Efficient Locality-aware Instruction Stream Scheduling for Stencil Computation on ARM Processors

Shanghao Liu
Beihang University
Beijing, China
somehow6@buaa.edu.cn

Hailong Yang
Beihang University
Beijing, China
hailong.yang@buaa.edu.cn

Xin You
Beihang University
Beijing, China
youxin2015@buaa.edu.cn

Zhongzhi Luan
Beihang University
Beijing, China
07680@buaa.edu.cn

Yi Liu
Beihang University
Beijing, China
yi.liu@buaa.edu.cn

Depei Qian
Beihang University
Beijing, China
depeiq@buaa.edu.cn

Abstract

Stencil computation is one of the fundamental computational patterns in scientific computing, commonly adopted in solving partial differential equations (PDEs) and a wide range of application fields. However, due to the memory-bound nature, it is challenging to achieve satisfactory performance on the ARM many-core processors with complex computation and memory hierarchies. In this study, we propose *independent instruction stream scheduling* with the Serial-FMA to Tree-Based Reduction (SFTBR) technique to decompose the stencil computation into multiple independent instruction streams for improved instruction-level parallelism. Furthermore, we propose a *locality-aware block scheduling* technique for locality-aware multi-level thread parallelism to address the complexities of cache and memory hierarchies on modern ARM many-core processors. Based on the above techniques, we implement a domain-specific compiler, AOStencil, to automatically generate optimized stencil codes on ARM many-core processors with genetic-algorithm-driven parameter tuning. Our evaluation results demonstrate that AOStencil achieves up to 4.39× speedup over the state-of-the-art domain-specific compilers on Kunpeng and Phytium platforms.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725760

CCS Concepts

• **General and reference** → **Performance**; • **Software and its engineering** → **Compilers**; • **Computer systems organization** → **Multicore architectures**.

Keywords

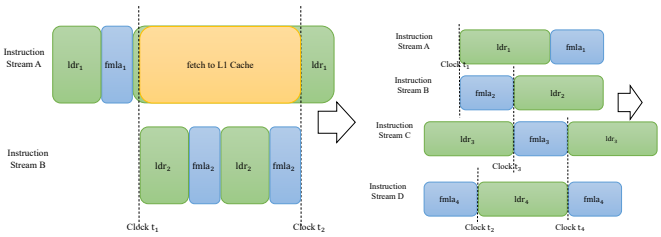
Stencil Computation, Optimization Strategies, Domain Specific Language, Many-core Architecture

ACM Reference Format:
Shanghao Liu, Hailong Yang, Xin You, Zhongzhi Luan, Yi Liu, and Depei Qian. 2025. Efficient Locality-aware Instruction Stream Scheduling for Stencil Computation on ARM Processors. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3721145.3725760

1 Introduction

Stencil computation is one of the fundamental computational patterns in scientific computing [9], crucial for addressing a broad spectrum of partial differential equations (PDEs) that are adopted in various application domains, including image processing, computational fluid dynamics, climate modeling, and nanoscale science. Moreover, with the advent of emerging deep learning techniques, stencil computation has broadened its applicability beyond scientific fields to widely deployed deep neural models such as convolutional neural networks (CNNs). Specifically, stencil computation updates a data grid by employing a predefined pattern of adjacent values to compute new values across time or space iteratively. Stencil computation often exhibits a relatively low computational intensity, which is generally categorized as memory-bounded [18, 29]. In practice, stencil computations often employ tiling techniques [2, 7, 13, 15] to enhance its performance with improved data locality. Unfortunately, the complexity of modern ARM many-core processors with diverse microarchitecture designs and memory hierarchies

poses significant challenges for optimizing stencil computations.



(a) Out-of-order execution (b) Dual-issue

Figure 1: Illustration of instruction-level parallelism with multiple individual instruction streams.

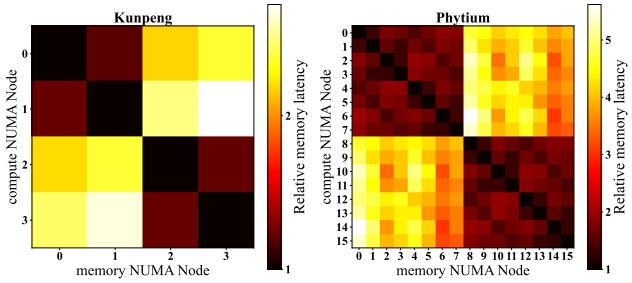


Figure 2: Latency across NUMA nodes on different Arm many-cores systems.

Modern processors commonly adopt superscalar microarchitectures that support instruction-level parallelism such as *out-of-order execution* (OoOE) and *multi-issue*, which can simultaneously execute multiple data-independent instructions. For simplicity without losing generality, Figure 1a demonstrates the enhancements achieved via OoOE with two individual instruction streams. When Stream A requires prefetching data from the Last Level Cache (LLC) or memory with high latency, the core can execute other streams (Stream B) that are data-independent of the current instruction stream. This approach minimizes idle CPU cycles by enabling concurrent execution of individual instruction streams. Meanwhile, modern processors can issue multiple instructions simultaneously within a single clock cycle (e.g., *dual-issue* to issue two instructions), thus boosting performance. As shown in Figure 1b, for dual-issue hardware, Streams A and B simultaneously issue the `ldr` and `fmla` instructions at clock t_1, and similar dual-issue operations occur at clocks t_3 and t_4, which enhances the Instructions per Cycle (IPC) during these specific cycles. Similarly, such a technique is effective only if there are multiple individual instruction streams without data dependencies operating simultaneously. The optimal number of multiple streams for exploiting multi-issue depends crucially on the execution latency of specific instructions such as `ldr` and `fmla` during stencil computation.

However, for ARM-based processors, the diversity in microarchitecture implementations presents a significant challenge in optimizing stencil computations via instruction-level parallelism. For instance, the ARM Cortex-A78 microarchitecture, which adheres to the ARMv8-A standard, features a 4-wide superscalar pipeline with 13 out-of-order execution (OoOE) pipelines, enabling it to issue two SIMD instructions concurrently (dual-issue) [3]. In contrast, the Kunpeng 920 microarchitecture, also based on the ARMv8-A standard, boasts a 4-wide superscalar pipeline but with only 8 OoOE pipelines, yet it is capable of issuing up to four SIMD instructions simultaneously [34]. Therefore, for the ARM ecosystem, the processors even with the same ARM standard differ in key specifications such as instruction latency, the size of the re-reorder buffer, the number of pipelines in out-of-order execution, and the number of dispatches supported by multi-issue mechanisms. These architectural differences necessitate distinct instruction scheduling strategies to effectively optimize stencil computations across various ARM implementations, thereby complicating the development of high-performance computing libraries for ARM platforms.

Unfortunately, previous stencil compilers [11, 18, 22, 36] have predominantly focused on optimizations on individual microarchitectures, often neglecting the impact of diverse microarchitectures on the optimization strategies. Besides, existing works also typically optimize stencils by unrolling loops within a stencil pattern using a single instruction stream with vectorization [24, 35], which highly depends on general-purpose compilers for instruction scheduling. Due to the limited hardware resources and lack of domain knowledge, such an instruction stream can be excessively long and even data-dependent which will harm the effectiveness of instruction-level parallelism. Therefore, it is essential to leverage the domain knowledge of stencil computation to decompose specific stencil computation into multiple independent instruction streams, which can further optimize the instruction scheduling for ARM-based processors.

To address these challenges, we introduce *independent instruction stream scheduling* with the Serial-FMA to Tree-Based Reduction (SFTBR) method, which decomposes the execution of stencil computations into multiple independent instruction streams to leverage advanced instruction-level parallelism. Notably, some `fmla` operations are divided into `vmul` and `vadd` instructions. According to ARM programming guide [3], a `fmla` instruction has a latency of 4 cycles, whereas `vmul` and `vadd` together incur a total latency of 5 cycles. Typically, on most processors, `fmla` executes slightly

faster than the combined `vmul` and `vadd`. However, AOStencil can decompose certain `fmla` instructions to generate *nps* independent instruction streams, thereby leveraging the processor's microarchitecture features such as OoOE and multi-issue capabilities. Within a single instruction stream, `fmla` operations execute sequentially with lower latencies. Multiple instruction streams can overlap memory access latency with `fmla` operations from other independent instruction streams, resulting in performance gains for stencil programs that are typically memory-bound. By fine-tuning the *nps* parameter, AOStencil can adapt to the hardware microarchitecture design, stencil access patterns, and system memory access latency. Without the above observations, general-purpose compilers such as GCC cannot accurately predict whether the benefits of splitting `fmla` instructions outweigh the costs, and therefore cannot perform such optimizations.

