A MIMD interpreter for Genetic Programming

Vinícius Veloso de Melo[1], Álvaro Luiz Fazenda[1]
Léo Françoso Dal Piccol Sotto[1], and Giovanni Iacca[2]

[1] Federal University of São Paulo
São José dos Campos
São Paulo, Brazil
[2] Department of Information Engineering and Computer Science
University of Trento
Via Sommarive 9, 38123 Povo, Italy

Abstract. Most Genetic Programming implementations use an interpreter to execute an individual, in order to obtain its outcome. Usually, such interpreter is the main bottleneck of the algorithm, since a single individual may contain thousands of instructions that must be executed on a dataset made of a large number of samples. Although one can use SIMD (Single Instruction Multiple Data) intrinsics to execute a single instruction on a few samples at the same time, multiple passes on the dataset are necessary to calculate the result. To speed up the process, we propose using MIMD (Multiple Instruction Multiple Data) instruction sets. This way, in a single pass one can execute several instructions on the dataset. We employ AVX2 intrinsics to improve the performance even further, reaching a median peak of 7.5 billion genetic programming operations per second in a single CPU core.

Keywords: Genetic Programming · Interpreter · Vectorization · Multiple Instruction Multiple Data.

1 Introduction

Genetic Programming (GP) [1] is an evolutionary algorithm that evolves computer programs to perform automatic programming. Standard GP represents individuals as *s*-expressions applying functions to variables, constants or other functions, which are recursively evaluated to obtain the individual's final value. This step is usually the bottleneck of the algorithm: for instance, when dealing with a classification or regression task, GP needs to employ an interpreter to calculate the outcome of each individual, for each sample in the dataset at hand.

Several researchers have investigated approaches to speed up the evaluation process, such as better individuals representation [2,3], compilation of individuals to machine-code [4,5], single-machine parallel (multi-core) evaluation [2], multi-machine distributed evaluation [6], or Single Instruction Multiple Data (SIMD) operations with hardware accelerators by means of FPGAs [7,8].

More recently, General Purpose Graphic Processing Units (GPGPUs) have also been proposed as a computing platform well-suited for parallel GP processes,

with a seminal work in [3], and further extensions focused on sub-tree parallelization [9], two-dimensional stacks [10] and quantum-inspired linear GP [11]. Given that GPUs have thousands of computing units, huge speedups can be achieved since each unit executes the interpreter on a different block of data. However, since GPUs are still considerably more expensive that CPUs (also due to the recent high demand of GPUs from the crypto-currencies market, that led to a further increase in the GPU prices), not everybody can have access to these devices nowadays. Therefore, CPUs can be considered still the main GP computing platform, which justifies further investigations in the direction of exploiting various levels of parallelism on CPUs. For instance, previous literature has proposed CPU-based approaches that try to exploit parallelism at instruction level by means of SIMD instruction sets, such as Streaming SIMD Extensions (SSE) instructions [2], or by converting individuals into SIMD assembly instructions [12]. This way, a single instruction such as an addition can be applied to several data from an array at same time, resulting in large speedups.

Aiming at a further performance improvement in CPU-based GP, this paper proposes the inclusion of MIMD (Multiple Instruction Multiple Data) instruction sets in the GP interpreter. These MIMD operators are compiled to AVX2 (Intel Advanced Vector Extensions version 2) intrinsics[3] and can perform up to four float operations in parallel. It is important to note, though, that this approach is not limited to CPUs: since we use an interpreter, any interpreted GP system can benefit from our approach, even those based on FPGAs and GPUs.

The paper is organized as follows. Section 2 presents the related works. Section 3 introduces our proposed approach. The experimental analysis is presented and discussed in Section 4. Conclusions are given in Section 5.

2 Related work

Over the past few years, a number of approaches have been proposed to improve the performance of GP, by implementing parallelism either at population level (i.e., parallelizing the evaluations of multiple individuals), or at instruction level (i.e., parallelizing the instructions within the evaluation of a single individual).

Most of the modern literature focuses on GPU-based implementations of GP. In this regard, several works have recently achieved parallelism at population level. For instance, Augusto and Barbosa [13] developed a GP system with the OpenCL framework, instead of the regular CUDA programming language used in most of the GPU-based GP literature [14,15,16,17,18], to parallelize multiple individual evaluations. Although not as fast as CUDA, an interesting characteristic of such framework is the automatic generation of machine-code for CPUs and GPUs of any vendor that provide compatibility, whereas CUDA is only available on Nvidia equipment. The authors tested their approach on different CPU and GPU architectures, showing that, as expected, GPUs are several times faster than CPUs.

