Efficient and Accurate Word2Vec Implementations in GPU and Shared-Memory Multicore Architectures

Trevor M. Simonton and Gita Alaghband
Computer Science and Engineering Department
University of Colorado Denver
Denver, CO 80204
Emails: {trevor.simonton, gita.alaghband}@ucdenver.edu

***Abstract*—Word2Vec is a popular set of machine learning algorithms that use a neural network to generate dense vector representations of words. These vectors have proven to be useful in a variety of machine learning tasks. In this work, we propose new methods to increase the speed of the Word2Vec skip gram with hierarchical softmax architecture on multi-core shared memory CPU systems, and on modern NVIDIA GPUs with CUDA. We accomplish this on multi-core CPUs by batching training operations to increase thread locality and to reduce accesses to shared memory. We then propose new heterogeneous NVIDIA GPU CUDA implementations of both the skip gram hierarchical softmax and negative sampling techniques that utilize shared memory registers and in-warp shuffle operations for maximized performance. Our GPU skip gram with negative sampling approach produces a higher quality of word vectors than previous GPU implementations, and our flexible skip gram with hierarchical softmax implementation achieves a factor of 10 speedup of the existing methods.**

I. Introduction

Dense, continuous vector representations of words are useful in machine learning models that rely on natural language. Word2Vec was developed to quickly and efficiently generate such word representations [1] [2]. Word2Vec offers a variety of techniques to train a neural network to represent words in continuous vector space. However, the Word2Vec algorithms do not scale onto multiple core CPUs with shared memory. Research has addressed this problem in the skip gram with negative sampling (SGNS) Word2Vec algorithm [3] [4], but the methods developed to improve SGNS have not improved the hierarchical softmax approach, or failed to maintain a high quality of output. In this work we present improved Word2Vec skip gram parallel algorithms that are optimized for multi-core shared memory CPU, and NVIDIA GPU. The ideas explored while developing this code have all been focused on batching word vectors into matrices. Hence the name of the package: **Wombat** – short for **wo**rd **m**atrix **bat**ches.

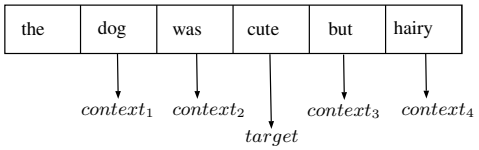
The contributions described in this paper are:

- New CPU implementation of the SGHS architecture optimized for multi-core shared memory machines. This method will perform at 10x the original SGHS speed on Intel machines with the MKL library support, or as much as 4x the original SGHS speed without MKL.
- Original CUDA kernels developed for speed and accuracy of the SGNS approach that offer a great improvement of accuracy over the current state of the art GPU Word2Vec implementation, BIDMach [3]. Our GPU implementation of SGNS is 8x faster than the original Word2Vec, without a loss of accuracy.
- A fast GPU SGHS method, which is the fastest and most flexible GPU implementation of this method that we know of. Our GPU implementation of SGHS is 10x faster than the original Word2Vec implementation, and 4x faster than leading improvements.

The rest of this paper is organized as follows. In Section II we discuss the original Word2Vec implmentation and previously published improvements. In Section III we present our new batched hierarchical softmax method. In Section IV we present our new heterogeneous SGNS and SGHS GPU training approach and describe our CUDA kernel design. In Section V we present the results of our experiments, and we conclude in Section VI.

II. Previous work

Word2Vec is a set of neural network algorithms that train a statistical language model. The neural network uses two weight matrices. The input matrix, W_i, and the output matrix, W_o, are both size $|V| \times w$ where V is the set of words in the vocabulary, and w is a configurable parameter, typically between 100 and 300. The output of the model is a set of $|V|$ vectors that represent words in the vocabulary. This set of vectors is exactly the matrix W_i, where every one of the $|V|$ rows of W_i, each size w, is a word vector. In all configurations of Word2Vec, training of the neural network maximizes the likelihood that target words appear within a window of context words.



