Accelerating Genetic Programming using PyCuda

Keiko Ono
Ryukoku University
Kyoto, Kyoto
kono@rins.ryukoku.ac.jp

Yoshiko Hanada
Kansai University
Osaka, Osaka
hanada@kansai-u.ac.jp

ABSTRACT

Nvidia's CUDA parallel computation is a good way to reduce computational cost when applying a filter expressed by an equation to an image. In fact, programs need to be compiled to build GPU kernels. Over the past decade, various implementation methods for the image filter using Genetic Programming (GP) have been developed to enhance its performance. By using GP, an appropriate image filter structure can be obtained through learning algorithms based on test data. In this case, each solution must be compiled; therefore, the required computational effort grows significantly. In this paper, we propose a PyCuda-based GP framework to reduce the computational efforts for evaluations. We verify that the proposed method can implement GPU kernels easily based on a sequential GP algorithm, thereby reducing the computational cost significantly.

CCS CONCEPTS

• **Computing methodologies** → *Discrete space search*;

KEYWORDS

Genetic Programming, GPU, PyCuda

ACM Reference Format:
Keiko Ono and Yoshiko Hanada. 2018. Accelerating Genetic Programming using PyCuda. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA , Article 4, 2 pages. https://doi.org/10.1145/3205651.3208760

1 INTRODUCTION

GPU has a massively parallel architecture consisting of thousands of CUDA cores, and is widely used in many fields, especially image processing. Recently, various image processing methods utilizing GP have been proposed to enhance performance. In GP, each individual is expressed by a tree or an equation, so it matches especially well with developing an image filter. However, GP evaluates many individuals in order to evolve in each generation. In addition, it requires large computational efforts when applied to developing an image filter, because it needs to evaluate every individual $I \times P$ times if an image has P pixels, where I is the number of individuals in a population. The GP methods with CUDA-C have been proposed over the past decade to overcome this problem. CUDA-C can compile and run ordinary C. Therefore, we must generate I C-code files or generate codes using a stack structure for individuals every generation to evaluate them[2]. Therefore, the number of file I/O increases, and the GP methods with CUDA-C cannot work with enough efficiency in many cases.

In this paper, we propose a PyCUDA-based framework to overcome this problem. PyCUDA is a module to access Nvidia's CUDA parallel computation and allows us to execute GPU kernels written in Python by automatically transferring their Python codes to CUDA-C[1]. PyCUDA can generate GPU kernels without file I/O by using *source module*.

2 PROPOSED METHOD

CUDA C is often used when applying an image filter to target images, and it helps reduce the computational time dramatically. In general, the image filter is predefined; however, the structure of an image filer is different for every individual when applying GP. We propose an accelerating genetic programming framework using PyCUDA for which file I/O is not necessary for compilation. Figure 1 shows the proposed framework.

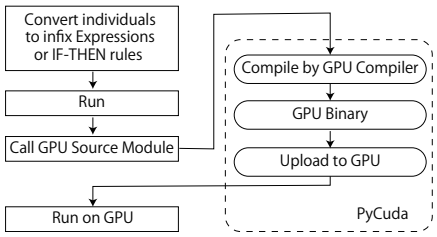

Figure 1: PyCuda GPU program compilation for GP.

In GP, each individual is expressed by a tree, so the proposed method first converts the tree to an infix-expression or an IF-THEN rule to compile CUDA-C. Then, a source module is called, where codes for filtering generally are declared directly in the source module in PyCUDA as follows:

```
mod = SourceModule("""
__global__ void function_name(arguments)
{
  .....
}
""")
```

However, this ordinary method in PyCUDA cannot change an equation or an IF-THEN rule for image filtering. Therefore, the proposed method passes the codes as arguments to enable their modification. The source model is called as "mod = source module(argument)", where the argument is codes written in CUDA-C.

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5764-7/18/07.
https://doi.org/10.1145/3205651.3208760

By calling the source module, the argument is compiled to generate GPU kernels.

3 PERFORMANCE EVALUATION

We first evaluated the effects of block or thread sizes on the GPU compared to the CPU. Table 1 shows the results, where the number of evaluations indicates the number of pixels and a filter is set to $\sin(x)$ to simplify [1]. We found that the calculation time rapidly increases as the number of evaluations increased when using the CPU, while the GPU was effective in increasing the number of evaluations. Specifically, the calculation time was less than 2 s. We also found that the calculation time of the GPU increased as the block size increased, but were not obvious differences between them. Therefore, we set the block size to 1 and the thread size to 1024 for the next experiments where we compared the effectiveness of the proposed method, as well as GP against the CPU.

