MN-Coreアーキテクチャのための プログラミングモデル

中里直人 (会津大学)
神戸大学　システムソフトウエア・ライブラリ調査研究グループ

2025/03/25

Introduction

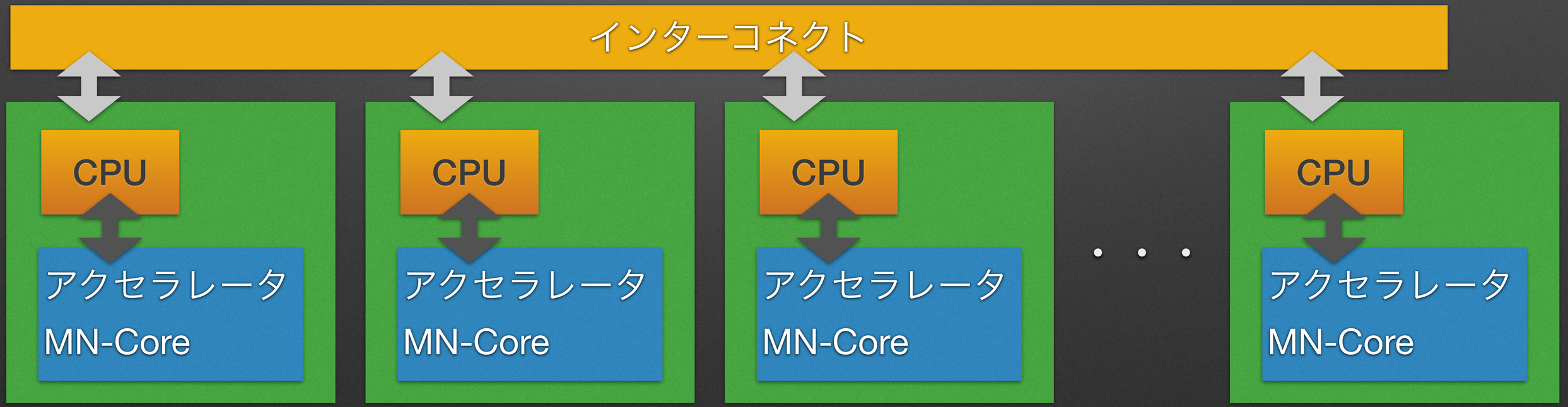
- 次世代計算基盤に係る調査研究：神戸大チーム、システムソフトウエア・ライブラリ調査研究グループの成果について報告する。

- 神戸大チームでは、MN-Coreをアクセラレータとして利用する並列計算機をターゲットとしており、我々のグループは、既存アプリケーションの移行に必要なソフトウエア環境について調査検討した。

