Evolving CUDA PTX Programs by Quantum Inspired Linear Genetic Programming

Leandro Cupertino, Cleomar Silva, Douglas Dias, Marco Aurélio Pacheco
Department of Electrical Engineering
Pontifical Catholic University of Rio de Janeiro
R. Marquês de São Vicente, 225
Rio de Janeiro, Brazil
{cuper, cleomar, douglasm, marco}@ele.puc-rio.br

Cristiana Bentes
Department of System Engineering
State University of Rio de Janeiro
R. São Francisco Xavier, 524
Rio de Janeiro, Brazil
cris@eng.uerj.br

ABSTRACT

The tremendous computing power of Graphics Processing Units (GPUs) can be used to accelerate the evolution process in Genetic Programming (GP). The automatic generation of code using the GPU usually follows two different approaches: compiling each evolved or interpreting multiple programs. Both approaches, however, have performance drawbacks. In this work, we propose a novel approach where the GPU pseudo-assembly language, PTX (Parallel Thread Execution), is evolved. Evolving PTX programs is faster, since the compilation of a PTX program takes orders of magnitude less time than a CUDA program compilation on the CPU, and no interpreter is necessary. Another important aspect of our approach is that the evolution of PTX programs follows the Quantum Inspired Linear Genetic Programming (QILGP). Our approach, called QILGP3U (QILGP + GPGPU), enables the evolution on a single machine in a reasonable time, enhances the quality of the model with the use of PTX, and for big databases can be much faster than the CPU implementation.

Categories and Subject Descriptors

I.2.2 [**Computing Methodologies**]: Artificial Intelligence—*automatic programming, program synthesis*

General Terms

Performance

Keywords

GPU, CUDA, PTX, quantum-inspired algorithms, genetic programming

GECCO'11, July 12–16, 2011, Dublin, Ireland.

1. INTRODUCTION

Genetic programming (GP) is a systematic method for automatically generating computer programs. The essence of GP is to solve a problem from a high-level statement based on Darwin's evolutionary theory to search through a space of possible computer programs. This evolutionary process, however, can be very costly when real world problems, like the ones in [15], are considered. For these problems, millions of programs need to be evaluated at each GP complete run. In order to tackle this problem, past work has focused on applying high performance computation techniques to speedup the GP task [1, 26]. GP is inherently parallel in that each candidate program can be evaluated independently from the others. Therefore, the parallelism derived from many candidate solutions being evaluated at the same time has been exploited in different solutions for multiprocessor machines or clusters of computers [23, 28, 2].

Recently, Graphics Processing Units (GPUs) offer a huge computing power that is frequently an order of magnitude larger than the most modern multicore CPUs. Driven by ever increasing requirements from the video game industry, modern GPUs are very powerful and flexible processors, while their price remains in the range of mass consumer market. The NVIDIA Compute Unified Device Architecture (CUDA), that specifies extensions to the C programming language for writing program targeting to the GPUs, enhances the viability of GPUs as a general-purpose computing platform. CUDA provided a straightforward programming model and language, and GPUs are considered attractive platforms for high performance GP.

Previously, the parallelism of GPUs has been exploited for running GP using two different approaches: (i) compiling each evolved program [4, 11, 12, 17], or (ii) interpreting multiple programs [16, 18, 25, 29]. In the compiling evolved programs approach, after the program is compiled to run on the GPU, all the fitness cases can be executed in parallel. In the interpreting multiple programs approach, multiple individuals can be evaluated simultaneously, by the implementation of a GPU interpreter. Both approaches have performance drawbacks. In the compiling evolved programs approach, before any evaluation can occur, the program has to be compiled on the CPU. The compilation time can generate a huge overhead. For real world problems, where the compilation of a CUDA program would take a few seconds, the evaluation of millions of individuals could be unfeasible.

In the interpreting multiple programs approach, although the interpreter is able to execute the evolved programs in parallel, typically interpreted code runs much slower than optimized compiler generated machine code.

We propose here a different approach to deal with these performance problems in the code generation/execution. In our approach, we propose the evolution of PTX (Parallel Thread Execution) programs. PTX is a pseudo-assembly language used in CUDA environment. The *nvcc* compiler translates code written in CUDA into PTX and than into a code that runs on the GPU, all on the CPU side. Evolving PTX programs is faster than using the compilation approach, since the compilation of a PTX program takes around 100 times less time than a CUDA program compilation on the CPU. Also, no interpreter is necessary, since the compilation of a PTX code results in a native code program which can be directly executed by the GPU. Another important advantage of evolving PTX programs is that we can focus on ways to parallelizing genetic programming that distribute the computational effort needed to compute fitness. In [14], Koza proposes three levels of parallelization: fitness case, individuals and independent runs. In the level of fitness case, all the fitness cases are executed in parallel with only one individual being evaluated at a time. In the level of individuals, multiple individuals are evaluated in parallel. The level of independent runs, several evolutions will occur at the same time.

Besides the problem of code generation/execution and the parallelization methodology, we also address here the evolution process itself. We employ linear GP, since it is best suited to evolve programs in imperative languages like C or assembly [3]. The number of instructions can be fixed or variable, which means that different individuals may have different sizes [24]. Our evolution of PTX programs, however, follows the Quantum Inspired Linear Genetic Programming (QILGP), recently proposed in [7]. QILGP evolves x86 machine code programs using a quantum-inspired approach.

