The Progress of OpenACC for MN-Core and MNCL Development: Compiler and Runtime

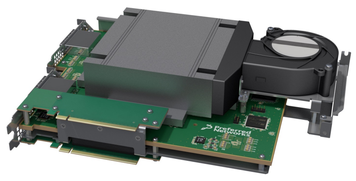
Ryuta Tsunashima[(1)], Naohito Nakasato[(2)], Ryozaburo Suzuki[(3)], Katsuhiro Endo[(4)], Hiroto Imachi[(5)], Junichiro Makino[(5,1)]

(1) Kobe University, (2) The University of Aizu, (3) Sinby Corporation, (4) National Institute of Advanced Industrial Science and Technology, (5) Preferred Networks, Inc.

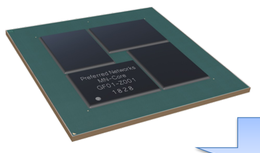
ACKNOWLEDGEMENT
This work was partially supported by MEXT as "Feasibility studies for the next-generation computing infrastructure".

MN-Core

- An AI and HPC accelerator by Kobe University and Preferred Networks (PFN)
  - The arithmetic unit supports double precision (difference from other AI accelerators)
- High power efficiency (= High FLOPS per watt) than same generation GPUs and CPUs
  - Ranked #1 three times in the Green500 in 2020～2021
- SIMD (Single Instruction, Multiple Data) Architecture
  - 8192 Processing Elements (PEs) run parallelly and synchronously without threads
  - Simpler behavior than GPUs, but Increased number of PEs per chip area
- Difference with other processors: memory model
  - MN-Core: Distributed memory type | GPUs, CPUs：Shared memory type
  - Each PE has its own local memory (SRAM), No cache memory ➔ No coherency

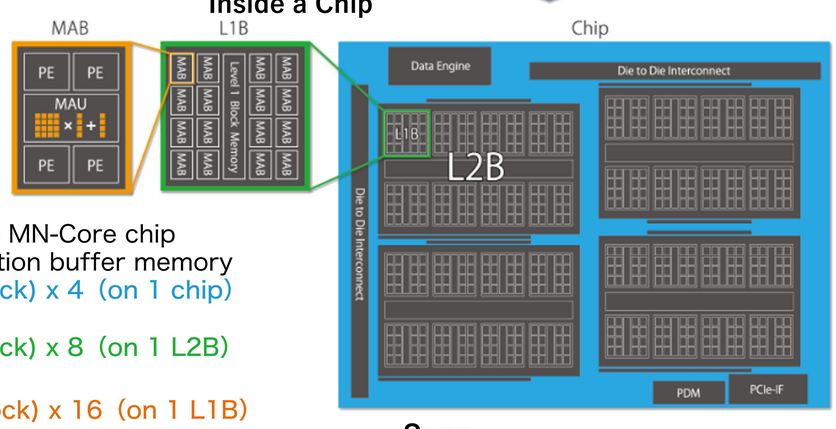


MN-Core Board



MN-Core Package has Chip x 4

Inside a Chip



Internal Structure of the Chip

- Tree hierarchical structure in a MN-Core chip
- Each layer has the communication buffer memory
  - L2B (Level 2 Broadcast Block) x 4 (on 1 chip)
    - L2B Memory: 72KB
  - L1B (Level 2 Broadcast Block) x 8 (on 1 L2B)
    - L1B Memory: 72KB
  - MAB (Matrix Arithmetic Block) x 16 (on 1 L1B)
    - PE (Processing Element, general purpose) x 4
      - Local memory: 72KB (on 1 PE)
        - Total 576MB (on 1 Package)
        - Directly accessible to PE
      - General Register File: 4.5KB
      - Temporary Register: 72B
    - MAU (Matrix Arithmetic Unit) x 1 (on 1 MAB)
      - Shared by 4 PEs

Spec

- Peak performance : 32.8TFLOPS (double)
- Power consumption : 500W (Predicted)
- Clock frequency : 500MHz
- DRAM (off chip)
  - Bandwidth : 100GB/s
  - Size : 32GB

General Purpose Language for MN-Core

- Existing languages for MN-Core: DSL or for AI
- Developing two general purpose languages (APIs) for HPC
  - MNCL (CUDA & OpenCL equivalent)
    - has C version and Fortran version
  - OpenACC (extended for MN-Core)