For example, the training sentence "the dog was cute but hairy" offers a number of possible context window positions,

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

each yielding a target/context set. One such position set would put "cute" as a target to the set of context words {dog, was, but, hairy}. In this example the context window is of size 2, meaning there are 2 context words on each side of the one target word, "cute." This window can be moved from the start of the sentence to the end to generate a training set for each word in a sentence. The size of the context window is a parameter that can be passed to Word2Vec, typically it is between 5 and 8. Our work will focus on the skip gram Word2Vec architecture, in which the network predicts the set of context words given a target word.

A. Hierarchical Softmax vs. Negative Sampling

Both the *hierarchical softmax* and the *negative sampling* training methods reduce the amount of weight matrix updates that are required during Word2Vec's training procedure. Word2Vec's skip gram with *hierarchical softmax* (SGHS) model is a hierarchical probabilistic language model [5]. These models require the vocabulary to be broken down into a binary tree. In Word2Vec, a Huffman binary tree is used [1]. Leaf nodes in the tree represent words in the vocabulary. A binary code is assigned to every leaf in the tree. The code represents the path taken from root to leaf in a walk down the tree structure. This walk passes a subset of inner tree nodes for each unique leaf. Every one of these inner nodes is represented by a vector in the neural network's output matrix, W_o. As the model is trained, each training step maximizes the probability that given a context word vector from the input matrix, W_i, we walk down the correct path from each node in the tree to the leaf that represents the target word. Each training step updates at most $log_2(V)$ inner tree node vectors, rather than all of the vectors in W_o, as would be necessary in a neural network model that did not use this hierarchical classification approach. An example is presented in Figure 1. The figure illustrates the classification tree for a vocabulary of 4 words. The target words in the tree are labeled t_{0-3}, and the nodes in the tree are represented by vectors W_{o0-o2}. The highlighted nodes are in the walk from the root to the word t_1, which has the code *01*. In this case, the walk to t_1 passes through two nodes, represented by vectors W_{o0} and W_{o1}. Where σ is the sigmoid function, the result of $\sigma(W_{o0} \cdot W_{i1})$ should be close to 0, since this is the proper label for the walk toward t_1, and the result of $\sigma(W_{o1} \cdot W_{i1})$ should be close to 1. Error is calculated with these labels and the matrices are updated accordingly.

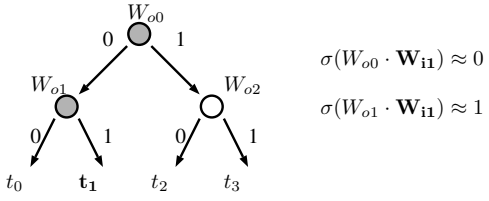


Fig. 1. Illustration of the meaning of nodes in hierarchical softmax.

When using the skip gram with *negative sampling* method (SGNS), the context words are represented by vectors in the input matrix, W_i, and the target words are represented as vectors in the output matrix, W_o. A configurable amount of random words are sampled from W_o each time a context word and target word are observed and used to train the network. For each target or sample word, $\sigma(W_o[target\ or\ sample] \cdot W_i[context])$ represents the probability that the words appear together in a context window. The actual words found in training are labeled with a probability of 1, while the negative samples are labeled with 0.

B. Original parallelization scheme

With all of the original Word2Vec architectures, each individual thread operates independently during the training process. After the vocabulary and network initialization is complete, each thread opens up its own file handle, and trains using a segment of the input file. As the number of threads increases, the number of file handles increases. Each thread then uses context windows from its own file segment to train the shared-memory matrix structures independently. Even though there is no locking or memory protection, there is still a performance problem when sharing the weight matrix structure in multiple cores. For example, some thread, **Thread 1**, might be reading from a vector in memory at the same time that some other thread, **Thread 2**, is writing to that vector. This will cause a false sharing issue in local cache and **Thread 1** will be slowed by cache coherence enforcement. It's this problem that starts to degrade the effectiveness of using multiple threads in the original implementation.

