Viterbi Algorithm Specialization Using Linear Algebra

Ivan Tyulyandin
Mathematics and Mechanics Faculty
Saint Petersburg State University
Saint Petersburg, Russia
i.tyulyandin@2015.spbu.ru

Daniil Berezun
Department of Mathematics and Computer Science
Saint Petersburg State University
Saint Petersburg, Russia
d.berezun@2009.spbu.ru

Semyon Grigorev
Mathematics and Mechanics Faculty
Saint Petersburg State University
Saint Petersburg, Russia
s.v.grigoriev@spbu.ru

***Abstract*— Algorithms based on linear algebra is widely used in various areas. Often programs of interest have some input data that are independent of the dataset being processed, and thus they can be optimized with respect to this input. In the paper, we study how partial evaluation affects the Viterbi algorithm as a step to research an application of partial evaluation to linear algebra-based algorithms. We evaluate the specialized multi-thread Viterbi algorithm against existing GPU-based CUDAMPF. The results show that the presented algorithm is slower but comparable to CUDAMPF.**

I. Introduction

Algorithms based on linear algebra is widely used in various areas such as machine learning [1], computer vision [2], statistics [3] analysis of logic programs [4], graph theory [5], etc. One of the typical cases is querying huge database or graph processing and can be executed in days or weeks making crucial even a simple constant time optimization. One way to optimize such data processing is to use some hardware capabilities such as different kinds of parallelism or to invent a more efficient algorithm or some tricky data representation. We focus on an alternative way to program optimization based on the following observation. Often programs of interest have some input data that are independent of the dataset being processed, and thus they can be optimized with respect to this input. Partial evaluation, a.k.a. program specialization, is a well-known program transformation technique aiming to perform such an optimization [6].

In the paper, we study how partial evaluation affects the Viterbi algorithm [7] as a step to research an application of partial evaluation to linear algebra-based algorithms. First, the Viterbi algorithm is used in bioinformatics [8], speech recognition [9], and financial computations [10]. Second, it can be expressed in terms of linear algebra [11]. Third, the algorithm has two parameters: a hidden Markov model (HMM) [12] and an observations sequence. Its goal is to count a probability for the sequence to be emitted by the given HMM. Next, a sequential application of the algorithm with a fixed HMM to a big bunch of observations sequences is usual. Finally, the main part of the Viterbi algorithm heavily depends on the HMM. All the above make the algorithm a good candidate to research partial evaluation application.

The rest of the paper organized as follows. Section II describes the background. In section III the Viterbi algorithm specialization is explained. Section IV reports benchmarks results. Related work is reviewed in section V. And section VII ends up the paper.

II. Background

In this section, we review specialization, hidden Markov models (HMM), and the Viterbi algorithm in terms of linear algebra.

A. Specialization

Specialization [6], or partial evaluation, is a well-known program transformation technique widely used when some of the input data is already known in compile time. A typical case is serial data processing when one of the input parameters is fixed while others vary. Fixed parameters are called *static* while other parameters are called *dynamic*. The idea behind specialization is that optimization of a program with respect to the static parameters together with executing the optimized program on a set of dynamic parameters may be more efficient than iterative execution of the initial program on both static and dynamic parameters.

The classical specialization example is the exponentiation function $f(x, n) = x^n$ where n is static. A simple implementation is shown below.

```
1 function f(x, n)
2   if n == 0 then 1
3   elif even(x) then f (x, n/2) ^ 2
4   else x * f (x, (n-1)/2) ^ 2
```

All recursive calls are *static*, i.e. are controlled by the static parameter only, and thus can be reduced. Given fixed n, say 5, a typical specialized version is

```
1 function f_spec(x) = x * (x ^ 2) ^ 2
```

Note, sometimes specialization is useless. For example, consider the exponentiation function with fixed base x but dynamic power n. Of course, one may use arithmetic tricks for some x but in general, there is no recipe for effective specialization. Moreover, since optimal specialization is obviously

undecidable there are some heuristics to ensure specialization termination. As a result, in some cases, specialization worsen program execution. For example, it is well-known that sometimes specialization negatively affects program execution caused by code expansion [6].

