Flash: A GP-GPU Ensemble Learning System for Handling Large Datasets

Ignacio Arnaldo, Kalyan Veeramachaneni, and Una-May O'Reilly

MIT, Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA
iarnaldo@mit.edu, {kalyan,unamay}@csail.mit.edu

Abstract. The Flash system runs ensemble-based Genetic Programming (GP) symbolic regression on a shared memory desktop. To significantly reduce the high time cost of the extensive model predictions required by symbolic regression, its fitness evaluations are tasked to the desktop's GPU. Successive GP "instances" are run on different data subsets and randomly chosen objective functions. Best models are collected after a fixed number of generations and then fused with an adaptive, output-space method. New instance launches are halted once learning is complete. We demonstrate that Flash's ensemble strategy not only makes GP more robust, but it also provides an informed online means of halting the learning process. Flash enables GP to learn from a dataset composed of 370K exemplars and 90 features, evolving a population of 1000 individuals over 100 generations in as few as 50 seconds.

Keywords: Genetic Programming, GPGPU computing, Ensembles.

1 Introduction

The fitness evaluation component of Genetic Programming (GP) symbolic regression (GPSR) dominates its cost because, at every generation, each model has to be evaluated on multiple fitness cases, i.e. the dataset. If the dataset does not saturate the memory of a single node, it can be replicated across nodes and the algorithm can be parallelized by splitting the population across them as *islands with migration*. On each node/island, "local" fitness evaluation is parallelized via multi-threading. If the dataset size exceeds single node memory capacity, *data-level* parallel approaches evolve models locally on each node with a subset of the data, then a *meta* model is built that fuses outputs from several of these models. Such ensemble strategies provide robustness to the learning process [8]. The *population-parallel* and *data-parallel* approaches are ideal if one has access to a cloud or cluster [18]. However some scenarios may prevent their adoption:

- Privacy and security policies around data may require that the machine learning and data mining be carried out locally.
- Large data transfers may be prohibitively expensive or require too extensive a prior setup.
- A scientist may wish to use GPSR as an exploratory tool, tightly integrated into a desktop workflow that requires timely delivery of models.

M. Nicolau et al. (Eds.): EuroGP 2014, LNCS 8599, pp. 13–24, 2014.
© Springer-Verlag Berlin Heidelberg 2014

Flash serves this class of scenarios wherein desktop computing is necessary or desired but the dataset is larger than the desktop's memory capacity. It is a serial, desktop alternative to a cloud-based GPSR ensemble system. Rather than node-based learning in parallel on different subsets of data and different parameter sets, the Flash GP learner is invoked sequentially; each time with a different subset of data and a different set of parameters. Because it exploits General Purpose GPUs to reduce the time cost of model evaluation, it is able to replicate GPSR ensemble functionality within a single desktop and still obtain reduced learning times, despite datasets sizes that are larger than the desktop's memory capacity. Flash comprises:

- **GPU-optimized GPSR:** Flash exploits different fitness functions that are well suited to run on the GPU. In particular, it uses the correlation between target values and predictions to drive GPSR. Diverse models are combined with the algorithm Adaptive Regression by Mixing [20].
- **Incremental Learning:** Flash, as opposed to cloud-based GPSR ensembles in which all the GP instances are run in a single batch of a preset size, decides after each run whether to run an additional instance or to stop learning.
- **High speed GPSR with large datasets:** Flash implements fitness evaluation in CUDA for execution in a low-cost gaming card, namely a NVIDIA Geforce GTX 690. The card has two GPUs and allows concurrent kernel executions. Our implementation enables GP to learn from a dataset composed of 370K exemplars and 90 features, evolving a population of 1000 individuals during 100 generations in as less as 50 seconds.

We proceed as follows: Section 2 reviews existing GPU implementations of GP algorithms. Section 3 presents the GPU specialized fitness functions. Section 4 describes Flash's ensemble approach. Section 5 describes the experimental setup while Section 6 presents the results. Section 7 concludes.

2 Related Work: Accelerating GP with GPUs

The capability of GPUs to speed up Genetic Programming has not escaped the attention of researchers. Table 1 summarizes the experimental conditions of significant contributions. In a nutshell, there are two recurrent approaches: compiling the population at each generation and executing the resulting program or creating an optimized interpreter of GP individuals.