III. New Wombat hierarchical softmax method

We propose a batched training approach to hierarchical softmax in which the set of vectors representing the hierarchy of nodes defining the target word are trained with the set of context word vectors in a single set of matrix operations. The batching algorithm is shown in Algorithm 1. Each thread will be executing these steps in parallel.

Algorithm 1: Wombat's hierarchical matrix to matrix training

```
1  foreach training words do
2      for i ← to context words around target do
3          W_iPrivate[i] ← W_i[context_i];
4      for j ← to k vocabulary tree node vectors do
5          W_oPrivate[j] ← W_o[node_j];
6      G ← W_oPrivate × W_iPrivate^T;
7      calculate error in G;
8      U_i ← G^T × W_oPrivate;
9      U_o ← G × W_iPrivate;
10     update W_i from U_i;
11     update W_o from U_o;
```

The vectors in W_o that represent the nodes in the target word's node set are batched into matrix $W_{oPrivate}$, and the vectors in W_i that represent the individual context words are

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

batched into matrix $W_{iPrivate}$. The work done by activating each individual pair of node vector to context word vector can now be done with a single call to $W_{oPrivate} \times W_{iPrivate}^{T}$. The result of this operation is a $k \times n$ matrix, where k is the number of vocabulary tree nodes that define the target word, and n is the number of context words surrounding the target word in the context window. Error is determined from the difference between the sigmoid of the value and the label of the given row and column in the label matrix. In the case of hierarchical softmax, the label matrix matches the codes of the Huffman binary tree associated with the target word. An example label matrix is exemplified in Figure 2. The subsequent matrix operations used to update the W_i and W_o structures follow a similar pattern to that used in [4].

$$
\begin{array}{c}
 \\
node1 \\
node2 \\
node3 \\
node4 \\
 \\
node(m-1) \\
node(m)
\end{array}
\begin{array}{c}
\begin{array}{ccccc} c1 & c2 & c3 & \ldots & c(n) \end{array} \\
\left[\begin{array}{ccccc}
1 & 1 & 1 & \ldots & 1 \\
0 & 0 & 0 & \ldots & 0 \\
1 & 1 & 1 & \ldots & 1 \\
1 & 1 & 1 & \ldots & 1 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \ldots & 0 \\
1 & 1 & 1 & \ldots & 1
\end{array}\right]
\end{array}
$$

Fig. 2. Label matrix for hierarchical softmax. Example word has Huffman code *1011...01*

IV. Batching Operations for GPU

There are numerous GPU implementations of Word2Vec, but none of them we have found offer a flexible, fast skip gram hierarchical softmax (SGHS). The GPU implementations we have found all focus on either the continuous bag of words method or the negative sampling variant of the skip gram method [3], and/or place limitations on the sizes of word vectors that can be trained [7]. Our work will also focus on the skip gram Word2Vec architecture, but we offer both a hierarchical softmax (SGHS) and negative sampling (SGNS) implementation within this architecture. We also present a new heterogeneous method of batching and training the network for the negative sampling approach that yields higher quality word vectors.

A. Data dependencies

As a context window is dragged from the start of the sentence to the end of the sentence, each individual word will be a target word once, but will be a context word multiple times for different neighboring target words. Proximate positions of the context window yield repeated updates to the same context word vector in W_i. For this reason, the updates cannot happen concurrently without a loss of information.

When batching context window training on the multi-core, shared memory CPU as described in Section III, there is no such overlap. The multi-core CPU implementation has each thread training independently, and each thread's context window is coming from a different sentence. In order to keep all CUDA cores well-fed with training operations, greater parallelism is required. Batching training operations for this level of parallelism requires careful consideration of this data dependency. This problem is also described in [7].