- 具体的にはMN-Coreのプログラミングモデルについて得られた知見について紹介する。

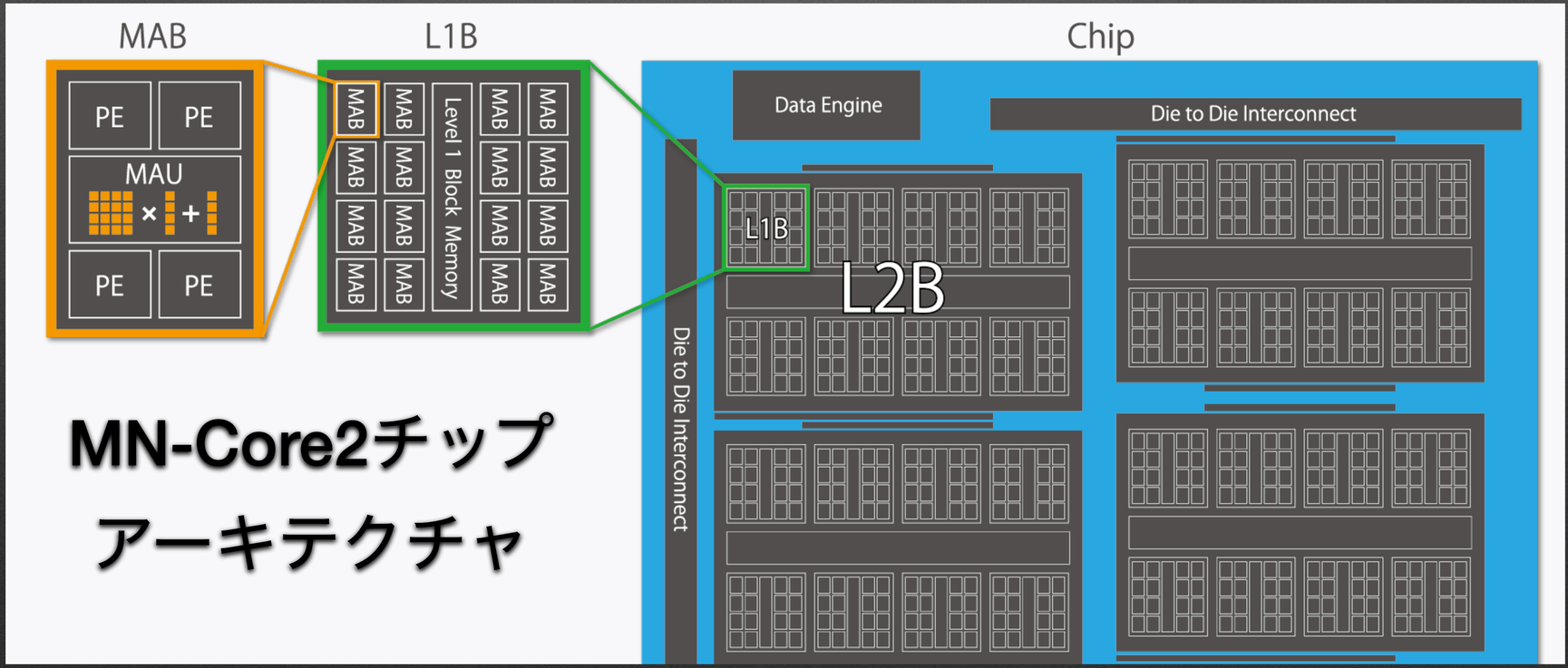
システム構成

- 検討していたシステムは複数のアクセラレータノードからなるクラスター型システム
- ノード間のプログラミングモデルは、既存の分散メモリ型



MN-Coreアーキテクチャ

- チップ全体をSIMD命令で制御するアーキテクチャ
  - 外部から供給する単一の命令列で制御する
- 演算器が占めるトランジスタの割合が大きい
- 大量のオンチップメモリを有効利用することで高性能



MN-Coreシリーズの比較

	MN-CORE	MN-CORE2	MN-CORE X
PE総数	8192	4096	> 4096
clock	500MHz	750MHz	> 1 GHz
倍精度行列積	32 TF	12 TF	
レジスタ	4.5 KB	4 KB	> 4 KB
ローカルメモリ	72 KB	32 KB	100x (?)
L1B メモリ	72 KB	64 KB	> 64 KB
L2B メモリ	72 KB	256 KB	> 256 KB
PDM メモリ	16 MB	16 MB	> 16 MB