[3] `https://software.intel.com/en-us/node/523876`

In [19], Harding and Banzhaf have explored yet another alternative to OpenCL and CUDA, namely GPU.NET, a commercial closed-source tool for programming GPUs. They reported promising results, although hindered by the immature level of this technology, that still lacks proper debugging tools.

Staats et al. [20] introduced instead a Python framework named Karoo GP. In the paper, the authors replaced a previous scalar architecture (based on SymPy) with a vector architecture (based on TensorFlow), and tested both approaches on multiple CPU cores and GPUs achieving up to 15x speedup.

Earlier attempts were made to apply instead the parallelism at instruction level. Seminal works in this direction were proposed by Chitty [21], and Harding and Banzhaf [22], who first developed a GPU-based implementation of GP fitness functions. In both works, the authors followed a data parallel approach to achieve high speedups, but they used an individual-compiled approach rather than individual-interpreted.

Langdon and Banzhaf [3] have proposed another GPU implementation where they replaced the traditional prefix-based recursive interpreter by a Reverse Polish Notation (RPN) *postfix* stack-based interpreter. This was the first GP work using SIMD instructions in GPUs. The authors reported a 7x speedup of the GPU version over the CPU version.

Vasicek and Slany [12] proposed a method that is able to efficiently compile a Cartesian GP genotype to an efficient binary machine code, without the need to call the external C compiler. The generated machine code contains SSE/SSE2 SIMD instruction calls operating with 128-bit vectors which may process two to four floating point numbers at once, depending on their precision. However, in this method the translations are performed by the assembly code, thus being dependent on the specific compiler and hardware configurations.

Finally, Chitty [2] investigated the approach of Langdon and Banzhaf [3], but on CPUs. The author used a stack-based interpreter with several improvements such as SIMD instruction sets -including a Multiply-Add operator- and a multi-threaded blocking population parallel approach. In this case the execution time of GP was significantly reduced by better utilizing the cache memory and making efficiency savings. Thus the energy cost of executing GP was also significantly reduced. To the best of our knowledge, this work is the closest to the present paper: however, as it will become clear in the next section, here we introduce more operators, and we use *array* variables, with compiler flags set to obtain the automatic vectorization and generation of AVX2 codes, while in [2] this was manually implemented by means of custom operators based on SIMD calls.

3 Proposed approach

In the SIMD approach, a traditional GP interpreter executes only a single instruction at a time over the entire dataset. Thus, an individual with thousands of instructions performs thousands of loops through the data to calculate the result. In the best-case scenario, an individual should be compiled into a single large expression that passes a single time on the dataset. However, the compi-

lation cost may be higher than running the interpreter. Here, we propose and evaluate a MIMD interpreter where the four arithmetic operations are fused to perform up to three instructions in a single loop.

By using MIMD operators, the interpreter should perform fewer memory accesses than the SIMD operators, due to a loop fusion over SIMD operators loops: this is because MIMD operators use a complex C++ arithmetic instruction in a single line instead of two or more single separated and dependent instructions, enclosed by separated loops.

The implementation used here uses the jump table idea (an array of information about the type of a node, including a function pointer, arity, name, and other characteristics) proposed in [23] where the authors state: *the primary benefit of such a jump table is that now we can select which function to execute with an array de-reference as opposed to a case statement. Although compilers will often compile a case statement into such a jump table, it will still do bounds checking on the index, and so will be slower than the hand coded jump table.* Thus, instead of checking conditions, the jump table allows for a highly efficient code that increments and de-references a pointer, references an array, and makes a function call.

In this paper, we are employing a *postfix* RPN GP interpreter with an explicit stack as proposed by Langdon and Banzhaf [3]. This modification allows the replacement of temporary arrays in the function operators, such as:

float add(valarray a, valarray b)* { *valarray tmp=a+b; return tmp;* }

by in-place operations on the stack like:

void add() { *stack[top-1]+=stack[top]; top−;* }

where *stack* is a 2d-stack structure as used in [2], and *top* is the stack top. In the 2d-stack [2] the first dimension represents the maximal tree depth used, and the second dimension consists of the number of fitness cases. This structure allows better cache-hit rates.

The traditional and proposed operators are shown in Table 1. One may notice that there are only eight functions using four arguments, while the total would be $4^3 = 64$. However, allowing only eight functions of four arguments should be enough to evidence any difference in performance.

Let us assume that a, b, and c are actually positions on the stack. The equation $a+=b+c$ can be either written as $a\ b+c+$ with the traditional '+' operator or as $a\ b\ c\ A$ with the new operator. Similarly, $(a+b)*(c-d)$ can be either written as $a\ b+c\ d-*$ or $a\ b\ c\ d\ U$. Finally, one may sum these two expressions with the traditional operator: $(a+b+c)+((a+b)*(c-d))$ as $a\ b\ c\ A\ a\ b\ c\ d\ U\ +$. One may obtain the same equation with the traditional operators using the following string: $a\ b+c+a\ b+c\ d-*+$. As one may observe, the MIMD operators result in a more compact expression, with 10 elements versus 13 in the previous simple example. This characteristic allows the creation of complex expressions with

Table 1. Traditional and proposed operators, where a, b, c, and d are arrays.

