture of threshold values (line 4), and texture with potential replacement information (line 5). The texture containing the original population texture to be mutated is passed as an argument to the shader on Table 10, line 9. The remainder of the C# instructions is executed similarly to the GPU fitness shader described in Table 5: An effect executes the shader program when the *Draw* method from the *SpriteRenderer* is used (lines 6–11). Following that, the texture is resolved (moved from GPU memory) and the data is retrieved from the texture (lines 12–14).

7 Results

In preceding sections of this paper, we described how to implement a general LGP algorithm for heterogeneous devices. In particular, we described implementations that maintained a balance of attempting to use the underlying hardware while maintaining a degree of generality of the algorithm. In this section we discuss the resulting interface for the heterogenous devices when the implementation is deployed (Sect. 7.1) on both PC and XBox 360. Empirical results comparing the performance of the CPU and GPU-based LGP implementations targeted for the two platforms are provided in Sect. 7.2.

7.1 Visual interpretation of results

The GP system screen output described in this section is the same for all platforms, whether or not implementations are parallel: both PC and Xbox 360 deployments, whether CPU only or GPU, display the same interface. The program begins by asking the user to specify whether to run fitness and mutation on the CPU or GPU, population size, mutation threshold, and number of generations. Once the user has entered the parameters, the best results for each of 50 trials are displayed at the bottom of the screen. As each trial is performed, the current state of the population and mutation textures is displayed on the top portion of the screen. The current state of the GP population at a particular generation and ongoing results of trials are displayed to the user as shown in Fig. 6.

From top to bottom of the screenshot in Fig. 6, the first two textures collectively represent instruction (texture row) over the population (each individual is a texture column). Thus, as described in Sect. 3, both textures have a width of the number of individuals in the population and a height corresponding to the number of instructions in an individual. Furthermore, the four components in each pixel correspond to the chromosomes of an instruction. These two textures will become composed of horizontal bands as the population converges toward a solution. Also, each pixel of the two populations consist of four of the chromosomes required to make up an instruction as described in Sect. 3: In the first texture, the components of a pixel specify a mathematical operator, the target register, and identification of current individual and instruction (called "pointer" on display). The latter two chromosomes were used internally and do not reflect evolution of the population. In the second texture, a flag component (corresponding to source 1) indicates whether information should be retrieved from internal registers or the current fitness case. A source component (corresponding to source 1 of Eq. 1) then determines from which of the internal registers/fitness cases the data should be retrieved. A flag and source component are similarly provided for source 2. The next texture displayed contains raw fitness (cumulative error over all fitness cases) and hits (number of fitness cases with acceptable error level) for each individual following evaluation of all fitness cases as described in Sect. 5. Since this information is contained in a single pixel, the texture is one pixel high with a width of the population size.

The current mutation threshold texture is then displayed, containing a float value from [0…1] in each component of each pixel. The threshold texture is a graphical representation of the mask used with the potential population replacement values located in the fifth and sixth textures as described in Sect. 6. That is, particular components of the pixels in the fifth and sixth textures could replace the component values of the pixels in the first and second textures representing the current population to form the next generation of population textures. Results at the bottom of the screen display numerically the best number of hits, best cumulative raw error over all fitness cases, and system clock time elapsed in seconds for each trial. These results were displayed and recorded from the screen, since the authors are not aware of a mechanism in XNA to export saved data from the XBox 360.

Fig. 6 GP implementation on the PC and Xbox 360. Textures at a given generation are displayed on the screen using 50 fitness cases for demonstration purposes. Individuals can contain 16 instructions and a population size of 200 is used. Textures displayed, from *top* to *bottom*, are: population textures (×2), resulting hits and raw fitness for each individual, mutation thresholds mask, and potential replacement chromosomes for the population (×2). In the lower portion of the screen, results indicate best hits, best raw error, and time (s) for each of 50 trials

7.2 Quantitative results

A popular GP regression problem was used to compare the performance of GPU and CPU versions of the implementation on the PC and Xbox 360 using XNA Game Studio 3.0. The CPU implementation adopted all shader functionality using C# instructions as described in Sect. 6. The sextic polynomial $x^6 - 2x^4 + x^2$ introduced by Koza [24] was implemented using float inputs in the range [0, 1] for 200 fitness cases. Individuals consisted of 16 instructions each. Four operators of addition, subtraction, multiplication and division comprise the function set. Fitness proportionate roulette wheel selection is used, with each trial consisting of 50 generations. Mutation occurs with a threshold of 0.1. A successful hit for a fitness case is a result within 0.01 of the actual answer. All implementations were successful at achieving a large proportion of hits over all fitness cases in all trials, and it is well documented that most GP systems should be able to readily solve the

