GPGPGPU: Evaluation of Parallelisation of Genetic Programming using GPGPU

Jinhan Kim, Junhwi Kim, Shin Yoo

Korea Advanced Institute of Science and Technology
Republic of Korea

Abstract. We evaluate different approaches towards parallelisation of Genetic Programming (GP) using General Purpose Computing on Graphics Processor Units (GPGPU). Unlike Genetic Algorithms, which uses a single or a fixed number of fitness functions, GP has to evaluate a diverse population of *programs*. Since GPGPU is based on the Single Instruction Multiple Data (SIMD) architecture, parallelisation of GP using GPGPU allows multiple approaches. We study three different parallelisation approaches: kernel per individual, kernel per generation, and kernel interpreter. The results of the empirical study using a widely studied symbolic regression benchmark show that no single approach is the best: the decision about parallelisation approach has to consider the trade-off between the compilation and the execution overhead of GPU kernels.

1 Introduction

Genetic Programming has been widely adopted by the Search Based Software Engineering community: its application ranges from fault localisation [7, 12, 14], Genetic Improvement [5, 9], and program repair [3, 8]. Improving its efficiency and scalability would have a far reaching impact across the application domains.

Parallelisation is one of the most promising technique for scalability. Population based evolutionary computation has been described as 'embarrassingly parallel', because the fitness evaluation of each individual solution in the population is often completely independent from each other and, consequently, can be performed in parallel. This is particularly the case with Genetic Algorithms (GAs): GAs need to apply the same fitness function(s) to the entire population, which essentially consists of *input* data to the fitness function(s).

General Purpose Computing on Graphics Processor Units (GPGPU) exploits the Single Instruction Multiple Data (SIMD) architecture of graphics shaders to parallelise computation [4]. The SIMD architecture fits the parallel fitness evaluation of GAs naturally, and has provided significant speed-ups for search-based test suite minimisation [13].

Genetic Programming (GP), on the other hand, keeps a population of *programs*. Parallelisation at the GP population level is not possible, as it would not fit the SIMD architecture. Instead, GP can be parallelised at the training data level. Usually, a single candidate GP solution has to be evaluated against many data points in the training set, which can be done in parallel.

However, this GPGPU based fitness evaluation for GP requires the conversion of GP trees into GPGPU executable kernels. The conversion involves the kernel compilation, which is a time consuming process that is external to the GP. The cost of kernel compilation raises the issue of cost-benefit trade-off for GPGPU.

This paper evaluates different methods of amortising the cost of kernel compilation using CUDA[1] toolkit. Kernel per individual method converts each individual GP tree into a separate CUDA kernel. Kernel per generation aggregates all individuals in the population and performs a single compilation of all individuals. Finally, kernel interpreter method uses an expression interpreter: after a single compilation of the interpreter, GP can evaluate whatever GP tree without any further compilation. We use the CPU based GP as the baseline, and the Dow chemical data symbolic regression as the benchmark problem [10].

2 Evaluating GP Trees Using GPGPU

To achieve data level parallelisation using GPGPU, the kernel should be generated dynamically. Here, we introduce three different approaches.

- **Kernel per Individual**: The most intuitive approach to convert GP trees into CUDA kernel is to generate a single CUDA kernel for each candidate solution. Since the only difference between candidates is the expression they represent, kernel source code can be generated using templates: we only need to convert GP trees into infix expressions that conform to CUDA kernel syntax. While intuitive, a drawback of this approach is that we have to invoke CUDA compilers for each individual. If the population size is large, this may cause a significant overhead.
- **Kernel per Generation**: To reduce the kernel compilation overhead, we can generate one kernel source code file per generation: the single file will contain multiple kernels, each corresponding to the individual solutions in the GP population. While this results in much longer kernel source code files (and hence increased compilation time), we expect to save the overhead of invoking CUDA compilers multiple times.
- **Kernel Interpreter**: One technique that has been studied in the GP literature [1,11] is to use a single kernel that can *interpret* GP candidate solutions. Using an interpreter, the GP population is transferred to the GPU as *data*, which are then interpreted and evaluated against the data points in the training data. The kernel interpreter method requires only one compilation throughout the entire GP run. While this significantly reduces the kernel compilation time, it increases the complexity of the CUDA kernel, which in turn affects the performance of GPGPU. We implemented an RPN(Reverse Polish Notation) based CUDA interpreter kernel for this study.

[1] Compute Unified Device Architecture from NVIDIA

3 Experimental Setup

3.1 Research Questions

Two major drivers of the computational load of GP fitness evaluation are the population size and the training dataset size. Both directly affects the number of fitness evaluations that have to be performed. We formulate our Research Questions around these two factors as follows:

- **RQ1.** Which approach performs best against different training dataset sizes?
- **RQ2.** Which approach performs best against different population sizes?

