SCIENCE CHINA
Information Sciences

January 2024, Vol. 67, Iss. 1, 112101:1–112101:19
https://doi.org/10.1007/s11432-022-3727-6

• RESEARCH PAPER •
From CAS & CAE Members

# Building a domain-specific compiler for emerging processors with a reusable approach

Mingzhen LI[1,2], Yi LIU[2], Bangduo CHEN[2], Hailong YANG[1,2*], Zhongzhi LUAN[2] & Depei QIAN[2*]

[1]*State Key Laboratory of Software Development Environment, Beijing* 100191, *China;*
[2]*School of Computer Science and Engineering, Beihang University, Beijing* 100191, *China*

Received 1 April 2022/Revised 28 November 2022/Accepted 17 February 2023/Published online 27 December 2023

**Abstract** High-performance computing and deep learning domains have been motivating the design of domain-specific processors. Although these processors can provide promising computation capability, they are notorious for exotic programming paradigms. To improve programming productivity and fully exploit the performance potential of these processors, domain-specific compilers (DSCs) have been proposed. However, building DSCs for emerging processors requires tremendous engineering efforts because the commonly used compilation stack is difficult to be reused. Owing to the advent of multilevel intermediate representation (MLIR), DSC developers can leverage reusable infrastructure to extend their customized functionalities without rebuilding the entire compilation stack. In this paper, we further demonstrate the effectiveness of MLIR by extending its reusable infrastructure to embrace a heterogeneous many-core processor (Sunway processor). In particular, we design a new Sunway dialect and corresponding backend for the Sunway processor, fully exploiting its architectural advantage and hiding its programming complexity. To show the ease of building a DSC, we leverage the Sunway dialect and existing MLIR dialects to build a stencil compiler for the Sunway processor. The experimental results show that our stencil compiler, built with a reusable approach, can even perform better than state-of-the-art stencil compilers.

**Keywords** domain-specific compiler, emerging processor, reusable dialect, performance optimization, MLIR

## 1 Introduction

The growing computation demands of high-performance computing (HPC) and deep learning (DL) have been flourishing in the design of processors, especially with the slowdown of Moore's law. Emerging processors include general-purpose processors, such as RISC-V and Sunway [1], and domain-specific accelerators, such as TPU and Graphcore [2]. These processors are usually designed with unique architectural features different from the mainstream processors (e.g., x86 CPU and GPU). Such an architectural difference eventually leads to exotic programming paradigms and barren software ecosystems. For real-world applications to fully utilize these emerging processors, it is mandatory for domain experts and optimization experts to pay great efforts to adapt programming paradigms and manually optimize performance, which tends to be time-consuming and error-prone.

The common approach to ease programming efforts and achieve high performance is the use of domain-specific compilers (DSCs, also known as domain-specific languages (DSLs) in existing literature). For example, DL compilers (e.g., XLA [3] and TVM [4]) perform DL-specific compilation to accelerate the DL training/inference, whereas compilers for scientific computing (e.g., Pluto [5] and Stella [6]) perform nested-loop optimizations to accelerate widely used computations, such as stencils. However, these compilers are specialized solutions for mainstream processors with poor portability to emerging processors because they usually leverage the compiler tool-chains (e.g., LLVM [7]) that may not be available yet on emerging processors. Therefore, emerging processors are commonly forced to rebuild their own DSCs

* Corresponding author (email: hailong.yang@buaa.edu.cn, depeiq@buaa.edu.cn)

 

from scratch, and it is infeasible to reuse the existing compilation stack, which hinders the prosperity of their software ecosystem.

The multilevel intermediate representation (MLIR) [8] is a new compiler infrastructure used to build reusable and extensible DSCs, with its multilevel abstractions that connect the application domains and hardware targets. It introduces dialects to represent multilevel abstractions. Currently, many DSCs [9–12] are actively adopting MLIR in their implementations. To incorporate an emerging processor into MLIR, one should design a hardware-specific dialect, the hardware backend, and corresponding lowering rules. Afterward, the entire MLIR ecosystem, including dialects (defined by MLIR officially or by other DSCs), progressive lowering rules, and optimization passes, can be reused, which eases the building of DSCs targeting emerging processors.

Sunway processors are the principal building block of the Sunway TaihuLight supercomputer that was the first to achieve the peak performance of 100 PFlops in double precision. The Sunway processor is a heterogeneous many-core processor with four core groups (CGs), which offers 3.06 TFlops peak performance in double precision. Each CG contains an MPE for computation management and 64 simplified CPEs for computation acceleration. It has unique architecture designs to control the computation hierarchy and memory hierarchy through a unique programming paradigm (e.g., Athread). However, even experienced programmers shall spend a considerable amount of time adapting programs to the architecture features for high performance. To our knowledge, only one DSC (MSC [13]) is available on Sunway that targets the optimization of stencil computation. In this study, we implement a new stencil compiler for the Sunway processor. Different from existing studies [13], we built the stencil compiler with a reusable approach leveraging the MLIR infrastructure. We hope that our work can enrich the MLIR ecosystem by providing a reference dialect and backend implementation for emerging processors, such as Sunway, and demonstrate a way to ease the development of DSCs for emerging processors.

Therefore, we propose swLego, which consists of a reusable Sunway dialect and code generation backend built with the MLIR infrastructure. It is designed to incorporate the Sunway processor into the MLIR ecosystem and alleviate the burden of building DSCs on Sunway. From the point of view of MLIR, the Sunway dialect sits at the same level as the GPU dialect, whereas the Sunway backend sits at the same level as the LLVM backend (on the MPE side) and NVVM/ROCDL/SPIR-V backend (on the CPE side). swLego also provides practical ways to exploit the architectural advantages through compilation optimizations and utilization of highly optimized kernel libraries. We demonstrate the effectiveness of swLego for building a DSC swLego-stencil, which optimizes the stencil computation on the structural grid for Sunway processors. The experimental results show that, with such a reusable approach, the engineering efforts for building a DSC can be significantly reduced while achieving a comparable performance as existing DSCs.

Specifically, this paper makes the following contributions.

- We propose swLego, a Sunway dialect and Sunway backend, extending MLIR with the architectural features and customized programming paradigm of Sunway processors. swLego broadens the MLIR ecosystem with support to emerging processors, such as Sunway, and provides a reusable compilation infrastructure for developing DSCs on Sunway.
- We implement a DSC swLego-stencil using swLego, which optimizes the stencil computation on a structural grid to demonstrate the reusability of swLego. swLego-stencil supports complex stencils with arbitrary shapes and inter-kernel dependency in large-scale execution and applies various performance optimizations, leveraging the compilation stack of swLego.
- We evaluate swLego-stencil with representative benchmarks and provide a sensitivity analysis and roofline model analysis to study the performance. In addition, we evaluate the scalability in a large-scale execution. The experimental results demonstrate that the stencil compiler built with swLego achieves better performance than existing stencil compilers.

The rest of this paper is organized as follows: Section 2 describes the background of the Sunway processor, MLIR, and stencil DSCs. Section 3 presents the design overview of building a DSC based on the MLIR infrastructure. Section 4 presents a reusable approach for implementing the Sunway dialect and Sunway backend. Section 5 presents a case study for building a stencil compiler based on swLego-stencil. Section 6 presents the evaluation results and compares swLego-stencil with state-of-the-art stencil compilers. Section 7 concludes the paper.

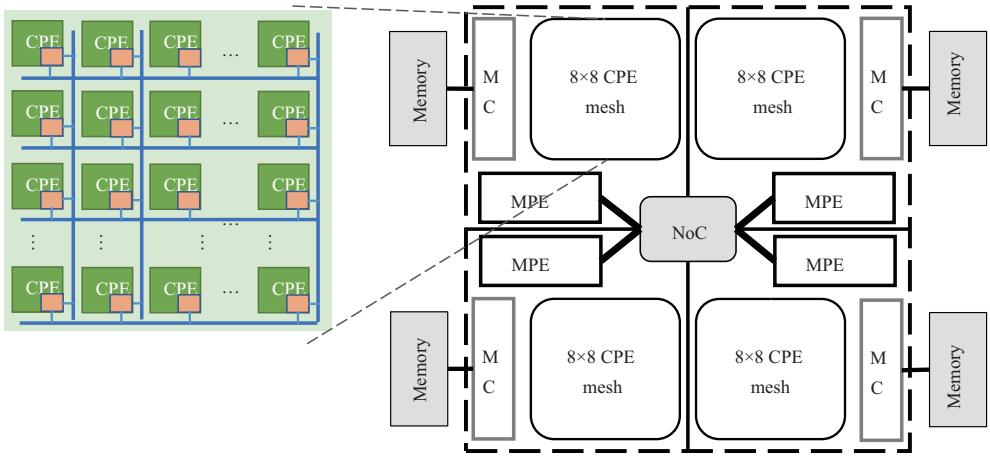


**Figure 1** (Color online) Architecture of the Sunway processor.

## 2 Background and motivation

### 2.1 Sunway processor and its ecosystem

The Sunway SW26010 heterogeneous many-core processor (Sunway processor for short) runs at 1.45 GHz and offers 3.06 TFlops peak performance in double-precision floating-point operations. Figure 1 presents the architecture of Sunway processors. It contains four CGs, where each CG consists of an MPE and 64 CPEs. Specifically, the CPE adopts the cache-less design and has no data cache. But each CPE contains a 64 KB manually-controlled scratchpad memory (SPM), whose bandwidth and latency are similar to L1 cache. Moreover, the CPEs can transfer continuous data between SPM and main memory through direct memory access (DMA). Programmers should use the unique programming model, Athread, to leverage the above computation hierarchy and memory hierarchy explicitly [1].

