Efficient Phase-Functioned Real-time Character Control in Mobile Games: A TVM Enabled Approach

Haidong Lan*
Wenxi Zhu*
Tencent AI Lab
Shenzhen, China

Du Wu
Shenzhen Institue of Advanced Technology, Chinese Academy of Science
Shenzhen, China

Qian Qiu, Honglin Zhu, Jingjing Zhao, Xinghui Fu
Tencent AI Lab
Shenzhen, China

Wei Liu
Southern University of Science and Technology
Shenzhen, China

Jintao Meng†
Shenzhen Institue of Advanced Technology, Chinese Academy of Science
Shenzhen, China
jt.meng@siat.ac.cn

Minwen Deng†
Tencent AI Lab
Shenzhen, China
danierdeng@tencent.com

ABSTRACT

In this paper, we propose a highly efficient computing method for game character control with phase-functioned neural networks (PFNN). The primary challenge to accelerate PFNN on mobile platforms is that PFNN dynamically produces weight matrices with an argument, *phase*, which is individual to each game character. Therefore existing libraries that generally assume frozen weight matrices are inefficient to accelerate PFNN. The situation becomes even worse when multiple characters are present. To address the challenges, we reformulate the equations and leverage the deep learning compiler stack TVM to build a cross-platform, high-performance implementation. Evaluations reveal that our solutions deliver close-to-peak performance on various platforms, from high-performance servers to energy-efficient mobile platforms. This work is publicly available at https://github.com/turbo0628/pfnn_tvm.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

Machine Learning, Phase-Functioned Neural Network, PFNN, TVM, Ansor, High Performance Computing, Deep Learning Compiler

ACM Reference Format:
Haidong Lan, Wenxi Zhu, Du Wu, Qian Qiu, Honglin Zhu, Jingjing Zhao, Xinghui Fu, Wei Liu, Jintao Meng, and Minwen Deng. 2022. Efficient Phase-Functioned Real-time Character Control in Mobile Games: A TVM Enabled Approach. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3545008.3545095

1 INTRODUCTION

Developing real-time character controllers has been challenging in video games. Traditional methods pack many motion templates into the game system's motion library, where motion matching algorithms are conducted in order to retrieve the corresponding motion templates. The game character is rendered according to the selected motion template subsequently. In recent years, game developers have largely refined character motion control, which inevitably expands the motion libraries to unbearable sizes, especially for mobile games. A novel class of algorithms, namely the phase-functioned neural networks (PFNN) [8], emerges to outperform the motion libraries approaches in terms of more natural motions and less demands on compute and storage resources. The PFNN algorithm and its derivatives [21] [14] [15] [16] successfully leveraged neural networks to diminish the need for the space-consuming motion library, and is therefore widely adopted by newly developed games.

PFNN has never been the first attempt to solve the character control problem with neural networks. Its success is owing to the introduction of the phase argument, which represents the status of the game character. The original PFNN algorithm trains neural network weights for 50 distinct phase values that equally divide the $[0, 2\pi)$ number range. For inference, PFNN proposed three protocols to generate the weight matrices, say the "constant", "linear" and "cubic" modes. The "constant" mode select the most corresponding set of weights according to the phase argument. Therefore, it has to keep all 50 sets of weights in memory for best inference latency. The "linear" mode preserves 10 weight sets and generates respective weight matrices with a linear spline interpolation function for arbitrary phase values. The "cubic" mode only uses 4 weights that corresponds to phase values $\{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\}$. The interpolation function is more complex while memory consumption is minimized.

There are three challenges to efficiently implement the PFNN algorithm in mobile games. (1) Mobile games have strict restrictions on memory usage. The "cubic" mode could meet the memory requirements while inference latency is compromised due to the more

*Both authors contributed equally to this research.
†Jintao Meng and Minwen Deng are the corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPP '22, August 29-September 1, 2022, Bordeaux, France

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545095

complex weight interpolation function. We need highly efficient compute method for the "cubic" mode. (2) There are several characters concurrently present in real game environments. As the weight matrices are generated with phase argument which is individual to each game character, we cannot compute for multiple characters in a single batch. The memory consumption and compute latency would grow linearly with the number of game characters if implemented with existing neural network inference libraries. (3) Mobile games are generally developed in a cross-platform workflow, where developers prototype and test on x86-based Linux, Windows or Mac workstations and deploy on Arm-based iOS and Android mobile devices.

To address the above challenges, we propose TVM-enabled efficient compute methods to accelerate the PFNN algorithm's cubic mode, especially when there are multiple concurrent game characters. Our contribution is three-fold:

- We transform the mathematical formulas of PFNN into a hardware-friendly form, especially for multiple concurrent game characters. The algorithmic optimization delivers an average acceleration ratio of **2.94x** implemented with Eigen [6].
- We develop a cross-platform inference library with TVM. The implementation is capable of supporting eight concurrent game characters on a single mobile CPU core with inference latency within **2ms**. Our methods outperform the Eigen baseline implementation by factors of **6.16x** to **7.07x** with eight game characters on server and mobile test platforms.
- We implement schedulers in both manual and automatic approaches. We modeled the compute and memory access complexity and analyzed the performance figures against the hardware roofline. This work can also serve as a case study.

The rest of this paper is organized as follows: Section 2 briefly provides the background of PFNN and tensor program compilers. Section 3 describes our algebraic optimizations and complexity analysis. TVM-enabled approaches are discussed in Section 4 and evaluated in Section 5. Section 6 concludes this paper.

