HAL open science

Computing Long Sequences of Consecutive Fibonacci Integers with TensorFlow

Georgios Drakopoulos, Xenophon Liapakis, Evaggelos Spyrou, Giannis Tzimas, Phivos Mylonas, Spyros Sioutas

To cite this version:

Georgios Drakopoulos, Xenophon Liapakis, Evaggelos Spyrou, Giannis Tzimas, Phivos Mylonas, et al.. Computing Long Sequences of Consecutive Fibonacci Integers with TensorFlow. 15th IFIP International Conference on Artificial Intelligence Applications and Innovations (AIAI), May 2019, Hersonissos, Greece. pp.150-160, 10.1007/978-3-030-19909-8_13 . hal-02363855

HAL Id: hal-02363855

https://inria.hal.science/hal-02363855

Submitted on 14 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Computing Long Sequences Of Consecutive Fibonacci Integers With TensorFlow

Georgios Drakopoulos[1], Xenophon Liapakis[2], Evaggelos Spyrou[3], Giannis Tzimas[4], Phivos Mylonas[1], and Spyros Sioutas[5]

[1] Department of Informatics, Ionian University, Hellas
{c16drak, fmylonas}@ionio.gr
[2] Interamerican SA, Hellas
liapakisx@interamerican.com
[3] NCSR "Demokritos"
espyrou@iit.demikritos.gr
[4] Technological Educational Institute of Western Greece
tzimas@teimes.gr
[5] Computer Engingeering and Informatics Department, University of Patras, Hellas
sioutas@ceid.upatras.gr

Abstract. Fibonacci numbers appear in numerous engineering and computing applications including population growth models, software engineering, task management, and data structure analysis. This mandates a computationally efficient way for generating a long sequence of successive Fibonacci integers. With the advent of GPU computing and the associated specialized tools, this task is greatly facilitated by harnessing the potential of parallel computing. This work presents two alternative parallel Fibonacci generators implemented in TensorFlow, one based on the well-known recurrence equation generating the Fibonacci sequence and one expressed on inherent linear algebraic properties of Fibonacci numbers. Additionally, the question of using lookup tables in conjunction with spline interpolation or direct computation within a parallel context for the computation of the powers of known quantities is explored. Although both parallel generators outperform the baseline serial implementation in terms of wallclock time and FLOPS, there is no clear winner between them as the results rely on the number of integers generated. Additionally, replacing computations with a lookup table degrades performance, which can be attributed to the frequent access to the shared memory.

Keywords: Fibonacci sequence · Linear recurrence equations · Finite differences · Google TensorFlow · GPU computing · Parallel computing.

1 Introduction

The sequence of Fibonacci integers $\langle f_k \rangle$ appears often in a broad spectrum of engineering applications including coding theory, cryptography, simulation, and software management. Additionally, Fibonacci numbers are very closely tied to

the golden ratio φ which is frequently encountered in nature, such as in population growth models and in botanics. Moreover, in architecture φ is almost considered synonymous to harmony. Thus, Fibonacci integers are arguably among the most signifiant sequences. Although their defining linear recurrence equation is simple, serially generating a long sequence of consecutive Fibonacci integers is by no means a trivial task.

However, with the advent of GPU computing, the efficient parallel generation of $\langle f_k \rangle$ has been rendered feasible. Indeed, by exploiting known properties of the Fibonacci integers it is possible to build parallel generators which exploit the underlying hardware potential to a great extent, achieving low response times. This requires specialized scientific software such as TensorFlow which not only contains very efficient libraries, but also facilitates the development of high quality custom source code.

The primary research contribution of this work is twofold. First, it lays the groundwork for two parallel Fibonacci integer generators, one based on a closed form for each number in the sequence and one based on certain linear algebraic properties of Fibonacci integer pairs. The performance of the two proposed generators developed in TensorFlow for Python is evaluated in terms of both total turnaround time and FLOPS against a serial implementation with the same software tools. Second, it explores the question whether it is worth substituting the computation of known quantities with a lookup table.

This conference paper is structured as follows. Section 2 reviews current scientific literature regarding the computational aspects of Fibonacci numbers. Their fundamental properties are described in section 3. The TensorFlow implementation is presented in section 4, whereas future research directions are outlined in section 5. Table 1 summarizes the notation of this work. Concerning notation, matrices and vectors are depicted with boldface uppercase and boldface lowercase respectively, whereas roman lowercase is reserved for scalars.