Although the Sunway processor can provide excellent peak performance, programmers should still make lots of efforts to utilize its maximum computing power. Firstly, they should keep the architectural features in mind and focus on (1) managing computation parallelism and task assignment across MPE and CPEs, (2) allocating and freeing the limited SPM space, and (3) controlling the DMA transfer between main memory and SPM. Secondly, programmers must write `C` codes for CPEs carefully with the customized Athread programming model. The general-purpose compilers for Sunway processor just support ahead-of-time (AOT) compilation rather than just-in-time (JIT) compilation, therefore they are customized compilers for Sunway, which can compile `C++`, `C`, and `Fortran` codes for MPE, but can only compile `C` codes for CPE (through the sw5cc compiler). A large amount of research studies [14–24] have been proposed to exploit the uniqueness of Sunway architecture for optimizing application performance at different scales.

### 2.2 MLIR

MLIR is a production compiler infrastructure, which provides abstractions (also known as dialects) at different levels across application domains, hardware targets, and execution environments, as well as the transformation across these abstractions. Because of the reusable and extensible dialects, MLIR is a well-suited basis of domain-specific compilers. The compiler developers can maximize the reuse of the various components of the MLIR ecosystem, and just focus on their unique implementations, such as the domain- or target-specific definitions and optimizations. Many DL frameworks and compilers (e.g., TensorFlow [3], PlaidML [25], ONNX [26]) have already leveraged MLIR to deploy DL models across both high-performance and embedded CPUs/GPUs. In addition, there are research studies utilizing MLIR to generate GPU kernels [27], build dense/sparse tensor compilers [9–12, 28], and optimize hardware accelerator designs [29, 30].

Unfortunately, the MLIR ecosystem only supports mature hardware targets: CPUs and GPUs, including ARM CPU, x86 CPU, Nvidia GPU, AMD GPU, and other SPIR-V compatible targets. It means that the emerging processors cannot connect to the MLIR ecosystem and fail to share the benefits either, while few studies aim to tackle this problem.

### 2.3 Stencil DSCs

The stencil computations for scientific applications have various computation patterns (i.e., shapes, point numbers, dimensions) and diverse computation dependencies. And they are executed on various hardware targets, such as CPU, CPU, and other emerging processors. Therefore, to execute the stencil computations efficiently with satisfying performance portability, quantities of studies resort to building stencil DSCs [6, 13, 31–35]. Specifically, Halide [33] is a domain-specific compiler for image processing, including the corresponding stencil computations. It allows users to define the stencil computation and the optimizing rules (without concrete parameter settings) separately. Then Halide can perform auto-tuning to search for the best parameter settings through the genetic algorithm automatically. There are also studies devoted to improving the auto-tuning technique [36, 37]. Lift [35] is another important programming framework with a series of reusable parallel primitives. With extensions of two extra parallel primitives, it has demonstrated the capability to support stencil computations [35]. Artemis [34] focuses on optimizing multiple-statement stencils with complicated dependencies and generates high-performance CUDA codes on GPUs. MSC [13] optimizes stencil computation with multiple time dependencies on many-core processors and generates high-performance stencil codes for large-scale execution. Notably, although researchers have invested much effort to develop stencil DSCs from various aspects, except a few (only MSC, to the best of our knowledge), most stencil DSCs are designed for CPU or GPU, missing support for emerging processors.

### 2.4 Motivation

The MLIR infrastructure enables building domain-specific compilers across various application domains and hardware targets through a reusable approach. However, it only provides the reusability for mature hardware targets (e.g., CPU and GPU) through the pre-defined hardware backends. To realize the missing support in MLIR for emerging processors such as Sunway, in this paper, we provide a reusable compilation infrastructure on Sunway processor by designing reusable dialect (Sunway dialect) and reusable backend (Sunway backend) through extending MLIR. Then to demonstrate the effectiveness for easing the development for domain-specific compiler with the proposed compilation infrastructure, we build a domain-specific compiler for stencil computation by reusing the Sunway dialect and Sunway backend as well as existing dialects within MLIR. The contribution of this paper comes from two folds: (1) it enriches the ecosystem of MLIR by providing a reusable dialect and backend for emerging processors such as Sunway, and (2) it provides a reusable compilation infrastructure for building domain-specific compilers with better productivity on Sunway processor. The most relevant work to us is the multi-level IR rewriting [38]. Although both studies focus on building reusable domain-specific compilers, they are orthogonal and complementary. Because multi-level IR rewriting [38] adopts the top-down approach by designing reusable domain-specific frontends based on the existing hardware-specific dialects and backends, whereas our work adopts the bottom-up approach by building a reusable dialect and backend for all the above frontends.

## 3 Design overview

Based on the MLIR infrastructure, building domain-specific compilers on emerging processors can be simplified through many reusable components, including various dialects, conversions within a single dialect, and optimization passes between dialects. Many mature hardware backends (mainly CPU and GPU backends) are already connected to the MLIR. They can take the hardware-specific dialects as the input and generate the executable codes. Generally, a domain-specific compiler contains three parts, as shown in Figure 2. The frontend takes the domain-specific definition as the input, and converts the definition to domain-specific dialects. It should provide a programming model design for this application domain, considering the trade-off between expressibility and efficiency. The intermediate representation (IR) abstracts the computation with MLIR dialects. The dialects contain domain-specific dialects, generic dialects, and hardware-specific IR. Various transformations and compilation optimizations across these dialects are provided, and finally, these dialects are lowered into hardware-specific dialects (e.g., llvm/gpu/nvvm dialects). The backend takes the hardware-specific dialects as the input, and generates executable codes on corresponding hardware targets.

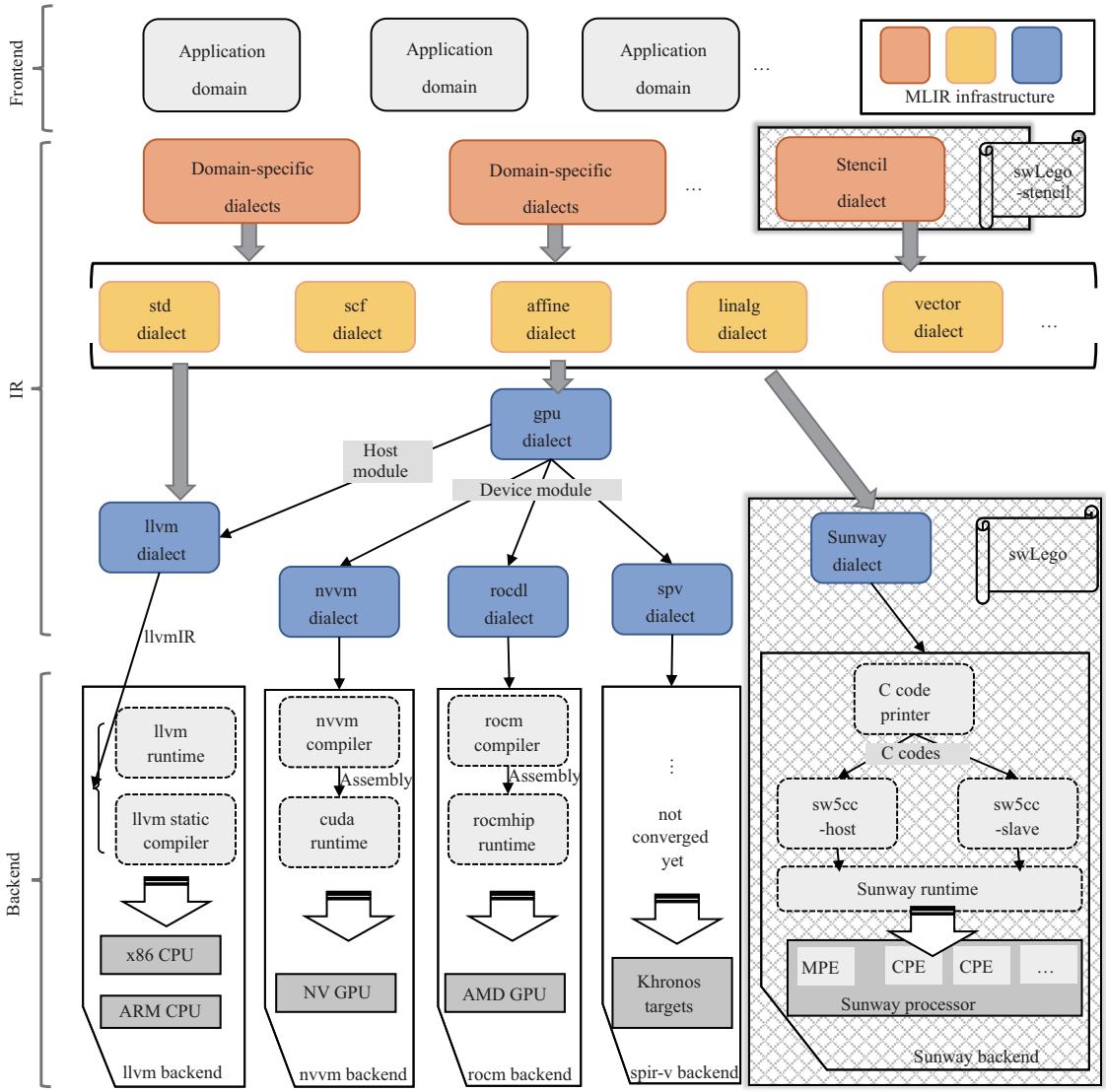


**Figure 2** (Color online) Design overview of building a domain-specific compiler based on the MLIR infrastructure and the extension of our work (shaded rectangle) to incorporate the Sunway processor.

