システム調査研究・神戸大学「オリジナルアクセラレータアーキテクチャによる超低消費電力次世代計算基盤の評価」結果報告

牧野淳一郎
神戸大学

2024 年度　「次世代計算基盤に係る調査研究」　　合同ワークショップ 2024/12/27

成果報告の概要

- 当初計画の調査内容
- 実際の調査内容・主要な成果

調査研究の概要（アーキテクチャ調査研究グループ）

取組概要

分野自体の技術トレンド、
半導体プロセス技術動向からポスト富岳時代に実現可能なシステム
がどのようなものかを検討し、評価対象アプリケーションによるシステム評価を行う

調査研究体制

調査内容

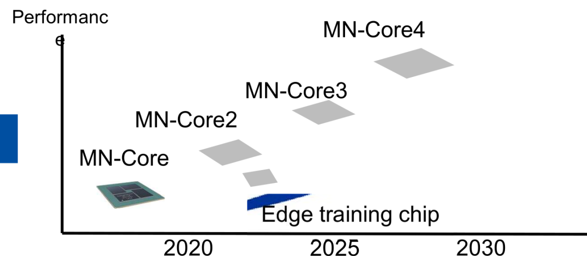
独自アクセラレータ・CPU構造見直しによる省電力化・効率改善に重点

アクセラレータ側:
現時点で同じ半導体技術では世界最高の電力当り性能を実現している

MN-Core をベースに 一層高い電力性能、チップ面積あたり性能を目指すと共に、アプリケーションとのコデザイン により高い実行効率を実現する

汎用プロセッサ側:

RISC-V ベースの独自設計で世界最高レベルの性能を目指す。



スケジュール

初年度
アーキテクチャのレファレンス設計

2年度
アーキテクチャ・システムソフトウェア・アプリケーショングループの共同による性能評価/アルゴリズム、設計改良、性能評価の繰り返し

調査研究の概要（システムソフトウェア・ライブラリ調査研究グループ）

取組概要

近年HPCアプリケーションで大きな課題となってきている「実行効率の低下」と¨アプリケーション開発の困難さの増大」に対する新しいアプローチによる解決の可能性をアーキテクチャグループ・アプリケーショングループと協力しつつ検討する。

階層キャッシュ・コア間共有メモリに頼る「力任せな」並列化ではなく、データの局所性を最大限に活用するソフトウェア制御による高効率な並列化をアプリケーション開発の負担を軽減しつつ実現する方向性を調査する。

調査研究体制

調査内容

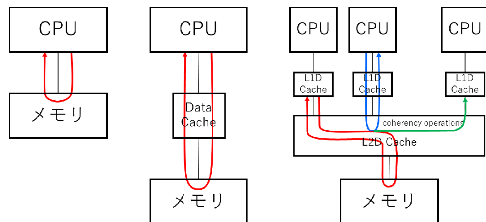
DSL・フレームワークによる高効率コードの自動生成に重点

深層学習では高レベルDSL/フレームワークが一般的なアプローチになった：TensorFlow, PyTorch, JAX

アプリケーション開発者が個別アーキテクチャ向けのコーディング、チューニングする必要がなくなっている

(PFN は Chainer/Pytorch、CuPy 等で実績がある

他のアプリケーションでも同様の方向での高い効率と開発の容易さを実現する可能性の評価に重点をおく。



階層キャッシュシステムではメモリアクセスもコア間通信もエネルギー消費大

スケジュール

初年度
DSL/フレームワーク/言語仕様検討・ プロトタイプ実装開発

2年度
アーキテクチャ・システムソフトウェア・アプリケーショングループの共同による性能評価/アルゴリズム、設計改良、性能評価の繰り返し

調査研究の概要（アプリケーション調査研究グループ）

取組概要

重要なアプリケーション、アプリケーションカーネルの抽出、 次世代システム向けアルゴリズムの検討と性能評価を合わせて行う。伝統的なHPCアプリケーションだけでなく、AI応用、OSS等についても十分な検討を行う。

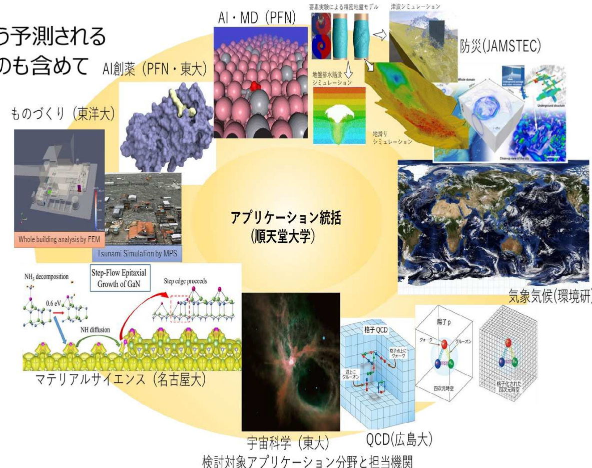
最重要なポイント:多くの分野において、実際に演算性能を有効に使えるようにアルゴリズム・アーキテクチャ双方の改良を進めること