Opcode	#Args	String	Traditional Operation	Proposed Operation
A	3	a b + c +	a = a+b; a = a + c	a += b + c
B		a b + c -	a = a+b; a = a - c	a += b - c
C		a b + c *	a = a+b; a = a * c	a = (a+b)*c
D		a b + c /	a = a+b; a = a / c	a = (a+b)/c
E		a b - c +	a=a-b; a = a + c	a -= b + c
F		a b - c -	a=a-b; a = a - c	a -= b - c
G		a b - c *	a=a+b; a = a * c	a = (a-b)*c
H		a b - c /	a=a-b; a = a / c	a = (a-b)/c
I		a b * c +	a=a*b; a = a + c	a = (a*b) + c
J		a b * c -	a=a*b; a = a - c	a = (a*b) - c
K		a b * c *	a=a*b; a = a * c	a *= b*c
L		a b * c /	a=a*b; a = a / c	a = (a*b)/c
M		a b / c +	a=a/b; a = a + c	a = (a/b) + c
N		a b / c -	a=a/b; a = a - c	a = (a/b) - c
O		a b / c *	a=a/b; a = a * c	a = (a/b)*c
P		a b / c /	a=a/b; a = a / c	a = (a/b)/c
Q	4	a b + c d + +	a=a+b; c=c+d; a = a + c	a += b+c+d
R		a b + c d + -	a=a+b; c=c+d; a = a + c	a = (a+b)-(c+d)
S		a b + c d + *	a=a+b; c=c+d; a = a * c	a = (a+b)*(c+d)
T		a b + c d + /	a=a+b; c=c+d; a = a / c	a = (a+b)/(c+d)
U		a b + c d - *	a=a+b; c=c-d; a = a * c	a = (a+b)*(c-d)
V		a b + c d * -	a=a+b; c=c*d; a = a - c	a = (a+b)-(c*d)
W		a b + c d * /	a=a+b; c=c*d; a = a / c	a = (a+b)/(c*d)
X		a b + c d / -	a=a+b; c=c/d; a = a - c	a = (a+b)-(c/d)

shallower trees than those using only SIMD operators. Therefore, from an evaluation perspective, smaller stacks can be used, saving memory and processing time.

Although both the SIMD and MIMD versions use a stack to interpret the individuals present in the population, MIMD operations decrease the frequency of push and pop operations. MIMD operators also allow FMA (Fused Multiply-Add) instructions, otherwise impossible with SIMD. Instruction Level Parallelism (ILP) was found in the assembly code.

In the next section, we investigate the performance of an interpreter using the proposed MIMD operators. We will first investigate the performance of each of them to evaluate whether they are useful and what is the expected performance increase, and then analyze the overall performance of the MIMD interpreter.

4 Experimental analysis

In the experimentation, we evaluate the performance of the proposed operators on synthetic datasets, so we can control their dimensions. All datasets have ten variables, randomly sampled from as uniform distribution in the interval $[-1.0, 1.0]$. The number of cases is 100, 500, 1K, 5K, 10K, 50K, 100K, 500K, and 1M. This way, we investigate the performance on different data sizes on the CPU cache.

Regarding the interpreter, we test two versions. The first one is the SIMD interpreter, using only the four arithmetical operators with a protected division; thus, if the module of the denominator is less than $1e$-8, then the result is zero. The second version is our proposed MIMD interpreter with the operators presented in Section 3.

Since we are investigating the interpreter's performance, not the solution quality, our code only generates and evaluates a finite number of random individuals (there is no selection, crossover, and mutation) in a *single* generation. Individuals are generated in a ramped half-and-half approach, with depth 20 for SIMD and 4 for MIMD, to approximate the average number of instructions in the population, which will be used as an evaluation criterion. This difference in the depths is necessary because the operators in the SIMD interpreter execute a single instruction and require two arguments, while in the MIMD interpreter an operator can execute up to three instructions on four arguments.

Here, we investigated the performance with populations of 100, 500, 1K, 5K, and 10K individuals. In our experiment, we first generate the population and then investigate only the evaluation process to obtain the *GPops* per second [3] and the cache information.

We have implemented both interpreters in C++ and have compiled the source codes with Intel ICPC compiler using the following flags to enable auto-vectorization and AVX2 instructions:

-Ofast -std=c++11 -funroll-loops -ffast-math
-march=native -mtune=native -xavx2 -m64 -fno-alias.

