High Performance Genetic Programming on GPU

Denis Robilliard
Université Lille Nord de France
LIL, 50 rue Ferdinand Buisson
BP 719, 62228 Calais, France
robillia@lil.univ-littoral.fr

Virginie Marion
Université Lille Nord de France
LIL, 50 rue Ferdinand Buisson
BP 719, 62228 Calais, France
poty@lil.univ-littoral.fr

Cyril Fonlupt
Université Lille Nord de France
LIL, 50 rue Ferdinand Buisson
BP 719, 62228 Calais, France
fonlupt@lil.univ-littoral.fr

ABSTRACT

The availability of low cost powerful parallel graphics cards has stimulated the port of Genetic Programming (GP) on Graphics Processing Units (GPUs). Our work focuses on the possibilities offered by Nvidia G80 GPUs when programmed in the CUDA language. We compare two parallelization schemes that evaluate several GP programs in parallel. We show that the fine grain distribution of computations over the elementary processors greatly impacts performances. We also present memory and representation optimizations that further enhance computation speed, up to 2.8 billion GP operations per second. The code has been developed with the well known ECJ library.

Categories and Subject Descriptors

I.2 [**Automatic Programming**]: Program modification

General Terms

Algorithms

Keywords

genetic algorithms, genetic programming, graphics processing units, parallel processing

1. INTRODUCTION

It is well known that the most time consuming part of a Genetic Programming (GP) run is the evaluation process. When dealing with complex real world problems like those addressed by Koza et al. in their last book [17], millions of programs need to be evaluated at each generation. Even if the training set is small, this huge population number makes GP runs impractical on common systems. A common strategy to tackle this problem is to parallelize or distribute the GP computations, e.g. [24, 13, 7, 4]. Newly introduced graphics processing units (GPUs) provide fast parallel hardware for a fraction of the cost of a traditional parallel system.

BADS'09, June 19, 2009, Barcelona, Spain.

GPUs are designed to efficiently compute graphics primitives in parallel to produce pixels of the video screen. Driven by ever increasing requirements from the video game industry, recent GPUs are very powerful and flexible processors, while their price remains in the range of mass consumer market. They now offer floating-point calculation much faster than today's CPU and, beyond graphics applications, they are able to address general problems that can be expressed as data-parallel computations.

Several general purpose high-level languages for GPUs have become available such as Brook, RapidMind, Accelerator from Microsoft Research, PeakStream or OpenCL. Thus developers do not need any more to master the extra complexity of graphics programming APIs when they design non graphics applications on GPUs (see `http://www.gpgpu.org` for a survey). With the G80 family GPUs, NVidia proposes the CUDA development kit (Compute Unified Device Architecture) that is based on a C-like general purpose language, allowing fine control over the hardware capabilities.

Evolutionary computation concepts were born in the 60s, inspired by Darwinian adaptation. They became popular with John Holland and David Goldberg's pioneering work on Genetic Algorithms (GA) [12, 8]. GAs have been successfully applied to a wide spectrum of problems ranging from combinatorial optimization (routing, assignment, scheduling,...) to game playing or robot control [6]. A derived scheme, Genetic Programming (GP) was introduced in 1992 by Koza [15]. In GP, programs are usually represented as trees. Leaves nodes are taken from a *terminal set*, usually composed of pseudo-variables storing the program inputs and random constants. The internal nodes are functions symbols operating on their subtrees. A GP run simulates the evolution of a population of programs by repeatedly evaluating programs quality, selecting the fitter programs, and then breeding (i.e. combining) them to produce hopefully better new programs. The evaluation phase is typically done in a supervised learning framework, running a program on a set of training (or fitness) cases and comparing the computed outputs versus the expected ones. Selection is usually stochastic with a bias towards best individuals. Breeding is done by applying operators that mimic the two main genetic transformations: cross-over and mutation. In the case of tree-coded programs, cross-over is performed by selecting two programs and exchanging sub-trees between them, and a typical mutation consists in randomly choosing a branch of an individual, deleting it and replacing it with a new random branch. The new generation replaces the previous one, and the overall process is iterated until some termination crite-

rion is satisfied (e.g. exhaustion of computation time or a satisfying solution found). GP has known a growing interest since its introduction and has been used to solve numerous problems including classification, robotics, and electrical engineering (see [2] for a survey).

Exploiting the power of GPUs within the framework of evolutionary computation has been done first for genetic algorithms, e.g. [25, 26, 22, 14]. Then implementation schemes of Genetic Programming on GPU have been recently released, based either on dynamic compilation of GP individuals [11, 10, 5] or on interpretation of the GP programs [19, 23]. Some GP application papers that take advantage of GPU power have already been issued [20, 9, 1] (see also the General Purpose Genetic Programming on GPU site (http://www.gpgpgpu.org).

