# Leonid: Exploring Automated Kernel Fusion in Performance-Portable Programming Models for Scientific Computation

Chenchen Zhang
Peking University
Beijing, China
zhangchenchen@stu.pku.edu.cn

Hao Luo
Peking University
Beijing, China
lhsms@pku.edu.cn

Chao Yang*
Peking University
Beijing, China
PKU-Changsha Institute of Computing and Digital Economy
Changsha, China
chao_yang@pku.edu.cn

## Abstract

With advances in hardware performance, architectural divergence and the growing gap between computational power and memory bandwidth have become increasingly pronounced. Existing performance-portable models address hardware divergence but lack automated kernel fusion to optimize memory-bound scientific applications. To address this issue, we propose Leonid, a performance-portable programming model designed to support automated kernel fusion. Leonid integrates separate modules for unified global and scratchpad memory management, and for unified parallel and serial execution patterns, both specifically tailored for automated kernel fusion, alongside an integrated automated kernel fusion module. These components ensure the compatibility across CPUs, GPUs, and Sunway platforms for automated kernel fusion. Performance evaluations demonstrate that Leonid achieves up to 1.52× speedup (averaging 1.19×) over manually implemented code, outperforms Kokkos and RAJA, and matches the efficiency of manually fused code in bandwidth-limited algorithms and applications. For bandwidth-limited and fusion-eligible code, Leonid offers a significant advantage over other performance-portable models that lack automated kernel fusion capabilities.

*Corresponding Author.


*ICS '25, Salt Lake City, UT, USA*

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3728489

## CCS Concepts

• **Computing methodologies** → **Parallel computing methodologies**.

## Keywords

Performance Portability, Programming Model, Automated Kernel Fusion, Parallel Computing, Memory Management

**ACM Reference Format:**
Chenchen Zhang, Hao Luo, and Chao Yang. 2025. Leonid: Exploring Automated Kernel Fusion in Performance-Portable Programming Models for Scientific Computation. 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/10.1145/3721145.3728489

## 1 Introduction

The enhancement of hardware has greatly driven the development of scientific computing, accompanied by the increasing diversity of hardware architectures. In addition to traditional multicore CPUs, GPUs have emerged as a dominant source of computational power. For instance, 9 out of the top 10 supercomputers in the current Top 500 rankings utilize hybrid CPU-GPU architectures [5]. Furthermore, significant progress is being made with high-performance processors such as ARM-based designs, Sunway (SW) processors in the Sunway supercomputer [56], MT3000 chips in Tianhe [31], and Google's TPU chips [36], among others.

Additionally, the gap between computational power and memory bandwidth is gradually widening. For instance, the NVIDIA H100 SXM5 GPU offers 34 teraflops of FP64 computational capacity but is constrained to a memory bandwidth of only 3 TB/s [14], resulting in a ratio of memory bandwidth to computational capacity of just 0.1 bytes per FLOP. This disparity is even more pronounced in the SW26010-Pro

manycore processor used in the latest generation of the Sunway supercomputer, where the ratio falls to approximately 0.004 [56].

These two trends pose significant challenges for scientific and engineering computing: ① First, hardware diversity forces developers to implement platform-specific codes using device-specific models (e.g., CUDA [2] for NVIDIA GPUs and HIP [1] for AMD GPUs) to ensure portability across accelerators. This incurs significant costs in development, verification, and maintenance due to the extensive nature of scientific and engineering codes. ② Second, the increasing gap between computational power and memory bandwidth prevents memory-bound applications from fully exploiting hardware performance. Most scientific applications exhibit low arithmetic intensity, with performance primarily constrained by memory bandwidth [41].

Performance-portable programming models are designed to address the challenge of hardware divergence across platforms for scientific and engineering applications. Among them, Kokkos [17], RAJA [9], OpenCL [35] and SYCL [27] are widely utilized in the field of scientific computing. They provide commonly used parallel patterns, support mainstream hardware, and offer additional performance features, making them a popular choice in many works [34, 38, 52, 54].

Reducing memory access to DRAM is crucial for improving the performance of scientific codes bound by memory bandwidth, and kernel fusion is the primary method to achieve this goal [46]. Several linear algebra libraries, such as Eigen [19], LIBXSMM [22], and DESOLA [43], adopt kernel fusion to optimize their codes. Automated kernel fusion, compared to manual fusion, is especially valuable in real-world scientific and engineering computations and is urgently needed. It automatically detects fusion opportunities and fuses kernels, thereby reducing development costs and improving maintainability. Nowadays, automated kernel fusion is widely adopted in machine learning frameworks such as TVM [13], TensorFlow [6], and ONNX [3], as well as in specific applications like GraphBlas [32].

However, there is currently no performance-portable programming model that supports automated kernel fusion for scientific computing, including Kokkos and RAJA. As a result, these models fail to provide automated fusion optimizations for memory-bound applications, exacerbating the challenge of bridging the gap between memory bandwidth and computational power in scientific and engineering applications.

To address the aforementioned challenges, this paper aims to explore feasible solutions for **automated kernel fusion within performance-portable programming models**. In particular, we introduce Leonid, a light-weight performance-portable programming model for scientific computing. Leonid draws on the core features of Kokkos and RAJA but is specifically tailored for automated kernel fusion, with the goal of investigating the possibility of integrating automated kernel fusion into models like Kokkos and RAJA in the future. In addition to supporting CPUs and GPUs, we deliberately include support for the SW platform due to its significant architectural differences from CPUs and GPUs, particularly its support for direct memory access (DMA), which introduces unique memory management challenges for automated kernel fusion that do not exist on CPUs and GPUs. To the best of our knowledge, this is the first attempt to integrate automated kernel fusion into a Kokkos-like performance-portable model and the primary contributions of this paper are as follows:

(1) We develop a memory-management module for global and scratchpad memory across platforms. The former provides memory trace information for automated kernel fusion, while the latter enables DMA and is fusion-friendly.
(2) We introduce unified programming patterns for latency execution across various hardware platforms, which abstract hardware-specific details to hide implementation complexities, thereby eliminating the need for device-specific programming models.
(3) We integrate an automated kernel-fusion mechanism into Leonid which automatically constructs the data dependency graph (DDG) and fuses dependent kernels while ensuring correctness. This reduces redundant global memory accesses, thereby improving performance.

Performance evaluations are conducted at both the algorithm and application levels. The experimental results demonstrate that Leonid can achieve up to a 1.52× speedup (with an average speedup of 1.19×) across multiple platforms for various algorithms and applications, outperforming Kokkos and RAJA and highlighting the importance of automated kernel fusion.

The remainder of this paper is organized as follows. In Section 2, we introduce the challenges of automated kernel fusion in performance-portable programming models, followed by a brief discussion of the design philosophy of Leonid, highlighting how our approach facilitates automated kernel fusion. Section 3 provides a in-depth introduction to the implementation details of Leonid. In Section 4, we present the evaluation results, comparing the performance of Leonid with Kokkos and RAJA at both the algorithm and application levels. Related work is discussed in Section 5, and conclusions are drawn in Section 6.

## 2 Challenges and Design Philosophy

### 2.1 Challenges for automated kernel-fusion

Integrating automated kernel fusion into performance-portable programming models presents several key challenges:

①**Redundant Memory Access Due to Scratchpad Memory**: Scratchpad memory is a small, fast, software-managed memory space used for temporary data storage, such as shared memory in GPUs. Unlike cache memory, it lacks caching behavior, requiring explicit manual management for data loading and storing. The absence of a cache mechanism in scratchpad memory necessitates repeated accesses to main/global memory when fused kernels attempt to retrieve the same data through scratchpad memory.

②**Lack of Kernel Fusion Support in Execution Patterns**: Existing programming models lack the data flow information of kernels required to guide automated fusion, hindering the ability to perform kernel fusion automatically.

③**High Fusion Overhead on GPUs**: GPUs feature a large number of threads, each typically processing only a small number of elements. Runtime kernel fusion generally relies on function pointers, and the overhead from invoking function pointers can become substantial, negating the performance benefits of fusion.

## 2.2 Design Philosophy

We compare the design philosophies of Kokkos and RAJA with Leonid to highlight how our framework is specifically tailored to overcome the challenges of automated kernel fusion.

In Kokkos and RAJA, kernels are executed immediately after being launched. While this approach simplifies the execution flow and ensures predictability, it cannot provide runtime information about the data flow between kernels and prevents Kokkos and RAJA from performing automatic kernel fusion, unless fusion operation is manually implemented. To enable automated kernel fusion, Leonid provides a **latency-execution** model. Under this model, kernels are not executed immediately when encountered. Instead, the kernel is stored in an executable format with its metadata—such as input arguments, loop ranges, and tiling sizes—being collected. A DDG is then constructed from this metadata, and a greedy algorithm is applied to automatically fuse kernels while maintaining the correct execution order. To address the challenge of high fusion overhead on GPU, we implemented a static DDG and a static fusion method specifically for GPU.

