Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

GPU-BASED ACCELERATION OF GENETIC ALGORITHMS

Mikuláš Dítě, mikulas.dite@gmail.com, Gymnzium Christiana Dopplera, Prague, Czech republic
expert advisor: Miroslav Čepek, Katedra počítaču ČVUT, Prague, Czech republic

ABSTRACT

Genetic algorithms offer answer to almost every problem of NP or even NP-hard complexity, which are merely solvable in nondeterministic polynomial time. This process is inspired in nature. As in real world, even in computer evolution exists a population of thousands of competing individuals. Those considered best move their offsprings to the next generation and the process repeats. It is important to note that with these algorithms we are never sure if the solution found is the best one - we assume it is good enough.

We measured how much these algorithms may be optimized by computing as much as possible in parallel. Even though CPUs offer threads and other technologies, modern GPUs with hundred times more cores are much better for parallelization. With this optimization, the total time required may be decreased to fractions of original time.

For reference values, we have simulated a Pole Balance [1] problem, in which a cart able to move in just one axis is taught to balance a pole connected to its base.

GENETIC PROGRAMMING

Many computing problems cannot be solved within a reasonable time span if all possible solutions have to be evaluated in order to find the best one. This is especially true about the NP hard complexity class (which also contains NP-complete class) [2]. Genetic algorithms (GA) can find a ”good enough” solution, but can never tell if the result found is the most optimal, the best.

INVERTED PENDULUM

The problem with genetic programming is the lack of universal benchmark problem [1]. However, most tests usually settle with Pole Balance Problem (often to be found as Inverted Pendulum). Inverted Pendulum is a pendulum which has its mass above its pivot point [3]. As the pendulum is very unstable, the cart is forced to a constant pending to left and right side. This construction is among the most difficult systems to control in the field of control engineering [4].

The pendulum axis is only flexible in one dimension, in which a track boundary (h) is created.

These limits are checked during the run time for each cart. If either of these two rules is broken, the cart failed to balance the pole and the test is forced to its end. A few rules are set for the implementation itself. These are hardcoded into the application and thus

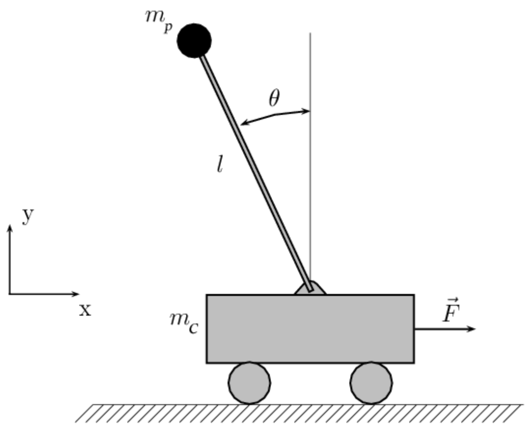

Fig. 1. Inverted Pendulum schematics: the cart can only move on the x axis

are not tested. The cart (respectively its engine) must always exert a non-zero constant force (F). The only decision the circuits are supposed to make is whether the force should be applied from left or right side of the cart and thus will move the cart in the direction desired.

Genetic algorithm for this exact problem involves evolving a set of constants (c_1 to c_4) for which a cart can balance the pole the fastest if set right. As the only variable a cart can affect is the force it exerts (or rather it's direction), let the force be computed as:

$$F(t) = F\left[\left(c_1 x + c_2 \dot{x} + c_3 \theta + c_4 \dot{\theta}\right) > 0\ ?\ 1 : -1\right] \quad (1)$$

where F is the absolute constant force which is multiplied by either one or minus one. The reason sign function was not used is that a cart must always move, which it would not if the sign returned zero. The sum in the round brackets actually takes every variable of a cart and its pendulum that is relevant and multiplies it with on of the constants we are evolving. The syntax used is a standard ternary operation.

GPU-ONLY IMPLEMENTATION

Original GPU implementation wrote to GPU memory and read from it every generation, so that could CPU prepare new generations and write them back. This led to excessive time usage: over 50% of total time consumption was simple I/O. The GPU-only version leaves data on graphic card memory and computes new generations itself. Though this basically eliminates data flow between processor and graphic card, there are two major issues.

