Population Parallel GP on the G80 GPU

Denis Robilliard, Virginie Marion-Poty, and Cyril Fonlupt

Laboratoire d'Informatique du Littoral,
Maison de la Recherche Blaise Pascal,
50 rue Ferdinand Buisson - BP 719, 62228 CALAIS Cedex, France
{robillia,poty,fonlupt}@lil.univ-littoral.fr

Abstract. The availability of low cost powerful parallel graphics cards has stimulated a trend to port GP on Graphics Processing Units (GPUs). Previous works on GPUs have shown evaluation phase speedups for large training cases sets. Using the CUDA language on the G80 GPU, we show it is possible to efficiently interpret several GP programs in parallel, thus obtaining speedups also for small training sets starting at less than 100 training cases. Our scheme was embedded in the well-known ECJ library, providing an easy entry point for owners of G80 GPUs.

1 Introduction

Newly introduced graphics processing units (GPUs) provide fast parallel hardware for a fraction of the cost of a traditional parallel system. GPUs are designed to efficiently compute graphics primitives in parallel in order to produce the pixels of the video screen. Driven by ever increasing requirements from the video game industry, GPUs have evolved into very powerful and flexible processors, while their price remained in the range of consumer market. They now offer floating-point calculation much faster than today's CPU and, beyond graphics applications, they are very well suited to address general problems that can be expressed as data-parallel computations (i.e. the same code is executed on many different data).

Moreover, several general purpose high level-languages for GPUs have become available such as Brook[1] and thus developers do not need any more to master the extra complexity of graphics programming APIs when they design non graphics applications[2]. In this paper, we work with an Nvidia GeForce 8800GTX graphics card that is built around the G80 GPU. We used an NVidia provided extension to the C language, named CUDA (Compute Unified Device Architecture) that runs on the G80 GPU family, allowing fine control over the hardware capabilities. Note that this toolkit is not available for other manufacturers hardware, and is not backward compatible with older Nvidia GPUs.

Up to now, exploiting the power of GPUs within the framework of evolutionary computation has been done mostly for genetic algorithms, e.g. [1,2,3,4,5]. At

[1] http://graphics.stanford.edu/projects/brookgpu/
[2] See `http://www.gpgpu.org` for a survey.

the time of the writing of this paper, using GPUs for Genetic Programming is still fresh matter: a first approach using Microsoft's Accelerator toolkit has been proposed by Harding and Banzhaf [6,7], a tutorial-like paper using a graphics API approach was issued by Chitty[8], and a technical report using the Rapid-Mind development kit has been made available by Langdon [9]. However we may expect a quickly increasing number of studies in the near future.

Harding and Banzhaf's and Chitty's works are both based on the same approach: every GP individual is compiled for the GPU native machine code and then evaluated on the fitness cases using the parallel ability of the GPU. This scheme is iterated on every individual, until the whole population has been evaluated. These authors have obtained interesting speedups but mainly for large individuals and/or several thousands fitness cases. In [6] this is acknowledged as a weakness of this scheme: "Many typical GP problems do not have large sets of fitness cases..." and "this leads to a gap between what we can realistically evaluate, and what we can evolve". We also think that evolving programs with hundreds of thousands training cases is not the most common setting in today's GP problems. For example, GP is often used to perform supervised classification, and it may be difficult to provide large sets of labeled training cases, noticeably when labeling requires human intervention like medical diagnosis. Moreover, a look at Koza's et al. last book (chapter 15 in [10]) suggests that solving real world problems with GP may be more in need of large populations (up to 5,000,000 individuals in [10]) than large data sets.

In order to also exploit the power of the GPU on training sets of modest size, we propose another parallelization scheme. Instead of evaluating sequentially the GP solutions, parallelizing the training cases, we share the parallel capacity of the GPU between GP programs and data. Thus we evaluate different GP programs in parallel, and assign to each of them a cluster of elementary processors to treat the training cases in parallel. This yields more data to fill the pipeline of each ALU of the GPU, in order to improve the efficiency. As a consequence more computational power is available for e.g. increasing the population size.