As shown in Figure 2, there are two kinds of hardware-specific dialects: for CPU and GPU. On CPU, the llvm dialect is compiled into llvm IR and then passed to the llvm backend, including llvm runtime (JIT) and the llvm static compiler (AOT), and finally executed on CPUs. On GPU, the gpu dialect separates the host module to llvm dialect and the device module to the dedicated platform-specific dialects (nvvm dialect for Nvidia GPU, rocdl dialect for AMD GPU, and spv dialect for OpenCL/Vulkan targets). The dedicated dialects are then passed to nvvm/rocm/spv backend to compile. In this paper, we design Sunway dialect and Sunway backend to add support to the Sunway processor into the MLIR infrastructure and enable easy and efficient construction of domain-specific compilers for this emerging processor.

## 4 swLego, a reusable approach

swLego is implemented as a Sunway dialect together with a Sunway backend in the MLIR compiler infrastructure. Therefore it inherits the benefits provided by the MLIR infrastructure. The core concepts in MLIR include operations, attributes, regions, blocks, values, and types [8]. An operation is a unit of semantics in MLIR, and the instructions, functions, and modules can be modeled as operations. The operations are not fixed, and the user-defined operations are encouraged. The attributes contain the

compiler-time information about operations, and their associated operation semantics determines their meanings. An instance of an operation may contain a list of attached regions, and each region contains a list of blocks. Each block contains a list of operations, which forms the nesting mechanism in MLIR. A value represents the data at runtime, and during compile time, its information is stored in a type. Then a set of relevant operations, attributes, and types can be logically grouped into a dialect and then share a unique namespace. The dialects are visible to each other, and the dialects can be intermixed, which provides reusability and extensibility of MLIR. Therefore, we port the Sunway processor to the MLIR ecosystem by adding a Sunway dialect with customized operations, attributes, and types. After that, the Sunway dialect is reusable and extensible, similar to other dialects within the MLIR ecosystem.

### 4.1 Sunway dialect

Sunway dialect provides complete control of MPE and CPE on the Sunway processor while still providing high-level abstractions. This dialect is abstracted and designed following the programming models and the programming principles summarized by the optimization experts. The common approach of implementing high-performance kernels or applications on Sunway processors can be summarized into the following steps.

**(S1) Offloading.** Do performance profiling on MPE and identify the time-consuming parts. Then consider offloading these time-consuming parts to CPEs in the same CG. The CPEs can only run in the user model and not support interrupt functions. Therefore, they should be explicitly invoked from the MPE because CPEs aim to provide maximum aggregated performance with simplified architectural design.

**(S2) Task assignment.** Decompose the computation task into several sub-tasks and assign the sub-tasks to CPEs. The granularity of sub-tasks should be coarse enough to make full use of computation capability, SPM space, and DMA bandwidth. Besides, the programmers should carefully determine the number of sub-tasks to achieve load balance among CPEs.

**(S3) Performance optimizations.** Manage the SPM space and DMA transfer to utilize the memory hierarchy, and apply SIMD vectorization to maximize the computation power. The implementations of optimizations are tightly coupled with the original algorithms. Sometimes, programmers should develop new algorithms for extreme performance to apply more optimization.

**(S4) Compilation.** Compile MPE and CPE codes separately with different compiler options (–host and –slave), and then link them together to get the executable binary.

The Sunway dialect contains the MPE- and CPE-specific abstractions designed with respect to the approach described above. Specifically, the Sunway dialect can be classified into data types and six categories of operations, as shown in Table 1.

**Data types.** The data types supported by the Sunway dialect include 32-bit integer (`i32`), 64-bit integer (`i64`), 32-bit float (`f32`), 64-bit float (`f64`), and pointer (`sw.memref`, a pointer/reference to a multi-dimensional tensor), as well as 256-bit `vector` (because the Sunway processor only supports 256-bit vectors). Both the main memory shared between MPE and CPEs as well as the SPM on CPEs is referenced by `sw.memref`.

**Module/function operations.** Modules and functions help to form the IR structure of Sunway IR, and they are implemented as operations in Sunway dialects, which follow the similar design of the builtin dialect. `sw.module` contains the code that is intended to be run on Sunway and wraps a set of MPE and CPE kernels (following (S1) & (S4)). `sw.func` specifies a kernel executed on CPEs. Each `sw.func` has a block. The block arguments specify the values to be passed to the kernel, and the block body contains the kernel implementation represented by control-flow operations, computation operations, and MPE operations. Similarly, `sw.main_func` and `sw.main_iteration_func` specify the MPE kernels. And `sw.main_iteration_func` simplifies the iterative MPE kernels by calling `sw.main_func` multiple times.

**Launching operations.** These operations are responsible for launching MPE and CPE kernel (following (S2)). Because CPEs should be explicitly invoked from the MPE, the `sw.launch` and `sw.launch_func` should only exist inside the blocks of `sw.launch_main_func` (corresponding to (S1)).

**Control-flow operations.** These operations are designed to express the "if-then-else" and "if" branches and the "for" loops.

**Computation operations.** Both the arithmetic and logical operations are supported. The arithmetic operations can take values with scalar type and vector type as the input (corresponding to (S3)), whereas logical operations can only take values with scalar type as the input.

**Table 1** Types and operations of Sunway dialect

| Category | Examples |
|---|---|
| Data types | i32, i64, f32, f64,<br>sw.memref (2D, 3D), vector (256 bits) |
| Module/Func operations | `sw.module`, `sw.module_end`,<br>`sw.func`, `sw.main_func`<br>`sw.main_iteration_func`, `sw.func_return` |
| Launching operations | `sw.launch`, `sw.launch_func`,<br>`sw.launch_main_func` |
| Control-flow operations | `sw.for`, `sw.if`,<br>`sw.else`, `sw.yield` |
| Computation operations | `sw.add`, `sw.sub`, `sw.mul`, `sw.div`,<br>`sw.land`, `sw.lor`, `sw.lnot`, `sw.cmp` |
| MPE operations | `sw.alloc`, `sw.dealloc`,<br>`sw.getMpiRank`, `sw.mpiExchangeHalo` |
| CPE operations | `sw.getID`, `sw.constant`,<br>`sw.load`, `sw.store`, `sw.broadcast`<br>`sw.vectorLoad`, `sw.vectorStore`<br>`sw.memcpyToSPM`, `sw.memcpyToMEM` |

**MPE operations.** These operations are specific to MPE, including operations that malloc/free the main memory space and operations responsible for MPI communication (corresponding to (S1)).

**CPE operations.** These operations are specific to CPEs. `sw.getID` gets the id of a CPE within the located CG. `sw.constant` defines a constant number in SPM. `sw.load`/`sw.store` reads/writes scalar values in SPM. `sw.broadcast` broadcasts a scalar value to a vector value. `sw.vectorLoad`/`sw.vectorStore` reads/writes aligned and unaligned vector values in SPM, and can be further lowered to aligned/unaligned instructions according to the starting address. `sw.memcpyToSPM` and `sw.memcpyToMEM` are responsible for DMA transfers between SPM and main memory, targeting for utilizing the memory bandwidth (corresponding to (S3)).

### 4.2 Sunway backend

The Sunway backend is responsible for executing the Sunway dialects. It takes the Sunway dialect as the input, separates the MPE modules and the CPE modules, and then lowers them to `C` codes following the `Athread` programming model (as shown in Figure 3). Finally, it leverages the native `C` compiler, sw5cc –host/slave to compile the codes for MPE and CPE, respectively (as shown in Figure 2). Notably, Sunway processor does not support LLVM for now. Therefore, we lower the Sunway dialect to C codes and use Sunway's native compiler to generate executable binaries. If Sunway supports LLVM in the future, we can lower Sunway dialect to LLVM IR and utilize LLVM runtime to generate the optimized binaries.

The first step to lower Sunway dialect to `C` codes is outlining. The CPE kernel is defined within the `sw.launch` operation and exists in the MPE module. So it should be outlined to a separate kernel function, `sw.func`, into a dedicated module, `sw.module`. As for the MPE module, the `sw.launch` is replaced by launching operation, `sw.launch_func`, to invoke the outlined CPE function, `sw.func`.

As shown in Figure 3(a), the MPE module usually manages the main memory and invokes CPE functions. The block arguments (`sw.memref`) specified in the basic block of `sw.main_func` are lowered to the stack arguments in main memory, and the `sw.alloc`/`sw.dealloc` is lowered to the calloc/free instructions to manage the heap arguments in main memory. Then the parameters of a CPE function are packed into a structure, which contains the required parameters. Because CPE functions can only receive one parameter as the input, the `sw.launch_func` is lowered to the athread_spawn instruction with the kernel name and the packed structure.