The *fno-alias* flag is necessary to inform the compiler that the vectors are independent from each other (there is no aliasing), allowing a better optimization. Also, because we are using a stack with indexes and a protect division operation, we must insert two *pragma* in each operator to *force* loop vectorization: *ivdep*, *vector always*, and *simd*. Also, memory alignment is forced with *_mm_malloc* function with 32 bytes, as recommended for AVX2 instructions. SIMD and MIMD operators use *_restricted* pointers to access the stack and data with *__assume_aliased* function on them. These procedures allow using our approach on the Intel C++ Compiler. Adaptations on these configurations are necessary to use other compilers, but performance may vary. In Figure 1, the source-code function to perform OpCode I (table 1) is shown, including the *pragmas* and specific data structure. As one can observe, the compiler identified the multiplication followed by the addition and employed the proper FMA Intel intrinsic[4]. Similarly the Fused Multiply Subtract operation was used for OpCode J (not shown).

The Intel compiler options joined with pragmas depicted above allows AVX2 code generation for an Intel CPU Haswell/Broadwell(R) processor line, as it is possible to check in the assembly code generated by the same compiler, for both SIMD and MIND codes. The FMA (Fused Multiply-Add) instructions are also used for MIMD operators. Instruction Level Parallelism (ILP) was found

[4] https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=FMA&expand=2549

C++

```
void eval_mult_plus()    {
  float * __restrict a = stack[top-2];
  float * __restrict b = stack[top-1];
  float * __restrict c = stack[top];
  __assume_aligned(a,32);
  __assume_aligned(b,32);
  __assume_aligned(c,32);

  #pragma ivdep
  #pragma vector always
  for (int i = 0; i < ncases; ++i)
    a[i] = (a[i]*b[i]) + c[i];

  top-=2;
  GPop+=2;
}
```

Assembly

```
vmovups    (%r10,%rax,4), %ymm2
lea        (%r11,%rax,4), %r15
vmovups    32(%r10,%rax,4), %ymm3
vmovups    (%r9,%rax,4), %ymm0
vmovups    32(%r9,%rax,4), %ymm1
vfmadd213ps (%r15), %ymm0, %ymm2
vfmadd213ps 32(%r15), %ymm1, %ymm3
vmovups    %ymm2, (%r15)
vmovups    %ymm3, 32(%r15)
addq       $16, %rax
cmpq       %rbx, %rax
```

Fig. 1. Opcode I C++ source-code function (left) and assembly code fragment (right) showing the FMA operation (vfmadd213ps) employed by the compiler.

in assembly code generated, including two FMA operations available for AVX registers in the supporting Intel CPU architecture.

The computational environment was an Intel(R) Xeon(R) CPU E5-2660 v4@2.40GHz (CPU frequency scaling set for best performance in all computing cores), with CentOS Linux 3.10.0-327.el7.x86_64 and Intel Compiler icpc 17.0.4 20170411. We ran all the experiments using a single process (single-core), to execute 30 independent runs of each interpreter.

4.1 Evaluation

As explained above, both the SIMD and MIMD interpreters are compiled with the same optimization flags and use the same in-place two-argument operators and stack-based approaches. Therefore, in the experiments we evaluate only the improvement obtained by using the MIMD operators in the evaluation of individuals (ignoring the time to generate the strings).

Langdon and Banzhaf [3] defined Genetic Programming Operations per Second ($GPops/sec$), a metric for measuring the speed of a Genetic Programming implementation. $GPop$ is the number of instructions performed by an individual, including the number of variables, times the number of cases in the dataset. Thus, for an individual $ind{=}a\ b\ +$ and a dataset with 1K cases, $GPop = 3 * 1,000 = 3,000$.

Each operator proposed here is a single Genetic Programming operation. However, we must compare the number of instructions, as the MIMD interpreter may execute several of them at a time. According to Table 1, the operators with three arguments execute two instructions (for instance, $+$, $+$). Therefore, two $GPop$ are counted for these operators. Similarly, three instructions are counted for the operators with four arguments.

To compare the $GPops/sec$ of the two interpreters, we executed the Wilcoxon Rank-Sum Test assuming a significance level of $\alpha = 0.05$. Thus, if $p\text{-}value < \alpha$, then the difference between the performances is considered significant and the interpreter with the highest median is the winner.

4.2 Analysis of each operator

In this first analysis, we compare the performance of SIMD and MIMD operators according to the implementation shown in Table 1. We did 1K independent runs of each operator for a dataset with 1M cases. For each run, we first copy data from temporary arrays to the required variables a, b, c, and d, and then execute the calculation. For the operations with only three arguments, we do not assign any value to d. This copy is to simulate pushing data onto the stack, in order to have a more realistic scenario. The average running times for each operator are shown in Table 2. As one may observe, for operators with three arguments, the speedup was approximately 17%, while for four arguments, the speedup was close to 25%. The average overall speedup was approximately 20%. This value is the expected speedup of the real-case scenario, as the tested equations combine two, three, and four argument operators.