The Quantum-Inspired Evolutionary Algorithms (QIEAs) [20] is one of the most recent advances in Evolutionary Computation. It is motivated by quantum computing, which describes computational processes that are based on making direct use of certain quantum mechanics phenomena (e.g. superposition of states) to perform data operations. These phenomena allow to construct computers that, in theory, comply with new and more permissive laws of computational complexity [27]. In a quantum computer, the basic unit of information, named *qubit*, can take the states $|0\rangle$, $|1\rangle$ or a superposition of both states. When observed, the *qubit* is brought to the classical level and the observed state is the value 0 or 1. This superposition of states gives quantum computers an exceptional degree of parallelism that, if properly exploited, allows them to perform some tasks that would be unfeasible for classical computers. In other words, QIEAs take advantage of quantum mechanics paradigms in order to improve the performance of computer algorithms.

There are some examples of successful QIEAs in the literature. The evolutionary algorithm using binary representation, originally proposed in [10], is one of them. This model uses a special representation which simulates a chromosome consisting of *qubits*, instead of a conventional binary representation. QIEA was also used in [13] in multiobjective combinatorial optimization problems with results superior to conventional genetic algorithms in terms of convergence time and quality of solutions. The evolutionary algorithm for numerical optimization proposed in [5] is another example of a QIEA, which is based on the principle of multiple universes of quantum physics. This QIEA is represented by real numbers and has a smaller convergence time for benchmark problems compared to conventional algorithms in [6]. QILGP uses QIEA to evolve programs and shows in [8] better performance for some symbolic regression and binary classification problems when compared to the reference model, AIMGP (Automatic Induction of Machine Code by Genetic Programming) [21], which is the most successful model of classical Linear GP in the evolution of machine code programs.

The approach we propose here for using GPUs to accelerate genetic programming is called QILGP3U (QILGP + GPGPU). The contributions of QILGP3U are twofold. First, it evolves assembly programs for GPUs, more specifically PTX programs for NVIDIA CUDA GPUs. Second it employs QILGP for evolving the PTX programs. Our approach uses PTX instead of machine code instructions, enabling a more flexible set of instructions. Our results show that the quality of the model varies depending on the instructions set used and for large regression data it can also be faster.

The rest of this paper is organized as follows. In Section 2 the Quantum Inspired Linear Genetic Programming is explained. The CUDA architecture and its compilation stages are explained in Section 3. Section 4 presents QILGP3U, our quantum-inspired GPU GP model. The experimental setup and results are shown in Section 5. Finally, conclusions and future work are presented in Section 6.

2. QUANTUM INSPIRED LINEAR GENETIC PROGRAMMING

Quantum Inspired Linear Genetic Programming (QILGP) is based on the following entities: the "quantum individual" chromosome, which represents the superposition of all possible programs for the defined search space, is observed to generate the "classical individual" chromosome, from which the machine code program is finally generated. That is, the classical individual chromosome is the internal representation of a machine code program. The chromosome of classical individual (CI) represents the functions by a "function token" (FT), which may take integer values from 0 to $(f-1)$, in order to uniquely represent each of f QILGP functions. Floating Point Unit (FPU) instructions have only one or no argument. In other words, all the set functions have only one terminal, which is represented by a "token terminal" (TT). For a function which has no terminal, its corresponding terminal token value is ignored by model. Such a chromosome can be represented by a structure with $(L \times 2)$ tokens, where: L is the maximum program length (in number of instructions). The execution order of the program is sequential from first to last instruction. In turn, each instruction is defined as a "gene". Although this chromosome is fixed-length, the effective program that it represents has variable length. This variation is obtained by adding the `NOP` instruction in the function set. In turn, the process of code generation ignores any gene in which a `NOP` instruction is present, i.e., any gene whose value of its function token is zero is ignored.

2.1 Quantum Individuals

QILGP is inspired by multilevel quantum systems [19]. Therefore, the basic information unit adopted by QILGP is the *qudit*. This information can be described by a state vector in a quantum mechanical system of d levels, which equates to a d-dimensional vector space, where d is the number of states in which the *qudit* can be measured in. That is, d represents the cardinality of the token that will have its value determined by the observation of its respective *qudit*. The state of a *qudit* is a linear superposition of d states and may be represented by Equation 1:

$$|\psi\rangle = \sum_{i=0}^{d-1} \alpha_i |i\rangle , \tag{1}$$

where the $|\alpha_i|^2$ value represents the probability p that *qudit* is found in state i when observed. The unitary normalization of this state guarantees: $\sum_{i=0}^{d-1} |\alpha_i|^2 = 1$.

The chromosome of a quantum individual is represented by a list of structures named "quantum genes". A quantum gene is composed by a function *qudit* (FQ), which represents the superposition of all functions predefined by the function set. It also has two terminals *qudits* (TQ), since functions can use two different types of terminals. One of the terminal *qudits* represents the FPU registers (TQ_{Reg}) and the other one represents memory locations (TQ_{Mem}). These registers and memory locations belong to a predefined terminal set. For example, the instruction `FADD ST(0), ST(i)` uses a terminal *qudit* that represents the superposition of the indexes i of registers `ST(i)`, while terminal *qudit* for the instruction `FADD` m represents the superposition of memory locations m. Since each quantum gene is observed to generate a classical individual gene (i.e. a complete instruction), both quantum (QI) and classical individuals (CI) have the same length.