B. Proposed batching method

We propose a heterogeneous method of Word2Vec training. We keep the input and output weight matrices completely in GPU memory. The multi-core CPU parses the input file and generates batches of training operations from context windows found in the training data. The multi-core CPU does this parsing at the same time that the GPU is using its floating point units to process training batches. Each CPU thread works with multiple file parts to fill a pinned memory buffer of diverse target/context word index sets. Each thread works with its own CUDA stream to send data and queue kernel launches when the buffer is filled. The GPU then handles the batched matrix operations and vector updates in the model. Each batch prepared on CPU is composed of integer references to W_i and W_o vectors in the GPU weight structures, and integer labels used for calculating error during training. By transferring a contiguous line of integer values, rather than complete floating point word vectors, we reduce the amount of CPU to GPU data transfer. Each training batch also contains a few other integers that point to boundaries of the training set. An example batch is illustrated in Figure 3.

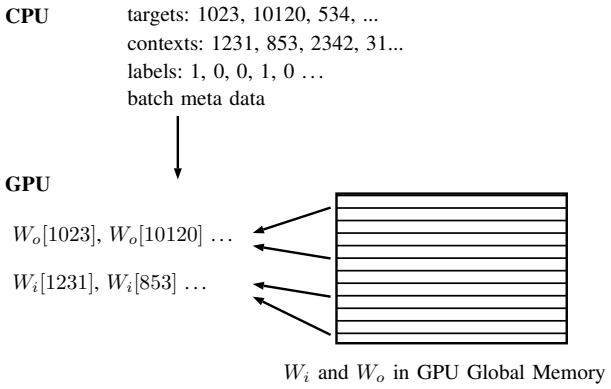


Fig. 3. CPU/GPU batch data transfer

There are $x \times (2c + t + B)$ integers sent to the GPU for each training sentence, where x is the number of words in the sentence, c is the context window size, t is the number of target words in each batch ($(1+k)$ for SGNS, where k is the number of negative samples, and around 15 to 30 tree node vectors for SGHS), and B is the amount of meta data (labels and boundary data) sent in each batch. In [7], only x integers are sent per sentence, and the target/context sets are parsed on the GPU. Our method requires more data transfer to the GPU, but it allows the multi-core CPU to handle parsing of input while the GPU need only handle floating point operations, which are still the bulk of the work in this application. The CPU to GPU memory transfer latency is hidden and overlapped by the amount of time it takes to complete the floating point

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

operations. By having a diverse set of training data in each batch, the issues relating to accuracy are addressed.

C. GPU Kernel Design

We have designed our own GPU kernels to handle target/context set training so we can keep data in GPU block-shared register memory as we work on each sub-operation required of each training step. For both SGHS and SGNS, the work done during training is similar. We re-use the same kernels for both methods.

It's important to note that the matrix sizes are not consistent in each training set. The sentence sizes are not always the same, not all words in the vocabulary tree have the same number of nodes, and there is a random dropping of training words that occurs during the CPU parsing. This complicates efficient use of the GPU streaming multiprocessors. The most efficient implementation should not allow warp divergence, so we have developed a CPU batching method and two GPU kernels that execute the necessary training steps in blocks of exactly 32 threads for each context window. By using 32 threads per block, we can drastically reduce or completely eliminate warp divergence, depending on the size of the word vectors. We then utilize GPU shared memory registers and shuffle operations to reduce interaction with GPU global memory, thus increasing performance.

A single training step of the network involves activation and adjustment of a variable number of target vectors and context vectors that represent words appearing in a context window. In SGNS, the number of target vectors is $1+k$ where k is the number of negative samples. In SGHS, the number of target vectors is 1 to about 40, depending on how long is the longest binary code assignable to a word in the classification tree. The number of context word vectors ranges from 2 to $2c$ where c is the context window size. There is a variable number of context word vectors because Word2Vec randomly drops context words during parsing. Since there are typically at least 4 target vectors, and at least 8 context vectors, we can have the CPU batch training operations into 4×8 sub-operations before sending data to the GPU. These operations can be carried out efficiently in a matrix-based GPU kernel. The remaining sub-operations can be handled with a separate vector-based GPU kernel. The two combined kernels handle the total given number of context windows per batch, which is given to the program as the "batch size" parameter. The step-by-step process is illustrated in Algorithm 2. Each CPU thread executes this sequence in parallel.