Table 1. Notation of this conference paper.

Symbol	Meaning
$\triangleq$	Definition or equality by definition
$\|\mathbf{x}\|$	Norm of vector $\mathbf{x}$
$\det(\mathbf{A})$	Determinant of matrix $\mathbf{A}$
$\mathrm{tr}(\mathbf{A})$	Trace of matrix $\mathbf{A}$
φ	Golden ratio
$\langle s_n \rangle$	Sequence of integers s_n

2 Previous Work

The Fibonacci sequence of integers, examined among others in [6] and [30], has perhaps the most applications not only in computer science and in engineering

but in science as a whole. Closed forms for the spectral norms of circulant matrics whose entries are either Fibonacci or Lucas integers are derived in [21]. In data structure analysis the Fibonacci heap [19] and the associated pairing heap [18] have efficient search and insertion operations with numerous applications such as network optimization. A pair of successive Fibonacci numbers are known to be the worst case in Euclidean integer division algorithm as shown in [9] as well as in [29]. Fibonacci numbers play a central role in estimating task duration and, consequently, task difficulty in scrum based software engineering methodologies [28][27], including inaccurate estimation discovery [25] and using agile methodologies to predict student progress [26]. Moreover, Fibonacci numbers are very closely linked to the golden ratio φ[6] as well as to symmetry of many geometric shapes, the latter having important implications in group theory [16]. Many identities in combinatorics regarding Fibonacci numbers can be found in [31] as well as in the most recent works [5] and [23]. Finally, the Lucas sequence $\langle \ell_k \rangle$ is closely associated with the Fibonacci sequence $\langle f_k \rangle$ since the two integer sequences constitute a Lucas complementary pair and share similar properties such as growth rate and generation mechanism [3][4].

TensorFlow, originally developed by Google for massive brain circuit simulation, is an open source computational framework whose algorithmic cornerstone is the dataflow paradigm as described in [2], [1], or [20]. A library for generating Gaussian processes in TensorFlow is presented in [24]. For a genetic algorithm implementation in TensorFlow for heuristically discovering community structure, a problem examined in [10], in large multilayer graphs, such as those presented in [12] and in [11], see [15]. In [14] the ways insurance and digital health markets can benefit from blockchain and GPU computing are explored. For a path and triangle based graph resilience metric in TensorFlow see [13]. A very popular front end for the low level TensorFlow is keras, which allows the easy manipulation of neural network layers, including connecticity patterns and activation functions [8]. Model training and prediction generation is done also easily in keras in four stages [22]. Convolutional kernels whose lengths depend on their relative location inside the neural network architecture for computational vision purposes implemented in keras are introduced [7].

3 Fibonacci Numbers

The n-th integer in the Fibonacci sequence $\langle f_n \rangle$ is defined as:

$$f_n \;=\; f_{n-1} + f_{n-2}, \quad f_0 \;=\; 0, f_1 \;=\; 1 \tag{1}$$

Theorem 1. *The n-th Fibonacci number f_n has the closed form:*

$$f_n \;=\; \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \;=\; \frac{\varphi^n + (1-\varphi)^n}{\sqrt{5}} \tag{2}$$

[6] OEIS sequence A001622

Proof. The characteristic polynomial of (1) is:

$$z^2 = z + 1 \Leftrightarrow z^2 - z - 1 = 0 \tag{3}$$

And its two real and distinct roots are:

$$r_{0,1} = \frac{1 \pm \sqrt{5}}{2} \tag{4}$$

Therefore, it follows that:

$$f_n = \alpha_0 r_0 + \alpha_1 r_1 = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \tag{5}$$

The constants α_0 and α_1 are computed using the initial conditions derived by the first two Fibonacci numbers as follows:

$$\begin{aligned} \alpha_0 + \alpha_1 &= f_0 = 0 \\ \alpha_0 r_0 + \alpha_1 r_1 &= f_1 = 1 \end{aligned} \tag{6}$$

The above conditions yield:

$$\begin{aligned} \alpha_0 &= \frac{1}{r_0 - r_1} = \frac{1}{\sqrt{5}} \\ \alpha_1 &= -\alpha_0 = \frac{1}{r_1 - r_0} = -\frac{1}{\sqrt{5}} \end{aligned} \tag{7}$$