As a consequence we must emulate a MIMD task (running different programs) on a basically SIMD hardware. A solution to this problem has been proposed in the 1990s [11] in the form on an interpreter that considers the set of programs as data. This was implemented for GP in the late 90s by Juillé and Pollack [12] on the MASPAR machine, and a similar technique is also proposed by Langdon [9] on the G80 GPU. Our approach differs from Langdon's since we use the CUDA development kit that allows a finer grain access to the hardware. Thus we can exploit a characteristic of the G80: it runs in Single Program Multiple Data (SPMD) mode, rather than SIMD, i.e. elementary processors run the same program (the interpretor) in parallel but they are divided into clusters that share their own program counter. This gives the opportunity to achieve increased speedups, since e.g. a cluster can interpret the "if" branch of a test while another cluster treat independently the "else" branch. On the opposite, performing the same computation inside a cluster is also possible, but the two branches are processed sequentially in order to respect the SIMD constraint: this is called divergence and of course it is less efficient.

We interfaced our CUDA based evaluation with the popular ECJ library[3], and retained the most part of its flexibility. Our experiments have been done with the mainstream tree representation for GP individuals, using tutorial benchmarks taken from the ECJ library. The GPU speedup values that we are giving are for complete evolution runs and not only for the evaluation phase. Thus we hope these figures are close to the speedup readers may expect with their usual setting. An archive containing a sample code for a regression application is available at http://www-lil.univ-littoral.fr/~robillia/EuroGP08/gpuregression.tgz .

The rest of the paper is organized in the following way: next section provides some information on the graphics processor unit and the CUDA programming language. In Section 3, the implementation of the GP system is described. Section 4 presents benchmarks and results. Section 5 concludes and discusses future works.

2 The G80 GPU architecture overview

The graphics card we used is a 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 an unified architecture: instead of the traditional specialized vertex and fragment processors that are found on many graphics cards, here the elementary processors are identical and managed as a pool of 16 so-called multiprocessors. A multiprocessor contains 8 elementary scalar stream processors that operate at a 1.35 GHz clock rate, giving a total number of 128 elementary stream processors on the graphics card. A multiprocessor also owns 16 kb of fast memory that can be shared by its 8 stream processors. Multiprocessors are SIMD devices, meaning their 8 stream processors execute the same instruction at every time step on its own data. However alternative and loop structures can be programmed: if some stream processor, among the 8 that are contained in a multiprocessor, should not perform a given instruction because e.g. the conditional expression of an *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.

Due to its architecture, the G80 GPU is able to function in SPMD mode (Single Program, Multiple Data) at the level of the multiprocessors: every multiprocessor must run the same program, but each of them owns its private program counter, thus they do not need to execute the same instruction at the same time step (as opposed to their internal stream processors). This flexibility allows to avoid divergence between multiprocessors by carefully dispatching the tasks on them, but up to now it can only be accessed with the CUDA development kit

[3] http://cs.gmu.edu/~eclab/projects/ecj/

proposed by NVidia. Other toolkits consider this GPU only as a SIMD device containing 128 elementary processors, thus increasing the risk of divergence and wasting computing power. This is why we used CUDA when implementing our population parallel scheme.

Note that our machine was equipped with another graphics card dedicated to display the X screen, while the 8800GTX card was reserved for the computations and thus not attached to a X server. This dual cards setting allowed us to obtain cleaner timings (no interference with the display). Note that it is perfectly possible to use the 8800GTX for both display and GP evolution, with some 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.

3 Population parallel model

As said above, previous works about GP on GPU have demonstrated interesting speedups for very large training sets and/or programs. Indeed if we execute one GP program at a time on the G80, parallelizing only the training data as it is proposed in [6], then we do not have enough data to fill all ALU pipelines of the 128 stream processors, thus the GPU is under-exploited. This phenomenon, in addition to compilation overheads, may also explain the bad GPU timings observed by [6] on the 7300 GPU with small training cases sets. A possible solution is to evaluate several programs in parallel to increase the computation load.

As the G80 is a SPMD device, SP meaning *Single Program*, we cannot perform the direct execution of several *different* programs in parallel. The same problem arose for Juillé and Pollack when they implemented GP on the MASPAR machine [12] and they proposed to bypass this limitation by interpreting GP solutions. In the same way, we run one program on the GPU: an interpreter dedicated to execute any GP program for our benchmark problems. The GP programs are simply considered as data from the interpreter point of view. This is clearly a trade-off choice: the computing time of iterating interpreted code on training cases is to be balanced against the time of compiling and iterating a compiled code. Few training cases means few iterations thus the interpreter may be a sensible trade-off.

