INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING
ICPP/2021/CHICAGO/USA
acm In-Cooperation
sighpc
AUGUST 9-12, 2021
HiPO

Automatic Code Generation and Optimization of Large-scale Stencil Computation on Many-core Processors

Mingzhen Li, Yi Liu, Hailong Yang, Yongmin Hu, Qingxiao Sun, Bangduo Chen,
Xin You, Xiaoyan Liu, Zhongzhi Luan and Depei Qian
Beihang University, Beijing, China

北京航空航天大學
BEIHANG UNIVERSITY

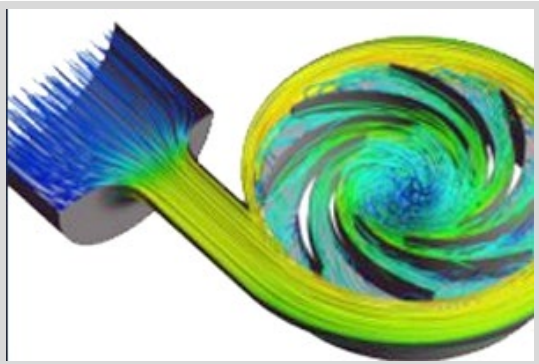
Outline

- **Background & Motivation**
  - **Stencil Computation & Optimizations**
  - **Emerging Many-core Processors**
  - **Motivation**
- Methodology & Implementation
  - Domain Specific Language
  - Compilation Optimizations
  - Communication Library
- Evaluation
  - Experiment Setup
  - Performance Analysis
- Conclusion

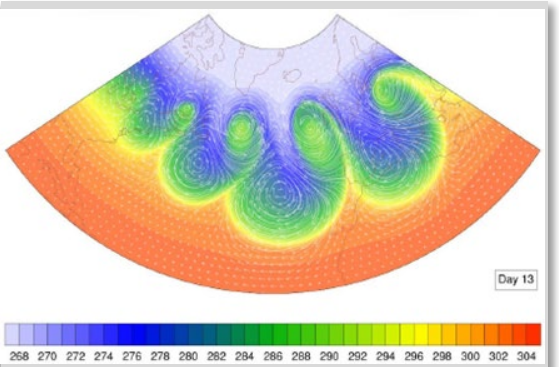
Stencil Computation

- A stencil defines a particular computation pattern on the structural grid.
  - (Spatial) It updates each element based on certain neighboring elements.
  - (Temporal) It updates the values of current timestep based on previous timesteps.

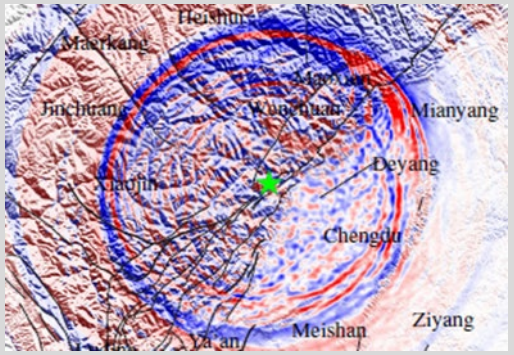
$$u_{i,j}^{t+1} = c_0\, u_{i,j}^{t-1} + c_1\, u_{i,j}^{t} + c_2\,(u_{i+1,j}^{t} + u_{i-1,j}^{t}) + c_3\,(u_{i,j+1}^{t} + u_{i,j-1}^{t})$$



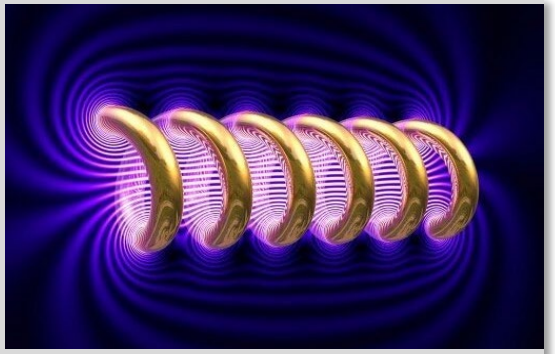
CFD



Climate Modeling



Seismic Processing



Electromagnetism

Real-world problems → **1) PDE equations** **2) Discretization** **3) Stencil computations**

Stencil Optimizations

- Diverse stencil patterns
  - Grid dimensions (e.g., 2D, 3D)
  - Shapes (e.g., box, star)
  - Number of neighbors (e.g., 7-point, 27-point)
  - Number of timesteps
- Performance optimizations
  - **Tiling**: overlapped [Zhou et. al, CGO12], trapezoid [Frigo et. al, SC05], diamond [Bertolacci et. al, SC15] …
  - **Streaming** [Nguyen et.al, SC10]
  - **Vectorization** [Henretty et.al, ICS13] …

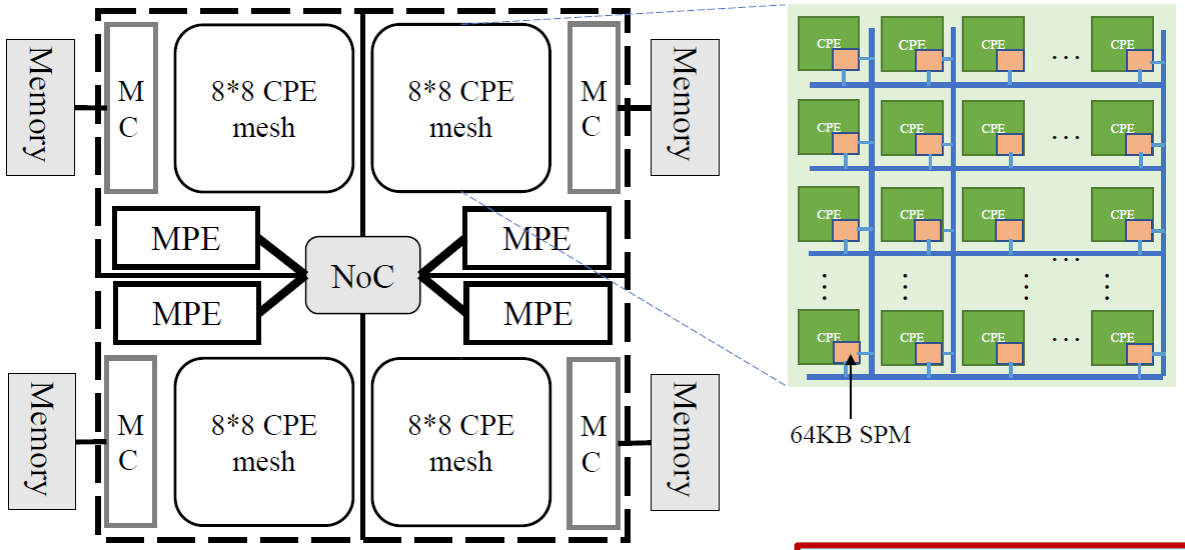
Manual Optimizations are tedious and error-prone

- Stencil DSLs & compilers
  - (INPUT) the stencil definitions described by domain specific languages
  - (OUPUT) the optimized codes (or binaries) on target hardware
  - Code transformations (optimizations)
    - tiling, streaming, vectorization
  - Representative stencil DSLs
    - Halide [Ragan et.al, PLDI13] [Denniston, et.al, PPoPP16]
    - Physis [Maruyama,, et.al, SC11]

- Lacking support for large-scale execution
- Lacking support for stencils with multiple time dependencies