分野抽出の理由:・これまで重要なアプリケーションであり、今後もそう予測される
・多様なアルゴリズムをカバーすることで、現在想定されていないものも含めて十分に広いアプリケーションで高い実行効率を実現する

調査内容

アプリケーション分野とそれを担当する研究機関は以下の通りである

- 創薬/MD/AI シミュレーション応用 PFN
- ゲノム科学 東京大学
- 防災・減災(地震・津波) JAMSTEC
- 防災・減災(気象・気候) 環境研
- ものづくり(CAE) 東洋大学
- 物性/材料 名大
- 基礎科学(素核) 広島大学
- 基礎科学 (宇宙) 東大
- OSS/商用アプリケーション



検討対象アプリケーション分野と担当機関

調査研究体制

- 神戸大学
  - ◎順天堂大学
  - Preferred Networks
  - JAMSTEC
  - 環境研究所
  - 東洋大学
  - 名古屋大学
  - 広島大学
  - 東京大学

評価内容:抽出したアプリケーション/アルゴリズムに対して、演算性能、メモリアクセス性能については想定するアーキテクチャでの演算カーネルの性能評価、それに基づいたアーキテクチャ、アルゴリズム双方の改良を行う。ネットワークについても同様に、想定アーキテクチャのレイテンシも考慮した性能評価を行う。

スケジュール

初年度
　商用および独自コードのアルゴリズム調査・カーネル抽出、ヘテロジニアスアーキテクチャでの性能評価
2年度
　アーキテクチャ・システムソフトウェア・アプリケーショングループの共同による性能評価/アルゴリズム、設計改良、性能評価の繰り返し

アーキテクチャ調査研究の概要

当初の目標

- 分野自体の技術トレンド、半導体プロセス技術動向からポスト富岳時代に実現可能なシステムがどのようなものかを検討し、評価対象アプリケーションによるシステム評価を行う
- アクセラレータ側: 現時点で同じ半導体技術では世界最高の電力当り性能を実現している MN-Core をベースに 一層高い電力性能、チップ面積あたり性能を目指すと共に、アプリケーションとのコデザイン により高い実行効率を実現する
- 汎用プロセッサ側: RISC-V ベースの独自設計で世界最高レベルの性能を目指す。

アーキテクチャ調査研究の主要な成果

- 半導体プロセス、各 GPU・CPU ベンダ、AI プロセッサベンチャーの現状の検討と将来予測を行った。
- MN-Core アーキテクチャでの将来システムの提案。特に 3DDRAM 利用によるバンド幅向上・電力性能向上可能性の評価を行った。
- CPU 評価。RISC-V 中心に利用可能なコアの性能評価を行った。

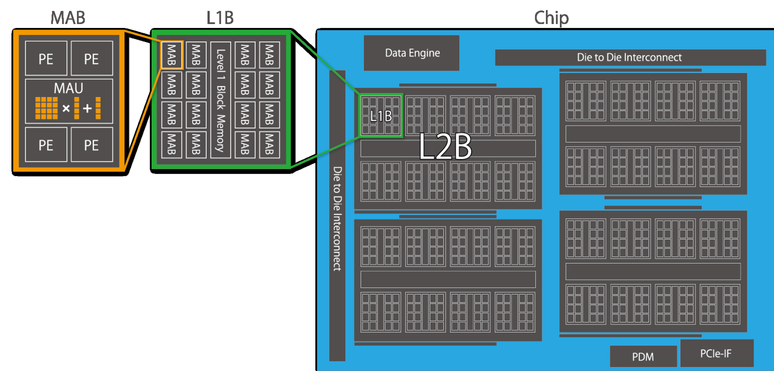
アクセラレータの予測と比較

予測	NVIDIA	AMD	MN-Core
製品発表年	2028	2028	2028
プロセス (量産開始年)	A16(26)	A16(26)	A16(26)
演算性能@FP16(TF)	5480	3289	(非公開)
演算性能@ FP64(TF)	98	207	(非公開)
FP16/FP64	55	16	16
周波数 (MHz)	2000	2625	(非公開)
TDP(W)	1500	1000	(非公開)
電力効率 GF/W@FP64	65.3	207	(非公開)
Die Size(mm2)	1600	1100(x2)	(非公開)
DRAM サイズ (GB)	384	384	(非公開)
DRAM バンド幅 (TB/s)	(24)	(12.8)	> 100
電力効率 GB/J@FP64	16	12.8	> 100