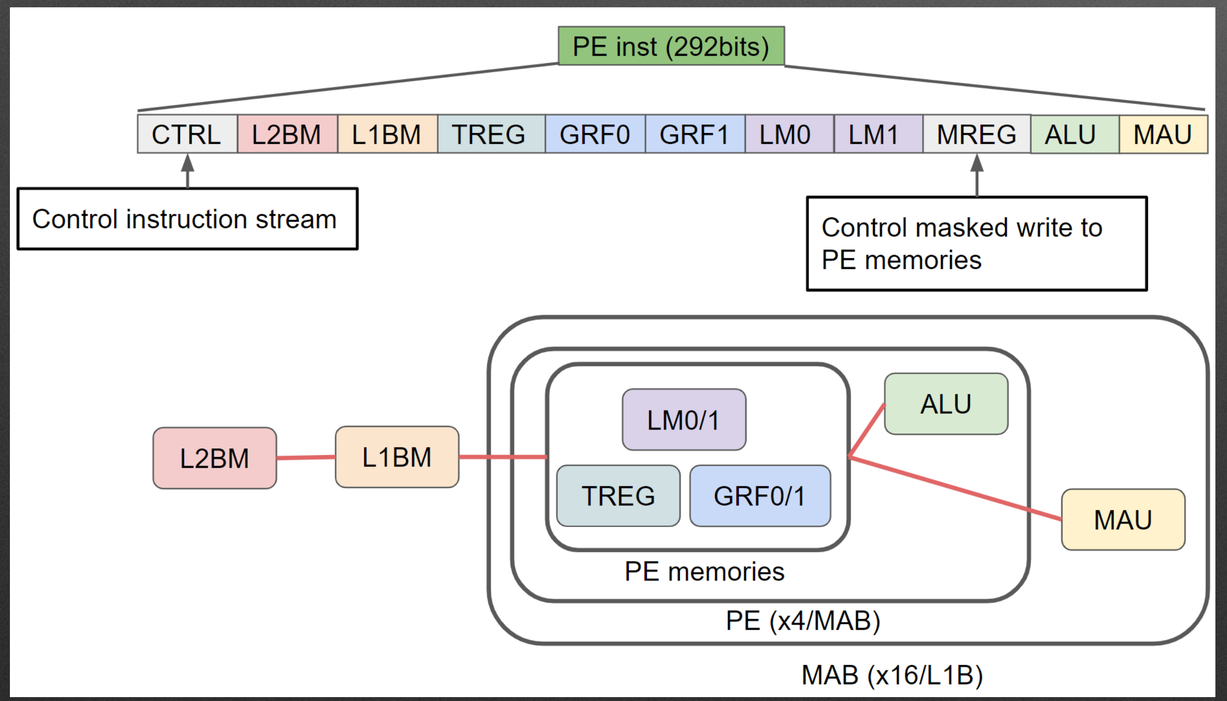
ローカルメモリ総量

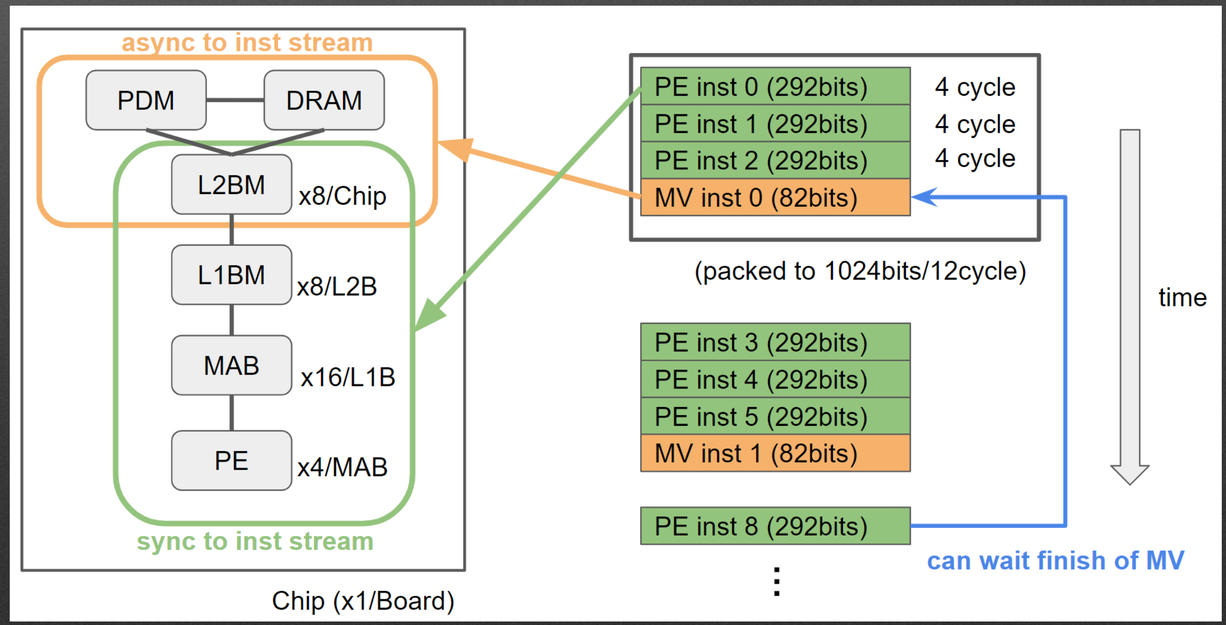
現行のMN-Core: 数百 MB

将来のMN-Core: GB超

MN-Coreの命令

- 命令は二つの部分に分けることができる
  - 演算器とレジスタ・メモリを制御する部分：計算のコア部分
  - L1B, L2B, PDM/DRAM間のデータ移動を制御する部分





演算部分の記述

- ML実装用のPyTorchによるアプリケーション記述を調査検討
- C言語ベースの演算カーネルコンパイラを複数検討し実装をし、MN-Coreの実機システムで実行・性能評価

数値積分の性能評価例

QMC積分カーネルの一部：PE間データ移動なし

```
x589=x5+x8+x9;

x1sq = x1*x1;
x2sq = x2*x2;
x3sq = x3*x3;
x5sq = x5*x5;
x7sq = x7*x7;
x9sq = x9*x9;
x9cu = x9sq*x9;

cc = -x279*x349*x5sq+x1567*x279*x349*x589-x349*x589*x7sq
   -2*x349*x5*x7*x9-x1567*x279*x9sq-x1567*x349*x9sq
   +x5sq*x9sq-x1567*x589*x9sq+2*x5*x7*x9sq
   +x7sq*x9sq+2*x1567*x9cu;
dd = x279*x3sq*x5sq+x2sq*x349*x5sq-x123*x279*x349*x5sq
   -x1567*x279*x3sq*x589-x1567*x2sq*x349*x589
   -x1sq*x279*x349*x589+x123*x1567*x279*x349*x589
   -2*x1*x2*x349*x589*x7+x3sq*x589*x7sq
   -x123*x349*x589*x7sq+2*x1*x279*x3*x5*x9
