2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-1016-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/HIPC53243.2021.00017

Faster Parallel Training of Word Embeddings

Eliza Wszola
Department of Computer Science
ETH Zurich
Zurich, Switzerland
eliza.wszola@inf.ethz.com

Martin Jaggi
School of Computer and Communication Sciences
EPFL
Lausanne, Switzerland
martin.jaggi@epfl.ch

Markus Püschel
Department of Computer Science
ETH Zurich
Zurich, Switzerland
pueschel@inf.ethz.ch

***Abstract*—Word embeddings have gained increasing popularity in the recent years due to the Word2vec library and its extension fastText that uses subword information. In this paper, we aim at improving the execution speed of fastText training on homogeneous multi- and manycore CPUs while maintaining accuracy. We present a novel open-source implementation that flexibly incorporates various algorithmic variants including negative sample sharing, batched updates, and a byte-pair encoding-based alternative for subword units. We build these novel variants over a fastText implementation that we carefully optimized for the architecture, memory hierarchy, and parallelism of current manycore CPUs. Our experiments on three languages demonstrate 3–20× speed-up in training time at competitive semantic and syntactic accuracy.**

***Index Terms*—machine learning, natural language processing, parallel computing, performance, word2vec, fasttext**

I. Introduction

Word embeddings are numerical vector representations of words that capture contextual relationships between words through vector operations. They have become a standard input representation and a common first step in pipelines for the majority of natural language processing (NLP) tasks, benefiting, e.g., classification [1], [2] or machine translation [3], [4]. Word embeddings have a long history [5]–[7], but have received much attention in recent years due to their efficient computation with word2vec [8].

More recently, state-of-the-art results on many language understanding tasks were achieved by deep transformer architectures such as BERT [9], which however are very compute intensive both at training and inference time, even with pretrained models and reduced parameter space. Thus, simpler and more lightweight static word embeddings such as fastText [10] are still widely used, due to their fast execution (both training and inference are an order of magnitude faster than BERT [11]), comparable or better results for particular tasks [12], [13], and the ability to produce a single vector per word, which helps in information retrieval with interpretability and search index construction [13].

We make the following contributions. We present algorithmic and code optimization techniques to improve the training time for word2vec and fastText embeddings on modern general-purpose multicore and manycore computers. We present polyalgorithmic open-source implementations of word2vec and fastText that are carefully optimized for the architecture, memory hierarchy, and parallelism of current manycore CPUs. The implementation encapsulates a number of algorithmic variants including dynamic hidden layer updates, batched updates, and subword units based on the byte-pair encoding approach. Our extensive evaluation on three languages shows that the best combinations of optimizations speed up training time by about 3–20 times while maintaining the accuracy of selected NLP tasks. We also compare against the state-of-the-art GPU implementation demonstrating an up to 3.75 times higher speedup over corresponding baselines. Additionally, we provide a performance analysis of the most performance-critical parts of word2vec and fastText. Our contribution is thus at the intersection of machine learning and HPC, and demonstrated by the high speedups, while maintaining accuracy, that we obtain over the current state-of-the-art.

II. Learning word embeddings

We provide necessary background on word embeddings and the state-of-the-art word embedding algorithms we aim to accelerate: word2vec and fastText.

Word embeddings. A word embedding is a mapping ϕ from textual representations of words as character strings to numerical vectors in $\mathbb{R}^d$. Typically, $d = 100$ or 300. Computing a useful embedding is an optimization problem which aims to minimize the distance in $\mathbb{R}^d$ between words with similar meaning while maintaining larger distances between unrelated words. For example, the words *Norway* and *Sweden* are expected to be close to each other, but distant from the word *panda*. Further, with the popular techniques discussed below relations between words are often captured by algebraic relations between the word vectors. For example, this means that a sentence "A is to B as C is to D" may be expressed as $\phi(B) - \phi(A) + \phi(C) \approx \phi(D)$, where $\approx$ approximates the answer by finding the word D that is closest to the exact solution.

Both training algorithms word2vec and fastText compute a word embedding in an unsupervised way: they take a text corpus as an input, and produce a vector embedding from a vocabulary of size V extracted from the training corpus.

Word2vec. Word2vec is built upon a simple bilinear regression model trained on word co-occurrence. Given a (current) word in a sentence, its objective is to maximize the likelihood of predicting surrounding (context) words. To achieve this, the model is trained to increase the probability of predicting

2640-0316/21/$31.00 ©2021 IEEE
DOI 10.1109/HiPC53243.2021.00017

particular words if they appear close to a given current word in the training text corpus. A popular variant also decreases the probability of predicting words that do not appear close to the current word (negative sampling [8], [14]). Each word w in the vocabulary of size V is represented as a source w_s by one row in the $V \times d$ *input* matrix M_{in} of word embeddings $\phi(w_s)$, and each word is represented as a target w_t by one row in the $V \times d$ *output* matrix M_{out} containing word vectors, that are used to calculate the training objective function (both matrices are indexed by words w). The goal is to maximize the inner products (i.e., minimize the difference) of real pairs of source word embeddings $M_{\text{in}}(w_s)$ with the target word vectors $M_{\text{in}}(w_t)$. With negative sampling, the algorithm also maximizes the difference between the source word embeddings and the target word vectors of words $w_{t'}$ picked randomly out of the source's context.

The algorithm starts by initializing M_{in} randomly and M_{out} to zero. Next, it processes the designated text corpus in a streaming fashion, performing multiple iterations. In each iteration, a current word w_i is processed together with its surrounding context words $\{w_{i-C}, ..., w_{i-1}\}, \{w_{i+1}, ..., w_{i+C}\}$, where C is the range of the context window. There are two modes of operation when training the model corresponding to the following prediction tasks:

- *Skip-gram (SG)*: predict target context words using the current word w_i as the source.
- *CBOW*: predict the target current word w_i using context words as the source.

Each iteration consists of constructing a hidden layer $\mathbf{h}$ from $M_{\text{in}}(w_s)$, and performing a series of loss function updates on one positive and n negative samples in M_{out}. In practice, each such update is a step of stochastic gradient descent (SGD), maximizing the inner product between $M_{\text{in}}(w_s)$ and $M_{\text{out}}(w_t)$, and minimizing the dot products between $M_{\text{in}}(w_s)$ and the negative sample vectors $M_{\text{out}}(w_{t'})$. During the $n+1$ loss function updates, the algorithm calculates a gradient $\mathbf{g}$ which is added to $M_{\text{in}}(w_s)$ at the end of the iteration. In skip-gram, these iterations are performed for each pair (current word, context word) as source and target separately, while in CBOW, all context word embeddings are averaged into $\mathbf{h}$ as source, and each current word is processed as target by a single update.

SGD is performed in parallel with p threads by splitting the training corpus into p parts and processing them asynchronously ("Hogwild" approach [15]). The final embedding of each word is its corresponding row in M_{in}. M_{out} is discarded at the end of training.

Another variant of word2vec replaces negative sampling with the so-called hierarchical softmax. As our experiments are focused on negative sampling, we will not discuss this approach, but only note the compatibility with our work where applicable.

More details follow in the description of fastText.

FastText. Word2vec is not designed to capture the similarities between words based on constituting morphemes (the smallest part of words that carry a meaning), i.e., the words $escalation$ and $escalate$ are close in meaning and share the morpheme $escalat$. The idea of fastText is to learn an embedding that captures this syntactic information by also utilizing subwords shared by different words. This becomes critical in morphologically rich languages such as Russian, or languages rich in word compounds, such as German.