2 BACKGROUND

2.1 Phased-Functioned Neural Network

PFNN [8] employs a multi-layer perception (fully-connected layers) as its neural network backbone. In each perception layer, PFNN dynamically calculates the corresponding weight matrix by interpolating from several static weight matrices. Specifically, the constant mode is configured to use 50 weight matrices $\beta = \{\alpha_0, \alpha_1, \ldots, \alpha_{49}\}$, the interpolation function Θ is defined as

$$\Theta(p;\beta) = \alpha_k$$

where

$$\alpha_k = \left\lfloor \frac{50p}{2\pi} \right\rfloor$$

The linear mode is configured to use 10 weight matrices $\beta = \{\alpha_0, \alpha_1, \ldots, \alpha_9\}$, the interpolation function Θ is defined as

$$\Theta(p;\beta) = (1-\mathbf{w})\cdot\alpha_{k_0} + \mathbf{w}\cdot\alpha_{k_1}$$

where

$$\mathbf{w} = \frac{10p}{2\pi}\ (mod\ \ 1)$$

and

$$k_n = \left\lfloor \frac{10p}{2\pi} \right\rfloor + n - 1\ (mod\ 10)$$

The cubic is defined similarly with 4 weight matrices $\beta = \{\alpha_0, \alpha_1, \alpha_2, \alpha_3\}$ and a more complex interpolation function.

$$\begin{aligned}\Theta(p;\beta) = \alpha_{k_1} & \\ &+\mathbf{w}(\frac{1}{2}\alpha_{k_2} - \frac{1}{2}\alpha_{k_0}) \\ &+\mathbf{w}^2(\alpha_{k_0} - \frac{5}{2}\alpha_{k_1} + 2\alpha_{k_2} - \frac{1}{2}\alpha_{k_3}) \\ &+\mathbf{w}^3(\frac{3}{2}\alpha_{k_1} - \frac{3}{2}\alpha_{k_2} + \frac{1}{2}\alpha_{k_3} - \frac{1}{2}\alpha_{k_0})\end{aligned} \tag{1}$$

where

$$\mathbf{w} = \frac{4p}{2\pi}\ (mod\ \ 1)$$

and

$$k_n = \left\lfloor \frac{4p}{2\pi} \right\rfloor + n - 1\ (mod\ 4)$$

We should note that, the "cubic" mode is generally considered to be expensive in terms of compute overhead. With this regard, all the PFNN derivations [21] [14] [15] [16] adopted the "linear" mode to balance compute and storage overhead. In this paper, however, we will demonstrate that the "cubic" mode could be very efficient, even when there are multiple concurrent game characters.

2.2 Tensor Program Compilers

In recent years, domain-specific compilers are becoming ubiquitous to accelerate deep learning workloads which are often referred to as *tensor programs*. Such compilers are designed to reduce manual efforts in delivering high performance across diverse hardware backends and tensor shapes. State-of-the-art works include Tiramisu [1], Tensor Comprehensions [18], MLIR [10], Halide [13] and its deep learning derivation [11] and TVM [2]. The tensor program compilers have their own hierarchies of intermediate representations (IR), on which optimization passes are conducted. To be more specific, tensor programs are lowered to the IRs where compute graphs and tensor operators are optimized at each level respectively.

Compared with conventional approaches of manually developed algebra libraries (OpenBLAS[20], MKL[19] and cuDNN[4]) or generated libraries with simple hand-written rules or templates (Eigen[6], libxsmm [7], BLIS[17] and many others), the tensor program compilers are far more flexible and productive because they only require a user to provide the compute graph definitions, namely the tensor programs. Despite the tensor program itself, optimization primitives are added in order to guide the compiler to create efficient schedulers. Multiple recent works are focused on automatically generating efficient schedulers. The AutoTVM [3] adopted a statistical cost model to accelerate operator search. The WoodPecker-DL [12] employed genetic algorithm and reinforcement learning to optimize the compute plan. The Ansor [22] took a hierarchical approach to construct the search space upon the compute graph that could efficiently strip most local optimal saddle points.

2.3 Novelty

The original PFNN work [8] proposed a straightforward implementation on desktop CPUs. A following acceleration work [9] implemented PFNN on CUDA-enabled GPUs. Both works are devoted to implement the algorithm rather than properly optimize with respect to specific hardware. We should note that previous works seldom adopt neural network inference library, the reason is that the weight matrices are dynamically generated. That said, the dedicated libraries cannot outperform BLAS libraries. Our approach is novel in three aspects compared with the two works: (1) This is the first efficient implementation that enables PFNN's "cubic" mode for mobile devices. (2) Our method is orders of magnitude faster by supporting eight characters at a latency below 2ms with one single mobile CPU core, while previous works can only compute for one character at the same latency level. (3) We deliver portable performance across multiple hardware backends by leveraging tensor program compilers.

3 ALGEBRAIC OPTIMIZATIONS

The PFNN's algebraic form is inefficient due to two reasons: (1) The form of Equation 1 can be further simplified. (2) The interpolation and neural network computation employs scalar-matrix and vector-matrix multiplications to implement, where efficiency can be promoted by reformulating the pattern to matrix-matrix multiplication. Hence, we perform algebraic transformations to simplify the equations and enable matrix-matrix multiplications with multiple concurrent game characters.

3.1 Equation Simplifications

We point out that variables $\mathbf{w}$ and k_n in Equation 1 are constant during the neural network inference procedure, as they only depend on the input argument p (phase). The weight matrix subscripts k_n also arrange the matrix order, which impedes the efficient memory layout of the α_i weight matrices.

