Exploiting Model-Level Parallelism in Recurrent Neural Network Accelerators

Lu Peng, Wentao Shi, Jian Zhang, and Samuel Irving
Louisiana State University

ABSTRACT

Recurrent Neural Networks (RNNs) have continued to facilitate rapid progress in a variety of academic and industrial fields, though their complexity continues to make efficient deployment difficult; when the RNN model size is not properly matched to hardware resources, performance can suffer from hardware under-utilization. In this work, we propose to explore model-level parallelism for LSTM-RNN accelerators in different levels of the model using a multi-core design. The multi-core design proposed in this work operates in three computing modes: *multi-programming* mode in which independent models are executed; *multi-threading mode* in which parallelism among layers of an LSTM model is explored and properly scheduled; and *helper-core mode* in which cores collaborate on a single LSTM layer in a lower model level comparing with *multi-thread mode*. Our design can achieve up to 1.98x speedup in "multi-programming" mode, a 1.91x speedup in "multi-threading" mode and a 1.88x speedup in "helper-core" mode over the single-core design.

I. INTRODUCTION

Deep Neural Networks (DNNs) have recently made revolutionary progresses in a variety of academic and industrial fields. Currently there are two dominating models: Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). Convolutional Neural Networks (CNNs) have achieved remarkable performance and broken the limits of previous algorithms for applications such as image recognition and computer vision. Recurrent Neural Networks (RNNs) have demonstrated superior performance in applications with temporal components such as Natural Language Processing (NLP). CNNs and RNNs can be combined to realize applications which involve both image and sequence, such as motional capture and video classification and image captioning.

Along with the exceptional accuracy improvements of the DNN models came a dramatically increased burden on hardware. Computing the output of the convolutional layers of a typical CNN is computationally intensive [1], requiring a relatively large amount of computing resources, while the fully-connected layers of CNNs or RNNs are memory-intensive [1], requiring high memory bandwidth. In consequence, traditional CPU-based platforms are no longer the best choices for deploying these algorithms because they do not provide sufficient parallelism. GPUs can provide improved performance but at the cost of higher power consumption. FPGAs and ASICs have garnered attention due to their application-specific nature, ability to achieve high degrees of parallelism, and high energy efficiency.

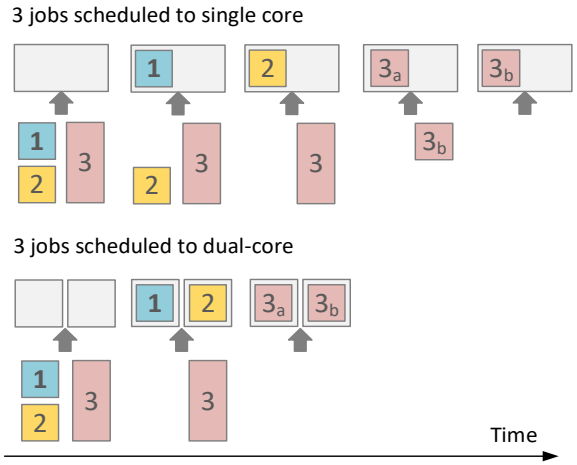

Figure 1. Motivating Example

Significant work has been done designing CNN accelerators for FPGAs and ASICs [2-7]. However, according to Jouppi et al. [8], CNNs comprise only 5% of the workload of Google's data center, while networks that use Long Short-Term Memory (LSTM) make up 29%. There is existing work on LSTM-RNNs accelerators [9-13]. The overall architecture of these work is considered as single-core accelerators because their ALU/DSP arrays cannot be used simultaneously by different jobs.

Single core LSTM-RNN accelerators have a disadvantage: all computing resources are arranged to form a single accelerator core with a large size, leveraging data-level parallelism. However, the accelerator is only able to process one job at a time - leading to potential inefficiency when multiple job requests occur at the same time, especially in large data centers. Secondly, when the size of the

target LSTM model is small, the hardware resources will not be fully utilized. As illustrated in Figure 1, three jobs are scheduled to a single core accelerator. All three jobs are independent jobs with no mutual data dependencies. The single-core accelerator processes the three jobs individually in a sequence over three timesteps. Using a dual-core accelerator, job 1 and job 2 can be processed simultaneously before processing job 3 during the second timestep.

Based on the motivating example mentioned above, we present a multi-core design for the inference process of LSTM-RNNs. This paper presents dual-core design for simplicity purposes. However, it can be extended to system with more than two cores.

This design is made to utilize hardware resources more efficiently and can operate in two kinds of modes: In multi-programming mode, the two cores run freely, processing independent jobs at the same time. In multi-threading mode and helper-core mode, the two cores run cooperatively on a single job in different approaches. Our design methodology can be applied either to generate FPGA designs according to the requirements of specific applications or implemented as an ASIC with fixed design parameters that have been optimized according to the application requirements. Our contributions are listed below:

1. We explore opportunities for LSTM model-level parallelism during the inference process.
2. We propose a dual-core LSTM accelerator design to leverage the parallelism based on existing single-core designs.
3. Optimizations are conducted to improve the performance of the dual-core accelerator under different scenarios.