Table 11 GP regression parameters

Function set	ADD, SUB, MUL, DIV (on floats)
Fitness	Fitness-proportionate roulette wheel
Population	100, 200, or 400 individuals
Mutation	Threshold = 0.1
Generations	50
Number of trials	10
Fitness cases/constants	200 cases: $x = [0, 1]$, $y = x^6 - 2x^4 + x$
Fitness metric	Number of hits, where a hit is Absolute(Reg[0] $- y$) ≤ 0.01

sextic polynomial equation. Since the focus of the experiments was to measure performance of the sequential and parallel implementations combined with underlying hardware, the quality of solutions to the sextic polynomial is not discussed here. However, results indicate that all implementations were able to generate solutions to the problem. Parameterization of the GP is summarized below in Table 11.

Experiments were conducted using Windows XP SP2, using an AMD Athlon 64 Processor 3500+ (2.21 GHz), 1024 MB of RAM, and an ASUS EN8800GTX video card using an nVidia GeForce 8800 GTX GPU. The nVidia GPU features 128 parallel stream processors with unified shader architecture. The Xbox 360 features a custom built IBM PowerPC-based CPU with three 3.2 GHz core processors. The Xbox GPU by ATI houses 48 parallel shaders with unified architecture and 10 MB of embedded DRAM (EDRAM) [25], with 512 MB of DRAM as main memory. The CPU and GPU of the Xbox 360 are customized for graphics-intensive computation, with the GPU able to read directly from the CPU L2 cache. A comparison of the execution performance of PC using CPU only, PC using GPU, Xbox 360 using CPU, and Xbox 360 using GPU is given in Fig. 7. Performance is given by the number of genetic programming operations that occur per second in Fig. 7, with actual mean execution times provided in Fig. 8 to provide practical context. Figure 9 shows the speed up in computing time of parallel execution over sequential execution for both PC and XBox 360 platforms. As the standard error bars and precise means are difficult to discern in each figure, these measures are provided in Tables 12, 13, 14 beneath each graph.

It should be noted that the Xbox 360 (released 2005) is not meant to be directly compared to a PC with a different processor, twice as much RAM, and housing a GPU with 128 processors for parallel processing (as opposed to its 48 processors). The hardware platforms are only used to determine general trends in their respective CPU and GPU implementations, which we now discuss. Examining the standard error in Figs. 7 and 8, it is noteworthy that there is very little variation across multiple trials. In Figs. 7 and 8 it is evident, as is common in genetic programming GPU literature, that a significant performance acceleration is gained by using the GPU for parallel processing over CPU for all populations on the PC. In Fig. 9 (and Table 14), we see that this speed up increases with population from 3 times to over

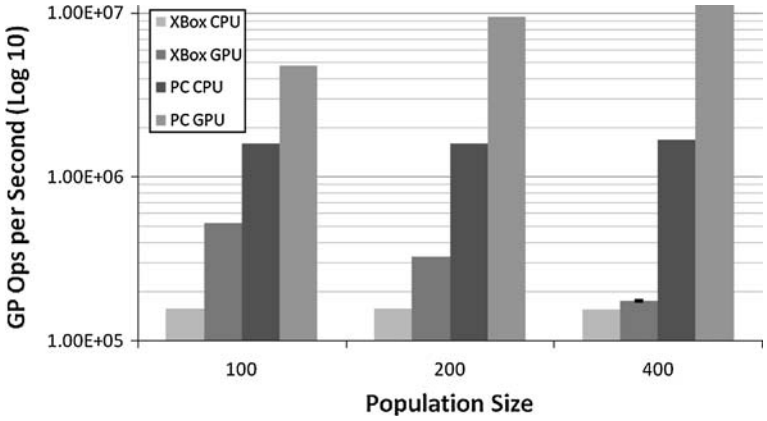

Fig. 7 Xbox 360 and PC performance in GP Ops/s ($\log_{10}$ scale) with standard error, based on 10 trials of 50 generations with populations of 100, 200, and 400 using sextic polynomial

