CCF Transactions on High Performance Computing
https://doi.org/10.1007/s42514-023-00159-7

REGULAR PAPER

swCUDA: Auto parallel code translation framework from CUDA to ATHREAD for new generation sunway supercomputer

Maoxue Yu[1] · Guanghao Ma[1] · Zhuoya Wang[1] · Shuai Tang[2] · Yuhu Chen[1] · Yucheng Wang[1] · Yuanyuan Liu[1] · Dongning Jia[1,2] · Zhiqiang Wei[2]

Received: 21 March 2023 / Accepted: 25 June 2023

Abstract
Since specific hardware characteristics and low-level programming model are adapted to both NVIDIA GPU and new generation Sunway architecture, automatically translating mature CUDA kernels to Sunway ATHREAD kernels are realistic but challenging work. To address this issue, *swCUDA*, an auto parallel code translation framework is proposed. To that end, we create scale affine translation to transform CUDA thread hierarchy to Sunway index, directive based memory hierarchy and data redirection optimization to assign optimal memory usage and data stride strategy, directive based grouping-calculation-asynchronous-reduction (GCAR) algorithm to provide general solution for random access issue. *swCUDA* utilizes code generator ANTLR as compiler frontend to parse CUDA kernel and integrate novel algorithms in the node of abstracted syntax tree (AST) depending on directives. Automatically translation is performed on the entire Polybench suite and NBody simulation benchmark. We get an average 40x speedup compared with baseline on the Sunway architecture, average speedup of 15x compared to x86 CPU and average 27 percentage higher than NVIDIA GPU. Further, *swCUDA* is implemented to translate major kernels of the real world application Gromacs. The translated version achieves up to 17x speedup.

Keywords Code translation · CUDA · ATHREAD · Sunway architecture

✉ Dongning Jia
dnjia@qnlm.ac

✉ Zhiqiang Wei
weizhiqiang@ouc.edu.cn

Maoxue Yu
mxyu@qnlm.ac

Guanghao Ma
ghma@qnlm.ac

Zhuoya Wang
zywang3@qnlm.ac

Shuai Tang
1737723493@qq.com

Yuhu Chen
yhchen@qnlm.ac

Yucheng Wang
ycwang@qnlm.ac

Yuanyuan Liu
yyliu@qnlm.ac

[1] Network and Information Center, Qingdao Marine Science and Technology Center, Qingdao, China

[2] Computer Science, Ocean University of China, Qingdao, China

1 Introduction

In recent years, heterogeneous computing architecture is widely applied in High Performance Computing (HPC) and AI domain. GPU and Sunway based architecture are the representative heterogeneous platforms. From the data of TOP500 list in November 2022, 13 supercomputer centers adopt NVIDIA GPU as accelerator in ranked top 20 (Strohmaier et al. 2022). Sunway TaihuLight (Fu et al. 2016) ranks at the top of the TOP500 list in 2016-2017. The new generation Sunway supercomputer is published with new SW26010P processors recently (Liu et al. 2021), showing powerful computing capability. Nowadays, a broad range of industries with over six hundred applications have already been accelerated by NVIDIA GPU (Nvidia 2018), including climate, weather, ocean model, Molecular dynamics, quantum chemistry, etc. Compute Unified Device Architecture (CUDA) based parallel kernel programming language is main stream of heterogeneous developing model. On the other hand, Sunway platform has very specific hardware characteristics and provides vendor-specific lightweight multi-thread library (ATHREAD) as kernel programming

Published online: 11 January 2024

language (Zhu et al. 2021), which is extended C-like language. Its software ecosystem is immature. Since both of GPU and Sunway have exclusive kernel programming library interface and these library are all extended C-like interface, designing automatically code translator from mature CUDA code to ATHREAD code will significantly improve programming productivity in Sunway platform.

However, developing code translator from CUDA to ATHREAD is a challenging work. First, as accelerator of heterogeneous platform, the hardware architecture between GPU Sunway CPE is different, which determines the low-level programming model. GPU is driven by streaming multiprocessors (SMs) (Nvidia 2023), but Sunway CPE is comprised of 8×8 slave core array. The hardware mapping model between GPU and Sunway CPE is needed to design as base of code translator. Second, the fine granularity of parallel programming model is different bewteen CUDA and ATHREAD. The CUDA programming model is thread parallel hierarchy, where three-dimensional block ID and thread ID is used to get data index and each thread calculate exclusive data in parallel. Sunway kernel programming model is parallel computing within the slave cores. Third, the different data usage of memory hierarchy greatly impact on the parallel performance. There is not general criterion for memory hierarchy optimization in Sunway. Several memory optimizing methods (e.g., data stride and double memory cache) are presented in (Liu et al. 2022), but it doesn't explain how to use in which situation. Fourth, since Sunway has no hardware support for atomic operation, write-conflict is big issue during random memory access pattern. (Chen et al. 2021) designs "dual-slice" partitioning algorithm to avoid the write-conflict in molecular dynamics simulations. But this solution is exclusive for specific application. There is not general solution for write-conflict.

To address these challenges, we propose an auto code translator framework *swCUDA* to efficiently transform CUDA codes to ATHREAD codes. *swCUDA* provides general transforming scheme and significantly improves programming productivity for Sunway. In summary, this paper makes the following contributions.

- Heuristic hardware mapping model between GPU and Sunway CPE is constructed as basis for code translation. Scaled affine algorithm is proposed to translate CUDA primitives of thread hierarchy to ATHREAD index.
- Directive based memory hierarchy translation and data redirection strategy is addressed to provide the optimal memory usage and data indexing transformation in ATHREAD Kernel.
- Directive based GCAR scheme provides general solution to remove random access issue due to no hardware support for atomic operation.

We evaluate our *swCUDA* by PolyBench suit and NBody simulation, where PolyBench is benchmark suite of numerical computations (Verdoolaege et al. 2013) and NBody is typical application in CUDA SDK (Han and Abdelrahman 2011). We demonstrate that translated application delivers an average 40x speedup compared to the baseline on the Sunway architecture, average speedup of 15x when compared to x86 CPU and average 27 percentage higher than NVIDIA GPU. Furthermore, as open-sourced molecular dynamics (MD) simulation, Gromacs (Kutzner et al. 2019) is selected for in-depth code translation study. Four complicated CUDA kernels are translated to ATHREAD codes. It shows up to 17x speedup compared to the baseline on the Sunway architecture.

2 Background

2.1 New generation sunway supercomputer

The new generation Sunway supercomputer integrates high-performance hetergeneous many-core processor SW26010P, each of which includes 6 Core Groups(CGs), and network communication on chip is adopted. Each CG consists of one Management Processing Element (MPE), one 8×8 CPE cluster and one Memory Controller(MC).

In terms of storage, each CG contains 16GB DDR4 memory which can be accessed by MPE and CPE through MC. Meanwhile, each CPE contains a 256KB fast Local Data Memory(LDM) and data transfer between LDM and main memory can be realized by Direct Memory Access(DMA). The SW26010P also has a performance advantage. MPE operates at 2.1GHz frequency, where CPE works at 2.25GHz. The peak performance of SW26010P processor is up to 14.026 TFLOP/s in double precision, corresponding memory bandwidth is 307.2 GB/s (Shang et al. 2021). The hardware promotion further enhances the the application performance.

2.2 Heterogeneous architecture and programming model

Generally, the spirit of heterogeneous programming is that host offloads computing intensive kernel code to accelerator unit. This model is used in both CUDA and ATHREAD programming language to accelerate scientific applications. Under the offload model, programmers need to write host code and device code, explicit data transfer between the host memory and accelerator memories. Host code is responsible for data organization and transmission, explicitly offloading kernel to computing accelerator. Device code is executed by many GPU threads in CUDA or by Sunway CPEs in

ATHREAD, in so-called Single Instruction Multiple Thread (SIMT) fashion.

In CUDA, threads are organized by grid. A grid contains three-dimensional thread-blocks, each thread-block contains three-dimensional threads (Nvidia 2023). Each thread is given unique thread ID. Built-in variables *blockIdx* and *threadIdx* is used as index of thread-blocks and threads to specify the position of data handled. Since thread-block in GPU has shared memory which can be accessed by its inner thread, it is generally used as basic execution unit for programmers. Instead of *blockIdx* and *threadIdx* usage in CUDA, each Sunway CPE has unique ID (*slaveID*) as index to handle the different data. From this point, thread-block in CUDA and single Sunway CPE act the compatible role as the basic execution unit in accelerator. On the other hand, codes are executed in parallel with inner threads of thread-block but sequentially for single CPE.

Furthermore, strict access authority and latency of various memory hierarchy has obvious contrast between CUDA and ATHREAD, which generates different memory accessing strategies. In CUDA, private registers are accessible by single thread and shared memory is visible to all threads within single thread-block. Global memory and constant memory is accessible for all threads. Variables defined in registers or shared memory in different thread-block can't access each other. The registers always have the lowest latency, secondly shared memory, at last global memory and constant memory. The only way to access the data in host by GPU is transferring data from host and device by DMA in CUDA. In ATHREAD, each CPE has its own LDM with lowest latency. Data defined in LDM can be accessed by other CPEs with remote memory access (RMA) API support (Zhu et al. 2021). Continuous shared memory in each CPE can be configured as global shared mode, which is accessed by all CPEs. Unlike CUDA, ATHREAD has flexible methods to access data in host by CPE: DMA transfer, cache access and remote non-cached access. The selection criteria is depending on the accessed data size combined with different memory accessing latency.

Both CUDA and ATHREAD provide specific programming API, declarator specifiers and primitives to support heterogeneous programming with extended C-like interface. The programming interface for thread hierarchy and memory hierarchy is foundation of heterogeneous programming and our focus of automatically translation framework.

2.3 Another tool for language recognition (ANTLR)

ANTLR (Parr and Fisher 2011; Parr 2013; Parr et al. 2014) is a powerful cross-language syntax parsing tool developed based on JAVA. It adopts the LL grammar parsing mode, a top-down parsing method, and can be used to read, process, execute and translate structured text or binary files. It is widely used in academia and industry to build all sorts of languages, tools, and frameworks. ANTLR parses lexical rules, syntax rules, and tree parsing rules by reading from the definition syntax file to generate corresponding lexer, parser, and tree-parser. The lexer converts the input character stream into word stream composed of phrases according to the lexical rules, thus obtaining the lexical analysis result of a specific language.The parser checks the syntax and combines these phrase word streams to generate a syntax tree, where all lexical information is stored at the leaf nodes. The tree-parser is to carry out traversal analysis and operation on the syntax tree, which is the tree representation of the abstract syntax structure of the source code, and finally generate the target parsing code. The main workflow of ANTLR is shown in the figure Fig. 1. The character stream passes through the parser from top to bottom, resulting in the object code.