In a previous work [23], we proposed a parallelization scheme based on interpretation rather than compilation of GP programs. The interpreter allows to evaluate simultaneously several different GP individuals in parallel, delivering speedups even for small training sets and as a consequence more computational power is available to increase the population size. A similar approach was proposed independently by Langdon and Banzhaf in [19].

In this paper we show that high performance GP on GPU requires access to fine grain details of the architecture, and thus may dictate the choice of the programming language and development kit. To illustrate this fact, we first compare two parallelization schemes that only differ by the way they dispatch evaluation of GP programs on elementary threads, and then we turn to the impact of memory optimizations. Depending on these low level implementation choices, performances are subject to important variations. Such critical points may be concealed to the developer using a too high level GPU development kit, thus limiting the gains offered by new GPUs.

All our experiments are done with the popular ECJ evolutionary library [21] and the CUDA language, i.e. only the evaluation of fitness cases is performed on the GPU and programmed in CUDA, while the rest of the evolutionary process is done by ECJ on the CPU. We need to transfer GP solutions and fitness cases from Java to CUDA, which is done via the Java Native Interface (JNI). Note that even if this paper is primarily targeted for Nvidia G80 and G90 GPUs, some of the concepts developed here are portable onto ATI graphic hardware, since both ATI and Nvidia new graphic cards share quite similar architectures. For instance, in the case of the R600 graphics processing unit from ATI, each SIMD array of 80 stream processors has its own sequencer and arbiter and so is close to the multiprocessor concept implemented by Nvidia.

The rest of the paper is organized in the following way: next section presents the basic traits of the G80 GPU family and the CUDA programming language. In Sect. 3 we present two different parallelization schemes in CUDA, managing computations either at the level of elementary processors or at the level of multiprocessors. In Sect. 4 we present our experimental setup and compare the performance of the two parallel schemes. Section 5 discusses memory and representation optimizations, and Section 6 unveils final performances and discusses future issues.

2. G80 GPU AND CUDA LANGUAGE

The graphics card we consider is an NVidia GeForce 8800 GTX based on the G80 GPU. It is natively limited to single precision floating point (32-bit data precision), although double precision can be used through a software library.

This hardware is based on a unified architecture managed as a pool of 16 so-called *multiprocessors*. A multiprocessor contains 8 elementary stream processors that operate at 1.35 Ghz clock rate, giving a total number of 128 stream processors on the graphics card. A multiprocessor also owns 16 kb of fast memory that can be shared by its 8 stream processors, 8 kb of texture and constant cache and an independent program counter.

Multiprocessors are SIMD devices, meaning their inner 8 stream processors execute the same instruction at every time step. However alternative and loop structures are available: if a stream processor should not perform a given instruction because e.g. the conditional expression of a *while* structure results as false when computed on its own data, then this stream processor is simply put into idle mode during the remaining loops performed by the others. This is called *divergence*, and of course it implies some waste of computing power.

At the level of multiprocessors the G80 GPU works in SPMD mode (Single Program, Multiple Data) : every multiprocessor must run the same program, but they do not need to execute the same instruction at the same time step (as opposed to their internal stream processors), because each of them owns its private program counter.

The execution model supported by the architecture (and also by the CUDA language) relies on 2 main concepts: *threads* and *blocks*. Threads can be roughly viewed as the smallest elementary computation processes on the parallel device. Their order of execution is not known in advance, this scheduling being used notably to amortize memory access delays. A block is a set of threads being executed on a given multiprocessor. If there are enough multiprocessors all the blocks are executed in parallel, otherwise a scheduler manages them and their order of execution is not known in advance. A block cannot access the fast memory bank or registers of another block. The number of threads in a block must be a multiple of 32 on the G80, thus there are always more threads than stream processors in a multiprocessor.

The CUDA language is an extension of the C language, it is free software although proprietary of Nvidia and several general purpose libraries are available (such as linear algebra, FFT, ...)[1]. Its main drawback is that it runs only on Nvidia hardware from the G80 and up, but it offers fine grain access to the architecture. As usual for GPU software toolkits, programs are divided in two parts, one runs on the *host* CPU and the other on the GPU *device*, this part of code being called *kernel* in the CUDA jargon. The host code is generally responsible for transferring data and loading the kernel code to the graphics card memory, performing input and output (with the obvious exception of graphics display), and calling the kernel code.