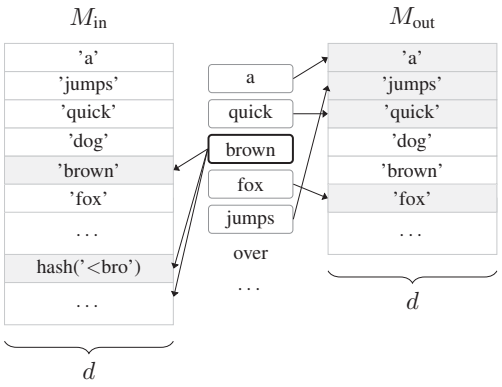


Fig. 1: Representation of source and target words in the input $\boldsymbol{M}_{\text{in}}$ and output $\boldsymbol{M}_{\text{out}}$ matrix in fastText (skip-gram). The words in the corpus are indices of the corresponding rows in $\boldsymbol{M}_{\text{in}}$ and $\boldsymbol{M}_{\text{out}}$.

FastText [10] modifies word2vec by utilizing subwords of source words during the training. A typical run of fastText uses subwords of lengths $k = 3 \ldots 6$, using delimiters $\langle$, $\rangle$ to represent the boundaries. For example, for the word *escalate* and $k = 3$ the subwords are: $\langle$es, esc, sca, ..., ate, te$\rangle$. In this case, the words $escalation$ and $escalate$ share the subwords $\langle$es, esc, ..., lat. FastText creates word representations by averaging their subwords, causing these two word vectors to be averaged from a range of shared subwords and, in consequence, to be close in the vector space.

In fastText, the embeddings M_{in} are extended to contain rows representing both entire words as well as hashes of all their subwords (in particular, two different words can share the same row in M_{in}, either because of their shared subwords, or due to the hash conflicts). Additionally, the representation of the entire word is added to the set of its subwords. The algorithm builds the hidden layer $\mathbf{h}$ by averaging vectors in M_{in} representing the source word's subwords. The final vector embedding for each word is also obtained by averaging these representations. M_{out} remains unchanged. Fig. 1 shows an example of how the word vectors are stored and accessed. A single update is described in Alg. 1. The algorithm can be split into the following high-level steps (σ is the sigmoid function, l is the learning rate of gradient descent):

Lines 1–5 average the source word and corresponding subword vectors $M_{\text{in}}(w_s)$ to obtain the hidden layer $\mathbf{h}$. Zero the gradient $\mathbf{g}$.

Lines 6–14: For the target word w_t (positive sample), compute positive score $\alpha = l(1 - \sigma(\mathbf{h} \cdot M_{\text{out}}(w_t)))$ reflecting similarity between 0 and the row $M_{\text{out}}(w_t)$. Update the target row $M_{\text{out}}(w_t) += \alpha \mathbf{h}$, and build gradient $\mathbf{g} += \alpha M_{\text{out}}(w_t)$ as a backpropagation step. Subsequently, pick n random words

Algorithm 1: A single iteration of the original fastText algorithm. In skip-gram, it is performed on each current-context word pair (as source-target). In CBOW, all context words are used as source words at the same time.

Data: source word(s) w_s, target word w_t, source word subwords $\text{sw}(w_s)$ learning rate l, number of negative samples n

1 **if** *skip-gram* **then** // **Initialize.**
2 $\quad \mathbf{h} = \frac{M_{\text{in}}(w_s)+\sum_{z\in \text{sw}(w_s)} M_{\text{in}}(z)}{\text{count}(\text{sw}(w_s))+1}$ // Average vectors of the source
3 **else if** *CBOW* **then** // word(s) and their subwords
4 $\quad \mathbf{h} = \frac{\sum_s (M_{\text{in}}(w_s)+\sum_{z\in \text{sw}(w_s)} M_{\text{in}}(z))}{\sum_s (\text{count}(\text{sw}(w_s))+1)}$ // to obtain the hidden layer.
5 $\mathbf{g} = 0$ // Reset the gradient.
// **Update the target word.**
6 $\alpha = l(1 - \sigma(\mathbf{h} \cdot M_{\text{out}}(w_t)))$ // Compute positive score reflecting similarity between $\mathbf{h}$ and the row $M_{\text{out}}(w_t)$ representing w_t.
7 $\mathbf{g} = \mathbf{g} + \alpha \cdot M_{\text{out}}(w_t)$ // Build the gradient.
8 $M_{\text{out}}(w_t) = M_{\text{out}}(w_t) + \alpha \cdot \mathbf{h}$ // Update the target word.
9 **for** $t' \leftarrow 1$ **to** n **do** // **Update negative samples.**
10 $\quad$ pick a random $w_{t'} \neq w_t$ // Pick a random negative sample.
11 $\quad \alpha = l(0 - \sigma(\mathbf{h} \cdot M_{\text{out}}(w_{t'})))$ // Compute negative score.
12 $\quad \mathbf{g} = \mathbf{g} + \alpha \cdot M_{\text{out}}(w_{t'}))$ // Build the gradient.
13 $\quad M_{\text{out}}(w_{t'})) = M_{\text{out}}(w_{t'})) + \alpha \cdot \mathbf{h}$ // Update the target word.
14 **end**
15 **if** *skip-gram* **then** // **Update the source rows(s).**
16 $\quad M_{\text{in}}(w_s) = M_{\text{in}}(w_s) + \mathbf{g}$ // The difference between rows in
17 $\quad$ **foreach** $z \in sw(w_s)$ **do** // $M_{\text{in}}(w_s)$ and M_{out} corresponding
18 $\quad\quad M_{\text{in}}(z) = M_{\text{in}}(z) + \mathbf{g}$
19 $\quad$ **end**
20 **else if** *CBOW* **then** // to positive and negative samples
21 $\quad$ **foreach** w_s **do** // drops and increases respectively.
22 $\quad\quad M_{\text{in}}(w_s) = M_{\text{in}}(w_s) + \mathbf{g}$
23 $\quad\quad$ **foreach** $z \in sw(w_s)$ **do**
24 $\quad\quad\quad M_{\text{in}}(z) = M_{\text{in}}(z) + \mathbf{g}$
25 $\quad\quad$ **end**
26 $\quad$ **end**

from outside the context (negative samples) and one by one, perform analogous update while setting each as a target word $w_{t'}$. For negative samples, the score should be negative, thus it is calculated as: $\alpha = l(0 - \sigma(\mathbf{h} \cdot M_{\text{out}}(w_{t'})))$. Update $M_{\text{out}}(w_{t'}))$ and $\mathbf{g}$ accordingly for each negative sample.

Lines 15–26: Update the source row(s) $M_{\text{in}}(w_s) += \mathbf{g}$ corresponding to the source word(s) w_s and their subwords. As a result, the difference between rows in $M_{\text{in}}(w_s)$ and M_{out} which correspond to positive samples is reduced, and the difference between rows in M_{out} which correspond to negative samples is increased.

Related work. FastText has been implemented as a part of the popular Gensim library [16] using Cython and a standard BLAS library (e.g., Intel MKL) for algebraic operations. In our experiments we found the code memory-expensive and slow: training 5 epochs on a 1 GB English Wikipedia dump with 24 threads took approximately 11 hours on a Knights Landing CPU, about 10 times slower than the original fastText when trained without the use of KNL's fast MCDRAM memory. Therefore, we use the original fastText code [17] as the baseline in all our experiments.

For skip-gram with negative sampling, pWord2Vec [18] transforms the "Hogwild" approach into "Hogbatch," by performing updates on multiple context words at once. We employ similar techniques. The work by Rengasamy et al. [19] extends this approach by context combining, where multiple contexts can share a set of negative samples and be updated all at once. We do not adapt this approach as it requires careful preprocessing rewarded by only a relatively small speedup.

Word2vec and fastText have been also implemented for GPU clusters. BlazingText [20] tackles the problem of efficient batch size and synchronization for multiple GPUs and achieves execution time on a single GPU comparable to a 16-threaded CPU fastText baseline. The work by Bae and Yi [21] reports up to 11× speedup of word2vec with negative sampling run on a K20 GPU over the single-threaded CPU word2vec, but only up to 1.6× speedup over a 12-threaded CPU run. We further improve the CPU implementation to obtain superior speedups. Word2vec and fastText are memory-intensive algorithms with fine-grained parallelism limited by the relatively small vectors typically used in the computations. These characteristics severely limit the potential advantages of GPU over CPU. The low GPU performance has also been reported by practitioners [22].