Figure 1 illustrates the creating process of a gene by the observation of a quantum gene from an example based on hypothetical case with "7" being the function token of `FMUL` m and the inputs being represented by an input vector I, as exemplied by Equation 2.

$$I = (V[0], V[1], 1, 2, 3), \tag{2}$$

where $V[0]$ and $V[1]$ contain the two input values of a problem, and where 1, 2 and 3 are the values of three predefined constants. This process can be explained by three basic steps, indicated by numbered circles in figure 1, as follows.

1. The function *qudit* (FQ) is observed and the resulting value (e.g. 7) is assigned to the function token (FT) of this gene.

2. The function token value determines the terminal *qudit* to be observed, since each instruction requires a different type of terminal amongst two: register or memory.

3. The terminal *qudit* (TQ) determined by the function token value is observed and the resulting value (e.g. 1) is assigned to the terminal token (TT) of this gene.

So in this example, the observed instruction is `FMUL V[1]`, since "7" is the function token value for this instruction and "1" is the terminal token value that represents $V[1]$ in input vector I defined in (2). That is, if a terminal token is for an instruction whose argument is a memory content, the terminal token value indicates the input vector position to be used as its argument. Therefore, in this case, the terminal token values from "0" to "4" indicates that the argument is $V[0]$, $V[1]$, 1, 2 or 3, respectively. However, if the terminal token is for an instruction whose argument is the contents of a FPU register, the terminal token value directly indicates which of the eight registers the argument is. Only the first four registers are used in this example: $ST(0)$ to $ST(3)$.

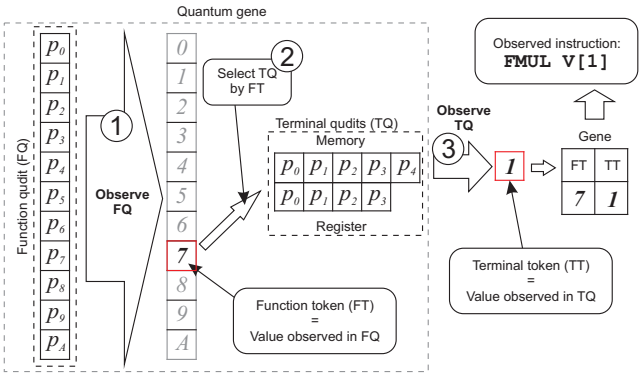

Figure 1: Creation of a gene by the observation of a quantum gene. Function and terminal *qudits* genes are expressed in terms of its probabilities p.

2.2 Quantum Operator

The quantum operator proposed in [7] acts directly on a probability p_i of a *qudit*, satisfying the normalization condition: $\sum_{i=0}^{d-1} |\alpha_i|^2 = 1$, where d is the *qudit* cardinality and $|\alpha_i|^2 = p_i$. Thus, this operator, here named "P operator", represents the functionality of a quantum gate, performing rotations in a vector which represents a state $|\psi\rangle$ of a *qudit* in a d-dimensional vector space.

This operator works in two basic steps. First, it increases a given probability of a *qudit*, as follows:

$$p_i \leftarrow p_i + s(1 - p_i), \tag{3}$$

where s is a parameter named "step size", which can assume any real value between 0 and 1. The second step is to adjust the values of all the probabilities of this *qudit* to satisfy the normalization condition. Therefore, P operator modifies the state of *qudit* increasing the value of its probability p_i by a quantity which, in turn, is directly proportional to the value of step size s.

On the asymptotic behavior of p_i, by equation 3, we can note that a probability never reaches the unit value. This is an important feature of this operator, because it avoids that a probability take its *qudit* to collapse, which could cause a premature convergence of the evolutionary search process, damaging the model's performance.

2.3 Overall Structure and Operation

Regarding its structure, QILGP has a hybrid population which, in turn, is composed of two populations, one quantum and one classical, both having the same number M of individuals. It also has M auxiliary classical individuals C_i^{obs}, which result from observations of quantum individuals Q_i, where $1 \leq i \leq M$. The four basic steps that characterize a "generation" of QILGP, are described below:

1. Each of M quantum individuals is observed once, resulting in M classical individuals C_i^{obs}.

2. The individuals of classical population and the observed individuals are jointly sorted by their evaluations. As a result, the M best individuals from the $2M$ evaluated ones are kept in classical population, ordered from best to worst, from C_1 to C_M. The other M classical individuals remain stored and sorted from best to worst, from C_1^{obs} to C_M^{obs}.

3. The P operator is applied to each individual of quantum population, with reference to their corresponding individuals in classical population. This step is where the evolution actually occurs, since every new generation, application of this operator increases the probability that quantum individuals' observations generate classical individuals more similar to the best ones found so far.

4. If any individual of classical population evaluated in the current generation is better than the best classical individual evaluated so far (compared to previous generations), a copy is stored in C_B, the best classical individual found by the algorithm so far.