II. RNN BACKGROUND

RNNs are an evolved version of an Artificial Neural Network (ANN) that possesses the ability to handle temporal information. In RNNs, the output of hidden layers will be cycled back to previous hidden layers as input during the next time step. Thus, connections can be formed as directed cycles between hidden layers, allowing for the detection of dynamic temporal behavior. Using these unique features, RNNs can be trained to learn from past information. However, RNNs have disadvantages limiting their use on real-life applications. According to Hochreiter and Schmidhuber [13], traditional RNNs suffer from gradient vanishing and gradient exploding. LSTM-RNNs address this problem by adding gates to control the amount of information memorized and forgotten by the network. LSTM-RNNs make the training process easier because they converge more quickly, allowing the model to learn from long-term dependences and past information.

Generally Neural Network algorithms contain two phases: training and inference. Training is used to find the best performing parameters (weights and biases) for a given model. After training is concluded, the model can be used to process input and generate output during the inference phase. Training is usually done offline i.e. the models are well-trained before being deployed to various platforms. In this work, we focus on the inference process for LSTM-RNNs. The unfolded inference process of a typical LSTM-RNN is illustrated in Figure 2. This figure shows that the input vector $\mathbf{x}_t$ is fed to the LSTM sequentially, and the outputs $\mathbf{c}_t$ and $\mathbf{h}_t$, generated by the LSTM cells, are fed to the same LSTM cell during the next time step. The LSTM architecture shown in Figure 2 is a 2-layer architecture. Architectures with more layers can be achieved by stacking additional layers on top of this model and using its output as input. The detailed inference process of LSTM-RNNs can be found in [13].

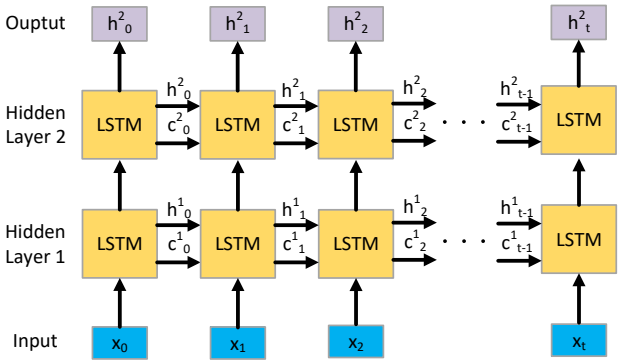

Figure 2. A Two-layer LSTM Model

III. OPPORTUNITIES FOR PARALLELISM

One intuitive parallelism can be achieved by letting two independent LSTM models run simultaneously in a system. Of course, the models can be the same such that parallelism for a particular model is improved. This kind of parallelism is not limited by the data dependencies inside an LSTM-RNN model since the granularity of parallelism is outside the LSTM-RNN models.

Batch parallelism is achieved by feeding multiple inputs into a Neural Network to generate multiple outputs in one inference round. Batching allows the parameters of the Neural Network model to be shared by multiple inputs such that parallelism among the inputs is realized. One benefit of batch processing is that the computation to communication ratio of the system can be improved, reducing the memory bandwidth burden for the system, potentially improve system throughput.

The two techniques mentioned above can be generalized to other types of models since they explore parallelism outside of the models. Inside-model-level parallelism is also feasible. For example, parallelism among the layers and logical time steps of the LSTM model can be explored. When exploring the parallelism opportunities inside LSTM