Population Compilation This approach has its roots in GP scenarios targeting the automatic generation of executable programs where the compilation step verifies the feasibility of candidate solutions. However it can be applied to GP problems in general for execution speed. Compiled expressions are expected to execute faster since compiled code is optimized (code reordering, removal of useless or redundant operations). This advantage must be balanced with the overhead of compilation however.

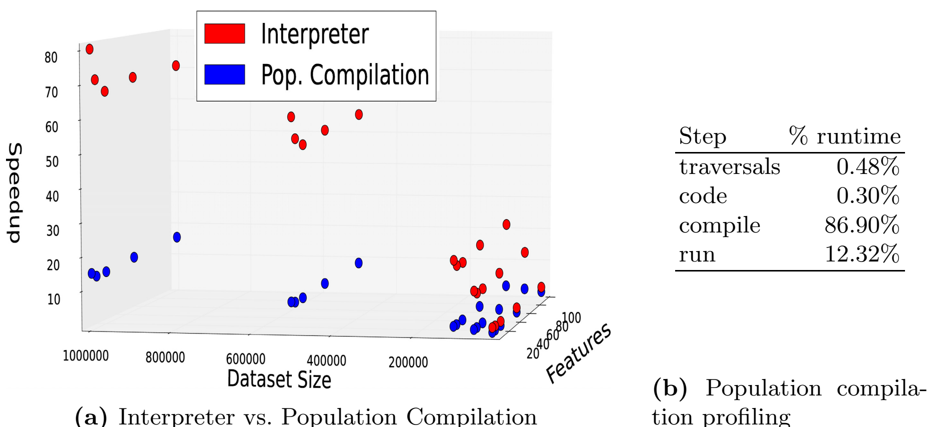
Interpretation An interpreter that is capable of evaluating the GP "language" of a given GP problem is implemented in the GPU. This approach waives compilation and its overhead. It exploits evaluation parallelism at the individual

Table 1. Experiment conditions of closely related work and reported speedup

Contribution	Pop. size	Test Cases	Approach	Speedup	As compared to
Harding-2007 [5]	100	65536	Pop. compilation	7351.06	Tree traversal
Chitty-2007 [3]		40000	Pop. compilation	29.98	CPU equivalent
Langdon-2008[9]	204800	1200	Interpreter	12.00	CPU equivalent
Banzhaf-2008 [1]		494021	Pop. compilation	7.40	CPU equivalent
Robilliard-2008 [16]	12500	2048	Interpreter	80.00	CPU equivalent
Wilson-2008 [19]	4000	251	Interpreter	2.51	Xbox CPU
Harding-2009 [7]	2048	10023300	Pop. compilation	55.00	Single GPU
Langdon-2009 [10]	262144	8192	Interpreter	34.00	CPU equivalent
Robilliard-2009 [17]	12500	100000	Interpreter	111.40	CPU equivalent
Lewis-2009 [12]	400	512	Interpreter	1259.57	CPU equivalent
Maitre-2010 [14]		1200	Interpreter	50.00	CPU equivalent
Langdon-2010 [11]	262144	8192	Interpreter	-	Never done in CPU
Maitre-2010b [13]	65536	65536	Interpreter	140.00	CPU equivalent
Harding-2011[6]		2000000	Interpreter	-	Never done in CPU

and fitness case level and suits GP scenarios with a small number of test cases. Because threads in charge of evaluating short expressions will be idle waiting for longer expressions to finish their execution, some resources are wasted.

Preliminary Analysis We compare the speedup obtained with the compilation and interpretation approaches against a standard tree traversal evaluation. We generate 5 random datasets composed of and 5,10,20,50, and 100 explanatory variables respectively and a million exemplars. For each of the five resulting datasets, we generate a population of 1000 individuals with a ramped half-and-half strategy. The time per population evaluation of these two approaches is shown in Figure 1. The comparative analysis shows that the interpreter approach obtains faster evaluations. As shown in Table 1b, the compilation time severely impacts the speedup obtained with the compilation approach. Therefore, we adopt the interpreter approach in our work.