Li et al. [23] discuss a distributed version for many GPUs aiming at the reduction of write conflicts in updates. Similarly (and independently), we made attempts at pre-scheduling a list of current-context word updates, but we found the overhead of this preprocessing prohibitive. Nonetheless, our algorithmic variants could be leveraged in a distributed setting.

Another popular word embedding model is GloVe [24]. While the Authors claim superiority over word2vec, a more thorough evaluation ([25], [26]) shows that there is no clear winner, as the results vary depending on the training corpus, evaluation task, and hyperparameters used. The standard GloVe does not scale well for large vocabularies, and lacks the information on word morphology provided by fastText. The latter can be mitigated by replacing the words in the vocabulary with their BPE tokens [27]. Since GloVe is based on a completely different algorithmic structure (creation and reduction of a global word co-occurrence matrix), there is no direct way to apply our code optimizations and variants.

III. Optimization techniques and algorithmic variants

The goal of this section is to identify the performance-critical parts of the algorithm, and introduce a variety of techniques to speed-up the execution, either by reducing the slowdown incurred by bottlenecks, or by diminishing the amount of computation necessary using algorithmic variants. To improve the training time, we first break down the updates into linear algebraic operations, and then identify the most expensive computations.

Assume that the source word(s) have a total of m subwords (including the entire word) and that we use n negative samples per target word. Additionally, assume that the accumulators of sums and dot products are initialized with zero. By the number

of reads and writes, we denote the number of 4-byte floating-point numbers read by and written to the arrays. Then each update comprises:

- Constructing $\mathbf{h}$: a sum of m rows from M_{in} of length d, followed by a division by a scalar m (line 2 or 4) yielding $d(m+1)$ operations. In the original code, each vector addition is executed separately, followed by a single division, leading to $d(2m+1)$ reads and $d(m+1)$ writes.
- Loss calculation: $n+1$ dot products of $\mathbf{h}$ and the selected rows of M_{out}, each followed by 2 vector scale-add operations on the rows of M_{out}, gradient $\mathbf{g}$, and the hidden vector $\mathbf{h}$ (lines 6–8, 11–13). $6d(n+1)$ flops in total. In the original code, this yields $6d(n+1)$ reads and $2d(n+1)$ writes.
- Gradient update: m vector additions of $\mathbf{g}$ to the relevant rows of M_{in} (lines 16–19 or 21–26), yielding dm operations. In the original code, this involves $2dm$ reads, and dm writes.

We obtain $d(2m+6n+7)$ operations per update, excluding a few scalar operations, $d(3m+6n+6)$ total array reads and $d(2m+2n+2)$ total writes. Thus we have low operational intensity, which means reducing memory movements is the key to speedup. The entire algorithm consists of $S \cdot E$ updates, where S is the size of the corpus (after preprocessing as described in [10]) and E is the number of epochs. To compute the total number of operations in the algorithm, one has to know the average number of subwords which varies for every processed word. By inspecting the vocabularies of selected languages, we observe that the number of subwords in words varies between languages, e. g., for English and the default hyperparameters, most words fit between 7 and 55 subwords. The number of subwords is usually large, therefore we assume that in a typical case, $m \gg n$.

In skip-gram, $\mathbf{h}$ is built only from a single current word vector, while in CBOW, it is constructed from all context word vectors. The loss function, in contrast, is computed once per each current-context word pair in skip-gram, while in CBOW, the loss is computed using the entire context. Empirically, for CBOW, the construction of $\mathbf{h}$ and the gradient update consumes most of the execution time, while for skip-gram, these operations take roughly the same amount of time as the loss function calculation (assuming the default parameter of $n=5$). All operations listed above are memory-intensive and therefore memory bound: thus, the best approach to optimize them is by reducing memory movement and avoiding unnecessary updates. It is further supported by our observations during the tests on isolated sections of the code.

To speed up the training, we first perform a number of code performance optimizations and then various algorithmic modifications compared to the original fastText. Some modifications depend on each other as illustrated in Fig. 2. Some, but not all, techniques apply to both modes of operation. All our improvements build on code_opt which is a CPU-specific optimization of the original fastText code. For skip-gram, we consider a batch variant and negative sharing across the context (NS_CT). For CBOW, we consider keeping track of the values in the hidden layer $\mathbf{h}$ and updating them dynamically rather than building this layer from scratch

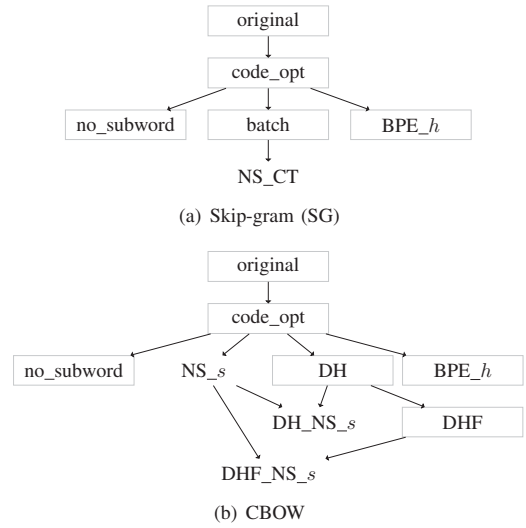


Fig. 2: Dependency between our code and algorithmic variants of skip-gram and CBOW. The experiments for the "NS" variants (no box frames) are not included in this paper due to inferior experimental accuracy.

in each iteration (DH: variable context window size, DHF: fixed context window size). We consider combinations of this technique with negative sharing involving different number s of positive samples that the negative samples are shared between (NS_s). Additionally, for both CBOW and skip-gram, we test no_subword where we remove the subwords from code_opt, making it equivalent to optimized word2vec, and BPE_h, where we replace samples with BPE tokens obtained from a pre-trained token set of size h.

All our variants use the Hogwild-style asynchronous updates and parallelization scheme as the original word2vec and fastText implementations. We describe the effects of asynchronous updates on our optimizations wherever relevant.

We next discuss these variants, referring to Alg. 1.

Code performance optimizations (code_opt). For efficient execution, we explicitly vectorize matrix and vector operations using AVX-512 intrinsics. We employ multiple accumulators in aggregation (sum and dot product) to increases ILP. Additionally, we add zero padding to each matrix row, such that the actual length of each row is divisible by 16. In a 4-byte floating point representation, 16 is both the length of an AVX-512 vector, and the number of elements in a single cache block. This makes it possible to utilize fully the vector parallelism, and align AVX vectors in cache.

Additionally, we block and merge operations involving multiple reads from the same location in memory, like averaging the rows of M_{in} or subsequently reading from and writing to M_{out}. Due to the typically short vector length, the processed data usually remains within the L1 cache. Nevertheless, blocking reduces the number of loads from cache for array accesses.

During the creation of the hidden layer $\mathbf{h}$ (line 2 or 4), we reduce the number of array accesses such that each element

of $\mathbf{h}$ is stored only once during summing up the vector representations of subwords. This improves the original code, which performs a separate store for each subword.

To speed up the loss function computation, we vectorize the dot product (lines 6, 11) with the use of eight accumulators to increase instruction-level parallelism without too much register pressure. We merge the update of the gradient $\mathbf{g}$ and the relevant rows of M_{out} (lines 7–8, 12–13) to avoid multiple reads from the latter. We still call the loss function once for each w_t and $w_{t'}$ separately, as the opposite approach had a negative impact on the convergence.

Similar to the creation of $\mathbf{h}$, we improve the update of $M_{\text{in}}(w_s)$ with $\mathbf{g}$ (line 16 or 21–22) by reading each element of $\mathbf{g}$ only once for all words and subwords w_s.