In addition, we propose a *locality-aware block scheduling* technique for locality-aware multi-level thread parallelism to address the complexities of cache and memory hierarchies in modern ARM many-core systems. Modern ARM-based processors are trending towards many-core architectures with complex cache and memory hierarchies. To integrate numerous computing cores within a system, these computing architectures often employ a cache hierarchy and memory hierarchy to build multi-level memory structures, typically including multiple levels of private and shared caches (L1, L2, and L3), cache-coherent Non-Uniform Memory Access (ccNUMA) nodes, and sockets. Such complexity introduces diverged performance impacts on memory load latency, which is crucial for memory-bounded stencil computations. For instance, Figure 2 illustrates the memory latency for ARM many-core systems such as the Kunpeng and Phytium platforms [12, 42], where the horizontal axis represents the memory NUMA node, and the vertical axis indicates the computation NUMA node. The heatmap reflects the normalized latency relative to the lowest observed latency on each platform. The results indicate that the fastest memory loads occur in local ccNUMA memory nodes, followed by adjacent local ccNUMA nodes, other ccNUMA nodes from the local socket, and finally ccNUMA nodes from other sockets. In the Kunpeng system, the maximum discrepancy of memory latency can reach nearly 3×, whereas, in the Phytium system, it can achieve 6×. Therefore, to achieve efficient stencil computation, it is essential to fully leverage the multi-level cache and memory hierarchy. Locality-aware block scheduling requires adopting the domain knowledge of stencil computation partitioning and communication, as well as an understanding of the system interconnect topology.

To accommodate diverse stencil patterns, we develop a domain-specific compiler, AOStencil, to automatically generate optimized stencil codes for ARM many-core systems from intuitive stencil descriptions. Specifically, the AOStencil compiler consists of a micro-kernel schedule optimizer and a block schedule optimizer enhanced by *independent instruction stream scheduling* and *locality-aware block scheduling*, respectively. The micro-kernel schedule optimizer also employs a genetic algorithm (GA) to automatically fine-tune optimization parameters for single-thread performance, while the block schedule optimizer maximizes multi-thread performance with better data locality.

Specifically, this paper makes the following contributions:

- We propose *independent instruction stream scheduling* with the Serial-FMA to Tree-Based Reduction (SFTBR) method to decompose the execution of stencil computation into multiple independent instruction streams for improved instruction-level parallelism.
- We propose *locality-aware block scheduling* for locality-aware multi-level thread parallelism to address the complexities of cache and memory hierarchies in modern ARM many-core systems, facilitating high-performant stencil computation.
- Based on the above techniques, we develop a novel domain-specific compiler, AOStencil, to automatically generate optimized stencil codes for ARM many-core systems with genetic algorithm-driven parameter tuning. AOStencil adopts a micro-kernel schedule optimizer and block schedule optimizer for the automatic optimization of diverse stencil patterns.
- Our comprehensive evaluation results demonstrate that the AOStencil achieves up to 4.39× speedup over the state-of-the-art domain-specific compilers on ARM many-core systems such as Kunpeng and Phytium.

The remainder of this paper is organized as follows. Section 2 provides the background of stencil computation and ARM many-core systems. Section 3 offers a general design overview and Section 4 dives into the detailed optimization applied in the AOStencil. The evaluation results of AOStencil are detailed in Section 5. We discuss the related work in Section 6 and conclude in Section 7.

2 Background

2.1 Stencil Computation

Stencil computation involves updating the value at each point on a structured grid based on a predefined stencil pattern, known as a stencil, which incorporates the values of neighboring points. This process involves both spatial and temporal dimensions: the spatial dimension defines the neighborhood of points (the stencil pattern), while the temporal dimension refers to the iterative application of the stencil over multiple time steps to simulate system evolution. The shape and complexity of the stencil can vary depending on the application, ranging from simple 5-point or 9-point

stencils in 2D grids to more complex 27-point stencils in 3D grids. These stencils can take forms such as star, box, or other patterns, depending on which neighboring points influence the computation. Stencils are typically memory-bound programs, meaning their performance is often constrained by memory access speeds rather than computational complexity. Due to the intensive memory access patterns, efficient utilization of cache and memory bandwidth is critical for optimizing stencil performance.

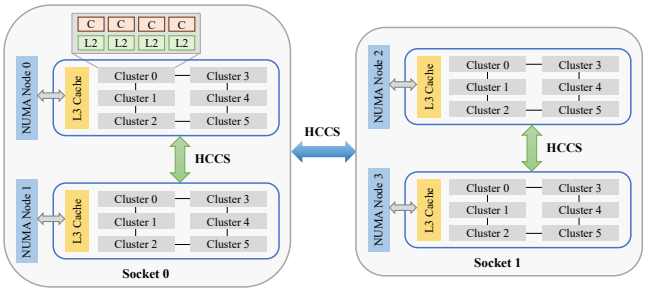
2.2 Instruction-level Parallelism

Instruction-level parallelism (ILP) in modern micro-architectures is critical for optimizing performance. Both out-of-order execution and multiple-issue mechanisms such as dual-issue play pivotal roles in exploiting ILP by allowing multiple instructions to execute concurrently, thus enhancing throughput and efficiency.

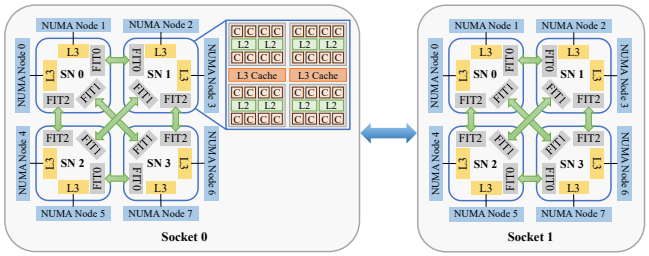
Out-of-order execution (OoOE), as demonstrated in Figure 1a, enables processors to dynamically reorder instructions to mitigate latency issues, such as long memory fetches. In this stencil example, two independent instruction streams, A and B, are managed such that delays in one stream (e.g., the ldr_1 instruction in Stream A) do not stall overall execution. By prioritizing instructions from Stream B while Stream A is delayed, the processor reduces idle time and maintains high utilization of its execution units. Once the delayed instruction (e.g., ldr_1) is ready, Stream A resumes execution seamlessly. This ability to handle independent streams out of their original order is fundamental to modern processors, allowing them to execute multiple instructions in parallel and thus significantly boost performance.

In addition, multi-issue mechanisms, such as dual-issue configurations, as demonstrated in Figure 1b, further enhance processor throughput by allowing the issuance of two instructions per clock cycle. In the context of this stencil example, a fused pair of instructions—such as *ldr* (load) and *fmla* (fused multiply-add)—can be issued concurrently across multiple independent streams (A, B, C, D). However, multi-issue is subject to specific constraints, particularly those related to the NEON unit. These constraints permit the simultaneous dual issuance of a load/store or certain instruction types alongside a NEON data-processing instruction in ARM architecture [4]. The use of multi-issue significantly improves the instructions per cycle (IPC) metric by facilitating the parallel execution of different instruction types across multiple streams, thereby maximizing instruction-level parallelism.

The OoOE and multi-issue mechanisms together highlight the substantial potential of ILP in modern micro-architectures. These techniques work in tandem to reduce latency and enhance the utilization of execution units. However, achieving optimal performance through ILP on diverse Arm hardware requires adapting instruction scheduling strategies to account for variations in the number of OoOE pipelines and multi-issue capabilities across different architectures.



(a) Kunpeng processor



(b) Phytium processor

Figure 3: Hardware architectures of Kunpeng and Phytium platforms.

2.3 Multi-level Thread Parallelism

The trend in modern processors towards increasing core counts, while effectively managing cache and memory hierarchies, has become a defining feature of contemporary multi-core architectures. Notable examples include the AWS Graviton4 (with up to 192 cores), Huawei Kunpeng 920 (up to 256 cores), Nvidia Grace CPU Superchip (scaling to 288 cores), and the Phytium FT-2000, used in the Tianhe-3 supercomputer system (with up to 128 cores).