(a) Interpreter vs. Population Compilation

Step	% runtime
traversals	0.48%
code	0.30%
compile	86.90%
run	12.32%

(b) Population compilation profiling

Fig. 1. Speedup obtained with the CUDA interpreter (80×) and compiling (14×) approaches with different data dimensionality and size as compared to a standard CPU-based evaluation, and profiling of the compilation approach

3 The Core GP Learner

We now present the core GP learner that uses the GPU. Most of the steps in the GP learning algorithm, like other approaches are carried out in the CPU. Fitness function evaluation, which is biggest bottleneck and scales with the data size is carried out on a GPU. To support ensemble learning we developed a library of interpreter approach based fitness evaluation functions. We then allow the flexibility to a choose a fitness function by the ensemble learning system. We illustrate how we implement two fitness functions on GPU: Mean Squared Error and Pearson Correlation coefficient between model output and target values computed as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2 \qquad r = \frac{\sum_{i=1}^{n}(\hat{Y}_i - \overline{\hat{Y}})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(\hat{Y}_i - \overline{\hat{Y}})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \overline{Y})^2}}$$

where Y is the target vector and $\hat{Y}$ is the model output or predictions. These two functions are embedded in a multi-objective GP based on NSGA-II targeting both accuracy and Subtree Complexity. The fixed structure of NSGA-II allows to evaluate the population in fixed-sized batches, which is useful for GPU execution.

3.1 Mean Squared Error and Pearson Correlation on GPUs

The computation required to score a model is broken in several steps. Note that, in Genetic Programming-based Symbolic Regression, it is typical to set the target and prediction values in the same range (see Step 0 and Step 3) prior to the computation of the error. This approach is meant to focus the search process to capture the shape of the curve rather than its scale.

Step 0: Normalization of the target values In this step, performed before the start of the GP algorithm, the target vector Y is normalized according to the minimum (Y_{min}) and maximum (Y_{max}) values.

Step 1: Evaluation of non linear model with GPUs In this step, nonlinear model $f(\bar{X})$ is evaluated for the n data points in D_T resulting in $y_{1...n}$. We implement a GPU interpreter capable of evaluating any possible arithmetic expression in postfix notation [9] [16] [19]. As depicted in Figure 2, several steps are required to obtain the postfix expression from a GP tree. We first traverse the tree in a depth-first in-order manner to generate the respective infix expression. We then apply Dijkstra's Shunting Yard algorithm [4] to obtain the postfix notation. The interpreter will evaluate the expression and produce an output value for each data point in the dataset. This task is computed via GPUs for datasets of size hundreds of thousands or even millions of independent test cases. In our GPU implementation, a CUDA thread is declared for each data point in the dataset. Thus multiple CUDA threads will execute the interpreter function shown in Figure 2 simultaneously on different data points, ensuring that all threads follow the exact same execution path. Note that conditional instructions such as *if* or *while* statements are pernicious for the performance of CUDA

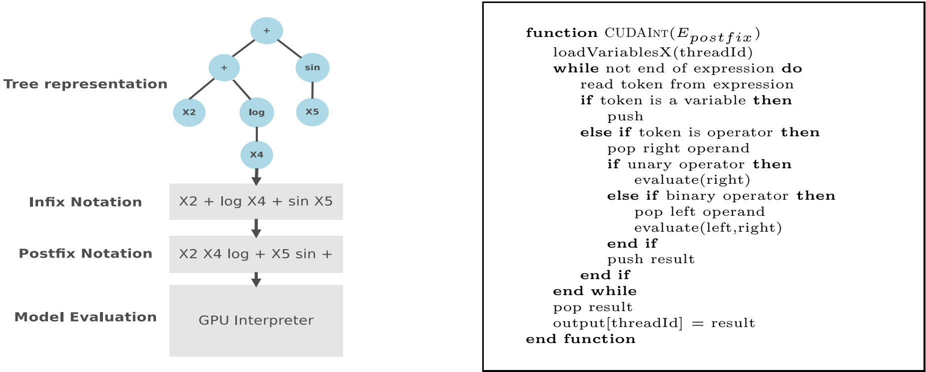


Fig. 2. GP Tree and corresponding infix, and postfix expressions (left) and pseudocode of the postfix interpreter (right)

programs only when they trigger a divergence in the execution of threads within a *warp* (see [15]). In such case, their execution is serialized. To benefit from coalesced memory accesses, we transpose the input matrix before storing it in global memory in such way that exemplars are displayed in columns while each line corresponds to an explanatory variable. This way, contiguous threads will access adjacent memory positions, thus reducing the number of expensive global memory accesses. At the end of this step we obtain the output of the model $\hat{Y}_{1...n}$ for all the n data points in D_T.

