Single Stream Parallelization of Recurrent Neural Networks for Low Power and Fast Inference

Wonyong Sung  
Seoul National University  
Seoul, Korea  
wysung@snu.ac.kr

Jinhwan Park  
Seoul National University  
Seoul, Korea  
bnoo@snu.ac.kr

ABSTRACT
As neural network algorithms show high performance in many applications, their efficient inference on mobile and embedded systems are of great interests. When a single stream recurrent neural network (RNN) is executed for a personal user in embedded systems, it demands a large amount of DRAM accesses because the network size is usually much bigger than the cache size and the weights of an RNN are used only once at each time step. We overcome this problem by parallelizing the algorithm and executing it multiple time steps at a time. This approach also reduces the power consumption by lowering the number of DRAM accesses. QRNN (Quasi Recurrent Neural Networks) and SRU (Simple Recurrent Unit) based recurrent neural networks are used for implementation. The experiments for SRU showed about 300% and 930% of speed-up when the numbers of multi time steps are 4 and 16, respectively, in an ARM CPU based system.

CCS CONCEPTS
• Computing methodologies → Parallel algorithms; Neural networks;

KEYWORDS
Neural networks, Recurrent neural networks, Parallel algorithm, QRNN, SRU, LSTM

ACM Reference Format:
Wonyong Sung and Jinhwan Park. 2018. Single Stream Parallelization of Recurrent Neural Networks for Low Power and Fast Inference. In Proceedings of ACM SAMOS conference (SAMOS’18). ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnn.nnnnnnn

1 INTRODUCTION
Neural network algorithms show high performance in many machine learning tasks, such as image classification, speech recognition, hand gesture recognition, and machine translation [2, 3, 6, 16–19]. Thus, many future embedded systems, including smartphones and cars, are expected to support many neural network applications. A few different neural network models are used according to the characteristics of the applications. For example, convolutional neural networks are widely used for image recognition [7, 12], while recurrent neural networks are applied to automatic speech recognition (ASR) and machine translation [3–6, 16]. Many neural network algorithms demand a very large number of arithmetic and memory access operations for real-time operation, and also require very large number of parameters, often exceeding one hundred megabytes (MBs).

Since the number of parameters is very large compared to the cache size, the overhead of DRAM access is mostly the bottleneck in real-time inference of neural networks on embedded systems. In sever based implementations, batch processing is widely used for lowering the number of DRAM accesses [20]. Increasing the batch size lowers the number of DRAM accesses for each stream. However, batch processing can hardly be used in many embedded systems because the application is intended for a single user. One example is an on-device ASR on smartphones. On-device ASR can reduce the delay of response and helps keeping privacy. Although, many researches are being conducted to execute neural networks efficiently with special purpose hardware, there are still DRAM access bottlenecks unless all the parameters are stored on on-chip memory [11]. Considering the available on-chip memory size of most embedded devices, it is still unavoidable to store the weights on DRAM. Thus, it is very needed to constrain the number of DRAM accesses for efficient execution of neural network algorithms.

Recurrent neural networks are used for sequence processing, and a sequence consists of multiple time steps. Therefore, the number of DRAM accesses can be reduced if multiple time steps are processed at a time. In server based batch processing, multiple streams are processed in parallel to exploit this characteristic. However, for single stream processing, it is very difficult to parallelize the recurrent neural networks because the feedback incurs the dependency problem. Although signal flow graph analysis allows some partial parallelization even for the LSTM (Long Short Term Memory) RNN, which is most widely used, the effect is limited [9]. In addition, disconnecting parallelism using data characteristics has also been attempted, but it is difficult to apply this technique to general applications [14]. Since this problem is due to the nature of recurrent neural network algorithms, it is difficult to overcome this by employing very smart parallelization or special hardware design techniques.

