SERVING RECURRENT NEURAL NETWORKS EFFICIENTLY WITH A SPATIAL ACCELERATOR

Tian Zhao [1] **Yaqi Zhang** [1] **Kunle Olukotun** [2]

ABSTRACT

Recurrent Neural Network (RNN) applications form a major class of AI-powered, low-latency data center workloads. Most execution models for RNN acceleration break computation graphs into BLAS kernels, which lead to significant inter-kernel data movement and resource underutilization. We show that by supporting more general loop constructs that capture design parameters in accelerators, it is possible to improve resource utilization using cross-kernel optimization without sacrificing programmability. Such abstraction level enables a design space search that can lead to efficient usage of on-chip resources on a spatial architecture across a range of problem sizes. We evaluate our optimization strategy on such abstraction with DeepBench using a configurable spatial accelerator. We demonstrate that this implementation provides a geometric speedup of 30x in performance, 1.6x in area, and 2x in power efficiency compared to a Tesla V100 GPU, and a geometric speedup of 2x compared to Microsoft Brainwave implementation on a Stratix 10 FPGA.

1 INTRODUCTION

Recurrent Neural Networks (RNNs) are a class of sequence models that plays a key role in low-latency, AI-powered services in datacenters (Fowers et al., 2018; Jouppi et al., 2017). In these services, the platforms assume that user requests come in individual samples and need to be served with very stringent latency window for real-time human computer interaction. An example of such workload is Google Translate, where inference happens concurrently when a user types. Despite its popularity, RNN model serving is hard to accelerate 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 a single computation 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 the 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, i.e. registers and on-chip scratchpads, tend to have single cycle access latency, the energy required to access these two types of memory would be widely different. Therefore, lack of cross-kernel fusion can lead to inefficient allocation of scratchpad resource and low energy-efficiency. Second, hardware accelerators tend to use large vectorization in compute and memory access to boost compute density when accelerating BLAS kernels. However, hardware accelerators tend to suffer from resource underutilization when the workload size is not multiples of the vector size. The utilization is worse with RNN applications that 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 (matrix-matrix) 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 from 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 strategies. First, we fuse all the gate functions with the element-wise, non-linear functions in the same time step. This way, all of our intermediate results are buffered in the registers as opposed to the scratchpads. Second, we spatially parallelize and pipeline the computation graph. We vectorize the inner-

---

[1]Department of Electrical Engineering, Stanford University, Stanford, USA [2]**AUTHORERR: Missing \sysmlaffiliation.** . Correspondence to: Tian Zhao <tianzhao@stanford.edu>, Yaqi Zhang <yaqiz@stanford.edu>, Kunle Olukotun <kunle@stanford.edu>.

loop of the tiled dot product to explore SIMD parallelism and fine-grain pipelining. We also explore tiled parallelism and coarse-grain pipelining by unrolling the outer loop nests based on the amount of available compute resources. These strategies exploit the gate-level parallelism in RNN cells, balance the pipelines of MVM and element-wise non-linear functions, and maximize the resource utilization when serving RNN models on different problem sizes. In addition, the entire pipeline is data-flow driven with no dynamic scheduling overhead.

We evaluate the proposed strategies by serving RNN tasks in DeepBench (Narang & Diamos, 2017) on the target spatial architecture. We implement the designs in Spatial (Koeplinger et al., 2018), a Domain-Specific-Language (DSL) that describes applications with nested loops and explicit hardware memory hierarchy. We choose Plasticine (Prabhakar et al., 2017), a coarse-grained reconfigurable architecture (CGRA), as the target spatial architecture. Furthermore, we propose augmentations to the Plasticine microarchitecture in order to support the mix-precision operations, which is critical for serving RNNs in real-time.

Finally, we compare the results to those obtained by serving DeepBench tasks on the state-of-the-art RNN serving platforms. We show that our implementation delivers consistently high FLOPS utilization across tasks of various sizes. We also 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 enable 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 a spatial architecture in order to enable low-precision operations.

3. Finally, we thoroughly evaluate 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 the RNN algorithms, the DSL and hardware platform used in this paper. Section 3 discusses

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.

the available RNN implementations on commercially available platforms. We then discuss the optimization strategies implemented in this work that address the inefficiency in these implementations. Section 4 discusses the architectural changes for supporting efficient RNN inference on the target spatial architecture. Section 5 details our evaluation methodology and experimental results. Section 6 discusses related works on available software and hardware optimization strategies for serving RNN applications. Section 7 offers concluding remarks.