We first rewrite the equation by combining the terms of the weight matrices $\alpha_0, \alpha_1, \alpha_2, \alpha_3$.

$$\begin{aligned}\Theta(p;\beta) &= (\mathbf{w}^2 - \frac{\mathbf{w}^3}{2} - \frac{\mathbf{w}}{2})\cdot\alpha_{k_0} + (\frac{3\mathbf{w}^3}{2} - \frac{5\mathbf{w}^2}{2} + 1)\cdot\alpha_{k_1} \\ &\quad + (2\mathbf{w}^2 - \frac{3\mathbf{w}^3}{2} + \frac{\mathbf{w}}{2})\cdot\alpha_{k_2} + (\frac{\mathbf{w}^3}{2} - \frac{\mathbf{w}^2}{2})\cdot\alpha_{k_3} \\ &= \begin{bmatrix}\alpha_{k_0} & \alpha_{k_1} & \alpha_{k_2} & \alpha_{k_3}\end{bmatrix} \begin{bmatrix}\mathbf{w}^2 - \frac{\mathbf{w}^3}{2} - \frac{\mathbf{w}}{2} \\ \frac{3\mathbf{w}^3}{2} - \frac{5\mathbf{w}^2}{2} + 1 \\ 2\mathbf{w}^2 - \frac{3\mathbf{w}^3}{2} + \frac{\mathbf{w}}{2} \\ \frac{\mathbf{w}^3}{2} - \frac{\mathbf{w}^2}{2}\end{bmatrix} \\ &= \begin{bmatrix}\alpha_{k_0} & \alpha_{k_1} & \alpha_{k_2} & \alpha_{k_3}\end{bmatrix} \begin{bmatrix}\mathbf{W}_0 \\ \mathbf{W}_1 \\ \mathbf{W}_2 \\ \mathbf{W}_3\end{bmatrix}\end{aligned} \tag{2}$$

Consider the fact that $k_n \in \{0, 1, 2, 3\}$ is simply an arrangement of the indices, we further transform this equation by arranging the term values $\mathbf{W}_j$ rather than α_i.

$$\begin{aligned}&\Theta(p;\beta) \\ &\quad = \begin{bmatrix}\alpha_0 & \alpha_1 & \alpha_2 & \alpha_3\end{bmatrix} \begin{bmatrix}\mathbf{W}_{t_0} \\ \mathbf{W}_{t_1} \\ \mathbf{W}_{t_2} \\ \mathbf{W}_{t_3}\end{bmatrix} \\ &\quad = \beta\cdot\mathcal{W}\end{aligned} \tag{3}$$

where t_i denotes the inverse index with respect to k_i:

$$t_{k_i} \leftarrow i$$

With this regard, the order of α_i is fixed. We can permute the weight matrices in an efficient layout when loading from files. Consider the common scenario that multiple game characters are present, for each character c, we have

$$\Theta(p_c;\beta) = \beta\cdot\begin{bmatrix}\mathbf{W}_{c,t_0} \\ \mathbf{W}_{c,t_1} \\ \mathbf{W}_{c,t_2} \\ \mathbf{W}_{c,t_3}\end{bmatrix} = \beta\cdot\mathcal{W}_c$$

Take the interpolation equation into the neural network equation, the layer-wise formula for each character becomes:

$$\Phi(x_c;\beta,b) = \Theta(p_c;\beta)\cdot x_c + \Theta(p_c;b) = (\beta\cdot\mathcal{W}_c)\cdot x_c + b\cdot\mathcal{W}_c \tag{4}$$

We can stack the feature vectors for all C game characters, yielding

$$\Phi(\mathbf{X};\beta,b) = \Theta(p;\beta)\cdot\mathbf{X} + \Theta(p;b) = (\beta\cdot\mathcal{W})\cdot\mathbf{X} + b\cdot\mathcal{W} \tag{5}$$

where

$$\mathcal{W} = \begin{bmatrix}\mathbf{W}_{t_{0,0}} & \mathbf{W}_{t_{1,0}} & \cdots & \mathbf{W}_{t_{C,0}} \\ \mathbf{W}_{t_{0,1}} & \mathbf{W}_{t_{1,1}} & \cdots & \mathbf{W}_{t_{C,1}} \\ \mathbf{W}_{t_{0,2}} & \mathbf{W}_{t_{1,2}} & \cdots & \mathbf{W}_{t_{C,2}} \\ \mathbf{W}_{t_{0,3}} & \mathbf{W}_{t_{1,3}} & \cdots & \mathbf{W}_{t_{C,3}}\end{bmatrix}$$

is a coefficient matrix of shape $4\times C$ which is derived from the phase argument (see Equation 2). We define it as the input argument which replaces the phase argument p. Here we have simplified the interpolation operation to regular matrix arithmetics. However, this form cannot directly invoke standard GEMM routines. We further transform the equation by leveraging the property that dot multiplications are right associative. We multiply $\mathcal{W}$ against $\mathbf{X}$ prior to β:

$$\Phi(\mathbf{X};\beta,b) = \beta\cdot(\mathcal{W}\cdot\mathbf{X}) + b\cdot\mathcal{W} \tag{6}$$

The Equation 6 indicates that interpolation arithmetics is conducted on the input vector $\mathbf{X}$ rather than weight matrices. A subsequent standard matrix-matrix multiplication is conducted to compute for a batch of game characters. As the input vector $\mathbf{X}$ is generally small, the efficient matrix multiplication constitutes the gross of computing load.

3.2 Complexity Analysis

We analysis compute and memory complexity of Equation 5 and 6, respectively. For convenience, we list the shapes of tensors as follows: (1) Input vectors $\mathbf{X}$ have shape $C\times K$, where C denotes the character number (batch size) and K denotes feature vector length. (2) Weight matrices α_i have identical shape of $K\times N$. The combo of

Algorithm 1: Non-fused Pseudocode for Equation 5

```
1:  for c = 1 to C do
2:    for k = 1 to K do
3:      for n = 1 to N do
4:        S = 0
5:        for i = 1 to 4 do
6:          S = S + W_{c,i} · α_i(k, n)
7:        end for
8:        I_c(k, n) = S
9:      end for
10:   end for
11: end for
12: for c = 1 to C do
13:   Below is equivalent to a GEMV call.
14:   for n = 1 to N do
15:     S = 0
16:     for k = 1 to K do
17:       S = S + I_c(k, n) · X_c(k)
18:     end for
19:     Φ(c, n) = S
20:   end for
21: end for
```

Algorithm 2: Fused Pseudocode for Equation 5

```
1:  for n = 1 to N do
2:    for c = 1 to C do
3:      S = 0
4:      for k = 1 to K do
5:        T = 0
6:        for i = 1 to 4 do
7:          T = T + W_{c,i} · α_i(k, n)
8:        end for
9:        S = S + T · X_c(k)
10:     end for
11:     Φ(c, n) = S
12:   end for
13: end for
```

weight matrices β has shape of $4 \times K \times N$ (3) Output vectors have shape $C \times N$. These notations will be used in the rest of this paper.