In this paper, we present a single sequence parallelization of QRNN (Quasi Recurrent Neural Network) and SRU (Simple
Recurrent Unit) based recurrent neural networks, and apply multi time step parallelization for efficient implementation on embedded systems. QRNN and SRU algorithms also contain the feed-back structure, but this part can be isolated and takes a small portion of the total computation. This simple feed-back structure allows multi time step parallelization even for a single stream input, and enables efficient implementations reducing DRAM accesses because one weight fetch from DRAM can be used for multiple time steps. QRNN and SRU algorithms are implemented on Intel and ARM CPU based systems, and the execution performances are compared with those of LSTM RNNs.

This paper is organized as follows. Section 2 explains the application of RNNs and popular RNN models, including LSTM RNN, QRNN, and SRU. Section 3 presents the strategy for multi time step processing of single stream input. Experimental results are shown in Section 4. Concluding remarks follow in Section 5.

2 RECURRENT NEURAL NETWORK ALGORITHMS

A recurrent neural network (RNN) connects the units to form a feedback along a sequence, which allows it to learn dynamic temporal behavior of a time sequence. RNNs can have internal states by either feedback or explicitly using memory cells. This makes them applicable to many sequence recognition tasks. Their applications as well as models that include LSTM, SRU, and QRNN are reviewed in this section.

2.1 Application of recurrent neural networks

RNNs are used for sequence analysis, including text processing, speech recognition, and handwriting recognition. Also, RNN is an indispensable component for sequence generation, such as text generation and foreign language translation. Three major structures of RNN application are shown in Fig. 1. Fig. 1 (a) shows the RNN acceptor that receives a whole sequence of input, and then generates an output at the end of the sequence. An application of the RNN acceptor is the sentiment analysis of the input text, such as movie and restaurant reviews [19]. Fig. 1 (b) shows the RNN transducer that receives the input sequence and generates the corresponding output at each time step. The RNN transducer is used for acoustic modeling, language modeling, and so on. Fig. 1 (c) shows the encoder and decoder model using RNN, where the encoder compresses the input stream and transfers the compressed context to the decoder at the end of the sequence. Thus, the encoder alone is similar to the RNN acceptor. The decoder receives the compressed context as the initial state, and then generates the output as the time step goes. The attention model is an extension of the encoder-decoder architecture [3]. The representative application of the encoder-decoder architecture is the language translation. The original text to translate is applied to the encoder, and the translated text is generated from the decoder. In many applications, bi-directional RNN models are used, where the sequence flow goes in both directions. The bi-directional RNN can be constructed by combining two RNNs operating at different directions.

![Figure 1: Application of recurrent neural networks.](image)

2.2 LSTM RNN

LSTM RNN is most widely used among several RNN models [8], and the block diagram is shown in Fig. 2. Also the equations describing LSTM RNN is in Eq. (1). LSTM RNN contains the memory cell, which can store long term information, and whose value is denoted as $c_t$. The $c_t$ is propagated to the next time step after being multiplied to the forget gate value, $f_t$. The output $h_t$ is generated by using $c_t$ and

![Figure 2: Block diagram of LSTM.](image)
The SRU is recently introduced [13], and the block diagram that can be represented as shown in Eq. (2). Here, we can find LSTM, the SRU has only one dependency relation through multiply-vector operations.

As illustrated in Eq. (1), the major computation of LSTM computation is executed as a sequence of \( h_0, h_1, h_2, \) ... and so on, where \( h_t \) is the output at time step \( t \), because evaluation of \( h_t \) needs \( h_{t-1} \) as illustrated in Eq. 1. Multiple time step processing refers concurrent computation of \( h_0, h_1, h_2, \) ... and \( h_T-1 \), where \( T \) is the block size for parallel processing. In single step processing, the weights, which is usually tens or hundreds of MBs, need to be loaded at each time step, and they are used only once. Unless the cache size is very large to accommodate all the weights, this incurs a very large number of DRAM accesses. In multi time step processing, a weight is fetched from DRAM and used for processing multiple time steps. Usually, we fetch one row of weight matrix, and use it for computing the output for multiple time steps. As a result, the number of DRAM accesses can be reduced as the number of time steps to process increases.