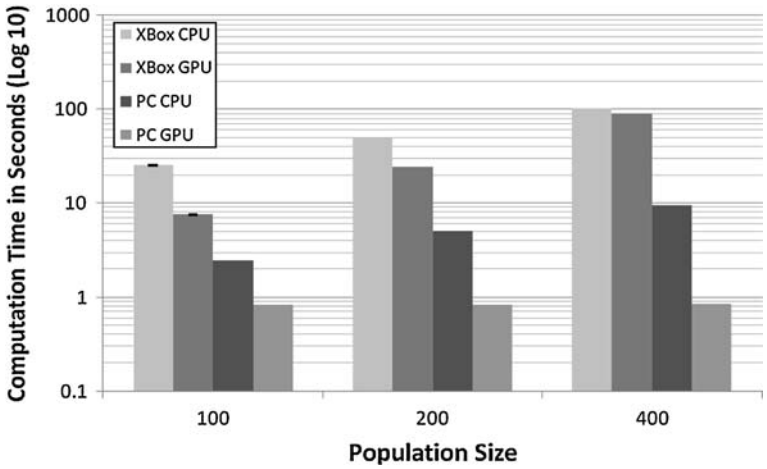

Fig. 8 Xbox 360 and PC performance in actual time (seconds, $\log_{10}$ scale) with standard error, based on 10 trials of 50 generations with populations of 100, 200, and 400 using sextic polynomial

11 times. Furthermore, with increasing population size on PC, the speed increases across GPU implementations: that is, the PC is able to execute more GP operations per second using the GPU as population increases (Fig. 7). This result follows from the fact that PC GPU computation time remains mostly constant with increasing population (Fig. 8) due to parallelizing the fitness and mutation operators of the population (Fig. 8). Rate of processing GP Ops for the CPU is basically constant across population sizes on the PC (Fig. 7) while computation time increases (Fig. 8) due to sequential processing, as would be expected. Like the PC, the GPU speed for every population on the Xbox 360 outperforms the CPU-only implementations in Fig. 8. In contrast to the PC implementation, however, as the GP population increases on the XBox 360, the performance (speed in GP Ops/s) decreases when

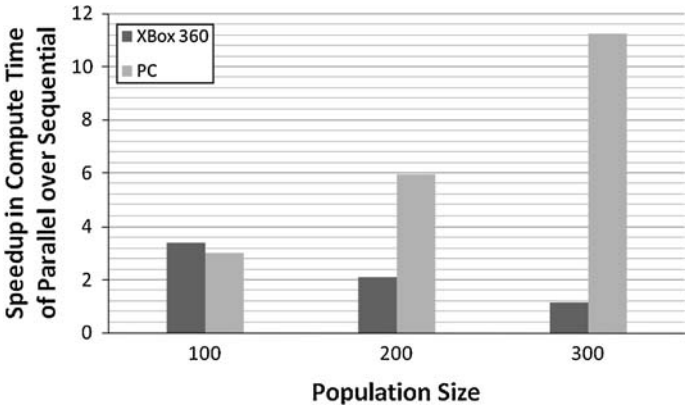

Fig. 9 Speedup in compute time of parallel implementation over sequential implementation for PC and XBox 360 with standard error, based on 10 trials of 50 generations with populations of 100, 200, and 400 using sextic polynomial

Table 12 Final mean performance in GP Ops/s ($\log_{10}$ scale) with standard deviation

Population size	PC CPU		PC GPU	
	Mean	SD	Mean	SD
100	1604018.68	5188.09	4821347.03	28420.87
200	1617191.33	2635.21	9606744.20	86827.19
400	1694851.29	9797.00	19073976.15	329578.00
Population size	XBox 360 CPU		XBox 360 GPU	
	Mean	SD	Mean	SD
100	157647.71	2455.71	533033.34	1814.87
200	157574.06	2464.67	330387.53	238.55
400	155682.09	781.07	175607.32	289.77

using the GPU (Fig. 7). Also in contrast to the PC, Fig. 7 shows that the XBox 360 CPU performance (GP Ops/s) is fairly constant across populations. Due to the trends just discussed, the benefit in implementing the parallel LGP algorithm over the sequential one actually diminishes with increasing population size for the XBox 360 (Fig. 9). The trends of the GPU (Fig. 9) for XBox 360 may be due to the effect of a lower number of GPU processors than the PC combined with the overhead of moving textures and data in and out of GPU memory on Xbox 360 for larger population textures. These results may also have been a product of our particular XBox 360 implementation, the unique architecture of its paired CPU and GPU, or a combination of these factors.

