Parallel GPQUICK

W. B. Langdon
Department of Computer Science, University College, London
w.1angdon@cs.ucl.ac.uk

ABSTRACT

We modified GPQuick to use SIMD parallel floating point AVX 512 bit instructions and 48 threads to give up to 139 billion GP operations per second, 139 giga GPops, on a single Intel Xeon Gold 6126 2.60 GHz server. The multi-threaded single instruction multiple data genetic programming GP interpreter has evolved binary trees of more than 396 million instructions using subtree crossover and run populations for a million generations.

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**;

KEYWORDS

Evolutionary computing, Sextic polynomial, performance

ACM Reference Format:
W. B. Langdon. 2019. Parallel GPQUICK. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3319619.3326770

1 INTRODUCTION

We built a new GP engine based on Andy Singleton's GPQUICK. This allowed us to switch from Boolean [8] to floating point and run up to a million generations [9]. Excluding some special applications or Boolean benchmarks based on graphics hardware (GPUs), this appears to be the fastest single computer GP system, see table.

Without size control we expect bloat and so we need a GP system not only able to run for a million generations[1] but also able to process trees with well in excess of a 100 million nodes[2]. The new system we use is based on Singleton's GPQuick [15],[1],[4], but enhanced to take advantage of both multi-core computing using pthreads and Intel's SIMD AVX parallel floating point operations. Keith and Martin [1] say GPQuick's linearisation of the GP tree will be hard to parallelise. Nevertheless, GPQUICK was rewritten to use 16 fold Intel AVX-512 instructions to do all operations on each node in the GP tree immediately. This leads to a single eval pass and better cache locality but at the expense of keeping a $T = 48$ wide stack of partial results per thread.

[1] The median run for a million generations (pop=48) took 39 hours [9, Figure 2]. Under ideal growing conditions, a million generations for E.Coli corresponds to 38 years.
[2] Again referring to the extended runs in [9, Figure 2], crossover creates highly evolved trees containing $4\,10^8$ nodes. These are by far the largest programs yet evolved.

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07.
https://doi.org/10.1145/3319619.3326770

Table 1: GP Primitives Interpreted Per Second. (Default GPU is nVidia GeForce 8800 GTX. Fragment of [7, Tab. 3].)

Experiment	Pop size	Prog size	Test cases	Speed 10^6 OP/S	GPU
Mackey-Glass	204 800	11.0	1200	895	
Mackey-Glass	204 800	13.0	1200	1056	
Mackey-Glass	204 800	10.2	1200	1720	
Protein	1 048 576	56.9	200	504	
Laser	18 225	55.4	151 360	656	
Laser	5 000	49.6	376 640	190	
Sextic	100	16	200	.5	XBox 360
Sextic	12 500	70.0	100 000	4 073	
Image processing	2 048	2048	$\approx 10^8$	26 200	28×8200
TMBL	120	300	65 536	191 724	260 GTX
Multiplexor-6	12 500	120.6	64	47	
Multiplexor-11	12 500	156.2	2 048	501	
Multiplexor-20	262 144	428.5	2 048	254 000	295 GTX
Multiplexor-37	262 144	915.6	8 192	665 000	295 GTX
GeneChip	16 384	≤63.0	200	314	
Cancer	5 242 880	≤15.0	128	535	
Cancer	5 242 880	12.9	91	1 352	C2050
Cancer	5 242 880	12.9	91	8 517	C2050
Sextic	**4000, 500, 48**	**$4\,10^8$**	**48**	**138 948**	**GPQuick**

The next section describes how GPQUICK was adapted to take advantage of Intel SIMD instructions able to process 16 floating point numbers in parallel and to use Posix threads to perform crossover and fitness evaluation on 48 cores simultaneously. Technical report RN/19/01 [9] describes the experiments and results on Koza's Sextic Polynomial ($x^6 - 2x^4 + x^2$ [3, Tab. 5.1]) with populations of 4000, 500 and 48 trees. RN/19/01 reports the earlier predictions of sub-quadratic bloat [5] and Flajolet limit (depth $\approx\sqrt{2\pi|\text{size}|}$ [6]) to essentially hold. [3]

2 GPQUICK

2.1 Sextic and GPQuick

Andy Singleton's GPQUICK [15] is a well established fast and memory efficient C++ GP framework. In steady state mode [16] it stores GP trees in just one byte per tree node. Using separate parent and child populations doubles this (although [2] shows doubling is not necessary). The 8 bit opcode per tree node allows GPQUICK to support a number of different functions and inputs. Typically the remaining opcodes are used to support about 250 fixed ephemeral random constants [12]. In the Sextic polynomial we have the traditional four binary floating point operations (+, −, × and protected division), an input (x) and 250 constants.

[3] Inspired by Dagstuhl Seminar 18052 on Genetic Improvement of Software.

2.2 AVX GPQuick Tree Evaluation

GPQUICK stores the GP population by flattening each tree into a linear buffer. To avoid heap fragmentation the buffers are all of the same size. Traditionally the buffer is interpreted once per test case by multiple recursive calls to EVAL and the tree's output is the return value of the outer most EVAL. Each nested EVAL moves the instruction pointer one position forward in the tree's buffer, decodes the opcode there and calls the corresponding function. In the case of inputs x and constants a value is returned immediately, whereas ADD, SUB, MUL and DIV will each recursively call EVAL twice to obtain their arguments before operating on them and returning the result. For speed GPQUICK's FASTEVAL, does an initial pass though the buffer and replaces all the opcodes by the address of the corresponding function that EVAL would have called. Thus, originally, EVAL processed the tree $T+1$ times (for T=48 test cases).