In this study, the Kunpeng platform and the Phytium platform serve as the primary platforms for analyzing multi-level thread parallelism. The Kunpeng platform, utilizing Kunpeng 920 processors (Figure 3a), consists of two sockets, each containing two ccNUMA nodes. These nodes are interconnected via Huawei's Cache-Coherent System (HCCS), which ensures efficient communication and data consistency across nodes. Each ccNUMA node comprises six cluster modules integrated into a single chiplet package, with each module containing four independent compute cores. Every core is equipped with a dedicated L2 cache, enabling fast, localized data access and reducing latency for computing tasks.

Similarly, the Phytium platform, powered by Phytium S2500 many-core processors (also known as Matrix processors), integrates 128 compute cores operating at 2.0 GHz. As depicted in Figure 3b, the system topology organizes these cores into four Supernodes (SNs), interconnected through a scalable on-chip communication network. Each Supernode is divided into four panels, with each panel housing eight compute cores. The cache architecture is hierarchical, with every four cores sharing an L2 cache, and all eight cores within a panel contributing to a shared L3 cache. Both the Kunpeng and Phytium platforms exemplify the intricate cache and memory hierarchies that are central to modern many-core architectures. Multi-level thread parallelism is essential for fully exploiting the potential of these architectures, particularly in data-intensive and highly parallel applications such as stencil computations. Optimizing parallel execution across multiple levels of caches, ccNUMA nodes, and sockets is crucial to enhancing performance and scalability in next-generation HPC systems.

3 Design Overview

The AOStencil compiler comprises two principal modules: the frontend and the backend, as illustrated in the design overview shown in Figure 4. The architecture design of the AOStencil DSL compiler is inspired by Halide's design philosophy, which separates computation from scheduling during optimization. In the AOStencil compiler, the connection between the frontend and backend is facilitated through a stencil Intermediate Representation (IR) that encapsulates stencil patterns. The backend is composed of several components, including micro-kernel schedule optimizer, locality-aware block schedule optimizer, and code generation.

3.1 Frontend

The AOStencil compiler processes a domain-specific language (DSL) designed for defining stencil patterns and the data properties of tensors used in computational tasks. The DSL specifies the stencil kernel's dimensions, offset indices for memory access, and coefficients for stencil elements, while also ensuring the integrity of the input tensor by defining its data type and shape. The compiler translates the DSL into Stencil Intermediate Representation (IR), composed of a sparse tensor storing the stencil pattern and associated attribution data (such as input tensor details, halo length, and bias). The sparse tensor encodes the offset values of coefficients, leaving non-used positions as zeros, as depicted in Figure 4. This structured representation of the stencil allows for feature extraction and subsequent optimizations, improving computational efficiency in stencil operations.

3.2 Backend

The AOStencil compiler backend processes the Stencil IR through micro-kernel and locality-aware block schedule optimizers to generate an optimized kernel, which is then converted into a C function API. *1) Micro-kernel schedule optimizer* enhances vector-folding, independent instruction stream scheduling, tiling, and loop unrolling within the stencil computations. The AOStencil compiler employs evolutionary algorithms to tune parameters such as the number of multiple streams, tiling size, and unrolling size. *2) Block schedule optimizer* uses stencil attribution to hardware-aware locality memory optimization, incorporating the locality-aware block scheduling and halo exchange specification. It statically divides blocks for local storage and optimizes memory-IO latency for inter-block communication. *3) Code generation* translates the optimized kernel into calls to pthread and libnuma libraries for parallelization and memory management with generated standard C codes. The compiler leverages Ahead-Of-Time (AOT) compilation for high-performance stencils on ARM platforms.

4 Methodology and Implementations

The AOStencil compiler incorporates multi-tiered end-to-end optimizations to achieve high-performance stencil code on large-scale multicore systems, primarily facilitated through micro-kernel schedule and locality-aware block schedule optimizers.

4.1 Micro-kernel Schedule Optimizer

At the core of stencil pattern optimization, micro-kernel schedule optimizer enhances micro-kernel performance through two key approaches, including *independent instruction stream scheduling* and *micro-kernel auto-tuning*. First, it leverages SIMD instruction capabilities by employing vector-folding optimization techniques and the SFTBR method to exploit instruction-level parallelism. The SFTBR method enables parallelization of multiple instruction streams by taking advantage of ILP capabilities such as OoOE and multi-issue, ultimately improving multi-stream processing efficiency. Second, in the context of single-thread stencil computation, micro-kernel schedule optimizer utilizes techniques such as tiling and loop unrolling. To achieve optimal performance, the micro-kernel schedule is further refined through the use of a genetic algorithm (GA) to optimize key parameters, including tile size, unroll size, and the number of streams in the SFTBR method. Notably, to maximize throughput during the parameter fine-tuning process for cache and memory hierarchy systems, the candidate stencil code, generated with the candidate parameters for a specific locality block is tested on an isolated hardware block, ensuring the use of an independent cache system (including L1, L2, and L3 caches).

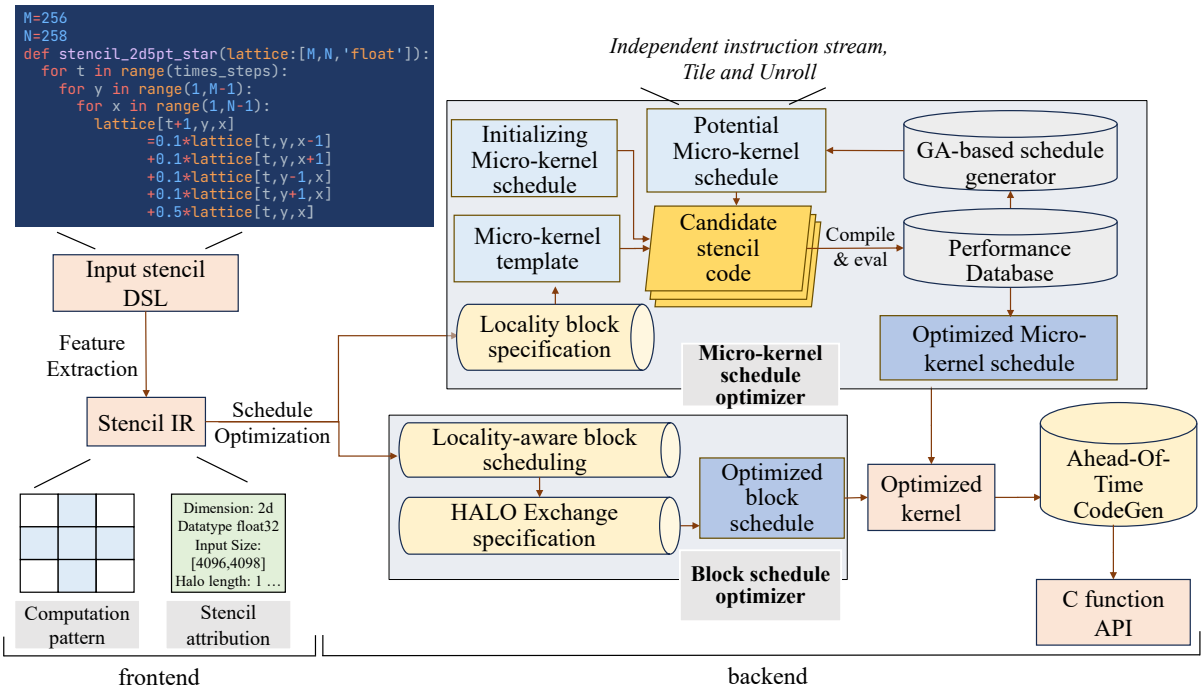


Figure 4: The design overview of the AOStencil compiler.

Algorithm 1 The basic stencil computation

Require: $A[K][M][N]$ ▷ Input 3D array
Ensure: $B[K][M][N]$ ▷ Output 3D array
1: **for** $z = 1$ **to** $K - 1$ **do**
2: **for** $y = 1$ **to** $M - 1$ **do**
3: **for** $x = 1$ **to** $N - 1$ **do**
4: **for** element $\in$ neighbors$(A[z][y][x])$ **do**
5: $B[z][y][x] +=$ coef $\times$ element
6: **end for**
7: **end for**
8: **end for**
9: **end for**

Algorithm 2 The SFTBR method

1: **for** $x = 1$ **to** $N - 1$ **step** vSize **do**
2: # Instruction-level parallelism
3: **for** $i = 0$ **to** $nps - 1$ **do**
4: vElems$_i \subseteq$ neighbors$(A[z][y][x])$
5: $vRegs_i \leftarrow$ vmul(coef, vElems$_i[0]$)
6: **for** $j = 1$ **to** $n - 1$ **do**
7: $vRegs_i \leftarrow$ fma(coef, vElems$_i[j], vRegs_i$)
8: **end for**
9: **end for**
10: $vReg \leftarrow$ treeBasedReduction$(vRegs)$
11: vstr$(\&B[y][x], vReg)$
12: **end for**

4.1.1 Independent Instruction Stream Scheduling. The basic stencil computation is outlined in Algorithm 1. Here, K, M, and N denote the dimensions of the kernel computation shape, excluding any HALO or boundary regions. The function *neighbors* is used to extract the requisite neighbor elements from the input tensor A that are essential for the specialized stencil kernel computation pattern. Notably, the stencil kernel computation pattern is also encapsulated within a for loop, which is targeted for independent instruction stream scheduling.