First, OpenCL (the library we used to communicate with graphic card directly from an application) does not provide pseudorandom function. Standard rand() in C language is very fast, but custom made functions (also called kernels) on GPU are slower. We used Xorshif pseudorandom [5], which was tested to pass tests of randomness very well and thus provides great entropy of generated numbers. This is indeed a problem, because generating random numbers on GPU takes as much as 38% of the fitness computing itself. On the other hand, it is still about four times faster then the original GPU version.

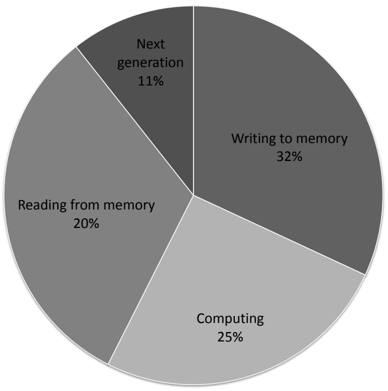

Fig. 2. Original GPU implementation is very time sensitive due to excessive IO

RESULTS

Generally, paralelization does save a great amount of time. The key point of this paper was to verify the theory that GPU is capable of computing more fitness functions in parallel. Because all implementations differs greatly, it is impossible to determine exact factor of time saved. For this exact test sample, porting the application to GPU made it significantly faster (at least four times at about 10000 entities per generation and even more for larger samples).

DISCUSSION

Fig. 6 was created by measuring time the algorithm took to iterate through a constant amount of generations. If the application would be stopped when a good enough solution was found, we would rarely reach the same amount of time and possibly for the huge statistical sample the final function would be either constant or slowly increasing linear function.

Every value was measured at least 200 times. Entities per generation started at 400 and ended up on 19600, with a delta of 400.

CPU implementation was expected to have a time complexity of ge (10). With constant g, we indeed see

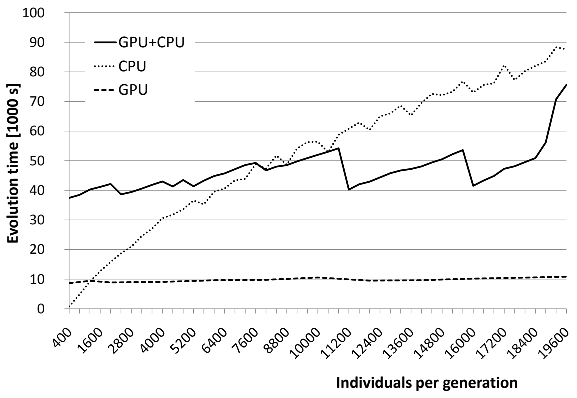

Fig. 3. Overview of times required to find best entities for different settings

a linear function on Fig. 6.

The results of slower GPU version (with IO between CPU and graphic card) are very interesting. The saw-like peeks are again caused by the limited memory GPU has, which is about 1/8 of the RAM for used hardware. It reallocates more often and also takes more time to do so.

Results GPU-only application provided are great. Eliminating the IO, the time was reduced to one fourth.

CONCLUSION

Genetic algorithms, while faster then other methods, are actually still too slow. Moving the algorithm to graphic card does provide good amount of time save, especially for many entities per generation. Best timesaving however comes when the whole computation is made on GPU: it is about four times faster then genetic algorithm communicating with CPU.

Source codes of these implementations are to be found at:

https://github.com/Mikulas/PoleBalance
https://github.com/Mikulas/PoleBalanceGPU

REFERENCES

1. BROWNLEE, J. *The pole balancing problem.* [online]. [quoted 8. 1. 2011]. Available from http://scr.bi/lXYmeR
2. Multiple authors. Wikipedia, the free encyclopedia *NP-hard.* [online]. [quoted 11. 6. 2011]. Available from http://en.wikipedia.org/wiki/NP-hard
3. Wikipedia, the free encyclopedia *Inverted pendulum.* [online]. [quoted 11. 6. 2011]. Available from http://en.wikipedia.org/wiki/Inverted_pendulum
4. SULTAN, Khalil. *Inverted Pendulum.* [online]. [quoted 15. 6. 2011]. Available from http://bit.ly/jbtaqg
5. MARSAGLIA, George *Xorshift RNGs.* [online]. [quoted 12. 6. 2011]. Available from http://bit.ly/jT5Wah