B. Hidden Markov model

Hidden Markov model is a deterministic probability automaton [12]. It has the following parameters:

- $S_{1..N}$ — N states of the automaton;
- $O_{1..K}$ — K possible observations;
- $B_{1..N}$ — a probabilities describing each state from $S_{1..N}$ to be a start one;
- $T_{1..N,1..N}$ — state transition matrix, $T_{i,j}$ is a probability to go from state S_i to state S_j;
- $E_{1..N,1..K}$ — emission matrix, where $E_{i,j}$ defines probability to emit observation O_j at state S_i.

With a given observation sequence, it is possible to calculate a maximum likelihood to be at a concrete state of the HMM, i.e. to reveal hidden states, according to the sequence. HMM makes a transition between states for each observation from the given sequence.

C. Viterbi algorithm

Let's fix the observation sequence as *Obs*, where the length of *Obs* is *lo*. The Viterbi algorithm [11] handles sequence *Obs* for a HMM. Its result is a maximum probability to reach each state of the HMM after handling *Obs*.

First of all, HMM probabilities are transformed into negative binary logarithm. We define such probabilities as *transformed* probabilities $t(p)$, where p is a some probability from the HMM definition.

$$t(p) = \begin{cases} p > 0: & -1 * log_2(p)) \\ p = 0: & +\infty \end{cases} \tag{1}$$

e.g. probability 0.5 will be expressed as $-1 * log_2(0.5) = 1$. It is done to reduce the loss of precision.

The key idea is to use a special algebraic structure, named semiring *Min_plus*. Elements of this semiring are floats. We define the addition's semantic as a minimum between two floats. The multiplication symbol means addition for floats. Neutral elements are $+\infty$ and 0 accordingly. This is an example of usage *Min_plus* semiring for matrix multiplication:

$$\begin{pmatrix} 0 & 1 \\ +\infty & 2 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} min(0+3, 1+4) \\ min(+\infty+3, 2+4) \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \end{pmatrix}$$

Expression $t_{i,j}$ denotes transformed probability to get observation j at state i, i.e. $t(E_{i,j})$. For each observation j we create diagonal matrix $P(j)$ as follows with data from E:

$$P(j) = \begin{pmatrix} t_{1,j} & \dots & +\infty \\ \vdots & \ddots & \vdots \\ +\infty & \dots & t_{N,j} \end{pmatrix}$$

The initial step of the Viterbi algorithm is to set up data according to the first observation from the observation sequence *Obs*. Symbol $\times$ stands for matrix multiplication using *Min_plus* semiring. Column B defines a probability distribution for states to be a start one.

$$Probs_1 = P(Obs_1) \times B$$

The next step is to handle the rest of the sequence, where t changes from 2 up to *lo*.

$$Probs_t = P(Obs_t) \times T^\top \times Probs_{t-1}$$

As a result, the column $Probs_{lo}$ will contain transformed probabilities to be in a certain state of the HMM if observation sequence *Obs* is handled.

III. Specialized Viterbi algorithm

Here we describe specialization of the Viterbi algorithm in terms of linear algebra. We fix HMM as a static parameter.

To the best of our knowledge, there is no stable partial evaluator maintaining parallel program transformation and providing expected results. In order to achieve specialization effect we've made ad-hoc generating extension, i.e. we provide an effective procedure to perform static data transformation and propagation together with handwritten specialized version of the Viterbi algorithm itself.

A. Theory

The goal is to embed data from the given HMM into the program and simplify expressions. These static data are S, O, B, T, and E. Given a fixed HMM, for all possible observations $o \in O$ the following matrices and matrix multiplications can be precalculated during the specialization phase according to the given in the previous section the Viterbi algorithm definition:

- $P(o)$,
- $P(o) \times B$, denoted latter as $PB(o)$,
- $P(o) \times T^\top$, denoted latter as $PT(o)$.

We can precalculate these operations and memoize the results for further use by the specialized algorithm. The precalculation procedure pseudocode is shown in Listing 1, function `spec_Viterbi`.