In order to interpret the GP programs, we first have to copy them into the graphics card memory. This is not straightforward, since we want to integrate the GPU evaluation inside the ECJ library, while retaining the most part of its flexibility. Indeed, GP tree nodes in ECJ are scattered into memory, so we need to compact them into a single chunk of memory that can be transfered to the GPU. We also translate the trees into linear stack-based postfixed notation code that will be easier to interpret, although it is not required. This is illustrated in Figure 1.

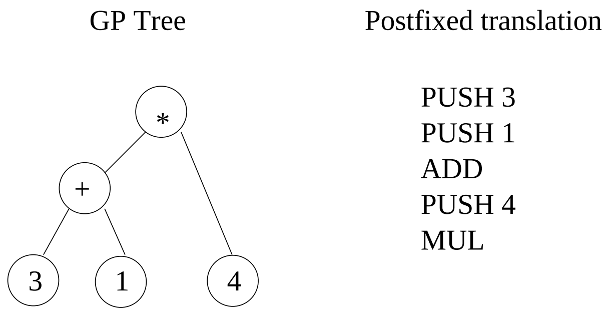

Fig. 1. Sample GP tree and its translation into linear postfixed notation, prior to its interpretation.

The interpreter code is run on the GPU and is quite simple, being composed of a main loop fetching the next instruction to process, and a switch that performs the operations required depending on this instruction, see pseudo-code in Table 1. The *if* structure and the short-circuit $\{And, Or\}$ operators are implemented as usual in code generation, i.e. bypassing branches that do not need to be evaluated (see [13]). A detailed GP oriented implementation is found in [12]. We use a stack to hold temporary results.

Table 1. Pseudo code of the interpreter.

```
sp = 0 ; // initialize stack pointer
GP_pc = baseAddress[i] ; // load base address of prog i
while (instructionArray[GP_pc] != RETURN) {
   switch (instructionArray[GP_pc]) {
      case OPERAND : stack[sp++] = data; // push data on stack
      case ADD : stack[sp-2] = stack[sp-1] + stack[sp-2]; sp-;
      case MUL : stack[sp-2] = stack[sp-1] * stack[sp-2]; sp-;
      ...
   }
   GP_pc++;
}
```

In order to limit the occurrences of divergence, we dispatch the population of GP trees in such a way that, at any time, each multiprocessor interprets only one GP tree. That is, GP trees are parallelized on the multiprocessors, giving up to 16 GP programs interpreted in parallel on the G80, and the fitness of a given tree is in turn parallelized on the 8 stream processors contained in the multiprocessor. This scheme is illustrated in Figure 2. So every stream processor evaluates $1/8^{th}$ of the training cases. This $1/8^{th}$ factor leaves enough data to fill the ALU pipelines in most cases, even with small training sets. In a scheme where only one GP program is run and only the training data are parallelized, each stream processor receives only $1/128^{th}$ of the training cases and this leads to under-exploitation with small training sets.

