Serving Recurrent Neural Networks Efficiently with a Spatial Architecture

Tian Zhao [1] Yaqi Zhang [1] Kunle Olukotun [1]

Abstract

Recurrent Neural Network (RNN) applications form a major class of AI-powered, low-latency data center workload. Most optimization strategies for RNN acceleration break the compute graph into BLAS kernels, which leads to significant inter-kernel data movement and underutilization of resources. We show that expressing an application with more general loop constructs that capture design parameters enables cross-kernel optimization while maintaining high programmability. Such abstraction level can lead to very efficient usage of FLOPS on spatial architecture across a range of problem sizes. We evaluate our fused RNN implementation with DeepBench on a coarse-grained reconfigurable architecture (CGRA). We demonstrate that our implementation provides an improvement of 10-20x in performance, 1.6x in area, and 2x in power efficiency compared to a Tesla V100 GPU, and up to 30x in performance compared to Microsoft BrainWave implementation on a Stratix 10 FPGA.

1 Introduction

Recurrent Neural Networks (RNNs) are a class of sequence models that play a key role in low-latency, AI-powered services in datacenters (Fowers et al., 2018; Jouppi et al., 2017). In these services, a platform assumes that an incoming request contains a single sample per batch, and needs to be served with very low-latency for real-time human computer interaction. An example of such workload is Google Translate, where the inference happens the same time when a user types. Despite its popularity, RNN serving is hard to implement efficiently. Modern software and hardware platforms support optimized BLAS routines. To serve RNNs on these platforms, a compiler tends to stitch multiple optimized BLAS kernels into one compute graph. While a hardware accelerator might execute each individual kernel efficiently, it misses the opportunity of global cross-kernel optimization that can dramatically improves performance and energy-efficiency. This approach leads to two issues. First, communication between BLAS kernels creates large intermediate results, which can lead to poor memory performance when blocking size is not properly tuned for the target system. Missing the opportunity of cross-kernel fusion can lead to huge performance loss due to different access latency at each level of memory hierarchy in a processor-based architecture. On a spatial architecture, while the first two levels of memory hierarchies, namely registers and on-chip scratchpads, tend to have single cycle access latency, the energy to access these two type of memories are widely different. Therefore, lack of cross-kernel fusion can lead to low energy-efficiency. Second, hardware accelerators tend to use large vectorization and blocking sizes to boost compute density when accelerating BLAS kernels, which suffers from resource underutilization when problem sizes are not a multiple of the blocking size. The utilization is worse with RNN applications, which are composed of sequences of small matrix multiplications due to small hidden unit sizes and many time steps. Moreover, many accelerator platforms are optimized for BLAS level-3 operations, e.g. NVBLAS Library for GPU (nvb), TPU (Jouppi et al., 2017), EIE (Han et al., 2016a), EyeRiss (Chen et al., 2017), and DaDianNao (Chen et al., 2014). These platforms suffer low resource utilization when serving single-batch, real-time RNN applications that contain a lot of matrix vector multiplication (MVM) executions.

To address these issues, we propose the following optimization strategies. First, we fuse all gate-level BLAS kernels with element-wise, non-linear functions within a single time step. This way, all of our intermediate results are buffered as registers as opposed to memories. Second, we spatially parallelize and pipeline the computation graph. We use small tiling factors for the dot product to explore vectorized parallelism, and explore tiled parallelism and coarse-grain pipelining by unrolling the outer loop nests based on the target resources. This approach exposes the gate-level parallelism in RNN cells and pipelining between MVM and element-wise non-linear functions, and minimizes the frag-

[1]Department of Electrical Engineering, Stanford University, Stanford, USA. Correspondence to: Tian Zhao <tianzhao@stanford.edu>, Yaqi Zhang <yaqiz@stanford.edu>, Kunle Olukotun <kunle@stanford.edu>.

Proceedings of the 2^{nd} SysML Conference, Palo Alto, CA, USA, 2019. Copyright 2019 by the author(s).

mentation for different problem sizes. Additionally, the entire pipeline is data-flow driven with no dynamic scheduling overhead.

We implement our design in Spatial (Koeplinger et al., 2018), which is a Domain-Specific-Language (DSL) that describes applications with nested loops and explicit hardware memory hierarchy. Next, We map our implementation on Plasticine (Prabhakar et al., 2017), which is a coarse-grained reconfigurable architecture (CGRA) with reconfigurable compute pipeline and hierarchical interconnection network that accelerate applications in Spatial. Furthermore, we propose augmentation to the Plasticine microarchitecture in order to support mix-precision operations, which is critical for serving RNNs in real-time.

Finally, we evaluate our optimization strategies on RNN tasks in DeepBench (Narang & Diamos, 2017), and compare to the state-of-the-art RNN serving platforms. We show our implementation delivers consistently high FLOPS across tasks of various sizes and demonstrate energy-efficiency advantage of spatial architectures compared to processor-based architectures.

The key contributions of this paper are:

1. We analyze the computation and memory layout of RNN cell implementations on commercially available platforms. We find that BLAS abstraction leads to expensive inter-kernel data movement and resource underutilization.

2. We address these issues by describing RNN applications using abstractions with more general loop constructs that allow cross-kernel optimization, spatial parallelization, and pipelining of arbitrary loop nesting. To achieve low-latency inference for RNN applications, we propose micro-architectural co-design to Plasticine in order to enable low-precision operations.

3. Finally, we thoroughly evaluate three commercial platforms, CPU, general purpose graphics processing unit (GPGPU), field-programmable gate array (FPGA), and a previously-proposed CGRA, as serving platforms for RNN applications.

The rest of the paper is organized as follows. Section 2 provides backgrounds on RNN algorithms, DSL and hardware platform used in this paper. Section 3 discusses the available RNN implementations on commercially available platforms. We then discuss the loop-level optimization implemented in this work that addresses the inefficiency in these implementations. Section 4 discusses the architectural changes for supporting efficient RNN inference on Plasticine. Section 5 details our evaluation methodology and experimental results. Section 6 discusses related work on available software and hardware optimization strategies for serving RNN applications. Section 7 offers concluding remarks and future work.

Name	Shape	Specification
x_t	D	LSTM cell's input vector
f_t	H	Forget gate's activation vector
i_t	H	Input gate's activation vector
o_t	H	Output gate's activation vector
j_t	H	Candidate of memory gate's activation vector
c_t	H	Memory gate's vector
$W_{h_{i,j,f,o}}$	H, H	Hidden state's weight matrices at gate i, j, f, o
$W_{x_{i,j,f,o}}$	H, D	Input vector's weight matrices at gate i, j, f, o
b	H	Bias vector at gate i,j,f,o

Table 1. LSTM specifications

2 Background

RNNs are widely used to model arbitrary sequential tasks. An RNN contains a cell unit to iteratively consume a T-step input sequence $x = [x_0, x_1, \cdots, x_T]$ in order to generate an output sequence $y = [y_0, y_1, \cdots, y_T]$. Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) and Gated Recurrent Unit (GRU) (Chung et al., 2014) are popular RNN cell units. In this paper, we use LSTM as an example. Nevertheless, our optimization techniques can be generalized to any other types of RNN cells. In Section 5, we also provide evaluation of GRU implemented using our techniques.