Emerging Many-core Processors

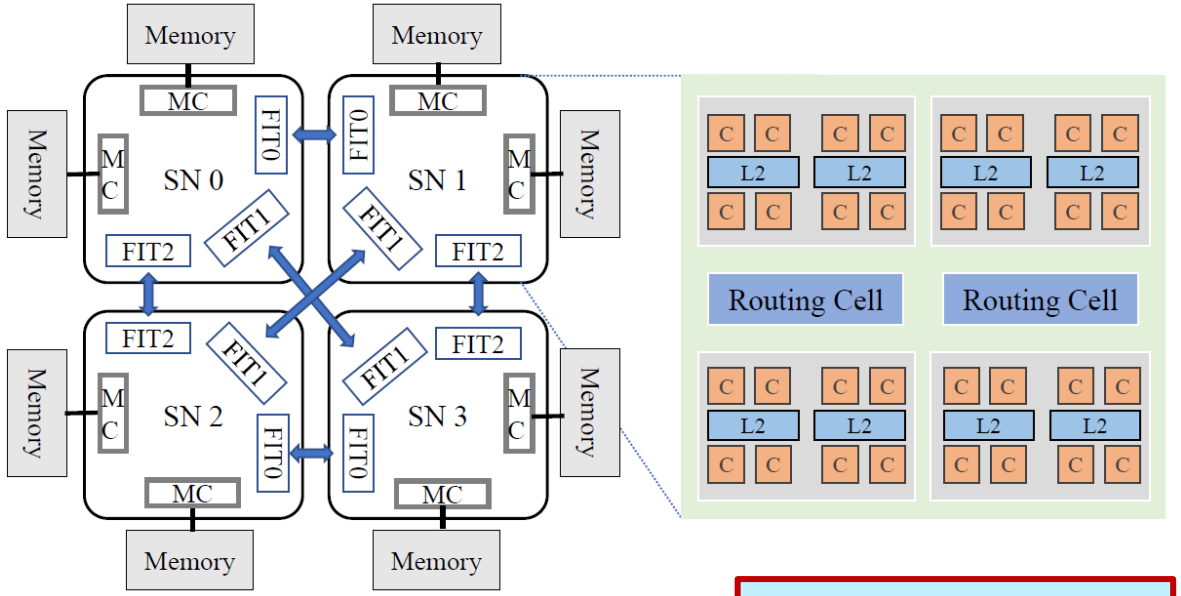
- Posing new challenges for stencil DSLs with diverse architecture designs



Sunway SW26010 processor
= Core Group * 4
= (MPE*1 + CPE*64) * 4

Heterogeneous architecture

- **64KB SPM, manually controlled**
- **Direct Memory Access (DMA)**
- **Programming model: *Athread***



Matrix MT2000+ processor
= SuperNode (32 cores) * 4

Homogeneous architecture

- **SN = panel (8 cores) * 4**
- **Each panel contains 8 cache-coherent compute cores.**
- **Programming model: *OpenMP***

Motivation

In general, existing stencil DSLs

- Lack support for emerging manycore processors (Sunway and Matrix)
- Focus on expressing and optimizing stencils on the spatial dimension
- Few (except YASK [Yount et.al, WOLFHPC16], Physis [Maruyama et.al, SC11], and STELLA [Gysi et.al, SC15]) can optimize stencils at large scale

We propose a new stencil DSL, *MSC*

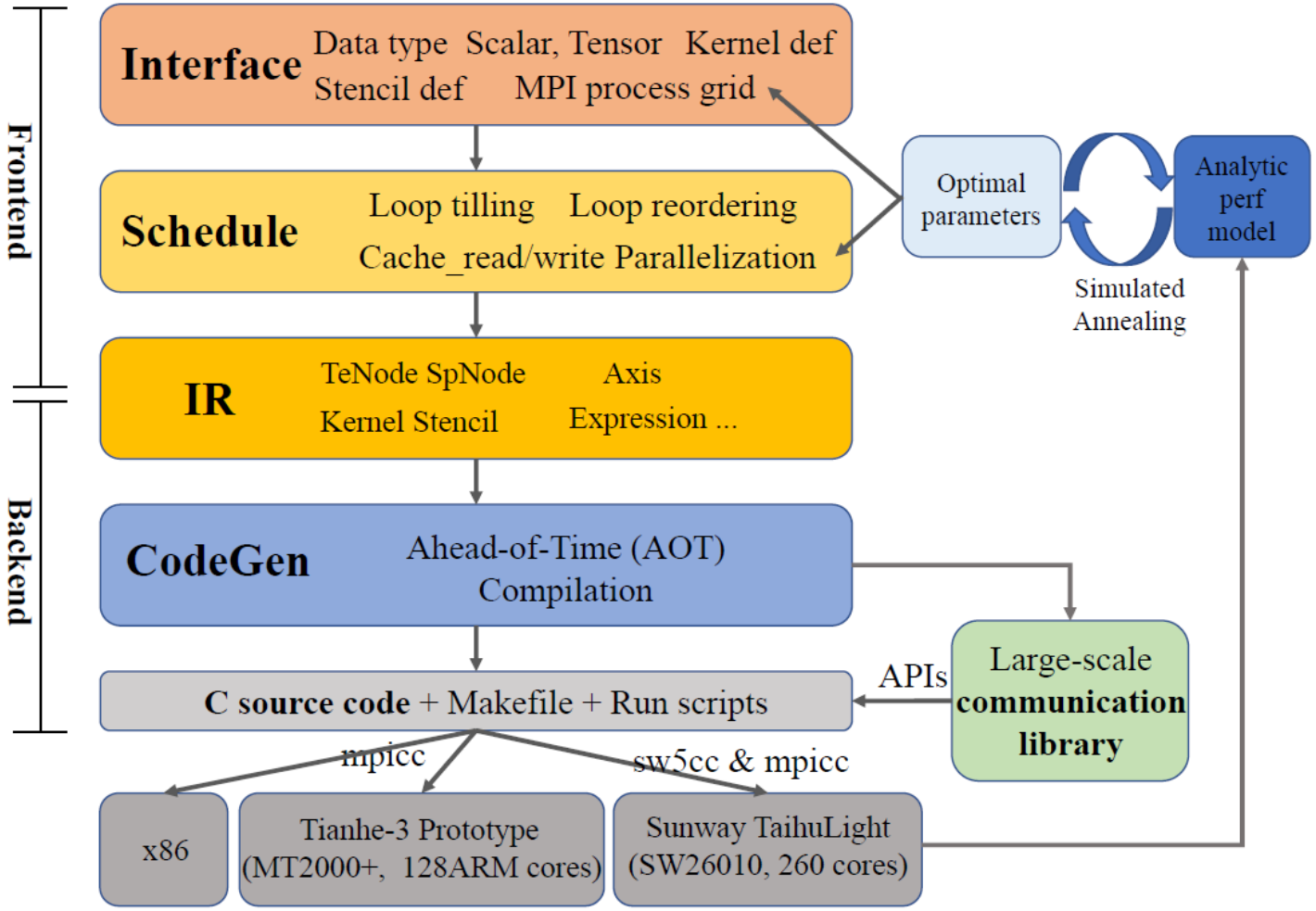
- adapt optimization passes tailored for many-core processors with support of multiple time dependency
- optimized to support the halo exchange for large-scale stencil computation

		MSC	Halide [30]	Pluto [5]	Tiramisu [3]	Patus [7]	Artemis [32]	YASK [40]	STELLA [18]	Physis [27]	OPS [33]	Devito [25]	Lift [19]	AN5D [28]	Polly [15]	Pochoir [36]	Loo.py [23]
Stencil	Single timestep	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Multiple timestep	✓					✓		✓			✓			✓		✓
Hardware	CPU	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓
	GPU		✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓
	Manycore	✓															
Optimization	Spatial tiling	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
	Streaming						✓				✓			✓			
	Temporal tiling			✓	✓						✓			✓	✓	✓	
	Auto-tuning	✓	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	
Distributed	Halo exchange	✓	✓ [8]		✓			✓	✓	✓	✓	✓					
	Pluggable library	✓	✓ [8]														

Outline

- Background & Motivation
  - Stencil Computation & Optimizations
  - Emerging Many-core Processors
  - Motivation
- **Methodology & Implementation**
  - **Domain Specific Language**
  - **Compilation Optimizations**
  - **Communication Library**
- Evaluation
  - Experiment Setup
  - Performance Analysis
- Conclusion

Design Overview



- Frontend
  - Stencil pattern, time iteration
  - Optimization primitives
- Backend
  - Ahead-Of-Time (AOT) compilation
  - Generate standard C codes
- Intermediate Representation (IR)
  - Lower expression to implementation
- Communication library
  - Enabling large-scale execution with flexibility and extensibility

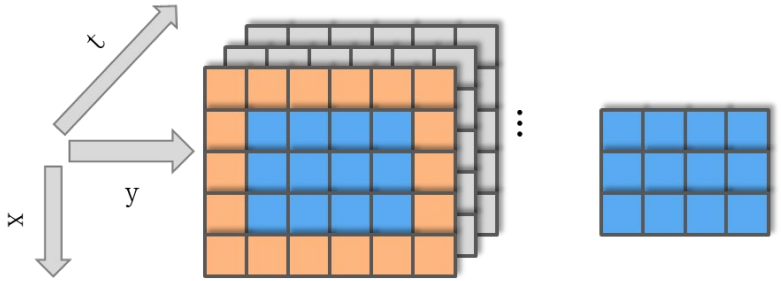
Our design principle – Separating stencil expression from optimization

Intermediate Representation of MSC

Type	Nodes	Description
Tensor	- SpNode - TeNode	Tensor w/i halo region Tensor w/o halo region
Nested loop	- Axis	Axis of nested loops
Expression	- AssignExpr - OperatorExpr - CallFuncExpr - IndexExpr	Value assignment expr. Unary / Binary expr. External function call expr. Index calculation expr.
Kernel	-	Basic stencil kernel
Stencil	-	Stencil with multiple time dependencies
Primitive	- tile, reorder, parallel - cache_read/write - compute_at	Optimization passes which rewrite the IR.