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 evaluations 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:

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

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 an LSTM cell.

2.2 Spatial Reconfigurable Architectures

Spatial reconfigurable architectures, such as FPGAs and CGRAs, are gaining traction as data center accelerators for their energy efficiency (Amazon, 2017; Putnam et al., 2014; Ouyang et al., 2014). Compared to processor-based architectures, spatial architectures can reach high resource utilization by reconfiguring memory and compute based on the applications and computation requirements. In addition to exploiting parallelism, pipelining of data-flow graph in a spatial architecture provides high compute throughput. Nonetheless, the traditional low-level programming interface and long synthesis time of FPGA is the major obstacle for it to become a mainstream accelerator. As opposed to bit-level flat-interconnection in FPGAs, CGRAs are usually configured at higher level of granularity and contain a hierarchical interconnection network. In exchange, the reduction in flexibility in hardware translates to lowered routing overhead and higher clock frequency. The reduced routing overhead provides higher compute density and memory capacity, which makes CGRA an attractive platform to accelerate deep learning workloads. Due to the flexibility in mapping applications, spatial architectures often require 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 targets FPGAs and a previously proposed CGRA, Plasticine. A user describes applications in un-parallelized pattern-based loops with explicit memory hierarchies. Spatial automatically schedules, parallelizes, and pipelines arbitrary loop nests. To scale the memory bandwidth with parallelism, Spatial banks the scratchpad memories. To sustain the throughput of pipelining, Spatial also buffers the intermediate 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 an external DSE engine to balance the pipeline stages and saturate resource for different tasks on different hardware targets.

2.4 Plasticine

Plasticine is a CGRA that accelerates general nested loops in Spatial. It consists of primarily two types of units: a pattern compute unit (PCU) containing a single instruction multiple data (SIMD) pipeline optimized for accelerating vectorized map and reduction loops, and a pattern memory unit (PMU) containing configurable memory that to support banking and buffering schemes for various access patterns. Plasticine supports parallelizing and pipelining arbitrarily nested loops from Spatial. More architectural details will be explained in Section 4.

Symbol	Processor	Reconfigurable Hardware
▢	Kernel	Inner Loop
▢	Memory Hierachy	On-chip Scratchpad
▨	Register File	Register
↗ UnrollParam		Unrolling factor using multiple hardware compute blocks
⬭ op	Element-wise Operation	
▢	Outer Loop	
↔ VecParam	Vectorization parameter for AVX or SIMD instructions	

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.

3 RNN COMPUTATION ANALYSIS

In this section, we first discuss the limitation of BLAS-based LSTM on processor and spatial architectures. Next, we discuss our implementation of loop-based LSTM on spatial architectures. Table 2 contains specifications for symbols and parameters used in this section.