2.1 LSTM Cell

At step t, an LSTM generates an output y_t and the next memory cell states c_t and h_t as follows:

$$i_t = \sigma(W_{h_i} h_{t-1} + W_{x_i} x_t + b_i) \quad (1)$$
$$j_t = \tanh(W_{h_j} h_{t-1} + W_{x_j} x_t + b_j) \quad (2)$$
$$f_t = \sigma(W_{h_f} h_{t-1} + W_{x_f} x_t + b_f) \quad (3)$$
$$o_t = \sigma(W_{h_o} h_{t-1} + W_{x_o} x_t + b_o) \quad (4)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ j_t \quad (5)$$
$$y_t = h_t = o_t \circ \tanh(c_t) \quad (6)$$

H, D are dimensions of hidden states and input features, respectively. R is the sum of hidden state and input feature dimensions. $\circ$ is the Hadamard product. Table 1 shows the specifications for each matrix and vector in LSTM cell.

2.2 Spatial Reconfigurable Architectures

Spatial reconfigurable architectures, such as FPGAs, are gaining traction as data center accelerators for their energy efficiency (Amazon, 2017; Putnam et al., 2014; Ouyang et al., 2014). By reconfiguring memory and compute based on applications and computation requirements, these platforms are able to avoid resource underutilization as compared to processor-based architectures. In addition to exploiting parallelism, spatial architecture offers pipelining that provides a larger space to remove bottlenecks and reduce latency. Nonetheless, the traditional low-level programming interface and long synthesis time of FPGA was the major obstacle for it to become a mainstream accelerator. A Coarse-grained reconfigurable architecture (CGRA) is usually configured at word-level granularity and contains a hierarchical interconnection network as supposed to bit-level flat-interconnection in an FPGA. In exchange, the reduction in connectivity in hardware translates to lowered routing overhead and higher clock frequency. The reduced routing overhead usually provides a higher compute density and memory capacity, which makes CGRA an attractive platform to accelerate deep learning workloads. Due to the complexity of mapping applications, spatial architectures tends to need design space exploration (DSE) in order to achieve good resource utilization and performance (Koeplinger et al., 2016; Liu & Schafer, 2016).

2.3 Spatial

Spatial is a hardware-centric DSL that expresses applications in pattern-based loop structures and explicit memory hierarchies. While a user describes applications in untimed, un-parallelized loops, Spatial automatically pipelines and retimes the inner loop body. It also unrolls and schedules the outer loop nests. To sustain the bandwidth requirement for parallelization and throughput requirement for pipelining, Spatial automatically banks and buffers the intermediate scratchpad memories. Spatial exposes important design parameters such as blocking size and unrolling factor. Using the exposed parameters, users can easily tune their design either manually or with a external DSE engine to balance pipeline stages and saturate resource for different tasks on different hardware platforms. Currently, Spatial supports a few FPGA targets and a previously proposed CGRA, Plasticine.

2.4 Plasticine

Plasticine is a tile-based CGRA that accelerates general nested loop patterns in Spatial as opposed to BLAS kernels in most accelerators. It consists of primarily two types of units: a pattern compute unit (PCU) which contains a single instruction multiple data (SIMD) pipeline optimized for accelerating vectorized map and reduction loops, and a pattern memory unit (PMU) which contains configurable buffered and banked memory that support various banking scheme for different access patterns. Plasticine can explore inner loop parallelism with vectorized SIMD lanes, inner loop pipelining with SIMD pipeline, outer loop parallelism with unrolled loops mapped across PCUs. It can also explore outer loop pipelining with pipeline stages between PCUs and with PMUs as buffers for the intermediate results.

Symbol	Processor	Reconfigurable Hardware
🟪	Kernel	Inner Loop
🟦	Memory Hiearchy	On-chip Scratchpad
🟩	Register File	Register
🔴 op	Element-wise Operation	
🟨	Outer Loop	
VecParam ⟷	Vectorization Parameter representing AVX or SIMD instructions	
↗ UnrollParam	Unrolling Parameter representing multiple hardware instances' concurrent execution in spatial architecture	

Parameter	Specification
hv	Vectorization Parameter on H
hu	Unrolling Factor on H
rv	Vectorization Parameter on R
ru	Unrolling Factor on R
G	Number of Gates in an RNN. For LSTM, G=4

Table 2. Specifications for symbols and parameters in Section 3

Spatial can map arbitrarily nested loops on Plasticine. By choosing design parameters such as parallelization factors for the loops, Plasticine can balance the pipeline for each stage at all loop levels. More architectural details about PCU SIMD pipeline will be explained in Section 4.

3 Application Analysis

In this section, we first discuss BLAS-based LSTM implementations on processor architectures. We then discuss implementations on spatial architectures. Last, we discuss our implementation of loop-based LSTM in Spatial and how it is mapped onto Plasticine. Table 2 contains specifications for symbols and parameters used in this section.

3.1 BLAS-based LSTM on Processor Architecture

Modern Machine Learning frameworks, such as TensorFlow (Abadi et al., 2016), divide the compute graph of an LSTM cell into BLAS kernels. Then, a BLAS kernel is accelerated via low-level optimized BLAS subroutines, such as Intel BLAS Library on CPU and NVBLAS Library on GPU. With a naive implementation of LSTM in TensorFlow, as shown in in Figure 1 (a), such implementation can create large memory footprint since all intermediate results are materialized in memory. A common strategy to tackle the issue is through fusing blocked kernels. With TensorFlow abstraction, this can only be achieved by expressing the entire problem as an optimized kernel, for example with `LSTMBlockFusedCell` and `GRUBlockCell` modules in TensorFlow, which are the fastest implementations of RNN cells in TensorFlow for CPU. User can tune a single inner kernel parallelism with MKL Library support. In practice, such implementation can provide significant performance improvement over the naive implementation, but is still very hard to saturate CPU compute capacity, potentially due to the high synchronization overhead across threads when MVM size is relatively small.