□

Another way to prove theorem 1 is the following:

Proof. Another way to directly compute the n-th Fibonacci number f_n is to rewrite the Fibonacci definition of equation (1) and the identity $f_n = f_n$ combined in matrix-vector format as follows:

$$\begin{bmatrix} f_n \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f_{n-1} \\ f_{n-2} \end{bmatrix} = \mathbf{A} \begin{bmatrix} f_{n-1} \\ f_{n-2} \end{bmatrix} = \mathbf{A}\mathbf{f}_{n-2} \tag{8}$$

The eigenvalues λ_1 and λ_2 (recall that $\det(\mathbf{A}) = \lambda_1\lambda_2$ and that $\mathrm{tr}(\mathbf{A}) = \lambda_1 + \lambda_2$) and the corresponding eigenvectors $\mathbf{e}_1$ and $\mathbf{e}_2$ are:

$$\lambda_1 = \frac{1+\sqrt{5}}{2} \triangleq \varphi \qquad \lambda_2 = \frac{1-\sqrt{5}}{2} = 1 - \varphi \tag{9}$$

$$\mathbf{e}_1 = \frac{1}{\sqrt{1+\lambda_1^2}} \begin{bmatrix} \lambda_1 \\ 1 \end{bmatrix} \qquad \mathbf{e}_2 = \frac{1}{\sqrt{1+\lambda_2^2}} \begin{bmatrix} \lambda_2 \\ 1 \end{bmatrix} \tag{10}$$

Notice that $\|\mathbf{e}_1\|_2 = 1$ and $\|\mathbf{e}_2\|_2 = 1$ and, additionally, $\mathbf{e}_1^T\mathbf{e}_2 = 0$. From the spectral decomposition of $\mathbf{A}$ it follows that:

$$\begin{aligned} \mathbf{A} &= \lambda_1 \mathbf{e}_1 \mathbf{e}_1^T + \lambda_2 \mathbf{e}_2 \mathbf{e}_2^T \\ \mathbf{A}^n &= \lambda_1^n \mathbf{e}_1 \mathbf{e}_1^T + \lambda_2^n \mathbf{e}_2 \mathbf{e}_2^T \\ \mathbf{A}^n \mathbf{f}_0 &= \mathbf{f}_{n-1} = \lambda_1^n \left(\mathbf{e}_1^T \mathbf{f}_0\right) \mathbf{e}_1 + \lambda_2^n \left(\mathbf{e}_2^T \mathbf{f}_0\right) \mathbf{e}_2 \end{aligned} \tag{11}$$

Observe that the first element of vector $\mathbf{f}_{n-1}$ is f_n and it is equal to:

$$f_n = \lambda_1^{n+1}\frac{1+\lambda_1}{1+\lambda_1^2} + \lambda_2^{n+1}\frac{1+\lambda_2}{1+\lambda_2^2} \tag{12}$$

Finally, the structure of $\mathbf{A}^n$ can be shown by induction to be:

$$\mathbf{A}^n = \begin{bmatrix} f_n & f_{n-1} \\ f_{n-1} & f_{n-2} \end{bmatrix} \tag{13}$$

□

Despite form (5), Fibonacci numbers as evident by the initial conditions and their generation mechanism. Another way to see this is the following theorem:

Theorem 2. *Fibonacci numbers are integers.*

Proof. Applying the Newton binomial theorem to (5) yields:

$$f_n = \frac{1}{\sqrt{5}}\sum_{k=0}^{n}\binom{n}{k}2^{-n}\underbrace{\left(\sqrt{5}^k - \left(-\sqrt{5}\right)^k\right)}_{\gamma_k} \tag{14}$$

When $k \equiv 0 \pmod 2$ then γ_k is zero, whereas if $k \equiv 1 \pmod 2$ then γ_k equals $2 \cdot 5^{\frac{k+1}{2}}$. Therefore:

$$f_n = \sum_{k}\binom{n}{k}2^{1-n}5^{\frac{k}{2}}, \qquad k \equiv 1 \pmod 2 \tag{15}$$

□

4 TensorFlow Implementation

As stated earlier, TensorFlow relies on the dataflow algorithmic paradigm, which essentially utilizes a potentially large operations graph in order to break down the desired computational task to smaller manageable components. Dataflow graphs have the following properties:

- Vertices represent a wide array of mathematical operations including advanced ones such as eingenvector computation, singular value decomposition, and least squares fitting.
- Edges describe the directed data flow between operation results.
- The operands between the various graph operations are tensors of aritrary dimensions as long as they are compatible.