In summary, we reduce the amount of array reads and writes as follows. During the creation of the hidden layer, we reduce the number of writes to d (including padding). During the loss update, we reduce the number of reads to $5d(n+1)$, and during the gradient update, we reduce the number of reads to $d(m+1)$. This yields $d(2m+5n+6)$ reads and $d(m+2n+3)$ writes. This increases the operational intensity and yields significant speedups when paired with careful vectorization (see Section IV).

The reduction of reads and writes is additionally expected to decrease the number of read and write conflicts on M_{out} in the asynchronous parallel scheme we use, at a cost of negligible delays between reading and storing the data.

The optimizations in the version code_opt are used in all algorithmic variants discussed next. Note that these optimizations can also be applied to other regularization schemes such as the hierarchical softmax used in the original word2vec [8].

No subwords (no_subword). The experiments in Section IV show that it can sometimes be useful to train word embeddings without any subword information. We provide a code variant which disables subwords, but applies all optimizations discussed above. It is algorithmically equivalent to word2vec with negative sampling. In Alg. 1, the set subwords $\text{sw}(w_s)$ in lines 1–4 and 15–26 thus is empty and does not need not be processed. While this is expected to improve the training time, especially for CBOW which dedicated a large part of its runtime to averaging and updating subword representations, the information on the word morphology becomes scarce. We will later see that the word embeddings trained without subword information do not perform well when used for syntactic tasks. On the other hand, the training then focuses on semantic information which is reflected in higher semantic quality of these embeddings. From the complexity point of view, for skip-gram it now always holds that $m = 1$, while for CBOW, m is equal to the number of words in the context.

Minibatching (batch for SG). For skip-gram, we implement a form of minibatching of the target words for each source word. Rather than following pWord2Vec [18], which merges all $M_{\text{in}}(w_s)$ rows in a minibatch into a matrix, we follow the original fastText's approach hitherto only applied to CBOW, which simply averages all these rows. The advantage of our minibatching over the original fastText skip-gram is being able to execute a single update for each context window of the current word w_i, rather than per each current-context word pair. This means that $\mathbf{h}$ and the relevant rows of the input matrix M_{in} are updated only once per each current word, independent of context window size. Lines 2 and 16 are now executed only once per context window, in a similar fashion as in lines 4 and 21–22 respectively. This creates an additional delay between reading and writing a word's subword representations increasing the possibility of write conflicts, but our experiments later show that the accuracy remains nearly unaffected. Minibatching can bring significant speed improvements to subword-based training due to the relatively high cost of building $\mathbf{h}$ and updating all subword representations. As mentioned, in fastText CBOW, this form of batching is already a part of its algorithmic structure. Minibatching reduces the number of operations and data movement in operations requiring reading and writing to M_{in}, i.e., creating $\mathbf{h}$ and updating the relevant rows of M_{in} with the gradient $\mathbf{g}$. The average number of updates per context window in skip-gram is $C+1$, the average of picking randomly from $\{2,\dots,2C\}$, where C is the context window range. Thus the average flop count now is $d(\frac{2m+1}{C+1}+6n+6)$. The number of reads drops to $d(\frac{2m+1}{C+1}+5n+5)$, and the number of writes drops to $d(\frac{m+1}{C+1}+2n+2)$.

Negative sharing (NS_CT and NS_s). We implement negative sharing proposed in pWord2Vec [18], but adapted for and built over SG_batch and the natural batching of fastText CBOW. For skip-gram, we share negative samples among all words in the entire context window of w_i (NS_CT). For CBOW, we share negative samples for s consecutive current words w_i (NS_s). In our implementation, s is a hyperparameter chosen by the user. Thus, line 10 is executed only n times every sth update. Despite improvements in execution time, NS yields inferior accuracy. Therefore, we do not report its results in this paper.

Dynamic hidden layer update (DH). CBOW spends a large portion of its execution time building $\mathbf{h}$ and updating relevant rows of M_{in} for each subsequent current word w_i and its context window. Therefore, we opt for adding and removing subwords only as their words move in and out of the context window as the algorithm processes the training text. After each shift of the context window, we update the rows of M_{in} for all removed subwords, readjust to the gradient $\mathbf{g}$, and add new subwords to $\mathbf{h}$. Thus, rather than performing the entire sum in line 4, the data is processed in five steps. Let u be the expected number of words falling out or into the context window, x the number of embeddings that remain inside the context window after a particular shift, and $\bar{m}$ the average number of subwords in a word. Then the algorithm proceeds as: 1) Denormalize $\mathbf{h}$: multiply $\mathbf{h}$ by the previous number of subwords that it comprises (d flops). 2) Update M_{in} for subwords falling out of the context window ($ud\bar{m}$ flops). 3) Subtract embeddings of subwords falling out of the context window from $\mathbf{h}$ ($ud\bar{m}$ flops). 4) Readjust to gradient $\mathbf{g}$: $\mathbf{h} = \mathbf{h} + x\mathbf{g}$ ($2d$ flops). 5) Add vectors of subwords falling into the context window

to $\mathbf{h}$ ($ud\bar{m}$ flops). 6) Normalize $\mathbf{h}$: divide $\mathbf{h}$ by the new number of subwords (d flops).

Thus, during each update, the algorithm performs $d(3u\bar{m}+4)$ flops, with the proportional number of memory reads and writes ($2ud\bar{m}$ reads and $ud\bar{m}$ writes, discarding lower complexity terms). To improve over *code_opt*, which uses on average $d(2\bar{m}+1)(C+1)$ (C denotes the context window range) operations to perform these updates within the entire context window, it has to hold that $3u < 2(C+1)$.

Given that the sizes of the context windows are picked uniformly from the odd numbers in range $\{3,\ldots,2C+1\}$, where two subsequent picks are independent from one another, we can compute that the expected number of words falling out of the context window in each update is $u = 1+\frac{C^2+2}{3C}$ (excluding line boundaries), which is also equal to the expected number of words falling into the context window. For the default hyperparameter $C = 5$, $u = 2.8$. Thus, we can easily infer that $3u < 2(C+1)$ indeed holds and is expected to yield a speedup.

Note that this approach creates additional delay between reading and writing to the rows of M_{in}, but empirically this does not harm the evaluation score of any of the tasks we present in Section IV.

Fixed window for dynamic hidden layer update (DHF). Since the window size is picked randomly in each iteration, some words will fall in and out of the context window multiple times, forcing DH to remove and add the same subword vectors to $\mathbf{h}$ multiple times over a short period of time. To mitigate this, we fix the window size. While potentially saving time, this approach comes with a pitfall: the variable window size is a natural way of sampling context words that are closer to a current word w_i with greater probability, which reflects a greater contribution of these words to the current word's meaning. DHF effectively ignores the impact of the distance of context words. When using this approach, u is always equal to 2, providing even further speedup.

Byte-Pair vocabulary (BPE_*h*). We also propose an alternative approach to subword embeddings, replacing the subwords by Byte-Pair Encoding (BPE) tokens [28]. These are produced with the Hugging Face Tokenizers library [29] in the form of token IDs for the h most frequent word fragments, where h is a hyperparameter. We expect this to reduce execution time and memory consumption as the number of tokens is typically an order of magnitude smaller than that of subwords. To our knowledge, this is the first attempt to apply BPE tokenization to provide additional subword information in a fastText-like fashion. In our experiments in Section IV, we train the tokenizer over the same training corpus as our embeddings, but both trainings could use different corpora. In case the BPE variant of fastText is unable to tokenize a word found in its training corpus (e.g., because it was absent from the corpus used for training the tokenizer), the word remains as it is, without additional embeddings. An alternative approach would be to create embeddings for tokens consisting of single characters: however, we found that if many words fail to be tokenized, this may cause a drastic slowdown, likely due to update conflicts on the single-character tokens. In Alg. 1, using the BPE variant means replacing "subwords" with "tokens."

