swATOP: Automatically Optimizing Deep Learning Operators on SW26010 Many-Core Processor

Wei Gao*
gaow17@mails.tsinghua.edu.cn
Tsinghua University

Jiarui Fang*
fjr14@mails.tsinghua.edu.cn
Tsinghua University

Wenlai Zhao
zhaowenlai@tsinghua.edu.cn
Tsinghua University

Jinzhe Yang
jinzheyang@gmail.com
Imperial College London

Long Wang
Wanglong12@baidu.com
System Department of Baidu

Lin Gan
lingan@tsinghua.edu.cn
Tsinghua University
National Supercomputing Center in Wuxi

Haohuan Fu
haohuan@tsinghua.edu.cn
Tsinghua University
National Supercomputing Center in Wuxi

Guangwen Yang†
ygw@tsinghua.edu.cn
Tsinghua University
National Supercomputing Center in Wuxi

ABSTRACT

Achieving an optimized mapping of Deep Learning (DL) operators to new hardware architectures is the key to building a scalable DL system. However, handcrafted optimization involves huge engineering efforts, due to the variety of DL operator implementations and complex programming skills. Targeting the innovative many-core processor SW26010 adopted by the 3rd fastest supercomputer Sunway TaihuLight, an end-to-end automated framework called swATOP is presented as a more practical solution for DL operator optimization. Arithmetic intensive DL operators are expressed into an auto-tuning-friendly form, which is based on tensorized primitives. By describing the algorithm of a DL operator using our domain specific language (DSL), swATOP is able to derive and produce an optimal implementation by separating hardware-dependent optimization and hardware-agnostic optimization. Hardware-dependent optimization is encapsulated in a set of tensorized primitives with sufficient utilization of the underlying hardware features. The hardware-agnostic optimization contains a scheduler, an intermediate representation (IR) optimizer, an autotuner, and a code generator. These modules cooperate to perform an automatic design space exploration, to apply a set of programming techniques, to discover a near-optimal solution, and to generate the executable code. Our experiments show that swATOP is able to bring significant performance improvement on DL operators in over 88% of cases, compared with the best-handcrafted optimization. Compared to a black-box autotuner, the tuning and code generation time can be reduced to minutes from days using swATOP.

KEYWORDS

Autotuning, Deep Learning Operators, SW26010

ACM Reference Format:
Wei Gao, Jiarui Fang, Wenlai Zhao, Jinzhe Yang, Long Wang, Lin Gan, Haohuan Fu, and Guangwen Yang. 2019. swATOP: Automatically Optimizing Deep Learning Operators on SW26010 Many-Core Processor. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337883

1 INTRODUCTION

Deep Learning (DL) has achieved superior performance far beyond traditional machine learning methods in several fields, e.g., image recognition, natural language processing, speech recognition and etc. The traditional general-purpose processors represented by the CPU have been unable to meet the growing computation demand of DL applications. In order to efficiently train, and deploy models in a massively parallel and heterogeneous computing environment, scalable DL systems must be built on top of many-core architectures in emerging cutting-edge supercomputers. As a result, optimization of arithmetic intensive DL operators, such as convolution, matrix multiplication, according to underlying hardware intrinsics, becomes the important foundation of DL system building.

Adapting DL operators to a new architecture demands substantial human efforts to rethink, redesign, and re-implement existing algorithms. This paper takes the SW26010, an innovative heterogeneous many-core processor integrating 260 computing cores, making Sunway TaihuLight the third supercomputer in the world, as a target to demonstrate how we can reduce human efforts. There have been a set of manual efforts to support deep learning applications on the Sunway TaihuLight system, including a linear algebra

*Both authors contributed equally to this research.
†Corresponding Author: ygw@tsinghua.edu.cn.

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337883

library called xMath [9], a DL operator library called swDNN [7] and a parallel framework called swCaffe [12]. However, there are still a number of obstacles that prevent these software from better applicability and optimal performance.

First, complex architectural intrinsics make it difficult for developers to map DL operators on new hardware for better performance and new functional modules. In term of the SW26010 (Sec. 2), it provides software controlled cache scheme, register-level data sharing between cores and memory access in a coarse-grained manner. These unique hardware features make some design techniques on the GPU and CPU invalid in our case.

Second, it is a huge engineering burden to develop a DL operator library that brings satisfactory performance to all possible parameter configurations. The search space of possible parameter configurations resulting from multidimensional tensors is colossal, while the optimal implementation of variant parameter configurations should be treated individually. As a result, it is extremely difficult to build a few of generic implementations manually to efficiently support all DL operator variants.

Third, compiler tools of new hardware are often not able to meet the demanding requirements of DL applications, and mature compiler tool chains (e.g. LLVM [10]) usually have not supported the new hardware and new features. As for SW26010, even programming with low-level intrinsics is not enough to lead compilers to produce an optimal assembly code. According to [7] [9], annoying assembly level code tuning is sometimes necessary.

A framework that can perform automatic design space exploration and discover near-optimal solutions is therefore highly desirable. Existing frameworks, such as Halide [16], TVM [5], Tensor Comprehensions [20], are able to automatically optimize DL operators. However, they can not take full advantages of many important architecture-specific features (for SW26010, e.g., pipeline, register data communication, DMA engine), resulting in non-trivial performance loss. For example, recent work swTVM [14] extends the TVM to generate code for SW26010. However, due to lack of support for register communication and pipeline, code generated by swTVM performs much slower than existing manual version. Besides, they heavily depend on low-level code optimization and generation tools such as LLVM, Polyhedral tools [1, 21], which is not supported by new hardware like SW26010. Third, their tuning methods emphasize too much on versatility, and ignore prior knowledge of specific hardware which is helpful to prune the search space.

In this paper, we suggest a more practical way to achieve optimal DL operator performance through an abstracted description of the algorithm and an automated tuning framework. Decomposed into a sequence of tensorized primitives , arithmetic intensive DL operators can be expressed in an auto-tuning-friendly form. We provide an automated framework called swATOP to find the best organization scheme of these tensorized building-blocks and generate near-optimal code. The major philosophy behind swATOP is to separate hardware-agnostic optimizations from hardware-dependent optimizations.

Our swATOP framework is built on a set of tensorized computing and memory access primitives, which can take full advantage of the hardware features of SW26010. The swATOP framework consists of a scheduler, an intermediate representation (IR) optimizer, an autotuner, and a code generator. These modules cooperate to perform automatic schedule space exploration, to apply a set of programming techniques for tensorized primitives, to discover a near-optimal solution, and to generate the executable code. Our framework can handle general performance issues of tensorizing DL operators, such as techniques to process the boundary where tensorized primitives cannot be used, and methods to hide memory access latency. For the SW26010 processor, a performance-model-based static autotuning method is adopted to quickly identify a near-optimal solution. swATOP can be used as a offline compiler by pre-generating near-optimal executable code, or be integrated into other frameworks to provide online autotuning.