Furthermore, while Kokkos and RAJA provide memory management mechanisms (e.g., Kokkos's *View* class), Leonid's corresponding module not only replicates this functionality but also aids kernel fusion by providing memory-access-range references. These references assist in deducing memory access ranges for kernel fusion.

Scratchpad memory support is another area where Leonid improves upon Kokkos and RAJA. While Kokkos provides scratchpad memory support, its default implementation does not effectively minimize unnecessary data transfers between scratchpad and main/global memory, especially in DMA operations on the SW platform. Leonid addresses this limitation by integrating cache-like scratchpad-memory management with delayed-write support, which reduces unnecessary data transfers and enhances performance.

## 2.3 Hardware Abstraction

Based on the design philosophy, in Leonid we introduce a unified hardware abstraction layer shown in Figure 1. This layer standardizes interactions across diverse processors, facilitating portability and enabling kernel fusion.

Inspired by Kokkos, Leonid organizes devices into memory and execution spaces, focusing particularly on managing scratchpad memory. Threads operate within the execution space, while data resides in the memory space. The execution space encompasses CPU cores, streaming processors (SP) on GPUs, and compute processing elements (CPEs) on SW processors. The memory space in Leonid is divided into two levels: main/global memory and scratchpad memory. On GPUs, shared memory serves as scratchpad memory, while SW processors use local device memory (LDM). Since CPUs lack dedicated scratchpad memory, a portion of their cache is repurposed for this role, maintaining a unified abstraction despite differences in cache behavior. This unified abstraction ensures seamless interaction across hardware architectures, allowing us to tailor scratchpad memory operations for kernel fusion.

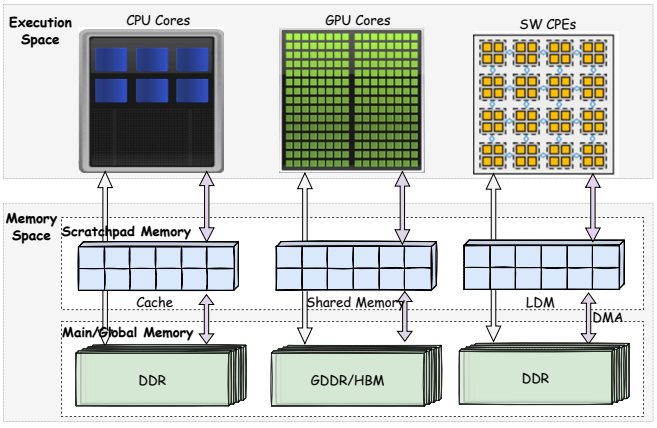


**Figure 1: Hardware abstraction of Leonid.**

# 3 Leonid Implementation

In this section, we introduce the framework of Leonid, a C++ performance-portable programming model designed to support automated kernel fusion. As illustrated in Figure 2,

Leonid addresses the aforementioned challenges through several key components: ① Memory Management: A dedicated memory-management module handles main/global memory across processors, complemented by a sub-module for unified scratchpad memory management, as detailed in Section 3.1. ② Execution Patterns: Unified execution models are implemented for key parallel patterns, including *Parallel For* and *Parallel Reduce*, along with support for serial execution and hardware-specific optimized kernels. These models enable latency execution and serve as kernels to facilitate the construction of DDG, directly addressing the lack of kernel fusion support in execution models. Further details are provided in Section 3.2. ③ Automated Kernel Fusion: Leonid integrates an automated kernel-fusion mechanism within the programming model. This method automatically builds a DDG and performs kernel fusion, supporting both horizontal and vertical fusion strategies. Further insights are provided in Section 3.3. ④ Execution backends: To ensure hardware compatibility, Leonid leverages OpenMP [37], CUDA [2], and SWUC [12] as execution backends for CPU, GPU, and SW processors, respectively.

## 3.1 Memory Management

*3.1.1* ***Main/Global Memory Management.*** In Leonid, we introduce a data structure named *Tensor* to manage multidimensional arrays, which functions similarly to *View* in Kokkos [17]. Additionally, *Tensor* provides allocation information to aid in deducing memory access ranges, which are used during the automated kernel fusion phase.

As illustrated in Figure 2(a), the main memory space is divided into host memory space and device memory space, akin to OpenCL [35]. The *Tensor* structure contains two pointers: one pointing to the host memory and the other to the device memory. In both CPU and SW environments, the host memory and device memory actually refer to the same memory region, resulting in identical pointers for both. However, in a GPU context, the host memory pointer refers to the main memory, while the device memory pointer points to the GPU's global memory. The memory layout provided is *LayoutRight* (i.e., C-style layout). Since our parallel patterns support multi-dimensional parallelism, *LayoutRight* is sufficient to achieve memory coalescing on the GPU.

We provide APIs to control data read and write operations, as well as transfer operations between host memory space and device memory space. Additionally, both *Copy* and *DeepCopy* operations are supported. A constant assumed size can serve as the template argument for the *Tensor*, providing additional information during compile time when our static kernel fusion method is applied.

To facilitate memory operations, we also offer *View* and *DeviceView* classes. A *View* of one *Tensor* shares the same memory space as the tensor but is restricted to read and write operations, and it cannot allocate or release the associated memory. On the other hand, *DeviceView* manages the device memory space of the tensor, allowing for operations on device memory through it. A specific example is provided in Listing 1.

```
1 /*set the device used*/
2 using Device=DEVICE::GPU;
3 /*Template parameters: data type, dimension,device,Is
      view, assumed size*/
4 Tensor<int,2,Device,false,{1024,1024}> Array({L1,L2});
5 Array.DataSynchHost2Device();
6 auto device_view = Array.GetDeviceView();
7 auto view = Array.CreateView();
```

**Listing 1: A case of Tensor operation**

*3.1.2* ***Scratchpad Memory Management.*** Both shared memory in GPU and LDM in SW serve as scratchpad memory, offering low latency and high bandwidth but with limited capacity. However, there are notable differences in their utilization. On GPU, shared memory is typically employed as temporary storage to promote data reuse and reduce unnecessary memory accesses. In contrast, on the SW platform, the use of LDM not only reduces memory accesses but also aims to enhance bandwidth. The DMA supported by SW facilitates high-bandwidth data transfers between main memory and LDM. If data is not accessed using DMA, the memory bandwidth is approximately halved compared to when DMA is employed. Therefore, by managing LDM, Leonid's support for DMA benefits memory access optimization in applications running on the SW system. Additionally, there is no scratchpad memory in the CPU, and we allocate an array in main memory with a size less than the cache capacity to enable unified management across different devices.

Low overhead is crucial for effective scratchpad memory management. To minimize overhead, we propose a lightweight memory pool with a limited number of blocks, with each bound to a unique global memory address. When a scratchpad memory block is requested, if a block is already bound to the corresponding global memory address, its associated scratchpad memory address is returned directly, and no new block will be allocated. In the GPU context, all threads within a single block share the same scratchpad memory management instance, while in the CPU and SW environments, each thread has its own dedicated instance. Furthermore, these objects are declared statically to avoid overhead introduced by construct and destruct operations. Read and write operations on SW are implemented through DMA.

Another goal of scratchpad-memory management is to minimize repeated access to global memory while achieving high bandwidth on SW and GPUs. As mentioned earlier, our