IV. Evaluation

Setup. We use a dual-socket Intel(R) Xeon(R) Silver 4114 CPU processor (Skylake-SP, 20 physical cores) in all experiments, except for a paragraph on KNL, where we use a single Intel Xeon Phi 7290 processor (KNL, 72 cores)[1], and a paragraph on AMD, where we use a single AMD EPYC 7742 processor (Zen 2, 64 cores).

For evaluation, we create an English corpus according to the fastText tutorial [30]. For other languages, we proceed in analogous fashion: download respective Wikipedia dumps [31], sanitize and lowercase with the script wikifil.pl authored by [32]. For each language, the script is modified to capture relevant characters and replace relevant words. We truncate the outputs to 1 billion characters. The resulting vocabulary sizes are: (a) 218,316 words for English, (b) 592,674 words for German, (c) 385,596 words for Russian.

The results for English are presented in Table I. The names of algorithmic variants match those from Section III and Fig. 2. We omit negative sharing (NS) due to low accuracy. In tokenized runs (BPE), we use $h = 20\text{K}, 40\text{K}, 200\text{K}$. All other hyperparameters are the fastText defaults. The speedups shown are over the original implementations SG_original and CBOW_original, respectively, from the original fastText [10], run with the same number of threads. The scaling column shows the speedup of our code when run with 20 threads compared to 1 thread. The runtimes are consistent over several runs. We measure performance in flops/cycle by first counting the floating point operations in an average run, and dividing this number by the measured runtime in cycles.

We perform various semantic and syntactic accuracy tests explained below. The best accuracy scores for each test are marked in blue. Pareto-optimal combinations of accuracy scores are shown bold-faced. Pareto-optimal means that no other algorithmic variant dominates it, i.e., is better on each score. Intuitively, the pareto-optimal variants are the best variants for a given task. Together with Fig. 2, the tables also show the incremental impact on accuracy and execution speed of each variant.

Accuracy tests. We perform multiple evaluation tasks to test the quality of our embeddings. First, we test our embeddings with the word analogy task script provided with word2vec [8] for both *semantic* and *syntactic* accuracy. For English, we use the questions-words (QW) dataset [33]. For German, we use its translation [34]. Additionally, we employ the Vecto library [35] to evaluate on the The Bigger Analogy Test Set (BATS) [36] on English embeddings, with the 3CosAdd method. For all analogy benchmarks, we observe that fastText performs better for syntactic than semantic tasks as was already noted [10]. Second, we compute word similarity scores with Facebook MUSE [4], using monolingual evaluation with

[1]Although the Xeon Phi CPUs have been discontinued, we find them a suitable candidate to demonstrate scaling on more cores with a high-bandwidth memory.

TABLE I: Accuracy and speedup achieved with our library over fastText when training on English Wikipedia corpus. Blue: best accuracy in category, bold: Pareto-optimal accuracy, "speedup": over the original fastText run with the same number of threads, "scaling": speedup 20 threads vs. 1 thread for our code. Higher is better for all metrics, except time.

algorithmic variant	accuracy						time (s)		speedup (times)		scaling	performance	
	QW		BATS		MUSE	Battig	1	20	1	20		1	20
	sem.	syn.	sem.	syn.			thread	threads	thread	threads		thread	threads
SG_original	**27.07**	**64.22**	**9.30**	**41.54**	**0.643**	**40.97**	33640	2221	1.0	1.0	15.2	0.6	9.0
Our work:													
SG_code_opt	**26.37**	**63.75**	**9.29**	**41.61**	**0.647**	**40.85**	9087	810	3.7	2.7	11.2	2.2	24.8
SG_no_subword	**47.42**	**48.07**	**12.97**	**27.81**	**0.635**	**41.27**	3309	253	10.2	8.8	13.1	3.1	40.9
SG_batch	**24.28**	**62.47**	**8.92**	**41.84**	**0.630**	**40.76**	7631	625	4.4	3.6	12.2	1.5	18.1
SG_BPE_20K	**40.78**	**53.85**	**12.86**	**34.50**	**0.642**	**40.43**	6843	527	4.9	4.2	13.0	1.6	20.9
SG_BPE_40K	**47.77**	**50.56**	**13.33**	**33.16**	**0.646**	**39.80**	6948	525	4.8	4.2	13.2	1.6	20.8
SG_BPE_200K	**54.98**	**44.79**	**12.65**	**27.73**	**0.634**	**41.46**	7041	526	4.8	4.2	13.4	1.5	20.7
CBOW_original	9.27	67.74	5.53	63.42	0.546	32.98	19614	1484	1.0	1.0	13.2	0.6	8.0
Our work:													
CBOW_code_opt	9.55	68.14	5.69	63.52	0.538	33.07	4285	651	4.6	2.3	6.6	2.8	18.3
CBOW_no_subword	**51.00**	**54.49**	**14.84**	**32.29**	**0.614**	**40.68**	951	73	20.6	20.4	13.1	2.3	29.8
CBOW_DH	**11.38**	**68.73**	**6.51**	**63.58**	**0.583**	**36.30**	4079	555	4.8	2.7	7.4	2.6	19.0
CBOW_DHF	**5.29**	**57.44**	**3.94**	**51.71**	**0.633**	**31.37**	4100	446	4.8	3.3	9.2	2.2	19.9
CBOW_BPE_20K	**22.65**	**46.01**	**10.17**	**36.98**	**0.607**	**35.56**	1717	135	11.4	11.0	12.8	1.6	20.8
CBOW_BPE_40K	30.85	40.47	10.13	33.38	0.614	36.07	1710	133	11.5	11.1	12.8	1.6	20.8
CBOW_BPE_200K	43.33	26.40	8.12	24.70	0.606	40.39	1714	131	11.4	11.3	13.0	1.6	20.7

word similarity tasks on semantic datasets. Third, we use the scripts provided by the Word Embedding Benchmarks package [37] to perform the concept categorization (word clustering) task. We evaluate on the semantic Battig test set [38].

Evaluating English skip-gram. First, we evaluate multiple variants of skip-gram presented in the first section of Tab. I. The dependencies between the variants is illustrated in Fig. 2(a). We observe that only optimizing for efficient execution (code_opt) yields for fastText a 2.7–3.7× speedup while maintaining accuracy. We investigated cache behavior with the Linux *perf* utility and found that a 20-threaded run of code_opt results in 6.7× fewer L1 cache reads than the original. The no_subword variant yields 8–10× speedup over original and about 3× speedup over code_opt. For word analogy, no_subword improves the embeddings semantically. The tokenized versions roughly balance between the fastText- and word2vec-style embedding quality, with an exception of BPE_200K where the number of tokens is close to the vocabulary size, effectively turning only the most common subwords into separate tokens. This approach provides a semantic accuracy even greater than original for both BATS and QW, however at a price of syntactic quality, all including roughly 4–5× speedup over original and up to 1.5× speedup over code_opt. For QW, the accuracies vary greatly, while BATS indicates that a smaller number of tokens is generally preferable. The batch variant maintains or slightly handicaps the accuracy of fastText, and provides a slightly smaller speedup than the tokenized versions. The different variants of skip-gram perform almost equally well on the word similarity and categorization tasks, and all of them yield Pareto-optimal results. All variants show good parallel scaling. A prominent trend is the SG_BPE variants balancing between the semantic and syntactic accuracy of code_opt-based variants and no_subword, which makes all of them pareto-optimal.