- A single level IR embedded in the abstract syntax tree.
- Kernel:
  - Composed of Tensor, Nested loop, and Expression IR
- Stencil:
  - Composed of Kernel, Tensor, Nested loop, and Expression IR
- Primitive:
  - Rewrite the Nested loop and Expression IR in Kernel

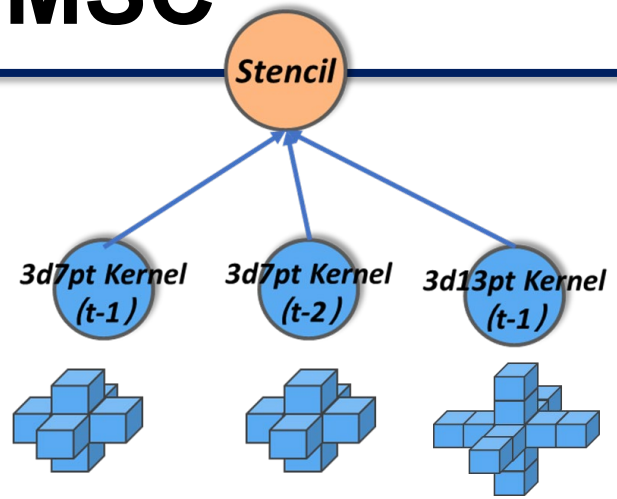
Intermediate Representation of MSC



2D SpNode

2D TeNode

- Two kinds of tensors in MSC
- ***SpNode***
  - Explicitly defined by users
  - With extra memory space to store the halo regions and the intermediate data within the time window.
- ***TeNode***
  - Implicitly used by the MSC compiler and is transparent to users
  - As a temporary buffer, without halo



- Separation of ***Kernels*** and ***Stencils***
- ***Kernel*** (within single timestep)
  - Element *(k, j, i)* is updated using its neighboring elements. E.g., *3d7pt*.
  - MSC provides various optimization primitives
- ***Stencil*** (with multiple timesteps)
  - *Stencil* aggregates the output of the kernels at different timesteps.

A ***Stencil*** can consist **multiple different *Kernels*** from **different timesteps**.

Programming Language of MSC

```
1.  ...
2.  const int M = N = P = 256;
3.  const int halo_width = 1;
4.  const int time_window_size = 2;
5.  DefVar(k, i32); DefVar(j, i32); DefVar(i, i32);
6.  DefTensor3D_TimeWin(B, time_window_size, halo_width, f64, 256, 256
7.  Kernel S_3d7pt((k,j,i), c0*B[k,j,i] + c1*B[k,j,i-1] + c2*B[k,j,i+1] + c3*B[k-
8.  // Optimizations
9.  ... Several optimization primitives
10. auto t = Stencil::t;
11. Result Res( (i,j) , B[i,j] );
12. Stencil st((i,j), Res[t] << S_3d7pt[t-1] + S_3d7pt[t-2]);
13. DefShapeMPI3D(shape_mpi, 4, 4, 4)
14. st.input(shape_mpi, B, "../data/rand.data");
15. st.run(1,10);
16. st.compile_to_source_code("3d7pt");
```

Line 2: the dimension (256^3) of input/output grid

Line 3-4: halo region width, time window size

Line 5: subscripts of the elements in the grid

Line 6: the input 3D **tensor** B

Line 7: 3d7pt stencil ***kernel***, element (k, j, i) is updated using six neighboring elements

Line 8-9: various optimization primitives

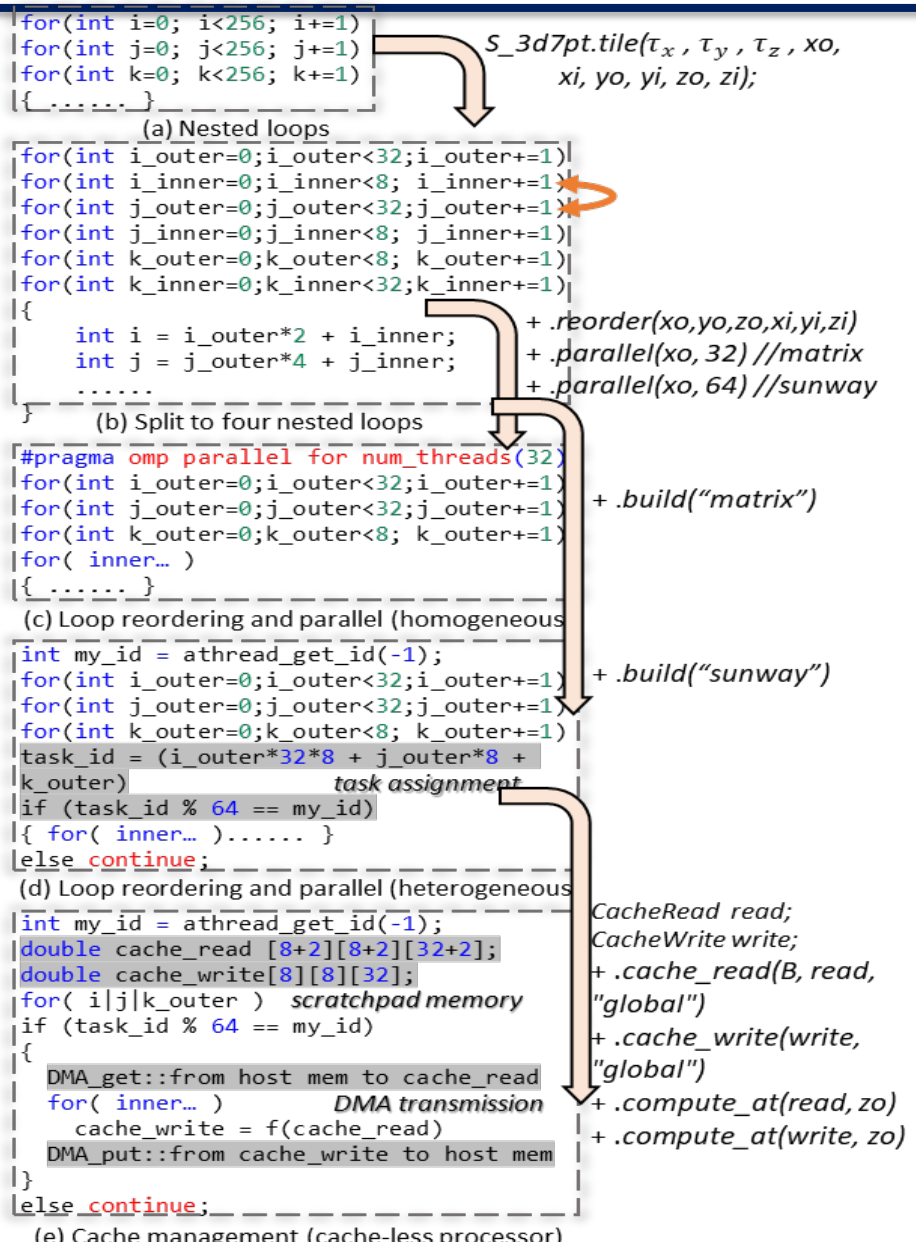
Line 12: ***stencil*** computation along the time dimension, which aggregates the output at timestep (t – 1) and (t – 2)