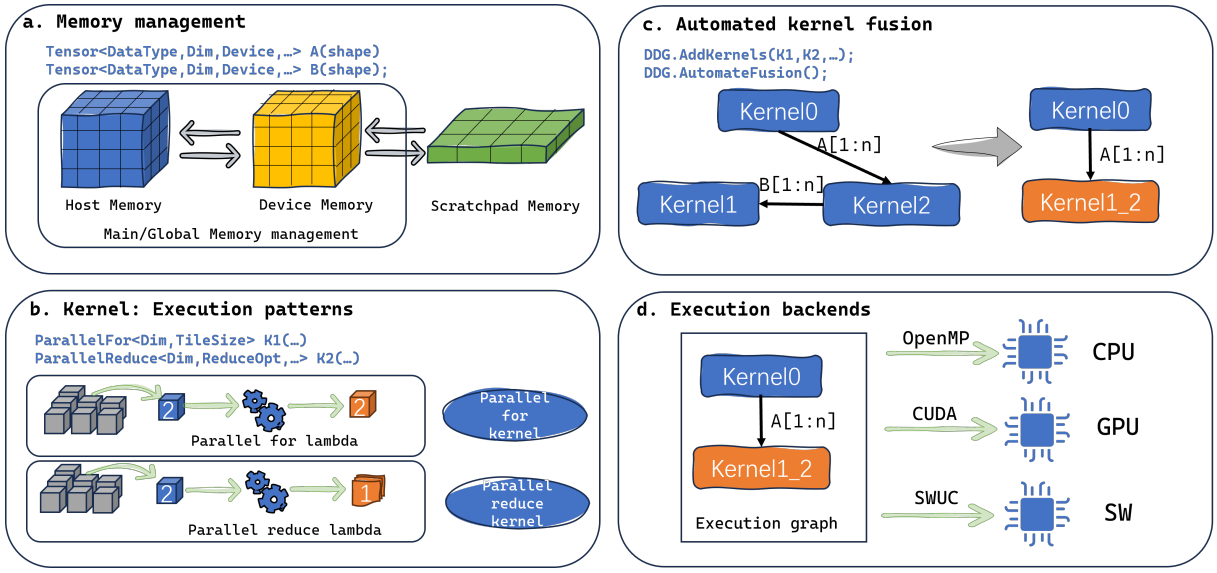


**Figure 2: System overview of Leonid**

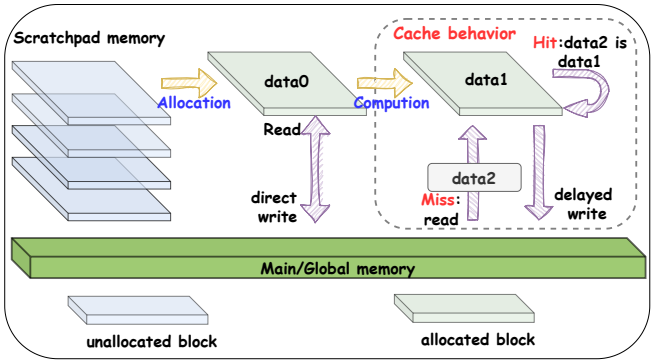


**Figure 3: Scratchpad memory allocation, read and write operations diagram.**

Leonid framework supports automated kernel fusion, which can potentially lead to repeated memory accesses when fused kernels operate on the same data through scratchpad memory. On the SW, this occurs because data loaded into LDM bypasses the cache. On GPU architectures, although data loaded into shared memory passes through the cache, the limited cache capacity often leads to the eviction of the corresponding data from the cache. To address this issue, we manage scratchpad memory in a cache-like manner. The read operation will access global memory only when new data needs to be fetched; otherwise, it will return directly. For the write operation, our scratchpad memory management supports both direct and delayed write. With delayed-write, data is written back only when new data is read in, thereby minimizing unnecessary write and read operations. This process is illustrated in Figure 3. Additionally, to minimize overhead, our replacement strategy ensures that the data in each memory block is only replaced with data from the same array. The number of supported blocks can be set prior to use. On both the GPU and SW platforms, users need to configure the size of the scratchpad memory and set the parameters according to their specific configurations.

## 3.2 Kernels: Execution Patterns

Leonid currently supports two general parallel patterns—the *Parallel For* pattern and the *Parallel Reduce* pattern—as well as one additional pattern for serial execution codes and hardware-specific optimization codes: the *Statement* pattern.

***Parallel For***: An example of vector addition operation using *ParalleFor* is provided in Listing 2. When utilizing the *Parallel For* pattern, the computation for each iteration is performed in parallel, and the execution order is not guaranteed. To ensure correct results, programmers must verify that there is no dependency between iterations. Tiling operations are supported on both CPU and SW platforms. Kernel fusion optimization relies on runtime information for both platforms, with kernels stored as function pointers to standardize their form. Given the limited number of threads on CPU and SW, tiling is an effective optimization method to mitigate the overhead associated with function pointer calls. The tiling operation is applied to the innermost loop to promote continuous memory access and enable the compilers' automated vectorization optimization. On GPUs, the tiling size is fixed at one and cannot be altered, as GPUs have a massive number of threads and each thread usually handles a few elements in the arrays. Under these conditions, runtime

fusion introduces significant overhead on GPUs. Therefore, a static method is employed, which will be discussed in Section 3.3.

***Parallel Reduce***: Similar to the *Parallel For* pattern, the execution order in *Parallel Reduce* is also not guaranteed, meaning that the reduced result should be independent of execution order. An example of vector dot production is provided in Listing 2. Tiling is supported in this pattern as well, mirroring its implementation in the *Parallel For* pattern.

In our *Parallel Reduce* pattern, users can provide their own reduction operation. This reduction operation class should include an initialization function to assign an initial value to the reduced data, as well as a reduction function that defines the specific reduction behavior. Additionally, for GPUs, users are required to provide a *ReduceToGlobal* function to optimize performance. Given the large number of threads on GPUs, reduction of results directly using atomic operations would lead to significant performance degradation. An optimized approach involves first performing intra-warp reduction using warp shuffle, followed by inter-warp reduction using shared memory, and finally reducing the results across different blocks using atomic operations.

To avoid data races, each thread keeps a private reduced parameter, which is combined into the final result after all threads complete their computations.

```
/*define the reduce operation*/
template<class T>
class SUM{
  HOST_DEVICE T InitialValue(){return static_cast<T>(0)
      ;}
  HOST_DEVICE T operator()(T a, T b){return a+b;};
  #ifdef __CUDACC__
  DEVICE void ReduceToGlobal(T* fininal_data,T a){
    /*reduce to global memory*/
  }
  #endif
}
constexpr size_t tile_size=1024, Dim=1;
/*vector addition: A and B are inputs, and C is the
    output*/
ParallelFor<Dim,tile_size> kernel0(ARG(A,B),ARG(C),
    RANGE(start_l,end_l));
kernel0.MakeLam([](auto i, auto A, auto B, auto C){C(i)
    =A(i)+B(i);}, A, B, C);
/*dot product: A and B are inputs, and sum is the
    output*/
ParallelReduce<Dim,int,SUM<int>,tile_size> kernel1(ARG(
    A,B),ARG(sum),RANGE(start_l,end_l));
kernel1.MakeLam([](auto i,auto& sum,auto A,auto B){sum
    +=A(i)*B(i)},sum,A,B);
```

**Listing 2: An example of performing vector addition and dot production using the *Parallel For* and *Parallel Reduce* patterns.**

***Statement***: The *Statement* pattern does not inherently have parallel semantics. However, users can define parallel operations within it to optimize some algorithms on specific hardware architectures, such as SpMV in CSR format on GPUs. Additionally, the *Statement* pattern can also be used to define serial or Message Passing Interface (MPI) operations. An example is shown in Listing 3.

```
/*A and B are scalar inputs, and alpha is the output*/
Statament kernel(ARG(A,B),ARG(alpha));
kernel.MakeKFLam([](auto& alpha,auto A, auto B){
  MPI_Allreduce(MPI_IN_PLACE,&A,1,MPI_DOUBLE,MPI_SUM,
      MPI_COMM_WORLD);
  MPI_Allreduce(MPI_IN_PLACE,&B,1,MPI_DOUBLE,MPI_SUM,
      MPI_COMM_WORLD);
  alpha=A/B
},alpha,A,B);
```

**Listing 3: A example of *Statement* pattern**

***Parallel For* and *Parallel Reduce* using scratchpad memory**: When scratchpad memory is utilized, tiling operation is done by the user. An example of a vector copy case is shown in Listing 4. Each time a kernel is called, the scratchpad memory module is initialized. The instance of this module can be directly obtained through related API.

```
constexpr size_t tile_size=1024;
constexpr size_t Dim=1;
/*vector copy: A  is input, and B is the output*/
ParallelFor<Dim,tile_size> kernel(ARG(A),ARG(B),RANGE(
    start_l,end_l));
kernel.MakeLams(
[tile_size](auto i, auto& A, auto& B){
  auto& sm=MakeInstance();
  auto A_buf=sm.Malloc(A.GetPtr(),tile_size);
  auto B_buf=sm.Malloc(B.GetPtr(),tile_size);
  sm.CacheReadFromGlobal(A_buf,0,A,i,tile_size,i);
  for(int ii=0;ii<tile_size;ii++)
  B_buf[ii]=A_buf[ii];
  sm.CacheStoreToGlobal(B,i,B_buf,0,tile_size);
},A,B);
```

**Listing 4: An example of vector copy using *ParallelFor* with scratchpad memory.**

***Other parallel patterns***: Most other parallel patterns used in scientific computing can be implemented using *Parallel For* pattern. For instance, Leonid additionally provides the *Parallel Scan* operation, which is implemented through *Parallel For* plus Hillis-Steele algorithm[24] and Blelloch algorithm[11]. For CPU and SW, we use the Hillis-Steele method for small vectors and switch to the Blelloch approach for large ones. Specifically, on GPUs, we exclusively employ the Blelloch method to minimize synchronization overhead. Both inclusive-scan and exclusive-scan are supported. The exclusive-scan mode is used in the Molecular Dynamics application in Section 4.

## 3.3 Automated Kernel Fusion

After the declaration of kernels, they are added to a DDG. It is crucial that kernels are added in a correct execution order, as the final execution graph will be generated based on this sequence. Interval analysis is employed to deduce the memory access range for each array parameter based on the context of the lambda expression and the Bounded Regular Section (BRS) method [21] is used to store this range.