3.1 BLAS-based LSTM on Processor Architecture

Modern Machine Learning frameworks, e.g. TensorFlow (Abadi et al., 2016), divide the computation graph of an LSTM cell into BLAS kernels. Then, the BLAS kernel is accelerated by calling low-level optimized BLAS subroutines such as Intel BLAS Library on CPU and NVBLAS Library on GPU. Figure 1 (a) shows the computation graph of a `BasicLSTM` cell in TensorFlow. This implementation can lead to large memory footprint since all the intermediate results are materialized in memory. A common strategy to tackle the issue is through fusing blocked kernels. With TensorFlow's abstraction, this can only be achieved by expressing the entire RNN cell as an optimized kernel. For example, TensorFlow provides `LSTMBlockFusedCell` and `GRUBlockCell` modules, which are the fastest TensorFlow implementations of RNN cells for CPU. In practice, such implementation can provide significant performance improvement over the `BasicLSTM` implementation. However, it is still very hard to saturate CPU compute capacity, potentially due to the high synchronization overhead across threads. Figure 1 (b) shows the computation layout of TensorFlow with cuDNN library (Chetlur et al., 2014) on GPU. cuDNN is an NVIDIA GPU library for accelerating deep neural networks. To minimize the 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 of size H between the MVM kernel and the element-wise operations.
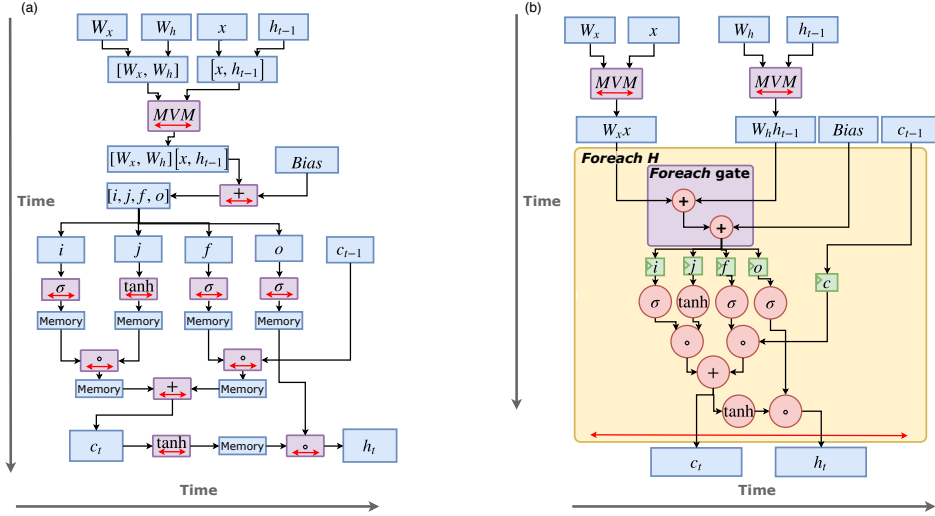
Figure 1. Compute and memory layout of TensorFlow `BasicLSTM` cell on CPU (a) and `CudnnLSTM` cell on GPU (b).

Compared to the `BasicLSTM` implementation, `CudnnLSTM` eliminates most of large intermediate memories. However, the MVMs 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 leads to serious underutilization of GPU.

Moreover, a processor-based architecture introduces large energy overhead of instruction decoding and scheduling. GPU especially suffers from its power-hungry, high-throughput memory hierarchy. For these reasons, both the CPU and GPU architectures 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. An FPGA has the flexibility of resizing MVM and VV units based on the application size. In addition, MVM and VV units can be implemented with hardware pipelines, which removes the instruction scheduling and control overhead on a processor-based architecture. The latest version of Intel Stratix 10 FPGA further boosts the compute power of FPGA with increasing number of hardened digital signal processing (DSP) blocks and on-chip memory capacity. Microsoft Brainwave (BW) (Fowers et al., 2018) is a state-of-the-art FPGA-based deep learning framework.

Figure 2 shows BW's compute and memory layout. In contrast to the CPU and GPU implementations, BW blocks the MVM along both row and column dimensions. It then 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, BW parallelizes the compute of multiple column tiles (ru, # MV Tiles in the original paper) of size $hv \times rv$ with multiple tiled engines, as shown in Figure 4 (a). Each tile engine contains hv (native dimension) number of dot product engines vectorized by rv (lanes) and achieves one tile per cycle throughput. Parallel tiles along the row dimension are then fed into a pipelined reduction and accumulation unit. Immediately after the accumulation, the multi-function units (MFUs) execute the element-wise operations on the hv vector chunk produced by the accumulator. Although BW's implementation still keeps the vectorized intermediate results, the size hv is much smaller than H in `BasicLSTM` cell. Nonetheless, with parallelization in ru, BW allocates lots of vectorized intermediate buffers that can still lead to energy inefficiency. BW performs one MVM operation in $\lceil \frac{H}{hv} \rceil \lceil \frac{R}{rv \cdot ru} \rceil$ iterations.

The MVM operations are executed on each gate of the LSTM sequentially. 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 a blocked floating-point format, where the vector of hv values share a single 5-bit exponent and have distinct signs and 2-5 bit mantissa for each value. As a result, they can achieve very dense low-precision compute and storage, 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 cannot be divided by hv and $rv \cdot ru$, Brainwave suffers from underutilization of the compute FLOPS, as shown in Figure 4 (a). The underutilization is worse with small problem sizes. In addition, BW computes

$W_x X$ and $W_h H$ separately rather than computing them with concatenated larger matrices, which can further aggravate the problem. This might be because BW's abstraction does not allow partial updates of an vector but only X is updated at the end of the step.