3. CUDA ARCHITECTURE

GPUs are highly parallel, many-core stream-processing units typically used as accelerators to a host system. It supports a great number of fine-grain threads, but its cores are simpler than CPU cores. The connection between the GPU and the host is done through the PCIe bus. The CUDA programming model was created for developing applications for this platform. CUDA an industry C-based development environment for GPUs, that includes a new parallel programming model and an instruction set architecture. CUDA allows the programmer to define a special C function, called a *kernel*, which executes in parallel on the GPU by different threads. The programmer organizes these threads into a hierarchy of grids of thread blocks. A thread block is a set of concurrent threads that can cooperate among themselves through barrier synchronization and shared accesses. A grid is a set of thread blocks that may be executed in parallel.

CUDA's programming model assumes that the threads execute on a physically separate device that operates as a coprocessor to the host running the C program. Both the host and the device have separate memory spaces, referred to as host memory and device memory, and the data transfers between host and device memory are made in the kernels.

3.1 NVCC Compilation

In short, CUDA compilation works as follows: the input program is separated by the CUDA front end (*cudafe*), into C/C++ host code and the GPU device code. This device code is further translated by the CUDA compilers/assemblers into a intermediate PTX code and then into CUDA binary (*cubin*), working as a two stage process. This binary code is merged into a device code descriptor which is included by the previously separated host code. This descriptor will be inspected by the CUDA runtime system whenever the device code is invoked by the host program, in order to obtain an appropriate load image for the current GPU [22].

The compilation of CUDA code to a GPU binds its *cubin* to one generation of GPUs. Within that generation, it optimizes the code to run on that specific GPU. The CUDA

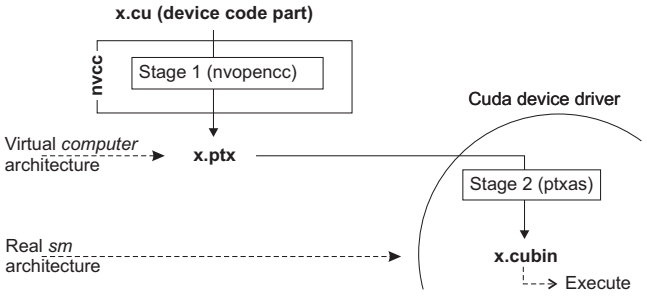

Figure 2: Just-In-Time compilation schema [22].

Just-in-time compilation (JIT) postpones the second compilation stage until application runtime, at which the target GPU is exactly known (Figure 2). JIT compilation embraces a larger coverage of different GPUs and enables users to evolve PTX code directly, running just the stage two of the compilation.

4. QILGP3U

The approach we propose here is called QILGP3U, and is based on the quantum inspired evolutionary method of QILGP, due to its fast convergence and high quality results [7, 8]. The main difference of QILGP3U and QILGP is the representation of the individuals and the way they are evaluated. QILGP3U represents its individuals as PTX code and evolve a program without the *nvcc* compilation overhead. This is made possible through the CUDA JIT compiler, which loads PTX modules into the GPU in real time. Thus, QILGP3U can overcome the overhead of interpreting the code during the executing, while the compilation time is not the bottleneck of the application.

QILGP3U explores two different levels of parallelization by two different implementations. The first one, called fitness parallel, explores the level of fitness cases, evaluating all patterns (training, validation and test) at once. This parallelization requires that a different program is uploaded to the GPU before each evaluation, which can be costly, due to the overhead of transferring data from CPU to the GPU. For small databases, there are fewer patterns than stream processors. In this scenario, some of the processors are idle. This problem will become worse as the number of stream processors increases with new GPU generations. Unfortunately, a large number of classic benchmark GP problems fit into this category. However, it is important to notice that there are other cases that need a huge amount of data to run, as for example, the analysis of wine quality, communities or crime, available in [9].

In order to overcome this idleness, our second implementation explores the parallelism, not only in the level of fitness cases, but also in the level of individuals. This enables the algorithm to evaluate an entire population of programs at once. The common approach so far for evaluating a population of programs in parallel is to implement some form of interpreter on the GPU [16, 18], but the interpreter executes the evolved programs in a pseudo-parallel manner, which will be slower than the GPU code, since it introduces a lot of code divergence. As the cost of JIT is lower than the cost of the *nvcc* compiler, our implementation creates a code where each individual is contained in a thread block. This

Table 1: Functional description of the instructions and their corresponding set.