The paper makes the following contributions.

- We propose an auto-tuning-friendly abstraction separating hardware-agnostic optimizations from hardware-dependent optimizations to combine autotuning techniques and architecture-specific tensorized primitives.
- We demonstrate that a performance-model-based autotuner based on prior knowledge of the hardware is a practical solution for latency-oriented many-core architectures. From days to minutes, the autotuner of swATOP is able to reduce over two orders of magnitude of tuning time cost compared to a black-box autotuner. Even in the worst case, it only brings **less than 8%** performance loss.
- We propose and implement an automated framework to optimize arithmetic intensive DL operators on SW26010, which is able to replace the time-consuming manual optimization. The code generated by swATOP outperforms the best manual optimization implementation in over **88%** cases.

2 SW26010 MANY-CORE PROCESSOR

The SW26010 many-core processor (Fig. 1) consists of four core groups (CGs) and provides a peek performance of 3.06TFlops with 136 GB/s hardware memory bandwidth. Each CG includes one management processing element (MPE), one computing processing element (CPE) cluster with 8×8 CPEs, and one memory controller (MC). These four CGs are connected via a network-on-chip (NoC). The processor connects to outside devices through a system interface (SI).

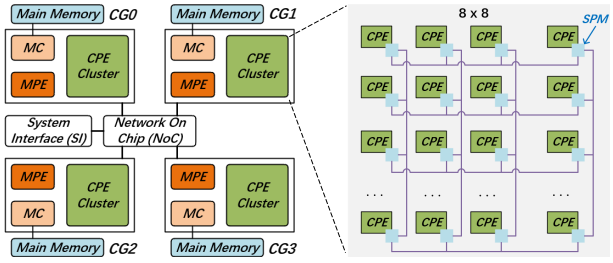

Figure 1: Overview of SW26010.

The MPE is a complete 64-bit reduced instruction set computer (RISC) core, which is an ideal core for handling management and communication functions. The CPE is designed to provide maximum aggregated computing power while minimizing the complexity of the micro-architecture. Each CPE has 16KB L1 instruction cache and a 64KB user-controlled scratch pad memory (SPM). The

CPE cluster is organized as an 8×8 mesh with a mesh network that provides low-latency register data communication among the 8×8 CPEs. Both the MPE and CPE support 256-bit vector instructions.

Compared with other multi-core or many-core processors, the CPE cluster of SW26010 design demonstrates a number of different features: (i) The SPM is usually configured as a user-controlled fast 'cache', and programmers have to explicitly move data onto or out of the SPM. A CPE provides two kinds of memory access, i.e., global load/store (GL/GS) and direct memory access (DMA) engine, to transfer data between main memory and SPM. Stream Triad Test in [24] shows that the bandwidth of GL/GS and DMA are 1.48GB/s and 22.6GB/s, respectively. As a result, exploring utilization of DMA is important in optimization. (ii) The CPE cluster offers fast register data communication among the 8×8 CPEs. Benchmarking in [24] shows that the integrated register communication bandwidth per CPE cluster is 647.25GB/s. Register communication provides an important data sharing capability at the CPE level. (iii) Each CPE includes two pipelines (P0, and P1) for the instruction decoding, issuing, and execution. P0 is for floating-point operations, and both floating-point and fixed-point vector operations. P1 is for memory-related operations. Both P0 and P1 support integer scalar operations. Therefore, identifying the right form of instruction-level parallelism, i.e., pipeline, can potentially resolve the dependence in instruction sequences, and further improve the computation throughput.

Makeing full use of above architecture-specific intrinsics including register communication, pipelines, SPM, DMA engine, and vectorization, is critical in automatical optimization of DL operators on SW26010.

3 AUTO-TUNING-FRIENDLY DL OPERATORS

Arithmetic intensive DL operators, such as multi-channel convolution, matrix multiplication, consume majority of total time in DL networks, for example, more than 85% in Convolutional Neural Network (CNN) [13]. Consequently, the key point in optimizing DL operators is accelerating these compute-intensive arithmetic operators. swATOP focus on these pivotal DL operators. The major computation part can be expressed in the form of nested for-loops with multiply-and-accumulate (MAC) operations. As shown in Alg. 1, a naive implementation of a direct convolution is a 7-level nested loops of a single MAC statement. A design space is derived by performing equivalent transformations on the nested loops. However, using MAC as the basic building-blocks will lead to a huge searching space for identifying the best candidate.

By transforming lower-order data to higher-order data, a unit of computation can be replaced with tensorized intrinsics, making it easy to leverage handcrafted micro-kernels. Therefore, we propose to use tensorized micro-kernels as the basic building-blocks for DL operators, and call such a process as *tensorization*.

Tensorization is able to decouple the part of scheduling the loops of the kernel, and part of implementing the kernel. The former is often hardware-agnostic and suitable for autotuning, while the latter is generally hardware-dependent and manually designed to leverage all architecture-specific features. We can then apply different strategies to the two optimization problems accordingly. In addition, since the dimension parameters of the tensors are discrete and limited, tensorization is able to facilitate a precise performance estimation. In contrast, traditional optimization methods are usually tightly coupled to the underlying hardware, making it difficult to drive an autotuning process.

Matrix multiplications are naturally suitable to be tensorized into GEMM micro-kernels in the form of three nested loops. For multi-channel convolution, as shown in Fig. 2, we propose three tensorized designs. The explicit-GEMM-based convolution (Fig. 2, left) is a common approach [4] to implement convolutional layers, which first expands the image into a column matrix (also known as *im2col* process), and performs a matrix-multiplication operation on the column matrix and the filter matrix.

The winograd method (Fig. 2, middle) computes minimal complexity convolution over small tiles using Winograd's minimal filtering algorithms [11], and is fast with small filter kernels (3x3, 4x4). The Winograd algorithm works on small tiles of the input image. The input tile and filter are transformed, the outputs of the transform are multiplied together in an element-wise fashion, and the result is transformed back to obtain the outputs of the convolution. In our design, every single element-wise multiplication can be packaged as a matrix multiplication. The transformations between the tensors and the matrices are done quickly on CPE clusters, so that most of the computation happens in matrix multiplications. Winograd method has to conduct a batch of GEMM operations, i.e., 16 multiplication for 3x3 kernels.