   -2*x1*x2*x349*x5*x9+2*x2*x3*x5sq*x9-2*x1567*x2*x3*x589*x9
   +2*x2*x3*x5*x7*x9+2*x3sq*x5*x7*x9-2*x123*x349*x5*x7*x9
   -2*x1*x3*x589*x7*x9+x1567*x2sq*x9sq+x1sq*x279*x9sq
   -x123*x1567*x279*x9sq+2*x1567*x2*x3*x9sq
   +x1567*x3sq*x9sq+x1sq*x349*x9sq-x123*x1567*x349*x9sq
   +2*x1*x2*x5*x9sq-2*x1*x3*x5*x9sq+x123*x5sq*x9sq
   +x1sq*x589*x9sq-x123*x1567*x589*x9sq+2*x1*x2*x7*x9sq
   +2*x1*x3*x7*x9sq+2*x123*x5*x7*x9sq+x123*x7sq*x9sq
   -2*x1sq*x9cu+2*x123*x1567*x9cu;
```

積分点数 ($\times 10^9$)	V100	H100	MN-Core(GPFN2)
1.22		1.767	0.8563657535
2.147	1.728	3.573	0.9931252071
8.599	2.223	5.196	1.189134253
12.14	1.948	4.669	1.220734277
24.29	1.973	4.921	1.246782282

Table 1: QMC による数値積分の性能評価

倍精度演算によるループ積分計算の評価

性能は 10^9 points / sec

データ移動のないアプリの動作実証

課題：数学関数の最適化

データ移動の実装レベル

- Level 1 : PE/L1B/L2Bの共有メモリをTreeトポロジーのNetwork On Chip (NoC)と考えて、宛先とデータのブロードキャストとマスク書き込みの組み合わせで実装：任意のデータ移動が可能