A BRS consists of lower and upper bounds, and a stride, clearly defining the access range and proving sufficient for most scientific computing applications. If the BRS cannot be deduced, the allocation information provided by memory-management module will be used to generate a reference BRS.

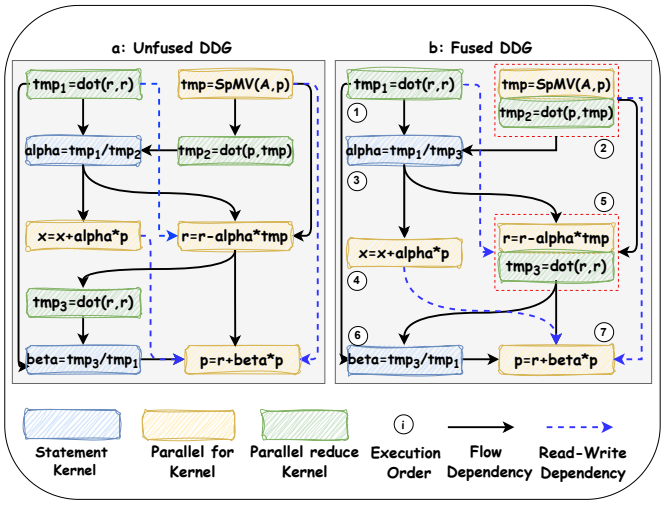


**Figure 4: The DDG before and after automated kernel fusion for CG algorithm.**

The DDG is a directed acyclic graph that describes the data dependencies between kernels, with nodes representing the kernels and edges representing the set of variables that cause dependencies. In Leonid, both runtime and static versions of DDG are implemented. The runtime DDG uses memory addresses to differentiate between variables. By employing the BRS along with memory addresses, we can ascertain the data access range in memory. If there is an overlap between the memory accessed by the output parameters of a pre-kernel and the input parameters of a post-kernel, we consider that a dependency exists. The order in which kernels are added to the DDG determines whether they are classified as pre-kernels or post-kernels. In contrast, the static version of the DDG is constructed at compile time and lacks runtime information; thus, it relies on variable analysis to establish dependencies. In addition, implicit read-write dependency is also considered for both kinds of DDG. An example of the Conjugate Gradient (CG) algorithm is shown in Figure 4(a).

Each kernel maintains two sets: one to indicate which kernels have already been fused with the current kernel (named *FusedSet*) and another to declare which kernels cannot be fused with the current kernel (named *AvoidSet*). They are used to assist in determining whether two kernels can be fused. During the construction of the DDG, each kernel includes itself in its own *FusedSet*. For *Parallel Reduce* kernels, their child nodes are added to their *AvoidSet*.

Both horizontal fusion and vertical fusion are implemented in Leonid and are determined based on the DDG. The prerequisites for horizontal fusion and vertical fusion are different, as shown in Figure 5 and outlined below:

***Horizontal Fusion***:

(1) The two kernels share at least one common parent node and have the same dependent vector variables with the parent node. Furthermore, these variables should exhibit sufficiently large overlapping memory access ranges.
(2) Neither kernel is a *Statement* kernel, and neither is included in the other's *AvoidSet*.
(3) There is no path connecting these two kernels in DDG.
(4) The fusion operation does not alter read-write dependencies.

***Vertical Fusion***:

(1) The two kernels form a producer-consumer relationship with dependent variable of vector type whose length exceeding a specified threshold.
(2) Neither kernel is a *Statement* kernel, and the child node is not included in the parent's *AvoidSet*.
(3) There is no other paths connecting the two kernels in the DDG except for the directly connected one.
(4) The fusion operation does not alter read-write dependencies.
(5) The difference between the index values of the variable does not exceed the threshold.

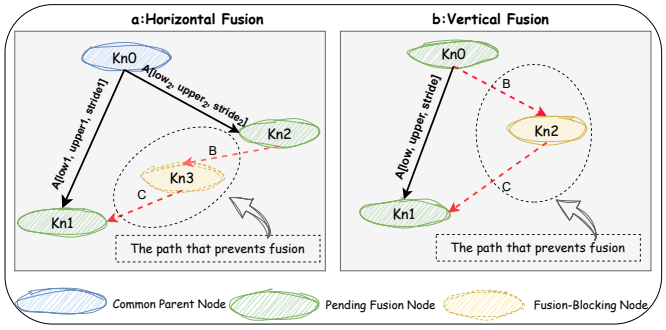


**Figure 5: Diagram illustrating the conditions for horizontal and vertical fusion.**

Leonid employs a greedy algorithm for automatic kernel fusion, aiming to fuse as many compatible kernels as possible while taking into account the limitations of cache and LDM capacity. When two kernels can be successfully fused, the *FusedSet* and *AvoidSet* will be updated using Equation 1 and 2. Figure 4(b) illustrates the fused DDG of CG algorithm after the automated kernel fusion, serving as an example.

$$FusedSet_{new} = FusedSet_0 \cup FusedSet_1 \quad (1)$$

$$AvoidSet_{new} = AvoidSet_0 \cup AvoidSet_1 \quad (2)$$

Runtime automated kernel fusion is implemented on CPU and SW platforms. In our implementation framework, kernels after tiling process are transformed into function pointers. When kernels are fused, Leonid generates a new composite function pointer that orchestrates the sequential execution of the corresponding function pointers associated with the original kernels.

A static approach is adopted for GPU to minimize the overhead associated with function pointer calls. Although tiling can be utilized, it will reduce the granularity of parallelism. Furthermore, due to the large number of threads on GPUs, the tiling method might lead to insufficient cache availability for all threads to benefit from data reuse.

Our static kernel fusion method is performed at compile time using the static DDG. The entire process of the static method is shown in Figure 6. In static kernel fusion, the types (including estimated vector size information in the *Tensor* type) and names of variables are used to construct the static DDG, with the automated kernel fusion process mirroring that of runtime fusion. Once kernels in the DDG are fused, an index sequence is generated to indicate which kernels will be combined. Kernel fusion is then performed based on this sequence, and the new kernels are used to build the execution graph. All of these steps occur at compile time, ensuring no additional runtime overhead.

Moreover, with the compiler's inline optimization, there are no concerns regarding optimization barriers for other compiler optimizations.

## 3.4 Backend Execution

After the automated kernel fusion, the DDG is used to generate an execution graph, which determines the correct execution order for launching the kernels. On the GPU, CUDA graphs are used to store the GPU kernels. These execution graphs can be reused multiple times, eliminating the overhead of building the DDG and performing kernel fusion operations at runtime.

OpenMP serves as the backend on CPU, parallelizing the outermost dimension of the loop. SWUC is utilized on SW, where the loop after tiling is fused into a one-dimensional loop, which can be computed in parallel by 64 CPEs. For GPU, CUDA is employed, supporting multi-dimensional parallelism with each thread responsible for calculating one of the iterations.

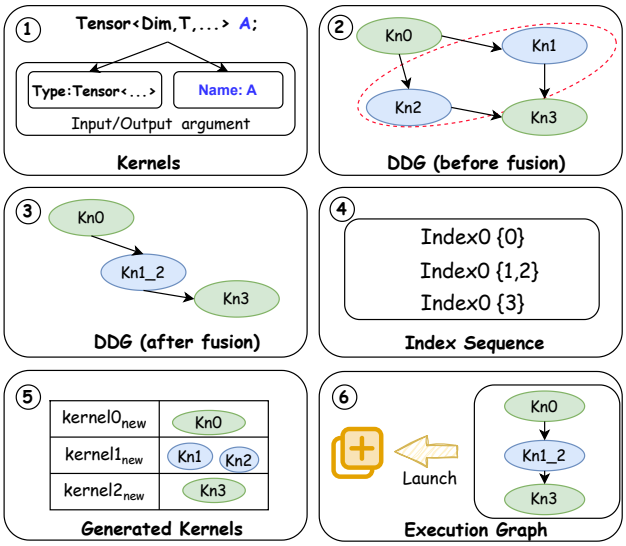


**Figure 6: Diagram of static automated kernel fusion for GPU.**

# 4 Performance Evaluation

In this section, we present a performance evaluation of Leonid on CPU, GPU, and SW26010-Pro processors, aiming to demonstrate its effectiveness and applicability, as well as quantify the benefits provided by its automated kernel fusion capabilities. We compare the performance of Leonid with manually implemented code, as well as implementations developed using Kokkos and RAJA, both at the algorithm level and the application level.

## 4.1 Experiment Setup

**Linear Solvers and Applications:** At the algorithmic level, we compare the performance of Leonid, Kokkos, and RAJA on several methods: Two-level V-cycle geometric Multi-Grid (**MG**) with Jacobi as the smoother, Preconditioned Conjugate Gradient (**PCG**) with Jacobi preconditioning, Quasi-Minimal Residual (**QMR**), and Minimum Residual (**MINRES**). These four algorithms are widely used in scientific and engineering computing, and we apply them to the 2-D **Diffusion Equation** using a five-point star stencil pattern. We implemented Sparse Matrix Vector Multiplication (SpMV) in a matrix-free manner, facilitating the calculation of the theoretical maximum speedup after kernel fusion.