Contrary to high-level GPU programming toolkits, such as RapidMind, that create and dispatch automatically threads over the underlying elementary processors without user control, the CUDA toolkit allows the programmer to request the block and thread identifiers at run time. Thus the program-

[1]see http://www.nvidia.com/object/cuda_home.html

mer can choose to associate particular tasks to given blocks or threads. In doing so we do not control on which multiprocessor or elementary processor the task will run, but we can e.g. insure that different tasks will not be computed in parallel on the same multiprocessor. That would not be important on a pure SIMD computing device, but this allows to take full advantage of the SPMD mode offered by recent GPUs, as explained in the next section.

3. POPULATION PARALLEL MODELS

First we discuss GP program compilation versus interpretation issues, then we present two parallel models where several individuals from a given generation are interpreted simultaneously.

3.1 Compilation versus interpretation

Harding et al. and Chitty's works [11, 10, 5] were based on the same approach: at any given time there is only one compiled GP individual being evaluated (i.e. executed) and its evaluation is done in parallel on the fitness cases. This process is iterated on every individual, until the whole population has been evaluated.

Of course, as the G80 is an SPMD device we cannot perform the direct execution of several *different* GP programs in parallel. Anyway emulating MIMD tasks can be done in the form on an interpreter that considers the set of programs as data. This solution was chosen by Juillé and Pollack when they parallelized GP on the MASPAR machine [13]. We showed in [23] that it yields speedups even for a small number of fitness cases and short programs and it was also successfully used by Langdon and Banzhaf in [19].

There is clearly a trade-off choice: the cost of iterating interpreted code on the training cases is to be balanced against the compilation overhead and reduced cost of iterating compiled instructions. If there are few training cases, meaning few iterations, then the interpreter is a sensible solution anyway. Moreover if we execute one GP program at a time (either compiled or interpreted), then we parallelize only the training data, and we might well have not enough data to fill all the ALU pipelines of the elementary stream processors. This would leave the GPU under-exploited, especially since new generation GPUs have several hundreds elementary processors. On the contrary interpreting several programs in parallel increases the computation load. We choose this interpreter approach here.

3.2 BlockGP scheme

In [23] we proposed a parallel model where, at any time, each multiprocessor interprets only one GP tree. That is, GP trees are parallelized at the level of multiprocessors, thus giving up to 16 GP programs interpreted in parallel on the G80. The fitness computation of a given tree is in turn parallelized on the 32 threads running on one multiprocessor, that are scheduled on its 8 stream processors. This scheme is illustrated in Figure 1. So every stream processor evaluates around $1/8^{th}$ of the training cases (variations may occur due to scheduling). We deem the $1/8^{th}$ factor leaves enough data to fill the ALU pipelines in most cases, even with small training sets.

To highlight the main characteristics of the *BlockGP* scheme:

- every GP program is interpreted by all threads running on a given multiprocessor;

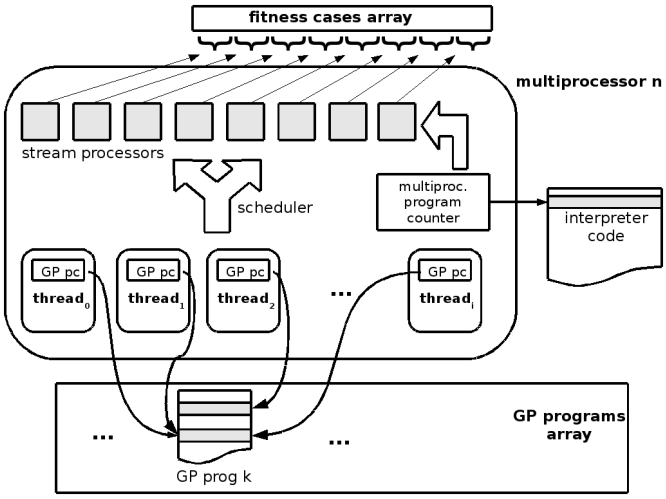

Figure 1: BlockGP parallelization scheme: each multiprocessor evaluates a different program. Each thread manages a software instruction pointer for interpretation of the program and evaluates a part of the fitness cases. Depending on the function set, threads may diverge and interpret different instructions as illustrated here.

- the fitness cases are processed in parallel on the 8 stream processors, so the computational intensity is higher than in a compiled scheme where they would be dispatched on 128 stream processors;

- divergence can occur between stream processors on the same multiprocessor, when e.g. an *if* structure resolves into execution of different branches within the set of fitness cases that are processed in parallel. However, there is no divergence due to differences between GP programs, since they are interpreted on independent multiprocessors, thus we take advantage of the SPMD architecture.

3.3 ThreadGP scheme

In the case of GPUs, we did not cover in our previous paper another and perhaps more natural way of parallelizing GP programs. This alternative scheme consists in each thread interpreting its own GP program. In this case, the total number of threads on the GPU is the population size and each thread evaluates its program on every fitness cases, so even with few fitness cases, the arithmetic intensity will be high. For the G80, threads are evenly spread in blocks respecting the execution model constraints (we use 32 threads per block), and blocks are automatically scheduled on the multiprocessors, as illustrated in Figure 2.