Step 2: retrieval of $\hat{Y}_{min}$ and $\hat{Y}_{max}$ A CUDA parallel reduction is employed to retrieve the maximum and minimum values of the model's output.

Step 3: normalization of the predictions $\hat{Y}$ We normalize the model's output according to minimum ($\hat{Y}_{min}$) and maximum ($\hat{Y}_{max}$) predictions.

The remaining steps are different for the computation of the Mean Squared Error (MSE) and the Pearson correlation coefficient (CORR):

MSE-Step 4: With both the target and prediction values in the same range, we compute the sum of the squared differences $\sum_1^n (\hat{Y} - Y)^2$ with a CUDA parallel reduction sum.
MSE-Step 5: The result of the sum is averaged over the number of exemplars of the dataset in CPU and assigned as fitness of the individual.

CORR-Step 4: We employ a CUDA parallel reduction sum to compute the mean of targets $\overline{Y}$ as well as the mean of the predictions $\overline{\hat{Y}}$

CORR-Step 5: Once again a CUDA parallel reduction sum is employed to compute the denominator $\sum_{i=1}^n (\hat{Y}_i - \overline{\hat{Y}})(Y_i - \overline{Y})$ and the numerator terms $\sqrt{\sum_{i=1}^n (\hat{Y}_i - \overline{\hat{Y}})^2}$ and $\sqrt{\sum_{i=1}^n (Y_i - \overline{Y})^2}$

CORR-Step 6: The Pearson correlation coefficient is computed in CPU and assigned as fitness of the individual.

3.2 Individual Level Parallelism

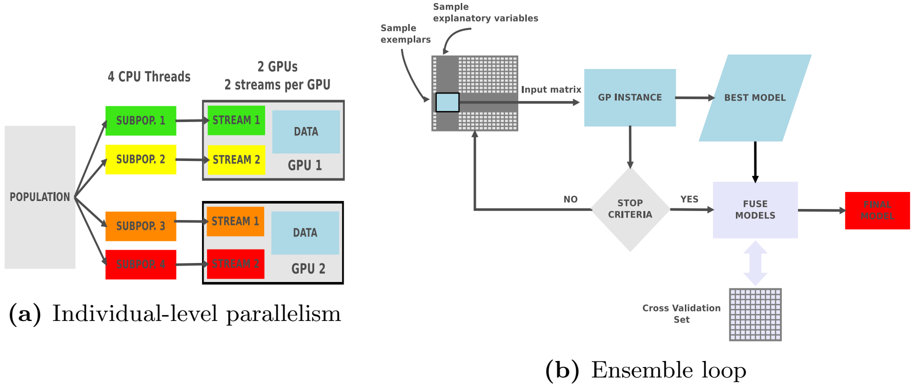
Modern CUDA compatible GPUs provide concurrent kernel execution, i.e. it is possible to execute several GPU functions at the same time. This allows us to parallelize GP at the individual level in a clean and easy way. Moreover, GPUs composed of two independent GPUs such as the employed NVIDIA GeForce GTX 690 are more and more frequent, granting further degrees of parallelism. As depicted in Figure 3a, the population is split into 4 different subsets. A CPU thread is declared for each subpopulation that calls the GPU evaluation function sequentially for each individual of the subset. The two first threads will employ the first GPU while the third and fourth threads employ the second GPU. The memory space of each of the two GPUs is independent, therefore data is copied to the Global Memory of both GPUs.

4 Flash - The GP-GPU Ensemble Learning System

Having designed a flexible core GP-GPU learner, we adopt an ensemble strategy in which several GP instances are run sequentially with different data subsets and parameters (such as fitness functions). In a step prior to the machine learning process, the targeted data D is split into training set D_T, cross-validation set D_{CV}, and test set D_{TEST}. GP instances learn from D_T while D_{CV} is employed to train the fused model. Finally, D_{TEST} is reserved to test the accuracy of the retrieved models. Figure 3b presents the Flash-GP approach: a subset of data is randomly selected and the objective function for the GP algorithm is picked randomly. The core GP learner is then executed with these parameters. After a fixed number of generations the best model is selected. The fusion module generates the fused model by training the weights within the "Adaptive Regression by Mixing" methodology using D_{CV}. A decision is made whether to continue the learning or not and the loop is repeated.