As shown in Table 2, the speedup remains more or less the same inside the same class of operators (three argument and four argument operators). Thus, allowing only eight four argument operators should not introduce any bias towards a specific speedup value, while also providing a higher total speedup than that provided only by three argument operators.

Table 2. Running time of each operator.

Opcode	SIMD	MIMD	Speedup	Opcode	SIMD	MIMD	Speedup
A	138	118	1.169	M	137	117	1.171
B	138	118	1.169	N	137	117	1.171
C	138	120	1.150	O	137	117	1.171
D	144	122	1.180	P	138	119	1.160
E	138	118	1.169	Q	191	152	1.257
F	138	118	1.169	R	191	152	1.257
G	138	118	1.169	S	191	152	1.257
H	140	119	1.176	T	192	153	1.255
I	138	118	1.169	U	191	152	1.257
J	136	116	1.172	V	191	152	1.257
K	136	116	1.172	W	191	152	1.257
L	137	117	1.171	X	192	153	1.255
Average					155.75	129.417	1.198

4.3 Analysis of the interpreter

In this section, we investigate whether the MIMD interpreter achieves the expected 20% speedup observed in the individual analysis of the previous section.

In Figure 2, we present a plot of the overall results. The plot is a median of 30 independent runs for each configuration and shows how the number of $GPops/sec$ varies according to the number of evaluations allowed and the size of the dataset.

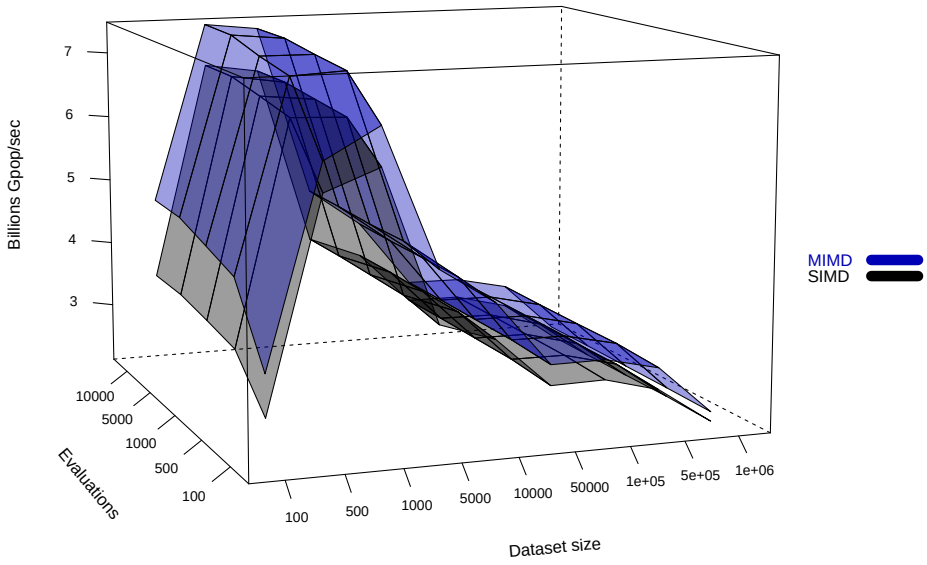

Fig. 2. Performance with respect to dataset size and number of evaluations.

First of all, the performance of the two interpreters show a similar pattern, which is an indirect indication that the MIMD operators are working properly.

As one can notice, the MIMD interpreter is able to outperform the SIMD interpreter in all configurations, executing more $GPops/sec$. A pattern that can be extracted from the plot is that the performance grows from 100 cases to 500 cases, stays stable for 1K cases and then starts dropping. The performance for 1M is lower than that with 100 cases. This is probably explained by the fact that larger datasets require more swap operations between memory and cache, decreasing the impact in performance from the MIMD operators.

The performance difference between the two interpreters is significant, but it also decreases as the dataset size increases. A possible explanation is again that, for larger datasets, the data swap between memory and cache is so frequent that the performance gain obtained with the MIMD operators is negligible. This is a crucial point, that suggests that further investigations are needed to find ways of reducing such swap and eventually increase the number of $GPops/sec$ even with larger datasets.

As for the number of evaluations, the results do not vary significantly when the population size increases. This behavior is expected as both interpreters are just evaluating more individuals on the same data.

The different *GPops/sec* can be better observed in Figure 3, which shows Violin Plots comparing the distribution of performances of the SIMD and MIMD interpreters for the 30 runs, and the *p-value* of the Wilcoxon Rank-Sum Test, for each dataset size and number of evaluations allowed.