At the application level, four mini-applications, the **HPCG** benchmark [23], the lid-driven cavity problem (**Cavity**) [44], Lattice Boltzmann Method (**LBM**) [45] and Molecular Dynamics (**MD**) [8] are employed to demonstrate the applicability of Leonid. HPCG solves a sparse linear system from the Poisson equation on a 3D grid using MGPCG with a 2-level

**Table 1: Hardware Platform and Corresponding Compilers for Performance Evaluation**

| Platform | Specification | Memory | Compiler |
|---|---|---|---|
| CPU | Intel Xeon ICX Platinum 8358 (32c) | 16 × TruDDR4 (32G) | gcc 14.2.0 |
| GPU | NVIDIA Tesla V100 PCIe | HBM2 (32G) | Clang 18.1.8; CUDA 12.4.99 |
| SW | SW26010-Pro(6 clusters) | 6 × DDR4(16G) | swgcc 7.1.0 |

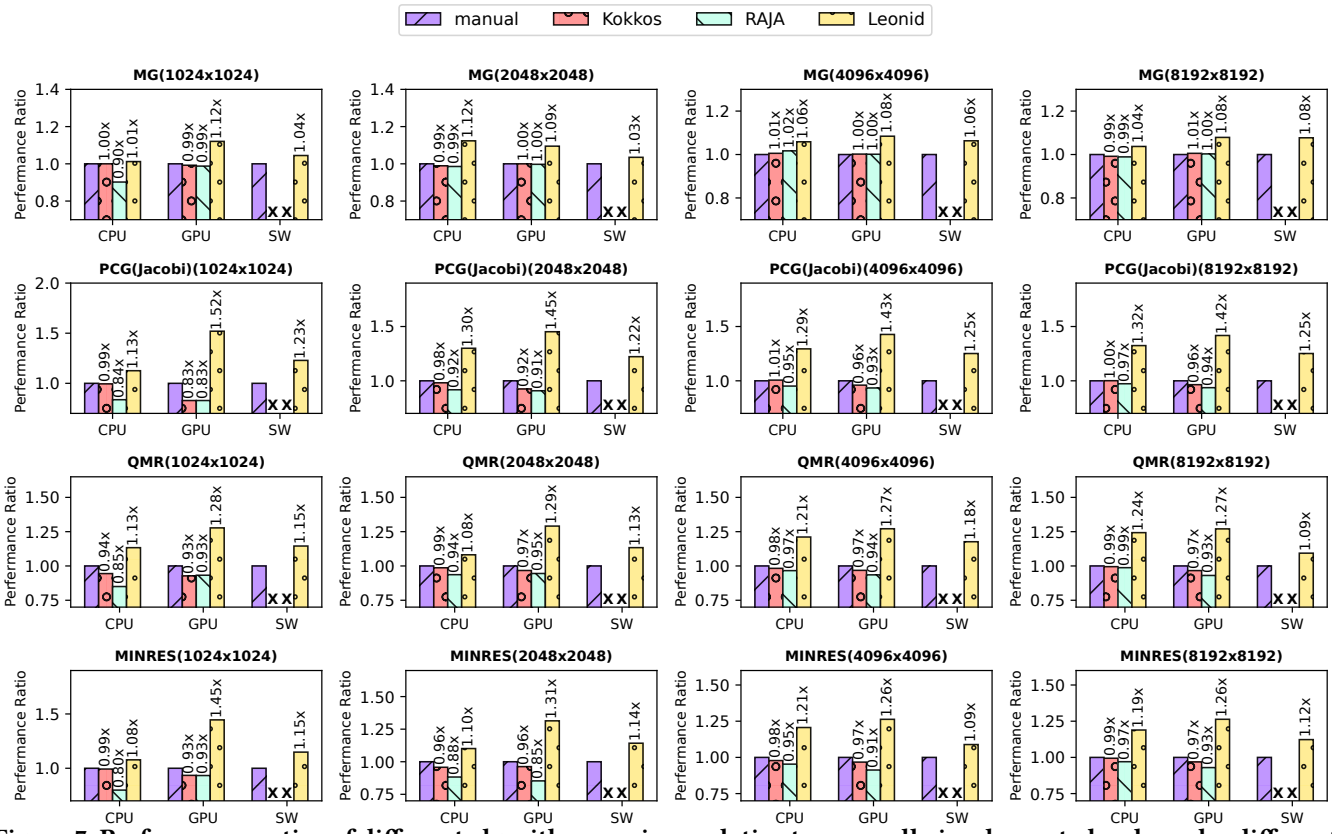


**Figure 7: Performance ratios of different algorithm versions relative to manually implemented code under different platforms and grid sizes ("×" indicates that Kokkos and RAJA don't support SW).**

V-cycle multigrid preconditioner and Jacobi smoother. The cavity, governed by the Navier–Stokes equations on a 2D grid, is discretized using a 2nd-order finite difference scheme, and the resulting linear system is solved using the biconjugate gradient stabilized method. The SpMV operations in both application are performed in a matrix-free manner. In the LBM application, we adopt the Bhatnagar-Gross-Krook (BGK) collision model [10] in a 2D box with a D2Q9 lattice for incompressible flow simulation. We use Verlet integration [49] for motion updates and a truncated Lennard-Jones potential [47] for energy calculations in the MD case. A neighbor list with a controlled cutoff radius ensures that each particle interacts with 2–3 others, with random initialization within the 2D box.

**Platforms:** Table 1 lists the specific hardware platform information and the compilers used. The CPU used in this section has an L3 cache of 48 MB per processor. Each Streaming Multiprocessor on NVIDIA V100 GPU has a configurable 128 KB combined L1 cache and shared memory, which we set to 64KB for L1 cache and 64 KB for shared memory during testing. The SW26010-Pro processor includes a scratchpad memory of 256 KB, of which we configured 128 KB as cache.

## 4.2 Evaluation on Algorithm Level

**Performance comparison with state-of-the-art methods:** Figure 7 shows the performance ratios of different algorithm versions compared to the manually implemented code across various grid sizes. On both CPU and GPU, we evaluate Kokkos, RAJA, and Leonid against the manual code. On the SW platform, only Leonid is compared, as Kokkos and RAJA do not currently support it.

Figure 7 demonstrates that Leonid outperforms both Kokkos and RAJA across all test cases. Overall, Leonid achieves a maximum speedup of 1.52× and an average speedup of 1.19× compared to the manually implemented code. This performance gain is primarily attributed to kernel fusion, which reduces redundant accesses to main or global memory. While Kokkos and RAJA can deliver performance comparable to the manually implemented code, they do not match the level of improvement achieved by Leonid.

Among all the profiled algorithms, PCG achieves the highest acceleration ratio, with a maximum speedup of 1.53x and an average speedup of 1.45×. In contrast, MG only achieves a maximum speedup of 1.12×. There are two main reasons for this performance difference: (1) A higher proportion of kernels in the PCG algorithm can be fused. In PCG, 11 parallel kernels are fused into 5, whereas in MG, only 7 parallel kernels are fused into 5. (2) In the MG algorithm, some computations are performed on the coarse grid level, where the kernel execution time is short. Compared to kernels running on the fine grid, the benefits of kernel fusion on the coarse grid are relatively limited.

When the grid sizes are $4096 \times 4096$ and $8192 \times 8192$, the acceleration ratio tends to stabilize. However, the acceleration ratio of kernel fusion on the SW platform is lower than that on the GPU and CPU, except for MG with a grid size of $8192 \times 8192$. This performance difference can be attributed to the fact that the SW platform has only one level of cache. On CPUs, Leonid can take advantage of the L1 cache, which provides a bandwidth of approximately 64 GB/s per core. On the GPU platforms, our static kernel fusion method enables register-level reuse. In contrast, a stream test on the SW platform indicates a much lower bandwidth value of only 6.75 GB/s.

**Overhead profile of Leonid:** We use the PCG and MINRES algorithms as examples to study the overhead of Leonid across different grid sizes and platforms. The result is illustrated in Figure 8. We compare the performance of Leonid with the manually implemented codes, both with and without kernel fusion. The manual fusion code's fusion patterns are based on the DDG generated by Leonid's automated kernel-fusion mechanism.

Overall, on all evaluation platforms, Leonid achieves performance comparable to the manually implemented code, regardless of whether kernel fusion is applied, except when the grid size is $1024 \times 1024$ on the CPU.

When the grid size is $1024 \times 1024$, a performance loss can be observed on CPU for Leonid with automated kernel fusion. This because Leonid can only use cache to implement data reuse, and the function pointer calls incur overhead. In contrast, manually fused kernel code can achieve register reuse. When the grid size is small, the overall computational workload and memory bandwidth pressure are low, making the overhead in Leonid more noticeable. In addition, the large L3 cache in the CPU may cause the speedup lower when the grid size is small. For example, when the grid sizes are $1024 \times 1024$ and $2048 \times 2048$, the vector sizes are 8M and 32M, respectively, which fit within the capacity of L3 cache. Under these conditions, the data reuse of certain arrays in the unfused code partially diminishes the performance benefits of kernel fusion.