4.1 GP Instances

Factorization. As depicted in Figure 3b, each GP instance learns from a subsample of the exemplars and explanatory variables of D_T.
Exemplars: Each GP instance samples from the test cases of the training data D_T which will speed up model fitness evaluation and result in diverse model results across the sequential GP instances.
Explanatory Variables: Sampling from different explanatory variables reduces the dimensionality of the targeted dataset. On the other hand, the evolved models might exhibit low accuracy if the sampled variables can't sufficiently relate to the target values Y.



(a) Individual-level parallelism

(b) Ensemble loop

Fig. 3. The evaluation is parallelized at the individual level by exploiting concurrent kernel executions in two GPUs. Flash-GP: GP instances learn from different samples of the data and the retrieved models are fused in a later step.

Best Model per GP Instance. GP instances learn from the sampled data and are executed with a time or computational budget. At each generation, we store the model exhibiting the highest fitness value (MSE or correlation) with respect to D_T. The motivation to save the best model per generation is that models from advanced generations might overfit the data while some of the models obtained earlier might exhibit better generalization capability, i.e. a better accuracy with respect to unseen data. Once the GP run is finished, the stored models (best per generation) are evaluated against D_{CV} to obtain their MSE. The model exhibiting the lowest error with respect to the validation set is then selected as the best model of the run and will be used in the fusion process.

4.2 Generating a Fused Model

We employ the algorithm Adaptive Regression by Mixing (ARM) [20] that allows to fuse a set of models M according to an estimation of their accuracy. The fused model z obtained with ARM is a linear combination of the models $m \in M$. Given a test sample $\overline{X_j}$, the prediction $\hat{z}_j$ issued by the fused model is the weighted average of model predictions $\hat{z}_j = \sum_{m=1}^{o} W_m \hat{Y}_{mj}$. Thus, the fusion process consists of learning the weight W_m for each model. Let $r = |D_{CV}|$ be the size of the fusion training set, and $o = |\text{M}|$ be the number of models in the ensemble. Here, we assume that the errors for each model are normally distributed. We use the variance in these errors to identify the weights by executing the following steps:

Step 1: Split D_{CV} randomly into two equally sized subsets $D_{CV}^{(1)}$ and $D_{CV}^{(2)}$.

Step 2: Evaluate σ_m^2 which is the maximum likelihood estimate of the variance of the errors, $\overline{e}_m = \{\hat{Y}_{mj} - Y_j | \overline{X_j}, Y_j \in D_{CV}^{(1)}\}$. Compute the sum of squared errors on $D^{(2)}$, $\beta_m = \sum_{j=\frac{r}{2}+1}^{r} (\hat{Y}_{mj} - Y_j)^2$.

Step 3: Estimate the weights using: $W_m = \frac{(\sigma_m)^{-r/2} exp(-\sigma_m^{-2} \beta_m / 2)}{\sum_{j=1}^{o} (\sigma_j)^{-r/2} exp(-\sigma_j^{-2} \beta_j / 2)}$

Step 4: Repeat steps 1-3 for a fixed number of times. Average the weights from each iteration to get the final weights for the models.

5 Experimental Setup

5.1 Million Song Dataset Year Prediction Challenge

The proposed approach is demonstrated with the Million Song Dataset (MSD) year prediction challenge [2], a regression problem in which the goal is to predict the year in which a given song was released. The dataset has 515K songs, each described with 90 features and a year label. The dataset is divided into D_T, D_{CV}, and D_{TEST} accounting for 70%, 10%, and 20% of the data respectively (see Table 2). In addition, we generate 20 subsets $D_T^{f1}...D_T^{f20}$ by sampling half the exemplars from the training set D_T. Note that the *producer effect* issue [2] has been taken into account to perform all the splits.

5.2 Ensemble Configurations

We set up different configurations of Flash by selecting different objective functions and choosing whether or not to factor the data. We select from three objective combinations:

1. MSE: Mean Squared Error + Subtree Complexity
2. CORR: Pearson correlation coefficient + Subtree Complexity
3. MSE-CORR: MSE + Pearson correlation coefficient + Subtree Complexity

All the studied configurations are multi-objective and all use the Subtree Complexity measure to prevent bloating issues. We compare two data strategies:

1. The complete training set D_{TR} is considered in each of the GP instances of the ensemble
2. Data factoring: each GP instance of the ensemble randomly selects a set D_{TR}^{fi}, where $i \in [1;20]$. Additionally, each instance randomly selects v explanatory variables of D_{TR}^{fi}, where $v \in \{5, 10, 20, 40, 60, 80, 90\}$. The f (as in *factoring*) suffix is appended to the name of the configuration when this data strategy is adopted.