Evaluating English CBOW. The CBOW results are shown in the second section of Tab. I; the dependencies between variants are in Fig. 2(b). The code_opt variant yields 2.3–4.6× speedup over original, less than for skip-gram, but the obtained accuracy is not Pareto-optimal. The number of L1 cache reads is 7.8× smaller for a 20-threaded run of code_opt over the original. For word analogy, CBOW generally performs better on syntactic than semantic questions. The no_subword variant provides good scaling, and over 20× speedup over original and about 8–9× speedup over code_opt. It diminishes the discrepancy between these scores, albeit impacting negatively the syntactic quality of the embeddings, while achieving highest scores for word similarity and categorization. None of the tokenized variants was able to beat no_subword both in speed and evaluation on these tasks, but they provide an improvement in semantic accuracy over original, as well as in word similarity and categorization. The BPE variants achieve roughly 11× speedup over original. The DH variant provides only a slight speedup over code_opt (2.7–4.8× speedup over original), but yields higher accuracies in all tasks, while DHF impacts negatively all scores except for MUSE, but provides a speedup over DH with multiple threads.

Comparison between skip-gram and CBOW. As a rule of thumb, the fastText implementations of skip-gram perform much better on semantic questions in word analogy tasks and slightly better in word similarity and categorization tasks. For syntactic questions, the CBOW_code_opt and CBOW_DH variants are a better option. On the other hand, CBOW_no_subword performs nearly as well for word similarity and categorization tasks as skip-gram. Therefore, in specific cases, the former can be used in lieu of skip-gram to boost

the execution speed.

Evaluation on German and Russian corpora. Table II contains results for German and Russian, presented analogously to those in English. In terms of evaluation accuracy, they are largely consistent with English, with the small exception of CBOW_BPE_20K, which performs better than CBOW_no_subword on the German corpus. This indicates the impact of the number of tokens used during training and opens opportunities for further investigation. Noteworthy, CBOW_DH achieves the best scores on syntactic tasks for all evaluated languages. Using skip-gram with BPE tokens rather than fastText-style subwords performs very well in terms of both speedups and accuracy scores, all of which are Pareto-optimal. The code_opt variants yields slightly better speedups than for English, and further optimizations lead to significantly greater speedups. The code_opt variants yield roughly 3.5–5$\times$ speedup over their respective original versions. The best achieved improvement is CBOW_no_subword, up to 50$\times$ for Russian. This shows that our improvements are particularly beneficial for morphologically-rich languages with a large number of subwords per word.

Experiments on synthetic data. We have additionally implemented code snippets imitating the execution of code_opt variants for the most computationally intensive parts of the code and run them on a single CPU core to estimate the achievable performance. Let V be a large number imitating the vocabulary size. We experiment on a $d \times V$ matrix A, and vectors $\mathbf{x}$, $\mathbf{y}$ of length d, containing synthetic data to analyze how different data access patterns affect the overall performance of our algorithm. We operate on warm cache, and average over 100,000 runs.

We consider three computationally heavy subroutines characterized in the beginning of Section III, modified such that:

- Update hidden layer: the division at the end is removed. The subroutine sums m rows of A and stores the result in $\mathbf{x}$. This involves md flops, $4d(m+1)$ bytes in compulsory reads, and $4d$ bytes in compulsory writes.
- Compute loss: all scalar operations are removed. Each of m iterations works with a single row of A_i. In each iteration, we perform a single dot product $r = \mathbf{x} \cdot A_i$, whose result is subsequently used in two vector scale-adds: $\mathbf{y} = \mathbf{y} + rA_i$ and $A_i = A_i + r\mathbf{x}$. This involves $6dm$ flops (the initial value of accumulators is zero), $4 \cdot 3d$ bytes in compulsory reads and $4 \cdot 2d$ bytes in compulsory writes.
- Save gradient: the subroutine adds $\mathbf{y}$ to m rows of A. This involves md flops, $4d(m+1)$ bytes in compulsory reads, and $4dm$ bytes in compulsory writes.

We consider four patterns of accessing the rows of A: (a) *local* operates on the first 8 rows of A. This ensures the data stays in L1 cache. (b) *sequential* operates on the first m rows of A. The rows are accessed directly. (c) *sequential-array* operates on the first m rows of A. The row numbers are accessed indirectly via an array containing values $\{0, \ldots, m-1\}$. (d) *random-array* operates on m random rows of A. The row numbers are accessed indirectly via a random array. This is the most accurate imitation of the data access pattern used by word2vec and fastText.

Figure 3 presents the results of this experiment. We observe that the performance drops significantly in Figs. 3(a), 3(c) when we operate on m rows of data. These are both capped by the throughput of L2 cache which can load **or** store only 64 bits per cycle. The maximum theoretical performance is thus 16 flops (one AVX-512 add) per cycle and every two cycles respectively. We obtain roughly half of these. Additionally, we observe that indirect access via an array in Fig. 3(c) impacts the performance negatively for a small number of subwords (when the data fits in L1 cache), even if the access pattern is sequential. In the same subroutine, we observe that the performance drops even further and becomes less predictable when the access pattern becomes random. In Fig. 3(b) we observe steady performance slightly lower than 16 flops/cycle, even when the data does no longer fit in L1 cache. This is likely because the dot product operates on short vectors, and is finalized by a series of additions and a reduce, which introduce dependencies in the code. In the second part of the subroutine, each loop iteration performs two FMAs and requires one L2 cache store and one L2 cache load. This means that one AVX-512 FMA (32 flops) can be performed every cycle.

We observe that the isolated performances are higher than those obtained in Tab I. This can be partly due to the actual algorithms spending a portion of time managing integer-based auxiliary data structures, such as arrays with ngram indices.

Execution on large data. To show scalability for increasing data size, we run the code_opt variants on a full English Wikipedia corpus which is about 39.6 times larger than the English corpus we used in the previous experiments. We observe that with 20 threads, the models take 41.7 times (Skip-Gram) and 39.0 times (CBOW) longer to train compared to their 20-threaded counterparts in Table I, which shows excellent scalability with data size.

Execution time on KNL. To demonstrate further scaling on manycore CPUs, we repeat the above trainings on the English Wikipedia corpus on a 72-core Intel Knight's Landing processor. We use the `numactl` command to use KNL's fast MCDRAM memory as preferred over regular DRAM. The results in Table III show that we obtain consistent speedups over the original fastText, as well as excellent scaling, up to 63$\times$, the latter largely due to high-bandwidth memory reducing the runtime of memory-intensive code.

Execution time on AMD EPYC.

In a final experiment, we measure execution times on a 64-core AMD EPYC processor. Note that as of now, no AMD microarchitecture supports 512-bit AVX-512 vector instructions, so we rewrote the code for the 256-bit AVX and AVX2, but without AVX-specific optimizations due to time constraints. Table IV illustrates our findings. Interestingly, for 64-threaded runs, we observe that scaling on AMD is far inferior to Intel, even with the original fastText code. Our scaling becomes even worse, providing little advantage over the original variants with all cores used. We attempt to investigate this issue by adding to our measurements 20- and 32-threaded runs, where we observe that due to scalability issues, our speedup over the

TABLE II: Accuracy and speedup achieved with our library over fastText when training on (a) German and (b) Russian Wikipedia corpora. Blue: best accuracy in category, bold: Pareto-optimal accuracy, "speedup": over the original fastText run with the same number of threads, "scaling": speedup 20 threads vs. 1 thread for our code. Higher is better for all metrics, except time.