As shown in Figure 3(b), the CPE module specifies the computation-intensive or bandwidth-intensive kernels to achieve high performance. The block arguments of `sw.func` are lowered to the main memory reference and SPM allocation according to the attributes of `sw.func`. Each CPE should have its ID, that is my_id, for task assignment, and the `sw.getID` is lowered to the athread_get_id instruction. And the `sw.for` is lowered to standard for loop with upper bound, lower bound, and stride. Note that programmers should control the loop counters and the stride to control the task assignment among CPEs. In Figure 3,

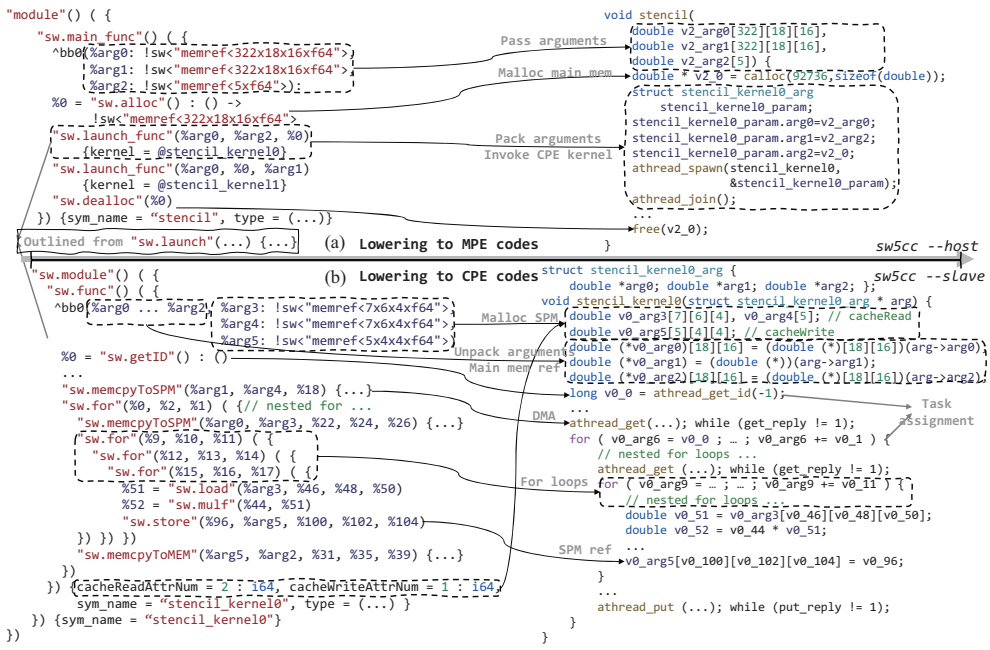


**Figure 3** (Color online) Lowering Sunway dialect to C codes for MPE and CPEs. The Sunway dialect in this example is generated from the stencil example described in Subsection 5.1.

the tasks whose task_id satisfies $\mathrm{mod}(\mathrm{task\_id}, 64) = \mathrm{my\_id}$ are assigned to CPE my_id. Besides, the `sw.memcpyToSPM/MEM` operations are lowered the DMA transfer instructions, athread_get/put, and the `sw.load/store` operations are lowered to direct SPM reference.

## 5 Building a stencil compiler

With the reusable swLego (i.e., Sunway dialect and Sunway backend), we can build domain-specific compilers on Sunway processors. As an example, we decide to build a stencil compiler, swLego-stencil on Sunway as a case study. We first design the programming language and develop a compiler frontend to parse the input (e.g., the definition of stencil computation) and transform it into domain-specific dialect (e.g., stencil dialect). Then we leverage the pattern rewriting mechanism provided by MLIR and transform the domain-specific dialect into other reusable dialects. At the same time, we also perform several compilation optimizations with domain-specific knowledge. Finally, we lower the above-transformed dialects to the Sunway dialect. Once passing the Sunway dialect to the Sunway backend, the final `C` codes can be derived through AOT compilation.

### 5.1 Programming language

To better illustrate the programming language of swLego-stencil, Figure 4 presents the example of a complex stencil computation, which contains time iteration and kernel dependency. The stencil name, the input grid, and coefficients are defined in Line 1, and the input grid is in-place updated during the computation. swLego-stencil supports stencils with time iterations, and the concrete number of iteration steps is defined through iteration (Line 2). Since multiple stencil kernels can be defined inside a stencil computation, the program should determine taking the output of which kernel to be the final output through operation (Line 3). The grid of MPI processes for large-scale execution is defined in Line 4, and the halo region's width in each spatial dimension is defined in Line 5.

Two stencil kernels, 3d5pt and element-wise add (3d1pt) are defined through the kernel function. The tile size of subsequent loop tiling optimization can be explicitly defined through tile (Line 8/17). The computation domain of a kernel can be smaller than the input grid, and its boundary is defined through domain (Line 9/18). The computation pattern of a kernel is defined through expr with enclosed curly braces, which updates the element (e.g., $(x, y, z)$) of the grid with the neighboring elements (e.g.,

```
1     stencil stencil_3d5pt_add(double U[322][18][16], c[5]) {
2         iteration(100)
3         operation(kernelName2)
4         mpiGrid(2, 2, 2)
5         mpiHalo([1,1][1,1][1,1])
6
7         kernel kernelName1 {
8             tile(5, 4, 4)
9             domain([1,321][1,17][0,16])
10            expr {
11                c[0]*U[x-1][y][z] + c[1]*U[x+1][y][z] +
12                c[2]*U[x][y+1][z] + c[3]*U[x][y-1][z] +
13                c[4]*U[x][y][z] }
14        }
15
16        kernel kernelName2 {
17            tile(5, 4, 4)
18            domain([1,321][1,17][0,16])
19            expr {
20                U[x][y][z] + kernelName1[x][y][z] }
21        }
22    }
```

**Figure 4** Programming language of swLego-stencil.

$(x-1, y, z)$, $(x+1, y, z)$). And the neighboring elements can locate on both the input grid (e.g., Line 11) and the output grid of other kernels (e.g., Line 20).

Then the frontend of swLego-stencil will parse the stencil definitions and transform them into the stencil dialect while preserving the domain-specific knowledge of stencils, such as the stencil kernels and the kernel dependency.

### 5.2 Reused dialects

Due to the reusable MLIR infrastructure, all components (e.g., dialects, transformations between dialects) can be reused to build a new domain-specific compiler. In swLego-stencil, we extend the stencil dialect, which is firstly proposed in the Open Earth Compiler [38], and reuse the std dialect, scf dialect, as well as the vector dialect, which belong to builtin MLIR dialects. And we extend MLIR by supplementing the Sunway dialect (described in Section 4) that represents the many-core and cache-less hardware architecture.

Specifically, we extend the stencil dialect with two operations: (1) `stencil.tile`, which allows partitioning the stencil computation of the whole grid into computation inside smaller tiles to exploit the data locality with improved parallelism. And (2) `stencil.iteration`, which allows expression of complex stencil computation with time iterations. Specifically, due to the complex stencil patterns and cache-less architecture of Sunway processor (i.e., manually controlled SPM on each CPE), the `tile` size can significantly impact the performance of generated stencil codes. Specifically, the tile size should be large enough to utilize the SPM for better data locality, but not too large otherwise more DMA bandwidth is wasted by larger halo regions, in addition to idle CPEs due to decreased parallelism. Therefore, we trade off the ease of programming for the performance of generated codes in the design of swLego-stencil, where we leave the setting of `tile` size to the programmer, which can be tuned with some auto-tuning algorithms.

We mix the std/scf/vector dialects with stencil dialect to express the arithmetic operations, control-flow operations, and vectorization operations. Since MLIR supports the mixing of dialects and progressive lowering, they can be lowered selectively after several optimization passes. And all of them are lowered into the Sunway dialect finally.

### 5.3 Optimizations

Two compilation optimizations are supported in swLego-stencil.

**Kernel fusion.** The complex stencils usually contain multiple stencil kernels, where the output of one kernel is used as the input of another kernel. In such a scenario, the dependent stencil kernels can be optimized through kernel fusion so that the fused kernel can just load the input grid once, and thus avoid the redundant intermediate data movement, such as storing the intermediate data in main memory and loading it into SPM. Besides, the kernel fusion reduces the number of kernel launching on CPEs, and thus reduces significant kernel launching overhead.

**Vectorization.** We apply vectorization to utilize the SIMD units on the Sunway processor. The arithmetic operations ($+ - \times \div$) (from std dialect) and memory reference (`load` and `store`) existing in stencil dialect are converted to vectorized operation in vector dialect, and then lowered to the vectorized operation in Sunway dialect. Specifically, the corresponding load/store and computation operations (e.g., `stencil.mulf`) are transformed as follows.

• `stencil.load` (load a `f64` scalar from SPM) → `stencil.load` (load a scalar from SPM) + `vector.broadcast` (broadcast this scalar into a memref⟨4xf64⟩ vector) → `sw.load` + `sw.broadcast`

• `stencil.store` (store a `f64` scalar to SPM) → `stencil.store` (store a memref⟨4xf64⟩ vector to SPM) → `sw.vectorStore`

• `stencil.mulf` (multiply two `f64` scalars) → `stencil.mulf` (multiply two memref⟨4xf64⟩ vectors) → `sw.mul` (can accept both scalars and vectors)

We have attempted to use `shuffle` operations for better data reuse. However, the shuffle parameters are tightly coupled with the stencil shape and the memory address of the center point. Determining these parameters to generate correct stencil codes is quite complicated, and thus we leave the support of `shuffle` for future work. Note that we only apply vectorization optimization to large stencils (with more than 50 points) to collaborate with the native Sunway compiler. Because the native Sunway compiler fails to apply automatic vectorization when tackling the complex data access patterns. Besides, the generic optimizations of LLVM have not been adopted in swLego-stencil, because the Sunway processor does not support LLVM currently.

For the optimizations adopted in swLego-stencil, the kernel fusion can be reused on other hardware platforms, since it only transforms the stencil dialect and is hardware agnostic. The vectorization optimization is specific to the Sunway processor, since it transforms the stencil dialect and vector dialect to the Sunway dialect. Therefore, it can only be reused on hardware platforms similar to Sunway (e.g., many-core and cache-less), or by other domain-specific compilers on Sunway processor.

Besides, swLego-stencil is feasible to invoke other optimized libraries, to reuse the state-of-the-art high-performance implementations. We reuse the communication library of MSC to handle the data exchange of halo regions when generating stencil codes running on multiple nodes. And we further improve this library to eliminate the redundant data transfer of irregular stencils with inconsistent halo sizes (e.g., halo = 0 in $(-1, 0, 0)$ direction while halo = 1 in $(1, 0, 0)$ direction).