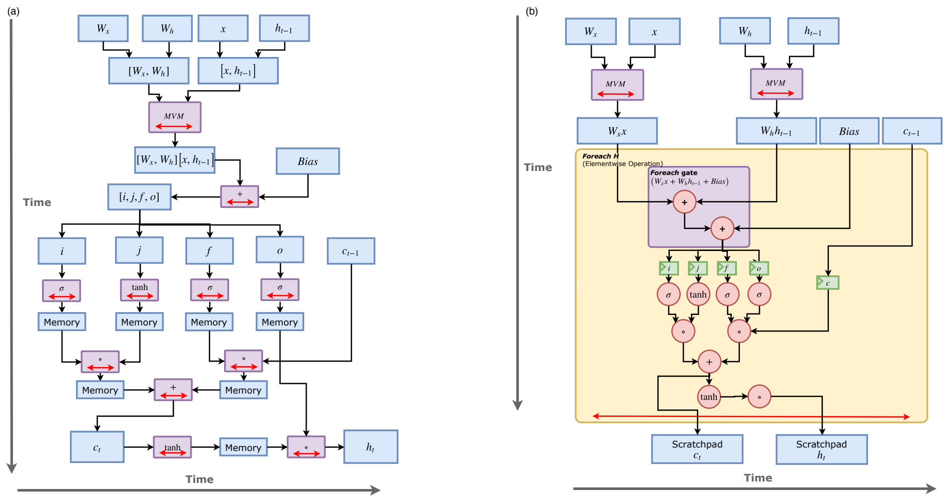

Figure 1. Computation and Memory Layout of LSTM implemented by TensorFlow, served on CPU (a) and GPU (b)

Figure 1 (b) shows the computation layout of TensorFlow with cuDNN library (Chetlur et al., 2014) on GPU. cuDNN is a NVIDIA GPU library for accelerating deep neural networks. To minimize data movement, cuDNN fuses all the vector-vector (VV) operations after MVM. Specifically, the bias add in Equation 1, 2, 3, 4, and all the operations in Equation 5, 6, are fused into one kernel. Nevertheless, there are still intermediate buffers that is of full H size between the MVM kernel and the element-wise operations.

Compared to an naive TensorFlow LSTM implementation, cuDNN approach eliminates many large intermediate memories. However, the MVM of Equation 1, 2, 3, 4 are all accelerated with BLAS3 kernels, which performs only matrix-matrix level operations. This turns MVM and VV bias add into Matrix Matrix Multiplication (MMM) and Matrix Matrix Addition (MMA), which can lead to serious under-utilization of GPU.

Moreover, a processor-based architecture introduces large energy overhead of instruction decoding and scheduling. GPU especially suffers from power hungry high throughput memory-hierarchy. For these reasons, both platforms are not suitable for energy-efficient low-latency RNNs serving platforms.

3.2 BLAS-based LSTM on Spatial Architecture

Previous work has studied the capability of using an FPGA as a low-latency serving platform. Implementing application serving on FPGAs gives the flexibility of resizing MVM and VV units based on the application size. In addition, MVM and VV units can be implemented with hardware pipelining, which removes instruction scheduling and control overhead compared to a processor-based architecture. The latest version of Intel Stratix 10 further boosts the compute power of FPGA dramatically by increasing hardened digital signal processing (DSP) blocks with floating point capability and expanding on-chip memory capacity. Microsoft BrainWave (BW) (Fowers et al., 2018) is a state-of-the-art FPGA-based accelerator that utilizes these features.

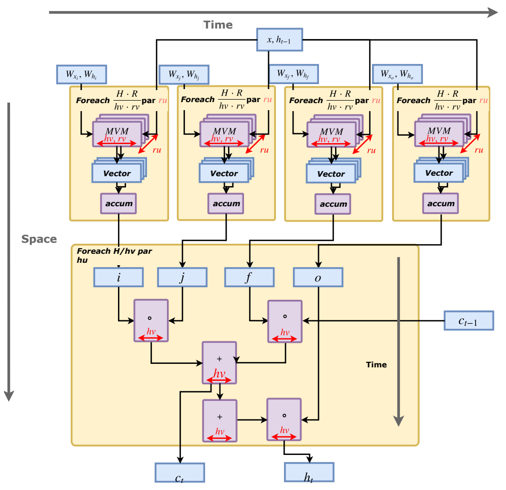

Figure 2. Computation and memory layout of LSTM in BrainWave, served on FPGA

Figure 2 shows BW's compute and memory layout. In contrast to CPU and GPU implementation, BW's implementation blocks the MVM along both row and column dimensions, and fuses the inner tiled MVM with element-wise non-linear functions. Specifically for a matrix of size $H \times R$ and a vector of size $R \times 1$, they compute $hv \times rv$ tiles with multiple tiled hardware units that explore row and column parallelism. A blocking diagram of the weight matrices is shown in Figure 3 (a). Parallel tiles along the row dimension are then fed into a pipelined reduction and accumulation unit. Immediately after the accumulation, multi-function units (MFUs) execute the element-wise operations on that

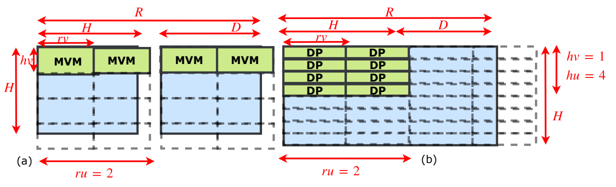

Figure 3. Fragmentation in BrainWave (a) and Spatial (b) in MVM.

hv vector chunk produced by the accumulator. Although BW's implementation still has vectorized intermediate results, their size hv is much smaller compared to the cross kernel intermediate buffers in TensorFlow's implementation, which are of size H. Nonetheless, with parallelization in ru and hu, lots of vectorized intermediate buffers are produced, which can still lead to energy inefficiency. BW performs the one MVM operation in $\lceil \frac{H}{hv} \rceil \lceil \frac{R}{rv \cdot ru} \rceil$ iterations, where ru is the number of parallel MVM units. The MVM operations are executed on each gate of the LSTM in a sequential manner. Similarly, element-wise operations hv using $\sigma, \tanh, \circ, +$ for the non-linear operators are also scheduled to execute on the vectorized multi-function units with size of hv, as shown with the arrow in time in Figure 2. To avoid DRAM communication overhead and improve compute density, BrainWave embeds MVM in blocked floating point format, where the vector of hv values share a single 5-bit exponent and have distinct 5 bit mantissa for each value. As a result, they can achieve very high density low-precision computation, with one adder per hv values and hv multipliers for a vector of hv. The remaining operations are performed in 16-bit precision.

When matrix dimensions are not multiples of hv and $rv \cdot ru$, BrainWave can suffer from underutilization of the compute FLOPS due to inefficient padding, as shown in Figure 3. The underutilization is worse with small problem sizes. They also compute $W_x X$ and $W_h H$ separately rather than computing them with concatenated larger matrices, which is another source of inefficiency due to fragmentation. We believe this might be because only X is updated at the end of the step. However, their abstraction level does not allow operational updates on partial array.