TensorFlow r1.12 was installed to Ubuntu 18.04 LTS for Python 3.6 using the pip package installer. An NVIDIA Titan Xp GPU based on Pascal architecture was available in the system and was successfully discovered by TensorFlow as gpu0.

Three Fibonacci generators were implemented in total. Each such generator yields a batch consisting of the first n consecutive Fibonacci integers. In the experiments, n ranged from 2 to 1024 with an exponentially increasing distance between two successive batch sizes. Since the particular GPU has 3840 CUDA cores, the parallelism potential is high. The generators are:

- A serial implementation which consists of a single loop which adds one new Fibonacci number with each pass.
- A parallel implementation which directly computes the k-th element of the sequence $\langle f_k \rangle$ based on equation (11).
- A second implementation which relies on the slightly simpler closed expression of equation (2).

Figure 1 shows the total number of floating point operations which were required for each batch size. The values are the arithmetic mean of ten executions for each batch size. Preceding each such execution there was a trial run not taken into consideration in the final results which served the single purpose of loading the data into system cache. Since the serial implementation is a loop, the number of additions is linear in terms of n. On the contrary, the parallel implementations require a number of auxiliary floating point operations, most notably the exponentiation of certai parameters. Thus, they require more operations whose number is a polynomical function, approximately quadratic, of batch size, with the second generator clearly always being more more expensive in terms of operations.

However, the fact that a generator requires more floating point operations does not necessarily makes it slower in terms of total execution time. Instead, the results shown in figure 2 indicate that both parallel generators achieve considerably lower wallclock execution time in milliseconds. Since the computation of $\langle f_k \rangle$ is GPU-bound process and the design of both parallel generators entail very low communication across the memory hierarchy, ordinary wallclock time in this case consists almost entirely of time spent to actual computations.

Notice that, unlike the previous figure, no parallel implementation appears to be ideal for every batch size. Specifically, the second implementation is better for lower sized batches, whereas the first one becomes more preferable as batch size grows despite its more complex formula. This can be attributed to the fact that the second generator achieves more locality as certain parameters are common across each batch size.

This difference in performance can be also seen by dividing the number of floating point operations to the wallclock time, yielding an approximation of the FLOPS for each generator as seen in figure 3. Although by no means a single absolute benchmark, especially within a parallel computation context, FLOPS are in this case indicative since the algorithmic core is purely computational. The difference from the serial implementation is now obvious, as is also the fact that the second generatorfor large batches performs better.

Notice that in both parallel implementations appear many consecutive powers of known quantities. In order to save floating point operations, a lookup table could have been used according the design principles found for instance in [17].

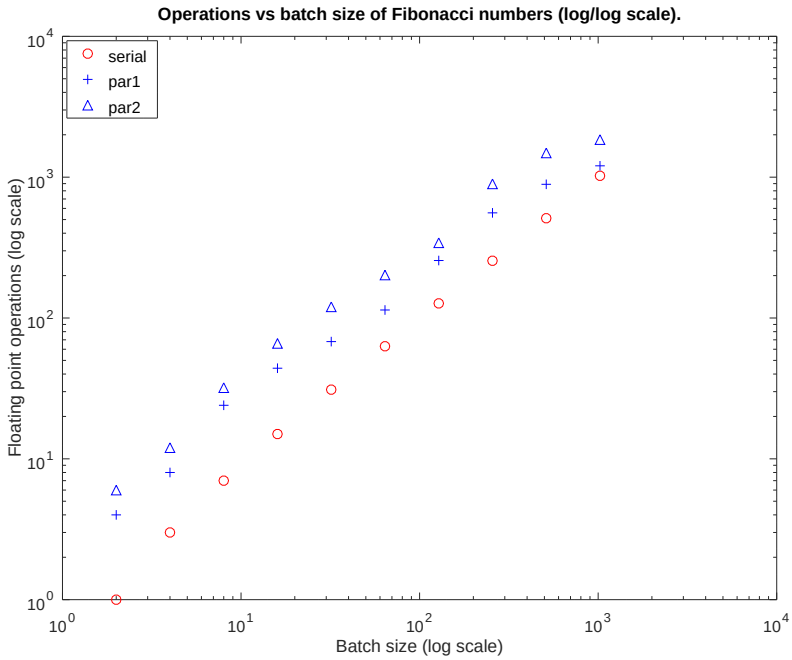

Fig. 1. Number of operations vs batch size of Fibonacci numbers.