As far as we know, there is no MPI dialect in MLIR. Since MPI itself is a complicated software, designing MPI dialect to support large-scale communication in general would inevitably increase the complexity of managing the compilation stack. The community is more agreed on supporting MPI through external library calls tailored for DSCs, in such a way the complexity of maintaining MLIR is isolated. So that MLIR can provide a loosely coupled compilation infrastructure for building DSCs. In swLego-stencil, we adopt the above principle by reusing an MPI communication library optimized for large-scale stencil computation from MSC and integrating it through external library calls during compilation.

## 6 Evaluation

### 6.1 Experiment setup

We collect various representative stencil benchmarks (shown in Table 2), including stencils with general patterns, stencils with arbitrary patterns, and the nested stencils, for performance evaluation. On a single Sunway processor, we compare the stencil codes generated by swLego-stencil with the codes manually optimized through OpenACC and Athread, as well as the code generated by the state-of-the-art stencil compiler MSC [13]. The OpenACC baseline, Athread, and MSC adopt the same optimization techniques (utilizing the CPE parallelism, SPM, and DMA) as swLego-stencil for a fair comparison. We then apply sensitivity analysis and roofline model analysis to study the performance of swLego-stencil. We measure the scalability on multiple Sunway processors using up to 66560 cores (including both MPE and CPE). Specifically, we evaluate the above codes ten times and report their average execution time in case of performance fluctuation. We also measure the relative errors between the swLego-stencil-generated codes and the serial codes. Across all benchmarks, the relative errors of swLego-stencil-generated codes are less than $10^{-10}$ in double precision, which indicates swLego-stencil can ensure the correctness of the stencil computation [38].

**Table 2** Stencil benchmarks used in the evaluation

| Benchmark | Read (Byte) | Write (Byte) | Ops ($+ - \times \cdots$) |
|---|---|---|---|
| 2d9pt_star | 72 | 8 | 26 |
| 2d81pt_box | 648 | 8 | 242 |
| 2d121pt_box | 968 | 8 | 363 |
| 2d5pt_arbitrary | 40 | 8 | 14 |
| 2d5pt5pt_nested | 80 | 16 | 28 |
| 3d13pt_star | 104 | 8 | 38 |
| 3d27pt_box | 216 | 8 | 80 |
| 3d125pt_box | 1000 | 8 | 374 |
| 3d7pt_arbitrary | 56 | 8 | 20 |
| 3d7pt9pt_nested | 128 | 16 | 46 |

**Table 3** Parameter settings of swLego-stencil across the stencil benchmarks on a Sunway (a CG) processor

| Stencil | Grid size | Tile size |
|---|---|---|
| 2d9pt_star | $4096^2$ | $(32, 64)$ |
| 2d81pt_box | | |
| 2d121pt_box | | |
| 2d5pt_arbitrary | | |
| 2d5pt5pt_nested | $4096^2$ | $(32, 64)/(16, 32)$ |
| 3d13pt_star | $256^3$ | $(2, 8, 64)$ |
| 3d27pt_box | | |
| 3d125pt_box | | |
| 3d7pt_arbitrary | | |
| 3d7pt9pt_nested | $256^3$ | $(8, 4, 4)/(4, 8, 8)$ |

### 6.2 Overall performance on a single node

On the Sunway platform, we execute the swLego-stencil compiler on the login nodes (equipped with x86 CPU) to generate the optimized stencil codes, then evaluate them on a Sunway node (equipped with Sunway processors). The grid size of 3D stencils is set to $256^3$, which is the same as Physis [32]. The grid size of 2D stencils is set to $4096^2$, so the 2D/3D stencils have the same number of points in the input grid. Table 3 shows the parameter settings of swLego-stencil across the benchmarks. Note that the nested stencil benchmarks (2d5pt5pt_nested and 3d7pt9pt_nested) have two tile sizes for their two nested kernels, respectively.

On Sunway, the users can adopt OpenACC and Athread to optimize the stencils manually. (1) We use codes optimized by OpenACC as the baseline. We select the directives (#pragma acc ...) of data caching (acc copyin/copyout), loop splitting (acc tile), and multi-threading (acc parallel) to accelerate the stencil codes. (2) The Athread instructions provide a more fine-grained approach to optimizing stencil codes than OpenACC, which requires more lines of codes. We also optimize the stencil codes with Athread for comparison. We also compare swLego-stencil with the state-of-the-art stencil compiler, MSC. However, MSC does not support nested stencils, and we give up MSC on the nested benchmarks. Figure 5 presents the execution time of OpenACC, Athread, MSC, and swLego-stencil codes under double precision.

The performance of swLego-stencil-generated codes and manual Athread codes outperforms the OpenACC baseline in all cases. Compared with the baseline, swLego-stencil-generated codes achieve 1.72× speedup on average, whereas manual Athread codes achieve 1.33× speedup on average. And the MSC-generated codes have 1.19× speedup on average compared with the baseline. Without applying the kernel fusion and vectorization, swLego-stencil achieves 1.33× speedup on average, which is similar to Athread codes. This is because both swLego-stencil and Athread can leverage the customized programming model of Sunway, to exploit the fine-grained control of architectural features, such as SPM and DMA transfer. Besides, thanks to the MLIR infrastructure, it is convenient to implement peephole optimizations rather than complicated loop transformations. Thus swLego-stencil can further accelerate the nested stencils with kernel fusion optimization and accelerate the large stencils with vectorization optimization, for superior performance. With kernel fusion, the nested stencils (e.g., 2d5pt_nested and 3d7pt9pt_nested) achieve an additional performance improvement, since the fused kernels can avoid redundant intermediate data movement and minimize the kernel launching overhead on CPEs. For vectorization, the native Sunway

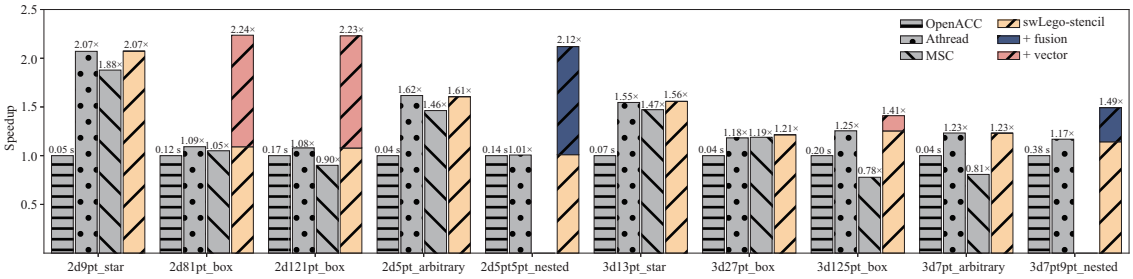


**Figure 5** (Color online) Performance comparison between swLego-stencil-generated codes, MSC-generated codes, and manually optimized OpenACC/Athread codes on a Sunway CG, where OpenACC is set as the baseline.

compiler can apply more aggressive vectorization optimization than swLego-stencil on small stencils (e.g., 2d9pt_star), and thus swLego-stencil achieves few performance improvements on these stencils. However, on large stencils (e.g., 2d81pt_box and 2d121pt_box), the native Sunway compiler fails to apply automatic vectorization due to the complex data access patterns, whereas swLego-stencil can better handle such stencils to generate vectorized codes with considerable performance improvements. Although these optimizations can also be manually implemented through Athread, implementing them for various stencils seems to be tedious and error-prone, while swLego-stencil alleviates the burden of hand-tuning.

Compared with MSC, swLego-stencil achieves 1.44× speedup on average. The MSC-generated codes are slower than the OpenACC codes in some stencil benchmarks, especially on large stencils (2d81pt_box, 2d121pt_box, 3d125pt_box) and arbitrary stencils (3d7pt_arbitrary). On large stencils, the slowdown is attributed to the redundant data indexing codes of the neighboring elements, which introduces more useless computation. In contrast, swLego-stencil avoids this redundancy by introducing common subexpression elimination (CSE). On arbitrary stencils, the slowdown is attributed to the redundant data in halo regions caused by the overlapped tiling. The halo size can be different in different directions, but MSC assumes all halo sizes are equal and forces to fix the halo size to the maximum. While swLego-stencil considers the difference of halo sizes and thus avoids this redundancy.

### 6.3 Sensitivity to tile size

The tile size of a stencil kernel should be defined explicitly through tile(tile_$x$, tile_$y$, tile_$z$) (3D stencils) and tile(tile_$x$, tile_$y$) (2D stencils). If the tile size is too small, the elements within a tile would be restricted, and the overlapped (useless) regions between tiles would be too large. Therefore, the efficiency of DMA transfer is harmed due to the small transfer size and the low-proportional useful data. If the tile size is too large, more SPM space of a CPE is required, which may lead to the over-subscription of SPM, and fewer tasks are available to be assigned to CPEs, which may lead to the load imbalance across CPEs. To evaluate the performance sensitivity of swLego-stencil to the tile size, we measure the performance of swLego-stencil-generated codes under different tile size configurations. As the nested stencils have more than one stencil kernels, which can have different configurations, we omit the nested stencils here.