3.3 Loop-based LSTM on Plasticine

We have made the following observations from analyzing BLAS-based LSTM implementations:

1. Constructing an LSTM cell's compute graph using BLAS subroutines introduces large intermediate buffers due to lack of cross-kernel fusion even though the kernels themselves are blocked. Each element on the RNN cells' non-reduction dimension of the MVM (H dimension of $H \times R$ matrix) can be computed completely independently inside one time step. This exposes the opportunity of fine-grain loop tiling and fusion across the entire LSTM kernel. Doing so will reduce the intermediate buffer between MVM and element-wise function to register-level memory.

2. MVM is the computation bottleneck in serving RNN cells. Spatial architecture allows us to distribute most of the compute resource to MVM to achieve pipeline balancing with the element-wise operation stage. Furthermore, reconfigurable compute units with hardware pipelines allow us to avoid scheduling overhead and achieve better hardware resource utilization across various application size.

3. Using low-precision operations can boost compute density and keep RNN weights on-chip to avoid high latency DRAM communication. We need to introduce efficient low-precision support in Plasticine that does not reduce Plasticine's reconfigure granularity.

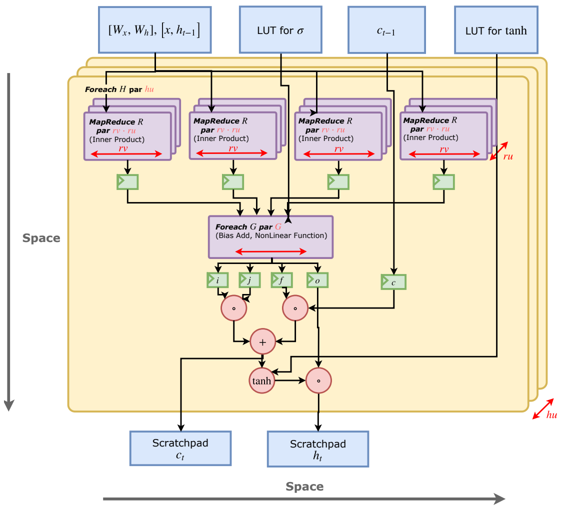

Figure 4. Computation and memory layout of LSTM in Spatial, served on Plasticine

To address the issue of large intermediate buffers, we choose to execute LSTM at a level lower than MVM. We observe that the computation for generating every single element in c_t and h_t can be completed independently from each other within a single step. We refer to this operation as an LSTM-1 operation. Iterating LSTM-1 operation for H times generates the same result as executing the full LSTM cell function using BLAS kernels. The basic constructs in LSTM-1 are either loop constructs (i.e. For loop) or parallel pattern (i.e. MapReduce). Since the size of an LSTM-1 operation is much smaller than the full LSTM function, an LSTM-1 operation can be mapped completely spatially on a spatial architecture. Moreover, all the intermediate buffers in LSTM-1 can be implemented using a small set of registers. Figure 4 shows a mapping of LSTM-1.

In Figure 4, each MVM unit is replaced by a MapReduce unit. Each MapReduce unit consumes rv elements and gen-

```
1  // Number of hidden units in h and features in x
2  val H, D = ...
3  // Loop unrolling / vectorization parameters
4  val hu, ru, hv, rv = ...
5  val c = SRAM[T](H) // SRAM storing C
6  val xh = SRAM[T](D+H) // SRAM storing [X,H]
7  // Weights [Wx,Wh] for different gates
8  val wi, wj, wf, wo:SRAM2[T] = ...
9  val bi, bj, bf, bo: SRAM[T] = ... // Bias
10 // Lookup tables for non-linear functions
11 val luti, lutj, luf, luto: SRAM[T] = ...
12 val tanh:SRAM[T] = ... // Lookup table for tanh
13 Sequential.Foreach (nSteps by 1){ step =>
14   // Loop range from 0 to H parallelized by hu
15   Foreach(H par hu){ ih =>
16     def fusedDotProductWithNonLinear(w:SRAM2[T], lut:SRAM[T],
           b:SRAM[T]) = {
17       // Tiled dot product with blocking size of rv
             parallelized by ru
18       val elem = Reduce(Reg[T])((D+H) by rv par ru){ iu =>
19         Reduce(Reg[T])(rv par rv){ iv =>
20           val iuv = iu + iv
21           w(ih, iuv) * xh(iuv)
22         }{ (a,b) => a + b }
23       }{ (a,b) => a + b }.value + b(ih)
24       lut(elem)
25     }
26     val i = fusedDotProductWithNonLinear(wi, luti, bi)
27     val j = fusedDotProductWithNonLinear(wj, lutj, bj)
28     val f = fusedDotProductWithNonLinear(wf, lutf, bf)
29     val o = fusedDotProductWithNonLinear(wo, luto, bo)
30     c(i) = i*j + c(i) * f
31     xh(i+D) = tanh(cNew) * o
32   }
33 }
```

Figure 5. Example of LSTM in Spatial

erates one temporary result using vectorized reduction. ru is the number of parallel MapReduce units. Each gate accumulates ru temporary results into one final result, adds the result with a bias, and passes the result through a chain of function units. Each function unit is dedicated to executes only one function, which can be $\sigma, \tanh, \circ$ or $+$. Execution through the chained dedicated function units is fully pipelined. At the outer loop, LSTM-1 runs for $\frac{H}{hu}$ iterations, where hu is the number of parallel LSTM-1 implementations.

Regarding utilization, our design causes less fragmentation compared to BW. Figure 3 shows the case where the size of a weight matrix is not a multiple of a MapReduce unit's capacity. BW contains 2-D fragmentation on both H and D dimensions, as shown in Figure 3 (a). Compared to BW, our implementation only contains 1-D fragmentation on R dimension, as shown in Figure 3 (b). Therefore, we only suffer from 1-D fragmentation. Compared to BW, we can achieve equal or higher hardware utilization rate on all sizes of RNN applications.

Using BLAS terminology, an LSTM-1 operation can be thought of as a new BLAS level-1 routine, i.e. element-wise operations. In contrast, previous work such as BW all focuses on optimizing level-2 or 3 routines. We find that optimizing at the lowest BLAS level allows us to utilize hardware resources more efficiently. In addition, using parallel patterns and loop constructs as the basic building block offers sufficient abstraction and does not lead to much engineering overhead.

Figure 5 shows our implementation in Spatial. In this implementation, we use $Foreach$, a loop construct, and $Reduce$, a parallel pattern construct which executes MapReduce, as the basic buildling blocks. To efficiently fine-tune an RNN application on a spatial architecture, we expose the loop tiling and unrolling factor for $Foreach$ and $Reduce$ as hu, ru.