The implicit-GEMM-based convolution (Fig. 2, right) [6, 7] is based on direct convolution. Different from the explicit-GEMM-based method, as shown in Alg. 2, a tensorized implicit-GEMM-based convolution can be derived by reordering and replacing some innermost loops with GEMM micro-kernels.

Algorithm 1 A MAC-based Convolution Implementation

Input: Input, Filter
Output: Output

```
1: for cB in range(0, B) do
2:     for cR_o in range(0, R_o) do
3:         for cC_o in range(0, C_o) do
4:             for cK_r in range(0, K_r) do
5:                 for cK_c in range(0, K_c) do
6:                     for cN_o in range(0, N_o) do
7:                         for cN_i in range(0, N_i) do
8:                             output_{cB,cN_o,cR_o,cC_o} =
   in_{cB,cN_i,cR_o+cK_r,cC_o+cK_c} * weight_{cN_o,cN_i,cK_r,cK_c}
```

4 METHODOLOGY

Based on the tensorized description of DL operators, the overview of swATOP is shown in Fig. 3. Users describe the computation of DL operators and schedule space using tensorized-primitive based DSL. Scheduler takes the DSL as input, traverses the defined schedule space, and generates all candidate implementations represented in IR structure (Sec 4.4). IR optimizer performs multiple customized IR optimizations. Then autotuner leverages a performance-based cost model to predict and pick best (or top k) implementations, and code generator generates efficient machine code for SW26010. In the rest of this section, we introduce these components in details.

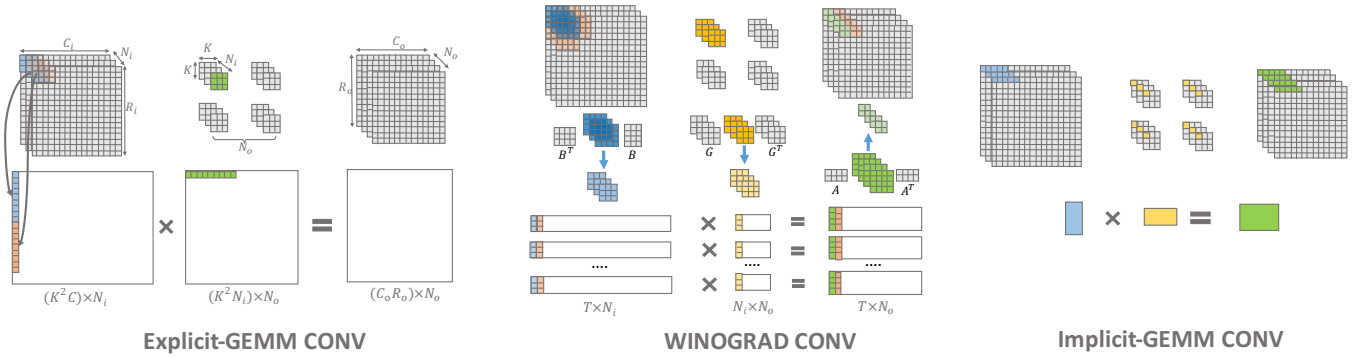

Figure 2: Three methods to decompose convolution into tensorized GEMM primitives.

Algorithm 2 A Tensorized Convolution Implementation

```
for cR_o = range(0, R_o) do
  for cC_o = range(0, C_o) do
    for cK_r = range(0, K_r) do
      for cKc = range(0, K_c) do
        cR_i = cR_o + cK_r
        cC_i = cC_o + cK_c
        DMA get D_i ← N_i × B channels of in_(cC_i, cR_i)
        DMA get W ← N_i × N_o channels of weight_(cK_c, cK_r)
        GEMM Primitives: D_o += W × D_i
    DMA put D_o → N_o × B channels of out_(cC_o, cR_o)
```

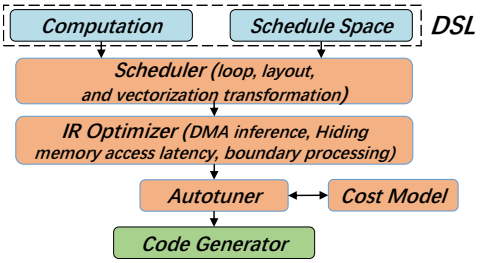

Figure 3: Overview of swATOP. swATOP takes DSL for DL operators as input and generates high performance code (C code) for SW26010.

4.1 Tensorized Primitives

The micro-kernels can be implemented as tensorized primitives to facilitate the autotuning process in swATOP. Our primitives mainly include a set of operations to do calculation in SPM and transfer data between memory and SPM using DMA engine. We implement a set of GEMM primitives in assembly language to do $C+ = A \times B$, where A, B and C reside in the SPM space. Our design leverages architecture-related features including **register communication**, the inherent parallelism between two **instruction pipelines**, and **vectorization**. Optimization details can be found in Appendix Sec. 9. The interface of our GEMM primitive is illustrated as follows, which is similar to the CBLAS interface. The difference is that we have added a parameter that indicates the vectorization dimension.

```
spm_gemm(int M, int N, int K, float ALPHA, float* A, int LDA, float* B, int LDB, float BETA, float* C, int LDC, swVecDim vd)
```

As shown in Fig. 12, each CPE needs to access a non-overlapping 2D block, so that the memory access pattern is strided. As a result, the DMA operation interface provides both continuous and strided memory access mode indicated by parameter `strideSize`. Since DMA works in asynchronous way, primitives to launch DMA engine and primitives to wait for finish need to be used in pairs.

```
swDMA(float* src, float* dst, size_t count, size_t blockSize, size_t strideSize, swMemcpyDirection dir, swReplyWord* replyword)
swDMAWait(swReplyWord* replyword, int replyTimes)
```

4.2 DSL

Using tensorized primitives as building-blocks, we introduce a language (Fig 4, left) implemented in embedding C++ to depict the computation of DL operator and corresponding schedule space. An initial tensorized implementation that only describe the computation is called a *schedule seed*. In DSL, the schedule seed is expressed using variables, tensors and computations. The implementation, which contains schedule information (e.g. loop, layout, vectorization schedule) and is logically equivalent to the schedule seed, is called as a *schedule strategy*. All valid schedule strategies constitute the *schedule space*. Our DSL provides many data structures and interfaces to depict the schedule seed and schedule space.

4.3 Scheduler

The scheduler is designed to discover all tensorized implementation candidates defined by DSL schedule space. Through the combination of three transformation methods, Loop Transformation, Layout Transformation, and Vectorization Transformation, we can build a schedule space with a large number of schedule strategies.