However, it is important to note that the computation time of the sequential implementation (CPU) never exceeds that of the parallel (GPU) for the XBox 360 (Fig. 8 and Table 13). Indeed, Fig. 9 (and Table 14) show that in all

Table 13 Final mean performance in seconds with standard deviation

Population size	PC CPU		PC GPU	
	Mean	SD	Mean	SD
100	2.49376	0.0081	0.82967	0.0050
200	4.94686	0.0081	0.83281	0.0076
400	9.44064	0.055	0.83907	0.015
Population size	XBox 360 CPU		XBox 360 GPU	
	Mean	SD	Mean	SD
100	25.3786	0.40	7.5043	0.026
200	50.7812	0.81	24.214	0.018
400	102.7759	0.52	91.1126	0.15

Table 14 Final mean speedup in compute time of sequential implementation over parallel implementation with standard deviation

Population size	PC		XBox 360	
	Mean	SD	Mean	SD
100	3.005807	0.018	3.381928	0.055
200	5.940395	0.054	2.09719	0.034
400	11.25415	0.19	1.128012	0.0060

implementations the use of the GPU decreased execution time for a given population size (since all speedups of parallel over sequential are greater than 1.0). The PC GPU implementation, however, has much less of an increase in compute time with rising population across populations (Fig. 8), so its acceleration with larger population size is much more significant than any other implementation in terms GP Ops/s (Fig. 7). In addition to speed of execution, it is also of interest to determine how much of the execution time is taking place on the GPU as opposed to the CPU. The percentage of execution occupied by the GPU for both PC and Xbox 360 for each population is shown in Fig. 10, with actual numeric values provided in Table 15.

There is a considerable difference in the percentage of GPU use accounting for total execution time between the PC and the Xbox 360. For the PC implementation, GPU usage only accounts for approximately 30–36% of the execution time. For the Xbox 360, the GPU usage accounts for approximately 80% of the time for the lower population (100), and over 90% for the higher populations of 200 and 400 (Table 15). Comparing Figs. 7 and 10 for the XBox 360, it is evident that the number of GP operations performed per second generally decreases as the proportion of execution time corresponding to GPU usage increases with increasing population (Fig. 9). This may be due to increased loading and unloading of GPU memory registers for the larger populations, which takes GPU execution time but is not directly useful in processing GP operations (in contrast to the actual shader

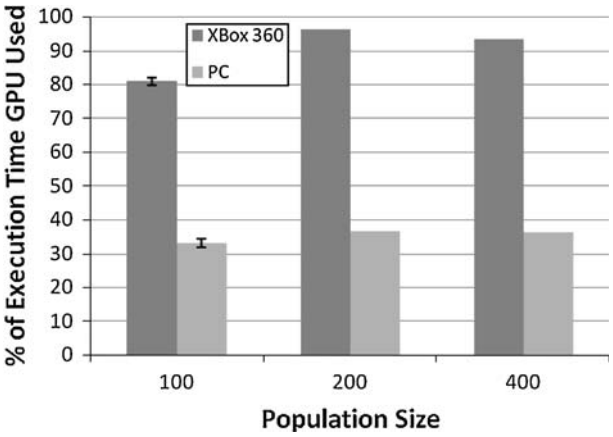

Fig. 10 Percentage of total execution time using GPU for PC and Xbox 360, with standard error, for populations of 100, 200, and 400 using sextic polynomial

Table 15 Final mean percentage of total execution time using GPU with standard deviation

Population size	PC		XBox 360	
	Mean	SD	Mean	SD
100	33.00	0.089	80.95	4.24
200	36.48	0.020	96.04	4.30
400	36.23	0.079	93.19	3.65

program processing during the GPU time). Implementation changes to the fitness function for the Xbox 360 described in Sect. 5.3 (namely, changing fitness case representation to float2[] in memory rather than a texture on the PC) may have further reduced the amount of available GPU memory and exacerbated the amount of loading and unloading of textures.