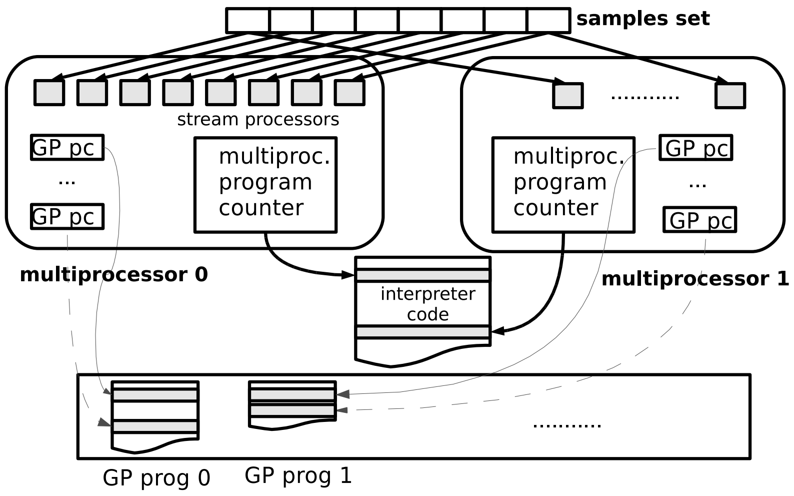

Fig. 2. Parallelization scheme: multiprocessors independently execute the interpreter code. On every multiprocessor, each stream processor handles a part of the training set and stores in register memory the current address of the GP program instruction to be interpreted (GP pc). Every GP pc do not need to point to the same part of the GP program. However, if the instructions to be interpreted in parallel are not the same for all the stream processors, this will imply divergence and loss of efficiency while some stream processors wait in idle mode.

To sum up some characteristics of our scheme:

– when the evaluation of a GP program is finished on a multiprocessor, there is no need to wait in idle mode for the completion of programs that are interpreted on other multiprocessors: another GP tree can be interpreted immediately; this is possible because we work in SPMD mode, versus the SIMD scheme proposed by [12,9];
– the same holds when two different programs contain *if* or *loop* instructions: this does not create divergence between programs;
– however we can incur divergence between stream processors on the same multiprocessor, as they always work in SIMD mode, when e.g. an *if* struc-

ture resolves into different cases within the set of 8 training cases that are processed in parallel[4].

4 Results and discussion

In this section, we assess the performance of our parallelization scheme on the G80 GPU against an Intel 2,6 GHz CPU (single core). We used three standard benchmarks taken from the ECJ library: real and boolean regression and a classification problem. Two of these benchmarks were also used by [6], although it is not possible to perform a direct comparison since we do not use the same hardware. Anyway, we do not focus on GP being able to solve these standard benchmarks — this has been covered in numerous previous works — but rather on the computing time speedup that can be brought by the GPU. Timings are monitored for the evaluation phase, that includes translation to postfixed code in the GPU case, and also for full evolutionary runs.

All runs were done using 32-bits floating point arithmetics on both CPU and GPU. We noticed small differences between the fitness values computed on both schemes. These differences were about 10^{-7} in magnitude and are implied by the parallelization scheme: the raw fitness is cumulated into a single loop on the CPU, while on the GPU each stream processor computes the fitness associated to a part of the training cases before the global result is cumulated. Thus small rounding errors appear that can change the result of the evolutionary selection phase, especially with the bigger populations where probabilities are higher to meet individuals with very close fitness values. These rounding errors tend to accumulate over generations and can yield slightly different runs between CPU and GPU. Here the situation is somewhat comparable to what happens when one performs a GP benchmark between machines with different precision levels. Thus, in order to obtain significant figures, we have done 30 independent evolutionary runs for each problem and setting, then we have averaged the running times. In turn, these average times are divided by the average evolved tree size observed respectively for the CPU and the GPU, in order to obtain a comparable time per node ratio. The speedup indicates how many times the GPU version is faster than the CPU one and is computed as:

$$\text{speedup} = \frac{\text{GPU mean tree size}}{\text{CPU mean tree size}} * \frac{\text{CPU mean running time}}{\text{GPU mean running time}}$$

The first benchmark is the standard regression problem $x^6-2x^4+x^2$ (see [14]), using population sizes of 100, 500, 2500 and 12500 individuals, 50 generations, and training set sizes of 64, 256 and 1024 training cases. The function set is {+, -, *, /, sin, cos, exp, log} and terminal set {ERC (i.e. Ephemeral Random

[4] Actually this is a bit more complex since CUDA schedules multiples of 4 computations per stream processor in order to amortize memory access overheads. A detailed explanation is not possible within the size constraints of this paper.

Constants), X}. Depending on the population and training set sizes, the average evolved tree size ranges from 30 to 66 nodes. Speedup figures are shown in Figure 3.