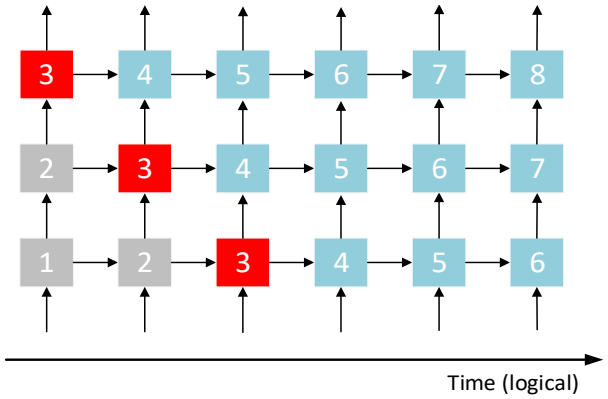

Figure 3. Parallelism among Layers and Logical Timesteps

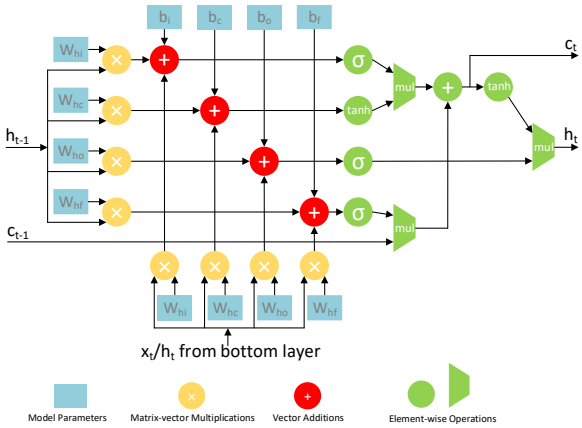

Figure 4. Parallelism inside an LSTM Layer

models, the data dependencies of the model should not be altered. From Figure 2 we can see that the computation of LSTM layers has several data dependencies. First, the computation that require $\mathbf{c}_{t-1}$ cannot start until $\mathbf{c}_t$ is available from the previous time step. Second, computations using $\mathbf{h}_t$ cannot start until ht-1 is generated. Third, if the LSTM layer takes output from bottom layer as its input, the computations using the input vectors must wait until bottom layer is finished. Figure 3 illustrates the data flow of a three-layer LSTM as an example to explore the parallelism. In Figure 3, each box in the vertical direction represents an LSTM layer and the horizontal boxes are the copies of these layers in different time steps. The arrows represent the data dependencies. The execution order is not necessarily restricted by the logical timing of the LSTM model. These boxes in Figure 3 can be launched as soon as the corresponding data dependencies are met. Figure 3 illustrates a possible execution scheme. As can be seen in Figure 3, the grey boxes which are finished are marked with grey, boxes which are being executed are marked with red and boxes to be executed are marked with blue. The execution order is indicated by the numbers in the boxes. The box marked with "1" is executed first, followed by the boxes marked with "2". After boxes with "2" are finished, the data dependencies of all the boxes marked with "3" are met, so they are executed in parallel. Similar rules applied to the rest of the boxes. This inter-layer parallelism is used by the "multi-threading" implementation, as will be described in Section 5.

Parallelism inside an LSTM layer is also feasible under the constrain that data dependencies should not be broken. Figure 4 shows the data flow inside an LSTM layer. As shown in Figure 4, the results of matrix-vector multiplications are added with the biases then passed through the activation functions. The results of these activation functions are then processed by the element-wise operations to get the final output. Based on this data flow, several possible ways to improve parallelism are shown in Figure 4. We observe that all the eight matrix-vector multiplications which are marked with yellow are independent, so that those matrix-vector multiplications can be processed in parallel. Similarly, the additions of the biases which are marked in read and the following activation functions are also parallelizable. Note that it is not necessary for all the parallelism opportunities illustrated in Figure 4 to be exploited. For example, if the computing capability of the system only supports executing two matrix-vector multiplications at the same time, then the eight matrix-vector multiplications should be divided into two groups which will be processed in parallel. This intra-layer parallelism is used by the "helper-core" implementation, as will be described in Section 5.

IV. ACCELERATION SCHEMES

Three acceleration schemes for the dual-core system are discussed in this section: "multi-programming" mode, "multi-threading" mode, and "helper-core" mode. They leverage parallelism at different levels and appropriately schedule computing tasks for the dual-core system. As described in Section 3, "multi-programming" mode runs different models or duplicates of the same model independently on the two cores, increasing the flexibility of the system. The "multi-programming" acceleration scheme has a critical limitation: one of the cores stays idle when there is only one computing job running in the system. An acceleration scheme in which both cores participate in a single job is necessary to improve hardware utilization in the single-job scenario. Two such acceleration schemes are described in this section: "multi-threading" mode in Section 4.1 which leverages parallelism among LSTM layers and time steps; and "helper-core" mode in Section 4.2 which leverages parallelism inside an LSTM layer.

4.1 Multi-threading Mode

As described in Section 3, the parallelism among the layers and time steps of a multi-level LSTM model can be leveraged. For a dual-core system, it is necessary to properly schedule the computation tasks to the two cores to real-