We calculate the computing complexity in the above equations by counting the minor number of floating-point arithmetic operations (FLOP). The arithmetics include interpolation and matrix multiplications. All of our target processors have Fused Multiply-Add (FMA) instructions. Therefore, we only count one operation where multiplication is followed by addition. Memory access complexity is calculated by the number of elements that are accessed, which is also known as *memory footprint*. Read and write of the same element are counted as two accesses as long as it cannot hold in processor cache or register files.

3.2.1 Interpolation on Weight Matrices. Equation 5 conduct the interpolation arithmetics $\beta \cdot \mathcal{W}$ by multiplying every element in $\mathcal{W}$ against α_i, and accumulate subsequently. The interpolation calculation can be decomposed into detailed compute steps with the following expression:

$$\beta \cdot \mathcal{W} = \begin{bmatrix} \mathbf{I}_0 \\ \mathbf{I}_1 \\ \dots \\ \mathbf{I}_C \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{3} \mathcal{W}_{0,i} \cdot \alpha_i \\ \sum_{i=0}^{3} \mathcal{W}_{1,i} \cdot \alpha_i \\ \dots \\ \sum_{i=0}^{3} \mathcal{W}_{C,i} \cdot \alpha_i \end{bmatrix}$$

It takes $4 \times C \cdot K \cdot N$ FLOPs and accesses at least $4 \times K \times N$ elements in α_i and $C \times K \times N$ elements to write $\mathbf{I}_i$. For the subsequent multiplication against the input vectors, the decomposed expression is

$$(\beta \cdot \mathcal{W}) \cdot \mathbf{X} = \begin{bmatrix} \mathbf{I}_0 \cdot X_0 \\ \mathbf{I}_1 \cdot X_1 \\ \dots \\ \mathbf{I}_C \cdot X_C \end{bmatrix}$$

The multiplication can be implemented with C consecutive invokes to an optimized GEMV (matrix-vector multiplication) routine. It employs a total of $C \times K \times N$ FMA FLOPs. For memory complexity, $C \times K \times N$ elements are accessed to read the $\mathbf{I}_c$ matrices, $C \cdot K$ and $C \cdot N$ elements are accessed to load inputs and write outputs, respectively.

Overall, the computing complexity is:

$$5 \cdot C \cdot K \cdot N \tag{7}$$

and the memory access complexity is

$$(2C + 4) \cdot K \cdot N + C \cdot (K + N)$$

However, we should be aware that interpolation and multiplication arithmetics can be fused in order to reduce memory access complexity. We illustrate the non-fused and fused pseudocode in Algorithm 1 and 2, respectively. The fused version computes $\mathbf{I}_c$ on the fly rather than storing into memory. Therefore, the overall memory access complexity can be reduced to

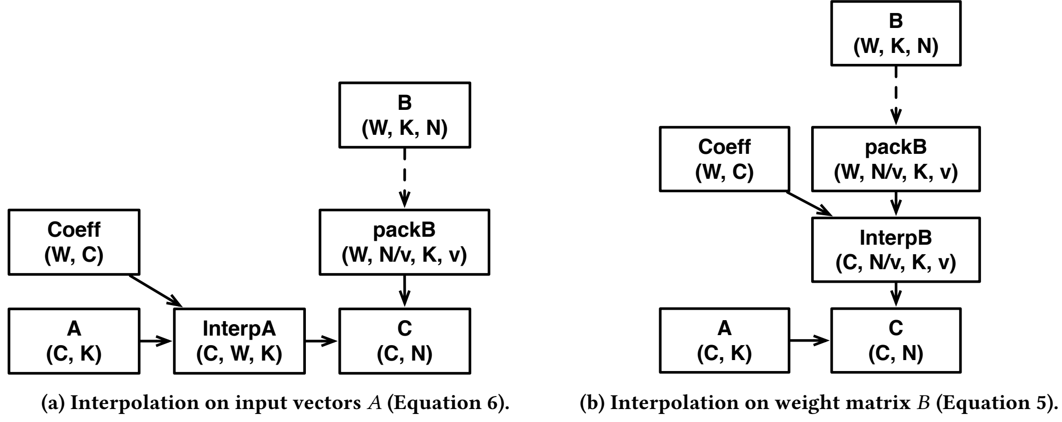
$$4 \cdot K \cdot N + C \cdot (K + N) \tag{8}$$

The reduction factor can be calculated with

$$\frac{(2C + 4) \cdot K \cdot N + C \cdot (K + N)}{N \cdot K \cdot 4 + C \cdot (K + N)} = \frac{1}{\frac{2}{C} + \frac{1}{2N} + \frac{1}{2K}} + 1$$