3.3 Loop-based LSTM

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

1. Constructing an LSTM cell's computation graph using BLAS subroutines introduces large intermediate buffers even when the kernels themselves are blocked. Each element on RNN cells' non-reduction dimension of the MVM (H) can be computed completely independently within one time step. This exposes the opportunity of fine-grain loop tiling and fusion across the entire LSTM kernel.

2. MVM is the computation bottleneck in serving RNN cells. Spatial architecture allows us to distribute most of the compute resource to MVM by parallelizing and pipelining MVM with element-wise operations.

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 the target spatial architecture without introduce too much overhead.

To address the issue of large intermediate buffers, we fine-grain tile and fuse MVM with non-linear functions. We refer to the computation for generating every single element in c_t and h_t as LSTM-1 operation, which can be computed independently in a single step. LSTM-1 is composed of four independent dot products of the row vectors of the weight matrices with the input vector immediately followed by the element-wise operations on output of the dot product. The resulting c and t vectors are produced by computing LSTM-1 operations for $H + D$ iterations.

As shown in Figure 3, each MVM unit is replaced by a MapReduce unit to compute the tiled dot product. Each MapReduce is vectorized by rv with pipelined map function followed by a pipelined reduction tree. ru is the number of parallel MapReduce units. Results of ru MapReduce blocks are reduced and accumulated with another reduction tree (not shown in Figure). Next, the dot product result is passed through a chain of function units for executing bias add and non-linear functions. Dot products, bias adds, and non-linear functions of the four gates can also be parallelized. Finally, the results of the four gates are pipelined through a set of function units for element-wise operation in LSTM cell. At the outer loop, LSTM-1 runs for $\frac{H}{hu}$ iterations,
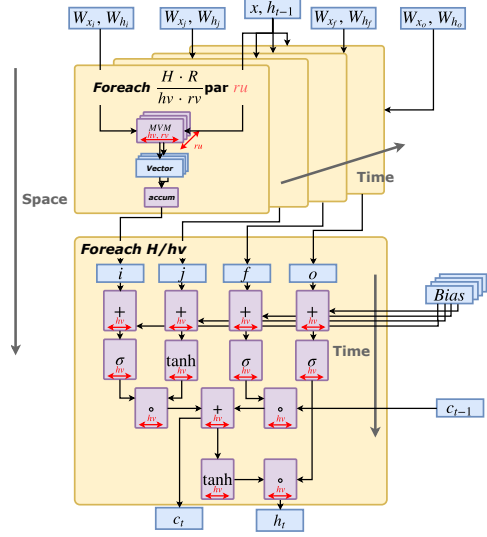

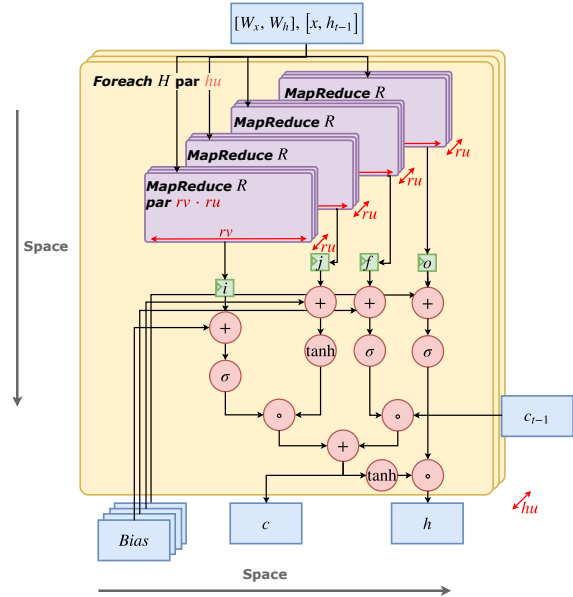Figure 2. Compute and memory layout of LSTM in Brainwave.

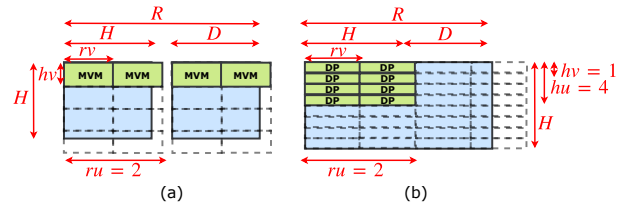Figure 3. Compute and memory layout of a loop-based LSTM design.