To gain efficient execution of the loop and parallel pattern constructs, we map our implementation onto Plasticine. $Foreach$ at Line $17, 19$ and $Reduce$ at Line $22, 23$ are mapped to PCUs on Plasticine. If the application size is small, these constructs are executed using pipelined SIMD lanes in 1 PCU. If the application size is large, multiple PCUs can be chained together to construct a deep pipeline or a large reduction tree.

To fit RNN's weights on-chip, we execute our application with low-precision. We discuss the microarchitectual changes needed for supporting low-precision arithmetics and reduction implementation in Section 4.1.

4 Plasticine Specialization for RNN Serving

In this section, we propose micro-architectural changes to Plasticine. We also discuss the strategy to choose architectural parameters for Plasticine to serve RNN applications efficiently.

4.1 Mixed-Precision Support in Plasticine

Previous work (Fowers et al., 2018; Jouppi et al., 2017) has shown that low-precision inference delivers promising performance improvements without sacrificing accuracy. In the context of reconfigurable architectures, e.g. FPGAs, low-precision inference reduces the amount of on-chip capacity requirement for storing weights and intermediate data. Moreover, this approach linearly increases the compute density of a reconfigurable architecture.

To support low-precision arithmetic without sacrificing coarse-grained reconfigurability, we introduce low-precision struct type, $4\text{-}float8$, and $2\text{-}float16$, in Spatial. Both types packs multiple low-precision values into a single precision storage. For example, a $4\text{-}float8$ float fits 4 8-bit floating-point numbers into a 32 bit slot. We support only 8 and 16-bit precisions, which are commonly seen in deep learning inference hardwares. Users can only access values that are 32-bit aligned. This constraint guarantees that the microarchitectual change is only local to the PCU. Banking and DRAM access granularity remains intact from the original design.

Figure 6 (a) shows the original SIMD pipeline in Plasticine PCU. Each FU supports both floating-point and fix-point

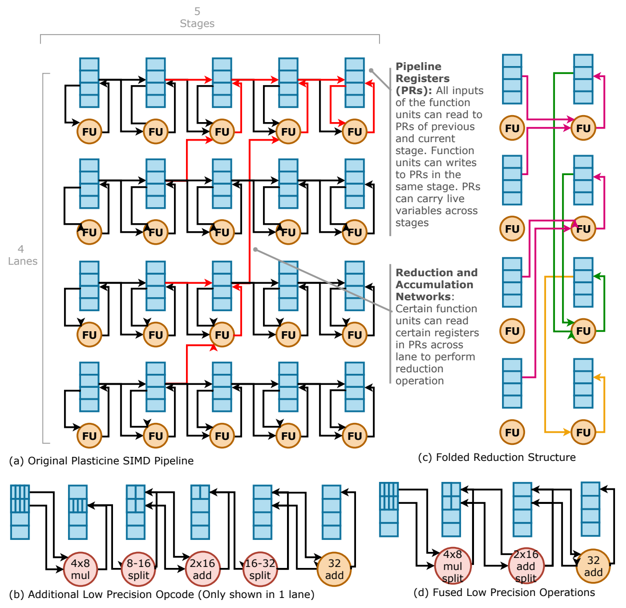
5 Stages
4 Lanes
Pipeline Registers (PRs): All inputs of the function units can read to PRs of previous and current stage. Function units can writes to PRs in the same stage. PRs can carry live variables across stages
Reduction and Accumulation Networks: Certain function units can read certain registers in PRs across lane to perform reduction operation
FU
(a) Original Plasticine SIMD Pipeline
(c) Folded Reduction Structure
4x8 mul
8-16 split
2x16 add
16-32 split
32 add
(b) Additional Low Precision Opcode (Only shown in 1 lane)
4x8 mul split
2x16 add split
(d) Fused Low Precision Operations

Figure 6. Plasticine PCU SIMD pipeline and low-precision support

operations. When mapping applications on Plasticine, the inner most loop body can be vectorized across lanes of the SIMD pipeline, and the linearized loop body can be mapped to different stages. Each pipeline stage contains a few pipeline registers (PRs) that allow propagation of live variables across stages. The PRs are accessible as both inputs and outputs of the FU. An FU can also read previous stage's PRs as its input value. Special cross lane connections as shown in red in Figure 6 enables reduction operations.

To support 8-bit element-wise multiplication and 16-bit reduction, we add 4 opcodes to the FU, shown in Figure 6 (b). The 1^{st} and 3^{rd} stages are element-wise, low-precision operations that multiply and add 4 8-bit and 2 16-bit values, respectively. The 2^{nd} and 4^{th} stages rearrange low-precision values into two registers, and then pad them to higher precisions. The 5^{th} stage reduces the two 32-bit value to a single 32-bit value. From here, we can use the original reduction network shown in Figure 6 (a) to complete the remaining reduction and accumulates in 32-bit connection. The red circles represent opcodes newly introduced opcode, and the yellow circles represent opcodes already supported by Plasticine.

With 4 lanes and 5 stages, a PCU first reads 16 8-bit values, performs 8-bit multiplication followed by rearrangement and padding, producing a 16 16-bit values after the second stage, stored in 2 PRs of each lane. Next, 16 16-bit values are reduced to 8 16-bit values, and rearranged to 8 32-bit value in 2 registers per lane. Next, the element-wise addition in 32-bit value reduces the two registers in each line into 4 32-bit values, which then feed through the reduction network that completes the remaining reduction and accumulation in two plus one stages.

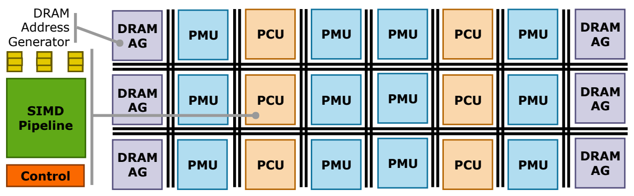

Figure 7. Variant configuration of Plasticine for serving RNN

In a more aggressive specialization, we can fuse the multiply and split into the same stage. We also fuse the first low-precision reduction with the next split (In Figure 6 (d), we fuse the first and second two stages from (c)). In this way, we can perform the entire low-precision map-reduce in 2 stages in addition to the original full precision reduction. In order to maximize hardware reuse, we assume that it is possible to construct a full precision FU using low-precision FUs.

In addition, we observe that the original reduction network in the SIMD lanes could lead to low FU utilization. To improve FU utilization, we fold the entire tree structure in a single stage. Figure 6 (c) shows the folded reduction accumulation structure. Specifically, latter reductions in the tree are mapped to earlier stages in the pipeline. In this setup, the entire reduction plus accumulation is still fully pipelined in $\log_2(\#_{LANE}) + 1$ cycles with no structural hazard. With fused reduced-precision multiplication and reduction, and folded reduction tree, a PCU is able to perform all map-reduce that accumulates $4 \times \#_{LANE}$ 8-bit values using 4 stages. All the operations are completed in $2 + \log_2(\#_{LANE}) + 1$ cycles.