Vector folding employs the Single Instruction, Multiple Data (SIMD) technique to parallelize the nested loop as indicated in line 3 of Algorithm 1, effectively folding the loop. This approach enables the simultaneous computation of multiple data operations within an instruction cycle, significantly augmenting the throughput of the stencil kernel computation. The vector folding method [35] substantially improves the computational efficiency and performance of the stencil kernel computation.

After implementing vector folding, there are two prevalent methods for arranging vector instructions for a stencil point

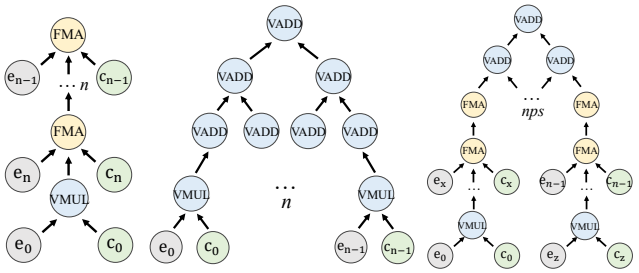


Figure 5: The illustration of three instruction stream methods, including (a) serial FMA, (b) tree-based reduction, and (c) SFTBR.

computation at the innermost loop: serial-FMA and tree-based reduction. The AOStencil compiler adapts the SFTBR method, which integrates these two approaches, adjusting the multi-stream settings via the parameter *nps* to adapt different stencil patterns and hardware micro-architectures.

The *serial-FMA* method employs a sequence of Fused Multiply-Accumulate (FMA) operations to compute stencil neighbor values in sequential order, as shown in Figure 5a. This ensures precise calculations with n-point stencils, requiring only n multiplications and additions. Hardware-level fusion of FMA operations reduces clock cycles compared to separate instructions. However, this sequential nature limits the use of the features of instruction-level parallelism, as seen in the default instruction layout generated by the GCC compiler optimization.

The *tree-based reduction* method uses an array of vector registers to first perform a vector multiplication between coefs and the neighbors in the stencil pattern, followed by a binary tree reduction for summation, as shown in Figure 5b. This advantage lies in its potential for OoOE and dual-issue. Since the computations at each node of the tree do not depend on the results of other nodes at the same level. However, this approach requires n multiplications and $2n - 2$ additions, utilizing a higher number of vector registers.

The *Serial-FMA to tree-based reduction* (SFTBR) method integrates the serial-FMA and tree-based reduction approaches, as detailed in the pseudo-code 2. In the SFTBR optimization, the parameter *nps* is used to determine the number of instruction streams in constructing the binary reduction tree. As depicted in Figure 5c, SFTBR first performs multi-stream FMA computations, followed by a tree-based reduction of the multiple vector registers obtained in streams of FMA. By combining the serial-FMA and tree-based reduction methods, SFTBR achieves multiple instruction streams that are independent and capable of OoOE and dual-issue, adding $2nps - 2$ addition operations. When *nps* equals 1, the SFTBR method defaults to serial-FMA. When *nps* equals the number of points in the stencil pattern, it transforms into the tree-based reduction method. During compilation, the AOStencil compiler unrolls the innermost loop to enhance execution efficiency in Algorithm 2.

4.1.2 Micro-kernel Auto-tuning. Kernel optimization for single-thread stencil computations focuses on optimizing the computation loops using techniques like tiling and unrolling. Tiling divides the iteration space into smaller cache-friendly blocks, enhancing data reuse across the x, y, and z dimensions. Unrolling expands the loop body to reduce loop control overhead, with the optimal factor dependent on the processor's vector registers and stream depth. Parameters such as tiling size, unrolling size, and the *nps* from the independent instruction stream scheduling in micro-kernel optimization greatly impact the performance of code generated by the AOStencil compiler.

This targeted testing is essential, as the fine-tuning of micro-kernel parameters is primarily dependent on the processor's cache. The specification of locality blocks ensures that the optimization process is closely tailored to the characteristics of the target hardware node, encompassing the individual cache hierarchies, including L1, L2, and L3 caches. To support this, the AOStencil compiler performs independent evaluations of micro-kernel scheduling on each node with its distinct cache configuration, thereby enabling parallel assessments across the entire system. The subsequent section provides a comprehensive explanation of the micro-kernel scheduling auto-tuning process, as illustrated in Figure 4.

1) Locality Block Specification: The input size of the stencil intermediate representation (IR) is partitioned to correspond with the evaluation of a locality hardware node, such as a ccNUMA node in a ccNUMA architecture, which possesses exclusive multi-level cache storage. This partitioning ensures that the computation is aligned with the distinct cache hierarchy of the node, facilitating optimized performance by leveraging the node's independent cache resources.

2) Micro-kernel Template: A micro-kernel template is generated from the stencil IR NUMA block Specification to provide a code template for subsequent micro-kernel schedule code generation.

3) Initializing Micro-kernel Schedule: In a genetic algorithm, a sufficiently uniform initial sample space provides suitable potential for further iterative reductions, avoiding premature convergence to local optima. The initial search space for micro-kernel scheduling includes the parameters *nps*, {x, y, z}*_tiling_size*, and {x, y, z}*_unroll_size*. The sampling space for the initial *tiling_size* is set to {1, 2, 4, 8, ..., 1024}, *unroll_size* ranges from 0 to 3, and *nps* ranges from 1 to 9. When setting initial values, *x_tiling_size* is traversed across all items from its sampling space, while the other

parameters are randomly sampled within their respective sampling spaces. Each *x_tiling_size* generates eight sets of initial micro-kernel schedules. Since *x_tiling_size* is the innermost blocking parameter of the array and has the greatest impact on performance with numerous local optima, sufficient and uniform samples are necessary to avoid getting trapped in local optima.

4) Candidate Stencil Codes: Using the micro-kernel template, the micro-kernel schedule is transformed into executable code, generating compilable candidate stencil code. These candidate stencil codes are then evaluated through real execution. Simultaneous testing of candidate codes is facilitated by locality block specification and ccNUMA node pooling, which optimizes stencil computations. The workload of each individual ccNUMA node is isolated by partitioning the stencil's input tensor into subtensors. The process of testing a candidate stencil code typically requires approximately 1 second for compilation and 5–15 seconds for execution. Each tuning round takes roughly 20–40 seconds on the Phytium platform and 100–180 seconds on the Kunpeng platform, depending on factors such as stencil shape, hardware specifications, and the number of ccNUMA nodes.

5) Performance Database: Across the entire system's multiple ccNUMA nodes, the Micro-kernel schedule optimizer tests multiple candidate stencil codes simultaneously, logging the performance outcomes of each micro-kernel schedule round into a performance database.

6) GA-based Schedule Generator: This generator iteratively evolves the parameter population by selecting high-performing parameters, performing crossover to amalgamate attributes from parental parameters, and introducing random mutations to foster diversity. These operations enable the GA to efficiently explore the optimization space, identifying promising optimization parameter sets to combine new potential Micro-kernel Schedules.

7) Potential Micro-kernel Schedules: This serves as the next batch of test samples, initiating a new testing cycle.

8) Optimized Micro-kernel Schedule: When the Performance Database fails to yield better-performing micro-kernel schedules over several testing rounds, the Micro-kernel schedule optimizer concludes, outputting the currently best-performing micro-kernel schedule.

4.2 Block Schedule Optimizer

The AOStencil compiler advances a locality-aware block schedule optimizer based on the locality-aware multi-level thread parallelism. During compilation, it performs locality-aware optimization and HALO exchange specification. It statically binds stencil tasks and the locality memory storage of input and output tensors, along with HALO exchange zones. Ultimately, this approach ensures that at runtime, stencil operates with multilevel parallelism across the cache and memory hierarchy.