Our study uses the Dow Chemical symbolic regression benchmark [10] to investigate the performance of different parallelisation approaches: it contains a training dataset with 747 data points consisting of 57 independent variables and 1 dependent variable. We answer **RQ1** by artificially controlling the size of the training dataset and comparing the efficiency of different GPGPU approaches as well as the CPU baseline. Since the aim of our study is not to improve the accuracy of the evolved expression, we simply repeat each data point in the training dataset $10^0, 10^1, 10^2, 10^3$, and 10^4 times to generate datasets with different sizes. We answer **RQ2** by running GP with population size of 50, 100, and 200, and comparing the results. Every experiment is conducted 30 times.

3.2 Configurations & Environments

We use DEAP[2] to implement different approaches: we use the 57 independent variables in the Dow Chemical problem set as GP terminal nodes, and include addition, subtraction, multiplication, division, and negation. The GP uses a three-way tournament selection, a single point crossover with the rate of 0.6, and uniform mutation with the rate of 0.01. The tree depth is set to 4. We set the termination criterion as reaching 10 generations.

The experiments were conducted with an Intel i7-6700 CPU machine with 32 GB of RAM, running Ubuntu 14.04.5 LTS. The GPGPU has been performed on an NVIDIA TITAN X with 12 GB of GDDR5X using CUDA version 8.0. DEAP has been executed using Python 3.4.

4 Results

Figure 1 shows the execution time of GP using three parallelisation approaches as well as CPU, against five different training dataset sizes and the population size of 50. The lines connecting boxplots are merely visual aids for identifying the same approach: they connect the mean value of each boxplot. As the training data size increases, the efficiency of CPU based GP and kernel interpreter deteriorate sharply, whereas both kernel per individual and kernel per generation

[2] https://github.com/DEAP/deap

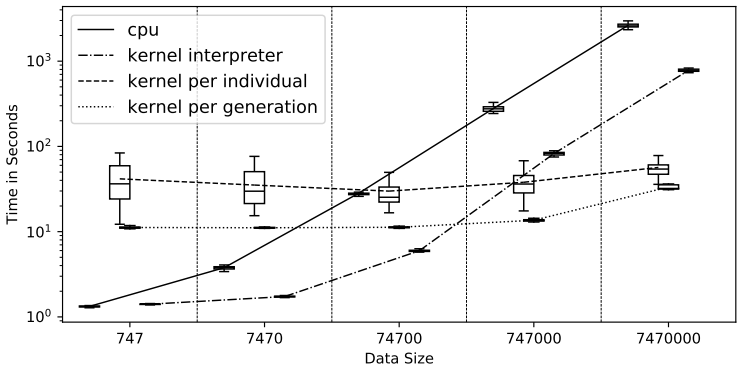


Figure 1: Plot of three parallelisation approaches and CPU based GP on the fixed population size 50. The y-axis is shown on logarithmic scale.

approach show relatively stable performance. However, up to the dataset size of 74,700, the interpreter approach shows the best performance.

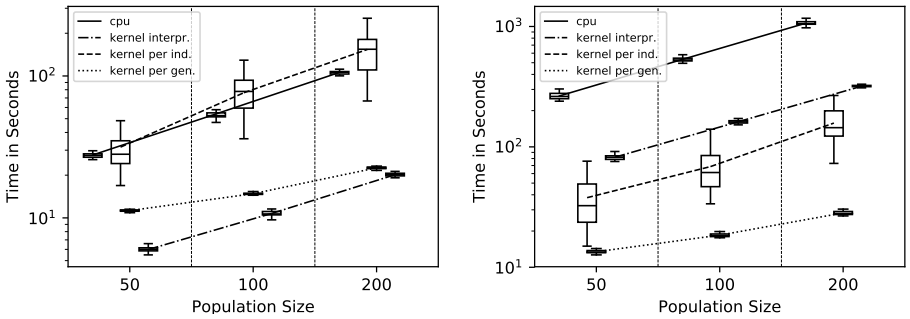
The performance deterioration of the interpreter approach is due to the overhead of RPN-based expression evaluation. When the size of the training dataset is relatively small, the savings in kernel compilation time compensate for this overhead. With larger dataset (i.e. more kernel execution), the interpreter overhead cancels out the savings in compilation time.

The interpreter overhead is mainly due to two factors. First, the interpreter uses CUDA registers to maintain a stack, increasing the I/O overhead compared to kernels that hardcode the expression (kernels per individual and kernel per generation). If the stack becomes too large to be contained within registers, we may have to rely on even slower memory, increasing the I/O overhead even further. Second, the interpreter makes more function calls internally, compared to the hardcoded kernels: the increased branching also deteriorates the performance of the kernel interpreter.

To answer **RQ1**: the best parallelisation approach is determined by the trade-off between compilation time and the computational overhead of the kernel interpreter. Above a certain number of kernel executions, the interpreter loses the savings from the fewer compilations.