algorithmic variant	accuracy			time (s)		speedup (times)		scaling
	QW		MUSE	1	20	1	20	
	sem.	syn.		thread	threads	thread	threads	
SG_original	**21.13**	**49.94**	**0.587**	51747	6095	1.0	1.0	8.5
Our work:								
SG_code_opt	19.41	49.12	0.575	14599	1233	3.5	4.9	11.8
SG_no_subword	**42.31**	**27.29**	**0.589**	4511	325	11.5	18.8	13.9
SG_batch	17.43	49.13	0.573	11249	870	4.6	7.0	12.9
SG_BPE_20K	**29.91**	**36.83**	**0.589**	9440	682	5.0	8.94	13.9
SG_BPE_40K	**34.55**	**29.96**	**0.593**	9908	678	5.2	9.0	14.6
SG_BPE_200K	**45.90**	**23.39**	**0.593**	9830	675	5.3	9.0	14.6
CBOW_original	4.71	58.94	0.507	31674	3837	1.0	1.0	8.3
Our work:								
CBOW_code_opt	4.30	59.09	0.510	7452	1114	4.3	3.4	6.7
CBOW_no_subword	**37.75**	**27.66**	**0.559**	1315	92	24.1	41.8	14.3
CBOW_DH	**5.96**	**59.81**	**0.527**	6406	911	4.9	4.2	7.0
CBOW_DHF	**1.86**	**54.17**	**0.591**	6117	845	5.2	4.5	7.2
CBOW_BPE_20K	11.19	31.31	0.542	2413	182	13.1	21.1	13.3
CBOW_BPE_40K	18.67	23.25	0.557	2389	174	13.3	22.0	13.7
CBOW_BPE_200K	27.81	16.14	0.560	2461	170	12.9	22.6	14.5

(a) German

algorithmic variant	accuracy			time (s)		speedup (times)		scaling
	QW		MUSE	1	20	1	20	
	sem.	syn.		thread	threads	thread	threads	
SG_original	**12.29**	**77.61**	**0.633**	30219	2959	1.0	1.0	10.2
Our work:								
SG_code_opt	**12.86**	**77.81**	**0.622**	7409	710	4.1	4.2	10.4
SG_no_subword	**23.82**	**42.89**	**0.588**	2501	179	12.1	16.5	14.0
SG_batch	10.58	76.32	0.629	5914	565	5.1	5.2	10.5
SG_BPE_20K	**14.22**	**51.86**	**0.621**	5179	376	5.8	7.9	13.8
SG_BPE_40K	**16.75**	**46.81**	**0.608**	5212	374	5.8	7.9	13.9
SG_BPE_200K	**22.90**	**45.91**	**0.600**	5230	370	5.8	8.0	14.1
CBOW_original	8.88	80.78	0.495	20510	2599	1.0	1.0	7.9
Our work:								
CBOW_code_opt	**9.18**	**79.79**	**0.497**	4541	569	4.5	4.6	8.0
CBOW_no_subword	16.01	37.35	0.532	731	51	28.1	50.5	14.2
CBOW_DH	**9.17**	**81.43**	**0.523**	4790	455	4.3	5.7	10.5
CBOW_DHF	**7.32**	**78.75**	**0.557**	4451	418	4.6	6.2	10.6
CBOW_BPE_20K	8.28	34.82	0.513	1371	103	15.0	25.2	13.3
CBOW_BPE_40K	8.96	40.66	0.493	1349	100	15.2	26.1	13.6
CBOW_BPE_200K	11.13	35.96	0.537	1343	96	15.3	27.0	14.0

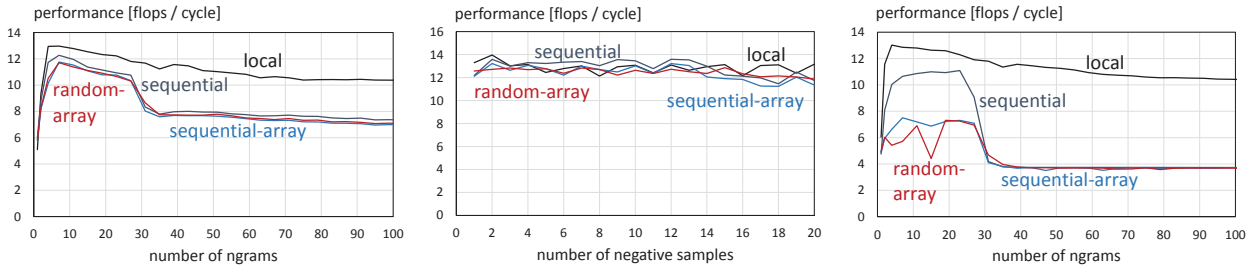
(b) Russian

original variants becomes gradually throttled as more threads are involved in the execution. These findings require further investigation and show that, as common in HPC, performance porting across micro-architectures requires a certain tuning effort.

Comparison against GPU speedup. We compare the speedup of our no_subword code over the original Word2vec library against the speedup obtained on GPU [21]. The Authors report up to 1.6× speedup over a 12-threaded CPU run. Our no_subword versions of the code are roughly 5 (skip-gram) and 6 (CBOW) times faster than the 20-threaded runs of the original word2vec. Thus, our speedups over the original word2vec implementation are 3.25–3.75× greater than those reported for a single GPU. Furthermore, our 20-threaded code_opt versions are 2.3–2.7× faster than the original fastText, making the speed comparison superior to a single-GPU BlazingText [20] run, which is comparable to a 16-threaded fastText run on CPU. While these are high-level comparisons, they indicate that our optimizations achieve greater benefits than that of running these algorithms on GPU.

V. Conclusions

We presented a thorough evaluation, and associated open-source implementation, of various optimization techniques



(a) Update hidden layer (b) Compute loss function (c) Save gradient

Fig. 3: The performance of parts of the algorithm run on different data with different access schemes for $d = 304$. $m =$ subwords.

TABLE III: Speedup achieved with our library over fastText when training on English Wikipedia corpus on a 72-core Intel Knights Landing processor.

algorithmic variant	time (s)		speedup (times)		scaling
	1 thread	72 threads	1 thread	72 threads	
SG_original	100400	1542	1.0	1.0	65.1
Our work:					
SG_code_opt	21768	430	4.6	3.6	50.7
SG_no_subword	14637	231	6.9	6.7	63.3
SG_batch	14899	307	6.7	5.0	48.6
SG_BPE_20K	15131	243	6.6	6.4	62.3
SG_BPE_40K	15087	243	6.7	6.3	62.0
SG_BPE_200K	15045	240	6.7	6.4	62.6
CBOW_original	76865	1289	1.0	1.0	59.7
Our work:					
CBOW_code_opt	14191	660	5.4	2.0	21.5
CBOW_no_subword	3750	62	20.5	20.6	60.0
CBOW_DH	12461	460	6.2	2.8	27.1
CBOW_DHF	12184	389	6.3	3.3	31.3
CBOW_BPE_20K	4363	77	17.6	16.8	57.0
CBOW_BPE_40K	4295	75	17.9	17.1	56.9
CBOW_BPE_200K	4247	74	18.1	17.3	57.1

for fastText and word2vec. In particular, these include performance optimizations for vector architectures, locality, and parallelism and, algorithmically, the use of BPE tokens rather than subwords. We highlight our best (Pareto-optimal) results, and their corresponding speedups. For example, for English, our code offers practitioners speedups in the range of 2.7–20.6×, while maintaining a single- or multi-dimensional notion of accuracy. We achieve good parallel scaling, which is expected to bring even more benefits in the future, as the number of cores further increases. The choice of algorithm depends heavily on the accuracy metric: for all languages, there is no universally best variant, which makes a case for our polyalgorithmic implementation and thorough evaluation of trade-offs. Due to the streaming nature of all variants and good scalability with respect to both the number of threads and the dataset size, they are applicable for both small scale computing and HPC/HTC. Our techniques should also apply to sent2vec [39] for sentence embeddings.

References

[1] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext.zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651*, 2016.

[2] J. Deriu, A. Lucchi, V. De Luca, A. Severyn, S. Müller, M. Cieliebak, T. Hofmann, and M. Jaggi, "Leveraging large amounts of weakly supervised data for multi-language sentiment classification," in *Proceedings of the 26th International Conference on World Wide Web*. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2017, pp. 1045–1052. [Online]. Available: https://doi.org/10.1145/3038912.3052611

[3] S. Jansen, "Word and phrase translation with word2vec," *arXiv preprint arXiv:1705.03127*, 2017.

[4] A. Conneau, G. Lample, M. Ranzato, L. Denoyer, and H. Jégou, "Word translation without parallel data," *arXiv preprint arXiv:1710.04087*, 2017.