4.2 Sizing Plasticine for Serving RNN

Evaluating an RNN cell containing N hidden units and N input features requires $2N^2$ computations and $N^2 + N$ memory reads. With large N, the compute to memory ratio is 2. The original Plasticine architecture uses a checkerboard layout with 1 to 1 ratio between PCU and PMU. A PCU has 6 stages and 16 lanes, and a PMU has 16 banks. This provides a 6:1 ratio between compute resource and on-chip memory read bandwidth. As a result of this layout, on-chip memory read bandwidth becomes the bottleneck for accelerating RNN serving applications. Given that RNNs cover a wide range of important applications, we select a Plasticine configuration tailored for RNN serving. Specifically, we choose a 2 to 1 PMU PCU ratio with 4 stages in each PCU. Figure 7 shows the layout of this Plasticine variant.

5 Evaluation

In this section, we evaluate real-time RNN serving tasks in various platforms. We start with methodology of our experiments, followed by a discussion of performance and power comparisons across these platforms.

5.1 Methodology

To evaluate RNN serving, we use LSTM and GRU tasks from Baidu DeepBench as our benchmarks. We evaluate the benchmarks across processor-based architectures – CPU, GPU, and spatial architectures – FPGA, and CGRA. Table 4 shows the detailed specifications of the targeting hardware, which includes state-of-the-art high performance platforms in each of the commercialized categories. Table 5 shows the summary of application configurations on each platform.

CPU We implement the applications in TensorFlow 1.10, and evaluate our implementations on Intel Xeon Scalable Processor (Skylake) CPU. We use the `LSTMBlockFusedCell` and `GRUBlockCell` kernels in TensorFlow. We further enable AVX2 vector instructions for CPU evaluation. Due to lack of low-precision support in both tool chain and platform, we used single-precision for our implementation.

GPU We use TensorFlow with cuDNN Library to target NVIDIA Tesla V100 GPU from Google Cloud. cuDNN is a GPU-accelerator Library from NVIDIA that is specialized for deep learning. We used the lowest precision supported, 16-bit precision, for our implementation on GPU. On both CPU and GPU platforms, we run *TensorFlow* profilers and collect the time spent only on evaluating the RNN cells.

Plasticine We implement the applications in Spatial, which targets Plasticine. Although Spatial also has FPGA back-end support, Stratix 10 is not commercially available at the time of submission of this work. The current FPGA targets that Spatial support are not comparable to Stratix 10 both in terms of memory and compute capacity. Therefore, we only use Spatial to target Plasticine for this evaluation. However, our approach should generally benefit an implementation on a high performance FPGA like Stratix 10. We choose Plasticine configuration that matches the peak 8-bit FLOPS and the on-chip scratchpad capacity of a Stratix 10 FPGA. The exact configuration of Plasticine is shown in Table 3. In order to minimize overhead of low-precision support, Plasticine only supports 8-bit, 16-bit, and 32-bit element-wise operations, and mixed precision reduction operation. In case of reducing 8-bit values, the first stage of the reduction is performed in 16-bit, while the remaining of the reduction and accumulation are performed in 32 bit operations. To study performance of Plasticine, we use a cycle accurate simulator for Plasticine. We modified the simulator to model our the proposed micro-architectural changes to support low-precision operations. We use the area and power characterization from the original paper for individual CUs and network switches, and rescale to the configuration shown in Table 3. As discussed in Section 4, we reduce number of stages in PCU from 6 stages to 4 stages with fused low-precision operations and folded reduction tree. We also increase the PMU to PCU ratio to better matching the compute to memory ratio for RNN inference applications. To match the memory capacity of Stratix 10, we shrink the scratchpad capacity of each PMU from 256kB to 84kB. We conservatively estimate the area and power of individual PCU and PMU to be roughly the same compared to the original Plasticine. Function units can be used to compose full precision units, and the overhead of the composability is less than the area and power of the removed two stages. For power calculations, we generate activity tracing of the CUs from simulation, and then integrate with characterized power to compute the total power. The power and area characterizations are synthesized at 28nm technology and reached 1GHz clock frequency.

Table 3. Plasticine configuration

# Row	24	# Column	24
# PCU	192	# PMU	384
# Lanes in PCU	16	# Stages in PCU	4
Scrachpad capacity per PMU	84kB		

Table 4. Hardware specifications for target platforms.

Specification	Intel Xeon Skylake (Dual core)	Tesla V100 SXM2	Stratix 10 280 FPGA	Plasticine
Max Clock Rate (GHz)	2.0/2.8*	1.38/1.53*	1	1
On-chip memory** (MB)	55	20	30.5	31.5
Peak 32-bit TFLOPS	–	15.7	10	12.5
Peak 8-bit TFLOPS	–	–	48	49
Technology (nm)	14	12	14	28
Die Area (mm^2)	64.4	815	1200	494.37
TDP (W)	15	300	148	160

* Base/Boosted Frequency ** Capacity of L3 cache for CPU, register file for GPU, and on-chip scratchpad for reconfigurable architectures.

BrainWave Finally, we also compared our results to Microsoft BrainWave framework, which is a cloud-based DNN acceleration platform for real-time serving. For this evaluation, we compare to BW implemented on top of Intel Stratix 10 FPGA. Brainwave was synthesized at 250MHz and all operations are performed in 8-bit floating point precision.

Table 5. Application configurations for target platforms.

Platform	Intel Xeon Skylake	Tesla V100 SXM2	Stratix 10 280 FPGA	Plasticine
Software Framework	TF+AVX2	TF+cuDNN	Brainwave	Spatial
Achieved Clock Frequency (GHz)	2	1.38	0.25	1
Precision	f32	f16	f8	mix f8+16+32