In order to evaluate the impact of relying on a lookup table to the total FLOPS for the two parallel generators, two variants of each were also implemented. The first version used locally half the known quantities required, whereas the second only only one quarter of them. In both cases, and in order to achieve comparable result accuracy for fairness reasons, spline interpolation was used. As it can be seen from figure 4, the introduction of a lookup table for both generators downgraded their FLOPS counter. This can be explained from the facts that TensorFlow has an efficient multipication algorthm especially for large numbers and that frequently accessesing the shared GPU memory eventually slowed computations down.

5 Conclusions And Future Work

This conference paper presented two parallel Fibonacci integer generators for TensorFlow running over Python 3.6 and an NVIDIA Titan Xp GPU and discussed certain implementation aspects. Both generators yield batches of n integers, with n ranging from 2 to 1024. The maximum batch size is smaller than the number of cores in the GPU, increasing thus parallel potential. The baseline was a serial implementation for TensorFlow consisting of a single loop generating one new Fibonacci integer in each pass.

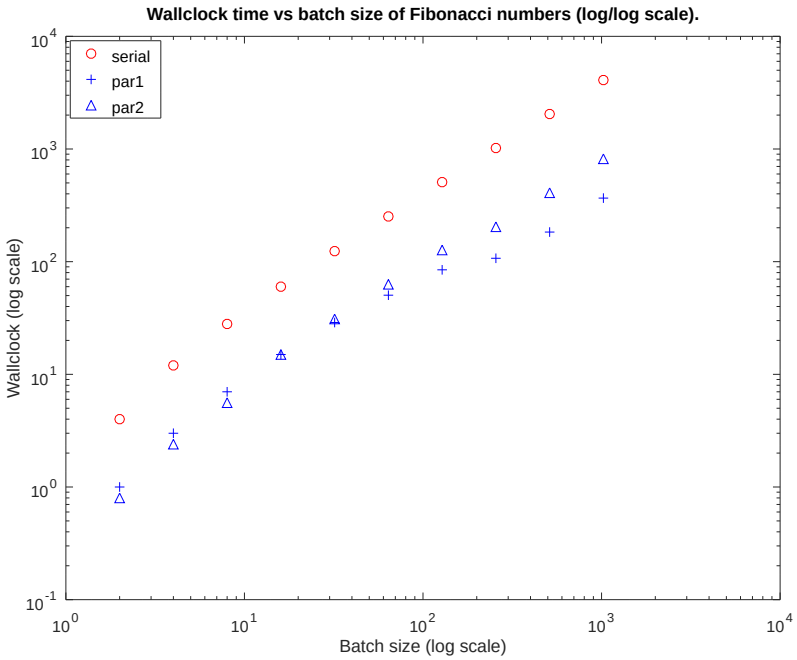

Fig. 2. Wall clock time (msec) vs batch size of Fibonacci numbers.

The primary finding of the experiments is the superior performance of parallel generators. Although requiring more floating point operations in total, both parallel implementations outperform the baseline in terms of wallclock time and of FLOPS. This is attributed to the efficient use of parallelism. A secondary finding was that replacing actual computations with frequent accesses to a shared lookup table led to lower FLOPS values. This can be explained by the latency caused by a large number of threads asking for the same information.

Concerning future research directions, there is a number of options which can be followed. Experiments with larger batch sizes should be conducted, especially with sizes which exceed the number of GPU cores. Additionally, more algorithmic schemes should be tested, such as those constructing Fibonacci integers bitwise, as they may led in generators with higher parallelism. Finally, the performance of the proposed algorithms to other parallel architectures can be evaluated, in order to understand whether a given hardware architecture is more appropriate for particular algorithmic principles.