Figure 4. Fragmentation in an MVM-based design (a) and a loop-based design (b) in MVM.

```
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   // Concatenated weights [Wx,Wh] for each gate in 2-D SRAMs
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      val cNew = i*j + c(ih) * f
31      c(ih) = cNew
32      xh(ih+D) = tanh(cNew) * o
33    }
34  }
```

Figure 5. Example of LSTM in Spatial.

where hu is the number of parallel LSTM-1 implementations.

In the loop-based design, all intermediate buffers are scalars as supposed to vectors. Regarding utilization, the loop-based LSTM design suffers from less underutilization due to unaligned problem size compared to the tiled MVM approach in BW. Figure 4 shows sources of such underutilizations. An MVM approach design would suffer from 2-D fragmentation on both the H and D dimensions (Figure 4 (a)), whereas the loop-based design only suffers from 1-D fragmentation on the R dimension (Figure 4 (b)).

Figure 5 shows a loop-based LSTM design implemented in Spatial. **Foreach** is a loop construct with a lambda that takes loop iterator as input. **Reduce** is a construct that executes MapReduce by taking a map function followed by a reduction function. User declare explicit on-chip scratchpads and registers with **SRAM** and **Reg**. To enable fine-tuning an RNN application, we exposes loop vectorization factor rv, hv and unrolling factors hu, ru.

4. PLASTICINE SPECIALIZATION FOR RNN SERVING

To show efficient execution of the loop and parallel pattern constructs, we map our implementation onto a spatial architecture, Plasticine. **Foreach** at Line $17, 19$ and **Reduce** at Line $22, 23$ are mapped to PCUs on Plasticine. When the application size is small, these constructs are executed using pipelined SIMD lanes within a single PCU. When the application size is large, multiple PCUs can be used to parallelize and pipeline the dot product across PCUs. Element-wise operations can be executed in a deep pipeline formed by chaining multiple PCUs.

To fit an RNN's weights on-chip, we execute our application with low-precision arithmetics. In this section, we propose the necessary micro-architectural changes to support low-precision arithmetics on Plasticine. We also discuss architectural parameter selection for Plasticine to serve RNN applications efficiently.

4.1 Mixed-Precision Support

Previous works (Fowers et al., 2018; Jouppi et al., 2017) have shown that low-precision inference can deliver promising performance improvements without sacrificing accuracy. In the context of reconfigurable architectures such as FP-GAs, low-precision inference not only increases compute density, but also reduces required on-chip capacity for storing weights and intermediate data.

To support low-precision arithmetics without sacrificing coarse-grained reconfigurability, we introduce two low-precision struct types in Spatial: a tuple of 4 8-bit and 2 16-bit floating-point numbers, `4-float8` and `2-float16` respectively. Both types packs multiple low-precision values into a single precision storage. 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 a Plasticine PCU. Each FU supports both floating-point and fix-point operations. When mapping applications on Plasticine, the inner most loop body is vectorized across the lanes of the SIMD pipeline, and different operations of the loop body are mapped to different stages. Each pipeline stage contains a few pipeline registers (PRs) that allow propagation of live variables across stages. Special cross-lane connections as shown in red in Figure 6 enable 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 1st and 3rd stages are element-wise, low-precision operations that multiply and add 4 8-bit and 2 16-bit values, respectively. The 2nd and 4th stages rearrange low-precision values into two registers, and then pad them to higher precisions. The 5th stage reduces the two 32-bit value to a single 32-bit value using the existing add operation. 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.
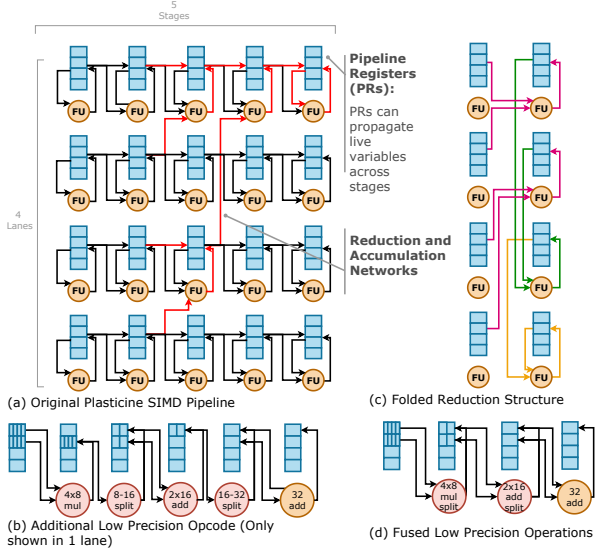
Figure 6. Plasticine PCU SIMD pipeline and low-precision support. Red circles are the new operations. Yellow circles are the original opertaions in Plasticine. In (d) the first stage is fused $1^{st}, 2^{nd}$ stages, and the second stage is fused $3^{nd}, 4^{th}$ stages of (b).