- Level 2 : 近接のデータ移動がある場合、近接したPE/L1B/L2B間のデータ移動命令を利用して命令数を大きく削減 (高屋敷さんの発表)

- Level 3 : データ移動命令と演算命令をオーバーラップして、さらに命令数を削減 (姫野ベンチマーク(PFN, 2024)の研究報告)

データ移動部分の記述

- PE/MV命令を記述することで、任意のデータ移動が実装できる
  - PE/L1B/L2B間の接続をNoCとして扱い、データ移動命令を構築する
- カーネル記述でのデータ移動記述については、複数案について検討した。最適な手法については今後も検討が必要である。
- 他、既存のアプリを移行する手段として
  1. BLAS/FFTなどのライブラリの利用
  2. アプリケーション特化のライブラリ/DSL：粒子法にはFDPSが一部対応済み
  3. プラグマベースコンパイラ

プラグマベースコンパイラの変換例

```
#pragma acc enter data copyin(array[0:128][0:128][0:128])

...

#pragma acc parallel present(array[0:128][0:128][0:128]) ¥
shadow(array[0:1][0:1][0:1])
for(int count=0; count<1000; count++) {

#pragma acc loop 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++){
                array[i][j][k] =
                    c1*(array[i+1][j][k]
                    +array[i][j+1][k]
                    +array[i][j][k+1]
                    +array[i][j][k]);
    } } }

#pragma acc reflect(array)

} // count の for 文（＝オフロード適用範囲）の終了

...

#pragma acc exit data copyout(array[0:128][0:128][0:128])
```

```
__kernel void main_kernel0(
__global double*** _arg_array,
__global double _arg_c1
) {
// 1PEあたりの袖領域を含めたサイズ
__private double array[9][9][5];
bm2 double array_bm2[9 * 4][9 * 4][5 * 32];
bm1 double array_bm1[9 * 4][9 * 4][5 * 4];
__private double c1;

// distribute、collect関数はデフォルト分割に対応している前提
distribute(array_bm1, array_bm2, _arg_array, 9 * 9 * 5);
distribute_pe(array, array_bm1, 9 * 9 * 5);
broadcast(&c1, &_arg_c1, 1);

{{
int count;
for(count = 0; (count) < (1000); (count)++)
{ {
int i;
for(i = 0; (i) < (8); (i)++)
{{
int j;
for(j = 0; (j) < (8); (j)++)
{{
int k;
for(k = 0; (k) < (4); (k)++)
{
array[i][j][k] = (c1) * ((((array[(i) + (1)][j][k]) + (array[i][(j) + (1)][k]))
+ (array[i][j][(k) + (1)])) + (array[i][j][k]));
}}}}}
}
// 袖通信（関数はデフォルト分割に対応している前提）
int array_size[3] = {9, 9, 5};
int array_shadow[3][2] = {{0, 1}, {0, 1}, {0, 1}};
halo_comm(array, sizeof(double), 3, array_size, array_shadow);
}  // count の for 文の終了
}}

collect_pe(array_bm1, array, 9 * 9 * 5);
collect(_arg_array, array_bm2, array_bm1, 9 * 9 * 5);
}
```

まとめ

- 本調査研究で検討したシステムは、アクセラレータベースの並列システムである。
- MN-Coreアーキテクチャの制御部分は、演算部分とデータ移動に二分される。演算部分ではアプリケーションの数式をカーネルとして記述する。この部分に対応したコンパイラを複数実装し、MN-Core実機で性能評価した。
- 現行MN-Coreでもオンチップメモリに収まるアプリケーションは高性能。将来のMN-Coreは大容量近接メモリを採用し、さらなる性能向上を目指している。
- アセンブリ命令で記述することで、任意のデータ移動パターンは実装できるが、カーネル記述と組み合わせた方式については今後も検討が必要である。