The specialized Viterbi algorithm is shown in Listing 1, function `run_Viterbi`, and works as follows. The initial step can be expressed as

$$Probs_1 = PB(Obs_1).$$

The rest of the sequence *Obs* is handled with multiplication

$$Probs_t = PT(Obs_t) \times Probs_{t-1}. \tag{2}$$

Comparing the specialized version with the initial one, there are fewer matrix multiplication operations in *Min_plus* semiring. For the first step, it is one matrix assignment instead of multiplication. For the remaining observations, we need to perform only one matrix multiplication against two. Thus, the initial Viterbi algorithm in terms of linear algebra requires to perform $1 + 2 * (lo - 1)$ matrix multiplications, where *lo* is the *Obs* length, while the specialized version requires only $lo - 1$ multiplications but requires additional memory to keep the precalculated matrices.

Since matrix multiplication in semiring *Min_plus* is associative one can handle two observations by

$$\begin{aligned} Probs_t &= PT(Obs_t) \times Probs_{t-1} \\ &= PT(Obs_t) \times (PT(Obs_{t-1}) \times Probs_{t-2}) \\ &= (PT(Obs_t) \times PT(Obs_{t-1})) \times Probs_{t-2} \end{aligned} \quad (3)$$

Since we know all $PT(o)$, we can precalculate these multiplications, i.e. compute $K \times K$ matrices, and use them as needed. This method can be extended to handle more observations at once, e.g. for three observations:

$$Probs_t = PT(Obs_t) \times PT(Obs_{t-1}) \times PT(Obs_{t-2}) \times Probs_{t-3} \quad (4)$$

For three observations there are $K \times K \times K$ evaluated matrices accordingly. We name equation 2 *1-level* specialization since only one observation handling is precalculated. By analogy, we name equations 3 and 4 by the *second* and the *third* specialization *levels*, and so on. N-level can be computed with PT and $N-1$-level as follows: for all o multiply $PT(o)$ for all matrices at the previous level.

```
HMM
PB[HMM.K]
PT[HMM.K]
level
// obs_lvl_handlers is a mapping from lists that
// contain level observations to a handler matrix
obs_lvl_handlers[HMM.K^level]

function spec_Viterbi()
  for i = 1..HMM.K
    PB[i] = P(HMM.O[i]) × HMM.B)
    PT[i] = P(HMM.O[i]) × (HMM.T)^T
  // To handle a sequence of length level,
  // appropriate level matrices from PT
  // should be multiplied.
  // Put handlers for all possible sequences of
  // length level into obs_lvl_handlers.
  get_combinations(obs_lvl_handlers, level, PT)

function Viterbi(Obs)
  // First observation handling
  Probs = PB[Obs[1]]

  lo = length(Obs)
  i = 2

  // While there is more observations than level
  while (lo - i) >= level)
    // Find matrix to handle next level observations
    handler = obs_lvl_handlers.find(Obs[i:i+lvl])
    Probs = handler × Probs
    i = i + level

  // The rest of the sequence
  for (; i < lo; i = i + 1)
    Probs = PT[Obs[i]] × Probs

  return Probs
```

Listing 1: The specialized with levels Viterbi algorithm

If the specialization with N-level is applied, $(lo-1)/N + (lo-1) \bmod N$ matrix multiplications are required to perform the partially evaluated Viterbi algorithm. Memory consumption rapidly increases with higher N, since K^N precalculated matrices have to be saved in memory.

B. Some implementation details

To perform matrix operations, we used SUITESPARSE:GRAPHBLAS [5]. It is a high-performance implementation of the GRAPHBLAS [13] standard, which is intended to handle graphs, e.g. hidden Markov model. Also, it defines various linear algebra primitives, such as *Min_plus* semiring. Our implementation uses custom formats to define HMM and observation sequences simplifying data parsing. The full source code is available online [14].

IV. EVALUATION

In this section, we compare the specialized Viterbi algorithm against the initial one and CUDAMPF [15].

CUDAMPF is a GPU implementation of the Viterbi algorithm. Since the Viterbi algorithm can be effectively paralleled, a GPU is a suitable choice. CUDAMPF works with hidden Markov models from bioinformatics. A model describes protein family. An observation sequence specifies a protein and contains amino acids. If a probability to be in some special state of the HMM is higher than a threshold, than protein belongs to the protein family.