Instruction	Description	A	FS
NOP	No operation	-	1, 2
add.f32 R0, R0, Xj	$R(0) \leftarrow R(0)+X(j)$	j	1, 2
add.f32 R0, R0, Ri	$R(0) \leftarrow R(0)+R(i)$	i	1, 2
add.f32 Ri, Ri, R0	$R(i) \leftarrow R(i)+R(0)$	i	1, 2
sub.f32 R0, R0, Xj	$R(0) \leftarrow R(0)-X(j)$	j	1, 2
sub.f32 R0, R0, Ri	$R(0) \leftarrow R(0)-R(i)$	i	1, 2
sub.f32 Ri, Ri, R0	$R(i) \leftarrow R(i)-R(0)$	i	1, 2
mul.f32 R0, R0, Xj	$R(0) \leftarrow R(0)\times X(j)$	j	1, 2
mul.f32 R0, R0, Ri	$R(0) \leftarrow R(0)\times R(i)$	i	1, 2
mul.f32 Ri, Ri, R0	$R(i) \leftarrow R(i)\times R(0)$	i	1, 2
div.full.f32 R0, R0, Xj	$R(0) \leftarrow R(0)\div X(j)$	j	1, 2
div.full.f32 R0, R0, Ri	$R(0) \leftarrow R(0)\div R(i)$	i	1, 2
div.full.f32 Ri, Ri, R0	$R(i) \leftarrow R(i)\div R(0)$	i	1, 2
exchange R0, Ri	$R(0) \leftrightarrows R(i)$ (swap)	i	1, 2
abs.f32 R0, R0	$R(0) \leftarrow \lvert R(0)\rvert$	-	1
sqrt.approx.f32 R0, R0	$R(0) \leftarrow \sqrt{R(0)}$	-	1
sin.approx.f32 R0, R0	$R(0) \leftarrow \sin R(0)$	-	1
cos.approx.f32 R0, R0	$R(0) \leftarrow \cos R(0)$	-	1
abs.f32 Ri, Ri	$R(i) \leftarrow \lvert R(i)\rvert$	i	2
sqrt.approx.f32 Ri, Ri	$R(i) \leftarrow \sqrt{R(i)}$	i	2
sin.approx.f32 Ri, Ri	$R(i) \leftarrow \sin R(i)$	i	2
cos.approx.f32 Ri, Ri	$R(i) \leftarrow \cos R(i)$	i	2

guarantees that there will be no divergence of code when evaluating the entire population.

4.1 Target Platform

QILGP3U uses PTX code for addition, subtraction, multiplication, division, data transfer, trigonometric and arithmetic instructions as the function set. The model uses some instructions of FPU, which can work with inputs and constants registers (X) and eight auxiliary FPU registers ($Ri \mid i \in [0\,..\,7]$). In order to make a fair comparison with the QILGP model, two instructions sets were used, the first one (FS1) contains the exact same instructions of QILGP, under using the ability of some instructions; the second one (FS2) allows that `abs`, `sqrt` and trigonometric functions have a register argument. As PTX does not contain the exchange (`FXCH ST(i)`) instruction, a sequence of `mov.f32` and an auxiliary register ($R8$) were used to create a similar instruction. Note that all instructions in this set have only one or no argument. Table 1 shows these instructions, their operation, the argument (A) of each instruction (if any) and the function set to which they belong (FS1 or FS2).

A PTX code program evolved by QILGP3U represents a solution. This program reads the input data from global memory, which are composed by the input variables of the problem and by some optional constants supplied by the user, e.g. vector 2. Finally, the result is stored on the global memory, in order to be read by the main CPU thread.

4.2 Evaluation of a Classical Individual

Each program evolved by QILGP3U, is a PTX code program consisting of three segments: header, body and footer. The header and footer are not affected by the evolutionary process. These segments are described as:

- *Header* – Loads the evaluation patterns from global memory to registers on the GPU and initializes its eight registers with zero.
- *Body* – Is the evolved PTX code itself.
- *Footer* – Transfers $R0$ contents to global memory, since this is the default output of evolved programs. Then executes the exit instruction to terminate the program and return to the evolutionary algorithm main flow.

The evaluation process treats the problems caused by instructions such as `div.full.f32` incurred in dividing by zero or instructions `sqrt.approx.f32` incurred in calculating a negative number square root, which directly affect the value resulting from the execution of an evolved program. In both cases, the value attributed as result of such fitness case is zero ($Ri \leftarrow 0$). This is the same approach adopted by QILGP, which promotes neutrality when comparing the results of the models.

In the population parallel implementation, the PTX code is only created at each generation. Its body is slightly different from the fitness parallel implementation because it has all the individuals in just one kernel, separated by blocks. This requires some conditionals commands to evaluate which block is running in order to locate different individuals. The best individual is stored on the CPU side and, at the end of the evolution; a body with only the best individual (without block conditionals) is presented to the user.

5. RESULTS

For the evaluation of QILGP3U, we used a well-known benchmark for regression problems with real attributes named "Mexican Hat" [3]. This benchmark is represented by a two-dimensional function given by equation 4, as follows:

$$f(x,y) = \left(1 - \frac{x^2}{4} - \frac{y^2}{4}\right) \times e^{\left(-x^2-y^2\right)/8}. \qquad (4)$$

Its surface is shown in Figure 3. GP has the task of reconstructing such surface from a given set of points.

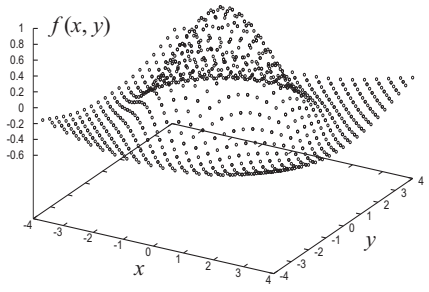

Figure 3: Mexican hat surface function.

The x and y variables are uniformly sampled in the range $[-4, 4]$ to generate the training, validation and testing data sets. The fitness value of an individual is its mean absolute error (MAE) over the training cases, as given by equation 5:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |t_i - V[0]_i| \qquad (5)$$

where t_i is the target value for the i case and $V[0]_i$ is the individual's output value for that same case.