8 Discussion

While quantitative analysis shows differences in performance comparing CPU to GPU execution times, the main focus of this work is to demonstrate how to create a general LGP algorithm for heterogeneous devices that can be parallelized using the GPU of the respective devices. That is, the algorithm construction involved accessing two platforms for parallel computation, while attempting to minimize the overhead of platform-specific adjustments. The resulting general algorithm variations with respect to CPU and GPU for each platform are summarized below in Fig. 11.

To summarize the general algorithm in Fig. 11, the CPU-only (non-parallel) implementations are the same for both PC and XBox 360 platforms in terms of

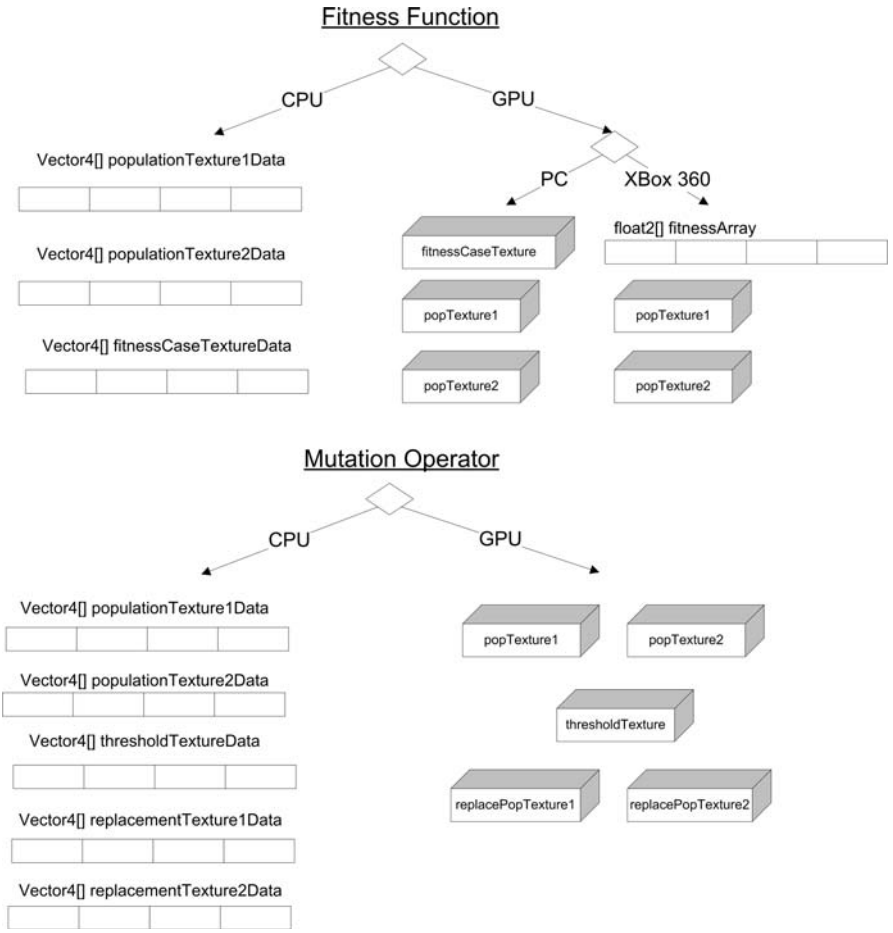

Fig. 11 Relation of CPU and CPU algorithm components for PC and video game console platform implementations of the general algorithm. Names of arrays and textures correspond to those previously used in pseudocode; arrays are represented by *one dimensional grids* and textures as *three dimensional blocks*

program instructions. The main data structures associated with CPU-side execution of the fitness function are, in all cases, the two population arrays (*populationTexture1Data* and *populationTexture2Data*) and the fitness array (*fitnessCaseTextureData*) (Fig. 11 *Fitness Function*, left side). If the user specifies parallel implementation of the algorithm, the GPU-side execution for the PC uses textures with the array data placed on textures (Fig. 11 *Fitness Function*, middle). The GPU-side execution for the XBox 360 uses only the population textures and accesses the fitness cases as an array (Fig. 11 *Fitness Function*, right). The GPU fitness function between PC and XBox 360 is the main divergence from a general implementation.