Line 13: MPI grid for large-scale execution

Line 14-15: input data, time iterations

Line 16: optimize, compile, and codegen

Compilation Optimizations – Overview



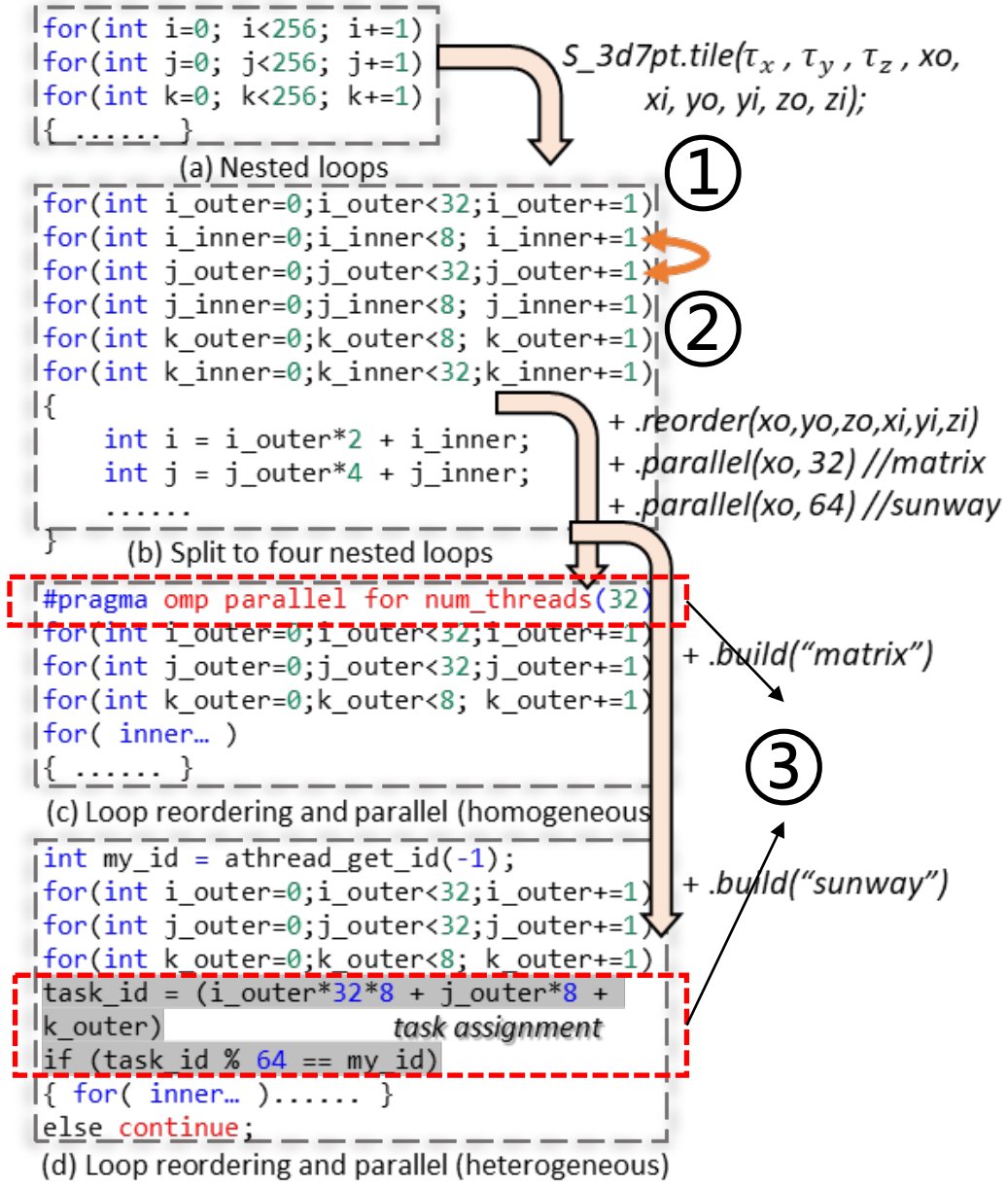
Architecture independent

- *Tile*: loop fission in all dimensions
- *Reorder*: reorder the nested loops
- *Parallel*: map the loops to cores

- Caching related primitives
  - *CacheRead* and *CacheWrite*
  - *cache_read* and *cache_write*
  - *compute_at*

Architecture dependent

Compilation Optimizations – Architecture independent



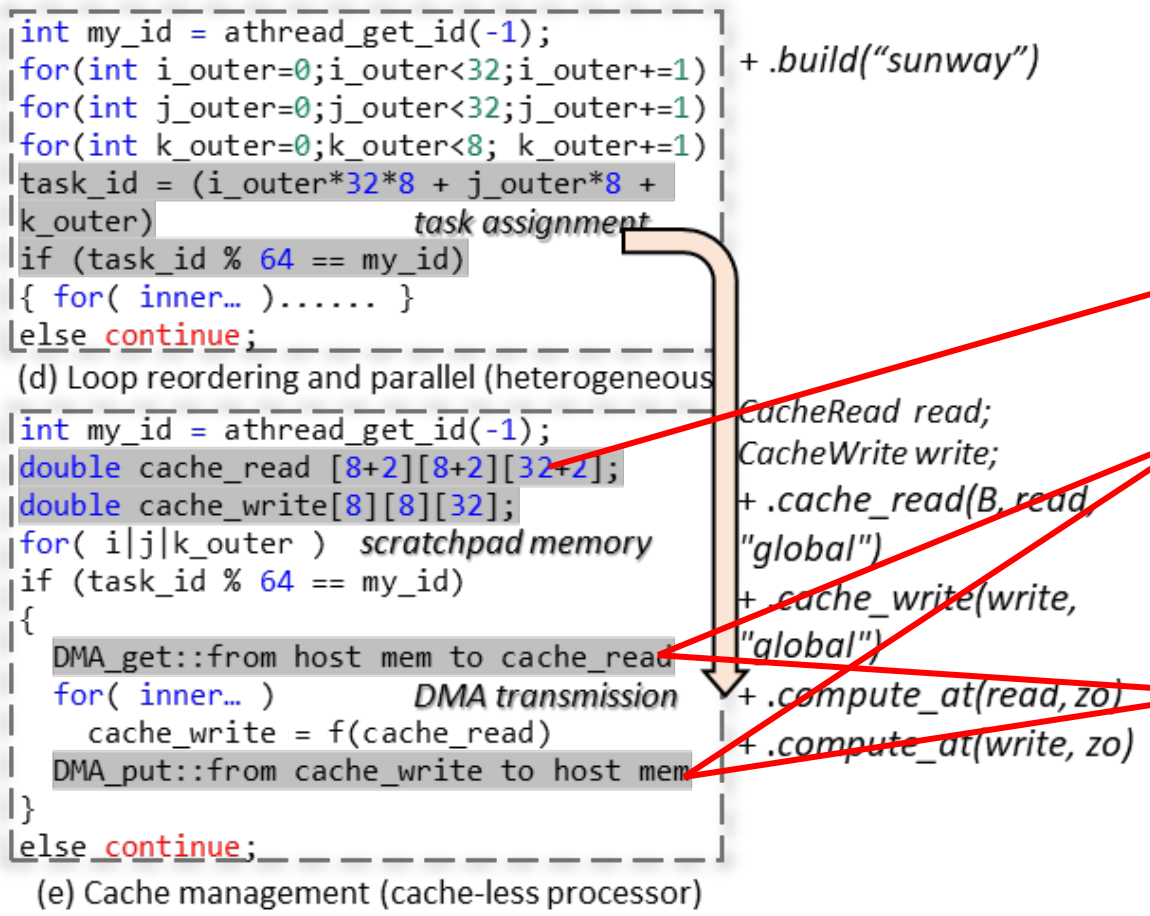
- *Tile* ① + *Reorder* ②

- Together, they split the stencil computation into **a sequence of computation tasks on tiles**.
- The tiles are assigned with overlapped halo regions to avoid computation dependencies.

- *+ Parallel* ③

- The tasks can be **mapped to the massive cores** of the many-core processors conveniently.
- (CPEs of Sunway processor, and compute cores of Matrix processor)

Compilation Optimizations – Architecture dependent



Explicit control the data access to utilize the fast local memory on cache-less processors.