Table 1: Mapping from stencil loop to locality-aware block scheduling

Thread level	2d stencil	3d stencil
Thread	x loop	x loop
NUMA Block	y loop	y loop
Socket Block	y loop	z loop
Socket Block cluster	y loop	z loop

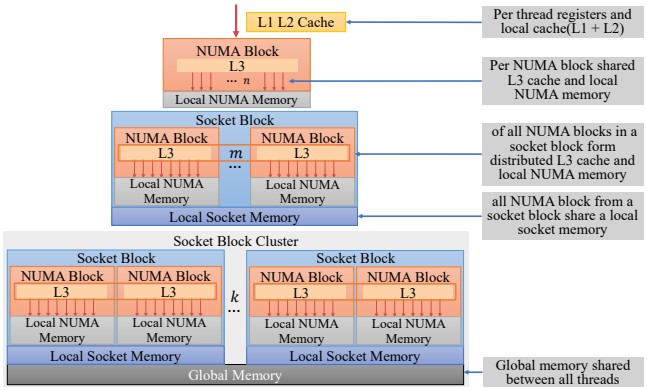


Figure 6: The illustration of multi-level thread parallelism in locality-aware block scheduling.

The *multi-level thread parallelism in locality-aware block scheduling* is illustrated in Figure 6, where n represents the number of processors within a ccNUMA node, m denotes the number of ccNUMA nodes within a socket, and k indicates the number of sockets within a server. This parallelism is structured across four hierarchical levels: the first level consists of a single thread, incorporating a processing core along with local registers, and L1 and L2 caches. The second level groups n threads into a NUMA block, which, at the hardware level, is integrated through the ccNUMA architecture to include an L3 cache and local NUMA memory. The third level assembles m thread blocks to form a socket block within a server's socket. The fourth and final level clusters k thread block clusters to form a socket block cluster, encompassing all sockets on the server. As demonstrated in Figure 2, the locality-aware block scheduling proposed in this paper aligns with the memory hierarchy structure observed in multi-core architectures. It is noted that with each increasing level, there is a corresponding increase in memory-IO latency, reflecting the complex interdependencies and latency penalties associated with data across the hierarchical levels of modern server architectures.

The locality-aware block scheduling is designed to achieve fine-grained parallelism in stencil in multi-core architectures.

It uses threads as the fundamental units, partitioning the storage input tensor of stencil into local NUMA memory for each thread. The mapping from the 2D/3D stencil loop to the *locality-aware block scheduling* is illustrated in Table 1. When mapping the stencil block to hardware units, each thread maximizes the use of the nearest memory and arranges adjacent blocks that require communication within a local hardware unit, such as within a ccNUMA node or socket. For instance, in the case of three-dimensional stencil parallelism:

- At the socket block cluster level, the z for loop of the stencil IR is divided among k socket blocks for parallel processing.
- At the socket block level, the z for loop is further divided among m NUMA blocks.
- At the NUMA block level, the y for loop of the stencil IR is divided among n threads for parallel computation.
- At the thread level, the x for loop of the stencil IR is executing within a thread.

The AOStencil compiler performs static partitioning of the stencil block during compile time. The subsequent generation of specific C code involves lowering each thread's *thread_function* to leverage the lightweight pthread library for thread creation and management. To improve thread execution performance and minimize conditional branching, the HALO exchange specification preprocesses and indexes the computation regions for each thread, embedding static positions within the thread-partitioned block. This specification encompasses three key components: 1) positioning the stencil block, 2) managing the movement of the input tensor's global storage position—prior to the stencil execution, data is copied into the ccNUMA local storage of the thread block, and after computation, it is copied back to global storage, and 3) handling the HALO region following each stencil timestep.

The locality-aware block scheduling effectively exploits the memory hierarchy inherent in modern ccNUMA and socket architectures. This optimization strategy primarily enhances the speed of memory access by aligning memory demands with the hierarchical memory layout. Secondly, as a thread block within a shared L3 cache accesses adjacent tensor blocks of the stencil computation by the L3 cache, this methodology improves the utilization of the cache. Finally, this approach mitigates cross-socket NUMA access on the server, which in turn alleviates pressure on the PCIe bus and the cache coherence system. This reduction in data exchange across L3 caches, NUMA nodes, and sockets leads to decreased latency and improved overall system performance.

Table 2: The hardware and software configuration

Platform	Kunpeng	Phytium
Processor	Kunpeng-920	Phytium-S2500
Sockets	2	2
NUMA nodes	4	16
Cores	96	128
Libnuma	1.0.0	1.0.0
GCC	8.4.0	9.4.0
OpenMP	201511	201511
Pthread	2.31	2.28
MPI	mpich-4.2.0	mpich-4.2.0
Clang	16.0.6	16.0.6

Table 3: Stencil pattern benchmarks

Stencil patterns	*2d9pt_star*	*2d9pt_box*	*3d7pt_star*	*3d27pt_box*
Read(Byte)	72	72	56	216
Write(Byte)	8	8	8	8
Min-memory IO(Byte)	16	16	16	16
Float Ops	17	17	13	53
Operation Intensity (Flops/Byte)	1.06	1.06	0.81	3.31
TimeDep.	2	2	2	2

5 Evaluation

5.1 Experimental Setup

This study utilizes two experimental platforms: the Phytium cluster with Phytium-S2500 processors (Phytium platform) and the Taishan v110 server with Huawei Kunpeng-920 processors (Kunpeng platform), which are the primary ARM platforms used in the latest supercomputing systems. Table 2 details the software and hardware configurations, while Figures 3a and 3b depict the system architectures. The GCC compiler is used by AOStencil. We evaluate stencil benchmarks with varying shapes, input sizes, and computational intensities, as shown in Table 3, to represent diverse scientific applications. 'Read' and 'Write' refer to memory operations for a single stencil element (for example, using the double type in Table 3). Minimum memory IO reflects optimal multi-level cache use, and Operation Intensity (Table 3, Figure 9) is the ratio of floating-point operations to minimum memory IO. In our experiments, the stencil time iterations are set to 100 steps by default. The experimental evaluation of the AOStencil compiler encompasses five aspects:

- **Performance Comparison:** We compare the performance of the AOStencil compiler with the manually optimized stencil codes and state-of-the-art stencil compilers, such as MSC [18] and Halide [11, 22].
- **Ablation Study:** We conduct an ablation study focusing on the methods of the AOStencil.
- **Roofline Analysis:** We perform a roofline analysis to evaluate the computational efficiency of the AOStencil compiler.
- **Scalability Analysis:** We analyze the scalability of the AOStencil compiler across different problem sizes and hardware configurations.
- **Auto-tuning Cost Analysis:** We assess the cost associated with the auto-tuning analysis performed by the AOStencil compiler.

5.2 Overall Performance

As illustrated in Figure 7, the experiments encompass a comprehensive comparison using two types of floating-point precision. The x-axis of Figure 7 represents the performance platforms and input tensor sizes. On each subplot, performance data for the Kunpeng platform appear on the left, while data for the Phytium platform are shown on the right. The study employs two and three-dimensional stencils with input tensor sizes of (8192^2), (16384^2), (24576^2) for 2D, and (256^2), (512^2), (768^2) for 3D. The y-axis indicates the performance of stencil codes generated by the manual optimization and the compilers.

Specifically, the *Manual Optimization* indicates manually optimized stencil codes with tiling, unrolling, and vectorization techniques from domain expert. The MSC compiler, lacking automatic parameter tuning capabilities and subject to specific grid size constraints for each MPI process, utilizes manually tuned parameters based on expert experience, as indicated in the performance data labeled *MSC*. MPI+OpenMP parameters are optimally set according to the hardware system specifications of each platform to ensure that each MPI process executes within a ccNUMA node. In contrast, the Halide compiler, which features automatic tuning capabilities [1, 21], provides performance data under the label *Halide-auto* based on the recommended automatic tuning framework. Despite Halide's automatic tuning feature, its performance does not always achieve optimality across all benchmarks. Therefore, for a comprehensive experimental approach, manually tuned performance data from expert experience, labeled *Halide-Manual*, are also included. The *AOStencil* data derive from performance measurements of stencil code automatically optimized by the AOStencil compiler. From Figure 7, it is evident that the AOStencil compiler demonstrates a better performance over both the MSC and Halide compilers, which is manifest in three main areas:

1) Cross-Platform Capability: The AOStencil compiler demonstrates strong cross-platform performance by optimizing for both microarchitecture and multi-core systems, surpassing the Halide and MSC compilers. Unlike Halide, AOStencil focuses on optimizing stencil computations at both the microarchitecture and multi-processor interconnect layers (e.g., ccNUMA, sockets). Specifically, AOStencil implements independent instruction stream scheduling optimizations to overlap memory latency and computations with out-of-order execution of multiple pipelines in modern ARM microarchitectures, which Halide ignores. While Halide performs well on the Kunpeng platform, it is less suited to the Phytium. The MSC compiler generally performs better on Phytium than Kunpeng.