The 6 resulting configurations: MSE, MSE$_f$, CORR, CORR$_f$, MSE-CORR, and MSE-CORR$_f$ are summarized in Table 3. The following settings are fixed in all the experiments. Each GP instance is run for 100 generations with a population of 1000 individuals. A time budget of an hour is imposed on the GP ensembles. Finally, the number of iterations of the ARM fusion process is set to 100. We perform 20 replicas of each of the ensemble configurations. Thus, in summary, we perform a total of: *6(configurations)×20(replicas)=120* ensemble runs. All the experiments are run on the same computer, equipped with an Intel Core-i7-3930K composed of 6 cores with hyper-threading running at 3.20GHz and a NVIDIA Geforce GTX 690 that counts two GPUs, each with 1536 CUDA cores. The GPU postifx interpreter (see Figure 2) is compiled with the *fast-math* flag.

6 Results

6.1 Prediction Error Analysis

A key question is whether the objective function designed to suit GPU usage and a strategy of learning with less data compromises quality. Our first merit of

Table 2. MSD splits

D	D_T	D_T^{fi}	D_{CV}	D_{TEST}
100%	70%	35%	10%	20%
515K	362K	181K	51K	102K

Table 3. Configuration of the compared ensembles

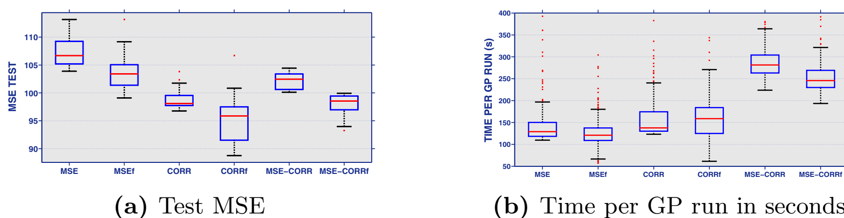
Configuration	Fitness Functions	Factor Data
MSE	MSE, Subtree Comp	no
MSE$_f$	MSE, Subtree Comp	yes
CORR	P. Corr, Subtree Comp	no
CORR$_f$	P. Corr, Subtree Comp	yes
MSE-CORR	MSE, P. Corr, Subtree Comp	no
MSE-CORR$_f$	MSE, P. Corr, Subtree Comp	yes

quality will be prediction error with respect to Mean Squared Error of the unseen data D_{TEST}. Figure 4a shows the boxplots generated with the MSE$_{TEST}$ errors corresponding to the 20 replicas of the runs. First, we observe that, independent of the objective functions, the data factoring strategy leads to a better accuracy. Second, we observe that the maximization of the Pearson correlation coefficient outperforms the standard MSE approach.

To statistically validate these observations, we perform a pairwise Anova test and multiple testing using the Tukey-Kramer or also known as Tukey's honestly significant difference (HSD) method for the prediction errors MSE$_{TEST}$ of the different ensemble configurations. The results are shown in Table 4 where each row presents a test result and the two entries $[x_l, x_u]$ represent the 95% confidence interval between true differences of the mean. Any time the confidence interval does not enclose 0 the difference is significant at $\alpha = 0.05$. The table verifies that factoring is a statistically superior configuration, regardless of objective function: MSE$_f$, CORR$_f$, and MSE-CORR$_f$ respectively outperform MSE, CORR, and MSE-CORR. The superior prediction accuracy of CORR$_f$ versus the remaining approaches is statistically significant.

6.2 Prediction Error vs. GP Instances

We study the impact of the number of GP instances forming the ensemble in the accuracy of the fused model. In Figure 5, we plot the average and standard deviation of the MSE$_{TEST}$ of the fused model when the number of GP instances increases. In all the studied cases, the prediction error of the fused model decreases when a higher number of GP runs are performed. However, a high variability can be observed in the cases where the factoring strategy is employed. It is due to the fact that a fraction of the GP instances learn from a



(a) Test MSE **(b)** Time per GP run in seconds

Fig. 4. Accuracy (a) and time per GP run in seconds (b) of the 6 ensemble configurations