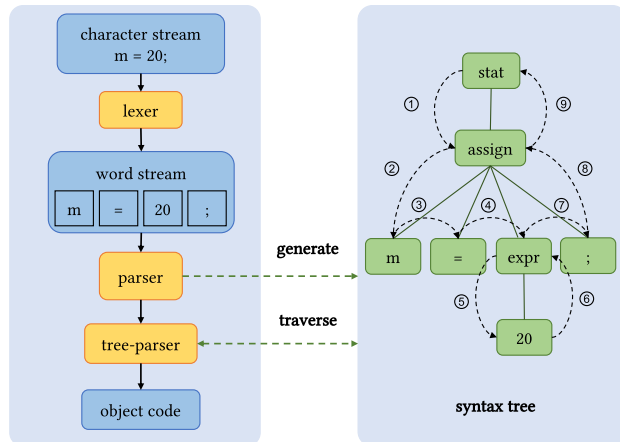


Fig. 1 The main workflow of ANTLR

Syntax tree is an important data structure that passes complete source code information through the parser to the rest of the system. To efficiently traverse the syntax tree, ANTLR provides listener and visitor mechanisms, which make the application logic and syntax files separated, the application program is packaged independently, so as to avoid the application logic scattered in the grammar file rules, so that in the listener can directly write the logic code entering rules and leaving rules, reduce the degree of coupling between programs. The ANTLR mentioned in this article is ANTRL4, which supports the generation of profilers for C, C++, Java, JavaScript, Objective-C, Perl, Python, Ruby and other programming languages, with a high degree of language freedom.

2.4 MD simulation application gromacs

As open-sourced molecular dynamics simulation, Gromacs is widely set up and run in versatile heterogeneous architecture

of HPC center all over the world. The simulation system, comprising thousands to millions of atoms according to molecule type, is executed for millions of time steps to derive the time evolution of atomic movement and produce MD trajectory (Kutzner et al. 2019). Therefore, number of HPC nodes is easily occupied for weeks in one molecular simulation. How to improve Gromacs performance in different heterogeneous architecture becomes progressive research. In Gromacs, there are two computing intensive parts: the short-range part of Coulomb and Van der Walls interactions, the long-range part of Coulomb interactions with PME method. By offloading these two computation parts to GPU, the performance is effectively accelerated.

Lennard–Jones potential is used to calculate Van der Waals interactions, shown in Eq. 1, where σ and ϵ is collision diameter and the depth of potential well, r is distance of particle pairs. Short-range coulomb force calculation combined with PME method is shown in Eq. 2, where $erfc(x)$ is the complementary error function. All the particles should be searched in cutoff radius. To improve the performance, nearest image algorithm and neighbor list search algorithm is mainly used under periodic boundary conditions.

$$u(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{1}$$

$$E(r) = \frac{1}{4\pi\epsilon_0}\frac{erfc(\beta r_{ij})}{r_{ij}}q_i q_j \tag{2}$$

Long-range coulomb force is mainly calculated by PME method in reciprocal space. Under the periodic boundary conditions, the equation of coulomb potential is shown in Eq. 3, where q_i is coulomb charges, r_i is coulomb positions, L is box length. Since this equitation is conditionally convergent, Ewald is decomposed by three parts: a direct sum in Cartesian space, a reciprocal sum in Fourier space and correction items (Essmann et al. 1995), shown in Eq. 4, 5 and 6. Direct space interactions are mapped to short-range coulomb, handled with cutoffs. Reciprocal space interactions are used to calculate the long-range coulomb force.

$$U_{coul} = \frac{1}{4\pi\epsilon_0}\frac{1}{2}\sum_{n}\sum_{i=1}^{N}\sum_{j=1}^{N}\frac{q_i q_j}{|r_{ij} + nL|} \tag{3}$$

$$E_{dir} = \frac{1}{2}\sum_{n}^{*}\sum_{i,j=1}^{N}\frac{q_i q_j erfc(\beta|r_j - r_i + n|)}{|r_j - r_i + n|} \tag{4}$$

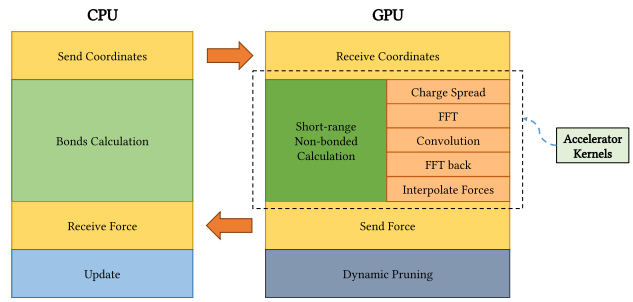


Fig. 2 Classic CPU+GPU heterogeneous parallel scheme of Gromacs

$$E_{rec} = \frac{1}{2\pi V}\sum_{m\neq 0}\frac{exp(-\frac{\pi^2 m^2}{\beta^2})}{m^2}B(m_1, m_2, m_3) \cdot F(Q)(m_1, m_2, m_3)F(Q)(-m_1, -m_2, -m_3) \tag{5}$$

$$E_{dir} = -\frac{1}{2}\sum_{(i,j)M}\frac{q_i q_j erf(\beta|r_i - r_j|)}{|r_i - r_j|} - \frac{\beta}{\sqrt{\pi}}\sum_{i=1}^{N} q_i^2 \tag{6}$$

The constant β is parameter that determines the convergence between the direct space sum and the reciprocal space sum. m is defined as the reciprocal lattice vectors of unit cell and its periodic images. The volume of unit cell is defined as V. Array $Q(m_1, m_2, m_3)$ is calculated and interpolated by fractional coordinates and spline coefficient, shown in Eq. 7. $F(Q)(m_1, m_2, m_3)$ is the transformation of array Q by 3DFFT. Array $B(m_1, m_2, m_3)$ is transforming coefficient, calculated by Euler exponential spline.

$$Q(k_1, k_2, k_3) = \sum_{i=1}^{N}\sum_{n_1,n_2,n_3} q_i M_n(u_{1i} - k_1 - n_1 K_1) \times M_n(u_2 i - k_2 - n_2 K_2) \cdot M_n(u_{3i} - k_3 - n_3 K_3) \tag{7}$$

In summary, the long-range Coulomb force calculation will experience five computing intensive parts. First, Charge spreading is performed to get the array $Q(k_1, k_2, k_3)$ by Eq. 7. Second, FFT is used to transform into reciprocal space to get array $F(Q)(m_1, m_2, m_3)$. Third, Calculation of convolutions is executed in reciprocal space. Fourth, iFFT back to direct space is executed. Finally, forces are interpolated to derive the forces at atom position (Jing et al. 2012; Harvey and De Fabritiis 2009). On the other hand, short-range calculation is executed by combines the Lennard–Jones and Coulomb force together in single accelerator kernel. The newest GPU offloading schemes is shown in Fig. 2.

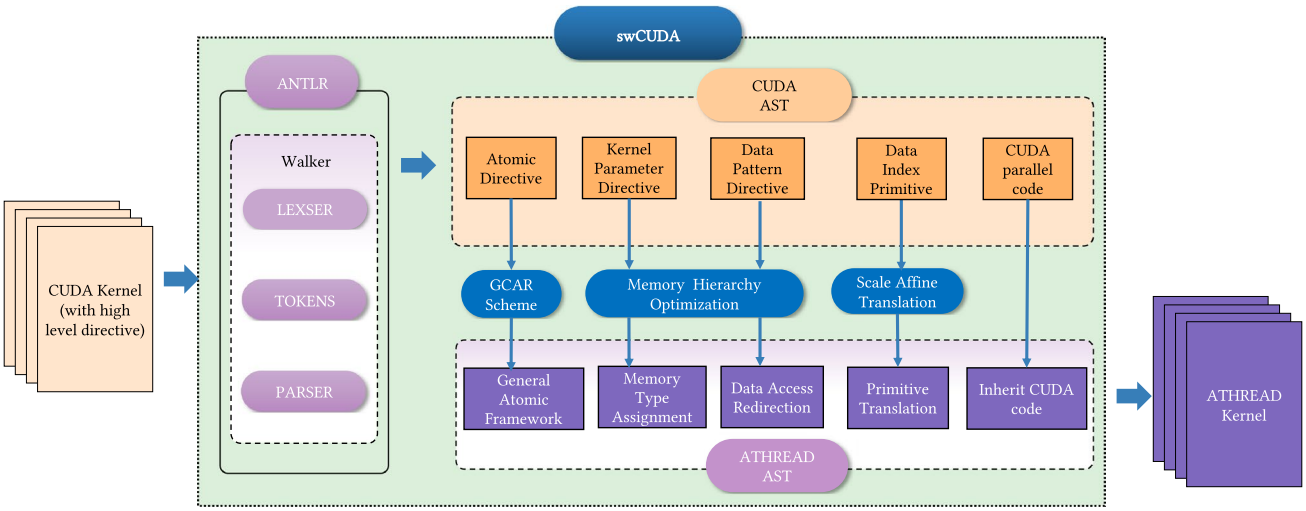


Fig. 3 *swCUDA* overiew

3 Algorithm and design

In this section, we first present the overview of *swCUDA*. Then, core mapping translation algorithm is addressed for CUDA to ATHREAD. After that, directive based memory hierarchy translation is introduced to assign optimal memory usage and data stride strategy. Finally, directive based GCAR scheme is presented to resolve random access issues. These optimization methodologies are detailed in depth in the following sections.

3.1 Overview

The architecture of *swCUDA* is shown in figure 3. First, high level directives, which is embedded to describe kernel input parameters, are used to provides accurate data management translation of our framework. Second, the AST of CUDA kernel code is constructed by ANTLR as compiler front-end. As powerful parsing tool, ANTLR provide *ParseTreeProperty* instance to associate property with a parse tree node. By using this class instance, we rewrite and store each node of CUDA AST and construct targeted ATHREAD AST. Third, our novel algorithms are invoked by corresponding to AST nodes of (1) directive declarations to store variable attributes, (2) statement of CUDA index primitive, (3) input parameter usage and (4) atomic operation. After rewriting the core AST nodes, entire CUDA parallel codes are inherited and organized to generate ATHREAD code. After that, although we have designed data access redirection translation, flexible data access pattern can't be all covered. Hence, annotation is embedded for uncovered data access pattern to make programmers to make quick data locality modification. Finally, with minor modification, the final ATHREAD kernel code is easily finalized. The powerful productivity is shown by *swCUDA*.

3.2 Core mapping translation algorithm

Core mapping translation algorithm is comprised of hardware mapping model and scale affine translation. Hardware mapping model provides the hardware mapping basis for the translation of low-level programming model. Scale affine translation provides data access redirection. By this algorithm, CUDA index primitives are effectively translated to ATHREAD primitives.