This scheme can be seen as a straight SIMD implementation, in the sense that we do not care about the existence of multiprocessors having their independent program counter. It is possible that development kits that do not give access to detailed management of threads, are indeed using this scheme when doing automatic dispatch of evaluation on the stream processors. For the sake of simplicity, we refer to this scheme as *ThreadGP*.

To highlight the main characteristics of ThreadGP:

- every GP program is interpreted by its own thread;
- this scheme is more computationally intensive than BlockGP for small training sets;
- as several threads interpreting different programs are run on each multiprocessor, the level of divergence could be a problem.

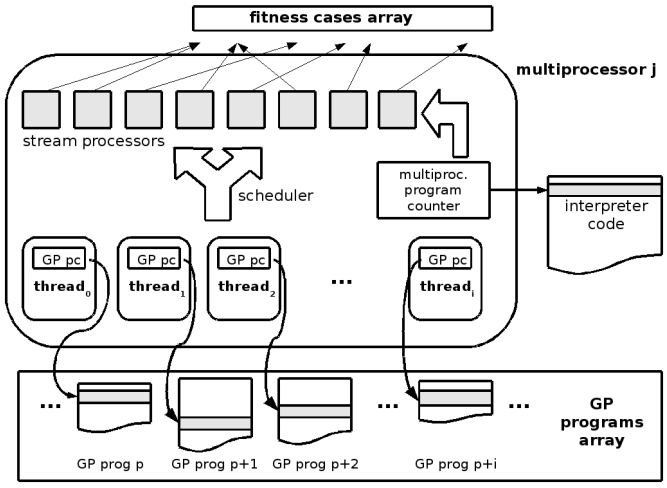

Figure 2: ThreadGP parallelization scheme on the G80: each thread is in charge of one GP program, and manages a software instruction pointer for its interpretation, processing all fitness cases.

In Section 4 we compare the efficiency of these two parallel schemes on several benchmarks.

4. EXPERIMENTS

Before examining performance timings, we describe our experimental framework.

4.1 Setup

Our test machine is equipped with a graphics card dedicated to the display, while the 8800GTX card is reserved for the computations and thus not attached to an X server. This dual cards setting allows to get cleaner timings (no interference with the display). Note that it is possible to use the 8800GTX for both display and GP evolution, although with constraints: during intensive computation, the user interaction with the X desktop is suspended; moreover any given call to the GPU (i.e. executing the interpreter in our case) cannot last more than 5 seconds, otherwise the process is killed by the X server watchdog process.

We assess the performance of our parallel schemes for three standard benchmarks taken from [16, 15]: sextic symbolic regression $x^6 - 2x^4 + x^2$, boolean 6- and 11- multiplexers, and intertwined spirals classification (see Table 1 and [23]). We do not focus on GP ability to solve these standard benchmarks — this has been amply covered in the GP literature — but rather on computing time performance. Running times have been obtained through the monitoring utilities of the ECJ library. In order to obtain significant figures, 30 independent evolutionary runs have been executed for each problem and the running times have been averaged.

4.2 Parallel schemes comparison

In this part we compare the performance of the two parallelization schemes presented in Section 3, BlockGP and ThreadGP, on the benchmark problems defined above. Selection and breeding phases are not impacted by our parallelization, so time comparisons are done only on the evaluation phase.

The results of BlockGP and ThreadGP are summed up in Tab. 2 and Tab. 3. The speedup is the ratio of mean ThreadGP running time over mean BlockGP running time, and is illustrated in Fig. 3 and 4: notice that it is always greater than 1, i.e. BlockGP is always faster than ThreadGP.

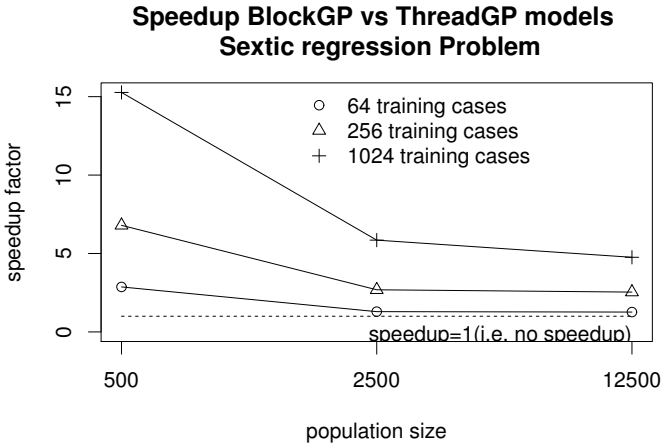

Figure 3: BlockGP versus ThreadGP speedup for the evaluation phase on sextic polynomial regression problem.