Table 6. Performance comparison of DeepBench Inference

Benchmarks			Latency (ms)				Effective TFLOPS				Plasticine Speedup (x)			Power (W)
	H	T	Xeon Skylake	Tesla V100	BW	Plasticine	Xeon Skylake	Tesla V100	BW	Plasticine	Xeon Skylake	Tesla V100	BW	Plasticine
LSTM	256	150	15.75	1.69	0.425	0.0419	0.010	0.09	0.4	3.8	376.3	40.4	10.2	28.5
	512	25	11.50	0.60	0.077	0.0139	0.009	0.18	1.4	7.6	830.3	43.2	5.6	53.7
	1024	25	107.65	0.71	0.074	0.0292	0.004	0.59	5.7	14.4	3,686.6	24.3	2.5	97.2
	1536	50	411.00	4.38	0.145	0.1224	0.005	0.43	13.0	15.4	3,357.8	35.8	1.2	102.7
	2048	25	429.36	1.55	0.074	0.1060	0.004	1.08	22.7	15.8	4,050.6	14.6	0.7	104.5
GRU	512	1	0.91	0.39	0.013	0.0004	0.003	0.01	0.2	7.6	2,182.3	942.4	31.2	61.9
	1024	1500	3,810.00	33.77	3.792	1.4430	0.005	0.56	5.0	13.1	2,640.3	23.4	2.6	109.1
	1536	375	2,730.00	13.12	0.951	0.7463	0.004	0.81	11.2	14.2	3,658.3	17.6	1.3	114.6
	2048	375	5,040.00	17.70	0.954	1.2833	0.004	1.07	19.8	14.7	3,927.5	13.8	0.7	101.2
	2560	375	7,590.00	23.57	0.993	1.9733	0.004	1.25	29.7	15.0	3,846.4	11.9	0.5	117.2
	2816	750	25,850.00	55.48	1.987	4.7595	0.003	1.29	35.9	15.0	5,431.2	11.7	0.4	117.3

Table 7. Loop unrolling and vectorization parameters for spatial architectures

Benchmarks			Stratix 9 BW			Plasticine			
	H	T	$hu \times ru$	hv	rv	hu	hv	ru	rv
LSTM	256	150	6	400	40	6	1	4	64
	512	25				4		8	
	1024	25							
	1536	50							
	2048	25							
GRU	512	1				2			
	1024	1500							
	1536	375							
	2048	375							
	2560	375							
	2816	750							

5.2 RNN Performance Analysis

Table 6 shows the performance comparison of LSTM and GRU with various number of hidden units (H) and step size (T) over four platforms. Overall, CPU and GPU deliver very poor utilization of the compute resources. The latency of the CPU implementations are far beyond the time budget for real-time serving. Both Brainwave and Plasticine deliver promising latencies within 5ms for all problem sizes. Generally, Brainwave provides better performance on large RNN feature size, up to 2x better than Plasticine on GRU (H=2816). On the other hand, Plasticine outperforms Brainwave on small feature sizes with up to 30x better performance for GRU (H=512). While Brainwave provides very high TFLOPS for the large problem size, it suffers from resource underutilization for small problem sizes. Plasticine maintains very consistent FLOPS across all problem sizes.

Processor-Based Architectures For CPU experiments, the RNN kernels from TensorFlow itself is not multi-threaded. Since we focus on real-time serving of RNN applications, we use batch size of 1 for all of our benchmarks, which expose no parallelism outside the kernel level. Consequently, the machine is still very under utilized even with AVX2 instruction. Although one can implement RNN directly c++, the MVM size in RNNs are too small to benefit from multi-threading due to synchronization overhead. V100 with cuDNN library provides significant acceleration compared to CPU. Nevertheless, the latency is still beyond real-time serving budget at 7ms. This is because GPU are designed for throughput oriented, rather than latency sensitive workload. Provided that only BLAS3 routines are supported on GPU, small batch since can cause significant resource underutilization because all MVM are executed as Matrix Matrix Multiplications. In Table 6, V100 shows very poor performance on GRU (H=512). This is likely the result of the initialization overhead when only running step size of 1, which should not be timed. So overall, Plasticine achieves around 10-20x speedup over a V100 GPU. From our evaluation, neither processor-based architectures are suitable for providing low-latency serving on RNN applications.

Spatial Architectures Table 7 shows the selected design parameters for each problem size from Brainwave and Plasticine. For Brainwave, $hv \times rv$ corresponds to the number of tile engines in their matrix vector multiplier. hv is the native dimension in their paper, which is 400 for Stratix 10. rv is the lane width in their dot product engine. Large hv and rv improve the data to control ratio by amortizing the scheduling overhead over a large vectorized instruction, but aggravate the fragmentation especially for small feature size at 256 and 512. Plasticine uses hv of size 1, which prevents fragmentation in the H dimension. Also with $hv = 1$, all intermediate buffers become registers, as opposed to a register file of size hv in Brainwave. Plasticine collects additional gate-level, X, and H parallelism as well as pipelining at element-wise functions, all of which are timely scheduled in Brainwave. Brainwave used a single set of parameters for all problem sizes. Although they can potentially tune parameters for different problem sizes, doing so will incur re-synthesis and place-and- route on FPGA, which is an order of magnitude longer than Plasticine compilation times. In addition, to exhaust hardware resources with a smaller hv, they would have to increase the number of matrix vector tile engines $hu \times ru$. Consequently, decoders and schedulers associated with these units will drive up the control-to-data overhead and deliver less FLOPS for larger problem sizes.

The downside of a CGRA compared to FPGA is that it cannot perform arbitrary low-precision operations without sacrificing the area and power efficiency of hardened coarse-grain hardware primitives. Although Plasticine can support 8-bit element-wise operation with low-precision support described in Section 4.1, the reduction and accumulation are still performed in mix of 16 and 32 bit precisions. Hence, the peak FLOPS when performing mixed precision map-reduce is much less than peak element-wise 8 bit FLOPS. This is the major reason why Plasticine is slower on large problem sizes. Nonetheless, latest RNN architectures (Mnih et al., 2016; Britz et al., 2017; Wu et al., 2016) start to use fewer than 1024 hidden units. Therefore, being able to achieve high utilization consistently across all problem size is critical for a serving platform.

5.3 Area and Power Analysis

Table 4 shows the die area comparison of different platforms. While the GPU has a publicly-reported die area measurement (Markidis et al., 2018), Xeon Skylake and Stratix 10 only have estimated die areas based on their estimated transistor counts (Cutress, 2017). With the rough area estimates, we can see that while CPU has the smallest area in this case, the performance gap is too large even after we scale up to a 28-core server. The GPU also delivers bad performance per area mostly due to the low utilization of compute FLOPS. Stratix 10 delivers the best performance for large problem, but also has the largest die area estimates with estimated 30 billion transistors (Gazettabyte, 2015). Plasticine's die area is based on synthesis at 28nm, which is one generation older than the other platforms. With technology scaling, Plasticine should possess double the amount of compute and memory resources at 14nm for the same die area, which will roughly match Stratix 10 on all RNN problem sizes. At the same time, Plasticine is more than 2x smaller than Stratix 10, which will give at least 2x - 60x performance per area improvement for all problem sizes.

Table 4 shows the thermal design power (TDP) of the four platforms, which is the peak power achievable for any workload (Intel; 2018; Durant et al., 2017). Brainwave also reports a measured peak power for the given set of benchmarks of 125W. Table 6 shows the simulated power for Plasticine for each benchmark. Overall, the peak power among benchmarks for Plasticine is 118W, which is slightly less than the peak power compared to Brainwave. With technology scaling, Plasticine can match the performance of Stratix 10 on the large problem size with roughly the same power.

