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
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**
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.

\[
   u_{i,j}^{t+1} = c_0 u_{i,j}^{t-1} + c_1 u_{i,j}^t + c_2 (u_{i+1,j}^t + u_{i-1,j}^t) + c_3 (u_{i,j+1}^t + u_{i,j-1}^t)
\]
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
  - (OUTPUT) 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
- \((MPE*1 + CPE*64) * 4\)
- 64KB SPM, manually controlled
- Direct Memory Access (DMA)
- Programming model: *Athread*

Matrix MT2000+ processor
- \(\text{SuperNode (32 cores)} * 4\)
- SN = panel (8 cores) * 4
- Each panel contains 8 cache-coherent compute cores.
- Programming model: *OpenMP*
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

---

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Single timestep</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Multiple timestep</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CPU</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Manycore</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Spatial tiling</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Streaming</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Temporal tiling</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Auto-tuning</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Halo exchange</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Pluggable library</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
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
Our design principle – Separating stencil expression from optimization

- **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
## Intermediate Representation of MSC

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

- **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

- A single level IR embedded in the abstract syntax tree.
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,j,i+2])
8. // Optimizations
9. ... Several optimization primitives
10. auto t = Stencil::t;
11. Result Res((i,j), B[i,j]);
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

- **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 independent

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**.
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
- **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
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.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Processor</th>
<th>Compiler</th>
<th>MPI</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunway TaihuLight</td>
<td>SW26010 (65 cores*4)</td>
<td>gcc-8.3, sw5cc</td>
<td>mpich-3.0</td>
<td>None</td>
</tr>
<tr>
<td>Tianhe-3 Prototype</td>
<td>MT2000+ (32 cores)</td>
<td>gcc-8.2</td>
<td>mpich-3.2</td>
<td>4.5</td>
</tr>
<tr>
<td>Local CPU Server</td>
<td>E5-2680v4<em>2 (14 cores</em>2)</td>
<td>gcc-8.3</td>
<td>openmpi-3.1</td>
<td>4.5</td>
</tr>
</tbody>
</table>

### Stencil benchmarks used in the evaluation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Read(Byte)</th>
<th>Write(Byte)</th>
<th>Ops(+,-,x,...)</th>
<th>Time Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2d9pt_star</td>
<td>72</td>
<td>8</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>2d9pt_box</td>
<td>72</td>
<td>8</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>2d121pt_box</td>
<td>968</td>
<td>8</td>
<td>231</td>
<td>2</td>
</tr>
<tr>
<td>2d169pt_box</td>
<td>1352</td>
<td>8</td>
<td>325</td>
<td>2</td>
</tr>
<tr>
<td>3d7pt_star</td>
<td>56</td>
<td>8</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>3d13pt_star</td>
<td>104</td>
<td>8</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td>3d25pt_star</td>
<td>200</td>
<td>8</td>
<td>41</td>
<td>2</td>
</tr>
<tr>
<td>3d31pt_star</td>
<td>248</td>
<td>8</td>
<td>50</td>
<td>2</td>
</tr>
</tbody>
</table>

- 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
MSC outperforms OpenACC in all cases, with the average speedup of $24.4 \times$ (fp64) and $20.7 \times$ (fp32).

MSC can generate optimized codes exploiting the architectural features such as SPM and DMA for superior performance.

The performance of MSC generated stencil codes is close to the manually optimized OpenMP codes. Average speedup of $1.05 \times$ (fp64) and $1.03 \times$ (fp32). But MSC has less LoC.

Matrix adopts homogeneous design, which is easier to optimize codes manually.
Strong and weak scalability

- Strong: when scaling to the maximum number of cores (8x over the minimum), the average speedup achieved by MSC is $6.74 \times$ and $5.85 \times$ on Sunway and Tianhe-3 platforms, respectively.

- Weak: almost linear, with $7.85 \times$ and $7.38 \times$ speedup on Sunway and Tianhe-3 platform, respectively.

<table>
<thead>
<tr>
<th>Dim</th>
<th>Weak Scalability</th>
<th>Strong Scalability</th>
<th>MPI Grid</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sub_grid per MPI</td>
<td>Sub_grid per MPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2D</td>
<td>4, 096</td>
<td>4, 096 × 4, 096</td>
<td>16 × 8</td>
<td>8 × 4</td>
</tr>
<tr>
<td></td>
<td>4, 096</td>
<td>4, 096 × 2, 048</td>
<td>16 × 16</td>
<td>8 × 8</td>
</tr>
<tr>
<td></td>
<td>4, 096</td>
<td>2, 048 × 2, 048</td>
<td>32 × 16</td>
<td>16 × 8</td>
</tr>
<tr>
<td></td>
<td>4, 096</td>
<td>2, 048 × 1, 024</td>
<td>32 × 32</td>
<td>16 × 16</td>
</tr>
<tr>
<td>3D</td>
<td>256$^3$</td>
<td>256 × 256 × 256</td>
<td>8 × 4 × 4</td>
<td>4 × 4 × 2</td>
</tr>
<tr>
<td></td>
<td>256$^3$</td>
<td>256 × 256 × 128</td>
<td>8 × 8 × 4</td>
<td>4 × 4 × 4</td>
</tr>
<tr>
<td></td>
<td>256$^3$</td>
<td>256 × 128 × 128</td>
<td>8 × 8 × 8</td>
<td>4 × 8 × 4</td>
</tr>
<tr>
<td></td>
<td>256$^3$</td>
<td>128 × 128 × 128</td>
<td>16 × 8 × 8</td>
<td>8 × 8 × 4</td>
</tr>
</tbody>
</table>
Comparison with Halide (Baseline -- Halide-JIT):
Halide-AOT $\rightarrow$ 2.92x
MSC $\rightarrow$ 3.33x
- Halide-JIT: JIT overhead
- Halide-AOT: redundant subscript expressions

Comparison with Patus (Baseline -- Patus):
MSC $\rightarrow$ 5.94x
- Patus: aggressive SIMD vectorization with SSE intrinsic

Comparison with Physis (Baseline -- Physis):
MSC $\rightarrow$ 9.88x
- Physis: centralized RPC coordinator
Three categories:

1) 2d9pt_star, 2d9pt_box, 3d7pt_star, 3d13pt_star,
   3d25pt_star, 3d31pt_star

2) 3d25pt_star, 3d31pt_star, and
   2d121pt_box, 2d169pt_box

3) high operational intensity → better performance

Category 2): 3D star shape → discrete (input grid) and redundant (halo region) data accesses → lower performance

For more results, please refer to our paper.
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
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](https://github.com/buaa-hipo/MSC-stencil-compiler)
Thanks! Q&A