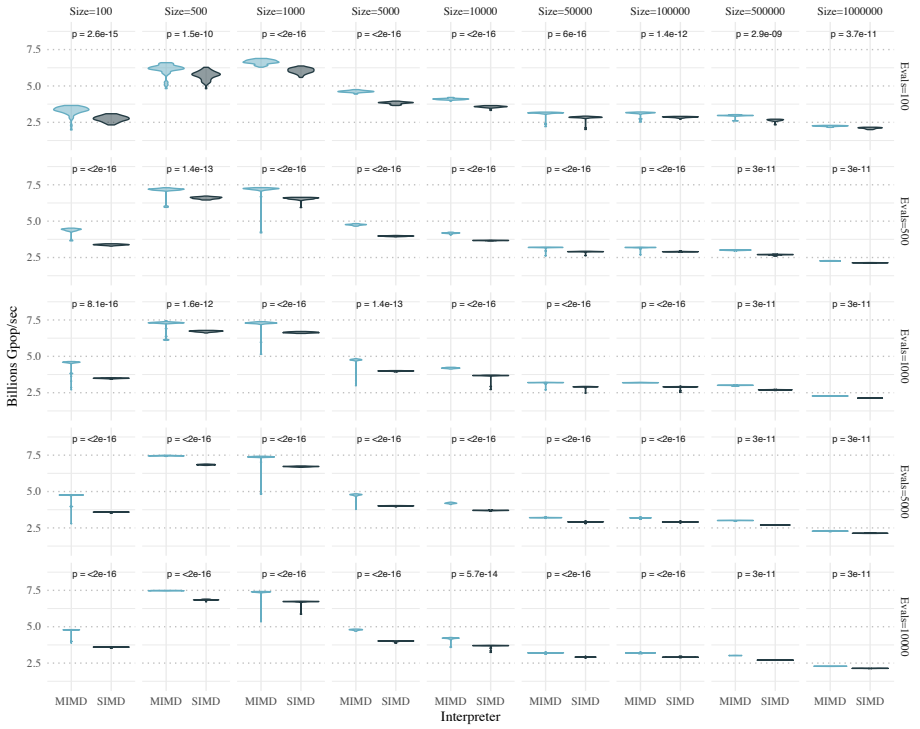

Fig. 3. Distributions of performances with respect to dataset size and number of evaluations.

As one can see, all the performance gains obtained by the proposed MIMD interpreter are statistically significant. The performance differences show the same pattern for all tested numbers of evaluations, as previously discussed when analyzing Figure 2, with the MIMD interpreter processing consistently more instructions per second.

The speedups obtained by the MIMD interpreter are presented in Figure 4 (top), where there is a line for each number of evaluations, depending to the dataset size. Again, there is no significant difference between the trends obtained with different numbers of evaluations, which was expected. However, differently from what happened to the number of *GPops/sec*, here the speedup is large on two dataset sizes (100 and 5K), but also decreases as the dataset size grows.

The unexpected behavior was observed for datasets of sizes 500 and 1K, that showed a substantial reduction in the speedup. In order to try to clarify this issue, we used PAPI(R) - Performance Application Programming Interface, an API which allows to access hardware counters to measure several aspects related to performance analysis, and obtain information about cache sizes and cache misses.

Figure 4 (bottom) shows the cache miss rate evaluated for three cache memory levels: L1, L2 and L3 over GPop ($L *_{misses} /GPop$) for a population of 100 individuals. For the CPU used in our tests, this cache memory levels has 448KB, 3.5MB, and 35MB capacity, respectively. From the figure, it is possible to note three important facts: a) L1 miss rate increases, starting at dataset size 500 up to 5K, where it achieves a steady state; b) L2 miss rate increases, starting at dataset size 5K up to 50K, then it plateaus until 500K, when it increases again; c) L3 miss rate increases, starting at dataset size 1M.

The L1 miss rate increase at dataset size 500 causes no significant performance variation on $GPops/sec$. Until dataset size 5K, the $GPops/sec$ rate continues to increase, due to an increase in the total number of operations. With a small number of GPop and, consequently, a small number of instructions issued, loop initialization, controlling instructions and array indexing influence the performance. However, as the array size increases, these controlling instructions become irrelevant.

The L1 and L2 miss rates that abruptly grow at dataset size 5K significantly decreases the performance, as already seen in Figures 2 and 3. The L2 miss rate continues to increase up to 50K. This L2 cache miss rate behavior decreases the $GPops/sec$ rate quite in the same way (see Figure 2). The L2 miss rate (as well as the performance in $GPops/sec$) reaches a stable state at 50K, up to 500K, but start to increase again at 1M, together with an L3 miss rate increase. This effect causes a further performance decrease.