Advantages of OpenACC

- No need to rewrite all of code for porting
  - –Reducing code volume than CUDA and OpenCL
- Just add directives to run accelerators
  - –Compiling code with directives as hints
- High productivity
  - –Applying directives in any part of the code

OpenACC for MN-Core

- Support languages: C, Fortran (except C++ now)
- Based on OpenACC 1.0
- APIs for separate compilation (part of OpenACC 2.0 and later)
- Unique extensions for distributed memory model
  - Data partitioning at each layer in MN-Core
  - Collective communications between host ⇔ PEs, PEs ⇔ PEs
    - Especially halo (shadow) exchange
- Supporting language standard precision (currently, half-precision is not expected)

Current Status

OpenACC C lang. Prototype

- Embarrassingly parallel computing with multi-D arrays can be compiled to current MNCL
- 1D-3D stencil calculation code with default divisions can be compiled to current MNCL too
  - We confirmed to compile Himeno benchmark (▼figure) to MNCL device code with proper division, memory allocation and communication

Himeno benchmark code with OpenACC/MN-Core

```
float jacobi(int nn) {
  float gosa,s0,ss;
#pragma acc enter data copyin(p[:MIMAX][:MJMAX][:MKMAX], wrk1[:MIMAX][:MJMAX][:MKMAX], ¥
  wrk2[:MIMAX][:MJMAX][:MKMAX], bnd[:MIMAX][:MJMAX][:MKMAX]) shadow(p[1][1][1])
#pragma acc parallel present(p[:MIMAX][:MJMAX][:MKMAX], wrk1[:MIMAX][:MJMAX][:MKMAX], ¥
  wrk2[:MIMAX][:MJMAX][:MKMAX], bnd[:MIMAX][:MJMAX][:MKMAX]) shadow(p[1][1][1])
{ int i,j,k,n;
  for(n=0 ; n<nn ; ++n){
    gosa = 0.0;
#pragma acc loop gang collapse(3) reduction(+:gosa)
    for(i=1 ; i<IMAX-1 ; ++i)
      for(j=1 ; j<JMAX-1 ; ++j)
        for(k=1 ; k<JMAX-1 ; ++k){
          s0 = a012 * p[i+1][j][k] + a012 * p[i][j+1][k] + a012 * p[i][j][k+1]
             + b * ( p[i+1][j+1][k] - p[i+1][j-1][k] - p[i-1][j+1][k] + p[i-1][j-1][k] )
             + b * ( p[i][j+1][k+1] - p[i][j-1][k+1] - p[i][j+1][k-1] + p[i][j-1][k-1] )
             + b * ( p[i+1][j][k+1] - p[i-1][j][k+1] - p[i+1][j][k-1] + p[i-1][j][k-1] )
             + c * p[i-1][j][k] + c * p[i][j-1][k] + c * p[i][j][k-1] + wrk1[i][j][k];
          ss = ( s0 * a3 - p[i][j][k] ) * bnd[i][j][k];
          gosa += ss*ss;
          wrk2[i][j][k] = p[i][j][k] + omega * ss;         }
#pragma acc loop gang collapse(3)
    for(i=1 ; i<IMAX-1 ; ++i)
      for(j=1 ; j<JMAX-1 ; ++j)
        for(k=1 ; k<JMAX-1 ; ++k)
          p[i][j][k] = wrk2[i][j][k];
#pragma acc reflect (p)
  } /* end n loop */       }
```

MNCL C Prototype (under development too)

- Improved MNCL host runtime overhead
  - The existing host runtime has overhead, taking about several hundred us for sending 8192 double variables to all PEs per one way in total
    - Because it is optimized for AI
  - By calling a more primitive library, we confirmed a significant overhead reduction
    - We will develop its wrapper library for users
- Continued developing the MNCL C ver. compiler to support stencil calculations with 1D to 3D arrays
  - The OpenACC/MN compiler supported it
- Distributed memory model in MN-Core can be described like figures (**bold font**) ▼▶
  - `__global` memory represents PCIe buffer memory (PDM) in MN-Core (currently)

```
int a_size[3] = {10, 10, 6};
int a_shadow[3][2] = {{1, 1}, {1, 1}, {1, 1}};
halo_comm(a, sizeof(double), 3, a_size, a_shadow);
```