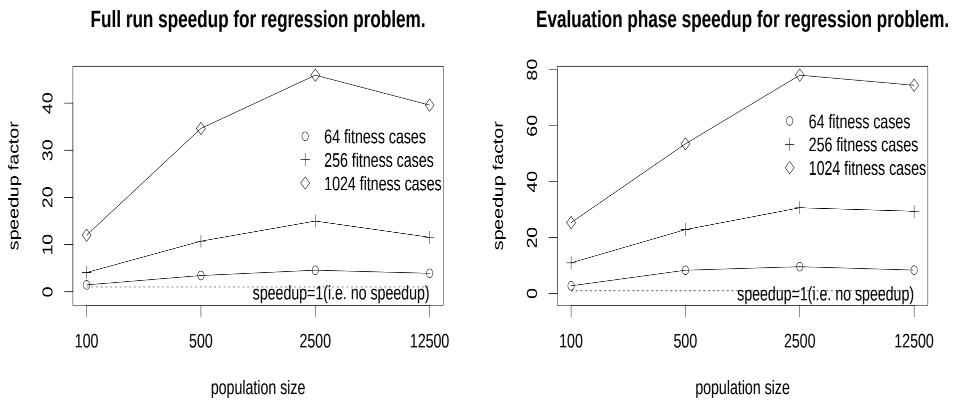

Fig. 3. GPU vs CPU speedup on regression problem $x^6-2x^4+x^2$. On the left, speedup for whole evolutionary runs, on the right speedup for evaluation phase only.

The second benchmark is based on the multiplexer-6 and multiplexer-11 problems (see [14]) with respectively 64 and 2048 training cases, for population sizes 100, 500, 2500 and 12500 individuals, and 50 generations. We used as function set {And, Or, Not, If} and terminal set {A0-A1, D0-D4}, resp. {A0-A2, D0-D7}. The "And" , "Or" and "If" are shortcut versions (i.e. bypassing branches that do not need to be evaluated) and boolean values are stored as integers, to obtain comparable results with the ECJ standard code. Depending on the population and training set sizes, the average evolved tree size ranges from 112 to 157 nodes. Speedup is illustrated in Figure 4.

The third benchmark is the intertwined spirals problem (see [15]), again population sizes range from 100 to 12500 individuals, and the training set size is fixed to 194. The function set is {+, -, *, /, sin, cos, Iflte[5]} and the terminal set is {ERC, X}. Depending on the population size, the average evolved tree size ranges from 119 to 208 nodes. Speedup is illustrated in Figure 5.

These Figures show that, in all but one case, evaluation on the GPU yields a speedup in computing time, for small training cases sets and short expression lengths: the largest average tree size we encountered was 208 nodes. However the CPU is superior for the multiplexer-6 problem: the memory transfer and tree-to-postfix translation overheads cannot be counter-balanced by the speedup in

[5] *Iflte* is a quaternary operator that stands for "If *sibling1* less than *sibling2* then *sibling3* else *sibling4*.

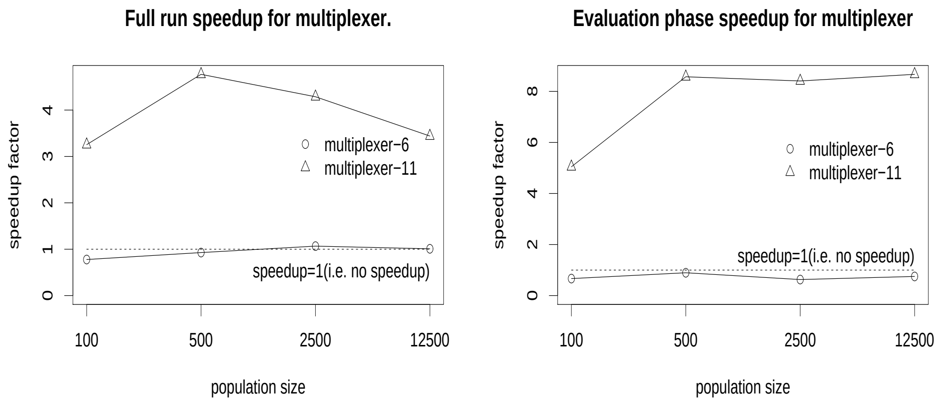

Fig. 4. GPU vs CPU speedup on multiplexer-6 and multiplexer-11 (64 and 2048 training cases respectively). On the left, speedup for whole evolutionary runs, on the right speedup for evaluation phase only.