Acknowledgements

This research was conducted within the project “Development of technologies and methods for cultural inventory data (ANTIKLIA)” under the EPAnEK

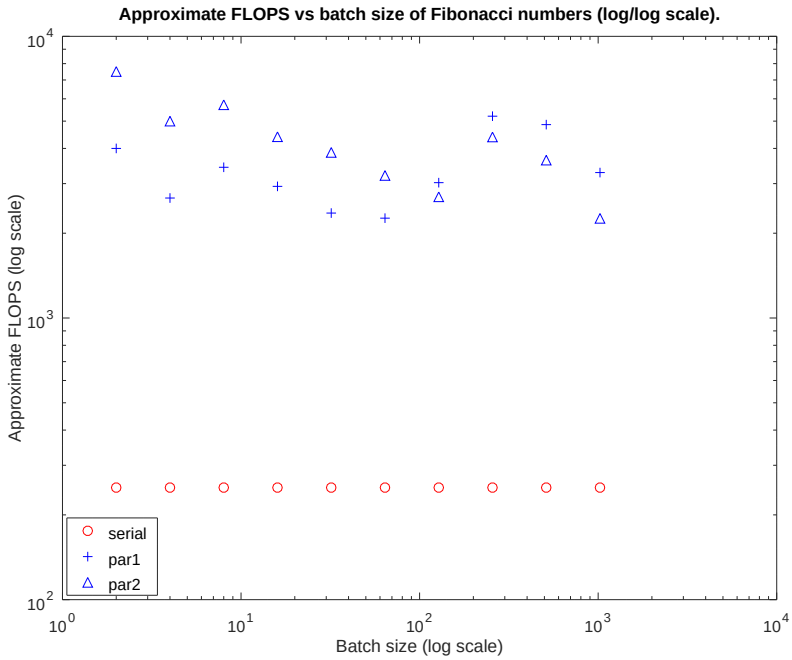

Fig. 3. Approximate FLOPS vs batch size of Fibonacci numbers.

2014-2020 Operational Programme Competitiveness, Entrepreneurship, Innovation.

Additionally, this conference paper is part of Project 451, a long term research initiative whose primary objective is the development of novel, scalable, numerically stable, and interpretable tensor analytics.

Moreover, the authors gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.

References

1. Abadi, M.: TensorFlow: Learning functions at scale. ACM SIGPLAN Notices **51**(9) (2016)
2. Abadi, M., et al.: TensorFlow: A system for large-scale machine learning. In: OSDI. vol. 16, pp. 265–283 (2016)
3. Akbary, A., Wang, Q.: A generalized Lucas sequence and permutation binomials. Proceedings of the American Mathematical Society **134**(1), 15–22 (2006)
4. Bilgici, G.: Two generalizations of Lucas sequence. Applied mathematics and computation **245**, 526–538 (2014)
5. Bolat, C., Köse, H.: On the properties of k-Fibonacci numbers. Int. J. Contemp. Math. Sciences **5**(22), 1097–1105 (2010)
6. Capocelli, R., Cerbone, G., Cull, P., Holloway, J.: Fibonacci facts and formulas. Springer (1990)

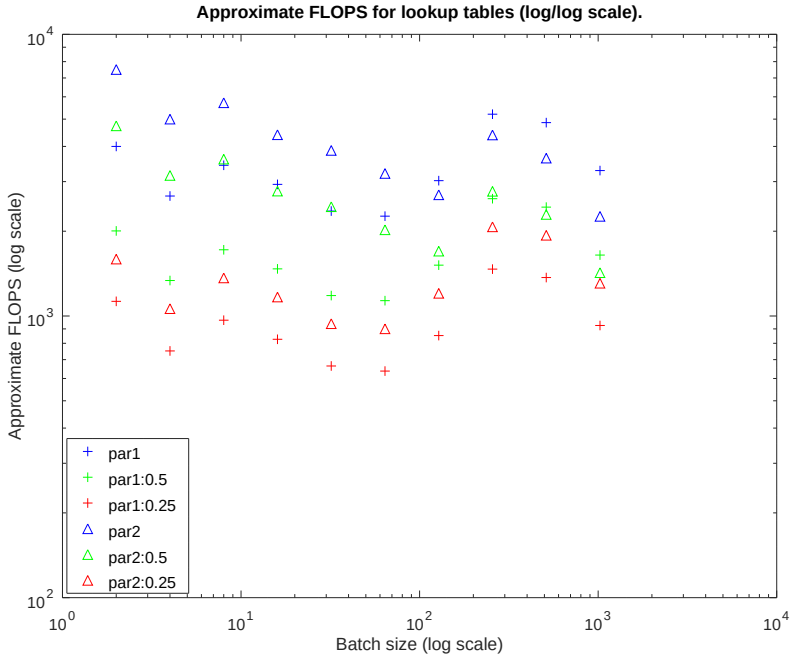

Fig. 4. Approximate FLOPS for two lookup table options.