These results show that BlockGP outperforms ThreadGP, whatever the benchmark, the population size and the number of fitness cases. It obviously pays to limit divergence by dispatching similar computations on the same blocks (i.e. multiprocessors). In particular, in the case of the sextic polynomial, running the same program in parallel on several fitness cases onto a single block yields no divergence at all, thus the larger gains for BlockGP versus ThreadGP.

In the other problems (multiplexers and spirals), part of the function set is implemented as short-cut operators: e.g. an *if* operator evaluates either its *then* or *else* subtrees depending on the value of its condition subtree. Thus the function set creates divergence even in the case of the BlockGP scheme. Nonetheless BlockGP is at least roughly twice faster as it does not suffer from extra divergence due to execution of different programs on the same blocks.

Notice that in the ThreadGP scheme you need to interpret at least 512 programs (32 threads times 16 multiprocessors) in order to have all threads contributing. This may explain part of the degraded performance of ThreadGP for a population of only 500 individuals.

These results question the way how several GPU programming toolkits do their automatic parallelization.

Table 1: Presentation of benchmark problems.

	sextic regression	**6-multiplexer**	**11-multiplexer**	**intertwined spirals**
# fitness cases	{64, 256, 1024}	64	2048	194
Terminals	{X, ERCs}	{A0-A1,D0-D3}	{A0-A2, D0-D7 }	{X, Y, ERCs}
Operators	{+,-,*,/,sin, cos, exp, log}	{ And, Or, Not, If }		{+, -, *, /, sin, cos, Iflte}
pop. size	{500, 2500, 12500}			
generations	50			

Table 2: Sextic polynomial regression $x^6 - 2x^4 + x^2$. Comparison of evaluation time between runs with one individual per block and runs with one individual per thread. Times are given in seconds and averaged for 30 runs.

Pop. size	Training set size					
	64		256		1024	
	Block	Thread	Block	Thread	Block	Thread
500	**0.53**	1.52	**0.64**	4.35	**1.08**	16.49
2500	**2.53**	3.27	**3.03**	8.14	**5.35**	31.31
12500	**16.05**	20.31	**16.65**	42.28	**27.55**	131.09

Table 3: 6-multiplexer, 11-multiplexer and intertwined spirals. Comparison of evaluation time between runs with one individual per block and runs with one individual per thread. Times are given in seconds and averaged for 30 runs.

Pop. size	6-multiplexer		11-multiplexer		Intert. Spirals	
	Training set size 64		Training set size 2048		Training set size 194	
	Block	Thread	Block	Thread	Block	Thread
500	**0.77**	3.68	**1.94**	11.43	**1.97**	7.92
2500	**3.89**	7.46	**7.56**	16.32	**8.63**	15.32
12500	**13.66**	30.67	**33.78**	69.77	**42.96**	74.87

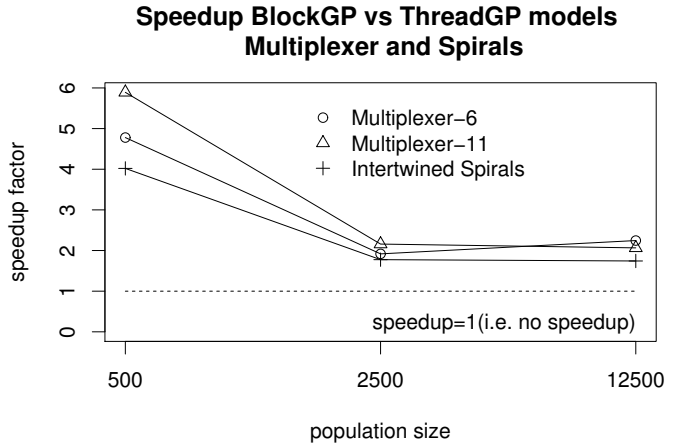

Figure 4: BlockGP versus ThreadGP speedup for the evaluation phase on 6-multiplexer, 11-multiplexer and intertwined spirals (64, 2048 and 194 training cases respectively).