To answer **RQ2**, we fixed the size of dataset and varied the population size. The results in Figure 2a show that, for the data size of 74,700, the interpreter method outperforms all other approaches, regardless of the population size (i.e. its performance overhead is still being cancelled out by the savings in the compilation time). Note that the kernel per individual approach performs *worse* than the CPU. However, in Figure 2b, kernel per generation performs the best. In fact, the relative order between approaches is the same as in Figure 1 with dataset size of 747,000, regardless of the population size.

We note that the kernel per individual approach shows wider variances as the population size grows. Since the approach relies heavily on an external process



(a) Fixed data size 74,700 **(b)** Fixed data size 747,000

Figure 2: Plot of three parallelisation approaches and CPU based GP on the fixed data size 74,700 and 747,000. Both y-axes are shown on logarithmic scale.

(i.e. CUDA compiler), we posit that it is more vulnerable to the external and environmental factors that can affect the execution time stochastically.

5 Related Works

The use of an interpreter has been suggested as a way to scale up GP on GPUs: Langdon and Banzhaf implemented an RPN-based interpreter for GP regression [1]. Wilson and Banzhaf implemented an entire Linear GP system on GPUs, parallelising not only the fitness evaluation but also the GP mutation [11]. Both approaches have been developed with earlier incarnations of GPGPU frameworks and do not benefit from the high level programming support of contemporary frameworks. Our work exploits the modern GPU development framework to compare approaches such as kernel per individual.

Other applications of GP in SBSE involves evolving not just expressions but arbitrary code [2, 3, 9]. To parallelise GP for these applications, we need to be able to execute arbitrary code on GPU. The existence of I/O operations or system calls prevents such use of GPGPU. However, there are ongoing works that attempt to overcome the limitations of GPU environment. For example, Silberstein et al. have tried to interface the host file system with GPU kernels [6].

6 Conclusion

This paper evaluates three different parallelisation approaches for GP fitness evaluation on GPU: kernel per individual, kernel per generation, and kernel interpreter. The empirical study using a symbolic regression benchmark problem shows that, while the kernel per generation performs best overall, the actual performance depends on multiple factors such as the size of the population and the volume of the training data. Consequently, we advise GP practitioners to choose their parallelisation approach carefully. Future work will investigate a wider range of benchmark problems.

References

1. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcazar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008. Lecture Notes in Computer Science, vol. 4971, pp. 73–85. Springer (March 2008)
2. Langdon, W.B., Harman, M.: Genetically improving 50,000 lines of C++. Tech. Rep. RN/12/09, Department of Computer Science, University College London (2012)
3. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International Conference on Software Engineering. pp. 3–13 (2012)
4. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1), 80–113 (2007)
5. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: Nicolau, M., Krawiec, K., Heywood, M.I., Castelli, M., Garcia-Sanchez, P., Merelo, J.J., Rivas Santos, V.M., Sim, K. (eds.) 17th European Conference on Genetic Programming. LNCS, vol. 8599, pp. 137–149. Springer (2014)
6. Silberstein, M., Ford, B., Keidar, I., Witchel, E.: GPUfs: Integrating a file system with GPUs. SIGARCH Comput. Archit. News 41(1), 485–498 (March 2013)
7. Sohn, J., Yoo, S.: FLUCCS: Using code and change metrics to improve fault localisation. In: Proceedings of the International Symposium on Software Testing and Analysis, *to appear*. ISSTA 2017 (2017)
8. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE '09). pp. 364–374. IEEE (16-24 May 2009)
9. White, D., Arcuri, A., Clark, J.: Evolutionary improvement of programs. IEEE Transactions on Evolutionary Computation 15(4), 515 –538 (August 2011)
10. White, D.R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaśkowski, W., O'Reilly, U.M., Luke, S.: Better GP benchmarks: community survey results and proposals. Genetic Programming and Evolvable Machines 14(1), 3–29 (2013)
11. Wilson, G., Banzhaf, W.: Deployment of CPU and GPU-based genetic programming on heterogeneous devices. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference (GECCO 2009). pp. 2531–2538. ACM Press, New York, NY, USA (July 2009)
12. Yoo, S.: Evolving human competitive spectra-based fault localisation techniques. In: Fraser, G., Teixeira de Souza, J. (eds.) Search Based Software Engineering, Lecture Notes in Computer Science, vol. 7515, pp. 244–258. Springer (2012)
13. Yoo, S., Harman, M., Ur, S.: GPGPU test suite minimisation: Search based software engineering performance improvement using graphics cards. Empirical Software Engineering 18(3), 550–593 (2013)
14. Yoo, S., Xie, X., Kuo, F.C., Chen, T.Y., Harman, M.: Human competitiveness of genetic programming in sbfl: Theoretical and empirical analysis. ACM Transactions on Software Engineering and Methodology, to appear (2017)