The parameter settings used for all experimental tests are shown in table 2. The total number of runs for each model was 10. Each run consists of a full evolution of the algorithm.

Table 2: Parameter settings of all models.

Parameter	Setting
Number of generations	400,000
Population size	6
NOP initial probability ($\alpha_{0,0}$)	0.9
Step size (s)	0.004
Maximum program length	128
Function set	(see Table 1)
Set of constants	$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

The hardware used for the experiments was a computer with an AMD Phenom II X4 965 processor (with 128Kb of L1 cache per core – 64KB for instruction and 64KB for data), running at 3 GHz, 16 GB of RAM and a nVidia Tesla C1060 GPU. This GPU has 240 processing elements (at 1.3 GHz) and 4 GB of RAM with a memory bandwidth of 102 GB/s through a 512-bit data bus.

5.1 Parallel Implementations Analysis

First, we evaluated the two parallel implementations: fitness case parallel and population parallel. To do so, we varied the number of code lines in a randomly generated PTX program from 1 to 128 (our maximum program length). Each of these programs was executed in both implementations, considering that for each 6 (our population size) executions of the fitness case, one execution of the population parallel with the same 6 programs was run. All programs executions used the same training patterns, with 256 cases sampled from equation 4 in a 16×16 uniform grid. An average of 1000 executions was used for the analysis.

In this experiment, we observed that the time spent on executing the possible solutions for the problem corresponds to less than 1% of the total evaluation time. Therefore, the time spent on compiling the solutions accounts for almost all the evaluation time. Figure 4 shows the difference in the PTX compilation time for the two parallel implementations. The curve shows the speedup of the population parallel compilation time over the fitness parallel compilation time, as a function of the number of code lines. In this graph, one can observe that, for small programs, the population parallel implementation compiles more than 2 times faster than the fitness parallel implementation. But the speedup obtained by the population parallel compilation decreases as the number of code lines per program increases. This occurs because for smaller programs, the header and footer of the program represent a large percentage of the total compilation cost. As the problem size increases, the influence of the header and footer on the PTX compilation decreases.

Although the execution time of the solutions obtained are negligible when compared to the compilation time, the execution times obtained by the population parallel implementation were about 4.63 times faster than the execution times for the fitness parallel implementation.

Due to the better performance results of the population parallel implementation over the fitness parallel implementation, the next experiments were done using only the population parallel implementation. Following, we analyze two

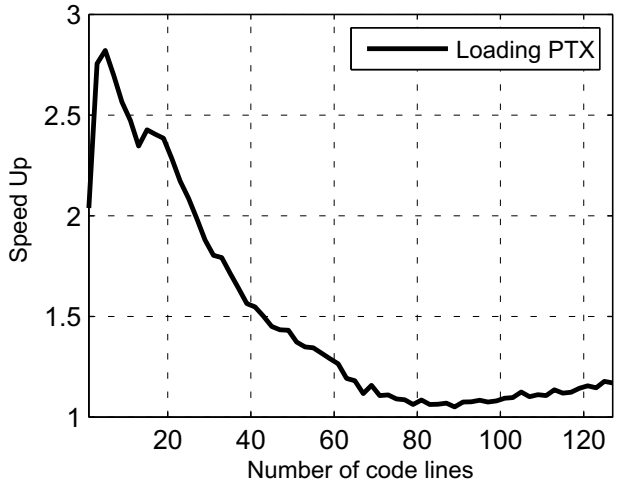

Figure 4: Speedup of the compilation time for population parallel implementation over the fitness parallel implementation.

different aspects of this implementation: the convergence (by inspecting the evolutionary graph of each model), and the performance (by varying the problem size and measuring their execution time).

5.2 Convergence Analysis

One can consider that the algorithm converges when the fitness of the best individuals stop improving significantly. To evaluate it, 289 training patterns were sampled from equation 4 in a 17×17 uniformly distributed grid, while the validation and testing data sets contains 256 samples in a 16×16 grid.

The two different function sets (FS1 and FS2), previously described in section 4, were used to generate an evolutionary graph of QILGP3U. This graph was built using the average fitness of 10 runs. FS1 is just a literal translation of CPU operations to GPU operations. However, a GPU provides more instruction flexibility that has to be explored. For example, in a GPU it is possible to perform some operations like sine, cosine and modulus using any register, while in a CPU it is necessary to move data to register zero before performing such operations. So in a CPU some extra data exchange operations is needed. FS2 has the objective of investigating the effect of GPU instructions flexibility. The need for extra exchange operations is eliminated, but the search space is increased.

Figure 5 shows the evolution of each QILGP3U function set as a function of evaluated individuals. One can observe that using the FS1 (QILGP-FS1), the GP converges faster than with FS2 (QILGP-FS2), and then remains almost constant at about 1.5 million individuals. Moreover, the FS1 final result has a slightly better MAE value than FS2. In spite of FS2 allowing greater flexibility of code, it increases the search space, hindering the evolution of the algorithm. These simple modifications degraded the convergence results, leading to a higher MAE.