Table 1: Calculation time.

#Evalutaions	#Blocks	#Threads	Time(GPU)[sec.]	Time(CPU)[sec.]
2^5	1	128	0.000917	0.000056
2^5	1	1024	0.000907	
2^5	128	128	0.000859	
2^5	128	1024	0.000986	
2^{13}	1	128	0.003017	0.317426
2^{13}	1	1024	0.003485	
2^{13}	128	128	0.003562	
2^{13}	128	1024	0.020838	
2^{19}	1	128	0.144891	1392.084
2^{19}	1	1024	0.160954	
2^{19}	128	128	0.178302	
2^{19}	128	1024	1.146410	

Next, we evaluated the effectiveness of the proposed method by using a symbolic regression problem. We investigated the symbolic regression problem for the function space $\mathcal{X}$ constructed by the labeled ordered trees of functional nodes {+, −, ×, /, sin, cos, log} and terminal nodes {s, 0.0, 1.00}, where s denotes a variable. Our training set was composed of J data points $\{(s_j, x_*(s_j)) \in \mathbf{R}^2 ; j = 1, \ldots, J\}$, where $s_j = 0.2/J(j-1) - 1$, and $x_*(s) \in \mathcal{X}$ is the true function to be identified, and J is set to $2\times 10, \ldots, 2\times 10^6$. For any $x(s) \in \mathcal{X}$, we define the fitness $f(x)$ by:

$$f(x) = \sum_{j=1}^{J} |x(s_j) - x_*(s_j)|,$$

and consider the maximization problem of $f(x)$. In our experiments, we employed three functions as $x_*(s)$,

$$\text{Function}: \quad x_*(s) \quad = s^4 - s^3 - s^2 - s.$$

We set the following parameters for GP: the number of generations was 50, the population number was 300, the crossover rate was 0.5, the mutation rate was 0.1, the maximum depth was 17, and the initial solutions were generated by Ramp-Half-and-Half.

[1] All of our experimentation was undertaken on a single PC with 6 Intel Xeon E5-1660 3.3 GHz processors, with 24 GB of memory and an NVIDIA Quadro K6000 running under Linux. NVIDIA Quadro K6000 is capable of around 5.2 TFlops and 2880 CUDA cores with 15 multiprocessors and 192 microprocessors.

Figure 2 shows the result of the average of calculation time for each individual. GPU took 1 to 2 s and there was no change in the calculation time as the number of loops increased [2]. Conversely, for the CPU, when the number of loops was greater than 2^5, it took a long time to evaluate an individual. The total time to evaluate individuals for 50 generations was 2362 s for the GPU and 215578 s for the CPU when $J = 2\times 10^6$. Figure 3 shows the history of the fitness of the best individuals in a population. From this figure, there are no differences in fitness in terms of GPU and CPU, and we found that the proposed method works well. From these results, we verified that the proposed method can reduce computational efforts significantly.

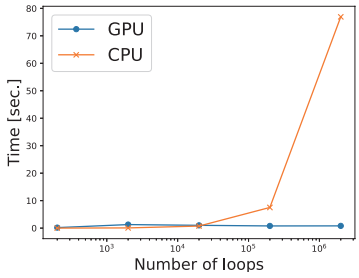

Figure 2: PyCuda GPU program compilation for GP.

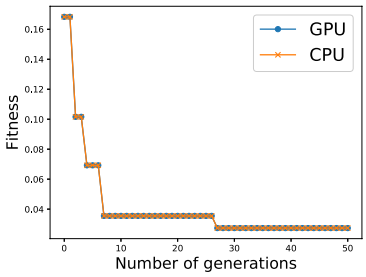

Figure 3: History of the fitness of the best solution.

4 CONCLUSIONS

We proposed a PyCUDA framework for GP that eliminates unnecessary file I/O by using a source module with parameter passing. By using a symbolic regression problem, we verified that the proposed method can reduce the computational time to evaluate individuals. When the number of pixels or the interval of the symbolic regression problem is greater than 10^6, the proposed method outperforms sequential models that rely on a CPU.

REFERENCES

[1] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.* 38, 3 (2012), 157–174.
[2] William B Langdon. 2010. A many threaded CUDA interpreter for genetic programming. (2010), 146–158 pages.

[2] we found that it took 1.3 s to load a source module based on the preliminary experiments.