Comparing the cache miss rate for SIMD and MIMD, it is possible to observe similar trends for both. However, the cache misses counts for L1 and L2 are greater in SIMD than MIMD: this is because SIMD uses more loops and load/store operations to deal with two or more single math instructions, as opposed to MIMD, where more math instructions are combined into a unique instruction.

5 Conclusions and future work

In this paper, we proposed a MIMD interpreter for GP, and evaluated its performance compared to that of a traditional SIMD interpreter. The proposed MIMD interpreter is based on a fusion of the four arithmetical functions into MIMD operators, which allow to execute up to three instructions in a single pass. Here, we tested only a few combinations of these functions, as the number of combinations increases exponentially with the number of functions. For instance, including *sin*, *cos*, *log*, etc. would create thousands of operators. Therefore, we recommend implementing the operators that provide the highest speedups.

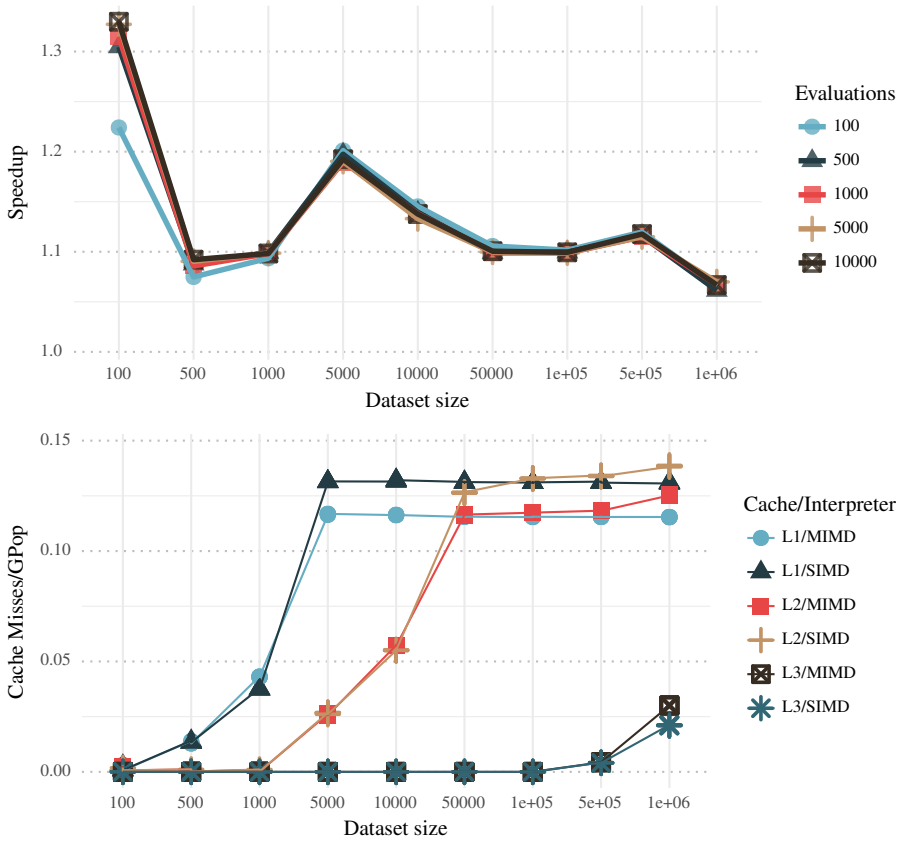

Fig. 4. Top: speedups (MIMD over SIMD) with respect to dataset size and number of evaluations (individuals tested). Bottom: Cache misses of different cache levels and interpreters per GPop.

Our experiments show that a median of 7.5 $GPops/sec$ was achieved in a single CPU core running at 2.4Ghz. Further improvements might be obtained by extending the proposed implementation to use parallel programming on multi-core CPUs and GPU co-processing. Algorithm and hardware acceleration could also be combined. In most of the times, algorithm speedup techniques always promote better results in parallel codes, however, a more detailed evaluation is required in order to check if the algorithm developed does not limit the concurrency, by requiring extra communication or synchronization.

We claim that swap operations between memory and cache are responsible for some of the observed effects, and show some supporting evidence in the cache miss rates. In future work, we could provide more metrics to justify that. We also want to investigate the impact of the proposed MIMD operators on actual GP runs. In this case, there is the necessity of parsing the tree into the list of

strings used by the interpreter, a step that is easily done by traversing the tree and identifying operators used in sequence in order to substitute them by one of the proposed operators. In this case, if more four argument operators were allowed, we should also have higher speedups, as the total speedup corresponds to an average between individual operators speedups.

Acknowledgements

This work was supported by Grant #2016/07095-5, São Paulo Research Foundation (FAPESP).