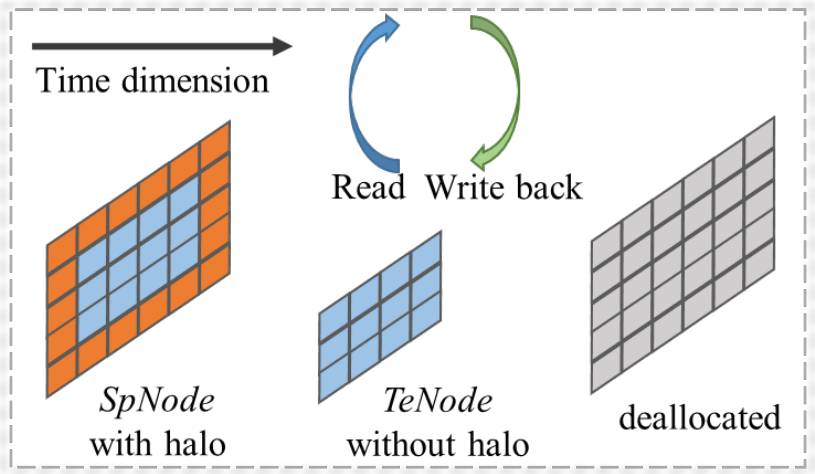
- **Caching primitives**
- *CacheRead* and *CacheWrite*
  - Allocate read/write buffers in local memory.
- *cache_read* and *cache_write*
  - Bind the input/output tensor to the read/write buffer.
- *compute_at*
  - Dictate the DMA data transfer:
  - *1)* the data to be transferred
  - *2)* the code position to invoke DMA

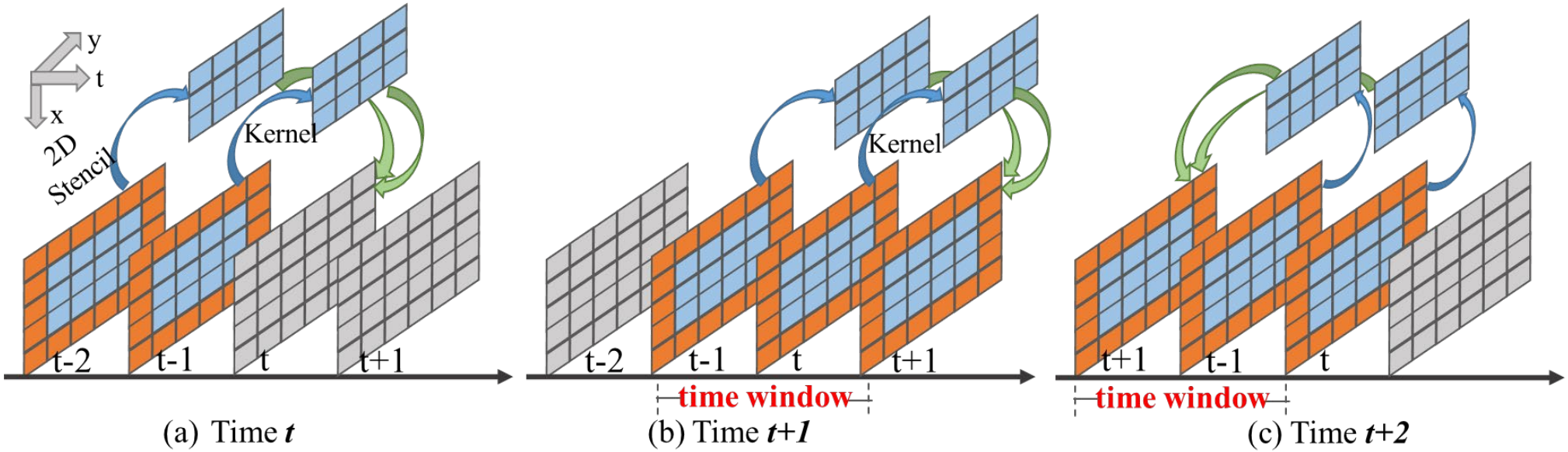
- Control the allocation of local memory (e.g., SPM on Sunway) for **better data reuse**.
- Manage the DMA transfer between local memory and main memory **automatically**.

Compilation Optimizations – Sliding time window

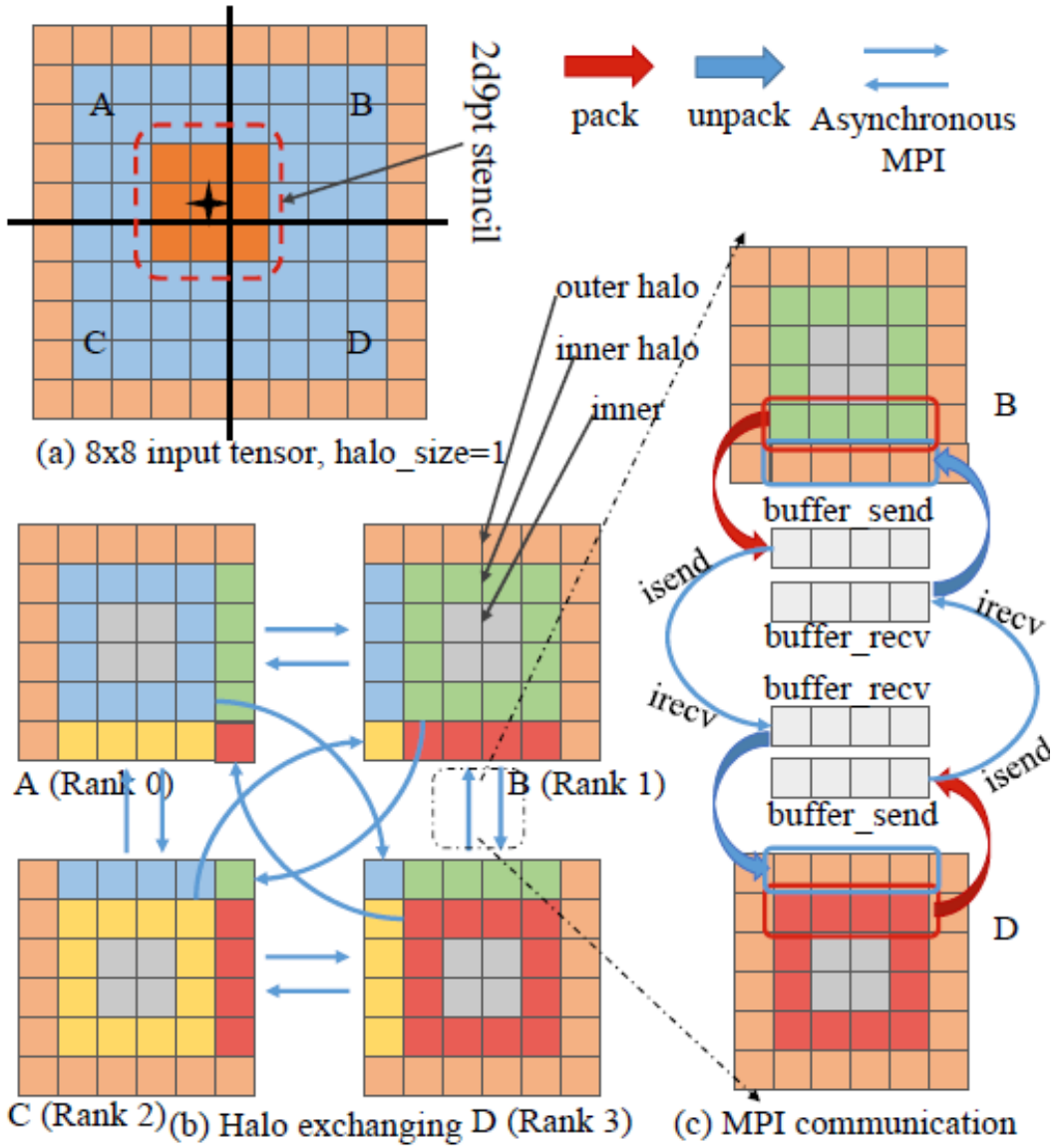
Enable iterating over a large number of timesteps

- Restrict the size of intermediate tensors to the window size (e.g., 3)
- By preempting the buffer of the oldest tensor, and assigning it to the new tensor





Communication Library



- Domain decomposition
  - One sub-tensor for one MPI process
  - Outer halo (orange color) / Inner halo (green color) / Inner (grey color)
- Halo exchange
  - Allocate the memory for the send buffer and the receive buffer
  - Then pack the data of the inner halo region in the send buffer
  - Then call MPI_isend to send the packed data to the neighboring MPI process
  - Call MPI_irecv and then unpack
  - Notably, all MPI processes are exchanging the halo region simultaneously
- Autotuning
  - Select the optimal sub-tensor size