4.3.1 Loop Transformation. The loop transformation is able to extend the schedule space by combining a set of classic concepts, like loop splitting, loop reordering, and loop fusion. Split refers to splitting a single loop into two loops. The dimension on which the loop iterates can be split by a factor, creating two new dimensions: an outer dimension, over the old range divided by the factor, and an inner dimension, which iterates within the factor. Reorder refers to

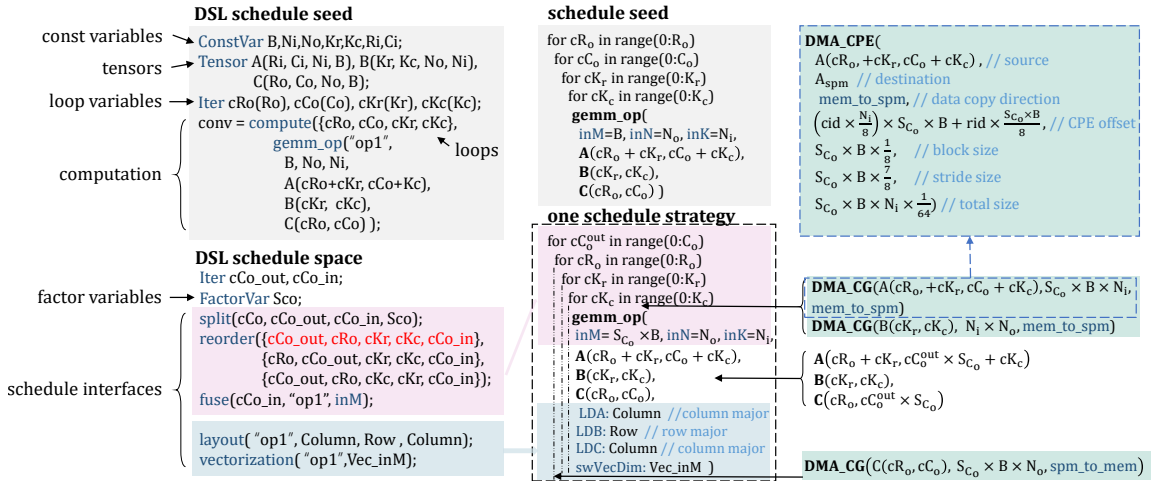

Figure 4: The left part is an example of DSL of implicit-GEMM-based convolution. DSL schedule seed describes the computation, while DSL schedule space defines the schedule strategies. `FactorVar` defines a factor variable used in `split` function. swATOP will automatically traverse all valid candidates of the factor. Since there are extremely numerous permutations of a set, `reorder` requires explicit candidates. The middle-top is the IR seed lowered from DSL schedule seed, and the middle-bottom is the lowered IR of one schedule strategy of schedule space. The right part is an example of DMA inference(Section 4.5.1). `rid, cid` is the row and column id of CPE in CG.

reordering the execution order of loops. Loop fusion merges multiple loops into a single one, which can be considered as a reverse operation of Split. Our loop fusion also can enlarge a specific dimension of GEMM primitives by merging loops into GEMM primitives. If n independent matrix multiplications share the same input, then they can be combined into one larger matrix multiplication with an output n times larger.

4.3.2 Layout Transformation. Even if the loop structure is determined, there can be a variety of implementations using different data layouts. Based on the loop transformation, there is room to expand the schedule space by using layout transformation. The data layout will affect the performance in two aspects. On one hand, it affects the DMA memory access mode by changing parameters such as contiguous memory block size, stride access length, and so on. On the other hand, the layout of data inside SPM affects the usage of GEMM primitive, such as parameter of leading dimension and transposition information.

In order to achieve high DMA efficiency and meet the rules of GEMM primitive usage, the layout transformation should follow some rules. The dimensions corresponding to outer loops should be placed as the high dimensions of data layout. The dimensions used as parameters of GEMM primitives should be placed as the leading dimensions.

4.3.3 Vectorization Transformation. The scheduling space can also be further extended by different vectorization transformation. As described in the Sec. 4.1, there are two ways to vectorize your calculation even if you use the GEMM primitives for the same input and output data. That is, either the M loop or the N loop in the three loops in order of (M, N, K) can be vectorized. Vectorization can bring higher computing performance, but at the same time introduce some restrictions on the length of loop. Sometimes we have to consider the specific circumstances of the layout and choose a feasible vector scheme.

4.4 Intermediate Representation

In order to support schedule transformation, following optimization and code generation, we design an intermediate representation (IR). The middle part of Fig. 4 shows a logical representation of IR. The IR is an abstract syntax tree (AST) composed of a serial of statement nodes, such as `for`,`if-then-else`, `DMA`, `gemm_op` and so on. Each statement node consists of some attributes. For example, `for` node contains attributes such as `iter`, `min`, `max`, and `stride`. IR is designed to support flexible mutations. Schedule strategies can be achieved by transforming the structure of IR and modifying attributes of the nodes. Optimizations (Sec 4.5) are achieved by mutating the IR structure.

Scheduler lowers each schedule strategy in schedule space into an IR structure by applying transformations to seed IR lowered from schedule seed (Fig 4, middle). These output IRs are used in following optimization (Sec 4.5), performance predictions (Sec 4.6), and code generation (Sec 4.7).

4.5 IR optimizer

IR optimizer is designed to optimize the output IRs of scheduler by mutating the IR structures. In this section, we highlight three optimization techniques in use: DMA inference, hiding memory access latency, boundary processing.

4.5.1 DMA inference. Users do not need to explicitly provide DMA information in the DSL. swATOP can automatically infer and inject required DMA nodes with attributes into IR. The attributes that the DMA node contains consist of source and destination memory addresses, direction of data copy, offset to the main memory address,

total size of the transferred data, block size of consecutive data, and stride size of two contiguous blocks. As stated in Sec. 4.1, the data required by GEMM primitives needs to be distributed across SPMs of all CPEs in a CG. The offset attribute needs to be carefully calculated to mark the address offset for each CPE.

We infer the DMA nodes for CPEs by first generating the DMA for whole CG (Fig 4, right-bottom):

`DMA_CG(addr, totalsize, direction)`

`addr`, `totalsize`, `direction` denotes the base address (to be read or written) on main memory, the total data size and the data copy direction, respectively. Then, DMA node for CPEs can be easily derived (Fig 4, right-top), i.e.,

`DMA_CPE(source, destination, direction, offset, block, stride, size)`

`size` can be calculated by `totalsize/64`, and a buffer on SPM, `buf_spm`, is implicitly allocated to cache the transferred data. If the `direction` is from main memory to SPM, the `source` is set to `addr`, otherwise, `buf_spm`. In a CG, CPE is identified by a pair of row and column id – `(rid, cid)`. `offset, block, stride` depends on the `(rid, cid)` and layout information in IR. For example, assuming that the input "`A`" is a two-dimensional, column-major (`i.e., LDA=Column`) matrix `A(M, N)`. The matrix is divided into 8x8 grids, and each CPE read the `(rid, cid)` tile. In details, `block, stride` is set to `M`$\times\frac{1}{8}$, `M`$\times\frac{7}{8}$, respectively. `offset` is set to `(cid`$\times\frac{N}{8}$`)×M + rid`$\times\frac{M}{8}$. To reduce redundant data copy, DMA nodes are injected into the IR as far as possible from `gemm_op`.