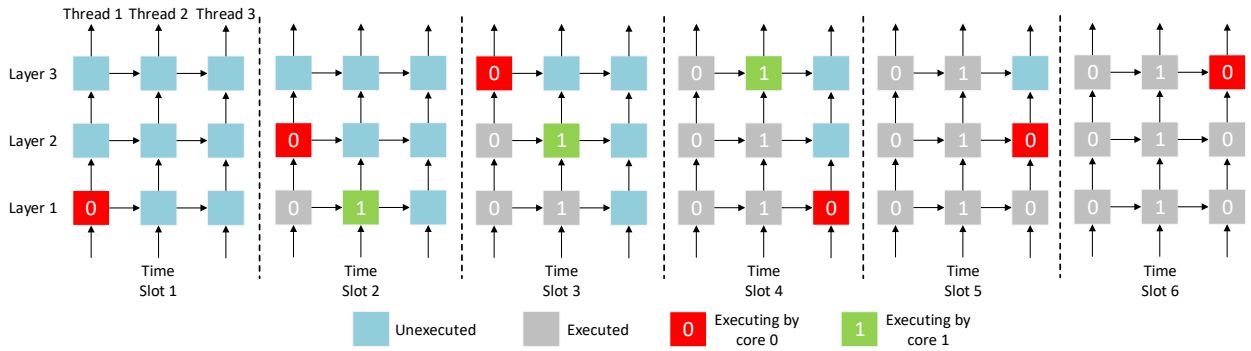

Figure 5. Multi-threading Mode Scheduling

ize high efficiency. One intuitive scheme is to execute the layers in parallel. The 3-layer LSTM model is executed in this manner in the example of Figure 3. However, this scheme has a disadvantage. For example, if this three-level LSTM is executed by a dual-core system using this scheme, layer 1 and layer 3 are assigned to core 0 and layer 2 is assigned to core 1. Core 1 can start to execute layer 2 as soon as core 0 has finished the first step of layer 1, thus these two layers are executed in parallel. However, after the completion of first two layers, only one core can be used to compute layer 3 since there is a data dependency chain connecting each step. This situation happens when the number of layers is not divisible by the number of cores. In other words, this scheme fails to generalize for all combinations of models and hardware resources.

We observe that the sequence length of an LSTM model is usually larger or much larger than the number of layers. For example, the LSTM model used to perform a translation task [14] is a 4-layer LSTM which has a sequence length of 20 to 30 or sometimes larger than 100. Based on this observation, the efficiency of the dual-core system can be improved by a new scheduling scheme. In this scheduling scheme, the computation tasks of each logical time step of the LSTM model are grouped into different threads. The threads are assigned to cores in an interleaved manner according to thread ID and then scheduled based on data dependencies. In the case where there are two cores, the threads with odd thread IDs are assigned to core 0 and the threads with even thread IDs are assigned to core 1. The execution process is divided into timeslots, each core proceeds one step for the current thread during each timeslot. Note that this scheduling scheme also has hardware underutilization when the number of threads is not divisible by the number of cores. Since the number of layers is usually much smaller than the sequence length, the length of the remaining thread is much shorter than an entire layer, thus the time with an idle core is shorter.

Figure 5 demonstrates an example that a 3-layer LSTM with sequence length 3 being executed by a dual-core system. In this case, thread 1 and 3 are assigned to core 0 and thread 2 is assigned to core 1. Core 0 executes layer 1 of thread 1 and core 1 stays idle during timeslot 1 because of the data dependency. In timeslot 2, core 0 executes layer 2 of thread 1 and core 1 executes layer 1 for thread 2. Both cores proceed one step in timeslot 3. In timeslot 4, core 0 jumps to first layer of thread 3 and core 1 executes layer 3 of thread 2. Core 0 finishes thread 3 at the end of timeslot 6 and core 1 stays idle during timeslot 5 and 6. Note that the sequence length is 3 only in this example, the typical sequence length of real-life LSTM model is normally much larger than number of layer, so the last two timeslots in which only one core is utilized is trivial comparing to the entire execution time.

4.2 Helper-core Mode

The multi-threading mode leverages the parallelism among layers and logical time steps of an LSTM model, the underlying parallelism inside a layer is not explored. The helper-core mode leverages parallelism at this level.

There are three observations that can be made: First, the amount of data transferred for the matrix-vector multiplication is much larger than the element-wise part for an LSTM model. Let M denote the size of an LSTM layer and let N denote the input size of this layer. The amount of data transferred for matrix-vector multiplications is 4MN + N + 4MM + M + 4M, the number of data reads for element-wise operations is 0 in our implementation since $\mathbf{i}_t$, $\mathbf{f}_t$, $\mathbf{o}_t$ and $\tilde{\mathbf{c}}_t$ are stored in the on-chip buffer. Second, the amount of computation for matrix-vector multiplication is much larger than that of element-wise operations. The number of operations for matrix-vector multiplication is 4MN + 4MM + 8M while the number of operations for element-wise operations is 4M. Third, as mentioned in Section 3, the eight matrix-vector multiplications are independent of each other. Based on these three observations, we conclude that the primary task to accelerate an LSTM model is to parallelize the matrix-vector multiplications since they are the bottleneck of both the computing and reading phases. Another observation is that the matrix-vector multiplications involving $\mathbf{x}_t$ are not on the data dependency chain. According to this ob-