1) GPU Matrix kernel: The matrix kernel uses shared memory and shuffle operations to quickly perform the necessary training operations. It is described in Algorithm 3. The block size is 32 threads. Figure 4 illustrates the GPU global memory to block shared memory load operations in Algorithm 3, lines 1 through 4. The threads are used as a 4×8 grid and then as a 8×4 grid. There is no synchronization required during these first two load operations. The next step is a matrix multiplication of the 4 vectors from W_o, and the 8 vectors from W_i. These 32 vectors are all in shared register memory

Algorithm 2: Wombat batching on CPU

```
foreach training words do
    collect context window batch data on CPU;
    split batch into 4 × 8 matrix;
    append sub-matrix operation to matrix batch;
    append remaining sub-operations to vector batch;
    if have given number of context windows in buffer then
        send buffered batches to GPU;
        MatrixKernel<<Number4x8Sets, dim3(8, 4)>>;
        VectorKernel<<NumberVectorSets, 32>>;
```

Algorithm 3: Matrix kernel: Each thread's steps on GPU

```
1  for i to w/8 (as a 4 × 8 grid) do
2      W_oShared ← part of W_o
3  for i to w/4 (as a 8 × 4 grid) do
4      W_iShared ← part of W_i
5  f ← W_oShared[my row] · W_iShared[my col];
6  calculate error in f;
7  for i to w do
8      uO ← f × W_iShared[my col + i];
9      uI ← f × W_oShared[my row + i];
10     shuffle down uI ×2;
11     thread 0 updates Wi;
12     shuffle down uO ×3;
13     thread 0 updates Wo;
```

as $W_{oShared}$ and $W_{iShared}$. Each thread performs a single dot product in w steps. After each thread independently calculates its proper dot-product of vectors in shared register memory, it then applies the sigmoid function and calculates error for its local value given its position in the label matrix. The value remains in thread-local memory.

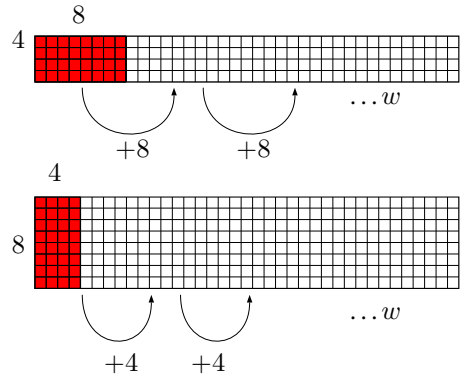


Fig. 4. Loading data to shared memory.

Two matrix multiplications are then used to calculate W_o and W_i updates. These multiplications are shown in Algorithm

3 lines 7 to 13, and is illustrated in Figure 5. The 4×8 block of threads can be thought of as a single matrix, G. Each thread holds a single value in its local memory representing the value of G at its respective row and column. We now perform $G^T \times W_{oShared}$ and $G \times W_{iShared}$. The individual matrix values for this operation are in each thread's local memory. By using a handful of shuffle operations, we can keep the matrix operations within our single warp of 32 threads, thus avoiding the next level of memory hierarchy. We can also do both matrix multiplications at once with a single loop of size w.

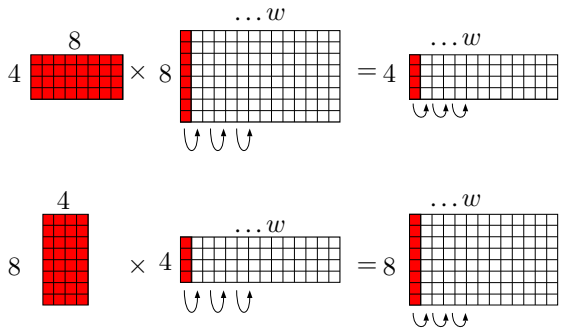


Fig. 5. Wombat update calculation illustration.