References

1. Koza, J.R.: Genetic programming as a means for programming computers by natural selection. Statistics and Computing **4**(2) (Jun 1994) 87–112
2. Chitty, D.M.: Fast parallel genetic programming: multi-core CPU versus many-core GPU. Soft Computing **16**(10) (2012) 1795–1814
3. Langdon, W.B., Banzhaf, W.: A SIMD Interpreter for Genetic Programming on GPU Graphics Cards. In: Genetic Programming, Berlin, Heidelberg, Springer Berlin Heidelberg (2008) 73–85
4. Fukunaga, A., Stechert, A., Mutz, D.: A Genome Compiler for High Performance Genetic Programming. In: Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, Morgan Kaufmann (1998) 86–94
5. Nordin, P.: A Compiling Genetic Programming System that Directly Manipulates the Machine Code. In: Advances in Genetic Programming. MIT Press, Cambridge, MA (1994) 311–331
6. Fernández, F., Spezzano, G., Tomassini, M., Vanneschi, L.: 6. In: Parallel Genetic Programming. Wiley-Blackwell, Hoboken, NJ (2005) 127–153
7. Eklund, S.E.: Time series forecasting using massively parallel genetic programming. In: Proceedings International Parallel and Distributed Processing Symposium, New York, IEEE (Apr 2003) 1–5
8. Heywood, M.I., Zincir-Heywood, A.N.: Register Based Genetic Programming on FPGA Computing Platforms. In: Genetic Programming, Berlin, Heidelberg, Springer Berlin Heidelberg (2000) 44–59
9. Cano, A., Ventura, S.: GPU-parallel Subtree Interpreter for Genetic Programming. In: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation. GECCO '14, New York, NY, USA, ACM (2014) 887–894
10. Chitty, D.M.: Faster GPU-based genetic programming using a two-dimensional stack. Soft Computing **21**(14) (Jul 2017) 3859–3878
11. da Silva, C.P., Dias, D.M., Bentes, C., Pacheco, M.A.C., Cupertino, L.F.: Evolving GPU Machine Code. Journal of Machine Learning Research **16** (2015) 673–712
12. Vašíček, Z., Slaný, K.: Efficient Phenotype Evaluation in Cartesian Genetic Programming. In: Genetic Programming, Berlin, Heidelberg, Springer Berlin Heidelberg (2012) 266–278
13. Augusto, D.A., Barbosa, H.J.: Accelerated parallel genetic programming tree evaluation with OpenCL. Journal of Parallel and Distributed Computing **73**(1) (2013) 86 – 100

14. Harding, S.L., Banzhaf, W.: Distributed Genetic Programming on GPUs using CUDA. In: Workshop on Parallel Architectures and Bioinspired Algorithms, Raleigh, NC, USA, Universidad Complutense de Madrid (Sep 2009) 1–10
15. Maitre, O., Lachiche, N., Collet, P.: Fast Evaluation of GP Trees on GPGPU by Optimizing Hardware Scheduling. In: Genetic Programming, Berlin, Heidelberg, Springer Berlin Heidelberg (2010) 301–312
16. Robilliard, D., Marion, V., Fonlupt, C.: High Performance Genetic Programming on GPU. In: Proceedings of the 2009 Workshop on Bio-inspired Algorithms for Distributed Systems. BADS '09, New York, NY, USA, ACM (2009) 85–94
17. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Population Parallel GP on the G80 GPU. In: Genetic Programming, Berlin, Heidelberg, Springer Berlin Heidelberg (2008) 98–109
18. Robilliard, D., Marion-Poty, V., Fonlupt, C.: Genetic programming on graphics processing units. Genetic Programming and Evolvable Machines **10**(4) (Oct 2009) 447
19. Harding, S., Banzhaf, W.: Implementing Cartesian Genetic Programming Classifiers on Graphics Processing Units Using GPU.NET. In: Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation. GECCO '11, New York, NY, USA, ACM (2011) 463–470
20. Staats, K., Pantridge, E., Cavaglia, M., Milovanov, I., Aniyan, A.: TensorFlow Enabled Genetic Programming. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. GECCO '17, New York, NY, USA, ACM (2017) 1872–1879
21. Chitty, D.M.: A Data Parallel Approach to Genetic Programming Using Programmable Graphics Hardware. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation. GECCO '07, New York, NY, USA, ACM (2007) 1566–1573
22. Harding, S., Banzhaf, W.: Fast Genetic Programming on GPUs. In: Proceedings of the 10th European Conference on Genetic Programming. EuroGP'07, Berlin, Heidelberg, Springer-Verlag (2007) 90–101
23. Keith, M.J., Martin, M.C.: Genetic Programming in C++: Implementation Issues. In: Advances in Genetic Programming. MIT Press, Cambridge, MA (1994) 285–310