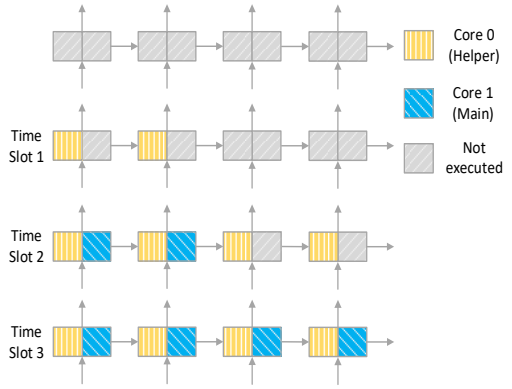

Figure 6. Scheduling of Helper core

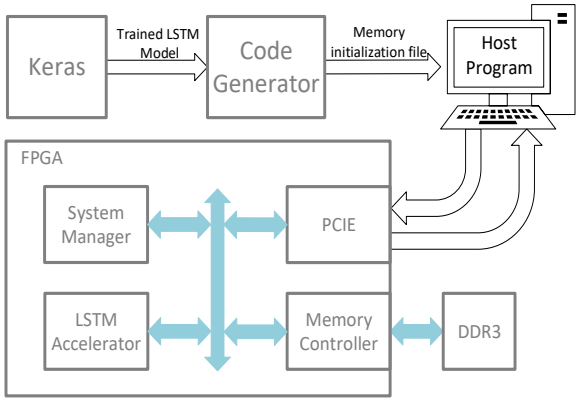

Figure 7. Experimental Setup

servation, the "helper-core" design decouples the matrix-vector multiplications involves $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ and assign them to two cores. The core which runs matrix-vector multiplications using $\mathbf{x}_t$ is called "helper-core" and the other core which runs matrix-vector multiplications using $\mathbf{h}_{t-1}$ is the main core.

4.2.1 Scheduling

In this scheduling scheme, two cores run simultaneously to boost the performance of a single LSTM layer. The helper core always runs ahead of the main core, computing all the matrix-vector multiplications of using $\mathbf{x}_t$ in advance so that the main core only needs to compute the matrix-vector multiplications using $\mathbf{h}_{t-1}$ and the element-wise operations. As a result, the overall system performance is better than that of a single core. If the main core runs too fast, it will have to stall until the helper core generates the necessary data. If the helper core outpaces the main core and there are multiple layers, it is going reach the point where input vector $\mathbf{x}_t$ of the helper core is the output $\mathbf{h}_t$ of the main core and data dependency could possibly be violated. Therefore, synchronization between the main and helper cores is needed to make sure that the two cores are always advancing at the same pace. Computing tasks are scheduled in using the "time slots" as a unit. A fixed number of computing tasks for each core are scheduled for each time slot. The cores do not proceed to the next time slot until both cores finished their current computing tasks.

Figure 6 illustrates an example in which the helper core runs ahead of the main core too boost performance. Each row in Figure 6 represents the status of an LSTM layer. Gray boxes represent work that has not yet been finished. Yellow and blue boxes represent work completed by the main and helper cores respectively. At the beginning none of the jobs have been finished. In time slot 1, the helper core has completed the first half of step 1 and step 2 and the main core stays idle. In time slot 2, the main core completes the remaining half of step 1 and step 2 while the helper core finishes the first half of step 3 and 4. Again, in time slot 3, the main core finishes the second half of step 3 and step 4 started by the helper core while the helper proceeds to the next cell. By making the main and helper cores run in such a synchronized manner, the helper core always stays ahead of the main core and maintain data dependencies.

Benchmark	Layers	Input Size	Layer Size	Application Domain
IMDB [15]	1	128	128	Sentiment Classification
LRCN [16]	1	320	256	Activity Recognition
Show & Tell [21]	1	512	512	Image Captioning
Shakespeare-2 [22]	2	65	128	Text Generation
CTC-3L-421-UNI [23]	3	121	421	Speech Recognition
Translation [24]	3	1024	1024	Translation

Table 1. Benchmark Information

V. EVALUATIONS

5.1 Experimental Setup

Figure 7 shows the overall experimental setup; first, we train the LSTM model using Keras [17]; then, the well-trained model is ready to be read by the code generator. The code generator has two functions: firstly, to read the input data and parameters of the LSTM model and arrange them in the memory initialization file; secondly, the code generator generates instructions for the accelerator, also stored in the memory initialization file. The host program on the host PC reads the memory initialization file and sends it to the FPGA through PCIE. The system manager on the FPGA transfers the data received from the host PC to the main memory during the initialization phase. After execution on the FPGA, the output of the LSTM-RNN are stored in main memory are sent back to the host PC which performs post-processing to get the results.