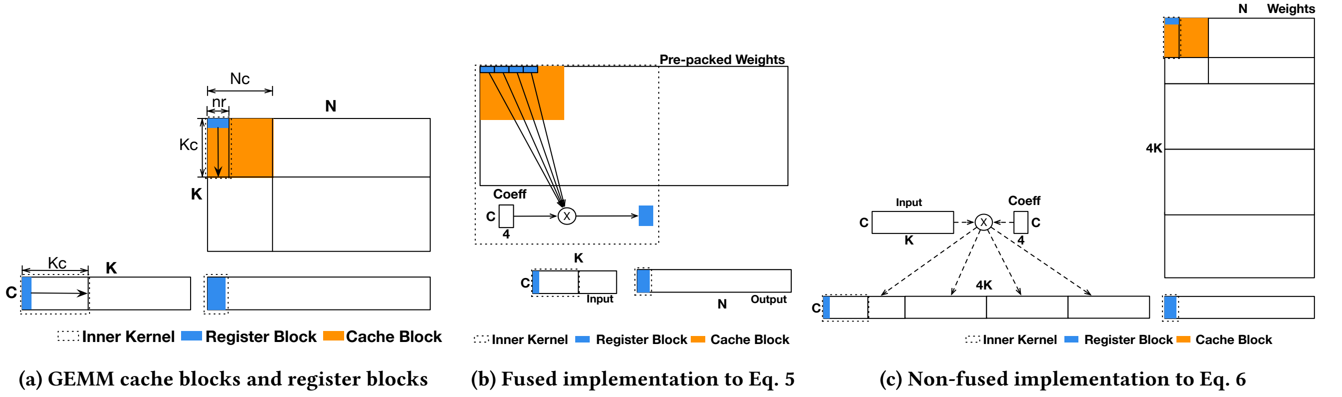
Our configurations generally set N and K from 256 to around 1500, C from 1 to 8. Hence $\frac{1}{2N}$ and $\frac{1}{2K}$ can be safely ignored. The above equations show that the fused approach can save memory footprint by a remarkable factor of **1.5x** ($C = 1$) to **5x** ($C = 8$). Therefore, we never adopt the non-fused version of Equation 5. The challenge to implement our fused algorithm is that there is no known manually-tuned optimized routines can be leveraged. The tensor program compilers, as will be demonstrated in later sections, are the utilities to build such high performance yet cross-platform solutions.

3.2.2 Interpolation on Input Vectors. Equation 6 has migrated the interpolation arithmetics onto input vectors. There are two major benefits: (1) the input vectors are much smaller than the weight matrices, which reduces computing complexity for interpolation



(a) Interpolation on input vectors A (Equation 6). (b) Interpolation on weight matrix B (Equation 5).

Figure 1: The compute graph in each layer. We label the tensor shapes and dependencies, where C denotes number of characters, W denotes number of weight configurations (always 4 for cubic algorithm) and dashed line indicates one-time compute at initialization.



(a) GEMM cache blocks and register blocks (b) Fused implementation to Eq. 5 (c) Non-fused implementation to Eq. 6

Figure 2: Comparison of memory access hierarchies for GEMM and our implementation to Equation 5 and Equation 6, respectively

and (2) the subsequent matrix multiplications can be efficiently implemented by invoking an optimized GEMM routine.

We analyze compute and memory access complexity similarly. For interpolation, the detailed compute steps can be expressed as:

$$\mathcal{W} \cdot \mathbf{X} = \begin{bmatrix} \mathcal{W}_{0,0}X_0 & \mathcal{W}_{0,1}X_0 & \mathcal{W}_{0,2}X_0 & \mathcal{W}_{0,3}X_0 \\ \mathcal{W}_{1,0}X_1 & \mathcal{W}_{1,1}X_1 & \mathcal{W}_{1,2}X_1 & \mathcal{W}_{1,3}X_1 \\ \cdots & \cdots & \cdots & \cdots \\ \mathcal{W}_{C,0}X_C & \mathcal{W}_{C,1}X_C & \mathcal{W}_{C,2}X_C & \mathcal{W}_{C,3}X_C \end{bmatrix}$$

where $\mathcal{W}_{i,j}$ is an element in matrix $\mathcal{W}$ and each X_i is a row in $\mathbf{X}$ which corresponds to the feature vector of a game character. The interpolation employs $4 \times C \times K$ FLOPs with memory footprint of at least $5 \times C \times K$ elements (read and write combined). The interpolation arithmetics actually expands the K dimension by a factor of 4. The matrix dimension is $C \times 4K$.

The subsequent matrix multiplication can invoke standard GEMM routines, with the computing complexity of $C \times N \times 4K$ and memory footprint of at least $C \times 4K + C \times N + 4K \times N$ elements.

Overall, the computing complexity for Equation 6 is

$$(\frac{4}{N} + 4) \cdot C \cdot K \cdot N \tag{9}$$

and memory access complexity is

$$9 \cdot C \cdot K + C \cdot N + 4 \cdot K \cdot N \tag{10}$$

Fusion is also feasible to reduce the memory footprint. The ideal case is that no extra memory access despite the GEMM. Therefore, the minimal memory foot print is

$$4 \cdot C \cdot K + C \cdot N + 4 \cdot K \cdot N \tag{11}$$

3.2.3 Complexity Comparison. We have analyzed the compute and memory access complexity for Equation 5 and 6. By comparing Equation 7 and Equation 9, it is clear that Equation 6 saves around 25% compute load than Equation 5. However, the memory access complexity has an inverse conclusion: Equation 5 is more memory efficient than Equation 6. We will see the differences later in performance evaluations.

4 TVM-ENABLED IMPLEMENTATIONS

In this section, we present details to implement Equation 5 and Equation 6 with TVM. We ignore the interpolated bias term $b \cdot \mathcal{W}$ as its implementation is trivial. We use *Tensor Expressions* (**TE**) to construct our compute graph and subsequently employ the *primitives* to guide the compiler to create an optimized schedule. We present our approach in three aspects: (1) tensor expression, (2) scheduler and optimizations and (3) code generation.

4.1 Tensor Expression and Compute Graph

We illustrate the compute graph of our deduced equations (Equation 5 and 6) in Figure 1. Tensor **A** holds the feature vectors of all characters and tensor **B** is the combined weight matrices. The weight matrix can be packed into more efficient memory layout **packB** because it is loaded only once at initialization and kept constant. The shape of **packB** in the figure is only an example as it changes according to the inner kernels. As for interpolation, our equation takes a $C \times 4$ parameter tensor **Coeff**, which corresponds to $\mathcal{W}$ in previous equations.