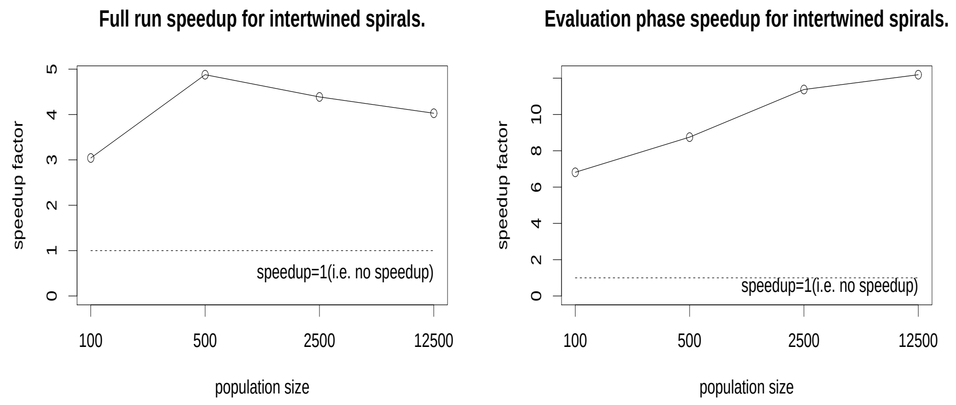

Fig. 5. GPU vs CPU speedup on intertwined spirals (194 training cases). On the left, speedup for whole evolutionary runs, on the right speedup for evaluation phase only.

parallel computation. Note that typical solutions to this problem contain many *If* operators and may be suspected to create a high divergence between the 8 inner stream processors that deal with one GP program. This means that many branches of the GP tree must be interpreted with part of the stream processors in idle mode to respect the SIMD constraint, with a drop of performance.

For full runs, the speedup increases with the population size until we reach a threshold where it begins to stagnate or drop. Of course the speedup cannot increase indefinitely and must anyway reach an upper bound when the GPU is saturated. But in our case, this phenomenon occurs earlier due to one basic implementation choice: the breeding phase is done by the ECJ library, so it is computed on the CPU and its cost increases faster than the evaluation cost, as can be seen on Figure 6. We recall that the evaluation time includes the cost of compacting the programs in linear form and translating them to postfixed notation. This phase is also performed on the CPU in our current implementation, and thus does not benefit from parallelization. This is responsible for the slight drop of evaluation speedup in the right plot of Figure 3 with population size 12500.

At last, speedup factors obviously depend on the problem, especially if it needs operators such as "If" that create unavoidable divergence between stream processors, wasting computation cycles. This explains the difference in performance between the regression benchmark and the two others.

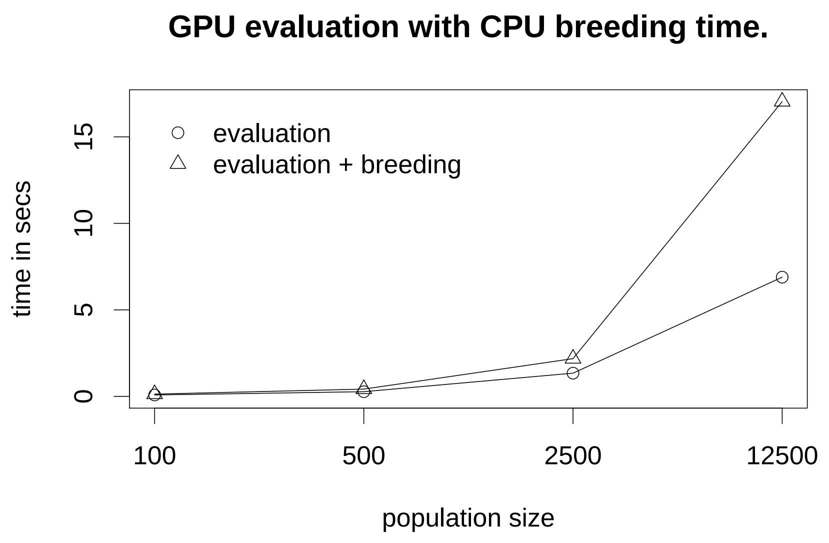

Fig. 6. Mean evaluation and breeding time for the GPU runs on the $x^6 - 2x^4 + x^2$ regression problem, with 1024 cases. As breeding is kept on the CPU, it becomes the bottleneck when processing large populations.