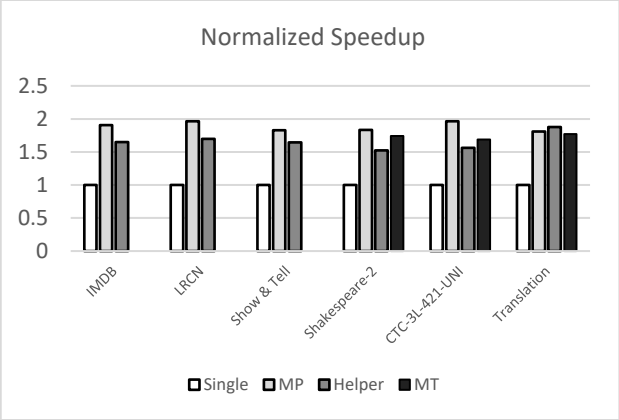

Figure 8. Normalized Speedup over Single-core

The hardware implementation of the single- and dual-core designs are built on the Xilinx VC707 Evaluation board using a Virtex-7 485t FPGA chip and runs at 125 MHz. The design entry is SystemVerilog and the developing environment is Vivado (2016.4). The MAC units and activation functions are implemented using the DSP48 Micro IP core. On-chip buffers are implemented with the Block Memory Generator IP core. The memory controller is implemented using the Memory Interface Generator IP core. The PCIE logic is implemented with Xillybus [18].

The data format used in this work is Q8.8 fixed-bit number format. As mentioned in work [12], the error generated by Q8.8 fixed-bit format is insignificant. In their work, the average error for $\mathbf{c}_t$ and $\mathbf{h}_t$ are 3.9% and 2.8% respectively. Another work [19] studies the relationship between number of quantization bits and the error rates. The authors use different numbers of quantization bits to optimize the LSTM model. The largest quantization width used is 8. Therefore, we choose Q8.8 fixed-point number in our implementation based on these previous works.

5.2 Performance Analysis

We benchmark our implementations with the 6 benchmarks listed in Table 1. Three of them are single-layer models and the other three are multi-layer models. The multi-threading implementation is only benchmarked with the multi-layer models since it does not boost performance of single-layer models. Figure 8 shows the normalized overall speedup of our implementations when the batch size is configured as 32. The reuse factor for the helper-core implementation is set to 4. We show the speedups normalized against the single core design. The legend MP denotes the multi-programming mode while legend Helper represents the helper-core implementation and MT is short for multi-threading. The MP mode is configured to run the same benchmark on the two cores independently and it achieves best performance among our implementations on all benchmarks except the "Translation" benchmark. The helper-core mode has performance loss in comparison with MP and MT implementations, but outperforms them on the "Translation" benchmark. The performance of MT is between MP mode and helper-core mode on "Shakespeare-2" and "CTC-3L-421-UNI" benchmarks but is outperformed by the helper-core on "Translation" benchmark.

To better understand Figure 8, we profile the run-time status of our implementations on those benchmarks and the results are shown in Figure 9. A core can be in one of the following four states: performing computing only, performing memory operations only, performing both, and neither. A hardware profiler is deployed to record the durations of each status. The profiler is implemented using simple state machines and counters; we use the debug probes to observe the states of each core. For the single core implementation, the ratio is simply the corresponding cycles divided by total cycles. For dual-core implementations, the ratio is obtained by averaging the corresponding cycles of the two cores then dividing by total execution cycles.

The time breakdowns for single-core implementation shows that the single-core implementation is working well because at least 99% of the time the computing units are busy which means the computing units are efficiently utilized.

Speedups of MP on IMDB and "Shakespeare-2" are slightly lower because the control overhead is relatively more significant in smaller models. Speedups on large models such as "Show & Tell" and the "Translation" are also slightly lower. As can be seen in Figure 9, memory operations of single-core implementation take over more than 50% of the execution time for "Show & Tell" and the "Translation" benchmarks. The MP implementation can be treated as two identical single cores sharing the same bus to access the main memory. When memory operations take less than 50% of the time for single-core, the memory interface can offer enough off-chip bandwidth to both cores. However, the off-chip memory bandwidth is not sufficient for two cores when memory operations take more than 50% of the time for single-core, so one core may have to start computing stage first and wait for another core finishes using the bus. In this case, the delayed memory operations will finish after the computing stage finishes, which is why MP implementation has more "memory only" time on "Show & Tell" and "Translation" benchmarks and the overall performance is slightly reduced.

There are 2 reasons that causes the performance loss for helper core implementation. The first reason is that only one core is busy during the first and last time slot of "helper-core" mode as mentioned in Section 5. The second reason is that all the element-wise operations are assigned to main core, which makes the amount of jobs assigned to two cores slightly unbalanced. Therefore, the core which is assigned fewer jobs will finish first and then wait until the other core finish, which creates synchronization gaps. As