4.2 Hand-written Scheduler

Put aside the interpolation arithmetics, the PFNN compute pattern degrades to matrix multiplications (GEMM), which denotes that PFNN features *GEMM-alike* compute patterns. We employ insights from classic GEMM optimization works [5][7], which make great efforts to improve cache efficiency and arithmetic intensity. Our approach takes the identical strategies in terms of cache blocking, register blocking and loop reordering. The memory access hierarchies of GEMM and our approaches are illustrated in Figure 2. In Figure 2, it appears that the memory access patterns can be easily migrated from GEMM to our approaches without modifications. The the **InterpA** approach (Equation 6) can also employ the GEMM inner kernel by leaving the interpolation arithmetics as an external "add-on" procedure. As the input vectors are small in size, we expect the promotion in compute efficiency can compensate the interpolation overhead. The inner kernels for the **InterpB** approach (Equation 5) need to be rebuilt as the interpolation arithmetics are most efficient at the innermost loop. As interpolation is now refactored to only use multiplications and additions (refer to Equation 5 and Algorithm 2 for details), it is straightforward to implement with tensor program compilers that are dedicated to such dense arithmetics. To be more specific, we employ the TVM primitives *tile*, *reorder*, *compute_at*, *compute_inline* and *vectorize* to build the aforementioned schedulers, and subsequently generate final code library with the LLVM backend. Parameters including cache block size and intermediate loop orders are further adjusted with the help of AutoTVM[3]. However, most of our manually set parameters are very close to the automatically searched results, we will not further discuss the impact of auto tuning.

4.3 Automatically Created Scheduler

We leverage Ansor [22] to automatically generate the scheduler. Ansor accepts the compute graphs as illustrated in Figure 1, build and search for the best scheduler internally. Thus, the user needs no prior knowledge of the underlying hardware features and GEMM optimization techniques. An application developer could focus on complexity analysis as we have conduct in Section 3.2, which is sufficient to build efficient implementations.

The main concern of this end-to-end workflow is its capability to stably converge to a scheduler that is comparable to or outperform the manual approach. We will specially compare the performance of Ansor generated schedulers and hand-written schedulers in the evaluation section.

Table 1: Layer Configurations of our PFNN neural network

Layer	K (InputVecLen)	N (OutputVecLen)
1	912	256
2	256	256
3	256	1032

Table 2: Hardware single-core specifications of our test beds, including the Qualcomm 855 and 730G SoC and Intel Xeon Platinum 8255c. The compute capability is measured in multiplication arithmetic throughput, where the unit is billion floating-point operation per second (GFLOPS).

	Q855	Q730G	I8255c
Vector Instruction Sets			
Name	Neon	Neon	AVX512
Clock	2.8GHz	2.2GHz	2.5GHz
Vector Width	128bit	128bit	512bit
GFLOPS	22	18	77
Cache size			
L1d	64KB	64KB	64KB
L2	512KB	256KB	1024KB
L3	2MB	1MB	35MB
Tested Single-Thread Bandwidth (GB/s)			
L2	84	83	85
L3	43	40	29
DRAM	25	17	12

4.4 Embedded Inner Kernel

Our standard TVM optimization procedure invokes LLVM for final code generation. LLVM produces good results in most cases, while it is not designed to squeeze out the last ounce of performance due to code generation quality and unexpected corner cases.

To settle this problem, we employ a TVM mechanism named *Tensorize* in order to leverage customized and hand tuned high performance kernel functions. We apply embbed inner kernel only for **InterpA** (Equation 6). We employ libxsmm [7] as the inner kernel for x86 architectures with AVX2/AVX512 instruction set, and a homegrown inner kernel that takes identical approach for the Arm64 instruction set.

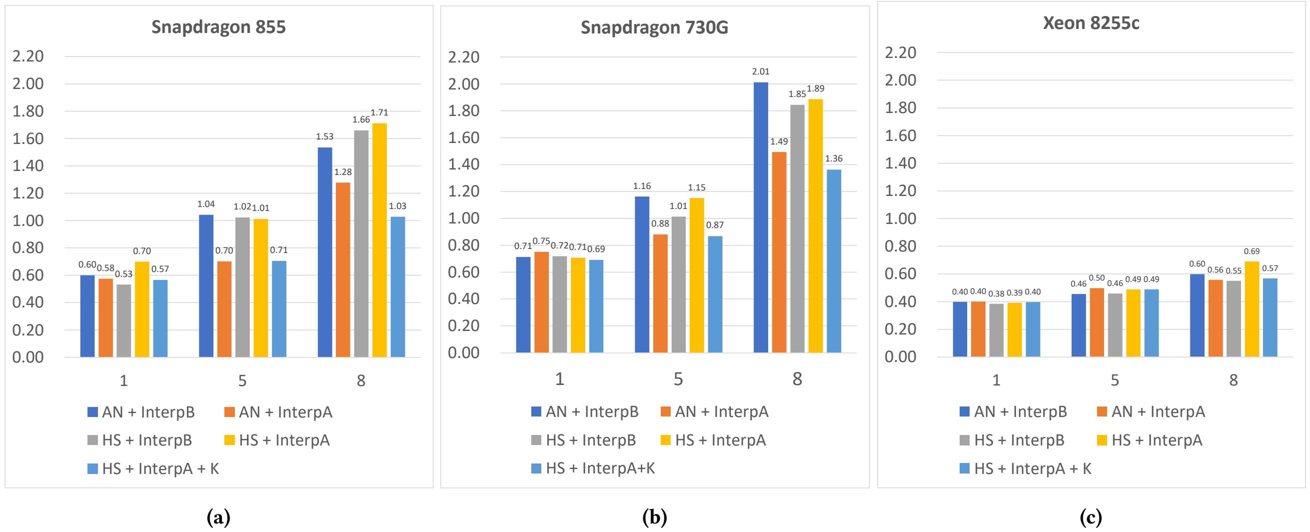


Figure 3: The inference latency (in millisecond) of our TVM-enabled approaches, lower is better. InterpB corresponds to Equation 5 and InterpA corresponds to Equation 6. AN denotes Ansor. HS denotes hand-written scheduler. K denotes embedded inner kernel. All tests are conducted with a single core.