7. Chollet, F.: Xception: Deep learning with depthwise separable convolutions. In: CVPR. pp. 1251–1258 (2017)
8. Chollet, F., et al.: Keras: Deep learning library for theano and TensorFlow (2015)
9. Dixon, J.D.: The number of steps in the Euclidean algorithm. Journal of number theory **2**(4), 414–422 (1970)
10. Drakopoulos, G., Gourgaris, P., Kanavos, A.: Graph communities in Neo4j: Four algorithms at work. Evolving Systems (June 2018). https://doi.org/10.1007/s12530-018-9244-x
11. Drakopoulos, G., Kanavos, A., Karydis, I., Sioutas, S., Vrahatis, A.G.: Tensor-based semantically-aware topic clustering of biomedical documents. Computation **5**(3) (May 2017)
12. Drakopoulos, G., Kanavos, A., Tsolis, D., Mylonas, P., Sioutas, S.: Towards a framework for tensor ontologies over Neo4j: Representations and operations. In: IISA (August 2017)
13. Drakopoulos, G., Liapakis, X., Tzimas, G., Mylonas, P.: A graph resilience metric based on paths: Higher order analytics with GPU. In: ICTAI. IEEE (November 2018)
14. Drakopoulos, G., Marountas, M., Liapakis, X., Tzimas, G., Mylonas, P., Sioutas, S.: Blockchain for mobile health applications: Acceleration with GPU computing. In: Vlamos, P. (ed.) GeNeDis 2018. Springer (2018)
15. Drakopoulos, G., Stathopoulou, F., Kanavos, A., Paraskevas, M., Tzimas, G., Mylonas, P., Iliadis, L.: A genetic algorithm for spatiosocial tensor clustering: Exploiting TensorFlow potential. Evolving Systems (January 2019). https://doi.org/10.1007/s12530-019-09274-9

16. Dunlap, R.A.: The golden ratio and Fibonacci numbers. World Scientific (1997)
17. Fateman, R.J.: Lookup tables, recurrences and complexity. In: International symposium on symbolic and algebraic computation. pp. 68–73. ACM (1989)
18. Fredman, M.L., Sedgewick, R., Sleator, D.D., Tarjan, R.E.: The pairing heap: A new form of self-adjusting heap. Algorithmica **1**(1-4), 111–129 (1986)
19. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM **34**(3), 596–615 (1987)
20. Goldsborough, P.: A tour of TensorFlow. arXiv preprint 1610.01178 (2016)
21. İpek, A.: On the spectral norms of circulant matrices with classical Fibonacci and Lucas numbers entries. Applied Mathematics and Computation **217**(12), 6011–6012 (2011)
22. Ketkar, N.: Introduction to keras. In: Deep learning with Python, pp. 97–111. Springer (2017)
23. Koshy, T.: Fibonacci and Lucas numbers with applications. Wiley (2019)
24. Matthews, D.G., et al.: GPflow: A Gaussian process library using TensorFlow. The Journal of Machine Learning Research **18**(1), 1299–1304 (2017)
25. Raith, F., Richter, I., Lindermeier, R., Klinker, G.: Identification of inaccurate effort estimates in agile software development. In: APSEC. vol. 2, pp. 67–72. IEEE (2013)
26. Rodríguez, G., Soria, Á., Campo, M.: Measuring the impact of agile coaching on students performance. Transactions on education **59**(3), 202–209 (2016)
27. Rubin, K.S.: Essential scrum: A practical guide to the most popular agile process. Addison-Wesley (2012)
28. Schwaber, K., Sutherland, J.: The scrum guide. Scrum Alliance **21** (2011)
29. Sorenson, J.: An analysis of Lehmer's Euclidean GCD algorithm. In: International symposium on symbolic and algebraic computation. pp. 254–258. ACM (1995)
30. Takahashi, D.: A fast algorithm for computing large Fibonacci numbers. Information Processing Letters **75**(6), 243–246 (2000)
31. Zhang, W.: Some identities involving the Fibonacci numbers. Fibonacci Quarterly **35**, 225–228 (1997)