5 Conclusions and future works

Previous works about parallelizing GP on GPU brought speedups with respect to large programs or/and large training sets. However it is not always possible to gather large training sets, e.g. when labeling training cases requires human intervention like medical diagnosis. It is neither always easy to evolve large GP individuals without incurring a high level of over-fitting. Thus it is also interesting to obtain speedups for small GP trees and small training cases.

We worked on one of the current fastest GPU, the Nvidia G80. Our solution consists in parallelizing both GP programs and training data, as opposed to run sequentially each compiled program and parallelizing only the training set. When several programs are run in parallel, they process proportionally more training cases on each elementary processor of the GPU, so we can expect a better filling of the ALUs and an overall increased efficiency. As running different GP programs on a basically SIMD architecture is not possible, we use an interpreter to process both programs and training cases as data. A typical problem raises when the main interpreter switch is required to execute different instructions in parallel: the GPU executes these instructions sequentially, putting alternatively some of the elementary processors in idle mode. This is called divergence. As our scheme relies on a fine grain parallelization allowed by the CUDA language, we have the opportunity to exploit the SPMD architecture of the G80. We dispatch one program per multiprocessor, thus divergence appears only in the case when the GP function set contains "if" or "loop" nodes.

With this parallelization scheme we obtained evaluation phase speedups ranging from 8 times to 80 times for 5 out of 6 benchmarks, using from 64 to 1024 training cases and mean evolved tree sizes from 30 to 208 nodes. However no speedup was obtained on the multiplexer-6 benchmark, which cumulates a small training set together with a high tendency to create divergence through the major part of its function set (if, shortcut and, shortcut or).

By implementing the GPU evaluation as part of the Java ECJ library, we also allow other users of G80 cards to easily develop their own GP applications. However, keeping ECJ flexibility has a drawback: the breeding phase is performed on the CPU and does not benefit from the GPU power. Experiences showed that when population size increases, the breeding time grows until it is no more negligible against the evaluation time.

Future works includes extending ECJ to store the GP population into an array and implementing an interpreter for prefix code, removing the need for compacting and translating GP trees to postfix. Nonetheless the cost of the breeding phase suggests that it is also required to implement it on the GPU, in order to take full advantage of the new graphics cards power to evolve large populations.

References

1. M. L. Wong, T. T. Wong, and K. L. Fok. Parallel evolutionary algorithms on graphics processing unit. In *Proceedings of IEEE Congress on Evolutionary Com-*

putation 2005 (CEC 2005), volume 3, pages 2286–2293, Edinburgh, UK, 9April 2005. IEEE.

2. Ka-Ling Fok, Tien-Tsin Wong, and Man-Leung Wong. Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems*, pages 69–78, 2007.
3. Qizhi Yu, Chongcheng Chen, and Zhigeng Pan. Parallel genetic algorithms on programmable graphics hardware. In *Advances in Natural Computation*, volume 3162 of *LNCS*, pages 1051–1059. Springer, 2005.
4. Karsten Kaul and Christian-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.
5. N. Nedjah, E. Alba, and L. de Macedo Mourelle. *Parallel Evolutionary Computations*. Springer-Verlag, 2006.
6. Simon Harding and Wolfgang 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.
7. Simon Harding and Wolfgang 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.
8. Darren 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.
9. W. B. Langdon. A SIMD interpreter for genetic programming on GPU graphics cards. Technical Report CSM-470, Department of Computer Science, University of Essex, Colchester, UK, 3 July 2007.
10. John Koza, Martin Keane, Matthew Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.
11. P. Sanders. Emulating mimd behavior on simd machines. In *Proceedings of International Conference on Massively Parallel Processing Applications and Development*, Delft, 1994. Elsevier.
12. Hugues Juille and Jordan B. Pollack. Massively parallel genetic programming. In *Advances in Genetic Programming 2*, chapter 17, pages 339–358. MIT Press, 1996.
13. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques and Tools*. Addison-Wesley, 1986.
14. John Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.
15. Kevin J. Lang and Michael J. Witbrock. Learning to tell two spirals apart. In Morgan-Kaufmann, editor, *Proceedings of the 1988 Connectionist Summer Schools*, 1988.