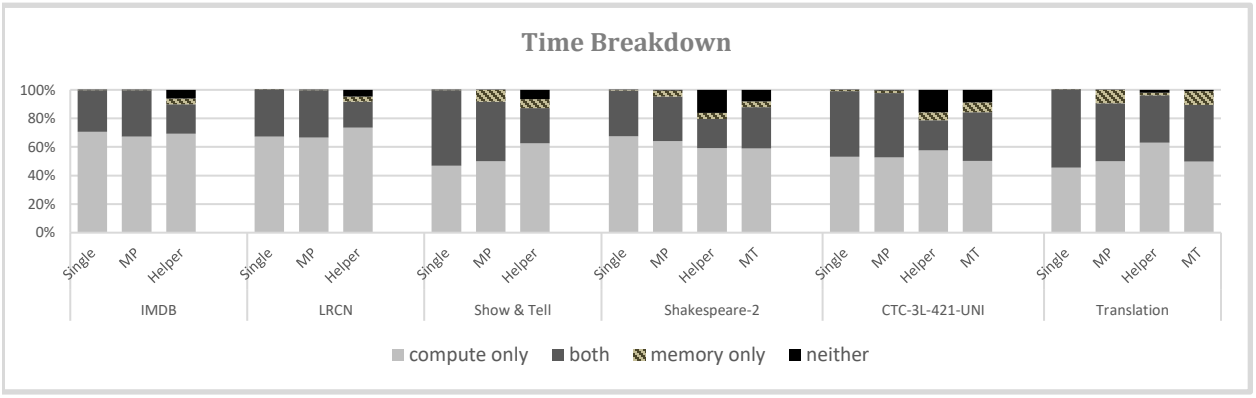

Figure 9. Execution Time Break Down

we can see in Figure 9, the idle state takes part of the execution time for helper-core mode, which means that one of the cores stays idle while waiting for the other core to finish within a time slot. An observation can be made that the idle time of helper-core on "LRCN" and "Translation" benchmarks is shorter, and better speedup is achieved, respectively. For "LRCN", the reason is that the input vectors $\mathbf{x}_t$ of "LRCN" is larger than $\mathbf{h}_{t-1}$, so the helper-core is assigned more matrix-vector multiplication jobs, which, to some extent, balances the tasks assigned to the cores. As for the "Translation" benchmark, as can be seen in Figure 9, the ratio of idle state and read-only state is lower than other benchmarks, and the helper core get better performance improvement. The reason for this is related to the size of the model. The size of LSTM model in "Translation" benchmark is 1024 with input vectors of size 1024, which makes it a relatively larger model than other benchmark models. According to the analysis in Section 5, the amount of computing tasks required for matrix-vector multiplications increases quadratically with the size of the model while the amount of computing tasks for element-wise operations increases linearly with the size of model. The time to execute element-wise operations on main core becomes even more trivial when the size of model is increased to 1024. Therefore, the computing tasks on the cores are more balanced than other benchmark models. In such case where the amount of computing on the two cores are balanced, the hardware utilization on helper-core implementation is higher, so it achieves better performance. Also note that MP and MT meet memory bandwidth bottleneck on "Translation" benchmark as explained before, but helper-core does not have such bottleneck here because of reusing of parameters in on-chip buffers. Another observation that can be made from Figure 9 is that the fraction of the idle time on benchmarks "Shakespeare-2" and "CTC-3L-421-UNI" is more than other benchmarks, and that the speedups of helper-core are lower. The reason for this is that the input vectors $\mathbf{x}_t$ for these 2 benchmarks are smaller than $\mathbf{h}_{t-1}$, which means that the helper-core is assigned fewer jobs in terms of matrix-vector multiplications. The assignment of the computing tasks is further unbalanced due to this reason, so the synchronization gaps for these two benchmarks are larger.

The speedup performance of MT mode is between the MP mode and helper-core mode on "Shakespeare-2" and "CTC-3L-421-UNI". The reason is straightforwardly shown in Figure 9: the idle durations for MT is shorter than those of helper-core because the jobs are assigned in a more balanced manner, so MT is faster than helper-core; and it is slower than MP because MP has no such idle states caused by synchronization. The MP is outperformed by helper-core on the "Translation" benchmark for the same reason as MP: the bottleneck of the system now is off-chip memory bandwidth and helper-core does not have such issue.

5.3 Comparing with CPU and GPU

We compare our implementations with CPU and GPU in this section. The metrics for performance is GOPS and the metrics for power efficiency is GOPS/W. The power consumption is obtained with Yokogawa WT210 power meter. The software configuration of FPGA is the same as Section 6.2. We run the benchmarks on 4-Core CPU (Intel i7 7700K) and GPU (NVIDIA 1080Ti) using Keras [17] with Tensorflow [20] as backend. The software configurations for CPU and GPU are the same as FPGA. Figure 10 shows the performance and power efficiency comparison with CPU and GPU. MP, Help-core, and MT outperform the 4-Core CPU in performance ranging from 3.92x-5.98x, 3.05x-5.38x, and 4.26x-5.10x respectively. The sizes of the models and data dependency chain in LSTM limit the utilization of massive computing resources of GPU whose performance is lower than FPGAs for most of benchmarks. For large models such as the "Translation" benchmark, it achieves higher performance.

As shown in Figure 10, MP, Helper-core, and MT have better power efficiency than that of the CPU ranging from 7.79x-12.19x, 8.90x-15.66x, and 10.87x-13.67x respectively. Compared with GPU, MP, Helper-core, and MT have better power efficiency ranging from 2.18x-23.19x, 2.96x-26.47x, and 2.58x-21.02x respectively.