We took 24 HMMs from CUDAMPF repository[1]. All HMMs have a different number of states but the same structure. Since these HMMs have a slightly different definition, we implement a converter into our custom format. We evaluate our solution on three different datasets. Two datasets are randomly generated, each contains three sequences consisting of 3500 and 7000 observations respectively. The third dataset is real-world 16 proteins taken from PFAM [16] database. The length of the proteins varies from 38 to 7096 observations. The number of possible observations, i.e. $K = 20$, for all datasets.

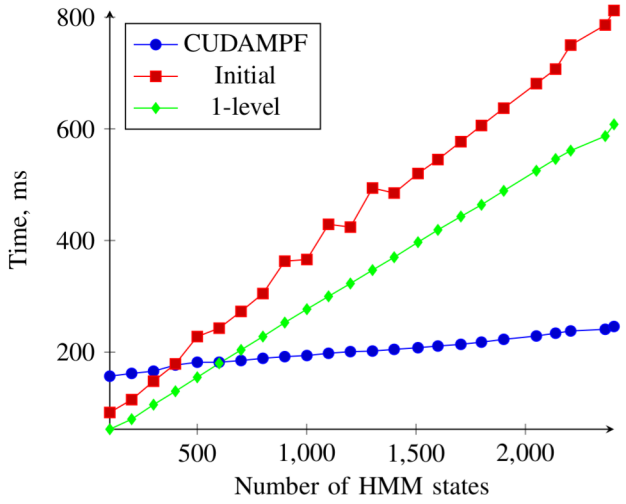
We run experiments on Ubuntu 20.04, Intel Core i7-6700 3.40 GHz, 64 Gb RAM, NVIDIA GeForce GTX 1070. Each implementation with concrete parameters was run 10 times, and a median was taken as a result. We evaluate only the first and second level specialization of the presented algorithm, since memory used for memoization grows by an exponent. For the third level the out-of-memory exception was thrown.

	CUDAMPF	Initial	1-level	2-level
3 x 3500	4854	10765	8062	215329
3 x 7000	9209	21062	16152	387464
Real-world	8796	15864	12036	298269

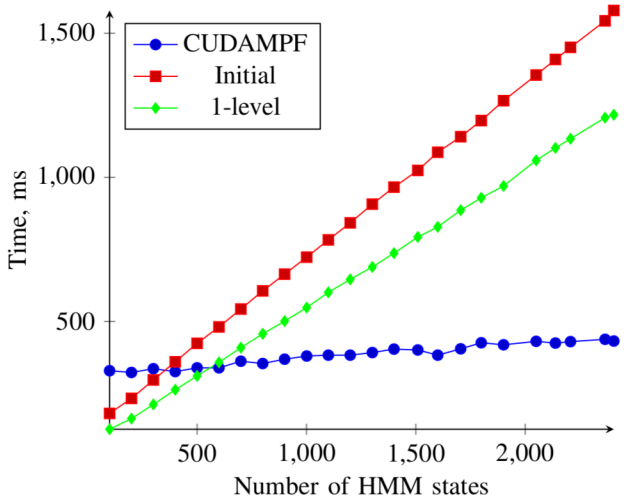
TABLE I: Result run time (both specialization and the Viterbi algorithm), ms

The results (see Figures 1a, 1b, 1c and Table I) show that the first level specialized version of the Viterbi algorithm, as expected, is faster, than the initial one. Unexpectedly, the second level implementation is significantly slower comparing to the initial and the first level implementations, and it is

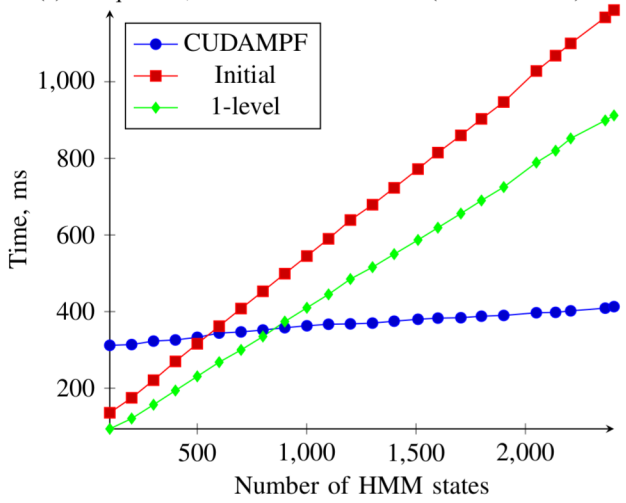
[1]https://github.com/Super-Hippo/CUDAMPF (date: 2021-12-02)