4.5.2 Hiding Memory Access Latency. Hiding memory access latency is a common technique that improves execution performance and resource utilization. swATOP automatically hides DMA latency by software prefetching (double buffer) by allocating two identical SPM buffers, one is used for computation and the other one is used for data fetching. The key point of auto-prefetching is how to automatically infer the memory address used by computation of the next loop iteration. Auto-prefetching of swATOP is readily applicable to loop nests in which the data access (e.g., memory address of DMA nodes) is function of the enclosing loop variables and parameters. In general, the input and output of arithmetic intensive DL operators are hypercubes, resulting in constant parameters of data access funcion. Consequently, data access can be considered as a function that maps value of enclosing loop variables onto the accessed memory address, i.e., $\Phi(\vec{I}) = addr$, where $\vec{I} = (iter_1, iter_2, ..., iter_k)$ is the vector of enclosing loop variables, and *addr* is the address of accessed data. As a result, inferring the next memory address is equivalent to inferring the value of $\vec{I}$ in next iteration. For the data access of each DMA node in IR, swATOP extracts $\vec{I}, \Phi$, and generates a nested `if-then-else` structure to infer the value of $\vec{I}$ in next iteration. Then swATOP injects the generated structure and a initializing DMA node into the IR to implement auto-prefetching.

4.5.3 Boundary Processing. Boundary issue occurs when the length of the loop cannot be divided by the split factor, and the boundary data cannot be processed using the original tensorized primitive. The boundary issue makes it more difficult to manually write efficient and correct programs, due to complex code, redundant computations and additional storage space.

swATOP employs two strategies to apply optimization for boundary processing. First, if data of boundaries can be processed with tensorized primitives of different parameters, swATOP is able to generate code that automatically switches to call DMA and computation primitive using new parameter at the boundary. Second, if the boundary size is too small to apply any tensorized primitives, we apply a lightweight zero-padding to tensors to make it valid to call a tensorized primitive. Traditional padding method is to allocate a new storage buffer and copy the original data to it. In this way, copying overhead is nontrivial. swATOP adopts a lightweight zero-padding scheme by copying only boundary data to auxiliary buffers. When processing the boundary data, it automatically switches to the auxilary buffers, thus reducing the copy overhead.

4.6 Autotuner

Autotuner leverages a peformance model to identify the best candidate from all implementations. A naive design is a black-box autotuner, which generates code for all schedule IRs and picks the best one by collecting real execution time. However, since there may be thousands of different schedules in a schedule space, the black-box autotuner will take a long time. In order to quickly find the best solution, based on the prior knowledge of the architectural features, we propose a performance-model-based autotuner.

The SW26010 is a latency-oriented many-core architecture. Work [23] proposed a static performance model for it. Based on their analysis, the execution time of a DL operator can be estimated by two parts: time of DMA engine cost T_{DMA}, and time of instruction execution units cost $T_{compute}$. To achieve a fast online performance estimation, swATOP requires building cost models for them in advance.

Autotuner estimates DMA time according to the special memory organization designed for cache-free architecture. CPEs access the main memory in the unit of DRAM *transaction*. Therefore, the occurred memory transactions rather than data request size actually reflect the effective DRAM throughput. Even if just 1 Byte of a transaction is touched, the entire transaction will be transferred. Our cost model of DMA is shown in Eq. (1), which is determined by a latency term $T_{latency}$ (also known as start-up overhead) and a transmission term calculated by the ratio of the total memory access size to the theoretical peak memory bandwidth *PEAK_BW*. *block_num* is the number of continuous access blocks. *block_size* is the size of continuous access block in a strided memory access. *#CPE* is the number of CPE participated in DMA access, which is always 64 in our scenarios. $waste\_size_i$ is the waste data of the ith memory access padded on left and right boundaries in memory transactions. We assume the first block size is 128 Byte aligned, and *waste_size* of each block can be inferred by the stride size.

$$T_{DMA} = T_{latency} + \frac{\sum_{i=1}^{block\_num} block\_size + waste\_size_i}{PEAK\_BW/\#CPE} \quad (1)$$

$T_{compute}$ consists of the clock cycles cost on issuing instructions, and the idle cycles arisen from the Read After Write (RAW) hazard of instruction scheduling. By our instruction pipelining scheme mentioned in the Appendix, there should be no idle cycles in the innermost loop of GEMM primitive. However, there are still extra cycles wasted on initialization and finalization of the innermost

loop, the overhead of outer loops, and the latency to switch register communication pattern. Fortunately, given a vectorization approach, it is a linear function of three dimension parameters M, N, K, and irrelevant to data layout. We fit a linear function to estimate the computation time by collecting the execution time of GEMM operations using different dimension parameters. Eq.(2) shows the linear function to estimate a single GEMM primitive execution time. The loop order from the innermost to outermost is defined as K, M, N. vecM is a 0/1 value to indicate whether to do vectorization on M, otherwise do vectorization on N.

$$T_{SPM\_GEMM} = \alpha K + \beta \frac{KM}{vecM \times 4} + \gamma \frac{KMN}{4} + \delta. \quad (2)$$

$T_{compute}$ can be calculated by summing T_{SPM_GEMM} of all involved iterations.

Because DMA works in asynchronous way and we adopt software prefetching to overlap DMA and calculation, the total execution time $T_{overall}$ is the maximum of $T_{DMA}, T_{compute}$. Our performance-based autotuner first calculates the estimated performance of all implementations and chooses the best one to generate code.

4.7 code generator

Code generator is used to lower the IR into C and assembly code. Code generator performs some memory relative optimizations. For example, code generator analyzes the memory usage information in the IR and allocates all buffers into a single coalesced region.

5 EXPERIMENTAL RESULTS

5.1 Performance Evaluation

5.1.1 Evaluation of Convolution Operators. We compare the convolution (CONV) operators generated by swATOP with the best manual implementations, i.e. swDNN [7] for the Implicit CONV, and xMath [9] for GEMM routines in the Winograd and the Explicit CONV methods. For each CONV operator, we test performance when the batch-size is 1 (for inference), 32 and 128 (for training).