2) GPU vector kernel: The vector kernel behaves similarly to the matrix kernel. Each kernel also uses a block of 32 threads. The threads first load two vectors into shared memory from global memory. Each of the threads then works on a subset of elements in each vector to perform element-wise multiplication, and then the threads use a shuffle operation to reduce the individual products down to a single dot product. The single dot product is broadcast to all 32 threads, which then update segments of the global memory vectors in parallel. The vector kernel does not perform as fast as the matrix kernel, but it is necessary to handle the training data that does not fit nicely into 4×8 sub-matrices.

Algorithm 4: Vector kernel: Each thread's steps

1 **for** i **to** $w/32$ **do**
2 $W_{oShared} \leftarrow$ part of W_o;
3 $W_{iShared} \leftarrow$ part of W_i;
4 $f \leftarrow 0$;
5 $x \leftarrow threadIdx.x \times w/32$;
6 **for** i **to** $w/32$ **do**
7 $f \leftarrow f + W_{oShared}[x+i] \cdot W_{iShared}[x+i]$
8 shuffle down f;
9 thread 0 calculate error in f;
10 thread 0 broadcasts f;
11 **for** i **to** $w/32$ **do**
12 update W_i with f and $W_{oShared}$;
13 update W_o with f and $W_{iShared}$;

V. Experiments and Results

The experiments presented below were performed on a single node of the Heracles cluster in our Parallel and Distributed Systems lab (http://PDS.ucdenver.edu) that has 2 Intel Xeon E5-2650v4 Broadwell-EP CPUs with 24 total cores (12 cores per processor). The CPUs run at 2.20 GHz, and the node has 128GB RAM. The operating system is CentOS Linux release 7.2.1511. Code was compiled with the Intel C++ Compiler version 17.0.1 and Intel MKL version 17.0 Update 1 for testing Wombat + MKL, and GCC version 4.8.5 was used to compile and test Wombat without MKL. Also on this node are 4 NVIDIA Tesla P100 16GB "Pascal" SXM2 GPUs. One GPU was used for all Wombat and BIDMach experiments. GPU code was compiled with NVCC V8.0.61. The CUDA driver is version 8.0, with compute capability 6.0. We compare our results to the performance of Gensim [8], as well as to Cythnn and the original C implementation. Gensim and Cythnn both offer SGHS CPU methods. Cythnn also contains a multi-core SGHS Word2Vec implementation that attempts to reduce cache coherence conflicts [6], but the technique proposed does not batch training sets into matrix-to-matrix operations. The BIDMach implementation of Word2Vec claims to have the fastest GPU implementation of SGNS Word2Vec [3].

Accuracy was evaluated for word similarity scores and word analogy scores for baseline comparison with various models. The WS-353 dataset [9] contains a large set of word pairs. Each word pair is given a similarity score that is sourced from human judgement. To evaluate the accuracy of our model, we use the Spearmans rank correlation coefficient [10] between the human judgement and the cosine similarity of our word vectors that represent the words in each pair. High similarity scores represent a model that can detect similarities between words as well as a human. We also used the Google analogy dataset [1] to evaluate the performance of the vectors in answering fill-in-the-blank analogy questions. The score represents the percentage of analogies correctly answered.

The Word2Vec parameters used were as follows: window size was 100, training file is the "text8" dataset from Matt Mahoney's website[1] (this is a truncated and cleaned export of Wikipedia articles that is often used to test the performance of Word2Vec), the downsampling parameter is 1e-4, 10 training iterations are used and the window size is 8. Cythnn was configured to cache the top 31 nodes in the hierarchy. For SGNS tests, 5 negative samples were used. The GPU Wombat program also takes the number of CPU threads as a parameter. The GPU experiments were executed with 8 CPU threads. This increases the rate at which GPU training batches are collected from the training file.