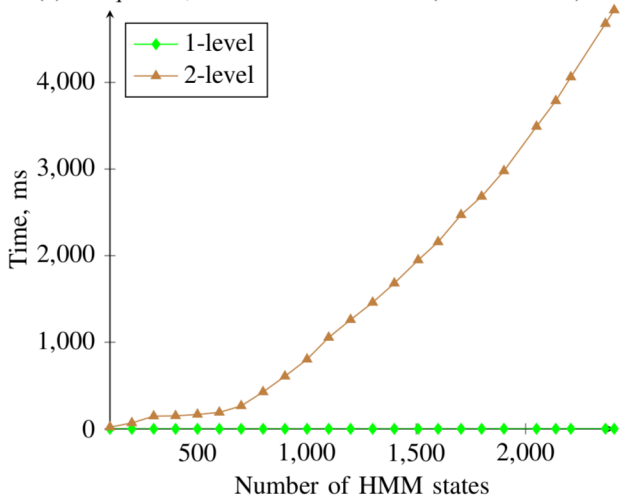
(a) 3 sequences, each 3500 observations (lower is better)



(b) 3 sequences, each 7000 observations (lower is better)



(c) Real-world dataset (lower is better)



(d) Specialization algorithm run time (lower is better)

Fig. 1: Evaluation results

not shown in some figures. One of the reasons for such a slowdown is the increased memory consumption. Nevertheless, even on a workstation with 8 CPU threads, CPU-based first level specialization outperforms GPU-based CUDAMPF on the small HMMs used in CUDAMPF benchmarks[2]. After all, these results are comparable with ones from CUDAMPF. It is also worth noting, that the parallel Viterbi algorithm, expressed in terms of linear algebra, is easier to implement than the CUDAMPF dynamic programming version, since the abstraction level is higher. All the above proves the specialization of the Viterbi algorithm is applicable in practice.

V. Related work

There are a lot of works, where specialization was successfully applied.

[2]For evaluation we used the exact set of HMMs from CUDAMPF benchmarks.

In [17] it was shown that the result of specialization of the naïve pattern match algorithm to some fixed pattern can be behavioural equivalent to the Knuth, Morris and Pratt algorithm. This result is often used as a strength test for partial evaluators to be "good enough". Since we consider the concrete algorithm specialziation only, the test is not applicable in our case.

A partial evaluation was used in ray tracing [18] by P. H. Andersen. The author optimizes a ray tracer, gaining speedups from 1.8 to 3.0 depending on the meta-parameters and compiler. The main performance improvement was reached with constant propagation and unrolling loops. It can be done by an optimizing compiler, but this partial evaluator is aggressive. That means sometimes a specialized algorithm can have an enormous code size and lead to performance regression. The static data was directly written inside source code, while our solution can run without any files' modifications.

A. Tyurin, D. Berezun and S. Grigorev have applied specialization to the naïve pattern match string search algorithm, implemented as a GPU program [19]. They got performance improvement up 8 times in some cases. GPU has a lot of simple algebraic logic units. All of them need to take data to work with. It means a data cache of GPU is a bottleneck. Using specialization, static data was moved to a code cache. Such transformation makes data cache miss less possible. One may call it a "hardware specialization".

C. Sakama et al. used linear algebra as a logic programs representation [4]. The authors introduce partial evaluation as a part of the algorithm to find a logic model of a program. If specialization is used, run time is decreased by 10 times.

VI. Discussion

There are some possible research directions and future work. The Viterbi algorithm specialization is the first step to find out if partial evaluation can be effectively applied to the linear algebra algorithms.

First of all, the next step is to run benchmarks at a GPU. SuiteSparse:GraphBLAS [5] is the reference CPU implementation of the GraphBLAS [13] standard. There are some GPU implementations, such as GraphBLAST [20] and GBTL [21], but to our knowledge, they are unstable.