Cases in classic CNNs. We first investigate CONV operators in 3 classic convolutional neural networks (CNNs): VGG16 [19], ResNet [8], and Yolo [17]. Fig. 5 shows performance improvement on Implicit CONV. Designing Implicit CONV of batch-size=1 is complicated, there is currently no manually optimized version in swDNN. The swATOP is able to bridge such gap, and achieve similar performance as big batch versions. For the cases of batch-size=32,128 which can be compared, swATOP performs always better than swDNN, and the average speedup is up to **1.44,1.32**, respectively. Fig. 6 shows performance improvement on Winograd CONV. swATOP acheives an average speedup of **2.20, 2.35, 2.33** for batch-size=1,32,128. Fig. 7 shows performance on Explicit CONV. In 43 different cases, swATOP is better in 40 cases of batch-size=1, 29 cases of batch-size=32, and 32 cases of batch-size=128, and the best speedup is up to 15.2. The significant performance improvement of Winograd and Explicit CONV is mainly due to the fact that swATOP can figure out the best schedule strategies and dynamically picks the optimal tensorized primitives according to parameters. It can be seen that the speedup of small batch in Implicit and Explicit CONV is slightly greater than those of big batch. This proves the stability of swATOP to handle all cases, while manually optimized version focus on big-batch cases that have sufficient workload.

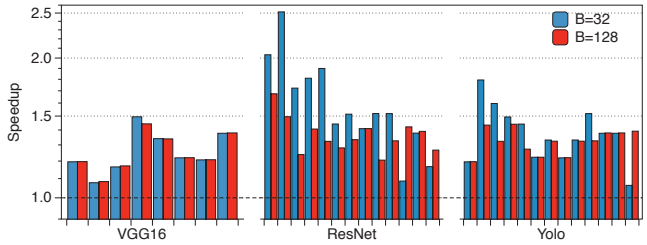

Figure 5: Performance improvement of swATOP for Implicit CONV on convolution layers of three networks (except the first layer of each network, because its input channel (Ni) is too small to be handled by implicit CONV).

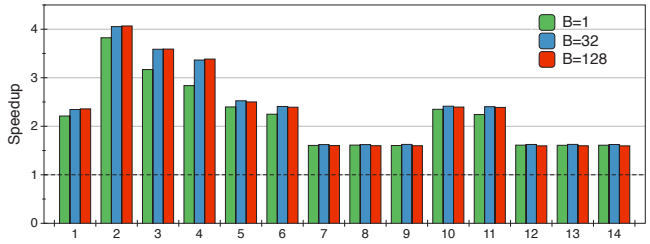

Figure 6: Performance improvement of swATOP on layers which Winograd CONV can be used.

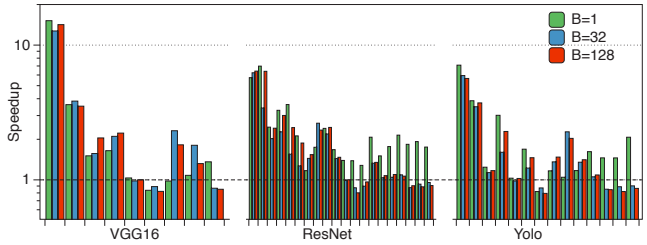

Figure 7: Performance improvement of swATOP for Explicit CONV on convolution layers of three networks.

Table 1: Evaluation on 225 parameter configurations. Faster (Slower) means swATOP is faster (slower) than the best manual version. Items indicates #cases(avg. speedup).

Batch	1		32		128	
	Faster	Slower	Faster	Slower	Faster	Slower
Implicit	75(+∞%)	0	75(+45%)	0	75(+44%)	0
Explicit	54(+21%)	21(-17%)	59(+23%)	16(-%22)	55(+26%)	20(-22%)
Winograd	75(+316%)	0	75(+295%)	0	75(+306%)	0

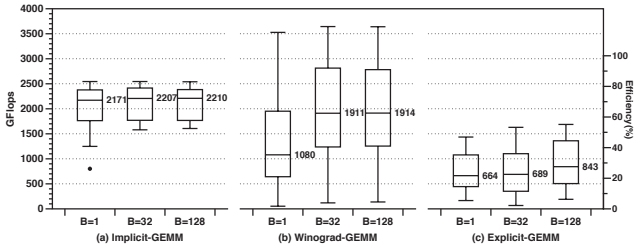

Figure 8: The overall performance and efficiency of three type of CONV operators with 225 parameters.

```
for B in 1 32 128;
 for Ni in 64 128 256 384 512;
  for No in 64 128 256 384 512;
   for Ro in 32 64 128 256;
    if[$Ni >= $No] ./test_swATOP $B $Ni $No $Ro
```

Listing 1: Parameter configurations for general test cases.

Cases for testing versatility. We benchmark swATOP for its versatility using the 225 parameter configurations produced from the script as List. 1. Tab. 1 summarizes the comparison between swATOP and the best handcrafted implementations. swATOP beats the manually optimized version in over 84% scenarios (100% in both Explicit and Winograd CONV and 75% in Explicit CONV). Note that a few cases of swATOP on Explicit CONV are slower than manual version, because these cases are large enough and just perfectly match the customized optimizations of manual version. Moreover, the average performance degradation (less than 25% for Explicit CONV) caused by swATOP is not significant compared to the performance improvement it brings (more than 300% for Winograd CONV).

Fig. 8 shows the performance and efficiency of swATOP individually. As can be seen, swATOP obtains high efficiency over different parameter configurations. On average, over 2.1 TFLOPS (70% efficiency) is obtained for Implicit CONV for both inference and training cases. The variance of Winogard method performance is relatively large. The best efficiency of Winograd is near 120% (Since the throughtput of Winograd method is evaluated using the number of GFLOPS required by direct convolution, the efficiency may exceed 100%). For training tasks, the average throughput is over 1.9 TFLOPS (60%) efficiency. For inference tasks, the efficiency is over 1 TFLOPS, due to the increasing of proportion of pre- and post-processing overhead. The average efficiency of Explicit CONV is relative low, so we only use it for cases where the other two methods cannot be applied.

5.1.2 Evaluation of Matrix-Multiplication. We evaluate the performance of swATOP for Matrix-Multiplication on 559 parameters generated from List. 2. Tab. 2 summarizes the performance comparison between swATOP and xMath. swATOP outperforms xMath in most cases, especially for unaligned parameters thanks to our optimization for boundary processing. Moreover, the average speedups brought by swATOP are up to 31.6% and 49.8%, while the performance loss is just 6.6% and 4.3%. For general matrix shapes, swATOP is able to to identify the most efficient schedule strategy and obtain significant performance improvement than handcrafted routines. However, for square-like matrix multiplications, which the xMath optimization is targeted on, swATOP is slightly worse.