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

In a more aggressive specialization, we can fuse the multiply and rearrange into the same stage. We also fuse the first low-precision reduction with the next rearrange as shown in Figure 6 (d). 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 $4\#_{LANE}$ 8-bit values using 4 stages. All the operations are completed in $2 + \log_2(\#_{LANE}) + 1$ cycles.
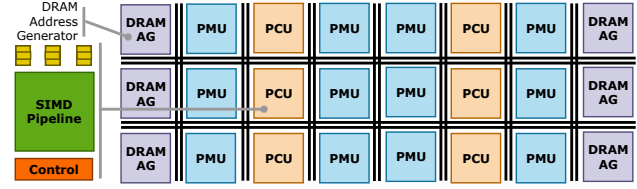

Figure 7. Variant configuration of Plasticine for serving RNN.

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 2:1. 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 the real-time RNN serving tasks on various platforms. We start with the 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 the LSTM and GRU tasks from Baidu DeepBench as our benchmarks. We evaluate the benchmarks across processor-based architectures including CPU and GPU, and spatial architectures including 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 summarizes application configurations of each platform.

CPU We implement the applications in Tensor-Flow 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 use 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 use 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 has FPGA back-end support, Stratix 10 is not commercially available at the time of the 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 on-chip scratchpad capacity of a Stratix 10 FPGA. The exact configuration of Plasticine is shown in Table 3. In order to minimize the overhead of low-precision support, Plasticine only supports 8-bit, 16-bit, and 32-bit element-wise operations, and mixed precision reduction operation. For our evaluation, the element-wise operations are performed in 8-bit precision, 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 measure the performance, we use a cycle accurate simulator for Plasticine. We modified the simulator to model the proposed micro-architectural changes to support low-precision operations. We use the area and power of individual CUs and network switches from the original Plasticine paper, and compute total area of configuration shown in Table 3. As discussed in Section 4, we reduce the number of stages in PCU from 6 stages to 4 stages with fused low-precision operations and folded reduction tree. Low preicision function units can be used to compose full precision units. We conservatively estimate the area and power of PCU stays the same with our proposed change and reduced two stages. We also increase the PMU to PCU ratio to better match 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. For power calculations, we generate activity tracing of the CUs from simulation, and then integrate with characterized power of individual PCU to compute the total power. The power and area characterizations are based off synthesis at 28nm technology at 1GHz clock frequency.

Brainwave Finally, we also compared our results to Microsoft Brainwave framework. For this evaluation, we compare to Brainwave implemented on top of Intel Stratix 10 FPGA. Brainwave is synthesized at 250MHz and all opera-

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.

tions are performed in blocked low-precision floating-point format described in section 3.3.

5.2 RNN Performance Analysis

Table 6 shows the performance comparison of LSTM and GRU with various numbers of hidden units (H) and step sizes (T) over the four platforms. Overall, both CPU and GPU significantly underutilize the available compute FLOPS. In addition, they cannot meet the latency requirement for real-time serving for all problem sizes. Both BW and Plasticine deliver promising latencies within 5ms for all problem sizes. When serving very large RNNs, BW provides better performance with up to 2x better than Plasticine on the largest GRU (H=2816). When serving small and medium size RNNs, Plasticine performs better than BW with up to 30x better performance on small GRU (H=512). We also observe that Plasticine delivers consistent FLOPS when serving all the 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.