One can try to apply partial evaluation to the other algorithms in terms of linear algebra. These experiments will reveal the limits of the specialization to such algorithms. There is a high chance that such experiments can be successfully used in production.

Since partial evaluation can lead to a performance increase, it can be useful to implement a linear algebra library with specialization primitives. It will let to develop more effective applications with linear algebra algorithms in less time.

Another approach is to do hardware partial evaluation, e.g. to make FPGA, where specialization program with static data will be embedded as a scheme.

VII. Conclusion

In the paper, we study an application of partial evaluation to the linear algebra-based algorithms on a particular example — the Viterbi algorithm with an HMM being fixed, i.e. static parameter. The specialized version of the Viterbi algorithm is presented. Our experiments show that on real benchmarks the presented algorithm can be comparable to the existing GPU-based Viterbi algorithm implementation CUDAMPF. Thus, the proposed approach is applicable in practice and further partial evaluation application to the linear algebra-based algorithms is a promising research direction.

References

[1] C. Aggarwal, *Linear Algebra and Optimization for Machine Learning: A Textbook*, 01 2020.

[2] R. Szeliski, *Computer Vision: Algorithms and Applications*, 1st ed. Berlin, Heidelberg: Springer-Verlag, 2010.

[3] J. Gentle, *Numerical Linear Algebra for Applications in Statistics*. Springer-Verlag, 1998.

[4] C. Sakama, H. Nguyen, T. Sato, and K. Inoue, "Partial evaluation of logic programs in vector spaces," EasyChair Preprint no. 172, EasyChair, 2018.

[5] T. A. Davis, "Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra," vol. 45. URL: https://doi.org/10.1145/3322125

[6] N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, 1993.

[7] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, 1967.

[8] S. R. Eddy, "Accelerated profile hmm searches," *PLoS computational biology*, vol. 7, no. 10, pp. e1 002 195–e1 002 195, 2011.

[9] L. R. Rabiner, *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, p. 267–296.

[10] A. Petropoulos, S. P. Chatzis, and S. Xanthopoulos, "A novel corporate credit rating system based on student's-t hidden markov models," *Expert Systems with Applications*, vol. 53, pp. 87–105, 2016.

[11] E. Theodosis and P. Maragos, "Analysis of the viterbi algorithm using tropical algebra and geometry," in *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, 2018, pp. 1–5.

[12] S. R. Eddy, "What is a hidden Markov model?" *Nature Biotechnology*, vol. 22, pp. 1315–1316. URL: https://doi.org/10.1038/nbt1004-1315

[13] A. Buluç, T. Mattson, S. McMillan, E. J. Moreira, and C. Yang, "Design of the graphblas api for c," *IPDPS Workshops*, pp. 643–652, 2017.

[14] Implementation repository. URL: https://github.com/IvanTyulyandin/Lin_alg_Viterbi (Date: 2021-13-02).

[15] H. Jiang and N. Ganesan, "Cudampf: a multi-tiered parallel framework for accelerating protein sequence search in hmmer on cuda-enabled gpu," *BMC bioinformatics*, 2016.

[16] J. Mistry, S. Chuguransky, L. Williams, M. Qureshi, G. A. Salazar, E. L. L. Sonnhammer, S. C. E. Tosatto, L. Paladin, S. Raj, L. J. Richardson, R. D. Finn, and A. Bateman, "Pfam: The protein families database in 2021," *Nucleic Acids Research*, vol. 49, no. D1, pp. D412–D419, 10 2020.

[17] C. Consel and O. Danvy, "Partial evaluation of pattern matching in strings," *Information Processing Letters*, vol. 30, no. 2, pp. 79–86, 1989.

[18] P. H. Andersen, "Partial evaluation applied to ray tracing," 1996.

[19] A. Tyurin, D. Berezun, and S. Grigorev, "Optimizing gpu programs by partial evaluation," pp. 431–432, 02 2020.

[20] C. Yang, A. Buluc, and J. D. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu," 2020.

[21] Graphblas template library (gbtl), v. 3.0. URL: https://github.com/cmu-sei/gbtl (Date: 2021-12-02).