\[
\begin{align*}
    f_t &= \sigma (W_f x_t + U_f h_{t-1} + b_f), \\
    i_t &= \sigma (W_i x_t + U_i h_{t-1} + b_i), \\
    o_t &= \sigma (W_o x_t + U_o h_{t-1} + b_o), \\
    c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \\
    h_t &= o_t \odot \tanh (c_t).
\end{align*}
\] (1)

As illustrated in Eq. (1), the major computation of LSTM network is due to 8 matrix-vector multiplication operations. The matrix values are weights, which are already determined through training, while the vectors are either the input \( x_t \) or the previous output \( h_{t-1} \). As for the matrix-vector multiplication with the input \( x_t \), there is no difficulty in multiple time step processing. But, the matrix-vector multiplication with the previous output \( h_{t-1} \) cannot be conducted employing the multiple time step fashion because of the dependency problem. Thus, even if we precompute the matrix-vector computation with the input vector \( x_t \), the number of DRAM accesses can be reduced just up to a half, when compared with the single time step processing.

\[
\begin{align*}
    \hat{x}_t &= \tanh (W^0 x_t + W^1 x_{t-1}), \\
    f_t &= \sigma (W_f x_t + W^1 x_{t-1}), \\
    o_t &= \sigma (W_o x_t + W^1 x_{t-1}), \\
    c_t &= f_t \odot c_{t-1} + (1 - f_t) \odot \hat{x}_t, \\
    h_t &= o_t \odot \tanh (c_t).
\end{align*}
\] (3)

### 3 MULTIPLE TIME STEP PARALLELIZATION

The SRU is recently introduced [13], and the block diagram that can be represented as shown in Eq. (2). Here, we can find the output gate \( o_t \). Note that the \( f_t \) and \( o_t \) are between 0 and 1, and they are formed by transforming both the input \( x_t \) and the previous output \( h_{t-1} \). Thus, without \( h_{t-1} \), it is not possible to compute \( f_t, i_t, \) and \( o_t \). As a result, the LSTM equation strictly dictates sequential processing. As indicated in (1), it is possible to precompute \( W_f x_t, W_i x_t, \) and \( W_o x_t \) using the multi time processing approach, but the remaining part that depends on \( h_{t-1} \) hinders fully multi time step processing. LSTM RNN approximately demands 8 multiply-vector operations.

\[
\begin{align*}
    f_t &= \sigma (W_f x_t + U_f h_{t-1} + b_f), \\
    i_t &= \sigma (W_i x_t + U_i h_{t-1} + b_i), \\
    o_t &= \sigma (W_o x_t + U_o h_{t-1} + b_o), \\
    c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \\
    h_t &= o_t \odot \tanh (c_t).
\end{align*}
\] (1)

2.3 SRU and QRNN

The SRU is recently introduced [13], and the block diagram is shown in Fig. 3. When the SRU is compared with the LSTM, the SRU has only one dependency relation through \( c_{t-1} \), while LSTM depends both \( c_{t-1} \) and \( h_{t-1} \). The SRU can be represented as shown in Eq. (2). Here, we can find that \( \hat{x}_t, f_t, \) and \( r_t \) are computed without \( h_{t-1} \).

\[
\begin{align*}
    \hat{x}_t &= W x_t, \\
    f_t &= \sigma (W_f x_t + b_f), \\
    r_t &= \sigma (W_r x_t + b_r), \\
    c_t &= f_t \odot c_{t-1} + (1 - f_t) \odot \hat{x}_t, \\
    h_t &= r_t \odot \tanh (c_t) + (1 - r_t) \odot x_t.
\end{align*}
\] (2)

The QRNN equation is shown in Eq. (3) [1]. When compared to the SRU, the gates are computed using the current and past inputs, but not the past output \( h_{t-1} \). The performances of SRU and QRNN have been studied intensively in recent years. Although the results differ according to the applications, their performances are comparable in many cases when their parameter sizes are similar [1, 13].