5 EVALUATION

5.1 Neural Network Configuration

Our PFNN configuration has three layers with 256 hidden neurons in the middle layer. The input and output dimensions of each layer is listed in Table 1. The tested character number C ranges from 1 to 8. Without loss of generality, we only compare $C = 1, 5, 8$ cases when automatic search and generation is involved, in order to reduce search time.

5.2 Test Bed Setup

We create clean test environments on different hardware platforms, including servers and mobile phones. Although our work was finally embedded into real games, the inference latency fluctuates in a wide range due to various game scenario and hardware resources. Therefore the time recorded in real production environment has minor reference value.

We compare performance on an Android mobile phone and an Intel server, where all features of TVM and Ansor can be easily enabled and therefore possible to reproduce. The hardware specifications are listed in Table 2.

5.3 End-to-End Inference Latency

We evaluated the end-to-end neural network inference latency in our test beds. Baseline is implemented with Eigen. We configure Ansor to search 2000 steps for all cases.

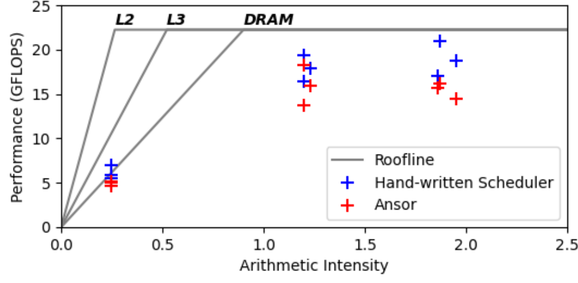
The performance of TVM-enabled approaches are compared and illustrated in Figure 3. We should note that on the server platform, all methods performed similarly. On mobile platforms, all methods achieved close performance for the $C = 1$ case. We observed distinct performance for the $C = 5, 8$ cases. The InterpA equation with hand-written scheduler and embedded inner kernel (**HS+InterpA+K** in the charts) achieved the best performance. The performance of Ansor, however, are quite comparable to our best hand-written scheduler. The hand-written scheduler outperformed Ansor by a factor of 24.3% and 9.56% with 8 characters on the 855 and 730G processors, respectively. By comparing **HS+InterpA** and **HS+InterpA+K**, we conclude that the embedded inner kernel contributes the most significant performance gain.

Table 3: Inference latency (millisecond) of Eigen implementations.

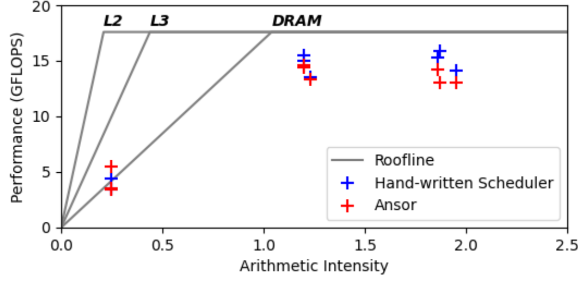
	InterpB (Naive)			InterpA		
	C=1	C=5	C=8	C=1	C=5	C=8
Q855	2.03	5.25	7.76	1.83	2.77	2.63
Q730G	2.63	8.68	12.67	2.38	3.36	3.53
I8255c	0.64	3.20	4.99	0.89	2.03	2.18

Table 3 lists the inference latency of Eigen baseline implementations. Our best TVM-enabled approach outperforms the Eigen baseline by average factors of **7.08x** on Intel 8255c, **6.16x** on Qualcomm 855 and **7.07x** on Qualcomm 730G, respectively. For the Eigen implementations with 8 concurrent game characters, the algebraic optimization alone achieved an average acceleration ratio of **2.94x** on the three platforms.

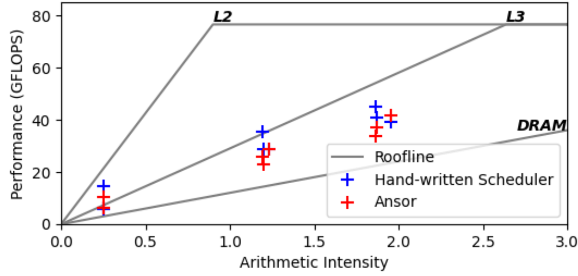
Another interesting result is that for the $C = 8$ case, the Ansor implemented InterpA outperforms InterpB by a factor of **20%** and **34%** on 855 and 730G, respectively. As we have already claimed in Section 3.2, InterpA is theoretically 25% faster than InterpB in terms of computing complexity. The results indicate that Ansor is capable to stably create high quality scheduler for both interpolation equations.



(a) Qualcomm Snapdragon 855



(b) Qualcomm Snapdragon 730G



(c) Intel Xeon Platinum 8255c

Figure 4: Layerwise performance comparison of hand-written scheduler with inner kernel (HS+InterpA+K) and Ansor (AN+InterpA) against the hardware roofline model on our test environments.