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. [Online]. Available: https://doi.org/10.1038/323533a0

[6] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[7] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. New York, NY, USA: Association for Computing Machinery, 2008, pp. 160–167.

[8] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," pp. 4171–4186, 2019. [Online]. Available: https://www.aclweb.org/anthology/N19-1423

[10] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017. [Online]. Available: https://www.aclweb.org/anthology/Q17-1010

[11] K. Fiok, W. Karwowski, E. Gutierrez, and M. Reza-Davahli, "Comparing the quality and speed of sentence classification with modern language models," *Applied Sciences*, vol. 10, no. 10, p. 3386, 2020.

[12] H.-C. Tseng, H.-C. Chen, K.-E. Chang, Y.-T. Sung, and B. Chen, "An innovative bert-based readability model," in *International Conference on Innovative Technologies and Learning*. Springer, Cham, 2019, pp. 301–308. [Online]. Available: https://doi.org/10.1007/978-3-030-35343-8_32

[13] P. Gupta and M. Jaggi, "Obtaining better static word embeddings using contextual embedding models," 2021.

[14] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *arXiv preprint arXiv:1402.3722*, 2014.

TABLE IV: Speedup achieved with our library over fastText when training on English Wikipedia corpus on a 64-core AMD EPYC processor. We show multiple threading setups to illustrate scaling.

algorithmic variant	time (s)				speedup (times)				scaling		
	1 thread	20 threads	32 threads	64 threads	1 thread	20 threads	32 threads	64 threads	20 threads	32 threads	64 threads
SG_original	20598	1206	824	582	1.0	1.0	1.0	1.0	17.1	25.0	35.4
Our work:											
SG_code_opt	6802	570	521	476	3.0	2.1	1.6	1.2	11.9	13.1	14.3
SG_no_subword	4655	379	335	304	4.4	3.2	2.5	1.9	12.3	13.9	15.3
SG_batch	5240	470	429	372	3.9	2.6	1.9	1.6	11.2	12.2	14.1
SG_BPE_20K	4700	409	337	302	4.4	2.9	2.4	1.9	11.5	13.9	15.6
SG_BPE_40K	4627	403	356	308	4.5	3.0	2.3	1.9	11.5	13.0	15.0
SG_BPE_200K	4533	432	358	334	4.5	2.8	2.3	1.7	10.5	12.7	13.6
CBOW_original	13602	879	632	553	1.0	1.0	1.0	1.0	15.5	21.5	24.6
Our work:											
CBOW_code_opt	3354	430	389	453	4.1	2.0	1.6	1.2	7.8	8.6	7.4
CBOW_no_subword	1015	84	77	73	13.4	10.5	8.2	7.5	12.1	13.2	13.8
CBOW_DH	3045	333	301	391	4.5	2.6	2.1	1.4	9.1	10.1	7.8
CBOW_DHF	2839	311	287	287	4.8	2.8	2.2	1.9	9.1	9.9	9.9
CBOW_BPE_20K	1116	91	84	105	12.2	9.7	7.6	5.3	12.3	13.4	10.6
CBOW_BPE_40K	1078	95	88	93	12.6	9.2	7.2	6.0	11.3	12.2	11.6
CBOW_BPE_200K	1073	95	90	98	12.7	9.3	7.1	5.7	11.3	12.0	11.0

[15] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems 24*. Curran Associates, Inc., 2011, pp. 693–701. [Online]. Available: http://papers.nips.cc/paper/4390-hogwild-a-lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf

[16] R. Rehurek and P. Sojka, "Gensim—statistical semantics in python," *Retrieved from genism. org*, 2011.

[17] Facebook Research, *facebookresearch/fastText: Library for fast text representation and classification.*, 2016, https://github.com/facebookresearch/fastText.

[18] S. Ji, N. Satish, S. Li, and P. Dubey, "Parallelizing word2vec in multi-core and many-core architectures," *arXiv preprint arXiv:1611.06172*, 2016.

[19] V. Rengasamy, T.-Y. Fu, W.-C. Lee, and K. Madduri, "Optimizing word2vec performance on multicore systems," in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3149704.3149768

[20] S. Gupta and V. Khare, "Blazingtext: Scaling and accelerating word2vec using multiple gpus," in *Proceedings of the Machine Learning on HPC Environments*, 2017, pp. 1–5.

[21] S. Bae and Y. Yi, "Acceleration of word2vec using gpus," in *International Conference on Neural Information Processing*. Springer, 2016, pp. 269–279.

[22] S. Pavlík, *Gensim word2vec on CPU faster than Word2veckeras on GPU (Incubator Student Blog)*, 2016, https://rare-technologies.com/gensim-word2vec-on-cpu-faster-than-word2veckeras-on-gpu-incubator-student-blog.

[23] B. Li, A. Drozd, Y. Guo, T. Liu, S. Matsuoka, and X. Du, "Scaling word2vec on big corpus," *Data Science and Engineering*, vol. 4, no. 2, pp. 157–175, 2019.

[24] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[25] B. Wang, A. Wang, F. Chen, Y. Wang, and C.-C. J. Kuo, "Evaluating word embedding models: Methods and experimental results," *APSIPA*

[26] P. S. Kumar, R. B. Yadav, and S. V. Dhavale, "A comparison of pre-trained word embeddings for sentiment analysis using deep learning," in *International Conference on Innovative Computing and Communications*. Springer, 2020, pp. 525–537.

transactions on signal and information processing, vol. 8, 2019.

[27] B. Heinzerling and M. Strube, "Bpemb: Tokenization-free pre-trained subword embeddings in 275 languages," *arXiv preprint arXiv:1710.02187*, 2017.

[28] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," pp. 1715–1725, 2016. [Online]. Available: https://www.aclweb.org/anthology/P16-1162

[29] A. Moi, *Fast State-of-the-Art Tokenizers optimized for Research and Production*, 2019, https://github.com/huggingface/tokenizers.

[30] Facebook Research, *Word representations*, 2016, https://fasttext.cc/docs/en/unsupervised-tutorial.html.

[31] Wikimedia Foundation, *Wikimedia Downloads*, 2001, https://dumps.wikimedia.org.

[32] M. Mahoney, *About the Test Data*, 2006, https://mattmahoney.net/dc/textdata.html#appendixa.

[33] T. Mikolov, *Questions-Words.TXT*, 2013, https://github.com/nicholas-leonard/word2vec/blob/master/questions-words.txt.

[34] M. Köper, C. Scheible, and S. S. im Walde, "Multilingual reliability and "semantic" structure of continuous word spaces," in *Proceedings of the 11th international conference on computational semantics*. London, UK: Association for Computational Linguistics, 2015, pp. 40–45. [Online]. Available: https://www.aclweb.org/anthology/W15-0105

[35] Vecto, *vecto-ai/vecto: Doing things with embeddings.*, 2018, https://github.com/vecto-ai/vecto.

[36] A. Gladkova, A. Drozd, and S. Matsuoka, "Analogy-based detection of morphological and semantic relations with word embeddings: what works and what doesn't." in *Proceedings of the NAACL Student Research Workshop*, 2016, pp. 8–15.

[37] S. Jastrzebski, *kudkudak/word-embeddings-benchmarks: Package for evaluating word embeddings.*, 2015, https://github.com/kudkudak/word-embeddings-benchmarks.

[38] W. F. Battig and W. E. Montague, "Category norms of verbal items in 56 categories a replication and extension of the connecticut category norms." *Journal of experimental Psychology*, vol. 80, no. 3p2, p. 1, 1969.

[39] M. Pagliardini, P. Gupta, and M. Jaggi, "Unsupervised learning of sentence embeddings using compositional n-gram features," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. New Orleans, Louisiana: Association for Computational Linguistics, 2018, pp. 528–540. [Online]. Available: https://www.aclweb.org/anthology/N18-1049