2) Performance on Multiple Stencil Patterns: The AOStencil compiler consistently outperforms Halide and MSC in benchmarks across four stencil patterns, thanks to its effective micro-kernel schedule optimization. Additionally, AOStencil addresses high memory access latencies on interconnect architectures through locality-aware block scheduling, as demonstrated in the roofline experiments in Figure 9. MSC performs better in 2D but struggles with 3D stencil patterns.

3) Performance Stability Across Input Tensor Sizes: The AOStencil compiler maintains stable performance across various tensor sizes, unlike MSC and Halide, which degrade on smaller inputs. This stability reflects the success of AOStencil's auto-tuning and locality-aware block schedule optimization, effectively mitigating the potential high latency of memory accesses that Halide overlooks.

In summary, in this comparative experiment, the AOStencil compiler provides competitive performance relative to the other two stencil DSL compilers. On the Kunpeng platform, the AOStencil compiler achieves an average performance improvement of 0.86 times over the Halide compiler and 6.84 times over the MSC compiler. On the Phytium platform, the AOStencil compiler achieves an average improvement of 6.73 times over the Halide compiler and 4.39 times over the MSC compiler.

5.3 Ablation Study

To further demonstrate the efficacy of the primary method discussed in this article, we conducted an ablation study. The experimental design included four sets of optimization strategies: 1) `baseline`: Utilized OpenMP parallelism and was compiled with GCC using the "-O3" optimization flag. 2) `+block schedule`: To enhance the stability of the ablation experiment and mitigate errors caused by the randomness of memory allocation, a stencil with a locality-aware block scheduling approach was implemented initially. 3) `+SFTBR`:

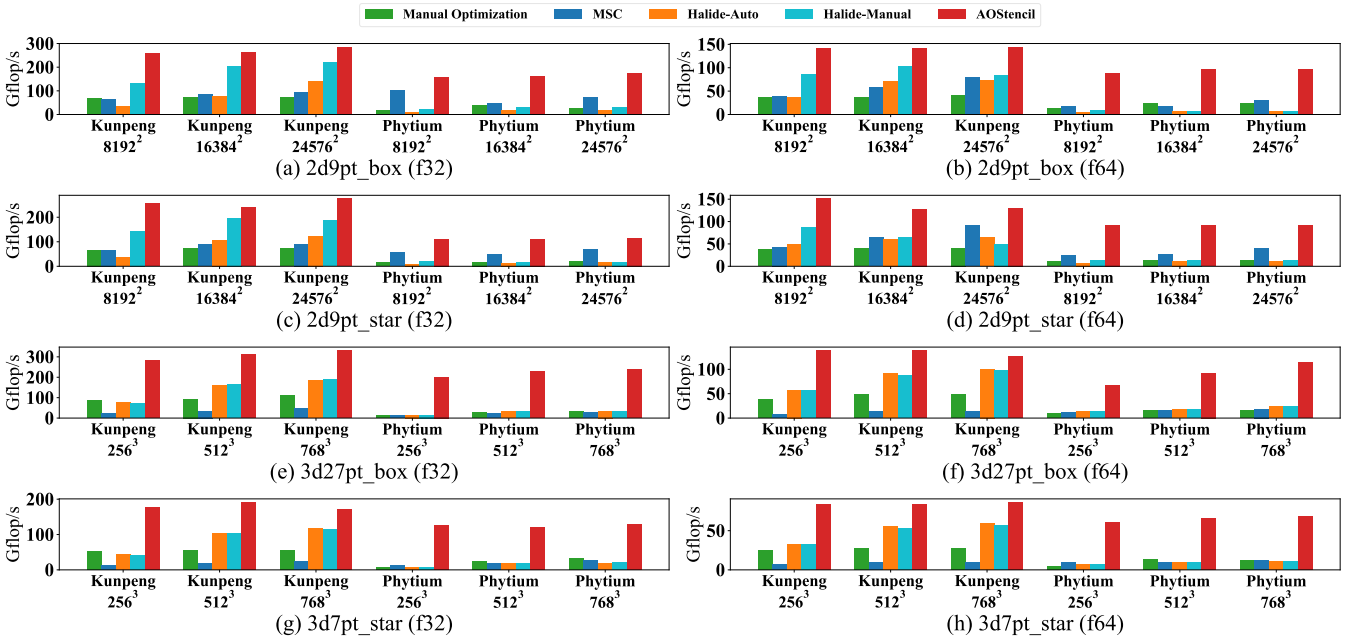


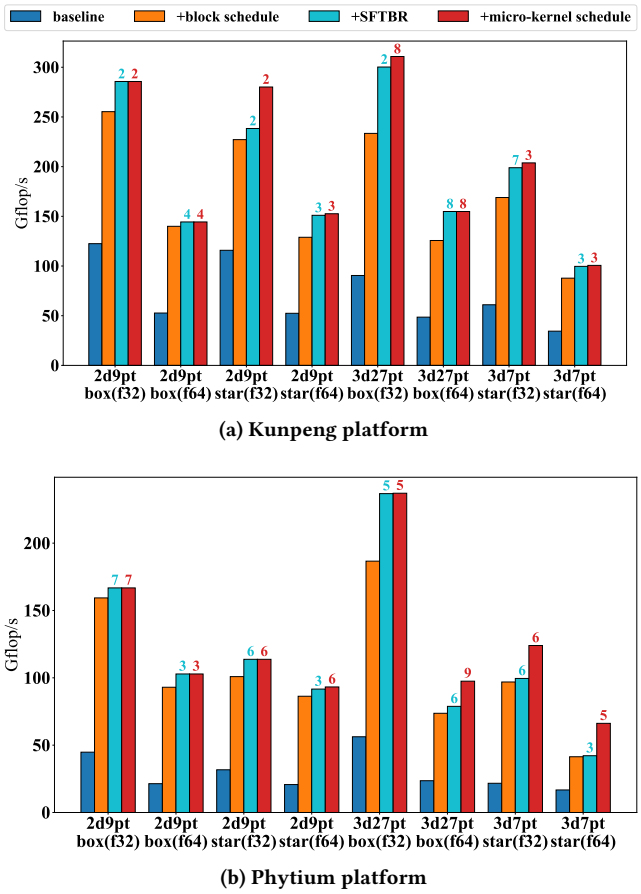
Figure 7: Overall performance of optimized stencil codes on Kunpeng and Phytium processors.

Optimized stencil computation using a micro-kernel schedule that incorporates only the SFTBR method, in conjunction with the block schedule. 4) `+micro-kernel schedule`: Combined the micro-kernel schedule with all optimization techniques, including unrolling, tiling, and SFTBR, alongside the block schedule, as implemented by the AOstencil compiler. The ablation study was performed using input tensor sizes of 16384^2 for 2D stencils and 512^3 for 3D stencils, employing both single-precision (f32) and double-precision (f64) floating-point formats across two different platforms.

The results of this ablation study are depicted in Figure 8, where each subplot corresponds to the platforms. In these figures, the x-axis represents a test of one stencil pattern at a specific precision, while the y-axis denotes flops performance. The selected parameter *nps* of the SFTBR method is indicated at the top of each data bar. From the results, it is evident that both the micro-kernel schedule and the locality-aware block schedule consistently enhanced performance across the board. Specifically, by the SFTBR method, there was an average performance increase of 16% on the Kunpeng platform and 10% on the Phytium platform. In most cases, the selected parameter *nps* of the SFTBR method was bigger than one, suggesting that increasing the number of instruction streams to a suitable value can effectively boost the computational performance of stencils by increasing ILP. The charts also reveal that the performance gains from employing both the micro-kernel schedule and the locality-aware block schedule were approximately 1.92x and 3.3x on the Kunpeng and Phytium platforms, respectively.