Figure 6 shows the performance heatmap under different tile size configurations. It is clear that with the increasing tile size, the generated stencil codes tend to achieve better performance. Besides, the large stencils (e.g., 2d81pt_box, 2d121pt_box, and 3d125pt_box) are less sensitive compared with other small stencils. Because large stencils have higher computation intensity, the DMA transfer time and the computation time can be overlapped better, which restricts the impact of the decreased DMA efficiency. However, if the tile size is too large (e.g., tile_$x$ × tile_$y$ = 32, tile_$z$ = 64 on 3d13pt_box and 3d125pt_box), the generated codes fail to execute on the Sunway processor due to over-subscription of SPM, and thus the corresponding blocks on the heatmap are left blank. Therefore, we configure the tile size of all stencil benchmarks to the value as large as possible for superior performance.

### 6.4 Roofline model analysis

The roofline model analysis helps us to better understand the swLego-stencil-generated codes. As shown in Figure 7, all stencil benchmarks lie on the left of the ridge point, meaning that their performance is all bounded by the memory bandwidth. The large stencils with box shapes (e.g., 2d81pt_box, 2d121pt_box, and 3d125pt_box) have higher operational intensity and achieve better performance than others. Because they access the input grid for more times and benefit from the data reuse. Similarly, this observation can

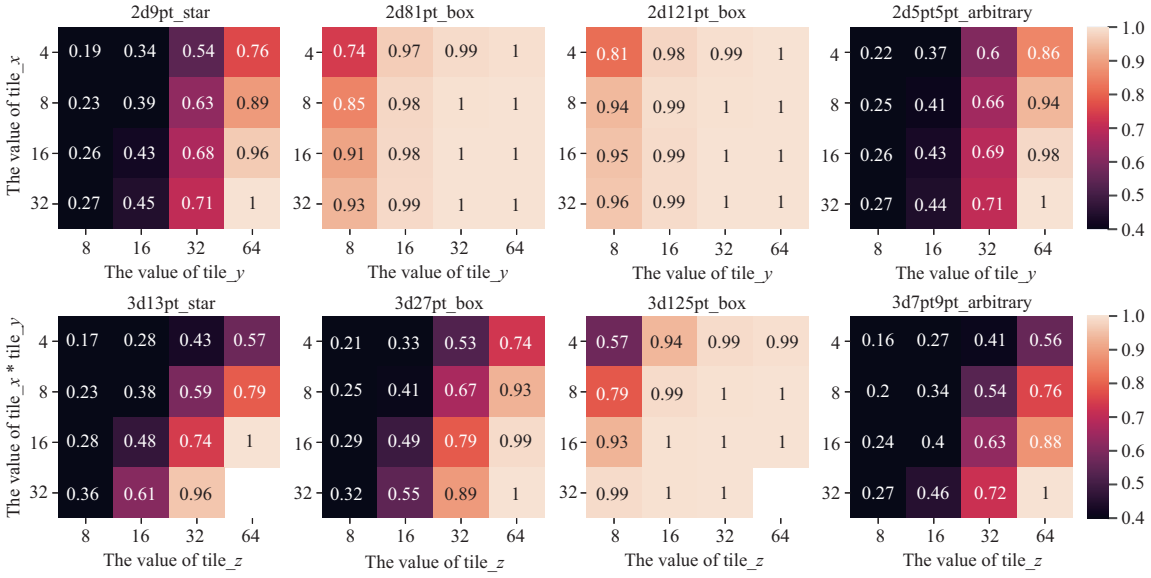


**Figure 6** (Color online) Performance sensitivity of swLego-stencil to the tile size. Each cell in the heatmap is the performance deviations to the optimal configurations.

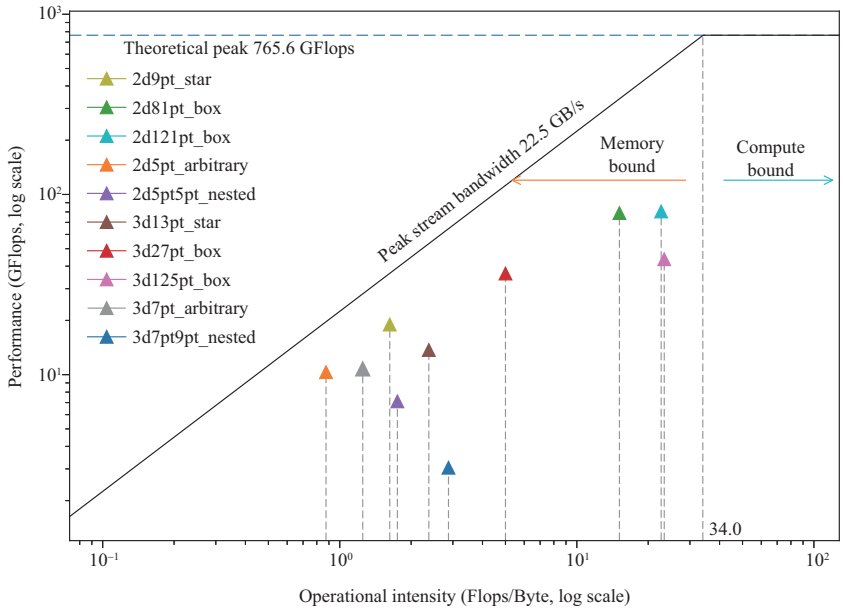


**Figure 7** (Color online) Roofline analysis of all stencil benchmarks optimized by swLego-stencil on a CG of Sunway processor.

apply to the stencils with star shapes. The stencil with arbitrary shapes and the nested stencils containing stencils with arbitrary shapes can also suffer from a similar problem when their shapes are relatively sparse. In addition, the stencils with larger neighboring regions (e.g., 3d125pt_box versus 2d121pt_box, 2d9pt_star versus 3d13pt_star) show lower performance, because they have larger halo regions and thus more redundant data transfer via DMA, which further degrades the bound of memory bandwidth.

Therefore, the stencil compilers should pay more attention to optimizing the memory reference patterns and improving data locality. They can leverage the polyhedral transformations and other techniques from deep learning domains to accelerate stencil computation. Based on the MLIR infrastructure, these optimization techniques can be reused by stencil compilers on Sunway processors in less complex ways.

**Table 4** Configuration of scalability experiments of swLego-stencil

| Dimension | Strong Scalability | Weak Scalability | MPI Grid | Processes |
|---|---|---|---|---|
| | Sub_grid per MPI | Sub_grid per MPI | | |
| 2D | $4096 \times 4096$ | $4096^2$ | $16 \times 8$ | 128 |
| | $4096 \times 2048$ | $4096^2$ | $16 \times 16$ | 256 |
| | $2048 \times 2048$ | $4096^2$ | $32 \times 16$ | 512 |
| | $2048 \times 1024$ | $4096^2$ | $32 \times 32$ | 1024 |
| 3D | $256 \times 256 \times 256$ | $256^3$ | $8 \times 4 \times 4$ | 128 |
| | $256 \times 256 \times 128$ | $256^3$ | $8 \times 4 \times 8$ | 256 |
| | $256 \times 128 \times 128$ | $256^3$ | $8 \times 8 \times 8$ | 512 |
| | $128 \times 128 \times 128$ | $256^3$ | $16 \times 8 \times 8$ | 1024 |

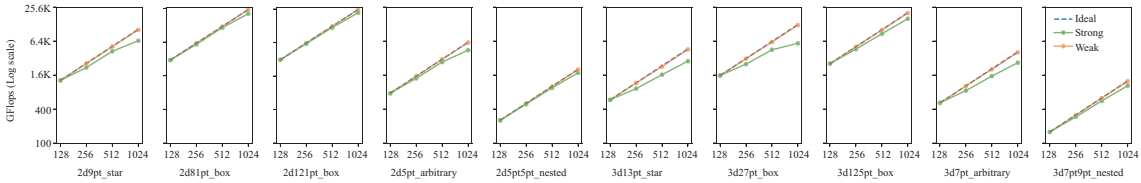


**Figure 8** (Color online) Strong and weak scalability of swLego-stencil with 128, 256, 512, and 1024 MPI processes.

### 6.5 Scaling to multiple nodes

We evaluate the scalability of swLego-stencil-generated codes on multiple Sunway processors. The configurations are shown in Table 4. We assign a CG to each MPI process, and execute the swLego-stencil-generated codes using 128–1024 MPI processes, that is, from 8320 cores (128 processes $\times$ 65 cores per CG) to 66560 cores (1024 processes $\times$ 65 cores per CG). For strong scalability, we keep the size of the input grid constant. When doubling the number of MPI processes, the size of the sub-grid per process is cut by half. For weak scalability, we keep the size of the sub-grid per process constant. When doubling the number of MPI processes, the input grid's size is consequently doubled.

Figure 8 shows the scalability results on all stencil benchmarks. As for strong scalability, swLego-stencil-generated codes achieve the scaling efficiency of 88.9%, 84.3%, and 72.6% on average with 2×, 4×, and 8× MPI processes, respectively. However, the scaling efficiency of stencils with star shapes and arbitrary shapes is lower than the average, especially with more MPI processes. This is because these stencils have larger halo regions and thus more MPI data transfer, which cannot overlap with the stencil computation. As for weak scalability, swLego-stencil-generated codes achieve the scaling efficiency of 99.6%, 98.9%, and 98.6% on average with 2×, 4×, and 8× MPI processes, respectively. It indicates that the weak scaling efficiency is almost ideal.

Note that the communication library embedded in swLego-stencil is borrowed from the pluggable library of the MSC compiler, and we perform several modifications on it to support 3D stencils with box shapes and stencils with arbitrary shapes. swLego-stencil still keeps the ability to leverage other communication libraries and even various kinds of kernel libraries, based on the Sunway dialect and Sunway backend.