3.2.1 Hardware mapping model

We construct hardware mapping model between GPU and Sunway CPEs, shown in figure Fig. 4. First, Sunway MPE is mapping to multi-core CPU, acting the same role as host, responsible for overall logic and data organization of applications. Second, Sunway CPEs are mapping to NVIDIA stream multi-processors (SMs), acting the same basic executing unit for Single-Instruction Multi-Thread (SIMT) parallelism. Third, LDM in Sunway CPE is mapping to the

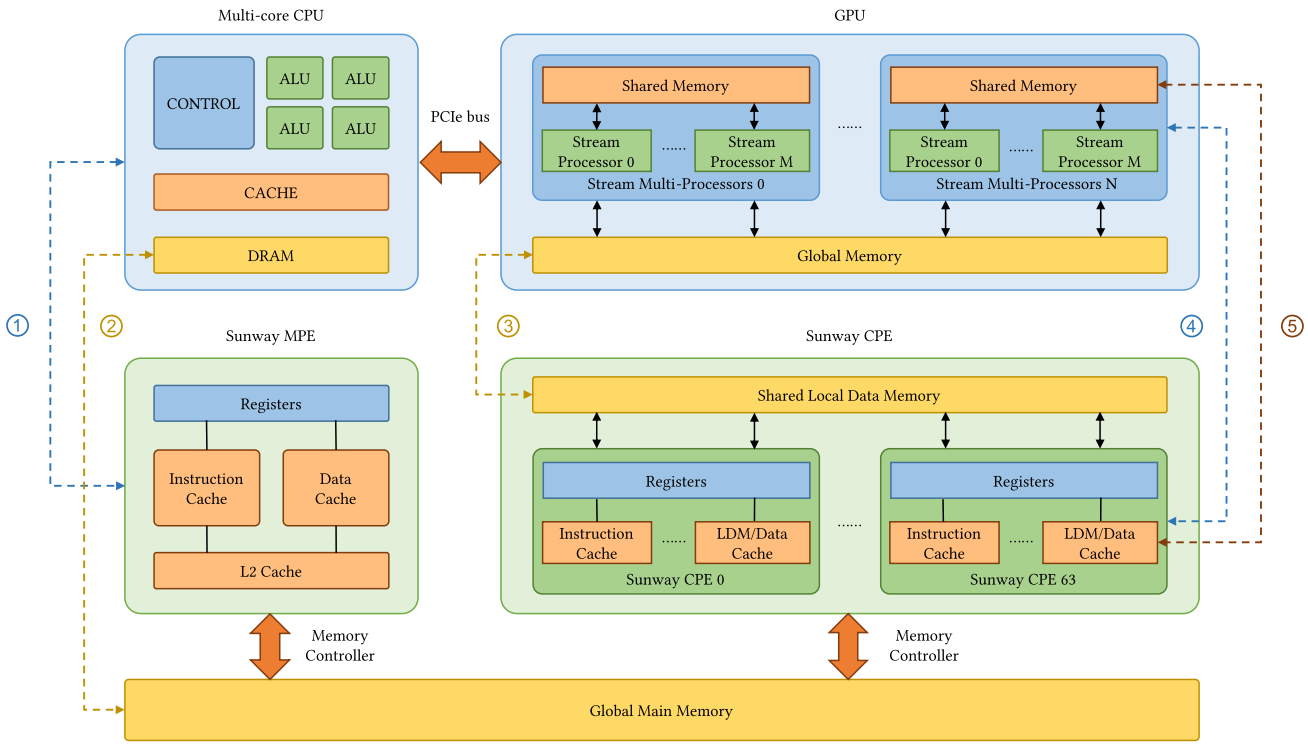


Fig. 4 Hardware Mapping Model. 1) host mapping. 2) main memory mapping. 3) on-board global memory mapping. 4) exclusive memory mapping. 5) computing accelerator mapping

exclusive shared memory for NVIDIA SM. Both of them are only accessed by single CPE or SM with low latency. Forth, Sunway host memory is akin to the Dynamic Random Access Memory (DRAM) of multi-cor CPU, where the input and output data should be stored in. Fifth, shared local data memory in Sunway is mapping to the on-board global memory of GPU, which have wider data accessing range, accessed by all SMs and Sunway CPEs. The mapping model comprises all the key hardware requirements of low-level programming.

3.2.2 Scale affine translation

Based on hardware mapping model, three-dimensional thread-blocks are executed by NVIDIA SM in parallel. Correspondingly, each thread-block should be executed in Single Sunway CPE. To map NVIDIA SM to Sunway CPE, the major challenge is how to arrange the thread-blocks to Sunway CPE. To address this issue, scale affine translation is proposed to transform CUDA thread and block level addressing to ATHREAD index. CUDA built-in index primitives are then transformed to ATHREAD primitive based code. Algorithm 1 describes the main translation steps.

First, towards to flexible numbers of thread-blocks, we partition and assign successive blocks to single CPE, referring line 3 to line 11 in Algorithm 1. The CPE index primitive *cpeId* is incorporated in the *blkStart* to indicate block stride index of each CPE. Second, for a three-dimensional *(Dx, Dy, Dz)* thread-blocks in CUDA, the thread-block id for block index *(x, y, z)* is $(x + y \times Dx + z \times Dx \times Dy)$ (Nvidia 2023), which is also applied to thread index. By this definition, three-dimensional block index primitives are translated to block index variable in ATREAD as index id for CPEs, as shown in line 17 to line 19 of Algorithm 1. Third, threads in single block are executed in parallel in CUDA. To map to Sunway CPE, we need to construct sequential execution in loop, where loop size is a total number of threads in single block. Each loop iterator represents one thread execution. Combined with loop iterator, thread index primitives are translated to ATHREAD variables, as shown in line 20 to line 26 of Algorithm 1.

Algorithm 1 Scale Affine Translation

Input: $cuGrid[3]$, $cuBlk[3]$: CUDA threads and thread-block index
1: $cpeId$: CPE index primitive
2: $cptNum$: computing core numbers of CPE
3: $blkSum \Leftarrow cuGrid[0] \times cuGrid[1] \times cuGrid[2]$
4: $thrdSum \Leftarrow cuBlk[0] \times cuBlk[1] \times cuBlk[2]$
5: $mod \Leftarrow blkSum/cptNum$
6: $rem \Leftarrow blkSum\%cptNum$
7: $blkStart \Leftarrow mod \times cpeId$
8: **if** $rem > 0$ **then**
9: $\quad blkStart+ = rem > cpeId?cpeId : rem$
10: **end if**
11: $blkEnd \Leftarrow blkStart + mod$
12: **if** $rem > 0$ **then**
13: $\quad blkEnd+ = rem > cpeId?1 : 0$
14: **end if**
15: $blk \Leftarrow blkStart$
16: **while** $blk < blkEnd$ **do**
17: $\quad blkIdz \Leftarrow blk/(cuGrid[0] \times cuGrid[1])$
18: $\quad blkIdy \Leftarrow (blk - blkIdz \times (cuGrid[0] \times cuGrid[1]))/cuGrid[0]$
19: $\quad blkIdx \Leftarrow blk\%cuGrid[0]$
20: $\quad$ **while** $thrd < thrdSum$ **do**
21: $\quad\quad thrdIdz \Leftarrow thrd/(cuBlk[0] \times cuBlk[1])$
22: $\quad\quad thrdIdy \Leftarrow (thrd - thrdIdz \times (cuBlk[0] \times cuBlk[1]))/cuBlk[0]$
23: $\quad\quad thrdIdx \Leftarrow thrd\%cuBlk[0]$
24: $\quad\quad$ **CUDA parallel code ...**
25: $\quad\quad thrd \Leftarrow thrd + 1$
26: $\quad$ **end while**
27: $\quad blk \Leftarrow blk + 1$
28: **end while**

Based on scale affine translation, the main CUDA index primitives are translated. Table 1 shows the complete translation, which is incorporated in *swCUDA* for auto translation. This novel algorithm build solid foundation to inherit the general CUDA parallel kernel to ATHREAD.

3.3 Memory hierarchy translation

Both NVIDIA GPU and Sunway CPE have flexible memory hierarchy mapping, as shown in figure 4. Unfortunately, the direct memory replacement to ATHREAD code is impracticable due to hardware difference. First, Sunway CPE has specific memory usage strategy (e.g., slave L1 cache direct access). Second, the shared local data memory in Sunway is far lower than on-board global memory of NVIDIA GPU. The data access redirection is required in ATHREAD. We present directive based memory hierarchy optimization and data redirection strategy to improve the computational accuracy of code translation and provide better parallel performance.

Table 1 Primitive affine translation

CUDA primitive	ATHREAD variable	Translation formular
gridDim.x	*cuGrid*[0]	*cuGrid*[0] = *gridDim.x*
gridDim.y	*cuGrid*[1]	*cuGrid*[1] = *gridDim.y*
gridDim.z	*cuGrid*[2]	*cuGrid*[2] = *gridDim.z*
blockDim.x	*cuBlk*[0]	*cuBlk*[0] = *blockDim.x*
blockDim.y	*cuBlk*[1]	*cuBlk*[1] = *blockDim.y*
blockDim.z	*cuBlk*[2]	*cuBlk*[2] = *blockDim.z*
blockIdx.x	*blkIdx*	*blk* [1] % *cuGrid*[0]
blockIdx.y	*blkIdy*	(*blk* − *blkIdz* × (*cuGrid*[0] ×*cuGrid*[1]))/*cuGrid*[0]
blockIdx.z	*blkIdz*	*blk*%(*cuGrid*[0] × *cuGrid*[1])
threadIdx.x	*thrdIdx*	*thrd* [2] % *cuBlk*[0]
threadIdx.y	*thrdIdy*	(*thrd* − *thrdIdz* × (*cuBlk*[0] ×*cuBlk*[1]))/*cuBlk*[0]
threadIdx.z	*thrdIdz*	*thrd*%(*cuBlk*[0] × *cuBlk*[1])

[1] *blk* presents block id assigned to CPE

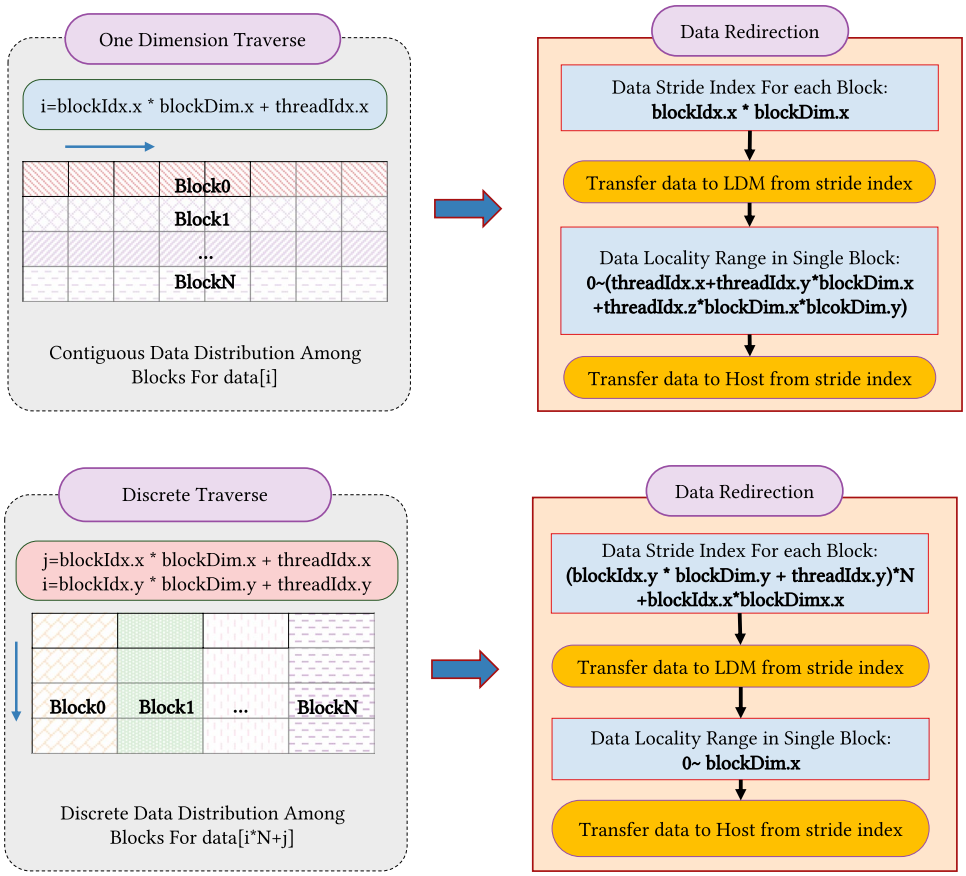
[2] *thrd* presents thread id executed in single CPE

3.3.1 Memory hierarchy optimization

As shown in Algorithm 2, we design concise directive to illustrate the characteristics of CUDA input variables by the format *paraVarAttr*(*type*, *var*, *size*, *attr*). Each directive starts with *paraVarAttr* as indicator for *swCUDA*, which is parsed as declarator specifier. The parameters of the *paraVarAttr* describes the attribute of CUDA kernel input variables, including variable name, type, size and character of transmission. *swCUDA* parse the directive and assign specific memory usage type for each kernel input variable.