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	blocked precision	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	0.37	3.8	376.3	40.4	10.2	28.5
	512	25	11.50	0.60	0.077	0.0139	0.009	0.18	1.37	7.6	830.3	43.2	5.6	53.7
	1024	25	107.65	0.71	0.074	0.0292	0.004	0.59	5.68	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	13.01	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	22.62	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	0.25	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	4.98	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	11.17	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	19.79	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	29.69	15.0	3,846.4	11.9	0.5	117.2
Geometric Mean											2,529.3	29.8	2.0	

Table 7. Loop unrolling and vectorization parameters for spatial architectures.

BENCHMARKS			STRATIX 9 BW			PLASTICINE			
	H	T	ru	hv	rv	hu	hv	ru	rv
LSTM	256	150				6		4	
	512	25							
	1024	25				4			
	1536	50							
	2048	25							
GRU	512	1	6	400	40	1	8		64
	1024	1500							
	1536	375				2			
	2048	375							
	2560	375							
	2816	750							

Consequently, the machine is still very underutilized even with AVX2 instruction. Although one could implement RNN directly in c++, the MVM sizes in RNNs are too small to benefit from multi-threading due to the synchronization overhead. V100 with cuDNN library provides significant acceleration compared to CPU. Nevertheless, the latency is still high. This is because GPUs are designed for throughput oriented rather than latency sensitive workloads. Provided that the library is based on BLAS3 routines, which are matrix-matrix operation, MVMs in RNN serving suffer from significant resource underutilization. In Table 6, V100 shows very poor performance on GRU (H=512). This is likely due to the initialization overhead which should not be timed. 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 for BW and Plasticine. On Stratix 10, BW uses 6 tile engines (ru) with native dimension of 400 (hv) and 40 lanes (rv). Large hv and rv improve the data-to-control ratio by amortizing the scheduling overhead over a large vectorized instruction. However, this design choice aggravates the underutilization for small RNN feature sizes at 256 and 512. Our implementation effectively uses hv of size 1 by performing dot product instead of MVM, which prevents fragmentation in the H dimension. With $hv = 1$, all the intermediate buffers are

stored in registers. In contrast, BW uses register files of size hv. In addition, our proposed implementation captures additional gate-level, X, and H parallelism as well as pipelining at element-wise functions. In contrast, BW schedules these operations in time and dispatches corresponding instructions to drive the compute units.

A CGRA is less flexible than an FPGA when performing arbitrary low-precision operations. In this example, we increase memory density of Plasticine by supporting quantile precisions as described in Section 4.1. All weights are stored in 8 bit format, so as the multiplication operations of MVM. The reduction and accumulation operations are implemented in mix of 16 and 32 bit precisions. Hence, the peak FLOPS when performing mixed precision map-reduce is much less than the peak FLOPS for blocked low-precision format in BW. As a result, Plasticine performs worse than BW on the large RNNs.

In addition, Plasticine delivers very consistent FLOPS for different problem sizes. For small problem size, the dot product can be fully unrolled with $rv * ru$. Therefore, we can increase hu to explore additional parallelism across the hidden units. For large problem size, dot product becomes the bottleneck of the pipeline. Hence, we reduce hu and increase ru to balance the throughput between dot product and element-wise operations. In this example, BW uses a single set of parameters for all problem sizes. Although one can potentially tune parameters for different problem sizes, doing so will incur re-synthesis and place-and-route on an FPGA, which is an order of magnitude longer than the compilation time needed for a CGRA design. In addition, to exhaust hardware resources with a smaller hv, one would have to increase the number of matrix vector tile engines $hu \times ru$ in BW. As a result, decoders and schedulers associated with these units will drive up the control-to-data overhead and deliver less FLOPS for larger problem sizes.

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 measure-

ment (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 the large RNNs, but with the largest die area estimates of 30 billion transistors (Gazettabyte, 2015). Plasticine's die area is based on the synthesis results at 28nm, which is one generation older than all 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's performance on all the RNN problem sizes. At the same time, Plasticine is more than 2x smaller than Stratix 10, which could also contribute 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 workloads (Intel; 2018; Durant et al., 2017). BW 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 BW.

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 datacenter. 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 a set of 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 a geometric speedup of 2x compared to the state-of-the-art FPGA-based platform.

8   ACKNOWLEDGEMENT

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/.

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.

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.