Table 3 shows the best individuals average and standard deviation (σ) for training, validation and test data set of each model. The average of the best individuals of QILGP3U-FS1 is better than the other function set for all data sets,

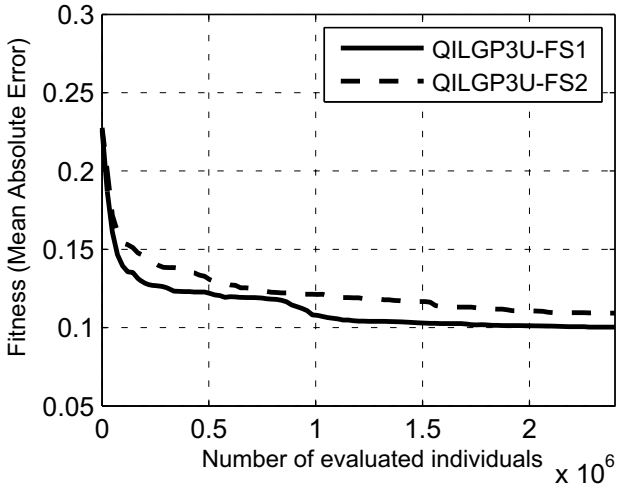

Figure 5: Evolutionary graphs of both function sets.

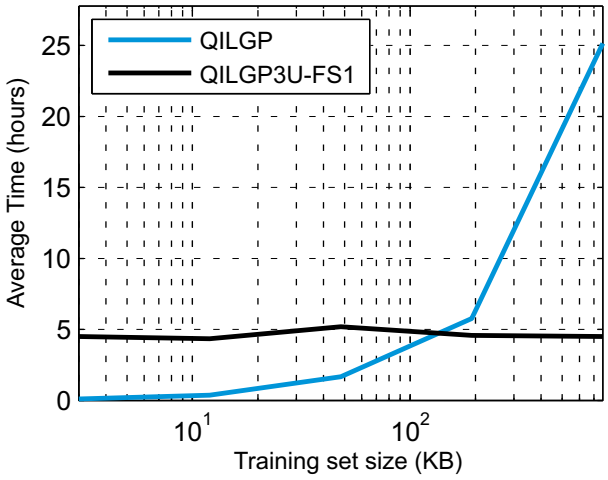

Figure 6: Execution time for each model.

being 1.4 times better than QILGP3-FS2. The standard deviations of all the cases are relatively low for the number of runs used.

Table 3: Mean Absolute Errors for QILGP3U.

		FS1	FS2
Training	Average	0.0963	0.1070
	σ	0.0218	0.0113
Validation	Average	0.1039	0.1114
	σ	0.0254	0.0230
Test	Average	0.1104	0.1152
	σ	0.0304	0.0292

5.3 Performance Analysis

In this section we are interested in the total execution time cost of the applications as function of the fitness case size. The execution times of the experiments related to QILGP and QILGP3U models were measured running 10 complete evolution runs, with different training sets. Each training pattern has three float numbers (x, y and $f(x, y)$). The training data set was created according to its sample grid, which varied from 16×16 (3KB) to 256×256 (768KB). In these results, only the best QILGP3U instruction set was analyzed (function set 1).

The execution time results are presented in Figure 6, where the total execution time is shown as a function of the total number of input samples. For small size problems, QILGP3U is slower than QILGP. However, as the problem size increases, QILGP3U becomes much faster than QILGP. If we take a closer look, we can see that the execution of QILGP3U becomes faster when the fitness cases gets bigger than 64KB, which is the L1 data cache size for each core of the processor used in these experiments. We can conclude that this performance gain is not just because of the large number of stream processors on the GPU, but also because of the cache behavior on the CPU for larger datasets.

Langdon and Harman demonstrated in [17] that it is possible to evolve a parallel GPU program or a CUDA Kernel. However, they showed that there are still many challenges in evolving GPU programs. In [12], the compilation of CUDA code was done using nvcc and took a long time, they proposed the use of a computer cluster to reduce the compilation overhead. In this paper, we are interested in investigating the possibility of using PTX code instead. When profiling the GPU implementation, we observed that more than 90% of QILGP3U total execution time is due to the stage 2 of JIT compilation (Figure 2). This bottleneck can only be eliminated by evolving the *cubin* code directly, which is very hard, because there is no official documentation about its instructions.

6. CONCLUSIONS

In this work, we propose a new approach to the parallelization of GP on GPUs. We focus on exploring the power of the GPU to parallelize the evaluations, and on reducing the overheads of traditional evolution approaches based on the GPU. Our approach, called QILGP3U, is based on the QILGP algorithm, a stochastic algorithm which evolves programs by using linear chromosomes. Another important aspect is that we evolved PTX code, instead of evolving CUDA code, enabling this way a faster compilation.

QILGP3U explores two different leves of paralellization: fitness parallel and population parallel. Our results show that the population parallel scheme obtained better performance over the fitness case parallel. In terms of the accuracy of our approach, QILGP3U is as accurate as the QILGP model. Another interesting point is that, even though the first stage of the compilation is not executed, the JIT stage still takes a expressive time to generate the *cubin* executable, and, at each generation, it is responsible for more than 99% of the evaluation time. We also observed that by changing the instruction set, to enable more flexibility in the final program, the evolution took more time, and converged to a worst solution. This occurs due to the increase of the search space. Finally, in terms of the increase in the total execution time as a function of the training set size, while the CPU implementation increases exponentially, the GPU implementation maintains constant.