CUDA input variables are categorized by *readonly* and *inout* indicated by *attr*, as shown in line 2 and line 8 in Algorithm 2. For *readonly* variable, it is unchangeable for all CPEs. Hence, it can be loaded by DMA to LDM (*LDM_BY_DMA*) or be directly accessed from slave L1 cache (*SLAVE_CACHE*) according the *size* in directive parameters. Since the total size of LDM is 256KB, we use 128KB as cut-off point to determine if we need load data to LDM. For *inout* variable, it is updated by all CPEs for parallel computing, where each CPE is charge of partial data. Hence, global shared memory (*GLB_SHR_MEM*) or data stride by DMA (*DMA_STRIDE*) is suitable for *inout* variable. The advantage of global shared memory variable is directly mapping to CUDA global memory and no extra modification for the data redirection. Here we choose 4096KB ($64KB \times 64$) as cut-off point since slave cache and LDM size should be taken into account. Towards to the variable assigned to memory type *DMA_STRIDE*, each CPE executes calculation with data stride indexing. By this situation, data redirection is required in Sunway, since CUDA

Fig. 5 Data redirection strategy



kernel can directly access the data from on-board global memory, whose capacity is above 16GB (Nvidia 2020).

3.3.2 Data redirection strategy

The memory type *DMA_STRIDE* requires data redirection in Sunway CPE, which need more manual efforts for programmers. To achieve automatically translation, data redirection strategy is addressed, as shown in Fig. 5. The data indexing pattern in CUDA is either contiguous or discrete distributed among blocks (Garland et al. 2008; Milakov 2015). Towards to the characteristics, we design tailored data redirection translation strategy. Data is accessed along the arrow directed order. Identifying data stride index among blocks and data locality in single thread-block are the key to realize the data redirection access in each Sunway CPE. For contiguous data indexing pattern, data stride index is directly followed by block index ID. The data locality range is thread organization of thread-block. On the other hand, discrete data indexing pattern is complicated, which is combined by at least two direction traverse. By parsing global data indexing usage (e.g., $data[i * N + j]$ in Fig. 5), data locality expression is acquired from the last combination of *threadIdx* usage, since data accessing is only contiguous along single thread index direction in discrete indexing pattern. Hence, data stride index is constructed by the reset expression of global data indexing usage, which combines block index id, thread dimension and thread index.

Algorithm 2 Memory Hierarchy Optimization

Input: $paraVarAttr(type, var, size, attr)$: Directive for CUDA input parameter variables
Output: $varMem$: variable memory usage type in Sunway

```
while paraVarAttr != NULL do
    if attr is readonly then
        if size less than 128KB then
            varMem ⇐ LDM_BY_DMA
        else
            varMem ⇐ SLAVE_CACHE
        end if
    else if attr is inout then
        if size less than 4096KB then
            varMem ⇐ GLB_SHR_MEM
        else
            varMem ⇐ DMA_STRIDE
        end if
    end if
end while
```

To implement data redirection strategy in *swCUDA*, high level directive is designed by format *dataPattern*(*pattern*1, *pattern*2, *usage*). Parameter *pattern*1 and *pattern*2 denote global thread indexing pattern in CUDA kernel, whose format is required to comprise *blockIdx* to represent global indexing. If there is only one global indexing pattern, *pattern*2 is filled with *NONE*. The parameter *usage* illustrates the actual data global indexing usage. As *swCUDA* parsed the directives, the data stride index and data locality usage are automatically generated. Data with global indexing usage is translated to local indexing with DMA transfer by data stride index in single Sunway CPE. Data redirection strategy improves memory hierarchy optimization, where flexible data redirecting method of the memory type *DMA_STRIDE* is generated according to the different data indexing pattern.

3.4 Grouping calculation asynchronous reduction scheme

Nvidia GPU supports atomic operation in hardware with low latency to remove random access issue easily. In the contrast, there is no hardware support for atomic operation in Sunway CPEs. Avoiding random access issue is always challenging work. In this section, we propose GCAR scheme to provide general solution to automatically transform atomic operation.

As shown is Algorithm 3, high level directive is used to describe the detailed atomic attribute by *AtomOpr*(*type*, *var*, *size*, *rcvNum*, *copyin*). Sunway CPEs is partitioned to computing cores (*cptCore*) and receiving cores (*rcvCore*). When the atomic directive is parsed, *swCUDA* embeds *cptCore* and *rcvCore* procedure respectively. CUDA code is inherited under *cptCore* procedure. *rcvNum* is used to determine how many *rcvCore*s is needed, dynamically adjusted according to the size of *var*. *copyin* indicates if atomic variable is needed to transfer to LDM. First, Line 1 in Algorithm 3 calculate atomic data stride at the beginning of kernel code. *rcvCore* uses it to transfer atomic data, where *cptCore* uses it to calculate the target core that should be sent by RMA. *rcvCore* is responsible for accumulating the atomic variable *var* and its local index transferred from computing cores by RMA. When RMA transfer is done, the asynchronous flag will be set. Then, receiving cores do accumulations(refers as line 11 in Algorithm 3). Once stop flag is received from all computing cores, the procedure of receiving core is finished and *var* is transferred back by DMA. On the other hand, Computing core do calculation and send data to receiving core by RMA asynchronously. Since *var* is transferred and spread in receiving cores, data locality and targeted receiving core are needed to redirect and calculate by *dataStride*, shown as line 24 to line 25 in Algorithm 3.

By explicit data partition method, RMA communication and fast synchronization in CPEs, GCAR Algorithm shows excellent performance and generality. We encapsulate the algorithm to the application program interface (API) to provide general solution for resolving the random access issue.

4 Application study

In this section, we first illustrate the translation of *swCUDA* for standard matrix multiply benchmark. Then, we perform in-depth analysis on complicated kernels in Gromacs. We present that *swCUDA* is adaptive to the flexible CUDA kernel programming.

```
//kernel parameter directive
userDef paraVarAttr(float,a,sizeof(float)*NI*NK,readonly);
userDef paraVarAttr(float,b,sizeof(float)*NK*NJ,readonly);
userDef paraVarAttr(float,c,sizeof(float)*NI*NJ,inout);
//data redirection directive
dataPattern(blockIdx.y*blockDim.y+threadIdx.y,
                blockIdx.x*blockDim.x+threadIdx.x, i*NJ+j);
//CUDA Kernel Funciton
__global__ void gemm(float *a, float *b, float *c){
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  if ((i < NI) && (j < NJ)){
    c[i*N+j] *= BETA;
    int k;
    for(k = 0; k < M; k++){
      c[i*NJ+j] += ALPHA * a[i*NK+k] * b[k*NJ+j];
    }
  }
}
```

Fig. 6 CUDA orginal kernel function with directive

4.1 General kernel translation

The general matrix multiplication (GEMM) computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product with CUDA code in Polybench suite is shown in figure 6, which calculates 512×512 symmetric matrix C by general matrices A and B. The data organization is 32×8 for thread-blocks and 16×64 for blocks. The directives are inserted to describe the attribute of kernel functional parameter variables. a and b are read only variables, where c is writable variable. The size of these three variables are 1MB.

Algorithm 3 Grouping Calculation Asynchronous Reduction

Input: $AtomOpr(type, var, size, rcvNum, copyin)$: Directive of Atomic operation
$ayncFlag[cptNum]$: aync status of $cptCore$
$cptEnd[rcvNum]$: done status of $rcvCore$

$dataStride$: number of data in each $rcvCore$
Procedure in $rcvCore$:
$finish \Leftarrow cptCore$
if $copyin$ **then**
  transfer var to $rcvCore$ with $dataStride$
end if
while true **do**
 For the i $cptCore$
  if $ayncFlag[i]$ is true **then**
    get val and $localPos$ from i $cptCore$
    $var[localPos] \Leftarrow var[localPos] + val$
    $ayncFlag[i] \Leftarrow false$
  end if
  if $cptEnd[i]$ is true **then**
    $finish \Leftarrow finish - 1$
  end if
 EndFor
  if $finish$ is zero **then**
    $break$
  end if
end while
transfer var back host with $dataStride$
Procedure in the i $cptCore$:
$tarCore \Leftarrow size/dataStride$
$localPos \Leftarrow size\%dataStride$
send val and $localPos$ to $tarCore$ by RMA
RMA set $ayncFlag[i]$ true
RMA set $cptEnd$ for all $rcvCore$ true

swCUDA executes scale affine algorithm and realizes the code translation, as shown in Fig. 7. First, *swCUDA* automatically organizes original CUDA thread-blocks array (*cuGrid[3]*), threads array (*cuBlock[3]*) and kernel parameter variable pointers to the structure variable of ATHREAD kernel function. By DMA transfer, the array value and pointer address can be accessed locally. Then, block and thread affine translation is automatically accomplished and embedded in the entry of kernel function of

```
//Kernel parameter organization
typedef struct kernelPara{
  int cuGrid[3];
  int cuBlock[3];
  float *a;
  float *b;
  float *c;
}Paras;
CRTS_dma_iget(Para, Paras, sizeof(Para), &dma_rply);
int blkSum = Para->cuGrid[0]*Para->cuGrid[1]*Para->cuGrid[2];
int thrdSum = Para->cuBlock[0]*Para->cuBlock[1]*Para->cuBlock[2];
int quotient   = blkSum / 64;
int remainder  = blkSum % 64;
int blockStart = quotient * cpeId;
if(remainder > 0)
  blockStart += (remainder > cpeId ? cpeId : remainder);
  int blockEnd = blockStart + quotient;
  if(remainder > 0)
    blockEnd   += (remainder > cpeId ? 1 : 0);
  for(int block = blockStart; block < blockEnd; block++){
    blkId_z = block / (Para->cuGrid[0] * Para->cuGrid[1]);
    blkId_y = (block - blkId_z * (Para->cuGrid[0]
                * Para->cuGrid[1])) / Para->cuGrid[0];
    blkId_x = block % Para->cuGrid[0];
    for(int threadId = 0; threadId < threadSum; threadId++) {
      thrdId_x = threadId % Para->cuBlock[0];
      thrdId_z = threadId / (Para->cuBlock[0]*Para->cuBlock[1]);
      thrdId_y = (threadId - thrdId_z * (Para->cuBlock[0]
                  * Para->cuBlock[1])) / Para->cuBlock[0];
      int j=blkId_x *Para->cuBlock[0]+thrdId_x;
      int i=blkId_y *Para->cuBlock[1]+thrdId_y;
```

Fig. 7 Scale affine translation

ATHREAD according to Algorithm 1. CUDA blocks are spread to CPE cores equally. Each CPE executes consecutive 16 blocks in GEMM example, where block index in each CPE is addressed by CPE index primitive *cpeId*. We use loop to iterate over the blocks, and CUDA block primitive is translated according Table 1. Within the block loop, we further embed thread loop to iterate each thread with same method. In this example, each block executes 256 threads totally.