Table 2: Comparison between swATOP and xMath on Matrix Multiplication. Faster (Slower) means swATOP is faster (slower) than xMath.

	Aligned		Unaligned	
	Count	Avg. Speedup	Count	Avg. Speedup
Faster	250	+31.6%	207	+49.8%
Slower	93	-6.6%	9	-4.3%

```
//test unaligned parameters requiring boundary processing
for M in 200 500 1000 2000 4000 8000;
 for N in 200 500 1000 2000 4000 8000;
  for K in 200 500 1000 2000 4000 8000;
   ./test_swATOP $M $N $K
//test aligned parameters without boundary processing
for M in 256 512 768 1024 2048 4096 8192;
 for N in 256 512 768 1024 2048 4096 8192;
  for K in 256 512 768 1024 2048 4096 8192;
   ./test_swATOP $M $N $K
```

Listing 2: Parameters for Matrix-Multiplication test cases.

5.2 Evaluation of Autotuner

Taking the tuning process of the most complex Implicit CONV operator as an example, we compare the time cost of performance-model-based autotuner and a black-box autotuner by brute-force searching. For a specific operator, The black-box autotuner runs every single schedule strategy of the schedule space to identify the optimal code, while our autotuner only runs the best strategy identified by the performance model. As shown in Tab. 3, the black-box autotuner costs 2~3 days to tune CONV layers of an entire CNN, while our autotuner only costs a few minutes. For a single CONV operator, the black-box autotuner costs several hours, while our autotuner costs less than 1 minute. The average speedups of VGG16, ResNet, and Yolo are up to **454x, 353x, and 365x**, respectively. Moreover, tuning CONV operator on GPU using TVM costs several hours, while our autotuner leverages the prior knowledge of SW26010 and dramatically reduce the tuning time.

Table 3: Tuning time of Implicit CONV for layers of three classic CNNs.

	Space Size		Black-box		swATOP	
	Total	Avg.	Total	Avg.	Total	Avg
VGG16	4068	454.2	47h 50m	5h 20m	6m 21s	< 1m
ResNet	7064	353.7	83h 6m	4h 9m	14m 7s	< 1m
Yolo	5112	365.1	60h 10m	4h 18m	9m 53s	< 1m

Fig. 9 illustrates the differences between the optimal version identified by our autotuner and real best version found by brute-force search on Implicit CONV. It shows the ratio of the performance of swATOP to the real best performance. On average, the performance-model-based autotuner brings **less than 2%** performance loss. Even for the worse case, the performance loss is still less than 8%. Negligible performance difference proves the high accuracy of our static performance model.

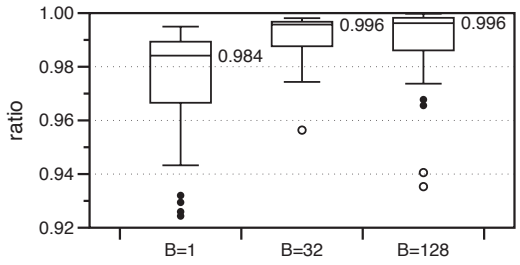

Figure 9: the ratio of the resulting performance of implementation generated by autotuner to the best performance on 225 different parameters from List. 1.

5.3 Evaluation of IR optimizer

5.3.1 Hiding Memory Access Latency Evaluation. To evaluate the improvement of our automatic memory latency hiding technique, we compare the performance of auto-prefetching and a baseline without software prefetching in Fig. 10. Given Ro, Co, we select 8 parameters where baseline version performs best. Even for the best cases of baseline, software prefetching can significantly improve performance by **an average of 65.4%**. This proves that automatical software prefetching can highly improve the overall performance on SW26010, and swATOP can exploit the ability of ovelapping DMA and calculation.

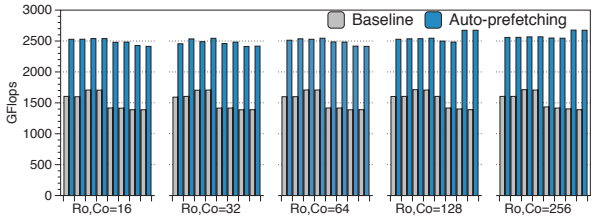

Figure 10: Auto-prefetching vs. Baseline.

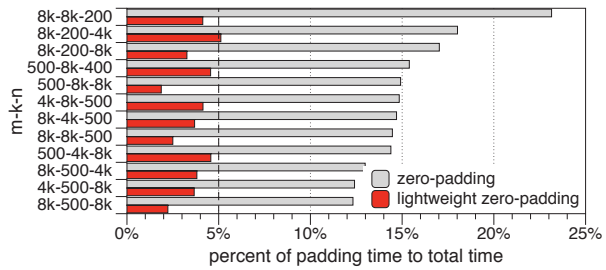

Figure 11: Lightweight Zero-padding vs. zero-padding.

5.3.2 Boundary Processing. We evaluate the padding overhead for 216 matrix multiplication benchmarks with code from unaligned parameters test of List. 2. All cases require zero padding on all three dimensions M, N, K. Fig. 11 presents the time of both lightweight and traditional zero-padding of those cases whose boundary processing overhead is more than 10%. Lightweight zero-padding dramatically reduces the amount of data copied, and significantly reduces the boundary processing overhead to **less than 5%**.

6 RELATED WORK

There have been many previous works on optimizing DL operators and generate code for various hardware (e.g., CPU, GPU, FPGA).

Glow [18] lowers DL operators into fine-grained operations, and performs operator stack and memory reuse optimizations. TVM [5] extracts context relation features of loops of DL operators, and combines Halide [16] with a machine learning model to automatically optimize operators on many devices. These frameworks abstract the common techniques of optimization and aim at generate code comparable to manually optimized libraries. On the one hand, above frameworks leverage mature tool chains such as LLVM and polyhedral tools, which usually have not been adapted to new hardware like SW26010. On the other hand, due to the lack of assembly level optimization, they cannot make full use of important architecture-specific features, which are usually optimized in handcrafted assembly language and play an important role in optimization. swATOP leverages these architecture-specific features by tensorized primitives, and can produce significantly more efficient code than manually optimized versions. Moreover, TVM uses machine learning method to speedup autotuning. High performance libraries, such as ATLAS [22], SPIRAL [15], widely used in scientific computing area, use various optimization methods such as Powell search, exhaustive search combined with pruning, evolutionary search etc. swATOP adopts an efficient but effective performance model to accelerate autotuning and obtains the best implementation in minutes, while projects, like TVM, require several hours.