Table 4. Pairwise MSE comparison with ANOVA test

	MSE	MSE_f	CORR	$CORR_f$	MSE-CORR
MSE	-				
MSE_f	[1.08;6.26]	-			
CORR	[6.02;11.20]	[2.35;7.53]	-		
$CORR_f$	[9.72;14.90]	[6.05;11.23]	[1.11;6.29]	-	
MSE-CORR	[2.59;7.77]	[-1.08;4.10]	[-6.02;-0.84]	[-9.72;-4.54]	-
$MSE\text{-}CORR_f$	[6.98;12.16]	[3.31;8.49]	[-1.64;3.55]	[-5.34;-0.16]	[1.80;6.98]

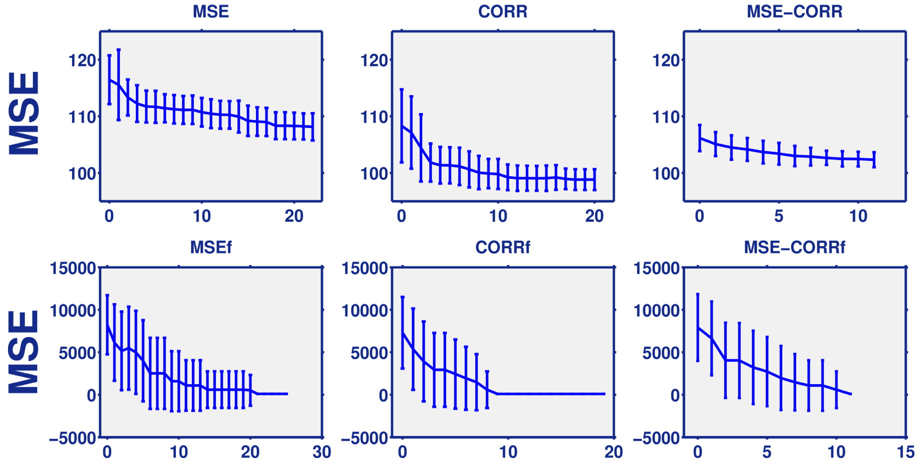


Fig. 5. Average MSE of the fused model with an increasing number of GP instances for the 6 different ensemble configurations

reduced set of non representative variables and achieve poor predictions. However, as more GP instances are considered, the average accuracy increases and the variability decreases. Therefore, ensemble approaches adopting the data factoring strategy need to consider a larger number of GP instances to minimize the variability in the prediction accuracy.

6.3 Runtime Analysis

We analyze whether the data factoring strategies lead to shorter runtimes. The time per GP instance of the compared approaches is shown in Figure 4b. It can be seen that the time necessary to evolve 1000 GP individuals during 100 generation varies from 50 seconds to approximately 400 seconds. To compare the different runtimes, we perform a pairwise Anova test and multiple testing using the Tukey-Kramer or also known as Tukey's honestly significant difference (HSD) method for the time per GP instance retrieved from the 20 replicas of the ensemble runs. The analysis presented in Table 5 shows that MSE_f and MSE-$CORR_f$ are respectively faster than MSE and MSE-CORR. However, the runtime of the CORR and $CORR_f$ approaches is not statistically different.

Table 5. Pairwise time per GP instance comparison with ANOVA test

	MSE	MSE_f	CORR	$CORR_f$	MSE-CORR
MSE	-				
MSE_f	[6.77;22.33]	-			
CORR	[-23.98;-7.57]	[-38.33;-22.33]	-		
$CORR_f$	[-28.61;-12.04]	[-42.95;-26.79]	[-13.04;3.95]	-	
MSE-CORR	[-157.74;-138.17]	[-172.11;-152.89]	[-142.14;-122.22]	[-137.65;-117.61]	-
MSE-$CORR_f$	[-126.21;-107.09]	[-140.58;-121.82]	[-110.61;-91.14]	[-106.13;-86.52]	[20.20;42.40]

7 Conclusions and Future Work

We have presented a GPU-based implementation of a GP ensemble strategy where different GP instances are run sequentially on a single desktop, and the models retrieved from the different runs are fused with Adaptive Regression by Mixing. Our approach is demonstrated with the Million Song Dataset year prediction challenge, a symbolic regression problem that has 515K exemplars. The execution of the evaluation step in a dual-GPU with concurrent kernels allows GP instances with 1000 individuals to run for 100 generations in as few as 50 seconds. The experimental work shows that the implementation of a data reduction strategy in which each GP instance of the ensemble samples a subset of the exemplars and explanatory variables of the data outperforms the standard strategy that considers the whole dataset. It also shows that employing the Pearson correlation coefficient between predictions and targets to drive the search leads to a higher accuracy than the generally used Mean Squared Error metric.