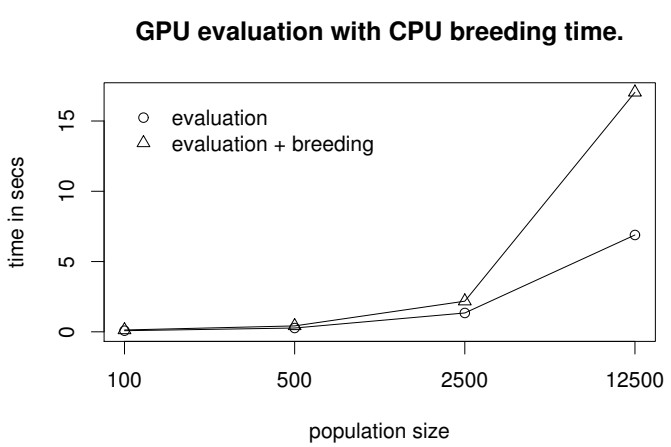

Figure 5: Mean evaluation and breeding time for the GPU runs on the $x^6 - 2x^4 + x^2$ regression problem, with 1024 cases. As breeding is kept on the CPU, it becomes the bottleneck when processing large populations.

5. CODE OPTIMIZATIONS

We now focuss on the faster BlockGP scheme.

Removing tree to postfixed translation. In our previous work [23], we used the standard ECJ tree representation. We needed a translation phase to convert trees to postfixed notation for interpretation of GP programs on the GPU, since it would have been very impractical to transfer pointer-based trees from CPU memory to GPU memory.

The breeding phase duration is usually unimportant since it is heavily dominated by the evaluation phase in GP runs. However the BlockGP evaluation speedup on non-diverging problems is such that breeding phase may become the bottleneck, as illustrated on Figure 5.

One obvious improvement to this situation is to modify the ECJ evolution engine in order to directly evolve linear postfixed trees (also denoted RPN for Reversed Polish Notation) in order to skip the translation phase and be able to perform a direct copy of the population onto the graphics card memory. This RPN evolution code is based along the lines proposed by W. B. Langdon in [18], that is:

- an individual is stored as a byte array, each byte coding an operator or an operand;
- we use a set of fixed ERCs (either random or predefined depending on the problem). Each ERC is associated to a given byte-code. A similar representation was also used for linear genetic programming by Brameier and Banzhaf [3];
- this representation leads to define a maximum length for individuals, or else the insertion of mutated inner subtrees would imply to shift arbitrary large array slices and thus would downgrade performances.

This scheme creates some incompatibilities with the rest of ECJ code, nonetheless it uses the standard breeding pipeline and evolutionary framework. It has the advantage of needing far fewer allocations than in the standard ECJ representation, and since memory management is costly in Java, this yields a reduced breeding time.

Table 4: Comparison of breeding time between standard ECJ tree representation and RPN representation for 11-multiplexer problem, with 768 instructions length individuals. Times are given in seconds and averaged for 30 runs. Both schemes are executed on the CPU.

	Population Size			
	100	500	2500	12500
std tree	0.18	0.98	7.29	57.71
RPN	**0.13**	**0.33**	**1.34**	**8.04**

As breeding is quite independent of the problem (for similar sizes of evolved trees), we illustrate results only for the 11-multiplexer, as an example of what can be expected in general. In Tab. 4 and Fig. 6, we show breeding times and speedups for both representations, standard ECJ trees and RPN "flat" trees: improvements are striking and the larger the population, the bigger the speedup.

Part of the gain is due to the much reduced number of object creations and deletions, so such reductions in breeding time are of course not guaranteed in other languages, such as C++, where object management or pointed tree transformations may be cheaper.

Memory access optimization. Memory management is another quite critical point to handle. Since program instructions will be accessed repeatedly for every fitness case, they qualify as a priority target for memory optimization. We choose to use shared memory, which is on-chip and is accessed significantly faster than the graphics card global memory. As we interpret only one program per multiprocessor in the BlockGP scheme, this means up to 16Kb are available per program. It seems more than enough for running a

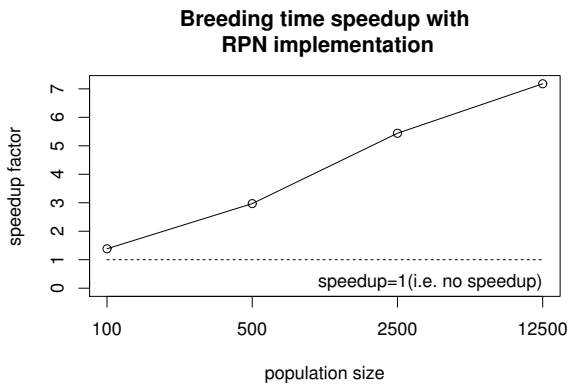

Figure 6: RPN representation versus standard ECJ trees speedup for breeding phase on 11-multiplexer, with 768 instructions length individuals. Both schemes are executed on the CPU.