On the GPU platform, when the grid size is $1024 \times 1024$, Leonid can outperform the manually implemented code by about 7% to 20%. This is because, the computational workload is relatively small for the GPU at this point, and the overhead of kernel launches becomes more significant. Leonid saves parallel kernels as CUDA graphs by default on the GPU, which reduces the overhead of kernel launches.

**DMA profile on SW:** The scratchpad-memory-management module in Leonid enables the use of DMA on the SW platform. We compare the performance of DMA-enabled Leonid with manually implemented code, both with and without DMA. In our implementation, only contiguous arrays leverage DMA operations, while non-contiguous arrays in SpMV continue to rely on non-DMA access. The performance evaluation is presented in Figure 9, where MG is replaced with CG, as CG involves fewer discrete data accesses compared to MG.

Compared to manually implemented code, Leonid achieves up to a 1.87× speedup. Furthermore, for manually implemented DMA code, Leonid without kernel fusion delivers performance nearly identical to the manual implementation. Notably, Leonid's automated kernel fusion method provides additional performance benefits, achieving a maximum speedup of 1.28×. The performance gain for the QMR algorithm is relatively lower compared to the other three algorithms. This is because the SpMV operation constitutes a larger portion of the QMR algorithm, and DMA was not applied to SpMV in our implementation.

## 4.3 Evaluation on Application Level

The evaluation on application level is shown in Figure 10. We compare the performance of Kokkos, RAJA and Leonid on CPU and GPU with Kokkos as the baseline, and we only calculate the speedup of automared kernel fusion on SW (For HPCG, we testes only up to a grid size of $256 \times 256 \times 256$ on SW due to memory capacity limitations).

On both the CPU and GPU, our Leonid outperforms Kokkos and RAJA. In the HPCG benchmark, Leonid achieves a maximum performance improvement of 17% (13% on average) on the CPU and 24% (19% on average) on the GPU over Kokkos . Meanwhile, RAJA achieves approximately 97% of Kokkos' performance. In the Cavity benchmark, Leonid achieves a maximum speedup of 1.27× (1.23× on average) on the CPU

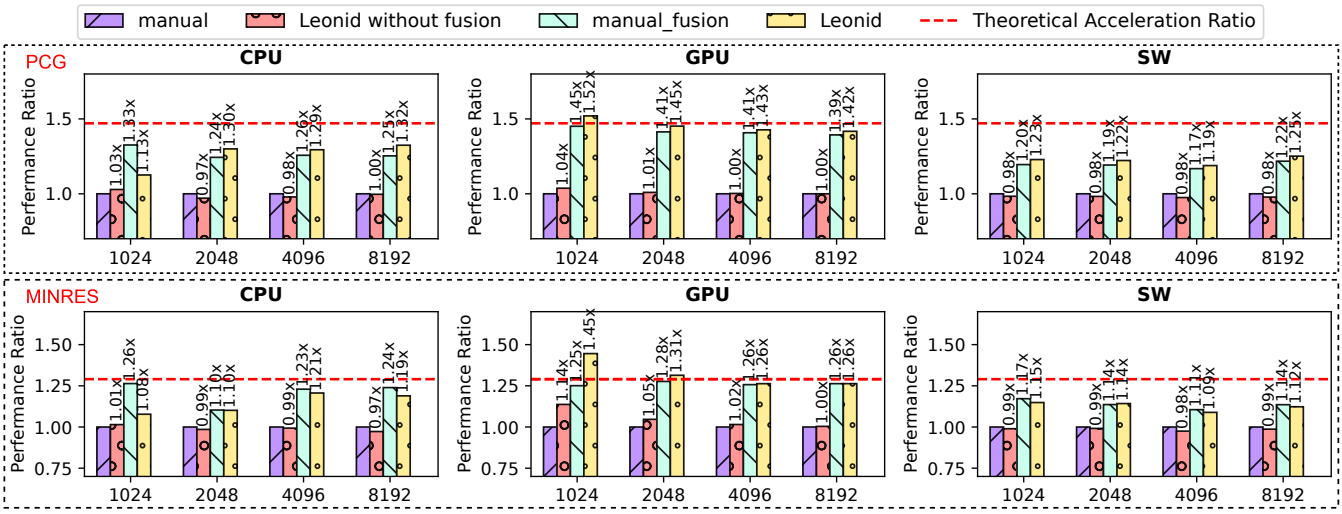


**Figure 8: Kernel fusion performance comparison: Leonid vs. manually implemented and fused codes in PCG and MINRES algorithms across different grid sizes. The x-axis represents the number of grid points along one side, eg., 1024 corresponds to a grid size of $1024 \times 1024$.**

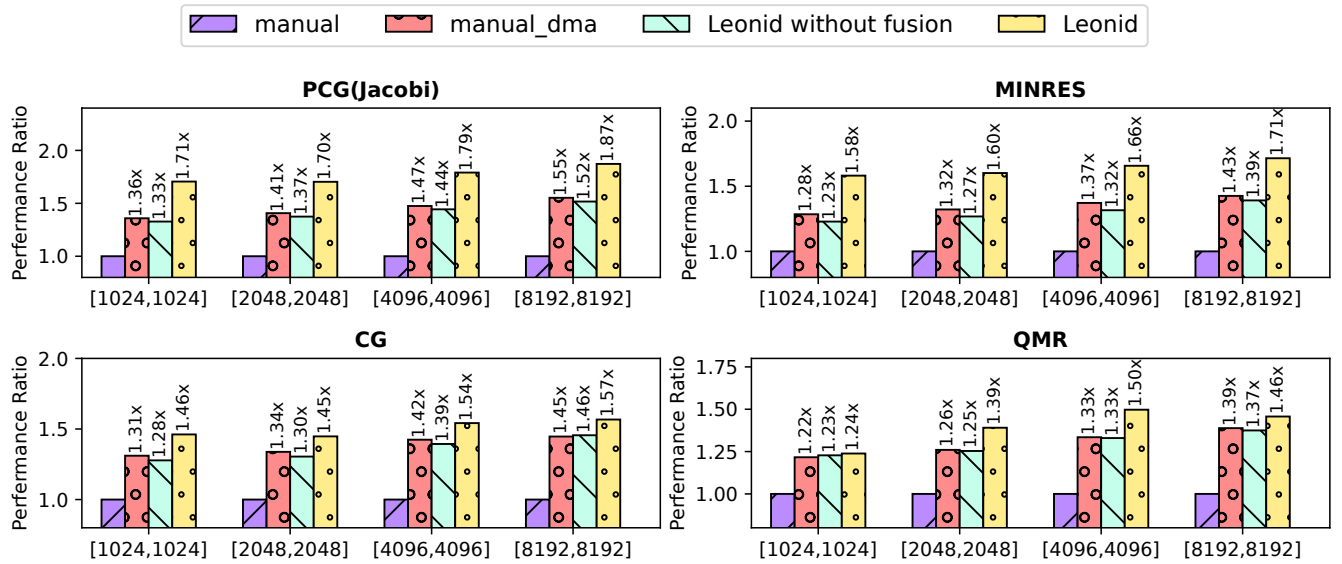


**Figure 9: Performance Profiling of Leonid with DMA Enabled on SW for PCG, MINRES, CG, and QMR Algorithms Across Different Grid Sizes.**

and 1.58× (1.30× on average) on the GPU relative to Kokkos. For LBM, Leonid achieves a maximum speedup of 1.19× (1.17× on average) on the CPU and 1.26× (1.24× on average) on the GPU compared to Kokkos. In MD, Leonid achieves up to 1.47× (1.45× on average) on the CPU and 1.34× (1.30× on average) on the GPU. On the SW platform, automated kernel fusion provides an average performance improvement of 16% in HPCG, 12% in Cavity, 15% in LBM, and 23% in MD.

We also successfully implemented an MPI version of the HPCG based on Leonid. The weak scaling parallel efficiency is shown in Figure 11, demonstrating that our Leonid is highly compatible with MPI.