## 7 Conclusion

In this paper, we propose swLego, a new dialect and backend for the Sunway processor, fully exploiting its architectural advantage and hiding its programming complexity. swLego is built on top of the MLIR infrastructure and then leverages reusable MLIR dialects to facilitate the development of DSCs for Sunway processors. The Sunway dialect abstracts a specific programming paradigm and unique optimization strategies that exploit the architectural advantage of the Sunway processor. Moreover, the Sunway backend executes the Sunway dialect by lowering to MPE and CPE codes and then compiles the codes in an ahead-of-time manner. To show the easiness of building a DSC, we implement swLego-stencil, a stencil compiler based on swLego for Sunway processors. It supports complex stencil computation with arbitrary shapes and inter-kernel dependency. Furthermore, by implementing tiling optimization, kernel fusion, and vectorization within swLego reusable compilation stacks, it can generate high-performance

stencil codes tailored for Sunway processors. The experimental results show that the code generated by swLego-stencil can achieve comparable performance with an expert-tuned stencil code and achieve even superior performance than the code generated by the state-of-the-art stencil compiler. We hope that our work could demonstrate a way to ease the development of DSCs on emerging processors.

**Acknowledgements** This work was supported by National Key Research and Development Program of China (Grant No. 2020YFB1506703), National Natural Science Foundation of China (Grant Nos. 62072018, 61732002, U22A2028), State Key Laboratory of Software Development Environment (Grant No. SKLSDE-2021ZX-06), and Fundamental Research Funds for the Central Universities (Grant No. YWF-22-L-1127).

**References**

1 Fu H H, Liao J F, Yang J Z, et al. The Sunway TaihuLight supercomputer: system and applications. Sci China Inf Sci, 2016, 59: 072001
2 Li M Z, Liu Y, Liu X Y, et al. The deep learning compiler: a comprehensive survey. IEEE Trans Parallel Distrib Syst, 2021, 32: 708–727
3 Leary C, Wang T. XLA: TensorFlow, compiled. TensorFlow Dev Summit, 2017. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html
4 Chen T Q, Moreau T, Jiang Z H, et al. TVM: an automated end-to-end optimizing compiler for deep learning. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation, Carlsbad, 2018. 578–594
5 Bondhugula U, Hartono A, Ramanujam J, et al. A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, 2008. 101–113
6 Gysi T, Osuna C, Fuhrer O, et al. STELLA: a domain-specific tool for structured grid methods in weather and climate models. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New York, 2015. 1–12
7 Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of International Symposium on Code Generation and Optimization, San Jose, 2004. 75–86
8 Lattner C, Amini M, Bondhugula U, et al. MLIR: scaling compiler infrastructure for domain specific computation. In: Proceedings of International Symposium on Code Generation and Optimization, Seoul, 2021. 2–14
9 Vasilache N, Zinenko O, Bik A J C. Composable and modular code generation in MLIR: a structured and retargetable approach to tensor compiler construction. 2022. ArXiv:2202.03293
10 Bik A J C, Koanantakool P, Shpeisman T, et al. Compiler support for sparse tensor computations in MLIR. ACM Trans Archit Code Optim, 2022, 19: 1–25
11 Tian R Q, Guo L Z, Li, J J, et al. A high performance sparse tensor algebra compiler in MLIR. In: Proceedings of Workshop on the LLVM Compiler Infrastructure in HPC, St. Louis, 2021. 27–38
12 Jeong G, Kestor G, Chatarasi P, et al. Union: a unified HW-SW co-design ecosystem in MLIR for evaluating tensor operations on spatial accelerators. In: Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques, Atlanta, 2021. 30–44
13 Li M Z, Liu Y, Hu Y M, et al. Automatic code generation and optimization of large-scale stencil computation on many-core processors. In: Proceedings of the International Conference on Parallel Processing, Lemont, 2021. 1–12
14 Yang C, Xue W, Fu H H, et al. 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, 2016. 57–68
15 Chen B W, Fu H H, Wei Y W, et al. Simulating the Wenchuan earthquake with accurate surface topography on Sunway TaihuLight. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, Dallas, 2018. 517–528
16 Duan X H, Gao P, Zhang T J, et al. Redesigning LAMMPS for peta-scale and hundred-billion-atom simulation on Sunway TaihuLight. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, Dallas, 2018. 148–159
17 Liu Y, Liu X, Li F, et al. Closing the "Quantum Supremacy" gap: achieving real-time simulation of a random quantum circuit using a new Sunway supercomputer. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, New York, 2021. 1–12
18 Liu C X, Xie B W, Liu X, et al. Towards efficient SpMV on Sunway Manycore architectures. In: Proceedings of the International Conference on Supercomputing, Beijing, 2018. 363–373
19 Li M Z, Liu Y, Yang H L, et al. Multi-role SpTRSV on Sunway many-core architecture. In: Proceedings of International Conference on High Performance Computing and Communications, Exeter, 2018. 594–601
20 Wang X L, Liu W F, Xue W, et al. SwSpTRSV: a fast sparse triangular solve with sparse level tile layout on Sunway architectures. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Vienna, 2018. 338–353
21 Li M Z, Liu Y, Yang H L, et al. Accelerating sparse Cholesky factorization on Sunway Manycore architecture. IEEE Trans Parallel Distrib Syst, 2020, 31: 1636–1650
22 Fang J R, Fu H H, Zhao W L, et al. swDNN: a library for accelerating deep learning applications on Sunway TaihuLight. In: Proceedings of International Parallel and Distributed Processing Symposium, Orlando, 2017. 615–624
23 Han Q C, Yang H L, Dun M, et al. Towards efficient tile low-rank GEMM computation on Sunway many-core processors. J Supercomput, 2021, 77: 4533–4564
24 Zhong X G, Li M Z, Yang H L, et al. swMR: a framework for accelerating MapReduce applications on Sunway Taihulight. IEEE Trans Emerg Top Comput, 2018, 9: 1020–1030
25 Zerrell T, Bruestle J. Stripe: tensor compilation via the nested polyhedral model. 2019. ArXiv:1903.06498
26 Jin T, Bercea G T, Le T D. Compiling ONNX neural network models using MLIR. 2020. ArXiv:2008.08272
27 Katel N, Khandelwal V, Bondhugula U. High performance GPU code generation for matrix-matrix multiplication using MLIR: some early results. 2021. ArXiv:2108.13191
28 Komisarczyk K, Chelini L, Vadivel K, et al. PET-to-MLIR: a polyhedral front-end for MLIR. In: Proceedings of Euromicro Conference on Digital System Design, Kranj, 2020. 551–556

29 Majumder K, Bondhugula U. HIR: an MLIR-based intermediate representation for hardware accelerator description. 2021. ArXiv:2103.00194
30 Zhao R Z, Cheng J Y. Phism: polyhedral high-level synthesis in MLIR. 2021. ArXiv:2103.15103
31 Yount C, Tobin J, Breuer A, et al. YASK-Yet another stencil kernel: a framework for HPC stencil code-generation and tuning. In: Proceedings of International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, Salt Lake City, 2016. 30–39
32 Maruyama N, Nomura T, Sato K, et al. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, 2011. 1–12
33 Ragan-Kelley J, Barnes C, Adams A, et al. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Seattle, 2013. 519–530
34 Rawat P S, Vaidya M, Sukumaran-Rajam A, et al. On optimizing complex stencils on GPUs. In: Proceedings of International Parallel and Distributed Processing Symposium, Rio de Janeiro, 2019. 641–652
35 Hagedorn B, Stoltzfus L, Steuwer M, et al. High performance stencil code generation with lift. In: Proceedings of the International Symposium on Code Generation and Optimization, Vienna, 2018. 100–112
36 Ansel J, Kamil S, Veeramachaneni K, et al. OpenTuner: an extensible framework for program autotuning. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Edmonton, 2014. 303–316
37 Sun Q X, Liu Y, Yang H L, et al. csTuner: scalable auto-tuning framework for complex stencil computation on GPUs. In: Proceedings of International Conference on Cluster Computing, Portland, 2021. 192–203
38 Gysi T, Müller C, Zinenko O, et al. Domain-specific multi-level IR rewriting for GPU: the open earth compiler for GPU-accelerated climate simulation. ACM Trans Archit Code Optim, 2021, 18: 1–23

# Profile of Depei QIAN

Prof. Depei QIAN graduated from Xi'an Jiaotong University in 1977, majoring in computer science. He was supported by the Chinese government to pursue graduate study in the United States from 1982 to 1984 and received an M.S. degree from North Texas State University. He was a senior visiting scholar at Hanover University in Germany from June 1991 to March 1992. He worked at Xi'an Jiaotong University from 1977 to 2010. In April 2000, he joined Beihang University and has been working at the university as a professor since then. He was invited by Sun Yat-sen University to serve as the dean of the School of Data and Computer Science from November 2015 to December 2019.

He has been working on computer architectures and systems for 45 years. In the 1980s, he participated in the development of China's first Lisp machine for AI applications, the Lips-M1 system. He was also involved in the development of a fault-tolerant computer for power grid stability control and several general-purpose micro-controllers (GMC family) for the control of automatic machine tools and other industrial applications. Since his visit to Hanover University, his research interests have shifted to parallel computer architectures. He developed two simulators (ArchSim and ParaSim) for parallel computer architecture research. In 1996, he was selected as the expert for the National High-tech R&D Program (the 863 Program). Since then, he has devoted more than 25 years of work to national research programs on information technologies. He has served as the director of the expert group of four national key projects on high-performance computing (HPC). Some of the achievements of these key projects include the national HPC environment (CNGrid) and several world-leading-class high-performance computers, such as Tianhe, Sunway, and Sugon. He has served as the editorial board member of several academic journals, including *National Science Review*, *Chinese Journal of Computers*, *Journal of Computer Research and Development*, and *Journal of Frontiers of Computer Science and Technology*. He is the editor-in-chief of *CCF Transactions on High Performance Computing* and *Frontiers of Data and Computing*.