The implementation of the two RNNs is illustrated in Fig. 3. The SRU computes the output through three matrix-vector multiplications as shown in Eq. (2). Here we can find that the computation of weighted input \( \hat{x}_t \), forget gate \( f_t \), and output gate \( r_t \) can be conducted using only the weight matrices and the input \( x_t \). There is no dependency relation with the output \( h_{t-1} \). In SRU, there is a dependency loop propagating the memory cell value \( c_t \), which is, however, vector element wise operations and demands much less operations compared to matrix-vector multiplications. Also, the vector element wise operation can be conducted up to the
parallelism of the layer width of RNN, which usually have the range of 128 to 1024. Thus, the computation of $c_t$ can be conducted using SIMD or multi-thread operations. The computation of output $h_t$ has no dependency constraints when its input $r_t$ and $c_t$ are all given.

The QRNN equation is shown in Eq. (3). When compared to the SRU, the gates are computed using the current and past inputs, but not the past output $h_{t-1}$. Thus, this structure can also be multi time step parallelized at the same way. When employing the multi time step approach, the forget gate signal, $f_t$, can be computed as shown in Eq. (4). $[f_0, f_1, ..., f_T]$ are computed at a time as Eq. (4) shows. Here, we can use the matrix-matrix multiplication. Note that $f_t$ is a column vector whose size is usually between 128 and 2,048 in many RNN applications. Matrix-matrix multiplication uses the weights several times, thus it demands far less external memory accesses for each arithmetic [15].

$$
[f_0 \ f_1 \ ... \ f_T] = \begin{bmatrix} W_f \\ \end{bmatrix} [x_0 \ x_1 \ ... \ x_T] \tag{4}
$$

The proposed idea is similar to the Manchester carry chain adder shown in Fig. 4. In the Manchester carry chain, the propagation and generation signals are pre-computed only using the input. Then, the carry is propagated very fast using the propagation and generation signals [21].

![Ripple carry adder](image1)

![Manchester carry chain adder](image2)

**Figure 4:** Ripple carry adder and Manchester carry chain adder.

4 EXPERIMENTAL RESULTS

We have measured the execution time of LSTM RNN, SRU, and QRNN models on Intel Core i7-3930K 3.2GHz CPU and Nvidia Denver2 ARMv8 64-bit 2.0GHz CPU. Small and large RNN models are used for the experiments. As for the small model, the LSTM RNN has the cell width of 350, while the SRU has the width of 512, and the number of parameters are approximately 1M. The large model LSTM RNN has the cell width of 700, and that of SRU RNN has the width of 1024, by which the parameter size of both models are comparable, approximately 3M.

The Intel CPU has 32KB L1 data cache and instruction cache, 256KB L2 cache, and 12,288KB L3 cache. The ARM CPU has 32KB L1 data cache, 48KB instruction cache, and 2,048KB L2 cache. The program was written in C++ language, and used the BLAS for optimum execution of matrix-vector and matrix-matrix multiplications. Intel Math Kernel Library (MKL) [10] and OpenBLAS [22] are used for Intel CPU and ARM CPU, respectively. The highest compiler optimization level, 02, is used for this experiments. The time is measured while processing 1,024 input samples.

Table 1 gives the execution time and the speed-up of the small model RNNs on the Intel CPU. The LSTM with single time step and the SRU with multiple time step parallelization are shown. 'SRU-$n$' refers to the SRU execution with n-step parallelization. The speed-up of SRU-$n$ is measured on the basis of the SRU-1. Here, we can find almost 400% of speed-up when 32 step parallelization is employed. A similar result can be found for the large model RNNs. Table 2 shows the execution time and the speed-up of large model RNNs.