## 4.4 Development cost discussion

Performance-portable frameworks simplify cross-device code development by supporting diverse platforms. For instance, the PCG case used in Section 4.2 spans 505 lines across CPU (122), GPU (230), and SW (153), while Leonid accomplishes the same functionality in 150 lines. Although adopting Kokkos, RAJA, or Leonid requires some extra effort compared to developing pure CPU code, these frameworks

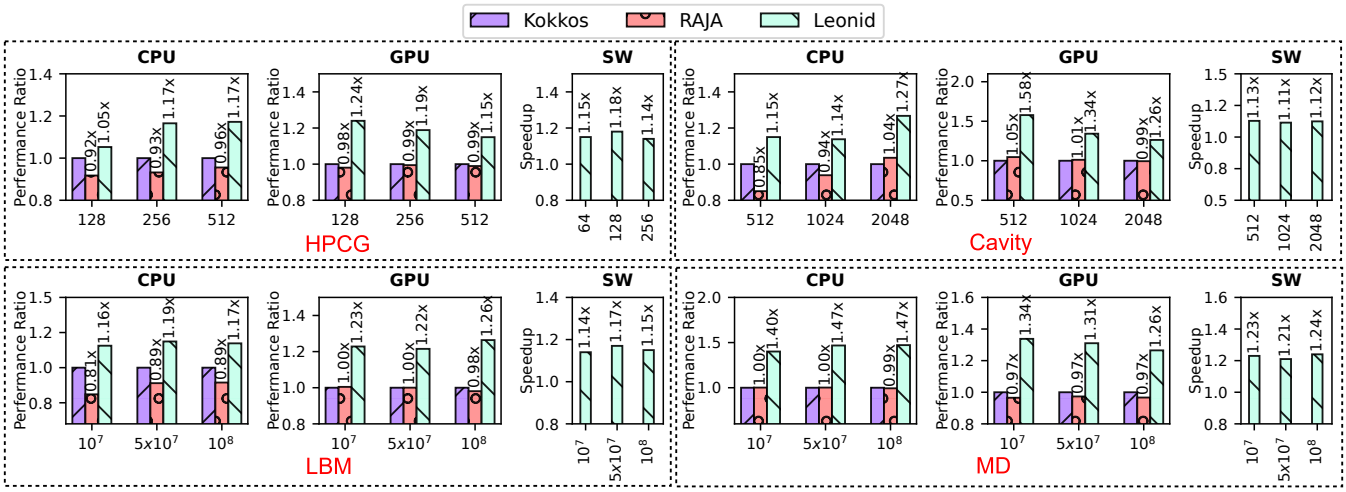


**Figure 10: Performance evaluation of mini-applications: comparisons of Kokkos, RAJA, and Leonid on both CPU and GPU, with the speedup achieved by Leonid's automated kernel fusion on the SW. For HPCG and Cavity, the x-axis ticks represent the grid size of each dimension, such as 128 in HPCG, which means the grid is $128 \times 128 \times 128$. In the LBM and MD subplots, the x-axis ticks represent the number of lattices and particles, respectively.**

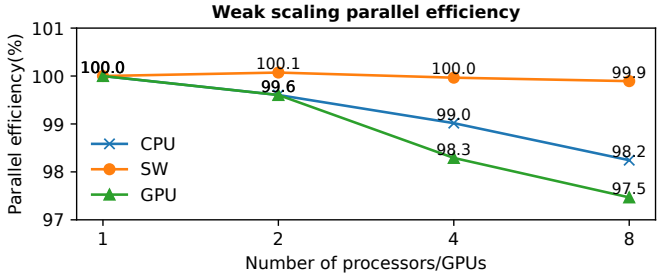


**Figure 11: Weak scaling parallel efficiency of HPCG on different platforms. The grid size is $512 \times 512 \times 512$ per GPU and CPU, and the size is $256 \times 256 \times 256$ on SW.**

enable multi-platform support with performance matching hardware-specific models, ultimately saving development costs across multiple platforms.

## 5 Related works

**Performance Portability.** Performance-portable programming models provide the solution to hardware diversity by abstracting hardware-specific details. OpenACC [4] and OpenMP [37] facilitate code offloading and data transfer through pragma directives, offering performance portability for heterogeneous systems and enabling large-scale heterogeneous simulations [15, 20, 55]. However, as directive-based programming models, OpenACC and OpenMP are user-friendly but lack fine-grained control over computing devices. OpenCL [35] and its extension SYCL [25] are performance-portable models at the programming language level. They both provide abstractions for managing data and controlling code execution, making it easier to achieve portability across diverse hardware platforms. Kokkos [17, 48] and RAJA [9] are notable libraries for performance portability. Kokkos evolved from the Kokkos Array [16] and has developed into a flexible programming framework for multiple mainstream hardware platforms. RAJA shares similar goals to Kokkos but also includes unique concepts and capabilities. While these performance-portable programming models mentioned above effectively address the challenges posed by hardware diversity, they lack support for automated kernel fusion, leaving them unable to fully tackle the performance bottlenecks of memory bandwidth-bound applications on modern hardware architectures. In these models, users are required to perform kernel fusion manually. Although SYCL [39] supports user-driven kernel fusion at the compiler level through its Just-In-Time (JIT) compilation framework, this approach is not directly applicable to models like Kokkos and RAJA. Moreover, the lack of automated kernel fusion significantly complicates the development and maintenance of large-scale projects involving dozens or even hundreds of kernels.

Performance-portable DSLs can be regarded as programming models tailored for specific applications. For instance, Halide [42] and TVM [13] are programming languages specifically designed for image processing and deep learning, respectively. These DSLs leverage the LLVM toolchain and code generation techniques to support multiple hardware

platforms and optimize kernel fusion. There are also DSLs designed for scientific domains with performance-portable, such as OpenArray [26] and JAX-CFD [28]. These are tailored for ocean model simulations and fluid dynamics computations respectively, and support multiple platforms by code generation.

**Kernel Fusion.** Kernel/Loop fusion is an effective optimization technique designed to improve the performance of bandwidth-bound codes. The significance of kernel fusion in optimizing bandwidth-bound applications is highlighted in reference [46]. Several linear algebra libraries, such as Eigen [19], LIBXSMM [22], and DESOLA [43], use loop fusion to enhance cache utilization. Eigen employs static techniques for fusion optimization. In contrast, LIBXSMM and DESOLA achieve kernel fusion through code generation and JIT compilation using runtime information. However, their approaches are limited to the linear algebra level and cannot be directly integrated into performance-portable programming models.

Automated operator fusion is widely used in the field of machine learning, with frameworks like TVM [13], TensorFlow [6], and ONNX [3] supporting it. This technique also offers significant benefits for optimization in scientific computing, and several works attempted to achieve automated kernel fusion for scientific applications. Mastoras et al. [32] designed a non-blocking execution GraphBlas, which enable automated kernel fusion through latency-execution. However, their approaches are specifically designed for GraphBLAS. Lamzed-Short et al. [29] proposed a novel LLVM optimization pass which can fuse C++ kernels automatically given a amount of additional information in the form of source code annotations. But this method requires modifying compilers and cannot currently be integrated into programming models like Kokkos and RAJA. References [18, 40, 51] provide source-to-source transformations that automatically convert non-fused code into fused codes for GPU kernels. However, their approaches are only applicable to GPUs, and the source-to-source method may encounter integration difficulties with existing scientific computing codes. Both Meng et al. [33] and Wahib et al. [50] proposed their own analysis frameworks that identify which kernels can be fused together, which only provide guidance for manual code fusion and do not fuse the kernels automatically.

Some studies on GPU kernel fusion focus not on reducing main memory access but on improving GPU utilization or power efficiency. Li et al. [30] and Usman Ahmed et al. [7] implemented fusion of CUDA and OpenCL kernels, respectively, to enhance resource utilization. Wang et al. [53] proposed a fusion method to reduce energy consumption and optimize power efficiency by fusing independent kernels.

## 6 Conclusion and Future work

In this paper, we present Leonid, a performance-portable programming model designed to support automated kernel fusion. Leonid integrates an automated kernel-fusion mechanism alongside modules for cross-device memory management and unified parallel and serial programming patterns, all of which are specifically tailored for automated kernel fusion. The memory-management module supports both main/global memory and scratchpad memory, enabling SW DMA operations and reducing unnecessary memory access through scratchpad management. The execution space currently supports the *Parallel For*, *Parallel Reduce*, and *Statement* execution patterns. On CPU and SW platforms, we use a runtime DDG for dynamic fusion, leveraging runtime information. On GPUs, we utilize a static DDG for compile-time fusion to minimize overhead.

Leonid delivers performance comparable to manually implemented versions for memory-bound codes, regardless of whether automated kernel fusion is applied. When automated kernel fusion is enabled, Leonid achieves up to 1.52× speedup (with an average of 1.19×) compared to manually implemented codes without kernel fusion, significantly outperforming Kokkos and RAJA. On the SW platform, with optimized scratchpad memory management, Leonid achieves up to 1.87× performance improvement compared to manually implemented codes without DMA, with a maximum speedup of 1.28× from automated kernel fusion alone.

The current version of Leonid is primarily focused on demonstrating the advantages of automatic kernel fusion and does not yet offer the extensive functionality provided by Kokkos and RAJA. In future work, we plan to integrate the automated kernel fusion method developed in this paper into Kokkos and RAJA to address this limitation. Furthermore, we aim to utilize the LLVM toolchain to implement additional optimizations using runtime information.