The data structures associated with CPU-side execution of the mutation operator are, in all cases, the two population textures *populationTexture1Data* and *populationTexture2Data*, a texture of randomly generated mutation probabilities (*thresholdTextureData*) to be compared to the user-specified mutation threshold, and two textures filled with randomly generated possible replacement chromosomes *replacementTexture1Data* and *replacementTexture2Data* (Fig. 11 *Mutation Operator*, left side). In contrast to the fitness function, the GPU-side implementation of mutation uses the same instructions for PC and XBox 360. In both cases, the texture forms of all CPU-side data are processed in parallel to the greatest extent possible: that is, every pixel on every texture is processed in parallel by the shader program (Fig. 11 *Mutation Operator*, right side).

The quantitative results in the previous section should be considered with a number of factors. The PC and XBox not only involve different hardware, but different underlying operating systems on which algorithms with different implementations are being run. The quantitative metrics are not meant to be a benchmark of any hardware, operating system, framework, or any algorithm because they are not consistent across experiments. The experimental results do, however, provide a feel for the computational expense of the different implementations of the general LGP algorithm when run given combined decisions regarding hardware (PC and XBox with their respective OSes) and parallelization (CPU-only and CPU with GPU). In terms of overall trends across configurations, in all cases (populations 100, 200, and 400) it was always of benefit to parallelize sections of the algorithms when possible (Fig. 9; Table 14). That is, using CPU and GPU was a better option than CPU alone in all cases. Because the implementation of GP for the XBox 360 console used the XNA framework, the execution of GP had to occur in the framework of a video game class. Other more straightforward PC implementations are naturally possible for parallel GPU processing, although they would preclude implementation on the XBox 360 GPU. Such PC-only alternatives were not used, since the main contribution of this work was intended to be the explanation of a general and parallelizable algorithm targeted for specific hardware on heterogeneous devices.

9 Conclusions

This work describes the steps to deploying a parallel version of linear GP (LGP) on GPUs across heterogeneous devices. Hardware-related considerations and associated engineering choices for parallel GP implementation on GPU hardware for both PC and video game console (XBox 360) were discussed. GPU parallelization was used for both the fitness function and mutation operator of the GP algorithm. The more sophisticated of the two parallel sections of the implementation, the fitness function, resulted in different implementations for the PC and the XBox 360 GPU hardware. The parallel GPU mutation implementation was more straightforward, and could be implemented in the same way for both PC and XBox 360.

The sextic polynomial regression problem was used to provide a means of examining the performance of the parallel (GPU-based) and sequential (CPU-based)

GP algorithm on the devices. GPU implementations outperformed CPU-only implementations on both the PC and XBox 360 in terms of speed. On the PC, GPU usage was responsible for only approximately 30–35% of the execution time. In contrast, on the Xbox 360, the GPU usage accounted for approximately 80–95% of the execution time.

This work established a definitive method for parallel GP execution using the GPU of a video game system, and discussed all the design decisions required for CPU and GPU implementations on the PC and XBox 360 platforms. The authors attempted to utilize the underlying hardware while still maintaining a general LGP algorithm implemented in the XNA framework common to the heterogeneous devices. Future hardware will likely expand the flexibility of the parallel implementation of GP algorithms across devices, but this work serves as a guide to those wishing to begin deployment on the devices currently in use. The general algorithm presented here, built around a framework targeted at heterogeneous devices, is independent of the changing hardware landscape and could prove useful for deployments in future devices.

Acknowledgements We would like to thank Simon Harding for his helpful feedback and suggestions. WB acknowledges funding from NSERC under the Discovery Grant Program RGPIN 283304-07 and from Canadian Foundation for Innovation under CFI 204503.