| Model     | Execution Time | Speed-up |
|-----------|----------------|----------|
| LSTM      | 673.667        | -        |
| SRU-1     | 475.43         | 100%     |
| SRU-2     | 288.729        | 164.7%   |
| SRU-4     | 197.765        | 240.4%   |
| SRU-8     | 153.39         | 309.9%   |
| SRU-16    | 129.591        | 366.9%   |
| SRU-32    | 118.247        | 402.1%   |
| SRU-64    | 96.302         | 493.7%   |
| SRU-128   | 93.219         | 510.0%   |

| Model     | Execution Time | Speed-up |
|-----------|----------------|----------|
| LSTM      | 2359.94        | -        |
| SRU-1     | 1880.63        | 100%     |
| SRU-2     | 1104.22        | 170.3%   |
| SRU-4     | 715.919        | 262.6%   |
| SRU-8     | 523.264        | 359.4%   |
| SRU-16    | 437.565        | 429.7%   |
| SRU-32    | 375.647        | 500.6%   |
| SRU-64    | 335.64         | 560.3%   |
| SRU-128   | 320.121        | 587.4%   |

**Table 1:** Execution time (msec) of small model RNNs on the Intel CPU for processing 1,024 input samples.

**Table 2:** Execution time (msec) of large model RNNs on the Intel CPU.
Table 3: Execution time (msec) of small model RNNs on the ARM CPU.

| Model | Execution Time | Speed-up |
|-------|----------------|----------|
| LSTM  | 1522.3         | -        |
| SRU-1 | 902.736        | 100%     |
| SRU-2 | 484.474        | 186.3%   |
| SRU-4 | 274.82         | 328.5%   |
| SRU-8 | 172.856        | 522.2%   |
| SRU-16| 108.414        | 328.5%   |
| SRU-32| 85.6596        | 1053.8%  |
| SRU-64| 96.1196        | 939.1%   |
| SRU-128| 93.3887       | 966.6%   |

Table 4: Execution time (msec) of large model RNNs on the ARM CPU.

| Model | Execution Time | Speed-up |
|-------|----------------|----------|
| LSTM  | 4583.75        | -        |
| SRU-1 | 3652.59        | 100%     |
| SRU-2 | 1925.07        | 189.7%   |
| SRU-4 | 1078.03        | 338.8%   |
| SRU-8 | 634.951        | 274.7%   |
| SRU-16| 392.163        | 931.4%   |
| SRU-32| 288.659        | 1265.4%  |
| SRU-64| 275.078        | 1327.8%  |
| SRU-128| 275.658       | 1325.0%  |

Fig. 5 summarizes the speed-up of SRU according to the number of parallelization steps. We can find that the benefit of multiple time step parallelization is bigger in ARM based systems when compared to Intel CPU based computer. This is because the benefit of reduced DRAM access due to multi time step parallelization becomes more prominent when the computer system has a poor memory system, such as low bandwidth DRAM and small cache size. Also, the larger RNN model that needs more parameters shows higher speed-up compared to the small one.

executing the small model on the ARM CPU, the speed-up is nearly 1,000% when the number of parallelization steps is 32. The speed-up of large model RNN is very impressive, more than 1,250%.

The experimental results on ARM CPU are given on Table 3 and 4 for the small and large models, respectively. When
Table 8: Execution time (msec) of large QRNNs on the ARM CPU.

| Model    | Execution Time | Speed-up |
|----------|----------------|----------|
| QRNN-1   | 6467.72        | 100%     |
| QRNN-2   | 3356.7         | 192.6%   |
| QRNN-4   | 1844.29        | 350.6%   |
| QRNN-8   | 1253.13        | 516.1%   |
| QRNN-16  | 712.439        | 907.8%   |
| QRNN-32  | 475.433        | 1360.3%  |
| QRNN-64  | 469.515        | 1377.5%  |
| QRNN-128 | 450.848        | 1434.6%  |

The experimental results for QRNN are shown in Table 5-8. Here, we can find the similar trends. We can find higher speed-up values in the ARM based system. Fig. 6 shows the speed-up of QRNN.