Communication Library

```
58 float TeNode1[256][256][256];
59 void func2(int (var21) , int (var22) ) {
60 for( int  (var20) =  (var21) ;  (var20) <=  (var22) ;  (var20) +=1 ) {
61  func1( ( (var20) + (-1) ) , TeNode1 );
62  exchange_halo_3D_float (  &SpNode0[ (var20) % 2 ][0][0][0],
63                            0, 256, 256, 256, 1, mpi_rank,
64                            1, 1, 1, mpi_size ) ;
65 }
66 }
67 void func3() {
68  InputData_float (  &SpNode0[0][0][0][0], 34347024, "../data/rand.data" )
69  exchange_halo_3D_float (  &SpNode0[ (0) % 2 ][0][0][0],
70                            0, 256, 256, 256, 1, mpi_rank,
71                            1, 1, 1, mpi_size ) ;
72  exchange_halo_3D_float (  &SpNode0[ (1) % 2 ][0][0][0],
73                            0, 256, 256, 256, 1, mpi_rank,
74                            1, 1, 1, mpi_size ) ;
75 }
76 void func4() {
77  func2( (1) , (100) );
78 }
```

- Seamlessly integrated:
  - MSC can insert the function calls in the generated codes automatically.
- Pluggable:
  - It works as a plugin to MSC.
  - Users can easily plug in their own halo-exchanging libraries following the same api.
- Extensible:
  - Various communication optimizations can be further implemented without modifying MSC.

Outline

- Background & Motivation
  - Stencil Computation & Optimizations
  - Emerging Many-core Processors
  - Motivation
- Methodology & Implementation
  - Domain Specific Language
  - Compilation Optimizations
  - Communication Library
- **Evaluation**
  - **Experiment Setup**
  - **Performance Analysis**
- Conclusion

Experiment Setup

Hardware & software configuration.

Platform	Processor	Compiler	MPI	OpenMP
Sunway TaihuLight	SW26010 (65 cores*4)	gcc-8.3 sw5cc	mpich-3.0	None
Tianhe-3 Prototype	MT2000+ (32 cores)	gcc-8.2	mpich-3.2	4.5
Local CPU Server	E5-2680v4*2 (14 cores*2)	gcc-8.3	openmpi-3.1	4.5

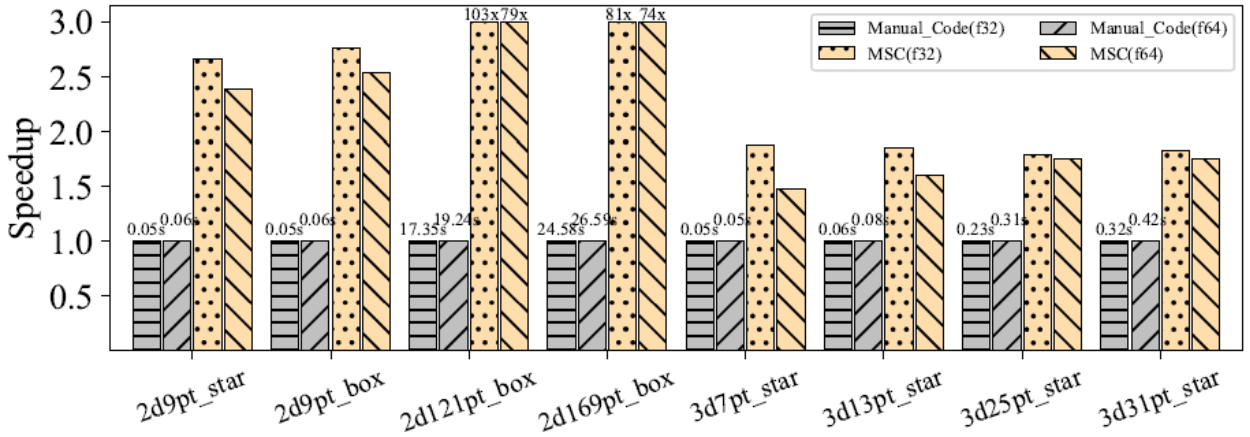
Stencil benchmarks used in the evaluation.

Benchmark	Read(Byte)	Write(Byte)	Ops(+-×...)	Time Dep.
2d9pt_star	72	8	17	2
2d9pt_box	72	8	17	2
2d121pt_box	968	8	231	2
2d169pt_box	1352	8	325	2
3d7pt_star	56	8	13	2
3d13pt_star	104	8	17	2
3d25pt_star	200	8	41	2
3d31pt_star	248	8	50	2

- Performance of MSC on a single many-core processor.
  - Baseline: serial version.
  - Comparison: OpenACC (on Sunway), OpenMP (on Matrix). Both adopt the **same optimizations** as MSC for a fair comparison.
- Weak and strong scalability.
- Performance comparison with SOTA DSLs on x86 CPU
  - Comparison: Halide, Patus, Physis

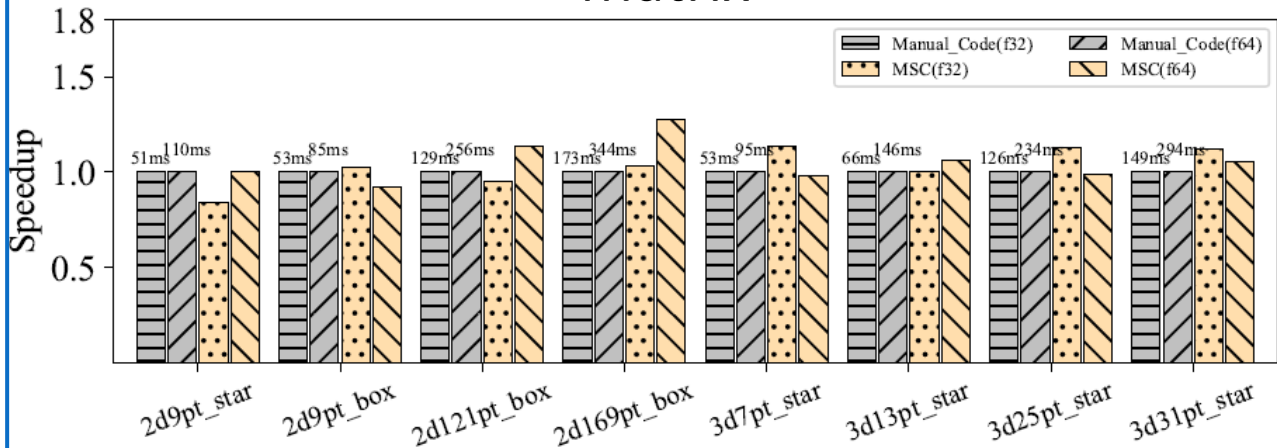
Performance on a single many-core processor

Sunway



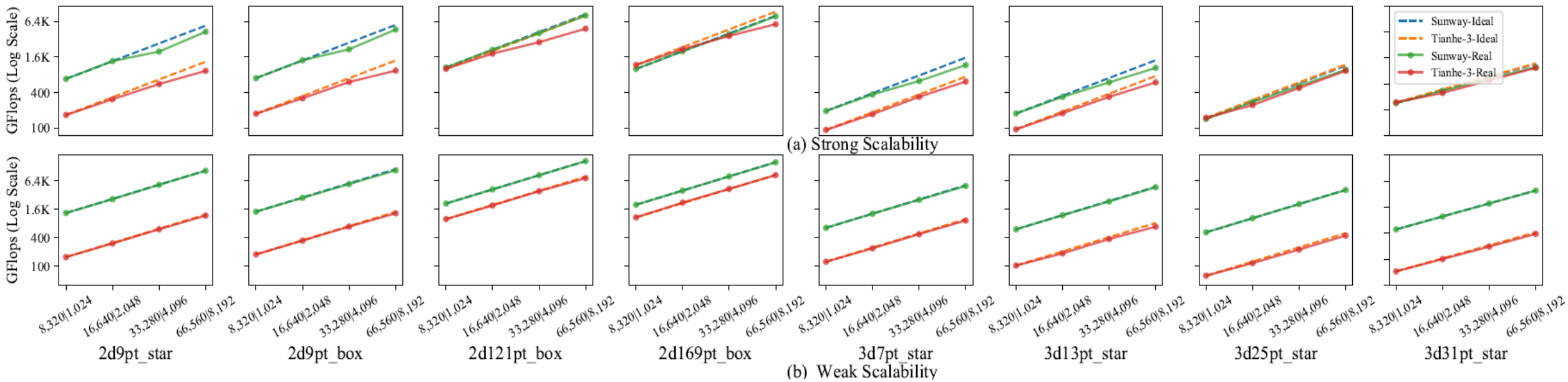
- MSC outperforms OpenACC in all cases, with the average speedup of **24.4×** (fp64) and **20.7×** (fp32).
- MSC can generate optimized codes exploiting the architectural features such as SPM and DMA for superior performance.