The AVX instructions process up to 16 floating point data simultaneously. EVAL was rewritten to take advantage of this. Indeed as we expect trees that are far bigger than the CPU cache (≈16 million bytes, depending on model), EVAL now processes each tree only once. This is achieved by EVAL processing all of the test cases for each opcode, instead of processing the whole of the tree on one test case before moving on to the next test case. EVAL now returns 48 floating point values. This is done via a stack, where each stack level contains 48 values. The AVX instructions operate directly on the top of this stack and EVAL keeps track of which instruction is being interpreted, where the top of the stack is, and (with PTHREADS) which thread is running it. AVX instructions are used to speed loading each constant into the top stack frame. Similarly all 48 test cases (x) are rapidly loaded on to the top of the stack. However, the true power of the implementation comes from being able to use AVX instructions to process the top of the stack and the adjacent stack frame (holding a total of 96 floats) in essentially three instructions to give 48 floating point results.

The depth of the evaluation stack is simply the depth of the GP tree. Fixing the buffer size also effectively places a limit on tree depth [6] near $\sqrt{2\pi \text{buffer size}}$. Thus the user specified tree size limit is converted into an expected maximum depth.

To avoid parallelism creating minor changes in calculated fitness, the final fitness summation is not done as a reduction but instead done in a fixed order with a for loop.

2.3 Posix threads version of GPquick

The second major change to GPQUICK was to delay fitness evaluation so that the whole new population can have its fitness evaluated in parallel. As trees are of different sizes, each fitness evaluation will require a different time. Therefore which tree is evaluated by which thread is decided dynamically. Due to timing variations, even in an otherwise identical run, which tree is evaluated by which thread may be different. However great care is taken so that this cannot affect the course of evolution.

EVAL requires a few data arrays. These are all allocated at the start of the GP run. Those that are read only can be shared by the threads. Each thread requires its own instance of read-write data. To avoid "false sharing", care is taken to align read-write data on cache line boundaries (64 bytes), e.g. with additional padding bytes and `((aligned))`. So that each thread writes to its own cache lines and therefore these cached data are not shared with other threads.

2.4 Multi-threaded Parallel Crossover

Crossover operations were also moved to these parallel threads. Each crossover is done immediately before EVAL is run on the newly created tree. Since crossover involves random choices of parents and subtrees these were unchanged (i.e. left in the sequential code, not done in parallel) and so the children remain unaffected by multithreading. Instead of performing the crossover immediately a small amount of additional information is kept to be read later by the threads. This allows the crossover to be delayed and performed in one of 48 C++ pthreads. This gives an additional ≈two-fold speed up which does not change the course of evolution.

3 CONCLUSIONS

The availability of multi-core SIMD capable hardware has allowed us to push GP performance on single computers with floating point problems to that previously only approached with sub-machine code GP operating in discrete domains [11, 13]. This in turn has allowed GP runs far longer than anything previously attempted whilst evolving far bigger programs.

The new parallel GPQuick code is in `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GPavx.tar.gz`

REFERENCES

[1] M. J. Keith and M. C. Martin. Genetic programming in C++: Implementation issues. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 13, pages 285–310. MIT Press, 1994.

[2] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[3] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

[4] W. B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming.* Kluwer, Boston, 1998.

[5] W. B. Langdon. Linear increase in tree height leads to sub-quadratic bloat. In T. Haynes et al., editors, *Foundations of Genetic Programming*, pages 55–56, Orlando, Florida, USA, 13 July 1999.

[6] W. B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119, Apr. 2000.

[7] W. B. Langdon. Large scale bioinformatics data mining with parallel genetic programming on graphics processing units. In S. Tsutsui and P. Collet, editors, *Massively Parallel Evolutionary Computation on GPGPUs*, Natural Computing Series, chapter 15, pages 311–347. Springer, 2013.

[8] W. B. Langdon. Long-term evolution of genetic programming populations. In *GECCO 2017: The Genetic and Evolutionary Computation Conference*, pages 235–236, Berlin, 15-19 July 2017. ACM.

[9] W. B. Langdon and W. Banzhaf. Faster genetic programming GPquick via multicore and advanced vector extensions. Technical Report RN/19/01, University College, London, London, UK, 23 Feb. 2019.

[10] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In L. Spector et al., editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.

[11] R. Poli and W. B. Langdon. Sub-machine-code genetic programming. In L. Spector et al., editors, *Advances in Genetic Programming 3*, chapter 13, pages 301–323. MIT Press, Cambridge, MA, USA, June 1999.

[12] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[13] R. Poli and J. Page. Solving high-order Boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. *Genetic Programming and Evolvable Machines*, 1(1/2):37–56, Apr. 2000.

[14] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, 1996.

[15] A. Singleton. Genetic programming with C++. *BYTE*, pages 171–176, Feb. 1994.

[16] G. Syswerda. A study of reproduction in generational and steady state genetic algorithms. In G. J. E. Rawlings, editor, *Foundations of genetic algorithms*, pages 94–101. Morgan Kaufmann, Indiana University, 15-18 July 1990. Published 1991.