After that, directives are parsed by *swCUDA* to execute memory hierarchy translation as shown in Fig. 8. We use label *userDef* to indicate *swCUDA* that directives are parsing. *paraVarAttr* is used to illustrate variable and its attributes, which are first aquired by *swCUDA* for the following translation. When *swCUDA* traverses the kernel functions, the *readonly* and *inout* variables are detected by comparing with pre-saved variables from directives. Since the size of *readonly* variable is larger than 128KB, described by Algorithm 2, they are set to *SLAVE_CACHE*. Then the variable is replaced with the structure variable from *Para* in the node of parsing AST by ANTLR. The variable can be

```
//global shared variable definition
__thread_local_share volatile float c_s[NI *NJ] = { 0 };
//moving data to global shared memory by DMA
CRTS_memcpy_sldm(c_s, Para->c, sizeof(float)*NI*NJ, MEM_TO_LDM);
int j=blkId_x *Para->cuBlock[0]+thrdId_x;
int i=blkId_y *Para->cuBlock[1]+thrdId_y;
if((i<NI)&&(j<NJ)){
  c_s[i *NJ+j]*=BETA;
  int k;
  for(k=0;k<NK; k++){
    c_s[i*NJ+j]+=ALPHA*Para->a[i*NK+k] *Para->b[k*NJ+j];
  }
}
//transfer data back to host by DMA
CRTS_memcpy_sldm(Para->c, c_s, sizeof(float)*NI*NJ, LDM_TO_MEM);
```

Fig. 8 Memory optimization with global shared Memory

```
dataLoad = true;
for(int threadId = 0; threadId < threadSum; threadId++) {
  int j=blkId_x*Para->cuBlock[0]+thrdId_x;
  int i=blkId_y*Para->cuBlock[1]+thrdId_y;
  local = threadId % Para->cuBlock[0];
  if(local == 0){
    glbInx = (i-1)*N+blkId_x*Para->cuBlock[0];
    dma_put(&Para->c[glbInx],c_s,sizeof(float)*Para->cuBlock[0]);
    dataLoad = true;
  }
  if(dataLoad){
    glbInx = i*N+blkId_x*Para->cuBlock[0];
    dma_get(c_s,&Para->c[glbInx],sizeof(float)*Para->cuBlock[0]);
    dataLoad = false;
  }
  if((i<NI)&&(j<NJ)){
    c_s[local]*=BETA;
    int k;
    for(k=0;k<NK; k++){
      c_s[local]+=ALPHA*Para->a[i*NK+k] *Para->b[k*NJ+j];
    }
  }
}
```

Fig. 9 Data redirection translation

accessed by slave cache, as shown in line 11 of Fig. 8. For *inout* variable, since its size lower than 4MB, it is assigned to *GLB_SHR_MEM*. Hence, *swCUDA* inserts specific keywords and API for the definition and data transfers of global shared variables.

If the matrix size of GEMM enlarges to 2048×2048, the *inout* variable is assigned to *DMA_STRIDE* due the limitation of LDM size. By this case, we need transfer data by DMA with stride and redirect CUDA variable index to meet ATHREAD indexing requirement. The data redirection directive *dataPattern* is parsed to save index pattern and data usage. The GEMM index pattern is discrete traverse with two dimension, but DMA transfer is required for single dimension for contiguous data. Hence, we use *dataLoad* to toggle DMA discrete transfer. As shown in Fig. 9, DMA stride index *glbInx* is calculated by usage and pattern of directive *dataPattern* and single DMA transfer size is determined by *pattern*2. Hence, global data index is automatically translated by this fixed indexing pattern. Data locality is then redirected to one dimension of threads.

With the tailored translation algorithm and high level directives, *swCUDA* successfully transforms benchmark CUDA kernels to ATHREAD kernels. Our flexible scale affine algorithm and memory hiearchy translation demonstrate the productivity and effectiveness of *swCUDA*.

4.2 Gromacs translation

As described in Sect. 2.4, there are totally six CUDA kernels for short-range pair interaction and long-range coulomb calculation with PME method. In this section, we select charge spreading kernel in PME method as example to present the practical code translation process by *swCUDA*. The same process is adapted to the other CUDA kernels of Gromacs for automatically translating to ATHREAD kernel.

Accelerating PME method in Sunway is challenging work since the algorithm of B-Spline interpolation in charge spreading causes memory random access issue (Lee et al. 1997). To get around the problem, PME method is reconstructed by local grid reordering in previous work (Zeng et al. 2021). This method needs further verification for generality. In this section, we presents how the charge spreading kernel is automatically translated to ATHREAD kernel by *swCUDA*.

In charge spreading method, each charge is mapped to grid location by scaled fractional coordinates of particle, and then distributed to surrounding grid volume, dependent on the spline interpolation order n (fixed at 4). Each charge is spread over $n^3 = 64$ grid points. In another word, each dimension of coordinate per particle has 4 spline parameters. In CUDA kernel implementation, each particle is assigned 4 threads for calculation. With the block size of 128 threads, atom number per block (*atomsPerBlock*) is 32 (128/4). The index of thread-blocks is organized by total atoms divided *atomsPerBlock*. CUDA threads organization is $cuBlock[3] = \{order, 1, atomsPerBlock\}$ and $cuGrid[3] = \{totalAtoms/atomsPerBlock, 1, 1\}$. Charge spreading kernel is decomposed to two parts: spline data calculation and spread charge. Spline calculation is used to calculate spline data with fractional coordinates. Then spread charge uses calculated spline data to execute $n^3 = 64$ interpolations in three dimensions per particle. It unrolls Z axis to 4 threads and loop x and y axis to calculate distributed charge with 64 interpolations. Due to large scaled paralleling charge spread calculation of multi-atoms and periodic boundary of simulation box, accumulating charge in one grid point would inevitably encounter random access issue (Kutzner 2008). It is solved by using atomic API in CUDA implementation.

With our novel algorithm, *swCUDA* completely translates charge spreading kernel with high efficiency, as shown in Fig. 10. First, atomic directive is parsed and translated as it brings code structure change in ATHREAD kernel. As result, API *writeConflictInit* is embedded to calculate the data stride indexing. Then, *cptCore* and *rcvCore* procedure is embedded. API *rcvCoreGetData* is main function of *rcvCore* procedure, which accumulates atomic data from *cptCore* by RMA, described in Algorithm 3. Main CUDA kernel code is inherited in *cptCore* procedure. CUDA atomic API operation is parsed and data offset and updated value is split, as shown in line 41 and 42 of Fig. 10b. The *spreadVal* contains global index and updated value. Then, asynchronous API *sendDataToRcvCore* sends it to target *rcvCore* by RMA. After all data sent to *rcvCore*, API *finishDataSend* is used to notify *rcvCore* to end receiving work. The above API functions are manually developed with high performance.

```
//atomic directive
AtomOpr(float,gm_grid,sizeof(float)*fSize,8,no);
//kernel parameter directive
paraVarAttr(float,d_coeff,sizeof(float)*atomSize, readonly);
paraVarAttr(float,d_coord,sizeof(float)*atomSize*3, readonly);
__global__ void pme_spline_and_spread_kernel(struct KP *pmePara){
  //CUDA variable definition
  __shared__ int sm_coeff[atomsPerBlock];
  __shared__ float sm_theta[atomsPerBlock*DIM*order];
  //CUDA data indexing
  int localIndex = threadIdx.z  * blockDim.x + threadIdx.x;
  int globalIndex = blockIdx.x * blockDim.z + localIndex;
  sm_coeff[localIndex] = d_coeff[globalIndex];
  //CUDA parallel code
  calculate_splines(pmePara,globalIndex,sm_coeff,sm_theta);
  __syncthreads();
  spread_charages(pmePara,sm_theta);
}
__device__ void spread_charages(struct KP *pmePara,
        float sm_theta[atomsPerBlock*DIM*order]){
  for (int ithy = ithyMin; ithy < ithyMax; ithy++){
    int iy = iyBase + ithy;
    float Val = thetaZ * thetaY * (*atomCharge);
    int offset = iy * pnz + iz;
    for (int ithx = 0; (ithx < order); ithx++){
      int ix = ixBase + ithx;
      int gridIndexGlobal = ix * pny * pnz + offset;
      const float thetaX = sm_theta[splineIndexX];
      atomicAdd(gm_grid + gridIndexGlobal, thetaX * Val);
    }
  }
}
```

(a) CUDA Kernel With Directives

```
void pme_spline_and_spread_kernel(struct KP *Para){
  //atomic initilization
  writeConflictInit(sizeof(float)*fSize, &dmaSize,&dataStride);
  //cptCore procedure
  if(cpeId < CptCore){
    //local variable definition
    int sm_coeff[atomsPerBlock];
    float sm_theta[atomsPerBlock*DIM*order];
    //Scale affine translation
    for(int block = blockStart; block < blockEnd; block++){
      blkId_x = block % Para->cuGrid[0];
      for(int threadId = 0; threadId < threadSum; threadId++) {
        thrdId_x = threadId%Para->cuBlock[0];
        thrdId_z = threadId/(Para->cuBlock[0]*Para->cuBlock[1]);
        int localIndex = thrdId_z*Para->cuBlock[0]+thrdId_x;
        int globalIndex = blkId_x*Para->cuBlock[2]+localIndex;
        sm_coeff[localIndex] = Para->d_coeff[globalIndex];
        calculate_splines(pmePara,globalIndex,sm_coeff,sm_theta);
      }
      for(int threadId = 0; threadId < threadSum; threadId++)
        spread_charages(pmePara,sm_theta);
    }
    finishDataSend(receiveId, &rw_rply[cpeId]);
  }else{
    //rcvCore procedure
    int offset = (cpeId-CMPT_CORE)*dataStride;
    rcvCoreGetData(&para_s->d_Grid[offset],dmaSize,dataStride);
  }
}
void spread_charages(struct KP *Para,
            float sm_theta[atomsPerBlock*DIM*order]){
  for (int ithy = ithyMin; ithy < ithyMax; ithy++){
    int iy = iyBase + ithy;
    float Val = thetaZ * thetaY * (*atomCharge);
    int offset = iy * pnz + iz;
    for (int ithx = 0; (ithx < order); ithx++){
      int ix = ixBase + ithx;
      int gridIndexGlobal = ix * pny * pnz + offset;
      float thetaX = sm_theta[splineIndexX];
      int receiveId = gridIndexGlobal / dataStride;
      spreadVal[receiveId][0] = thetaX * Val;
      spreadVal[receiveId][1] = (float)gridIndexGlobal;
      sendDataToRcvCore(receiveId, sizeof(float) * 2,
        spreadVal[receiveId], remoteVal[cpeId], &rw_rply[cpeId]);
    }
  }
}
```

(b) Auto Translated AHREAD Kernel

Fig. 10 Charge spreading translation

Second, scale affine translation is embedded in the *cptCore* procedure. Third, memory hierarchy optimization is executed by parameter directive. Since parameter variables are *readonly* and size is beyond 128KB, memory type *SLAVE_CACHE* is assigned. CUDA shared memory variables with label *__share__*, only accessed in single thread-block. Hence, these variables are treated as local LDM variable in ATHREAD. At last, CUDA primitive *syncthreads* means all the data in thread-blocks should be updated here. Hence, the thread loop is ended by parsing *syncthreads* and start next thread loop here, as shown in line 20 of Fig. 10b. By our novel algorithm, the core CUDA parallel function *calculate_splines* and *spread_charge* is inherited with high translation efficiency.