## Acknowledgments

This work was supported in part by National Key R&D Program of China (No. 2023YFA1011703), National Natural Science Foundation of China (No. 12131002, 12288101) and Changsha Science and Technology Bureau (No. KH2301001). In addition, special thanks to Chengdi Ma for assistance with the Cavity case discretization and valuable suggestions during writing.

## References

[1] 2024. *AMD HIP homepage.* https://www.amd.com/zh-cn/developer/resources/rocm-hub/hip-sdk.html

[2] 2024. *NVIDIA CUDA homepage.* https://developer.nvidia.com/cuda-toolkit

[3] 2024. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[4] 2024. *OpenACC homepage.* https://www.openacc.org/

[5] 2024. *Top 500 supercomputer lists.* https://www.top500.org/lists/top500/2024/06/

[6] Martín Abadi, Ashish Agarwal, Paul Barham, et al. 2016. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[7] Usman Ahmed, Jerry Chun-Wei Lin, and Gautam Srivastava. 2022. A ML-based resource utilization OpenCL GPU-kernel fusion model. *Sustainable Computing: Informatics and Systems* 35 (2022), 100683.

[8] B. J. Alder and T. E. Wainwright. 1959. Studies in Molecular Dynamics. I. General Method. *The Journal of Chemical Physics* 31, 2 (1959), 459–466.

[9] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, and Adam J. Kunen. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability, and Productivity in HPC (P3HPC)*. IEEE, 71–81.

[10] P. L. Bhatnagar, E. P. Gross, and M. Krook. 1954. A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Physical Review* 94, 3 (1954), 511.

[11] Guy E. Blelloch. 1990. *Prefix Sums and Their Applications.* Technical Report CMU-CS-90-190. Carnegie Mellon University.

[12] Huanqi Cao and Jiajie Chen. 2022. Design and Implementation of Shenwei Universal C/C++. *arXiv preprint arXiv:2208.00607* (2022).

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[14] NVIDIA Corporation. 2022. *NVIDIA H100 Tensor Core GPU Architecture: Exceptional Performance, Scalability, and Security for the Data Center.* White Paper. NVIDIA Corporation.

[15] Jose Monsalve Diaz, Kyle Friedline, Swaroop Pophale, Oscar Hernandez, David E. Bernholdt, and Sunita Chandrasekaran. 2019. Analysis of OpenMP 4.5 offloading in implementations: correctness and overhead. *Parallel Comput.* 89 (2019), 102546.

[16] H. Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. 2012. Manycore performance-portability: Kokkos multidimensional array library. *Scientific Programming* 20, 2 (2012), 89–114.

[17] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.

[18] Jan Fousek, Jiři Filipovič, and Matuš Madzin. 2011. Automatic fusions of CUDA-GPU kernels for parallel map. *ACM SIGARCH Computer Architecture News* 39, 4 (2011), 98–99.

[19] G. Gaël et al. 2021. Eigen v3. http://eigen.tuxfamily.org.

[20] A. Hart, R. Ansaloni, and A. Gray. 2012. Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers. *The European Physical Journal Special Topics* 210, 1 (2012), 5–16.

[21] Paul Havlak and Ken Kennedy. 1991. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems* 2, 3 (1991), 350–360.

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

[23] Michael Allen Heroux and Jack Dongarra. 2013. *Toward a new metric for ranking high performance computing systems.* Technical Report SAND2013-4744. Sandia National Laboratories.

[24] W. DANIEL HILLIS and JR. GUY L. STEELE. 1986. Data Parallel Algorithms. *Commun. ACM* 29, 12 (1986).

[25] Lee Howes and Maria Rovatsou. 2015. SYCL integrates OpenCL devices with modern C++.

[26] Xiaomeng Huang, Xing Huang, Dong Wang, et al. 2019. OpenArray v1.0: a simple operator library for the decoupling of ocean modeling and parallel computing. *Geoscientific Model Development* 12, 11 (2019), 4729–4749.

[27] Khronos SYCL Working Group. 2021. *SYCL 2020 Specification.* Technical Report. Khronos Group.

[28] Dmitrii Kochkov, Jamie A. Smith, Anastasiya Alieva, et al. 2021. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences* 118, 21 (2021), e2101784118.

[29] Andrew Lamzed-Short, Timothy R. Law, Andrew Mallinson, Gihan R. Mudalige, and Stephen A. Jarvis. 2020. Towards automated kernel fusion for the optimisation of scientific applications. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 45–55.

[30] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic horizontal fusion for GPU kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 14–27.

[31] Kai Lu, Yaohua Wang, Yang Guo, et al. 2022. MT-3000: a heterogeneous multi-zone processor for HPC. *CCF Transactions on High Performance Computing* 4, 2 (2022), 150–164.

[32] Aristeidis Mastoras, Sotiris Anagnostidis, and Albert-Jan N. Yzelman. 2022. Design and implementation for nonblocking execution in GraphBLAS: Tradeoffs and performance. *ACM Transactions on Architecture and Code Optimization* 20, 1 (2022), 1–23.

[33] Jiayuan Meng, Vitali A. Morozov, Venkatram Vishwanath, and Kalyan Kumaran. 2012. Dataflow-driven GPU performance projection for multi-kernel transformations. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[34] Richard Tran Mills, Mark F. Adams, Satish Balay, et al. 2021. Toward performance-portable PETSc for GPU-based exascale systems. *Parallel Comput.* 108 (2021), 102831.

[35] A. Munshi. 2009. The OpenCL Specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 1–314.

[36] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, et al. 2021. The design process for Google's training chips: TPUv2 and TPUv3. *IEEE Micro* 41, 2 (2021), 56–63.

[37] OpenMP Architecture Review Board 2021. *OpenMP Application Programming Interface* (version 5.2 ed.). OpenMP Architecture Review Board.

[38] S. J. Plimpton, S. G. Moore, A. Borner, et al. 2019. Direct simulation Monte Carlo on petaflop supercomputers and beyond. *Physics of Fluids* 31, 8 (2019), 080607.

[39] Víctor Pérez, Lukas Sommer, Victor Lomüller, Kumudha Narasimhan, and Mehdi Goli. 2023. User-driven online kernel fusion for SYCL. *ACM Transactions on Architecture and Code Optimization* 20, 2 (2023), 1–25.

[40] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. 2019. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 242–253.

[41] Milan Radulović. 2019. Memory bandwidth and latency in HPC: system requirements and performance impact. *Tesi doctoral, UPC, Departament d'Arquitectura de Computadors* (2019).

[42] 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.

[43] Francis P. Russell, Michael R. Mellor, Paul H.J. Kelly, and Olav Beckmann. 2011. DESOLA: An active linear algebra library using delayed evaluation and runtime code generation. *Science of Computer Programming* 76, 4 (2011), 227–242.

[44] Mehmet Sahin and Robert G. Owens. 2003. A novel fully implicit finite volume method applied to the lid-driven cavity problem—Part I: High Reynolds number flow calculations. *International Journal for Numerical Methods in Fluids* 42, 1 (2003), 57–77.

[45] S. Succi. 2001. *The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond*. Oxford University Press.

[46] Siham Tabik, G. Ortega, and Ester M. Garzón. 2014. Performance evaluation of kernel fusion BLAS routines on the GPU: iterative solvers as case study. *The Journal of Supercomputing* 70, 2 (2014), 577–587.

[47] Andrij Trokhymchuk and José Alejandre. 1999. Computer Simulations of Liquid/Vapor Interface in Lennard-Jones Fluids: Some Questions and Answers. *The Journal of Chemical Physics* 111, 18 (1999), 8510–8523.

[48] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, and Nathan Ellingwood. 2021. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 805–817.

[49] Loup Verlet. 1967. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review* 159, 1 (1967), 98.

[50] Mohamed Wahib and Naoya Maruyama. 2014. Scalable kernel fusion for memory-bound GPU applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 191–202.

[51] Mohamed Wahib and Naoya Maruyama. 2015. Automated GPU kernel transformations in large-scale production stencil applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 259–270.

[52] Chen Wang, Jian Xia, and Long Chen. 2024. A heterogeneous hybrid-precision finite volume method for compressible flow on unstructured grids. *Computers & Fluids* (2024), 106505.

[53] Guibin Wang, YiSong Lin, and Wei Yi. 2010. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. IEEE, 344–350.

[54] Junlin Wei, Pengfei Lin, Jinrong Jiang, et al. 2024. Accelerating LASG/IAP climate system ocean model version 3 for performance portability using Kokkos. *Future Generation Computer Systems* (2024).

[55] Chuanfu Xu, Xi Wang, Dali Li, Yonggang Che, and Zhenghua Wang. 2019. OpenMP 4.5-enabled large-scale heterogeneous Lattice Boltzmann multiphase flow simulations. *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 1007–1016.

[56] Qianchao Zhu, Hao Luo, Chao Yang, et al. 2021. Enabling and scaling the HPCG benchmark on the newest generation Sunway supercomputer with 42 million heterogeneous cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.