Matrix



- The performance of MSC generated stencil codes is close to the manually optimized OpenMP codes.
- Average speedup of 1.05× (fp64) and 1.03× (fp32). But MSC has **less LoC**.
- Matrix adopts homogeneous design, which is easier to optimize codes manually.

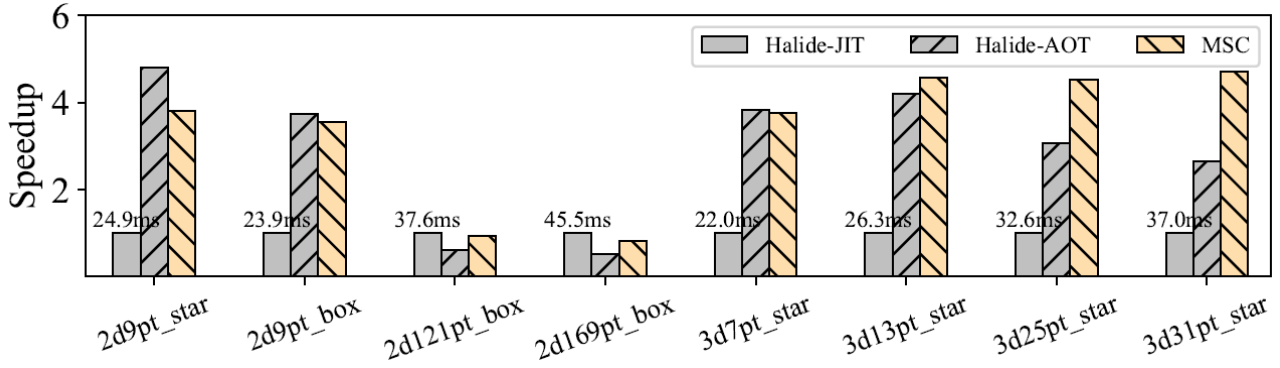
Strong and weak scalability

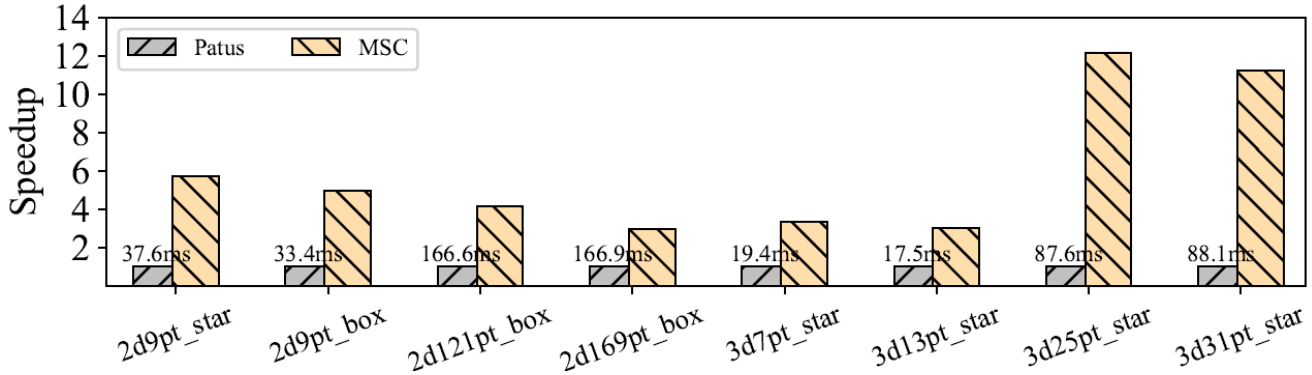


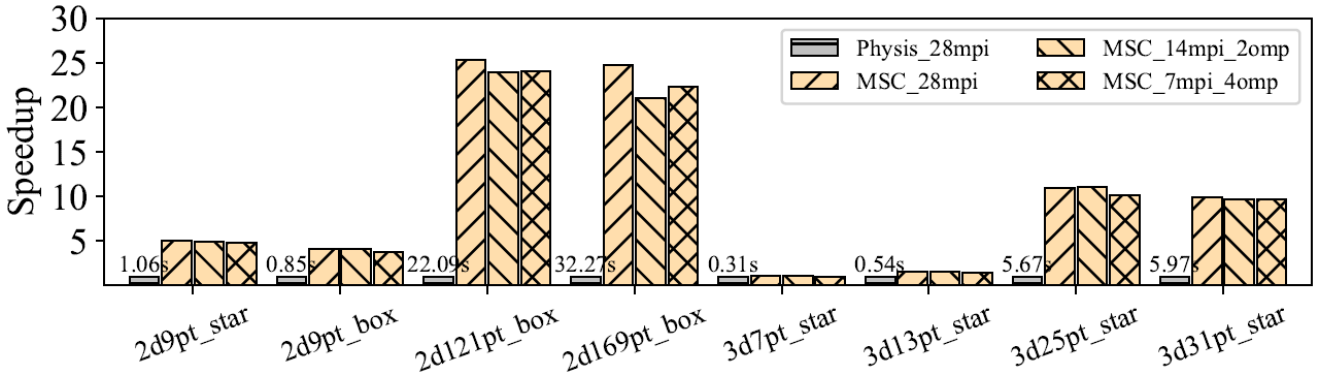
Dim	Weak Scalability Sub_grid per MPI	Strong Scalability Sub_grid per MPI	MPI Grid	Processes
2D	$4,096^2$	$4,096 \times 4,096$	$16 \times 8 \mid 8 \times 4$	128 | 32
	$4,096^2$	$4,096 \times 2,048$	$16 \times 16 \mid 8 \times 8$	256 | 64
	$4,096^2$	$2,048 \times 2,048$	$32 \times 16 \mid 16 \times 8$	512 | 128
	$4,096^2$	$2,048 \times 1,024$	$32 \times 32 \mid 16 \times 16$	1024 | 256
3D	256^3	$256 \times 256 \times 256$	$8 \times 4 \times 4 \mid 4 \times 4 \times 2$	128 | 32
	256^3	$256 \times 256 \times 128$	$8 \times 8 \times 4 \mid 4 \times 4 \times 4$	256 | 64
	256^3	$256 \times 128 \times 128$	$8 \times 8 \times 8 \mid 4 \times 8 \times 4$	512 | 128
	256^3	$128 \times 128 \times 128$	$16 \times 8 \times 8 \mid 8 \times 8 \times 4$	1024 | 256

- Strong: when scaling to the maximum number of cores (8x over the minimum) , the average speedup achieved by MSC is **6.74×** and **5.85×** on Sunway and Tianhe-3 platforms, respectively.
- Weak: almost linear, with **7.85×** and **7.38×** speedup on Sunway and Tianhe-3 platform, respectively.

Performance comparison on x86 CPU







Comparison with Halide (Baseline -- Halide-JIT):

Halide-AOT → 2.92x

MSC → 3.33x

- **Halide-JIT: JIT overhead**
- **Halide-AOT: redundant subscript expressions**

Comparison with Patus (Baseline -- Patus):