The results show more than a 4x speedup when using Wombat's batched training method without Intel MKL over the original C implementation on all 24 cores. The scalability is linear across the two processors for the non-MKL batched method. At 12 cores, Wombat with MKL is more than 10x faster than the original C implementation. Cythnn and Gensim

[1] http://mattmahoney.net/dc/text8.zip

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

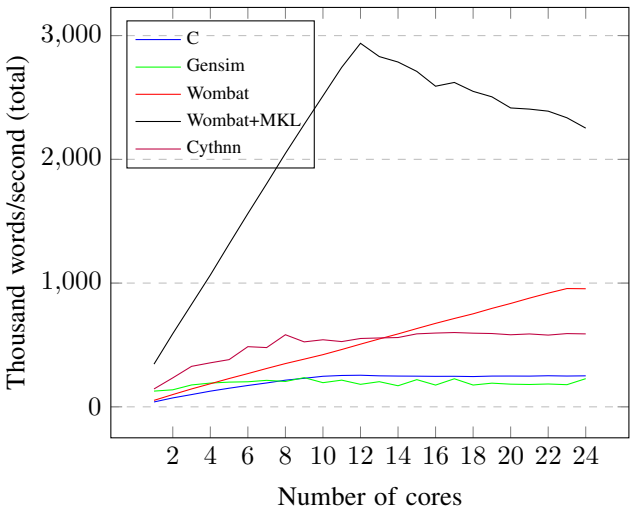


Fig. 6. Speedup for batched hierarchical softmax CPU method compared to leading implementations.

Program	Similarity score	Analogy score
Wombat	68.7	31.5
Cythnn with top-31 cache	67.7	31.1
Original Word2Vec	68.0	34.0

TABLE I
CPU MODEL ACCURACY

do not scale to the second processor. This is likely due to restrictions of Python itself. Although Cythnn outperforms the Wombat with a smaller number of cores, Wombat's MKL-enhanced implementation is much faster in all numbers of cores and processors. There is a dip in Wombat+MKL's performance when the threads start to use the second processor. Each processor has 12 cores, and each set of 12 cores shares an L3 cache. When less than 13 cores are used, all of the work is done in a single processor. When 13 or more cores are used, a second L3 cache and a second processor are introduced, yielding coherence issues between the two processors. These issues are slower to resolve because the two L3 caches reside in separate processors. An increase of threads running in the second processor yields an increase of coherence issues, and an increase of data transfers through main memory. This degrades overall performance. When MKL is absent, this degradation is hidden by the extra time it takes to do the matrix operations. As far as we know, this is the fastest implementation of skip gram hierarchical softmax Word2Vec to date. The increased speed comes at a minimal loss of accuracy. On GPU, batch sizes greater than 512 for Wombat do not increase speed. All experiments for Wombat use a batch size of 512 target/context sets (split into a variable number of matrix and vector batches). The BIDMach batch sizes are not documented. It's not clear what this means, but we discovered that reducing the batch size does seem to have an effect on accuracy and speed. Reducing the batch size to values smaller than 100,000 causes the program to generate errors.

Program	Batch size	Word similarity score	Analogy score	Words/sec
BIDMach SGNS	1,000,000	24	2.9	3.236M
BIDMach SGNS	100,000	33.5	1.6	1.492M
Wombat SGNS	512	68.2	31.9	4.633M
Wombat SG**HS**	512	67.2	32.2	1.942M

TABLE II
EXECUTION TIME AND ACCURACY OF GPU MODELS

Note that the speeds reported here for the BIDMach GPU implementation are much lower for BIDMach than what has been widely reported in other research. In [3] a figure of 8.5 million words/second is reported. This speed is possible with the default learning rate and *vexp* parameters, but the output model returns 0% accuracy in all tests with these parameters. The exact function of the *vexp* parameter is not documented. Only by setting this to 0 and reducing the learning rate does the model actually produce useful vectors. It then produces them at the slower speeds reported in Table II. Also note that BIDMach's words per second reports do not factor in the required preprocessing of the training data that is unique to BIDMach. Wombat does not require any preprocessing. The Wombat GPU implementations for SGNS are similar in speed to BIDMach or faster, and they provide much greater accuracy.