As future work, we propose the evolution of the GPU machine code, (*cubin*), to generate faster codes. *Cubin* is an architecture-specific code, and the use of this binary code should be investigated in the QILGP3U model. Besides,

both the PTX and *cubin* code evolution enables the evolution of parallel code. This can be done by inserting some thread control instructions on the instruction set. Another important acceleration can be obtained through the complete implementation of the QILGP code into the GPU, and not only the evaluation of individuals. The quantum operator could be applied independently at each gene, and the observation process of classical individuals could benefit from aspects of this implementation.

7. REFERENCES

[1] D. Andre and J. Koza. *Parallel genetic programming: a scalable implementation using the transputer network architecture*, pages 317–337. MIT Press, Cambridge, MA, USA, 1996.

[2] F. Bennett III, J. Koza, J. Shipman, and O. Stiffelman. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In *Genetic and Evolutionary Computation Conference*, pages 1484–1490. Morgan Kaufmann, 1999.

[3] M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer-Verlag, 2007.

[4] D. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1566–1573, London, 2007. ACM Press.

[5] A. da Cruz, M. Vellasco, and M. Pacheco. Quantum-Inspired Evolutionary Algorithm for Numerical Optimization. In *Hybrid Evolutionary Algorithms*, volume 75 of *Studies in Computational Intelligence*, pages 19–37. Springer, 2007.

[6] A. da Cruz, M. Vellasco, and M. Pacheco. Quantum-Inspired Evolutionary Algorithms applied to numerical optimization problems. In *Congress on Evolutionary Computation*, pages 1 –6, 2010.

[7] D. Dias and M. Pacheco. Toward a quantum-inspired linear genetic programming model. In *Congress on Evolutionary Computation*, pages 1691 –1698, 2009.

[8] D. Dias and M. Pacheco. Quantum-Inspired Linear Genetic Programming. Submitted to Journal of Genetic Programming and Evolvable Machines, 2010.

[9] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[10] K.-H. Han and J.-H. Kim. Genetic quantum algorithm and its application to combinatorial optimization problem. In *Congress on Evolutionary Computation*, volume 2, pages 1354 –1360, 2000.

[11] S. Harding and W. Banzhaf. Fast Genetic Programming on GPUs. In *Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2007.

[12] S. L. Harding and W. Banzhaf. Distributed genetic programming on GPUs using CUDA. In *Workshop on Parallel Architectures and Bioinspired Algorithms*, Raleigh, USA, 2009.

[13] Y. Kim, J.-H. Kim, and K.-H. Han. Quantum-inspired Multiobjective Evolutionary Algorithm for Multiobjective 0/1 Knapsack Problems. In *Congress on Evolutionary Computation*, pages 2601 –2606, 2006.

[14] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. The MIT press, 1992.

[15] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic programming IV: Routine human-competitive machine intelligence*. Springer, New York, NY, USA, 2005.

[16] W. Langdon and W. Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85. Springer, Naples, 2008.

[17] W. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In *IEEE Congress on Evolutionary Computation*, pages 1 –8, 2010.

[18] W. Langdon and A. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 12:1169–1183, 2008.

[19] B. Lanyon, M. Barbieri, M. Almeida, T. Jennewein, T. Ralph, K. Resch, G. Pryde, J. O'Brien, A. Gilchrist, and A. White. Quantum computing using shortcuts through higher dimensions. *arXiv:0804.0272 [quant-ph]*, 2008.

[20] N. Nedjah, L. Dos Santos Coelho, and L. De Macedo Mourelle, editors. *Quantum Inspired Intelligent Systems*. Studies in Computational Intelligence. Springer, Berlin, 2008.

[21] P. Nordin. AIMGP: A formal description. In *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, WI, USA, 1998. Stanford University Bookstore.

[22] Nvidia. *The CUDA Compiler Driver NVCC*, 2010.

[23] J. Page, R. Poli, and W. B. Langdon. Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study. In *Second European Workshop on Genetic Programming*, pages 39–48, London, UK, 1999. Springer-Verlag.

[24] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com`, 2008. (contributions by J. R. Koza).

[25] D. Robilliard, V. Marion, and C. Fonlupt. High performance genetic programming on GPU. In *Proceedings of the 2009 workshop on Bio-inspired algorithms for distributed systems*, BADS '09, pages 85–94, New York, NY, USA, 2009. ACM.

[26] A. Salhi, H. Glaser, and D. D. Roure. Parallel implementation of a genetic-programming based tool for symbolic regression. *Information Processing Letters*, 66(6):299 – 307, 1998.

[27] L. Spector. *Automatic Quantum Computer Programming – A Genetic Programming Approach*. Springer, Boston, USA, 2004.

[28] I. Turton, S. Openshaw, and G. Diplock. Some geographic applications of genetic programming on the Cray T3D supercomputer. In *UK Parallel'96*, pages 135–150, University of Surrey, 1996. Springer.

[29] G. Wilson and W. Banzhaf. Linear genetic programming GPGPU on Microsoft's Xbox 360. In *Congress on Evolutionary Computation*, pages 378 –385, 2008.