5.4 Roofline Analysis

We employ hardware roofline model to evaluate the quality of our implementations. The roofline model calculates the maximum compute performance with respect to **arithmetic intensity**, which is defined as the fraction of computing complexity versus memory access complexity. The roofline calculation can be expressed with:

$$\mathcal{R} = min(\mathbf{B} * \mathcal{A}, \mathbf{P})$$

where $\mathcal{R}$ denotes the roofline performance on y-axis, $\mathcal{A}$ denotes arithmetic intensity on x-axis, **P** denotes peak compute performance and **B** denotes memory bandwidth. In this work, the arithmetic intensity of our approaches has already been analyzed in Section 3.2. We summarize as follows:

- For Equation 5 (the **InterpB** manner), we calculate with the fully fused equations.

$$\mathcal{A} = \frac{5 \cdot C \cdot K \cdot N}{4 \cdot K \cdot N + C \cdot (K + N)}$$

- For Equation 6 (the **InterpA** manner), we calculate with the non-fused equations in order to employ GEMM kernels.

$$\mathcal{A} = \frac{(\frac{4}{N} + 4) \cdot C \cdot K \cdot N}{9 \cdot C \cdot K + C \cdot N + 4 \cdot K \cdot N}$$

In calculation of the roofline model, we found that the arithmetic intensity $\mathcal{A}$ is mostly determined by the character number C as N and K are at the same order of magnitude. Therefore the points in the plots will form three clusters on the X-axis, corresponding to $C = 1, 5, 8$ cases respectively. We evaluate the layer-wise performance by repeatedly running each layer for 1000 rounds. We illustrate the results as roofline charts in Figure 4, where we only test the top performers in latency test (see Figure 3), namely **AN+InterpA** and **HS+InterpA+K**.

Figure 4a and 4b reveal that the $C = 1$ case is bound by L3 bandwidth while others are bound by FMA compute capability. For the compute bound cases, our methods utilize 85% − 95% of the devices' compute capability. Ansor also exhibits specially good performance considering the fact that it is end-to-end automated. The performance of Ansor only lags slightly behind our carefully optimized hand-written schedulers. The problem is that it needs long search time (1 to 2 hours) on every device. We would expect the following work of Ansor could shorten the scheduler search time.

Figure 4c shows that the PFNN algorithm is inherently bound by memory bandwidth in our cases. This chart well explains why the inference latency in Figure 3c are neck-to-neck among different methods. Another observation is that the performance figures are very close to the L2/L3 roofline bound, which indicates the memory access patterns in our methods are efficient.

6 CONCLUSION

We have presented an efficient computing method for the phase-functioned neural networks to deploy the algorithms in mobile games. Algebraic optimization techniques on the algorithm level and TVM compiler-based optimization techniques on the implementation level are adopted in conjunction to deliver portable high performance across mobile and server CPUs. We simplified and transformed the original PFNN equations for the algebraic optimizations to enable batched inference computation for multiple concurrent game characters. We conducted meticulous compute and memory access complexity analysis to guide and explain the design of our high-performance solutions. We manually created schedulers with human knowledge and insights for the compiler-based optimizations and built end-to-end automated solutions with Ansor.

Performance evaluation reveals that our TVM-enabled approaches outperform a baseline Eigen implementation by factors from **6.16x** to **7.08x** on multiple platforms. We also modeled the

roofline of respective test platforms to demonstrate that our methods achieved performance that is very close to hardware peak on all tested hardware platforms. Finally, the network's inference latency is below **2ms** for eight concurrent game characters on a single mobile CPU core. Such efficiency is essential to enable real-time PFNN character controllers in mobile games, where hardware compute resources are stressed out.

ACKNOWLEDGMENTS

This work was partly supported by the National Key Research and Development Program of China Grant No. 2021YFF1200100 and 2021YFF1200104, Strategic Priority CAS Project XDB38050100, the Key Research and Development Project of Guangdong Province under grant No. 2021B0101310002, the National Science Foundation of China under grant No. U1813203, the Shenzhen Basic Research Fund under grant No. KQTD20200820113106007, the General Program of Key Technology from Shenzhen under grant No. JSGG20190220164202211 and JSGG20191129110221063. We also want to thank the editors and reviewers for their professional comments which have greatly improved this manuscript.

REFERENCES

[1] Riyadh Baghdadi, Abdelkader Nadir Debbagh, Kamel Abdous, Fatima Zohra Benhamida, Alex Renda, Jonathan Elliott Frankle, Michael Carbin, and Saman Amarasinghe. 2020. TIRAMISU: A polyhedral compiler for dense and sparse deep learning. *arXiv preprint arXiv:2005.04091* (2020).

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[3] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166* (2018).

[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[5] Kazushige Goto and Robert A van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)* 34, 3 (2008), 1–25.

[6] Gaël Guennebaud, Benoit Jacob, et al. 2010. Eigen. *URl: http://eigen. tuxfamily. org* 3 (2010).

[7] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 981–991.

[8] Daniel Holden, Taku Komura, and Jun Saito. 2017. Phase-functioned neural networks for character control. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1–13.

[9] Ping Kuang, Dingli Luo, Haoshuang Wang, and Lixue Zhang. 2019. An improved calculation system for phase-functioned neural network and implementation in unreal engine. *Cluster Computing* 22, 6 (2019), 15505–15516.

[10] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[11] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–13.

[12] Yongchao Liu, Yue Jin, Yong Chen, Teng Teng, Hang Ou, Rui Zhao, and Yao Zhang. 2020. Woodpecker-DL: Accelerating Deep Neural Networks via Hardware-Aware Multifaceted Optimizations. *arXiv preprint arXiv:2008.04567* (2020).

[13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[14] Sebastian Starke, He Zhang, Taku Komura, and Jun Saito. 2019. Neural state machine for character-scene interactions. *ACM Trans. Graph.* 38, 6 (2019), 209–1.

[15] Sebastian Starke, Yiwei Zhao, Taku Komura, and Kazi Zaman. 2020. Local motion phases for learning multi-contact character movements. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 54–1.

[16] Sebastian Starke, Yiwei Zhao, Fabio Zinno, and Taku Komura. 2021. Neural animation layering for synthesizing martial arts movements. *ACM Transactions on Graphics (TOG)* 40, 4 (2021), 1–16.

[17] Field G Van Zee and Robert A Van De Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS)* 41, 3 (2015), 1–33.

[18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[19] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

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

[21] He Zhang, Sebastian Starke, Taku Komura, and Jun Saito. 2018. Mode-adaptive neural networks for quadruped motion control. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–11.

[22] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 863–879.