Polyhedral-based methods [2, 3] convert the nested loops to affine space, and make efforts to find an high-level transformation to maximize parallelism and locality, and minimize communication cost. PPCG [21] deploys parallel code for CUDA. Pencil [1] presents a DSL and can automatically generate optimized OpenCL and CUDA code. Recently, Tensor Comprehensions [20] uses polyhedral model and genetic search techniques to autotune tensor operators on CPU and GPU. Since there are no effective theory to express hardware features with affine function, polyhedral-based methods are difficult to produce optimal implementation towards specific hardware. swATOP takes advantage of both high-level transformation and hardware features to generate best implementation in practice.

7 CONCLUSION

In this paper, we present an automated framework called swATOP in combination with a tensorized algorithm abstraction to tune and generate DL operators. It not only reduces huge engineering burden of manual optimization, but also brings excellent performance improvement. According to our experimental results, swATOP is better than the best handcrafted code in 1095 of 1234 cases (over 88%) for CONV and GEMM operators. The performance-model-based autotuner of swATOP is able to reduce tuning time cost of a operator from hours to minutes and obtain an almost optimal implementation (less than 8% in the worse case). Our method of combining architecture-specific tensorized primitives and autotuning techniques can be applied to other new hardwares.

8 ACKNOWLEDGMENTS

This work was supported in part by the National Key Research & Development Plan of China (grant no. 2017YFB0202204), by the National Natural Science Foundation of China (grant no. 51761135015 and 91530323), by the China Postdoctoral Science Foundation (Grant No. 2018M641359), and by the Center for High Performance Computing and System Simulation of Pilot National Laboratory for Marine Science and Technology (Qingdao).

9 APPENDIX

This section shows our optimization on GEMM primitive: C += A×B, where A, B and C are resident in LDM space. As illustrated in Figure 12, our design leverages the following architectural characteristics.

Register Communication: The matrices are partitioned vertically and horizontally in a uniform way into 64 pieces and distributed across the CPE cluster. Therefore, each CPE can only obtain 1/8 of the final result by performing GEMM using 1/64 local data. To achieve the final result, remote data should be fetched by register communication from CPEs located in the same row and column.

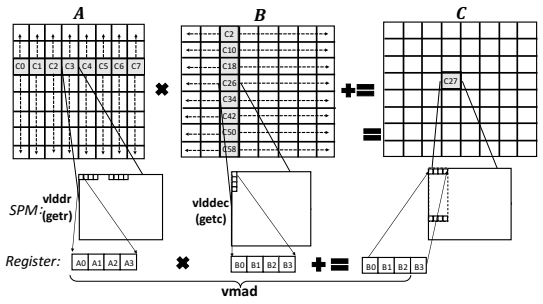

Figure 12: Design overview of GEMM primitives.

Vectorization: To achieve the best performance, operations should be conducted with vectorized MAC operation `vmad`. For this reason, we used SIMD instructions for memory access. Two sets of unique instructions of SW-ISA (Instruction Set Architecture) are used. Set1: `vlddr`/`vlddc` are to load four floating-point (FP) data from SPM as a vector and broadcast them through row/column communication bus. The dimension is accessed in this way is called vectorized dimension. Set2: `vlddec`/`vldder` are to load a single FP data from SPM, extend them into a vector of four copies and broadcast them through row/column communication bus.

Register Blocking: To increase data reuse inside registers, a 4×4 register blocking scheme [7] is adopted. 16 element vectors of matrix C is fixed in registers during the execution of the innermost loop, so as to reduce repeated read and write.

Pipelining Instruction Execution: Our GEMM primitive is able to fully utilize instruction execution units. It is able to finish 16 `vmad` operations in 16 cycles by carefully avoiding Read After Write hazard, as well as increasing instruction-level parallelism of two instruction execution pipelines.

The GEMM design adopting the above optimization techniques has **eight variants** considering the following differences. First, both A and B in SPM can be stored in column-major or row-major layout. Second, the dimension to apply vectorization can be different. Third, vectorization may be achieved along the nested loop dimensions M or N. We have adopted a template-based method to generate eight different optimized assembly kernels.

REFERENCES

[1] Riyadh Baghdadi, Ulysse Beaugnon, et al. 2015. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 138–149.
[2] Uday Bondhugula, Sanjeeb Dash, Oktay Gunluk, and Lakshminarayanan Renganarayanan. 2010. A model for fusion and code motion in an automatic parallelizing compiler. In *Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on*. IEEE, 343–352.
[3] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, Vol. 43. ACM, 101–113.
[4] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft.
[5] Tianqi Chen, Thierry Moreau, 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.
[6] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, et al. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).
[7] Jiarui Fang, Haohuan Fu, Wenlai Zhao, Bingwei Chen, Weijie Zheng, and Guangwen Yang. 2017. swDNN: A library for accelerating deep learning applications on sunway taihulight. In *Parallel and Distributed Processing Symposium (IPDPS)*.
[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
[9] Lijuan Jiang, Chao Yang, Yulong Ao, et al. 2017. Towards highly efficient DGEMM on the emerging SW26010 many-core processor. In *Parallel Processing (ICPP), 2017 46th International Conference on*. IEEE, 422–431.
[10] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 75.
[11] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition*.
[12] Liandeng Li, Jiarui Fang, Haohuan Fu, Jinlei Jiang, et al. 2018. swCaffe: A Parallel Framework for Accelerating Deep Learning Applications on Sunway TaihuLight. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*.
[13] X. Li, G. Zhang, H. H. Huang, Z. Wang, and W. Zheng. 2016. Performance Analysis of GPU-Based Convolutional Neural Networks. In *2016 45th International Conference on Parallel Processing (ICPP)*. 67–76.
[14] Changxi Liu, Hailong Yang, Rujun Sun, Zhongzhi Luan, and Depei Qian. 2019. swTVM: Exploring the Automated Compilation for Deep Learning on Sunway Architecture. *arXiv preprint arXiv:1904.07404* (2019).
[15] Markus Puschel, José MF Moura, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.
[16] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines.
[17] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.
[18] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*.
[19] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[20] Nicolas Vasilache, Oleksandr Zinenko, et al. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv preprint arXiv:1802.04730* (2018).
[21] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* (2013).
[22] R Clinton Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *Supercomputing, 1998. SC98. IEEE/ACM Conference on*. IEEE, 38–38.
[23] Shizhen Xu, Yuanchao Xu, Wei Xue, et al. 2018. Taming the" Monster": Overcoming program optimization challenges on SW26010 through precise performance modeling. In *2018 International Parallel and Distributed Processing Symposium*.
[24] Zhigeng Xu, James Lin, and Satoshi Matsuoka. 2017. Benchmarking sw26010 many-core processor. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 743–752.