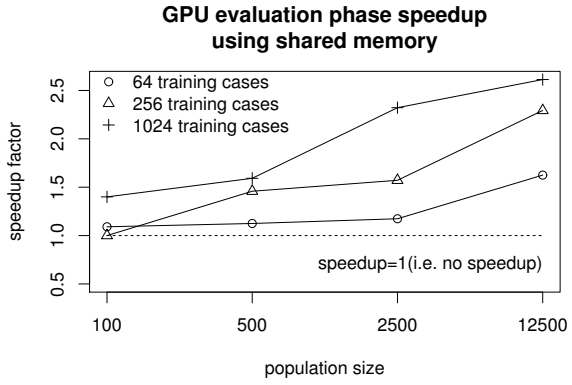

Figure 7: Evaluation time speedup on GPU using shared memory versus using global memory. Experiments are done on sextic polynomial regression, with 256 instructions length individuals.

majority of the GP experiments from the literature.

We can expect the efficiency of such a cache technique to be roughly independent of the problem, up to the size of evolved trees, and we present results for the sextic polynomial regression in Tab. 5 to illustrate its potential, with speedup plots in Fig. 7. All runs show improvement up to roughly three times faster. The bigger the fitness cases set, the more loops over the same cached programs, the better the improvement as expected. Notice that the speedup also increases with the population size: this is due to memory transfer overheads that becomes negligible against the computing time with more individuals.

Again, we see that having access to architecture details can greatly enhance the performance of algorithms, and thus can influence the choice of the GPU programming toolkit.

6. CONCLUSIONS

In Table 6, we report the final performances of our optimized BlockGP scheme (i.e. with RPN representation and cached individuals). To our three previous benchmarks, we add the MackeyGlass regression problem as defined by Langdon et al. in [19], in order to allow comparison with their implementation based on the RapidMind high-level toolkit. Following a suggestion from [19], we state the performance in terms of the number of GP operations per second, denoted GPop/s, that measures how many GP nodes have been computed per second. To clarify this notation, let us suppose a run duration of 1 second for 5 generations with a population of 2 trees, each of 10 nodes, evaluated on 12 fitness cases. This hypothetic run would exhibit a performance of: $(5 \times 2 \times 10 \times 12)/1 = 1200$ GPop/s. We also give the speedup versus CPU, i.e. how many times the GPU version is faster than the original ECJ code running on a 2.6Ghz processor.

Among our benchmarks, the top performance was achieved on the sextic polynomial regression, for 1024 fitness cases and 12500 individuals, with a value of $2.8E+9$ GPop/s. For the multiplexer and spirals problems, one can see that the number of evaluated nodes is much smaller than the raw size of programs due to the short-cut operators. The resulting divergence lowers the performance in comparison with the non diverging sextic experiment, although it remains an order of magnitude faster than the original CPU version. We notice a more than 70% increase in performance on the MackeyGlass versus the experiments reported in [19] that were implemented in Rapidmind, a toolkit that does not allow a fine control of the underlying architecture. This benchmark could not be run with standard ECJ trees due to memory limitation with this population size.

From our experiments, we can draw some lessons for optimizing GP on GPU:

- speedup depends on the problem and more particularly on the presence of operators that create divergence between stream processors (e.g. `if` operator);
- for non diverging operators GPU speed tends to move away the evolutionary bottleneck from evaluation to breeding;
- dispatching one GP program per multiprocessor limits the divergence and thus it is superior to the scheme that simply associates one program to each thread, even for problems that implement operators with unavoidable divergence;
- optimizing the allocation of data on specific high speed memory yields significant benefits.

These two last points question the way high-level GPU toolkits do their automatic parallelization of computations. Explicit management of the multiprocessors could become part a standard API since GPU makers rely on a similar architecture for their most powerful chipsets.

SPMD architectures like the G80, built around a set of SIMD cores, exhibit a lower complexity than true multi-core systems, and are currently very competitive in term of price. Since they offer a significant gain in computing power versus current CPU, we think that GPU processing is to be taken into account by artificial evolution practitioner, at least in the mid-term future. Working towards implementing a full GP algorithm on GPU is also a current objective.

7. REFERENCES

[1] D. T. Anderson, R. H. Luke, and J. M. Keller. Speedup of fuzzy clustering through stream processing

Table 5: Comparison of evaluation times on GPU with and without caching instructions in shared memory. Timings are given in seconds and averaged for 30 runs on the sextic polynomial regression, with 256 instructions length individuals.

Population size	Training set size					
	64		256		1024	
	Without	With Cache	Without	With Cache	Without	With Cache
100	0.12	**0.11**	0.16	0.16	0.35	**0.25**
500	0.18	**0.16**	0.35	**0.24**	1.21	**0.76**
2500	0.54	**0.46**	1.43	**0.91**	5.48	**2.36**
12500	2.73	**1.68**	8.21	**3.58**	29.58	**11.32**

Table 6: Full run benchmarks — Measured speed in million of GPops^{-1}. Columns are respectively : terminals set size + number of ERCs, non terminal function set size, population size, average program size, average evaluated nodes, number of fitness cases, speed, speedup versus CPU.