5 Evaluation

In this section, we firstly describe the experimental setup and simulation systems. Then, we conduct comprehensive experiments to validate the performance of automatically translated benchmark by *swCUDA*. After that, we evaluate the performance of translated ATHREAD kernels of Gromacs in single core group of Sunway. Finally, we demonstrate the strong scalability by different nodes and particle size.

5.1 Experimental setup

In order to evaluate *swCUDA*, Polybench suite[1] and NBody simulation (Cheng et al. 2014) are used as CUDA input benchmark to demonstrate the productivity and performance. Then, we choose Gromacs 2021.1 stable version as real world application to validate the performance of auto translated kernels by *swCUDA*. All the translated applications are performed on the new generation Sunway supercomputer.

Benchmark application and implementation. As shown in table 2, Polybench suite contains linear algebra solvers, data-mining and stencil (Grauer-Gray et al. 2012), extracted from operations in various application domains and written with different programming languages. NBody simulation contains CUDA implementation for GPU and OpenMP implementation for X86 CPU, which is used to validate the performance of atomic operation. In our experiments, we first use *swCUDA* to automatically translate CUDA kernels to ATHREAD kernels. This implementation is labeled as *TransVer* for comparison. Furthermore, based on the *TransVer*, we manually optimize *readonly* variables with memory type *SLAVE_CACHE* for contiguous accessing order, which effectively enhance cache hit rate of Sunway and improve the

[1] http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/GPU/index.html.

Table 2 Typical parameters of translated benchmark

Code name	Size of array dimension	Number of kernels	Memory type	CUDA code size	Translated code size	Total MFLOPs
2DCONV	4096	1	*DMA_STRIDE*	79	188	436.1
2 MM	2048	2	*DMA_STRIDE*	132	319	35184.6
3DCONV	256	1	*DMA_STRIDE*	75	165	625.4
3 MM	512	3	*GLB_SHR_MEM*	141	345	805.3
ATAX	4096	2	*DMA_STRIDE*	98	262	67.1
BICG	4096	2	*DMA_STRIDE*	106	266	352.3
CORR	2048	4	*DMA_STRIDE*	254	568	8837.1
COVAR	2048	3	*DMA_STRIDE*	131	398	8806.4
DOITGEN	128	2	*DMA_STRIDE*	99	297	538.9
GEMVER	4096	3	*DMA_STRIDE*	166	456	201.3
GESUMMV	4096	1	*DMA_STRIDE*	52	148	67.1
GRAMSCHM	2048	3	*DMA_STRIDE*	142	365	17776.6
MVT	4096	2	*DMA_STRIDE*	88	255	67.1
SYR2K	2048	1	*DMA_STRIDE*	49	147	61583.4
SYRK	1024	1	*GLB_SHR_MEM*	50	122	3307.5
NBody	30720	2	*DMA_STRIDE*	168	493	4308

performance significantly. This version is labeled as *OptVer*. Both *TransVer* and *OptVer* are used for evaluation.

Accuracy verification Since Polybench suite and NBody simulation both support CPU and GPU, they have implemented their own accuracy examination. Polybench compares each value of output matrix and NBody use mean square error (MSE) for examination with preset precision error. We use the same method to evaluate our translated many-core version. All the accuracy examination result shows corrected for different combination: between Sunway MPE and CPE, Intel CPU and Sunway CPE.

Compilation settings. The translated application is compiled with several cross compile toolchains for Sunway platform. Main compiler SWGCC v1307 is vendor-provided tool chain based on GCC 7.1.0. It provides hybrid compilation and link for MPE and CPE program. SWMPI v20220608 is dedicated message passing library for Sunway platform. Special optimization flags for SWGCC are -O3, -msimd, -mieee, -mfma.

Gromacs simulation systems. To better evaluate the performance and generality of our translated ATHREAD kernels, several representative bio-molecular systems are selected from Gromacs official website,[2] ranged from 16k to 3 million atoms as shown in Table 3. Protein RNAse comprises 16k atoms with dodecahedron box. Protein ADH comprises 95k atoms with dodecahedron box. Water-bare-hbonds system has several different simulating sizes. We choose 384k atoms and 3 million atoms systems from it. The RNAse, ADH and *WBH_KB* systems are used to validate the performance for Sunway single core group. The *WBH_M* is used to validate the strong scalability. These MD systems cover major simulating size in practical usage.

Table 3 Gromacs simulation system

Simulation name	Size of atom	Description
RNAse	16k	with dodecahedron box
ADH	95k	with dodecahedron box
WBH_M	3million	water-bare-hbonds with 3 million atoms
WBH_KB	384k	water-bare-hbonds with 384k atoms

5.2 Benchmark performance

In this section, First we evaluate the many-core acceleration of entire ATHREAD kernels of Polybench suites by using version *TransVer* and *OptVer* respectively. NBody simulation can't be used for the evaluation of many-core Acceleration, since Sunway MPE does not support OpenMP. It is used to compare with Intel CPUs and NVIDIA GPU. Then we further conduct comprehensive comparison with Intel CPU and NVIDIA GPU by all translated benchmarks.

5.2.1 Performance of many-core acceleration

All the tests are performed on single core group in new generation Sunway platform. We use sequential CPU implementation as base version of Polybench suites, which is executed on Sunway MPE. As shown in Fig. 11, We get the average 18x speedup for direct translated

[2] https://ftp.gromacs.org/benchmarks/.

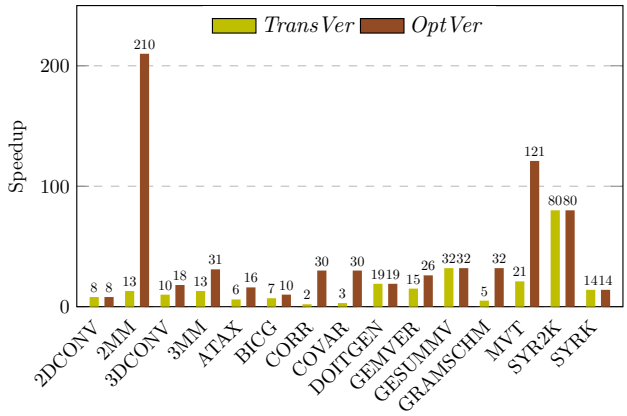


Fig. 11 many-core acceleration for translated Polybench suite

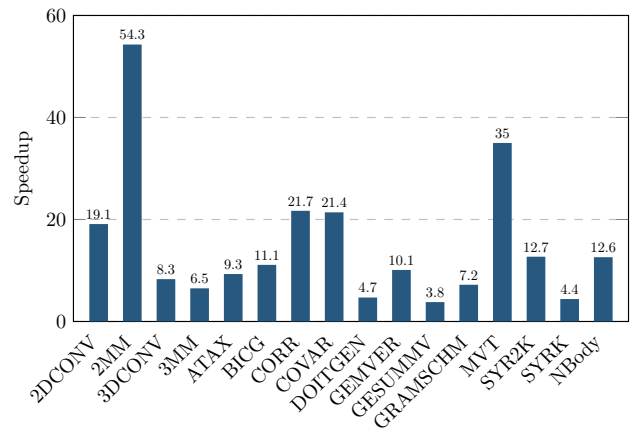


Fig. 12 Performance comparison between sunway CPE and intel CPU

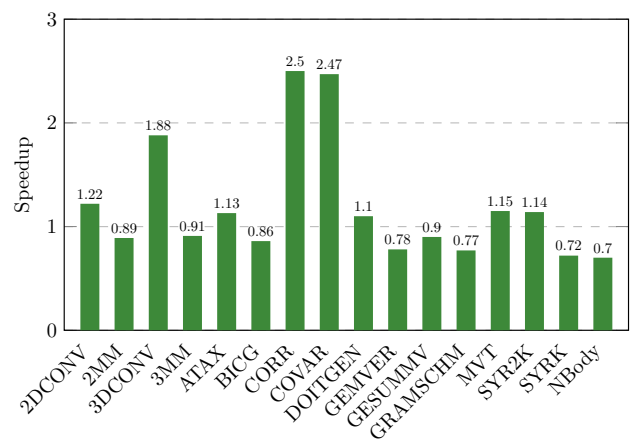


Fig. 13 Performance comparison between sunway CPE and NVIDIA GPU

kernel version *TransVer* of entire Polybench suites and 40x speedup for further optimized version *OptVer*. For the version *TransVer*, the acceleration performance depends on the ratio of compute-to-memory operation. SYR2K achieves the maximum 80x speedup, since its floating point operation (FLOPs) is the biggest, as shown in Table 2. Benefiting from our memory hierarchy optimization, matrix array is effectively transferred to LDM with data stride index for multiply and coefficient operation. On the other hand, CORR achieve the minimum 2x speedup, The reason is *readonly* variable is assigned to *SLAVE_CACHE* but the reading pattern is discrete. CUDA kernels don't need to consider if data accessing order is contiguous or not, since GPU usually have big enough independent memory. This causes that accessing single *readonly* variable by *SLAVE_CACHE* will re-flush cache in Sunway, which decrease the cache hit rate and further decrease performance seriously. Hence, our *OptVer* transpose the matrix for *readonly* variable to make it contiguous accessed. From our experiments, 10 benchmarks adopt this manual optimization and significantly improved the performance.

5.2.2 Comparisons with intel CPU

To compare with CPU, we select Intel® Core I5-8300 CPU at 2.4GHZ with 4 cores as the target hardware. We evaluate the performance of the single process in Sunway and Intel CPU respectively. The executing time of all the benchmark programs run on this target CPU is used as baseline and version *OptVer* on Sunway is used for comparison. Polybench suite can be executed without modification run on Intel CPU, where NBody simulation requires OpenMP multi-thread parallelization strategy in Intel CPU to avoid random access issue. As shown in Fig. 12, version *OtpVer* run on Sunway CPE achieves average 15x speedup than baseline.