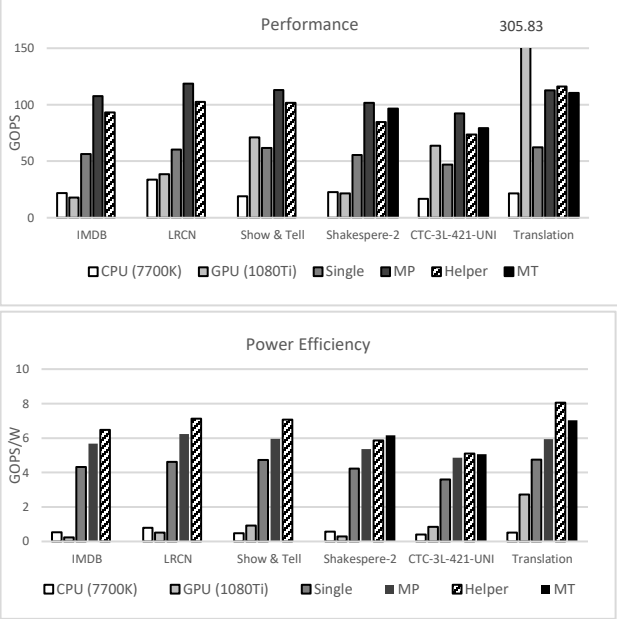

Figure 10. Performance and Power Efficiency

VI. CONCLUSION

In this paper, we observe that single-core accelerators for LSTM-RNNs thwart the use of model-level parallelism. Thus, we propose a dual-core accelerator for LSTM-RNN: able to execute multiple jobs simultaneously or have cores collaborate on a single job. According to our experimental results and analysis, the ratio of computation to communication ratio is critical to the overall performance of the system. Model-level parallelism along with batch processing can be leveraged to improve performance. The "multi-programming" implementation increases throughput up to 1.98x when the system is computation-bound. However, it has no improvement on latency. The "multi-threading" implementation achieves a good throughput improvement and reduces latency as well when system is computation-bound; the "helper-core" reduces the off-chip memory bandwidth bottleneck via the reuse of model parameters when the system is memory-bound and can achieve up to a 1.64x speedup when the batch size is 1.

REFERENCES

[1] L. Huimin, et al., "A high performance FPGA-based accelerator for ge-scale convolutional neural networks," in 2016 26th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–9, 2016.

[2] T. Chen, et al.,"Diannao: A small-footprint high-throughput accelerator forubiquitous machine-learning," in ACM Sigplan Notices, vol. 49, pp. 269–284.

[3] Y. Chen, et al., "Dadiannao: A machine-learning supercomputer," in 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 609–622, Dec 2014.

[4] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 367–379, 2016.

[5] C. Zhang, et al., "Optimizing fpga-based accelerator design for deep convolutional neural networks," in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 161–170, ACM.

[6] J. Albericio, et al., "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), pp. 1–13, 18-22 June 2016.

[7] Y. Ma, et al., "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in FPGA 2017, pp. 45–54.

[8] N. P. Jouppi, et al., "In-datacenter performance analysis of a tensor processing unit," in Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. 1–12.

[9] S. Li, et al., "FPGA acceleration of recurrent neural network based language model," in 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, pp. 111–118, 2-6 May 2015.

[10] E. Nurvitadhi, S. Jaewoong, D. Sheffield, A. Mishra, S. Krishnan,and D. Marr, "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC," in 2016 FPL, pp. 1–4, Aug. 29 2016-Sept. 2 2016.

[11] S. Han, et al., "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in FPGA, pp. 75–84.

[12] A. X. M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on FPGA," in 2017 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–4, 28-31 May 2017.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Computation, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] I. Sutskever, et al., "Sequence to sequence learning with neural networks," in Advances in Neural Information Processing Systems 27.

[15] https://github.com/keras-team/keras/blob/master/examples/imdb_lstm.py.

[16] J. Donahue, et al., "Long-term recurrent convolutional networks for visual recognition and description," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 4, pp. 677–691, 2017.

[17] https://keras.io

[18] http://xillybus.com/

[19] S. Shin, K. Hwang and W. Sung, "Fixed-point performance analysis of recurrent neural networks," 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Shanghai, 2016, pp. 976-980. doi:10.1109/ICASSP.2016.7471821

[20] https://www.tensorflow.org/

[21] https://doi.org/10.1145/3061639.3062187

[22] Vinyals, O., Toshev, A., Bengio, S., & Erhan, D. (2015). Show and tell: A neural image caption generator. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 3156-3164.

[23] https://github.com/karpathy/char-rnn.

[24] A. Graves, A. r. Mohamed and G. Hinton, "Speech recognition with deep recurrent neural networks," 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, Vancouver, BC, 2013, pp. 6645-6649. doi: 10.1109/ICASSP.2013.6638947