6 Related Work

Previously proposed serving platforms focus on exploiting data locality by mapping RNN cells onto spatial architectures. For example, Chang et al presented an FPGA-based implementation of an LSTM network (Chang et al., 2015). This approach works well for supporting small RNNs. However, for a large RNN, the weights would be too large to fit on-chip. As a result, the serving latency would be dominated by DRAM data loading. To address the issue of fitting RNN weights on-chip, several previous works (Han et al., 2016b; Wang et al., 2018; See et al., 2016; Narang et al., 2017) have studied the approaches for compressing RNN weights. For example, Han et al presented a compression scheme called DSD (Han et al., 2016b). It iteratively removes parameters in the weight matrices and retrains the sparse model to minimize the accuracy loss introduced by sparsity (Han et al., 2016b). With this compression scheme, Han et al were able to deploy an LSTM network containing 3.2 million parameters onto a modern FPGA without sacrificing accuracy. Compared to serving on CPU and GPU platforms, serving a sparse LSTM network on FPGA provides much lower latency and higher energy efficiency. However, we find that it could be hard to generalize this compression scheme for all the RNN tasks. RNNs are very flexible in terms of their model structures. Applying a DSD-like compression scheme to all the RNN models requires hand-tuning the compression heuristics for every model. To avoid hand-tuning, He et al proposed an approach that uses reinforcement learning techniques for automatic compression tuning (He et al., 2018). However, their approach focuses on compressing CNN tasks on edge devices, which may not be transferrable to the case of serving RNN tasks in data center. Observing that the sparsity-based compression schemes are still under active development, we choose to support compression schemes that focus on representing RNN weights using low-precision data format. Commercially available platforms such as Google TPU (Jouppi et al., 2017) and Microsoft BrainWave (Fowers et al., 2018) support these schemes.

7 Conclusion

In this paper, we describe techniques for performing cross-kernel optimization within RNN cells. We identify that by moving away from BLAS abstraction and focus on optimizing loop-level construct, we are able to achieve consistent hardware utilization when serving RNN cells of different sizes. We show that we are able to perform 10-20x performance improvement at a less advanced technology compared to the state-of-the-art GPU platform, and up to 30x compared to the state-of-the-art FPGA-based platform.

References

Dense linear algebra on gpus. https://developer.nvidia.com/cublas.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pp. 265–283, 2016.

Amazon. Ec2 f1 instances with fpgas now generally available. Technical report, Amazon, 2017. https://aws.amazon.com/blogs/aws/ec2-f1-instances-with-fpgas-now-generally-available/.

Britz, D., Goldie, A., Luong, M.-T., and Le, Q. Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906*, 2017.

Chang, A. X. M., Martini, B., and Culurciello, E. Recurrent neural networks hardware implementation on fpga. *arXiv preprint arXiv:1511.05552*, 2015.

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622. IEEE Computer Society, 2014.

Chen, Y.-H., Krishna, T., Emer, J. S., and Sze, V. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Cutress, I. The intel skylake-x review: Core i9 7900x, i7 7820x and i7 7800x tested. Technical report, AnandTech, 6 2017. https://www.anandtech.com/show/11550/the-intel-skylakex-review-core-i9-7900x-i7-7820x-and-i7-7800x-tested/.

Durant, L., Giroux, O., Harris, M., and Stam, N. Inside volta: The world's most advanced data center gpu. Technical report, NVIDIA, 5 2017. https://devblogs.nvidia.com/inside-volta/.

Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Adams, L., Ghandi, M., et al. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pp. 1–14. IEEE Press, 2018.

Gazettabyte, R. R. Altera's 30 billion transistor fpga. 6 2015. http://www.gazettabyte.com/home/2015/6/28/alteras-30-billion-transistor-fpga.html.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 243–254. IEEE, 2016a.

Han, S., Pool, J., Narang, S., Mao, H., Gong, E., Tang, S., Elsen, E., Vajda, P., Paluri, M., Tran, J., et al. Dsd: Dense-sparse-dense training for deep neural networks. *arXiv preprint arXiv:1607.04381*, 2016b.

He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., and Han, S. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 784–800, 2018.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Intel. Product specification. Technical report, Intel. https://ark.intel.com/products/codename/37572/Skylake.

Intel. An 787: Intel stratix 10 thermal modeling and management. Technical report, Intel, 8 2018. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an787.pdf.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pp. 1–12. IEEE, 2017.

Koeplinger, D., Prabhakar, R., Zhang, Y., Delimitrou, C., Kozyrakis, C., and Olukotun, K. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 115–127, June 2016. doi: 10.1109/ISCA.2016.20.

Koeplinger, D., Feldman, M., Prabhakar, R., Zhang, Y., Hadjis, S., Fiszel, R., Zhao, T., Nardi, L., Pedram, A., Kozyrakis, C., and Olukotun, K. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pp. 296–311, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192379. URL http://doi.acm.org/10.1145/3192366.3192379.

Liu, D. and Schafer, B. C. Efficient and reliable high-level synthesis design space explorer for fpgas. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Aug 2016. doi: 10.1109/FPL.2016.7577370.

Markidis, S., Der Chien, S. W., Laure, E., Peng, I. B., and Vetter, J. S. Nvidia tensor core programmability, performance & precision. *arXiv preprint arXiv:1803.04014*, 2018.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937, 2016.

Narang, S. and Diamos, G. Baidu deepbench. *GitHub Repository*, 2017.

Narang, S., Elsen, E., Diamos, G., and Sengupta, S. Exploring sparsity in recurrent neural networks. *arXiv preprint arXiv:1704.05119*, 2017.

Ouyang, J., Lin, S., Qi, W., Wang, Y., Yu, B., and Jiang, S. Sda: Software-defined accelerator for largescale dnn systems. Hot Chips 26, 2014.

Prabhakar, R., Zhang, Y., Koeplinger, D., Feldman, M., Zhao, T., Hadjis, S., Pedram, A., Kozyrakis, C., and Olukotun, K. Plasticine: A reconfigurable architecture for parallel paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 389–402. ACM, 2017.

Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P. Y., and Burger, D. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pp. 13–24, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4. URL `http://dl.acm.org/citation.cfm?id=2665671.2665678`.

See, A., Luong, M.-T., and Manning, C. D. Compression of neural machine translation models via pruning. *arXiv preprint arXiv:1606.09274*, 2016.

Wang, S., Li, Z., Ding, C., Yuan, B., Qiu, Q., Wang, Y., and Liang, Y. C-lstm: Enabling efficient lstm using structured compression techniques on fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 11–20. ACM, 2018.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.