Experiment	|T|+ERCs	|F|	pop	prog. size	eval. nodes	cases	speed ($\times 10^6$ GPops^{-1})	speedup
Sextic	1+10	8	12500	67.22	67.22	1024	2797	144
Mult-11	11+0	4	12500	156.24	24	2048	501	10.4
Inter. Spir.	2+10	7	12500	163.68	82.95	194	212	13.7
MackeyGlass	8+128	4	204800	10.22	10.22	1200	1720	–

on graphics processing units. In J. Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, pages 1101 – 1106, Hong Kong, 2008. IEEE Press.

[2] W. Banzhaf, P. Nordin, R. Keller, and F. Francone. *Genetic Programming An Introduction*. Morgan Kaufmann, 1999.

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

[4] S. M. Cheang, K. S. Leung, and K. H. Lee. Genetic parallel programming: Design and implementation. *Evolutionary Computation*, 14(2):129–156, Summer 2006.

[5] D. M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In *Proceedings of the 2007 Genetic and Evolutionary Computing Conference (GECCO'07)*, pages 1566–1573, London, UK, July 2007. ACM Press.

[6] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[7] F. Fernandez, M. Tomassini, and L. Vanneschi. An empirical study of multipopulation genetic programming. *Genetic Programming and Evolvable Machines*, 4(1):21–51, Mar. 2003.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, 1989.

[9] S. Harding. Evolution of image filters on graphics processor units using cartesian genetic programming. In J. Wang., editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[10] S. Harding and W. Banzhaf. Fast genetic programming and artificial developmental systems on GPUs. In *proceedings of the 2007 High Performance Computing and Simulation (HPCS'07) Conference*, page 2. IEEE Computer Society, 2007.

[11] S. Harding and W. Banzhaf. Fast genetic programming on GPUs. In *proceedings of the 10th European Conference on Genetic Programming, EuroGP 2007*, volume 4445 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2007.

[12] J. H. Holland. *Adaptation in Natural and Artificial Systems*. Michigan Press University, 1975.

[13] H. Juillé and J. B. Pollack. Massively parallel genetic programming. In *Advances in Genetic Programming 2*, chapter 17, pages 339–358. MIT Press, 1996.

[14] K. Kaul and C.-A. Bohn. A genetic texture packing algorithm on a graphical processing unit. In *Proceedings of the 9th International Conference on Computer Graphics and Artificial Intelligence*, 2006.

[15] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[16] J. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press, 1994.

[17] J. Koza, M. Keane, M. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[18] W. B. Langdon. Evolving programs on graphics cards — C++ code. Available at http://www.cs.ucl.ac.uk/external/W.Langdon/ftp/gp-code/gpu_gp_1.tar.gz, 2008.

[19] W. B. Langdon and W. Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85, Naples, 26-28 Mar. 2008. Springer.

[20] W. B. Langdon and A. P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*, 2008. Special Issue. On line first.

[21] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, E. Popovici, K. Sullivan, J. Harrison, J. Bassett, R. Hubley, and A. Chircop. ECJ 18 — a Java-based evolutionary computation research system. Available at http://cs.gmu.edu/~eclab/projects/ecj/, 2008.

[22] Z. Luo and H. Liu. Cellular genetic algorithms and local search for 3-sat problem on graphic hardware. In *IEEE Congress on Evolutionary Computation — CEC 2006.*, pages 988–2992, 2006.

[23] D. Robilliard, V. Marion-Poty, and C. Fonlupt. Population parallel GP on the G80 GPU. In M. O'Neill, L. Vanneschi, S. Gustafson, A. I. Esparcia Alcazar, I. De Falco, A. Della Cioppa, and E. Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 98–109, Naples, 26-28 Mar. 2008. Springer.

[24] P. Tufts. Parallel case evaluation for genetic programming. In *1993 Lectures in Complex Systems*, volume VI of *Santa Fe Institute Studies in the Science of Complexity*, pages 591–596. Addison-Wesley, 1995.

[25] M. L. Wong, T. T. Wong, and K. L. Fok. Parallel evolutionary algorithms on graphics processing unit. In *Proceedings of IEEE Congress on Evolutionary Computation 2005 (CEC 2005)*, volume 3, pages 2286–2293, Edinburgh, UK, 2005. IEEE.

[26] Q. Yu, C. Chen, and Z. Pan. Parallel genetic algorithms on programmable graphics hardware. In *Advances in Natural Computation*, volume 3162 of *LNCS*, pages 1051–1059. Springer, 2005.