◁ **Halo communication code excerpted from the OpenACC to MNCL Himeno benchmark code compilation result**

Ongoing Plans

- We will release OpenACC prototype environment on WWW in fiscal year 2025 (before next March)
  - Users will be able to touch an early version of the MNCL compiler and the OpenACC/MN compiler

Running on MN-Core

ex) OpenACC to MNCL device code compilation (1D vector add code)

```
#include <stdio.h>
#define N 8192

int main() {
    double array1[N], array2[N], array3[N];
    for(int j=0; j<N; j++) {
        array1[j] = 0.0f;
        array2[j] = 1.0f;
        array3[j] = 1.0f;
    }

#pragma acc enter data copyin(array1[:N], ¥
    array2[:N], array3[:N])

#pragma acc parallel loop gang present( ¥
    array1[:N], array2[:N], array3[:N])
    for(int i=0; i<N; i++) {
        array1[i] = array2[i] + array3[i];
    }

#pragma acc exit data copyout(array1[:N], ¥
    array2[:N], array3[:N])
}
```

```
__kernel void main_kernel0(
__global double* _arg_array1,
__global double* _arg_array2,
__global double* _arg_array3
) {
__private double array1[1];
bm2 double array1_bm2[1 * 512];
bm1 double array1_bm1[1 * 64];
__private double array2[1];
bm2 double array2_bm2[1 * 512];
bm1 double array2_bm1[1 * 64];
__private double array3[1];
bm2 double array3_bm2[1 * 512];
bm1 double array3_bm1[1 * 64];

distribute(array1_bm1, array1_bm2, _arg_array1, 1);
distribute_pe(array1, array1_bm1, 1);
distribute(array2_bm1, array2_bm2, _arg_array2, 1);
distribute_pe(array2, array2_bm1, 1);
distribute(array3_bm1, array3_bm2, _arg_array3, 1);
distribute_pe(array3, array3_bm1, 1);

array1[0] = array2[0] + array3[0];

collect_pe(array1_bm1, array1, 1);
collect(_arg_array1, array1_bm2, array1_bm1, 1);
collect_pe(array2_bm1, array2, 1);
collect(_arg_array2, array2_bm2, array2_bm1, 1);
collect_pe(array3_bm1, array3, 1);
collect(_arg_array3, array3_bm2, array3_bm1, 1);
}
```

- 1D embarrassingly parallel computing can be executed
  - OpenACC 1D vecadd code (▲figure) has already been confirmed to be running on the actual MN-Core
  - We compiled OpenACC ➔ MNCL device code ➔ binary

Evaluation

- We confirmed to be able to compile OpenACC via MNCL to theoretical peak performance assemblies (just calculation part) (no nop, minimum num. of inst. lines, except vector inst.)
  - Compiled 7-point stencil code (like an example above) by the 3D supported simple implementation MNCL compiler

OpenACC/MN-Core Interface

Loop level explanation

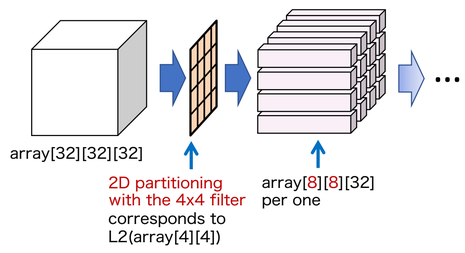
- gang: PEs parallel
- `vector`: vector instruction parallel
- `worker`: undefined (no suitable layer)
  - Not using it is the same as the GPU convention
  - Ignored even if specified

```
#pragma acc enter data copyin(a[N][N],b[N][N],c[N]) ¥
    L2(a,b[4][4];c[16]) L1(a,b[4][2];c[8]) ¥
    mab(a,b[2][8];c[16]) pe(a,b[4][1];c[4])
```

▲ ex) Specifying data division in C

```
!$acc enter data copyin(a(1:N, 1:N),b(1:N, 1:N),c(1:N)) &
    !$acc& L2(a,b(4, 4);c(16)) L1(a,b(4, 2);c(8)) &
    !$acc& mab(a,b(2, 8);c(16)) pe(a,b(4, 1);c(4))
```