Acknowledgments. The ALFA group gratefully recognizes the financial support of the Li Ka Shing Foundation and the G.E. Global Research Center. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of G.E.

References

1. Banzhaf, W., Harding, S., Langdon, W., Wilson, G.: Accelerating genetic programming through graphics processing units. In: Genetic Programming Theory and Practice VI. Genetic and Evolutionary Computation, pp. 1–19. Springer US (2009)
2. Bertin-Mahieux, T., Ellis, D.P., Whitman, B., Lamere, P.: The million song dataset. In: Proceedings of the 12th International Conference on Music Information Retrieval, ISMIR 2011 (2011)
3. Chitty, D.M.: A data parallel approach to genetic programming using programmable graphics hardware. In: Proceedings of the 9th Annual GECCO Conference, GECCO 2007, pp. 1566–1573. ACM, New York (2007)
4. Dijkstra, E.W.: Algol 60 translation. Supplement, Algol 60 Bulletin 10 (1960)
5. Harding, S., Banzhaf, W.: Fast genetic programming on GPUs. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 90–101. Springer, Heidelberg (2007)

6. Harding, S., Banzhaf, W.: Implementing cartesian genetic programming classifiers on graphics processing units using GPU.NET. In: Proceedings of the 13th GECCO Conference, GECCO 2011, pp. 463–470. ACM, New York (2011)
7. Harding, S.L., Banzhaf, W.: Distributed genetic programming on GPUs using CUDA. In: Hidalgo, I., Fernandez, F., Lanchares, J. (eds.) PABA Workshop, Raleigh, NC, USA, September 13, pp. 1–10 (2009)
8. Kotanchek, M., Smits, G., Vladislavleva, E.: Trustable symbolic regression models: using ensembles, interval arithmetic and pareto fronts to develop robust and trust-aware models. In: Riolo, R., Soule, T., Worzel, B. (eds.) Genetic Programming Theory and Practice V. Genetic and Evolutionary Computation Series, pp. 201–220. Springer US (2008)
9. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 73–85. Springer, Heidelberg (2008)
10. Langdon, W.: A CUDA SIMT interpreter for genetic programming. Tech. Rep. TR-09-05, Department of Computer Science, Strand (June 2009) (revised)
11. Langdon, W.B.: A many threaded CUDA interpreter for genetic programming. In: Esparcia-Alcázar, A.I., Ekárt, A., Silva, S., Dignum, S., Uyar, A.Ş. (eds.) EuroGP 2010. LNCS, vol. 6021, pp. 146–158. Springer, Heidelberg (2010)
12. Lewis, T.E., Magoulas, G.D.: Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In: Proceedings of the 11th GECCO Conference, GECCO 2009, pp. 1379–1386. ACM, New York (2009)
13. Maitre, O., Querry, S., Lachiche, N., Collet, P.: EASEA parallelization of tree-based Genetic Programming. In: 2010 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8 (2010)
14. Maitre, O., Lachiche, N., Collet, P.: Fast evaluation of GP trees on GPGPU by optimizing hardware scheduling. In: Esparcia-Alcázar, A.I., Ekárt, A., Silva, S., Dignum, S., Uyar, A.Ş. (eds.) EuroGP 2010. LNCS, vol. 6021, pp. 301–312. Springer, Heidelberg (2010)
15. NVIDIA Corporation: NVIDIA CUDA C programming guide, version 3.2 (2010)
16. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Population parallel GP on the G80 GPU. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 98–109. Springer, Heidelberg (2008)
17. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Genetic programming on graphics processing units. Genetic Programming and Evolvable Machines 10(4), 447–471 (2009)
18. Veeramachaneni, K., Derby, O., Sherry, D., O'Reilly, U.M.: Learning regression ensembles with genetic programming at scale. In: Proceeding of the Fifteenth GECCO Conference, GECCO 2013, pp. 1117–1124. ACM, New York (2013)
19. Wilson, G., Banzhaf, W.: Linear genetic programming GPGPU on Microsoft Xbox 360. In: IEEE Congress on Evolutionary Computation, pp. 378–385 (2008)
20. Yang, Y.: Adaptive regression by mixing. Journal of the American Statistical Association 96(454), 574–588 (2001)