5.4 Roofline Analysis

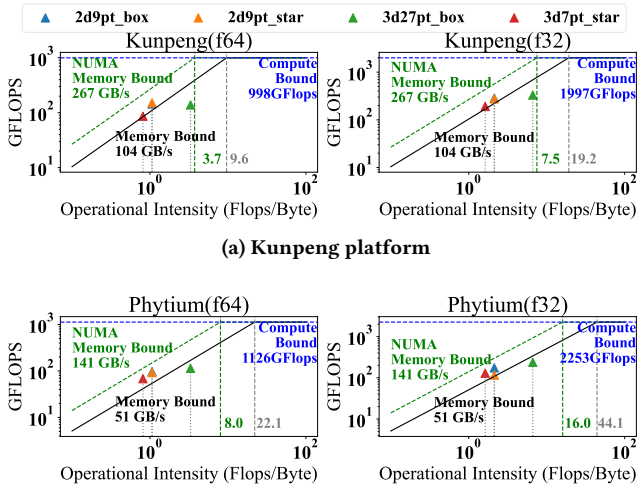
To elucidate the performance of the stencil code produced by the AOStencil compiler, we conducted a roofline model analysis on both the Phytium and Kunpeng platforms. This analysis performance across both f32 and f64 precision. The roofline model offers insights into the computational efficiency and memory bandwidth constraints of the AOStencil-generated code, facilitating the identification of performance bottlenecks and potential optimization avenues. Figures 9a and 9b depict the roofline models for the Kunpeng platform in FP32 and FP64 precision, respectively. Each roofline graph includes two memory bounds: one for the global memory of the hardware system and another for local NUMA memory utilization. Global memory refers to the memory directly allocated to each computing core by the hardware system, irrespective of the NUMA node, while local NUMA memory pertains to memory allocated by computing cores within their respective local NUMA nodes, as detailed in Figure 6. Analyses from Figures 9a and 9b reveal that the memory bound from global memory is respectively 45% and 36% of the memory bound from local NUMA memory on the Kunpeng and Phytium platforms. These findings underscore the significant performance improvements that can be achieved by effectively utilizing multi-level memory in a multi-core architecture for memory-bound programs.



(a) Kunpeng platform

(b) Phytium platform

Figure 8: Performance ablation study of the proposed optimization techniques.



(a) Kunpeng platform

(b) Phytium platform

Figure 9: Roofline analysis on (a) Kunpeng and (b) Phytium platforms.

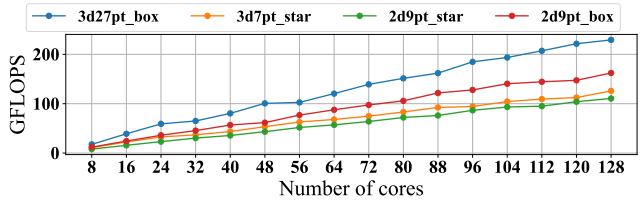


Figure 10: Scalability analysis of different stencils.

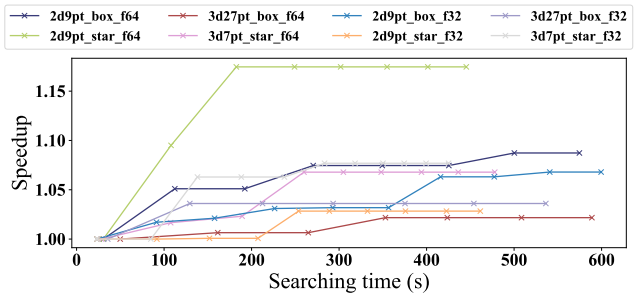
As depicted in Figures 9a and 9b, on both the Kunpeng and Phytium platforms, stencil patterns such as *2d9pt_star*, *2d9pt_box*, and *3d7pt_star* have exceeded the upper bounds of the global memory. This demonstrates the effective utilization of multi-core architectures by the AOStencil compiler's locality-aware block schedule optimization, significantly enhancing the performance of stencil on these architectures. For the *3d27pt_box*, the AOStencil compiler also showcased commendable performance on the higher peak floating-point capable Phytium platform, while further improvements are awaited on the Kunpeng platform.

5.5 Scalability

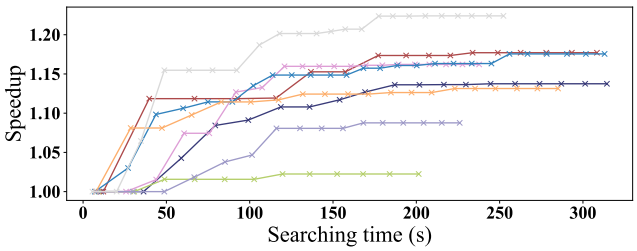
This study designed scalability experiments to verify the multicore scalability of the AOStencil compiler, conducted on the Phytium platform, with input tensor sizes of 16384^2 for 2D stencil and 512^3 for 3D stencil with f32 precision, encompassing four groups of stencil patterns. As illustrated in Figure 10, the x-axis represents the number of cores utilized in the experiment, incrementally increased in steps corresponding to the processor core count per ccNUMA node. All stencil pattern benchmarks demonstrated linear speedup on the Phytium platform as the number of cores increased. Notably, when scaled to the maximum number of cores, the AOStencil achieved an average speedup of 13 times compared to its performance at the minimum number of cores. These scalability experiments substantiate the AOStencil compiler's capability to generate stencil code that exhibits scalability.

5.6 Auto-tuning Cost Analysis

Most of the execution time of the AOStencil compiler is spent on micro-kernel fine-tuning. Figure 11 illustrates the auto-tuning cost for each micro-kernel on the Kunpeng and Phytium platforms with input tensor sizes of 16384^2 for 2D stencil and 512^3 for 3D stencil. The names of each stencil case are shown on the right side of the figure, encompassing



(a) Kunpeng platform



(b) Phytium platform

Figure 11: The performance (speedup to the first iteration) of each kernel over the auto-tuning.

both f32 and f64 precisions. As shown in Figure 11, the auto-tuning costs reveal that the evolutionary search algorithm employed during the auto-tuning phase utilizes an efficient initial search space configuration. As a result, the speedup in the first iteration is moderate, averaging approximately 7% on the Kunpeng platform and 13% on the Phytium platform, respectively. The auto-tuning method in AOStencil can achieve 95% of its best performance on average after only two tuning iterations. The AOStencil utilizes independent ccNUMA nodes for parallel execution, which significantly reduces the time required for fine-tuning. On the Kunpeng platform, tuning is completed in less than 500 seconds for all tests. On the Phytium platform, all stencil operators were tuned within 500 seconds, with most completing in under 200 seconds. These results demonstrate the efficiency of the AOStencil compiler's auto-tuning process, which significantly reduces the time required for fine-tuning.

6 Related Work

In previous studies focused on general stencil optimization techniques, the approaches primarily fall into several categories: utilizing instruction-level parallelism through SIMD vectorization [5, 13, 35, 40], enhancing cache efficiency with various tiling strategies [6, 30], parallelism [7], and reducing data exchange overhead in parallel implementations [32]. Given that stencils are inherently memory-bound with favorable memory locality, tiling techniques have been extensively explored in this context, including various tiling shapes such as conventional rectangular tiling [17], tessellating tiling [38], and diamond tiling [7]. Some studies [26, 39] have implemented two-level parallelism using MPI+OpenMP to accommodate multicore NUMA architectures. The work of Rawat [24] presents a statement reordering framework that models stencil computations as DAG of trees to optimize register usage on GPUs by minimizing register pressure with the limited number of registers.

To address the challenges posed by diverse stencil patterns and complex hardware environments, several compilers and code generators that take Domain Specific Languages (DSLs) describing stencil patterns as input have been proposed. The YASK compiler [36] employs vector folding to vectorize a loop layer, achieving high-throughput stencil. The OpenEarth compiler [16] utilizes Multi-Level Intermediate Representation (MLIR) for efficient code generation of stencil DSLs on GPUs. The Artemis compiler [25] focuses on graph optimization problems for complex stencils. The MSC compiler [18] is designed for large-scale stencils on many-core systems such as Sunway and Matrix, integrating loop tiling, loop reordering, and cache-IO optimization methods. The MSC compiler demonstrates superior performance compared to the Patus [8] and Physis [20] compilers on CPUs. Additionally, the Halide compiler [22] illustrates the advantage of decoupling computation from scheduling, enabling automatic optimizations that often outperform manually-tuned implementations.

However, the fragmented ARM ecosystem poses performance challenges for cross-microarchitecture adaptation. Although both Kunpeng and Phytium architectures are based on ARMv8-A, they differ in key specifications such as instruction latency, the size of the reorder buffer, the number of pipelines in out-of-order execution, and the number of dispatches supported by multi-issue mechanisms [3, 34]. These factors significantly impact the optimal instruction scheduling, thereby complicating the development of high-performance computing libraries for ARM platforms. Previous stencil compilers, including YASK, MSC, and Halide, have predominantly focused on optimizations on individual microarchitectures, often neglecting the influence of diverse microarchitectural implementations on the effectiveness of optimization methodologies. In contrast, the SFTBR strategy introduces a multi-streams instruction optimization model that uses the *nps* parameter to optimize various stencil operators across different microarchitecture implementations. Furthermore, no multi-level threading models for locality-aware have been proposed to adapt to the emerging large-scale many-core under memory hierarchy architectures with

multi-ccNUMA and multi-socket. The AOStencil compiler introduced in this paper fills these research gaps and leverages these optimizations, which achieves substantial performance improvements on two experimental platforms compared to previous compilers.