For the performance of translated NBody simulation, we get 12x speedup than OpenMP version. The main contribution is that our novel GCAR algorithm effectively eliminates random access issue and asynchronous reduction guarantees the high acceleration performance.

5.2.3 Comparisons with NVIDIA GPU

We select NVIDIA GTX 1050 Ti as the target hardware. Its Theoretical FP32 performance is up to 2.138 TFLOPS, which is two times than single CG of Sunway (1.16 TFLOPS). We evaluate the Polybench suite and NBODY performance on Sunway and NVIDIA GPU respectively.

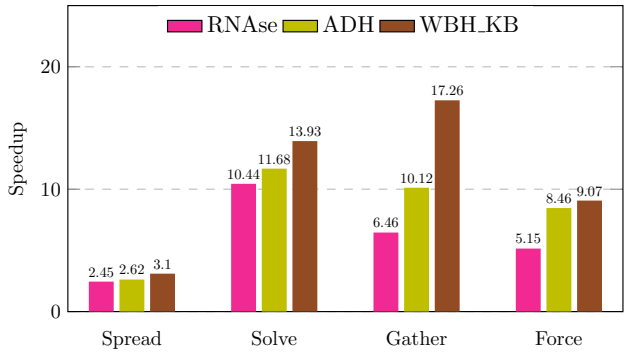


Fig. 14 major kernels acceleration in Gromacs

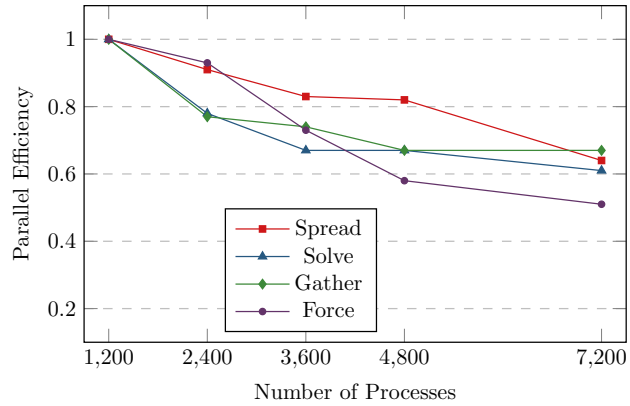


Fig. 15 Strong scalability for water-bare-bonds with 3 million atoms

GPU test result is used as baseline. As shown in Fig. 13, version *OptVer* executed on Sunway platform gets average 27 percentage higher than GPU version. From the test result, there are part of benchmarks which executing time on GPU is a little bit faster than Sunway. The main reason is that these benchmarks are required to access more *readonly* variables by cache and brings lower cache hit rate. Another important reason is that the computing capacity of NIVIDIA GPU is two times faster than Sunway. If we get same computing capacity for single CG of Sunway, the performance will enhance a lot. The overall performance of our optimized version on Sunway is better than NVIDIA GPU.

5.3 Gromacs performance

In this section, We evaluate the performance of major translated kernels in Gromacs by *swCUDA* with different MD system size. First, we present the acceleration performance in single node of Sunway platform. Then, we demonstrate the parallel efficiency of our translated Gromacs.

5.3.1 Single node evaluation

In the experiment, original Gromacs run on Sunway MPE is used as baseline without modification. all the tests are run on a single node with 6 CGs. Figure 14 shows performance speedup for different translated kernels of Gromacs with three different simulation systems. The result shows up to 17x speedup performance. The acceleration performance is improved along the atom size of simulation systems. Hence, *WBH_KB* system gets the maximum speedup than other systems.

Non-bonded force kernel achieves 9x speedup compared with MPE version in *WBH_KB* system. The translated kernel shows up to 80% memory access bandwidth for each super i-cluster interaction, improving the compute to memory access ratio. RNAse system only gets 5x speedup due to its smaller atom size, which decreases memory access band-width.

For PME spread kernel, the spatial randomness of charge griding causes more possibility that computing cores update data to same *rcvCore* by RMA communication. Thereby, the synchronization time of *rcvCore* is longer. Hence PME spread kernel only gets average 2x speedup. PME gather kernel gets more than 10x speedup regardless the difference of atom size, presenting the stable performance improvement by our translation framework.

5.3.2 Scalability

To evaluate the scalability for our translated kernels of Gromacs, we choose *WBH_M* simulating system with 3 million atoms. Figure 15 shows the strong scalability for our translated kernels, using performance of 1200 processes as baseline. As the number of Sunway processes increase from 1200 to 7200, the average parallel efficiency of these four kernels is up to 63%. Gromacs parallel algorithm constructs complex message passing interface (MPI) mechanism, which includes domain decomposition grid and dynamic load balancing (Hess et al. 2008). Each CPE handles less number of particles and the communication time consumption is increased along the increase of processes. Benefiting from our translating optimization for both long-range coulomb force calculation and short-range non-bonded calculation, the performance of load balance is improved between PME processes and non-bonded processes. The result of strong scalability is better than Zhang's work (Zhang et al. 2019).

5.4 Limitations and discussions

The translated kernels by *swCUDA* have proven outstanding performance, but it still has space to improve speedup. Since translated kernels are based on the algorithm of CUDA, they are suitable for SIMD parallelism (Chen et al. 2021). The performance can be speedup if we further optimize them.

Meanwhile, the double buffer technology (Chu et al. 2021) can be leveraged by advanced code modification, which is proven the effectiveness for improving performance. By these optimizations, programmers can further improve the performance based on the translated code framework.

6 Related work

6.1 Code translator

Along with the rapid increase for variable heterogeneous accelerator, the research for programming language auto translation focuses on the two aspects recently: inter translation for heterogeneous low level programming languages (e.g., CUDA, OPENCL) and automatic parallelization for heterogeneous architecture. Gabriel Martinez proposed an AST-Driven auto code translator CU2CL to translate CUDA to OPENCL (Martinez et al. 2011). Since the programming model and hardware architecture has high similarity between these two languages, CU2CL is mainly focusing on the primitive translation, based on the Clang framework. Targeting to Sunway architecture, Li et al. propose a translator from OpenMP to ATHREAD (Li et al. 2021). It maps the thread in OpenMP to single CPE, where parses and translates OpenMP directives. It provides simple memory mapping strategy. The mapping from shared memory to host main memory decreases the performance. For mutual exclusion primitive translation, it proposes master–slave distributed lock mechanism. Since it requires synchronization between host and slave, the overhead is up to 1000 cycles from its data. On the other hand, swCUDA provides auto diverse memory optimization algorithm to effectively assign data to different memory type. Global memory of CUDA is mapping to several memory type depending on different circumstance. Our general write-conflict scheme efficiently resolves the mutual exclusion issue, where the average overhead is only 130 cycles. The closest approach to swCUDA is OPENCL compiler for Sunway (Lee et al. 2010), whereas its key mapping algorithm and memory optimization method is relative fixed and only can handle basic matrix operations. swCUDA provides high level directives based complicated data affine algorithm and memory hierarchy optimization, successfully applied to the multi kernel translations in molecular dynamics simulation software.

Automatic parallelization is used to translate sequential computing intensive codes to paralleling code suitable for heterogeneous architecture. The representative translators are PPCG (Verdoolaege et al. 2013), DawnCC (Mendonça et al. 2017) and HIPAcc (Membarth et al. 2016). PPCG automatically translates static control loop nest to data parallel computation codes based on the polyhedral model of compiler, by implementing multi-level tiling strategy and combined affine transformation. C code fragment is translated to CUDA with memory hierarchy optimization, validated in PolyBench suite. DawnCC is source-to-source compiler, built on top of the LLVM framework. It automatically inserts OpenACC or OpenMP directives in sequential C/C++ code which is interpreted by compilers. Symbolic range and dependence analysis are used for intermediate representation as input to infer the memory boundary of data movement directives. HIPAcc is source-to-source compiler for image processing based on Domain-Specific Languages (DSL). A suite of DSL is designed and embedded in C++. Image processing algorithms written in DSL code can be captured and translated to targeting CUDA, OpenCL and Renderscript. Besides, the memory hierarchy is optimized according predefined feature in DSL. Heuristic for kernel configuration and tiling dependence results in good memory bandwidth utilization. Although these works focus on sequential to parallel code translation, they are profoundly heuristic for methodology to swCUDA.

6.2 Porting MD simulations in sunway

Molecular dynamic softwares, e.g., Gromacs, LAMMPS, Amber and NAMD, are wildly used in versatile heterogeneous architecture of HPC centers. Porting them to Sunway architecture mainly aims to the computing intensive parts: PME method and short-range pair interaction. The PME method is first ported for Gromacs to Sunway recently (LIN Zeng and Jun-shi 2021). To resolve the random memory access issue in charge spreading and force interpolation of PME method, blocking strategy based on local grid order and data reorganization algorithm are addressed. This method redesigns the B-spline algorithm and can't be universal to resolve memory access issue for other applications. On the other hand, *swCUDA* provides general GCAR scheme to resolve memory access issue and completely inherits the parallel codes of B-spline algorithm and force interpolation method. This solution is validated in both charge spreading and short-range interaction kernels. Meanwhile, the accuracy and performance are both guaranteed.

To get better performance of short-range pair interaction in Sunway for LAMMPS, (Dong et al. 2016) proposed double-buffering mechanism to overlaps memory access and calculation time. SIMD vectorization is further used to speed up. But the optimization is only focus on the neighbor list builder and the coulomb force calculation is not related. (Duan et al. 2018) redesign LAMMPS by hybrid memory update strategy to solve write-conflict, software cache strategy to optimize memory bandwidth, customized math functions to eliminate searching lookup tables and pipeline acceleration targeting interactions and neighbor list building. (Yu et al. 2017; Zhang et al. 2019) propose a parallel pipeline model between MPE and CPE clusters to refactor

Gromacs in Sunway. MPE is responsible for task partitioning and data updating, CPE is responsible for calculation. Since force array is updated serially by CPEs, write-conflict is avoided. Moreover, CPEs are divided to scheduling group and computing group to decrease the main-memory access. (Chen et al.. 2021) present a highly efficient short-range force kernel of Gromacs in Sunway with super cluster-based neighbor list. Dual-slice partitioning scheme is addressed to solve the write-conflict issue and reduce the data reduction overhead. SIMD paralleling and instruction reordering is used optimize the performance. All of these works are needed to redesign the paralleling algorithm for original MD software. Meanwhile, memory hierarchy optimization and write-conflict solution are totally different, which can't be leveraged in other applications. All of these draw-backs are overcome by *swCUDA*. Moreover, the previous works only focus on either PME method or short-range pair interaction. Our work is first time to port both of these two sections by unified translating interface, which is productive.

7 Conclusion

In this paper, we present a novel parallel code transformation framework *swCUDA*, greatly enhancing the efficiency of ATHREAD programming. In this framework, three fundamental algorithms are presented. The algorithm of scaled data affine translation provides the transformation from CUDA primitive of thread-blocks and thread to ATHREAD index, the foundation for CUDA code translation. The GCAR algorithm resolves the central issue towards to the hardware drawback in Sunway CPE. The algorithm of memory hierarchy optimization automatically assigns the appropriate memory usage for variable with different size to improve the compute to memory access ratio. Based on these algorithms, we design concise high level directives to enhance the translation efficiency. By these effective method, *swCUDA* expresses strong productivity and applicability.