VI. CONCLUSION

In this work we presented new methods for calculating Word2Vec word vectors at speeds faster than previously reported. We applied these techniques to both the CPU and the GPU with success. Increasing thread locality plays a great role in reducing cache coherence conflicts, which in turn provides enormous speedup. This is true on GPU, too. By using thread-local memory and shared memory registers, kernels can more quickly share data within a block than they can by using global memory. Our fast matrix multiply kernel takes advantage of this to provide a high speed of training. Currently we are only splitting out 4×8 sub-matrices for this optimized training approach. The rest of our training steps fall into a slower vector-based kernel. It is possible to further develop the CPU batch preparation process to also create other sub-matrix sizes, such as 8×4, 2×16, 6×6, etc. This could be useful for training batches that don't have enough context words or target words to efficiently divide the work into 4×8 sub-matrices. We leave exploration of these other sub-matrix sizes as a future work. Our GPU skip gram with hierarchical softmax implementation provides speed of calculation that exceeds many skip gram with negative sampling implementations. Unlike other open source GPU versions of this algorithm, ours is does not constrain the input parameters. Our GPU SGNS implementation is as accurate as the original Word2Vec implementation, but comparatively as fast as leading GPU implementations that do not maintain a high level of accuracy. In the future we hope to continue developing Wombat[2] to extend our methods to other Word2Vec CBOW methods.

[2]https://github.com/tmsimont/wombat

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

REFERENCES

[1] Mikolov, T., Chen, K., Corrado, G. and Dean, J., Efficient estimation of word representations in vector space. in *ICLR Workshop,* (Scottsdale, AZ, USA, 2013).

[2] Mikolov, T., Sutskever, K., Chen, K., Corrado, G. and Dean, J. Distributed Representation of Words and Phrases and their Compositionality. *Advances in Neural Information Processing Systems, 26* (2013). 3111-3119.

[3] Canny J., Zhao H., Chen Y., Jaros B. and Mao J., Machine learning at the limit. in *IEEE International Conference on Big Data*, (Santa Clara, CA, USA, 2015), IEEE, 233-242.

[4] Ji, S., Satish, N., Li, S. and Dubey P. Parallelizing Word2Vec in shared and distributed memory. *preprint arXiv:1604.04661v2*, 2016.

[5] Bengio M. and Bengio Y. Hierarchical probabilistic neural network language model. in *Proceedings of the international workshop on artificial intelligence and statistics*, (Barbados, 2005). Society of Artificial Intelligence and Statistics, 246-252.

[6] Vuurens, J., Eickhoff, C. and de Vries, A. Efficient Parallel Learning of Word2Vec. *preprint arXiv:1606.07822*, 2016.

[7] BAE, S. and Yi, Y. Acceleration of Word2vec Using GPUsA. in *Hirose A., Ozawa S., Doya K., Ikeda K., Lee M., Liu D. (eds) Neural Information Processing. ICONIP 2016. Lecture Notes in Computer Science, vol 9948*, (2016), Springer, 269-279.

[8] Řehůřek, R. and Sojka, P., Software Framework for Topic Modelling with Large Corpora. in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, (Vallettta, Malta, 2010), ELRA, 45-50.

[9] Finkelstein, L., Gabrilovich E., Matias Y., Rivlin E., Solan Z., Wolfman G. and Ruppin E., Placing search in context: The concept revisited. in *ACM Transactions on Information Systems, 20* (2002) 116-131.

[10] Spearman, C., The proof and measurement of association between two things. *American Journal of Psychology, 15 (1904).*

978-1-5386-3472-1/17/$31.00 ©2017 IEEE