▲ ex) Specifying data division in Fortran



▲ Division image if `L2(array[4][4])` is described

◀ ▲ Specifying number of data divisions on each layer in the chip

◁ Like through grid pattern filters

- Red code is a MN-Core extension
- **This can be omitted by default**
  - Default: Near-minimum halo communication cost divisions for square and cube
  - Total number of division with all layers:
    - 2D: [128][64], 3D: [16][16][32]

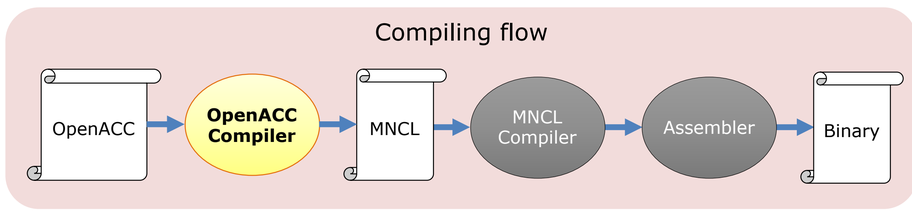
◀ Example of 7-point stencil code in C

- Just add directives (**bold**) to sequential code for CPU
  - Red code is a MN-Core extension (divisions are omitted)
- `shadow` clause: Specifying halo size of each dimension
- `reflect` directive: Specifying halo exchange timing
- Feasibility: In XcalableMP that is an API for multi-node parallel, there are implementation of halo communication compilation with like above interfaces

```
#pragma acc parallel create(temp[0:128][0:128][0:128]) ¥
    copy(a[0:128][0:128][0:128]) shadow(a[1][1][1])
for(count=0; count<N; count++) {
#pragma acc loop gang vector collapse(3)
    for(int i=1; i<128-1; i++){
        for(int j=1; j<128-1; j++){
            for(int k=1; k<128-1; k++){
                temp[i][j][k] =
                    c1*(a[i-1][j][k]+a[i+1][j][k]
                    +a[i][j-1][k]+a[i][j+1][k]
                    +a[i][j][k-1]+a[i][j][k+1])
                    +c2*a[i][j][k];
    } } }
#pragma acc loop gang vector collapse(3)
    for(int i=1; i<128-1; i++){
        for(int j=1; j<128-1; j++){
            for(int k=1; k<128-1; k++){
        a[i][j][k] = temp[i][j][k];
    } } }
#pragma acc reflect(a)
}
```

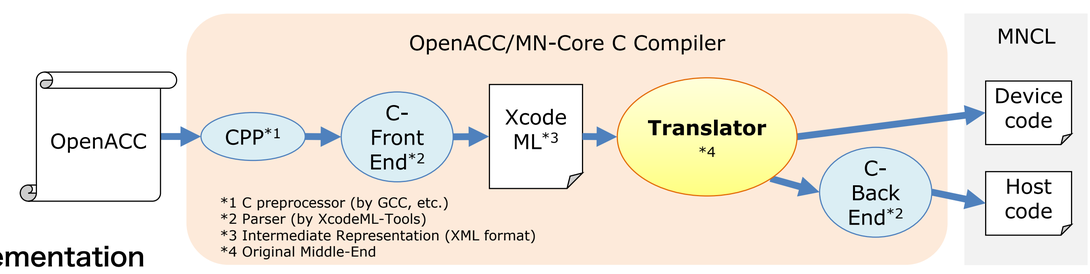
OpenACC/MN-Core Compiler

- The OpenACC to MNCL source-to-source compiler
  - Compiling MNCL to assembly with MNCL compiler
- Currently, developing the prototype compiler for C lang.



Advantages of Source-to-Source

- Reducing implementation cost
  - –Commonize the process after compiling MNCL with the MNCL compiler
- Enabling sophisticated performance tuning with MNCL in OpenACC
  - –Allows partial MNCL function calls from OpenACC
    - This is included in the OpenACC specification (CUDA kernels can be called by OpenACC for GPU)



Implementation

- Intermediate Representation (IR): Used for code analysis in compilers
- XcodeML：An XML format IR has fully compatible C and Fortran (reversible)
  - XcodeML can be included OpenACC information
- The translator analyzes and translates XcodeML for “OpenACC to MNCL”