5 CONCLUDING REMARKS

RNN execution demands a large memory bandwidth, especially when only a single stream input data is processed. We have reduced the number of DRAM accesses by parallelizing the RNNs in the time domain. In this approach, one weight fetch is used for execution in multiple time steps, thus the efficiency increases as the number of time steps to parallelize grows. This parallelization technique is applied to simple recurrent units (SRUs) and quasi-RNNs (QRNNs), which are both recently developed models with a simple recurrence structure. The experiments were conducted on Intel CPU and ARM CPU based systems. We achieved a speed-up of more than 500% at the Intel CPU based system, and that of more than 1,250% at the ARM CPU based system. This technique can be utilized for high speed inference of RNNs on VLSI or GPUs (Graphics Processing Units).

ACKNOWLEDGMENTS

This work is supported in part by the Brain Korea 21 Plus Project and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2015R1A2A1A10056051). This work is also supported by Samsung Advanced Institute of Technology.

REFERENCES

[1] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. 2016. Quasi-recurrent neural networks. arXiv preprint arXiv:1611.01576 (2016).
[2] William Chan, Navdeep Jaitly, Quoc Le, and Oriol Vinyals. 2016. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on. IEEE, 4960–4964.
[3] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. Syntax, Semantics and Structure in Statistical Translation (2014), 103.
[4] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulecde, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistic Machine Translation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP). 1724–1734.
[5] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In Acoustics, speech and signal processing (icassp), 2013 IEEE international conference on. IEEE, 6645–6649.
[6] Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Networks 18, 5-6 (2005), 602-610.
[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition. 770–778.
[8] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. Neural computation 9, 8 (1997), 1735–1780.
[9] Kyuyeon Hwang and Won Yong Sung. 2015. Single stream parallelization of generalized LSTM-like RNNs on a GPU. In Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on. IEEE, 1047–1051.
[10] Intel. 2015. Intel Math Kernel Library Developer Reference. URL: https://software.intel.com/en-us/articles/mkl-reference-manual (2015).
[11] Jonghong Kim, Kyuyeon Hwang, and Won Yong Sung. 2014. X1000 real-time phoneme recognition VLSI using feed-forward deep neural networks. In Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on. IEEE, 7510–7514.
[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet classification with deep convolutional neural networks. In Advances in neural information processing systems. 1097–1105.
[13] Tao Lei and Yu Zhang. 2017. Training RNNs as Fast as CNNs. arXiv preprint arXiv:1709.02755 (2017).
[14] Peng Ouyang, Shouyi Yin, and Shaojun Wei. 2017. A fast and power efficient architecture to parallelize LSTM based RNN for cognitive intelligence applications. In Proceedings of the 54th Annual Design Automation Conference 2017. ACM, 63.
[15] Michael J Quinn. 2003. Parallel Programming. TMH CSE 526 (2003).
[16] Hågim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In Fifteenth annual conference of the international speech communication association.
[17] Sungho Shin and Won Yong Sung. 2016. Dynamic hand gesture recognition for wearable devices with low complexity recurrent
neural networks. In *Circuits and Systems (ISCAS)*, 2016 IEEE International Symposium on. IEEE, 2274–2277.

[18] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM neural networks for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association*.

[19] Duyu Tang, Bing Qin, and Ting Liu. 2015. Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the 2015 conference on empirical methods in natural language processing*. 1422–1432.

[20] Volodymyr Turchenko, George Bosilca, Aurelien Bouteiller, and Jack Dongarra. 2013. Efficient parallelization of batch pattern training algorithm on many-core and cluster architectures. In *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2013 IEEE 7th International Conference on*, Vol. 2. IEEE, 692–698.

[21] Neil HE Weste and Kainran Eshraghian. 1994. Principles of CMOS VLSI design: A systems perspective second edition. *Addision-Wesley Publishing, California*, 1994 (1994).

[22] Zhang Xianyi, Wang Qian, and Zaheer Chothia. 2012. OpenBLAS. *URL: http://xianyi.github.io/OpenBLAS* (2012), 88.