Prof. Qian has received several awards for achievements in science and technology, including three national-level second-class awards (2007, 2009, and 2020), two provincial-level first-class awards (2007 and 2014), and four ministerial/provincial-level second-class awards (1988, 1889, 1993, and 2007). He is a fellow of the China Computer Federation. In 2021, he was elected as an academician of the Chinese Academy of Sciences. His current research interests include parallel computer architectures, performance evaluation and optimization, parallel programming support, and novel architectures for deep learning.

## Improve parallel programming with a holistic approach

With the advent of multi-/many-core processors, parallel programming has become an essential issue in all domains using computing as a tool. The huge scale, complex structure, and heterogeneous nature of modern supercomputers make parallel programming a highly formidable task. Prof. Qian and his team have proposed a holistic approach to attack the difficulties in parallel programming from the aspects of the program model, language/compiler, debugging, runtime, and architectural support.

**(1) Programming models.** Emerging workloads on GPUs involve more dynamic data sharing among threads. However, dynamic data sharing implemented with locks tends to have poor scalability and introduces livelocks caused by the SIMT execution paradigm of GPUs. Accordingly, a novel software transactional memory mechanism for GPU (GPU-STM) is proposed. It uses hierarchical validation and encounter-time lock-sorting to deal with the two challenges, respectively (Xu et al., 2014). The GPU-STM prototype is implemented on the commercial GPU platform. The evaluation shows that the GPU-STM outperforms coarse-grain locks on GPUs by up to 20×. To adapt to more emerging applications, a software-based thread-level synchronization mechanism called lock stealing for GPUs is proposed to avoid livelocks (Gao et al., 2020). The mechanism is implemented as a thread-level locking library (TLLL) for commercial GPUs. The evaluation results show that, compared with the state-of-the-art ad-hoc GPU synchronization for Delaunay mesh refinement, the TLLL improves the performance by 22% on average on a GTX970 GPU and shows up to 11% performance improvement on a Volta V100 GPU. Online transaction processing (OLTP) on GPUs is also facing the challenges of branch divergences caused by the SIMT execution paradigm and lack of fine-grained synchronization mechanisms and pointer-based dynamic data structures. A high-performance in-memory transaction processing system GPU-TPS is proposed to perform synchronization among transactions and optimize indexing structures in OLTP systems (Gao et al., 2019). The experiment results show that the GPU-TPS outperforms the state-of-the-art CPU implementation DrTM by 3.8× for SmallBank and by 1.9× for TPCC and outperforms the GPU implementation by 1.6× for SmallBank and by 1.8× for TPCC.

**(2) Parallel debugging.** Sequential consistency (SC) is the most intuitive memory model used to guarantee the correctness of parallel program execution. For performance, most modern processors adopt relaxed SC models and rely on the correct placement of synchronizations, such as FENCE, to maintain SC execution. Record and deterministic replay (R&R) of multithreaded programs on relaxed-consistency multiprocessors has been a long-standing open problem. Volition, the first hardware R&R scheme, was proposed to address this problem (Qian et al., 2013). Volition leverages cache coherence protocol transactions to dynamically detect cycles in memory-access orders across threads and precisely detects SCVs in a relaxed-consistency machine in a scalable manner for an arbitrary number of processors in

the cycle. Based on Volition, two hardware R&R schemes, Pacifier (Qian et al., 2014) and Rainbow (Qian et al., 2013), were proposed for the distributed directory-based protocol and snoopy protocol, respectively. These schemes significantly reduce the log size and overhead of R&R. An evaluation with simulations of 16, 32, and 64 processors with release consistency running SPLASH-2 applications indicates that Pacifier incurs only 3.9%–16% larger logs and 10.1%–30.5% slowdown compared with native execution. Simulations with 10 SPLASH-2 benchmarks show that Rainbow reduces the log size by 26.6% and improves the replay speed by 26.8% compared with Strata, another hardware R&R scheme.

**(3) Runtime optimization.** Current distributed graph processing frameworks cannot precisely enforce the semantics of user-defined functions, leading to unnecessary computation and communication. SympleGraph (Zhuo et al., 2020), a novel distributed graph processing framework, was proposed to deal with this issue. It precisely enforces loop-carried dependency; i.e., when a condition is satisfied by a neighbor, all following neighbors can be skipped. Enforcing loop-carried dependency requires the sequential processing of the neighbors of each vertex distributed in different nodes. Circulant scheduling is adopted by the framework to allow different machines to process disjoint sets of edges/vertices in parallel while satisfying the sequential requirement. In a 16-node cluster, SympleGraph outperforms the state-of-the-art system Gemini on average by 1.42× and up to 2.30×. The communication reduction compared with Gemini is 40.95% on average and up to 67.48%. Redundant zeros cause inefficiencies, in which the zero values are loaded and computed repeatedly, resulting in unnecessary memory traffic and identity computation. A fine-grained profiler, ZeroSpy (You et al., 2020), was developed to identify redundant zeros caused by the inappropriate use of data structures and useless computation. ZeroSpy provides intuitive optimization guidance by revealing locations where redundant zeros happen in source lines and calling contexts. The experimental results demonstrate ZeroSpy can identify redundant zeros in programs that have been highly optimized for years. Based on the optimization guidance revealed by ZeroSpy, a significant speedup can be achieved after eliminating redundant zeros. SpTFS, a framework for the automatic prediction of the optimal storage format for input sparse tensors, was developed (Sun et al., 2020). SpTFS lowers high-dimensional sparse tensors into fix-sized matrices and accurately predicts the optimal format. The experimental results show that SpTFS achieves a prediction accuracy of 92.7% and 96% on the CPU and GPU, respectively.

**(4) Architectural support.** A hardware-enhanced cache coherence protocol is proposed to improve the performance of traditional directory-based protocols (Wang et al., 2015). The protocol is motivated by the observation that many referenced memory blocks were only accessed by a single core and can be treated as private memory blocks. A novel hardware approach is adopted to dynamically identify the shared memory blocks at the cache block level and bypass the coherence procedure for private memory blocks. Experimental results show that this approach can, on average, (1) avoid the coherence tracking of ∼54% referenced memory blocks, (2) reduce the coherence overhead by 77%, (3) avoid 8% L2 cache misses, and (4) shorten the execution time of parallel applications by 13%. Processors allowing continuous execution of atomic blocks of instructions called chunks can achieve high parallelism while keeping the correctness of the SC model. However, checking whether conflicts occurred in the chunk at the commit point of a chunk becomes the bottleneck. BulkCommit, a scheme providing a scalable and fast chunk commit for a large many-core system, is proposed (Qian et al., 2013). The scheme achieves a fast and scalable commit through (1) the serialization of the write sets of output-dependent chunks to avoid squashes and (2) the full parallelization of directory module ownership by the committing chunks. Simulations with PARSEC and SPLASH-2 codes for 64 processors show that BulkCommit eliminates most squashes and commit stall times, speeding up the codes by an average of 40% and 18% compared with previously proposed schemes.

## Selected publications


• Sun Q X, Liu Y, Yang H L, et al. CoGNN: efficient scheduling for concurrent GNN training on GPUs. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, 2022. 538–552

• Li M Z, Liu Y, Liu X Y, et al. The deep learning compiler: a comprehensive survey. IEEE Trans Parall Distr Syst, 2021, 32: 708–727

• Sun Q X, Liu Y, Yang H L, et al. Input-aware sparse tensor storage format selection for optimizing MTTKRP. IEEE Trans Comput, 2021, 71: 1968–1981

• Sun Q X, Liu Y, Dun M, et al. SpTFS: sparse tensor format selection for MTTKRP via deep learning. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, 2020. 1–14

• You X, Yang H L, Luan Z Z, et al. ZeroSpy: exploring software inefficiency with redundant zeros. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, 2020. 1–14

• Zhuo Y W, Chen J J, Luo Q Y, et al. SympleGraph: distributed graph processing with precise loop-carried dependency guarantee. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, London, 2020. 592–607

• Gao L, Xu Y L, Wang R, Luan Z Z, et al. Thread-Level Locking for SIMT Architectures. IEEE Trans Parall Distr Syst, 2020, 31: 1121–1136

• Gao L, Xu Y L, Wang R, et al. Accelerating in-memory transaction processing using general purpose graphics processing units. Future Gener Comput Syst, 2019, 97: 836–848

• Wang H, Wang R, Luan Z Z, et al. Improving multiprocessor performance with fine-grain coherence bypass. Sci China Inf Sci, 2015, 58: 012104

• Xu Y L, Wang R, Goswami N, et al. Software transactional memory for GPU architectures. In: Proceedings of International Symposium on Code Generation and Optimization, Orlando, 2014. 1–10

• Qian X H, Sahelices B, Qian D P. Pacifier: record and replay for relaxed-consistency multiprocessors with distributed directory protocol. In: Proceedings of International Symposium on Computer Architecture, Minneapolis, 2014. 433–444

• Qian X H, Torrellas J, Sahelices B, et al. Volition: scalable and precise sequential consistency violation detection. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems, Houston, 2013. 535–548

• Qian X H, Huang H, Sahelices B, et al. Rainbow: efficient memory dependence recording with high replay par-

allelism for relaxed memory model. In: Proceedings of International Symposium on High Performance Computer Architecture, Shenzhen, 2013. 554–565

- Qian X H, Torrellas J, Sahelices B, et al. BulkCommit: scalable and fast commit of atomic blocks in a lazy multiprocessor environment. In: Proceedings of IEEE/ACM International Symposium on Microarchitecture, Davis, 2013. 371–382