References

1. W. Banzhaf, S. Harding, W. Langdon, G. Wilson, *Accelerating Genetic Programming Through Graphics Processing Units. Genetic Programming Theory and Practice (GPTP)* (Springer, New York, 2008), pp. 229–248
2. M. Harris, *Mapping Computational Concepts to GPUs. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* (Addison-Wesley Professional, Boston, 2005)
3. D. Tarditi, S. Puri, J. Oglesby, in *Accelerator: Using Data Parallelism to Program gpus for General-Purpose Uses. Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)* (ACM Press, San Jose, 2006), pp. 325–335
4. M. Wong, T. Wong, K. Fok, in *Parallel Evolutionary Algorithms on Graphics Processing Unit. Proceedings of IEEE Congress on Evolutionary Computation 2005 (CEC 2005)* (IEEE Press, Edinburg, 2005), pp. 2286–2293
5. F. Musgrave, *Genetic Textures. Texturing and Modeling: A Procedural Approach*, 2nd edn. (AP Professional, Cambridge, 1998), pp. 373–385
6. J. Loviscach, J. Meyer-Spradow, in *Genetic Programming of Vertex Shaders. Proceedings of EuroMedia 2003* (Eurosis, Plymouth, 2003), pp. 29–31
7. M. Ebner, M. Reinhardt, J. Albert, in *Evolution of Vertex and Pixel Shaders. Proceedings of the 8th European Conference on Genetic Programming* (Springer, Lausanne 2005), pp. 261–270
8. F. Lindblad, P. Nordin, K. Wolff, in *Evolving 3D Model Interpretation of Images Using Graphics Hardware. Proceedings of the 2002 Congress on Evolutionary Computation (CEC 2002)* (IEEE Press, Honolulu, 2002), pp. 225–230
9. Q. Yu, C. Chen, Z. Pan, in *Parallel Genetic Algorithms on Programmable Graphics Hardware. Proceedings of the First International Conference on Natural Computation, ICNC 2005, vol. LNCS 3612* (2005), pp. 1051–1059
10. K. Fok, T. Wong, M. Wong, Evolutionary computing on consumer graphics hardware. IEEE Intell. Syst. **22**(2), 69–78 (2007)

11. S. Harding, W. Banzhaf, in *Fast Genetic Programming on GPUs. Proceedings of the 10th European Conference on Genetic Programming* (Springer, Valencia, 2007), pp. 90–101
12. D. Chitty, in *A Data Parallel Approach to Genetic Programming Using Programmable Graphics Hardware. Proceedings of the 2007 Genetic and Evolutionary Computation Conference (GECCO 2007)* (ACM Press, London, 2007), pp. 1566–1573
13. W. Langdon, W. Banzhaf, in *A SIMD Interpreter for Genetic Programming on GPU Graphics Cards. Proceedings of the 11th European Conference on Genetic Programming* (Springer, Naples, 2008), pp. 73–85
14. G. Wilson, W. Banzhaf, in *Linear Genetic Programming GPGPU on Microsoft's Xbox 360. Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2008)* (IEEE Press, Hong Kong, 2008), pp. 378–385
15. G. Wilson, W. Banzhaf, in *Deployment of CPU and GPU-Based Genetic Programming on Heterogeneous Devices. Proceedings of the 2009 Genetic and Evolutionary Computation Conference (GECCO 2009)* (ACM Press, Montreal), pp. 2531–2538
16. N.L. Cramer, in *A Representation for the Adaptive Generation of Simple Sequential Programs. Proceedings of the First International Conference on Genetic Algorithms* (1985), pp. 183–187
17. P. Nordin, W. Banzhaf, in *Complexity Compression and Evolution. Proceedings of the Sixth International Conference on Genetic Algorithms* (1995), pp. 310–317
18. P. Nordin, Evolutionary program induction of binary machine code and its applications, Ph.D. Thesis, University of Dortmund, Department of Computer Science, 1997
19. W. Banzhaf, P. Nordin, R. Keller, F. Francone, *Genetic Programming: An Introduction* (Morgan Kaufman, San Francisco, 1998)
20. Microsoft Corporation, Xbox 360 Tools and Middleware Program (2009), http://www.xbox.com/en-US/dev/tools.htm
21. S. Scarle, Implications of the turing completeness of reaction-diffusion models, informed by GPGPU simulations on an XBox 360: cardiac arrhythmias, re-entry and the Halting problem. Comput. Biol. Chem. **33**, 253–260
22. Microsoft Corporation, XBox 360 Programming considerations (2009), http://msdn.microsoft.com/en-us/library/bb203938(XNAGameStudio.10).aspx
23. Shawn Hargreaves, Point sprites on XBox (2007), http://blogs.msdn.com/shawnhar/archive/2007/01/03/point-sprites-on-xbox.aspx
24. J. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs* (MIT Press, Cambridge, 1998)
25. J. Andrews, N. Baker, XBox 360 system architecture. IEEE Micro **26**, 25–37 (2006)