To validate the practicability and performance of *swCUDA*, we conduct comprehensive experiments under entire Polybench benchmark suite and NBody simulation. When compared to the baseline, multiple benchmarks achieve an average speedup of 18x. When compared with Intel CPU, we get an average 3x speedup. Further, we choose molecular dynamics simulation Gromacs as real world application. Towards to complicated CUDA kernels of PME method and short-range pair list interaction, *swCUDA* successfully translates the CUDA codes to ATHREAD codes. Experiments show up to up to 17x speedup performance and average 63% scalability.

As a part of future work, we plan to extend the capabilities of *swCUDA* to support more parallel architecture conversions, such as Heterogeneous Interface for Portability (HIP) programming for AMD GPU platforms. With the exception of a few functions that are not commonly used, HIP copies almost the entire CUDA API, including function names, keywords, etc. In most cases, CUDA-to-HIP portability can be accomplished at the source level by substituting 'cuda' and 'hip' characters. After supporting HIP translation with *swCUDA*, programs on more extensive hardware platforms will be able to quickly translate to run on Sunway platform and improve programming productivity.

Acknowledgements This work was supported in part by National Key Research and Development Program of China (Grant No. 2021YFF0704000). The corresponding author are Dongning Jia (email: dnjia@qnlm.ac) and Zhiqiang Wei (email: weizhiqiang@ouc.edu.cn).

Data availability Furthermore, new data availability algorithm is considered as the main research interests to improve the speedup for translated codes.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

Chen, J.S., An, H., Han, W.T., et al.: Towards efficient short-range pair interaction on sunway many-core architecture. J. Comput. Sci. Technol. **36**(1), 123–139 (2021). https://doi.org/10.1007/s11390-020-9826-z

Cheng, J., Grossman, M., McKercher, T.: Professional CUDA C Programming, 1st edn. Wrox Press Ltd (2014)

Chu, G., Li, Y., Zhao, R.: et al Md simulation of hundred-billion-metal-atom cascade collision on sunway taihulight. ArXiv (2021) https://arxiv.org/abs/2107.07866

Dong, W., Kang, L., Quan, Z.: et al Implementing molecular dynamics simulation on sunway taihulight system. In: 2016 IEEE 18th International Conference on High Performance Computing and Communications. In: IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp 443–450, https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0070 (2016)

Duan, X., Gao, P., Zhang, T.: et al. Redesigning lammps for peta-scale and hundred-billion-atom simulation on sunway taihulight. In: SC18: International Conference for High Performance Computing Networking, Storage and Analysis Doi: https://doi.org/10.1109/SC.2018.00015(2018)

Essmann, U., Perera, L., Berkowitz, M., et al.: A smooth particle mesh ewald method. J. Chem. Phys. **103**, 8577 (1995). https://doi.org/10.1063/1.470117

Fu, H., Liao, J., Yang, J., et al.: The sunway taihulight supercomputer: system and applications. Sci. China Informat. Sci. **59**, 1–16 (2016). https://doi.org/10.1007/s11432-016-5588-7

Garland, M., Le Grand, S., Nickolls, J., et al.: Parallel computing experiences with cuda. IEEE Micro **28**(4), 13–27 (2008). https://doi.org/10.1109/MM.2008.57

Grauer-Gray, S., Xu, L., Searles, R. et al.: Auto-tuning a high-level language targeted to gpu codes. In: 2012 Innovative Parallel Computing (InPar), pp 1–10, https://doi.org/10.1109/InPar.2012.6339595 (2012)

Han, T.D., Abdelrahman, T.S.: hicuda: High-level gpgpu programming. IEEE Transact. Parall. Distribut. Syst. **22**(1), 78–90 (2011). https://doi.org/10.1109/TPDS.2010.62

Harvey, M., De Fabritiis, G.: An implementation of the smooth particle mesh ewald method on gpu hardware. J. Chem. Theory Comput. (2009). https://doi.org/10.1021/ct900275y

Hess, B., Kutzner, C., van der Spoel, D., et al.: Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. J. Chem. Theory Comput. **4**(3), 435–447 (2008). https://doi.org/10.1021/ct700301q

Jing, S., Li, X., Liu, Z., et al.: Gpu-enabled implementations of particle-mesh-ewald method. Comp. Appl. Chem. (2012). https://doi.org/10.1021/acs.jctc.0c00744

Kutzner, C.: Improving pme on distributed computer systems. (2008) https://www.mpinat.mpg.de/632110/kutzner08talk-workshop.pdf

Kutzner, C., Páll, S., Fechner, M.: More bang for your buck Improved use of gpu nodes for gromacs 2018. J. Comput. Chem. **40**, 2418–2431 (2019). https://doi.org/10.48550/arXiv.1903.05918

Lee, J., Kim, J., Seo, S et al.: (2010) An opencl framework for heterogeneous multicores with local memory. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. Association for Computing Machinery, New York, NY, USA, PACT '10, p 193-204, (2010) https://doi.org/10.1145/1854273.1854301

Lee, S., Wolberg, G., Shin, S.: Scattered data interpolation with multilevel b-splines. IEEE Transact. Visualizat. Comp. Graph. **3**(3), 228–244 (1997). https://doi.org/10.1109/2945.620490

Li, M., Pang, J., Yue, F. et al.: Openmp automatic translation framework for sunway taihulight. In: 2021 International Conference on Communications, Information System and Computer Engineering (CISCE) (2021) Doi: https://doi.org/10.1109/CISCE52179.2021.9445916

Liu, F., Ma, W., Zhao, Yea.: xmath2.0: a high-performance extended math library for sw26010-pro many-core processor. CCF Transactions on High Performance Computing pp 2524–4930. (2022) https://doi.org/10.1007/s42514-022-00126-8

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. Association for Computing Machinery, New York, NY, USA, SC '21, (2021) https://doi.org/10.1145/3458817.3487399

Martinez, G., Gardner, M., Feng, Wc.: Cu2cl: A cuda-to-opencl translator for multi- and many-core architectures. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems, pp 300–307, (2011) https://doi.org/10.1109/ICPADS.2011.48

Membarth, R., Reiche, O., Hannig, F., et al.: Hipacc: a domain-specific language and compiler for image processing. IEEE Transact. Parall. Distribut. Syst. **27**(1), 210–224 (2016). https://doi.org/10.1109/TPDS.2015.2394802

Mendonça, G., Guimarães, B.: Dawncc: Automatic annotation for data parallelism and offloading. ACM Trans. Archit. Code Optim. (2017). https://doi.org/10.1145/3084540

Milakov, M.: Gpu pro tip: Fast dynamic indexing of private arrays in cuda. https://developer.nvidia.com/blog/fast-dynamic-indexing-private-arrays-cuda/ (2015)

Nvidia, C.: Gpu-accelerated applications. (2018) https://www.nvidia.cn/content/gpu-applications/PDF/gpu-applications-catalog.pdf

Nvidia, C.: Nvidia v100 tensor core gpu. (2020) https://images.nvidia.cn/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf

Nvidia, C.: Cuda c++ programming guide. (2023) https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Parr, T.: The definitive antlr 4 reference. The Definitive ANTLR 4 Reference pp 1–326 (2013)

Parr, T., Fisher, K.: Ll(*): The foundation of the antlr parser generator. SIGPLAN Not **46**(6), 425–436 (2011). https://doi.org/10.1145/1993316.1993548

Parr, T., Harwell, S., Fisher, K.: Adaptive ll(*) parsing: the power of dynamic analysis. SIGPLAN Not **49**(10), 579–598 (2014). https://doi.org/10.1145/2714064.2660202

Shang, H., Li, F., Zhang, Y. et al.: Extreme-scale ab initio quantum raman spectra simulations on the leadership hpc system in china. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Association for Computing Machinery, New York, NY, USA, SC '21, (2021) https://doi.org/10.1145/3458817.3487402

Strohmaier, E., Dongarra, J., Simon, H. et al.: Top 500 supercomputer lists. https://top500.org/ (2022)

Verdoolaege, S., Carlos Juega, J., Cohen, A.: Polyhedral parallel code generation for cuda. ACM Trans. Archit. Code Optim. **10**(1145/2400682), 2400713 (2013)

Yu, Y., An, H., Chen, J. et al.: Pipelining computation and optimization strategies for scaling gromacs on the sunway many-core processor. In: Ibrahim, S., Choo, K.K.R., Yan, Z., et al. (eds.) Algorithms and Architectures for Parallel Processing, pp. 18–32. Springer International Publishing, Cham (2017)

Zeng, L., Zheng, W., Hong, A.: Porting and optimizing pme algorithm on sunway taihulight system. J. Chin. Comp. Syst. **42**(1), 9 (2021)

Zeng, L.I.N., Zheng, A.H.W.U., Jun-shi, C.: Porting and optimizing pme algorithm on sunway taihulight system. J. Chin. Comp. Syst. **42**(1), 9 (2021)

Zhang, T., Li, Y., Gao, P. et al.: Sw_gromacs: Accelerate gromacs on sunway taihulight. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. Association for Computing Machinery, New York, NY, USA, SC '19, (2019) https://doi.org/10.1145/3295500.3356190

Zhu, Q., Luo, H., Yang, C. et al.: Enabling and scaling the hpcg benchmark on the newest generation sunway supercomputer with 42 million heterogeneous cores. In: SC21: International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–13, (2021) https://doi.org/10.1145/3458817.3476158

Maoxue Yu is senior engineer in Network and Information Center of Qingdao Marine Science and Technology Center. His research interests include the high performance computing and auto parallelization.

Guanghao Ma is senior engineer in Network and Information Center of Qingdao Marine Science and Technology Center. His research interests include the high performance computing and scientific visualization.

Zhuoya Wang is engineer in Network and Information Center of Qingdao Marine Science and Technology Center. Her research interests include the Computer Aided Drug Design and AI-Driven Drug Design.

Shuai Tang is postgraduate in the faculty of Computer Science and Technology of the Ocean University of china. His research interests include high performance computing and auto parallelization.

Yuhu Chen is engineer in Network and Information Center of the Qingdao Marine Science and Technology Center. His research interests include HPC and global climate model simulation

Yucheng Wang is senior engineer in Network and Information Center of Qingdao Marine Science and Technology Center. He received PH. D degree in Ehime University, Japan. His research interests include big data analytics and regional ocean modeling simulation.

Yuanyuan Liu is an engineer at Network and Information Center of Qingdao Marine Science and Technology Center. Her research interests include the high performance computing and bigdata.

Dongning Jia is professor in Faculty of Information Science and Engineering, Ocean University of China and director in Network and Information Center of Qingdao Marine Science and Technology Center. He received PH. D degree in Ocean University of China. His main research interests include pattern recognition, parallel computing and machine learning.

Zhiqiang Wei received the Ph.D. degree from Tsinghua University, China, in 2001. He is currently a Professor with the Ocean University of China. His current research interests are in the fields of intelligent information processing, social media, and big data analytics.