MSC → 5.94×

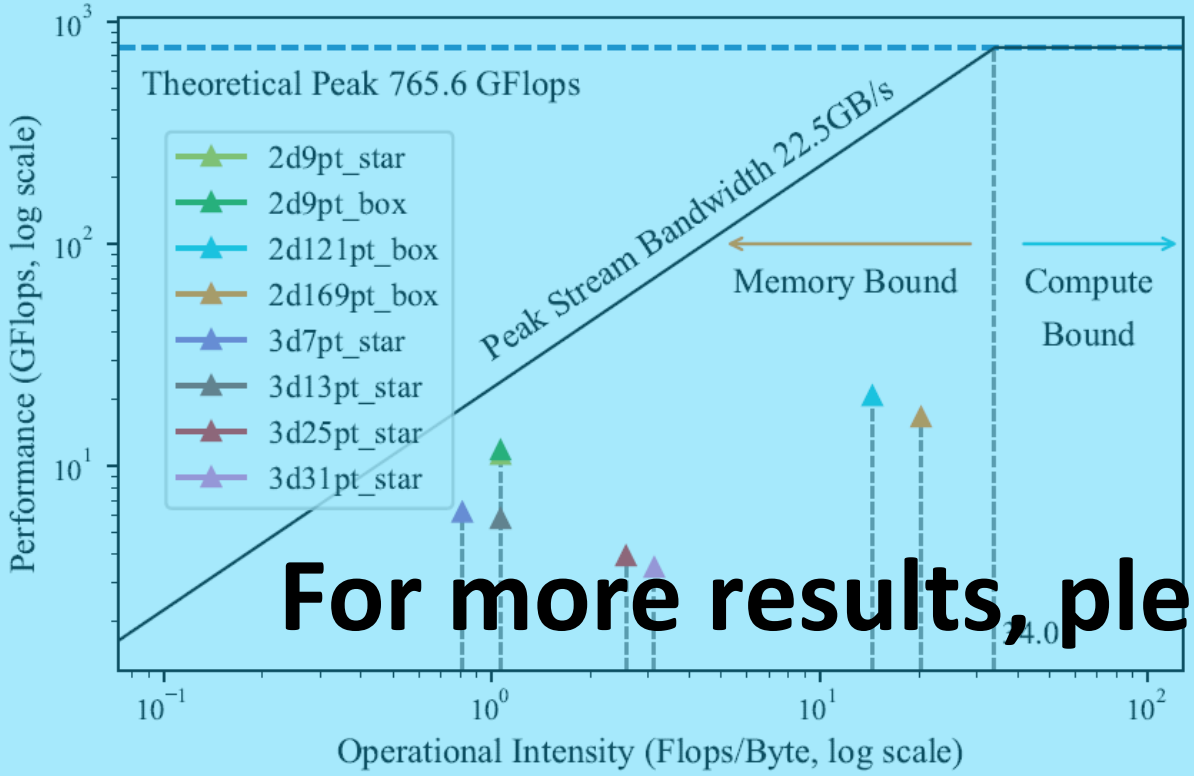
- **Patus: aggressive SIMD vectorization with SSE intrinsic**

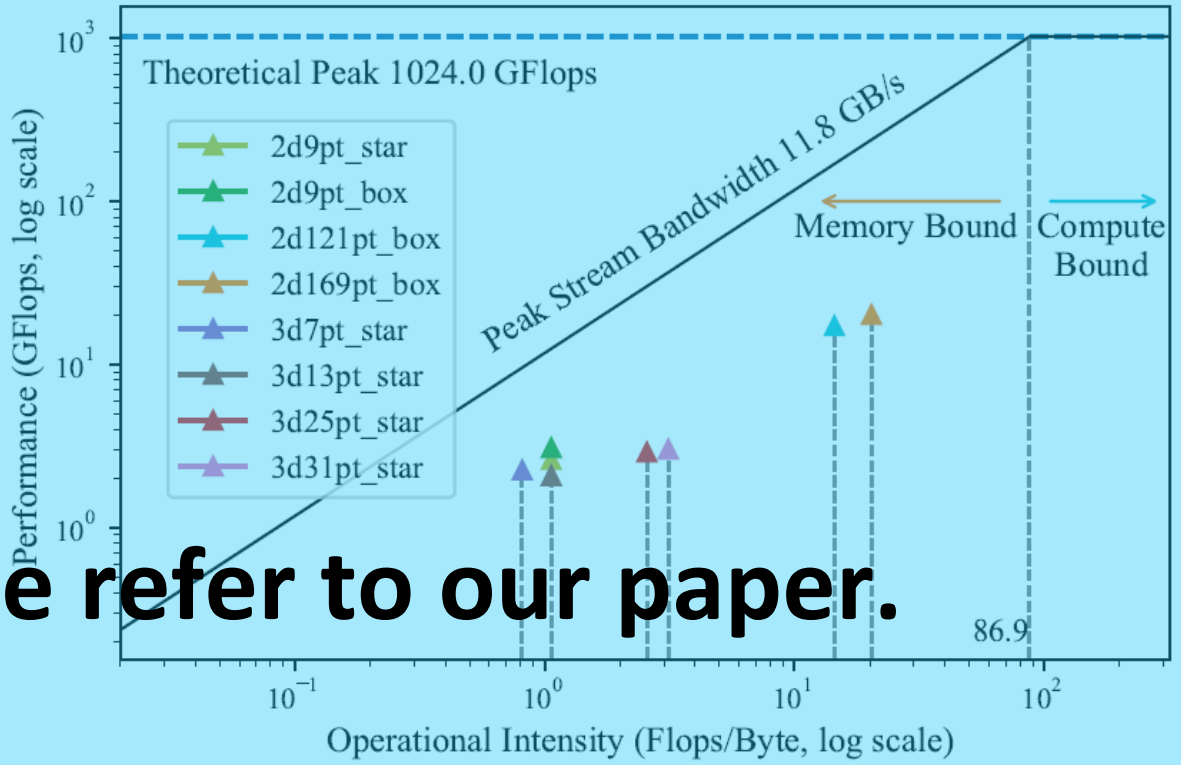
Comparison with Physis (Baseline -- Physis):

MSC → 9.88×

- **Physis: centralized RPC coordinator**

More results – roofline analysis

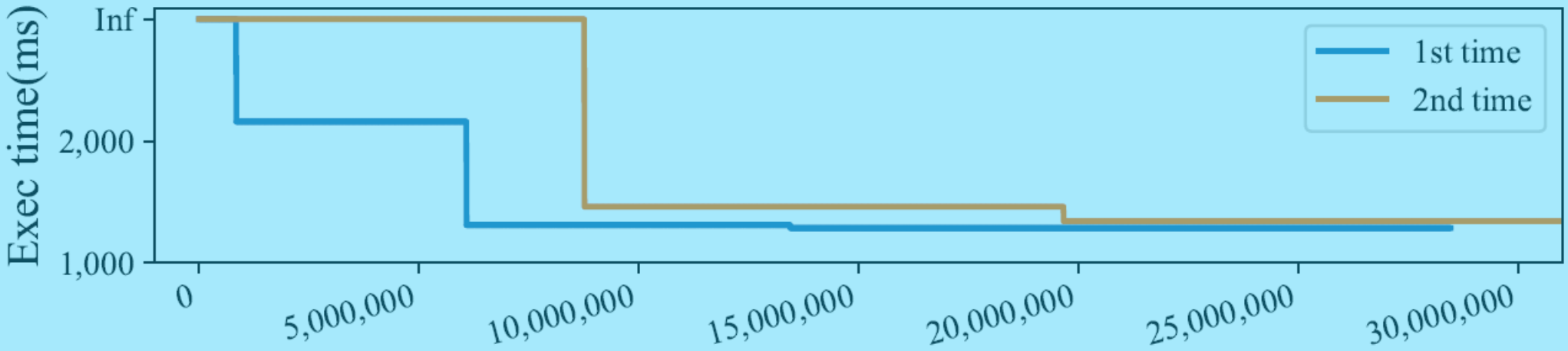




For more results, please refer to our paper.

- Three categories: ***1)*** 2d9pt_star, 2d9pt_box, 3d7pt_star, 3d13pt_star, ***2)*** 3d25pt_star, 3d31pt_star, and ***3)*** 2d121pt_box, 2d169pt_box
- Category 3): high operational intensity → better performance
- Category 2): 3D star shape → discrete (input grid) and redundant (halo region) data accesses → lower performance

More results – autotuning



For more results, please refer to our paper.

- Stencil: 3d7pt_star
- Input grid: 8192*128*128
- Stop after 13,460,000 (around **13 minutes**) and 19,670,000 (around **16 minutes**) iterations
- **3.28x** performance improvement

Conclusion

***MSC* --** a new stencil DSL that generates optimized stencil codes targeting emerging many-core processors

- Support expressing stencil computation with **multiple time dependencies**
- Provide **various optimization primitives** to exploit the parallelism and data locality across the computation and memory hierarchies
- Integrate a pluggable halo exchanging library in **large-scale** stencil codes

- MSC shows competitive performance
  - 24.4x over OpenACC on Sunway
  - 1.05x over OpenMP on Matrix, with less LoC
  - 1.14x over Halide, 5.49x over Patus, 9.88x over Physis on x86 CPU

Open-sourced at https://github.com/buaa-hipo/MSC-stencil-compiler

Thanks! Q&A