7 Conclusion

In conclusion, this study has demonstrated the critical importance of optimizing both instruction-level parallelism and multi-level thread parallelism strategies in stencil computations for modern many-core architectures. By leveraging the Serial-FMA to Tree-Based Reduction (SFTBR) method and implementing a *locality-aware block scheduling*, this research has effectively addressed the challenges posed by complex cache and memory hierarchies and memory-bound workloads. The AOStencil compiler, which integrates both the optimizations proposed in this paper and conventional optimization techniques, has demonstrated significant performance improvements. These include speedups of up to 0.86x and 4.39x on platforms such as Kunpeng and Phytium with other stencil DSL compilers. These findings underscore the necessity of co-optimizing computational and memory access strategies to fully exploit the potential of many-core systems. The demonstrated improvements affirm that multi-level thread parallelism, alongside instruction-level optimizations, is a key part of advancing stencil computation performance in future HPC systems.

Acknowledgments

This work is supported by National Key Research and Development Program of China (Grant No. 2023YFB3001501), National Natural Science Foundation of China (No. 62322201 and U23B2020), and the Fundamental Research Funds for the Central Universities (YWF-23-L-1121, JKF-20240198 and JK2024-58). Hailong Yang is the corresponding author.

References

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[2] Yulong Ao, Chao Yang, Xinliang Wang, Wei Xue, and Wenjing Ma. 2017. 26 PFLOPS Stencil Computations for Atmospheric Modeling on Sunway TaihuLight. *IEEE* (2017).

[3] ARM. 2025. ARM cortex-A78 core software optimization guide. https://documentation-service.arm.com/static/60a5413bd63d3c31550c391e/

[4] ARM. 2025. Dual issue for NEON instructions *version*1.0. https://developer.arm.com/documentation/den0018/a/Optimizing-NEON-Code/Scheduling/Dual-issue-for-NEON-instructions

[5] Adrià Armejach, Helena Caminal, Juan M Cebrian, Rubén Langarita, Rekai González-Alberquilla, Chris Adeniyi-Jones, Mateo Valero, Marc Casas, and Miquel Moretó. 2020. Using Arm's scalable vector extension on stencil codes. *The Journal of Supercomputing* 76, 3 (2020), 2039–2062.

[6] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling stencil computations to maximize parallelism. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[7] Ian J Bertolacci, Catherine Olschanowsky, Ben Harshbarger, Bradford L Chamberlain, David G Wonnacott, and Michelle Mills Strout. 2015. Parameterized Diamond Tiling for Stencil Computations with Chapel parallel iterators. In *International Acm Conference on International Conference on Supercomputing*.

[8] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 2011. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 676–687.

[9] Phillip Colella. 2004. Defining software requirements for scientific computing. *DARPA HPCS*. (2004). https://www.krellinst.org/doecsgf/conf/2013/pres/pcolella.pdf

[10] Kaushik Datta, Samuel Williams, Vasily Volkov, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. 2009. Auto-tuning the 27-point stencil for multicore. In *In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, Vol. 70.

[11] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. 2016. Distributed halide. *ACM SIGPLAN Notices* 51, 8 (2016), 1–12.

[12] Jian-Bin Fang, Xiang-Ke Liao, Chun Huang, and De-Zun Dong. 2021. Performance evaluation of memory-centric armv8 many-core architectures: A case study with phytium 2000+. *Journal of Computer Science and Technology* 36 (2021), 33–43.

[13] Matteo Frigo and Volker Strumpen. 2005. Cache Oblivious Stencil Computations. In *19th ACM International Conference on Supercomputing*.

[14] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. 2022. *Processor microarchitecture: An implementation perspective*. Springer Nature.

[15] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, and David Padua. 2010. Writing productive stencil codes with overlapped tiling. *Concurrency & Computation Practice & Experience* 21, 1 (2010), 25–39.

[16] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2021. Domain-specific multi-level IR rewriting for GPU: The Open Earth compiler for GPU-accelerated climate simulation. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 4 (2021), 1–23.

[17] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *International Conference on Supercomputing*.

[18] Mingzhen Li, Yi Liu, Hailong Yang, Yongmin Hu, Qingxiao Sun, Bangduo Chen, Xin You, Xiaoyan Liu, Zhongzhi Luan, and D. Qian. 2021. Automatic Code Generation and Optimization of Large-scale Stencil Computation on Many-core Processors. *Proceedings of the 50th International Conference on Parallel Processing* (2021).

[19] Xiaoyan Liu, Xinyu Yang, Kejie Ma, Shanghao Liu, Kaige Zhang, Hailong Yang, Yi Liu, Zhongzhi Luan, and Depei Qian. 2024. Moirae: Generating High-Performance Composite Stencil Programs with Global Optimizations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 20, 15 pages.

[20] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. 2011. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[21] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–11.

[22] 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.

[23] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, and P Sadayappan. 2016. Effective resource management for enhancing performance of 2D and 3D stencils on GPUs. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 92–102.

[24] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P Sadayappan. 2018. Register optimizations for stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 168–182.

[25] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-Noël Pouchet, and P Sadayappan. 2019. On optimizing complex stencils on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 641–652.

[26] Lakshminarayanan Renganarayana, Manjukumar Harthikote-Matha, Rinku Dewri, and Sanjay Rajopadhye. 2007. Towards optimal multi-level tiling for stencil computations. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–10.

[27] Naser Sedaghati, Renji Thomas, Louis-Noël Pouchet, Radu Teodorescu, and P Sadayappan. 2011. StVEC: A vector instruction extension for high performance stencil computation. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 276–287.

[28] Mageda Sharafeddine, Komal Jothi, and Haitham Akkary. 2012. Disjoint out-of-order execution processor. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 3 (2012), 1–32.

[29] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. 2015. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 207–216.

[30] Larisa Stoltzfus, Bastian Hagedorn, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2019. Tiling Optimizations for Stencil Computations Using Rewrite Rules in Lift. *ACM Transactions on Architecture and Code Optimization (TACO)* (2019).

[31] Qingxiao Sun, Yi Liu, Hailong Yang, Zhonghui Jiang, Zhongzhi Luan, and Depei Qian. 2022. Stencilmart: Predicting optimization selection for stencil computations across gpus. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 875–885.

[32] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. 2009. Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization. *IEEE* (2009).

[33] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[34] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. 2021. Kunpeng 920: The first 7-nm chiplet-based 64-core arm soc for cloud services. *IEEE Micro* 41, 5 (2021), 67–75.

[35] Charles Yount. 2015. Vector Folding: improving stencil performance via multi-dimensional SIMD-vector representation. In *2015 IEEE 17th international conference on high performance computing and communications, 2015 IEEE 7th international symposium on cyberspace safety and security, and 2015 IEEE 12th international conference on embedded software and systems*. IEEE, 865–870.

[36] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. Yask—yet another stencil kernel: A framework for hpc stencil code-generation and tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. IEEE, 30–39.

[37] Liang Yuan, Hang Cao, Yunquan Zhang, Kun Li, Pengqi Lu, and Yue Yue. 2021. Temporal vectorization for stencils. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[38] Liang Yuan, Yunquan Zhang, Peng Guo, and Shan Huang. 2017. Tessellating stencils. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[39] Kaifang Zhang, Huayou Su, and Yong Dou. 2021. Multilevel parallelism optimization of stencil computations on SIMDlized NUMA architectures. *The Journal of Supercomputing* 77, 11 (2021), 13584–13600.

[40] Kaifang Zhang, Huayou Su, Peng Zhang, and Yong Dou. 2020. Data layout transformation for stencil computations using arm neon extension. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 180–188.

[41] Kaifang Zhang, Huayou Su, Peng Zhang, and Yong Dou. 2020. Optimization and Performance Modeling of Stencil Computations on ARM Architectures. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 113–121.

[42] Peng Zhang and Lang Lu. 2020. A4-Way CC-NUMA System Based on S2500. In *2020 International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE)*. IEEE, 338–342.