他社の DRAM バンド幅と電力は、HBM (Bufferless でも) と現在の階層キャッシュアーキテクチャを継続するなら同時には実現困難。

MN-Core 後継アーキテクチャは他社と比べて FP64 電力性能で大きく上回り、電力あたりメモリバンド幅で 5 倍以上を実現可能。

MN-Core アーキテクチャ



- トップレベル: 4 チップ内に 4 個 (合計 16 個) の L2B(レベル 2 放送ブロック)
- L2B の中: 8 個の L1B(レベル 1 放送ブロック)
- L1B の中: 16 個の MAB(行列演算ブロック)
- MAB の中: 1 つの MAU(行列演算ユニット) と 4 個の PE(プロセッシングエレメント)

ポスト富岳向け提案では基本的に数を増やし、クロックをあげる。また、PE/MAB レベルで DRAM をつける。

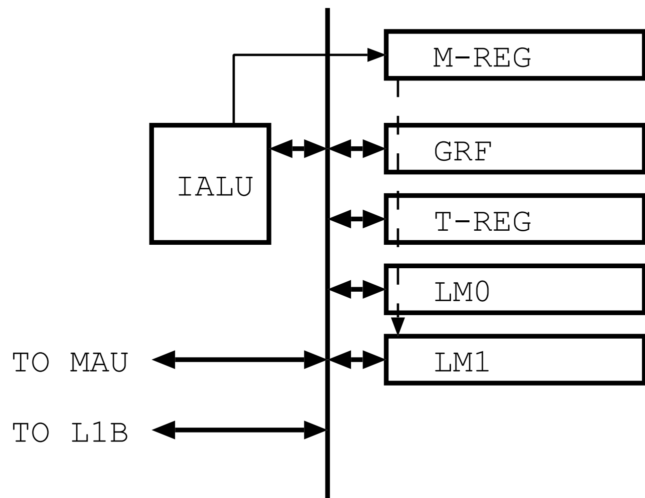
アーキテクチャ続き

- 概念的には全体が同期して単一の命令流を処理する。 L2B 以下は実際にクロック同期。チップ間を含めて L2B 間は (周波数は同じだが) 命令実行にずれがあることを許す。
- チップ毎に DRAM ブロックがある。DRAM は L2B とブロック転送でデータ交換できる。放送、個別転送、チップ内分配等いくつかのモードがある。
- ポスト富岳向け提案では DRAM は個別の PE にアドレッシング可能なメモリとしてつく。(複数の PE でアドレスをシェアする可能性はある)

詳細構造

- PE(プロセッシングエレメント)
- L1B(レベル１放送ブロック)
- L2B(レベル２放送ブロック)
- 全体

PE(プロセッシングエレメント)(1)



- 演算器は IALU と MAU
- MAU は倍精度、単精度、半精度精度の行列ベクトル積を実行
- L2B の中: 8 個の L1B(レベル 1 放送ブロック)
- L1B の中: 16 個の MAB(行列演算ブロック)

- バスみたいに書いてますが実際の回路はマルチプレクサ
- GRF は (MN-Core では) 1R1W 2 ポート。LMx はシングルポート。
- T-reg: 補助レジスタ。1R1W で 1 ベクトル分

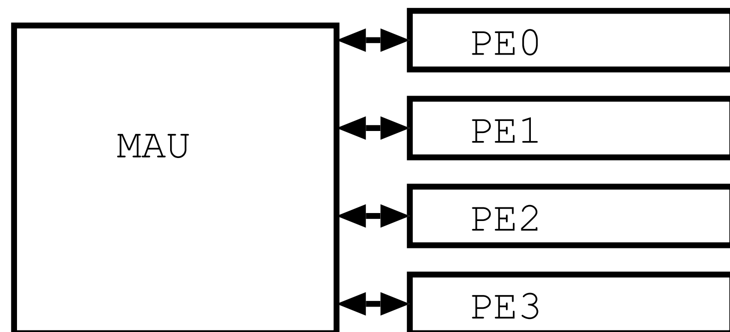
PE(プロセッシングエレメント)(2)

- LMx (ローカルメモリ) は 2048 64 ビット語 x 2、 GRF は 512 語
- ベクトル長 4 の固定長ベクトル命令。
- ポスト富岳向け提案では LM1 を DRAM に置き換える。
- 直前の命令の演算結果を入力オペランドにとることが可能 (フィードバック)
- ベクトル命令でのアドレッシングは固定、連続アドレス、ストライドアクセスが可能 (ベクトルレジスタもストライドアクセス可能)
- T-レジスタを使った間接アドレスも可能
- アドレスはベースアドレスレジスタで修飾
- IALU の演算結果から大小比較、一致比較等のフラグベクトルを生成して M レジスタに格納可能

PE(プロセッシングエレメント)(3)

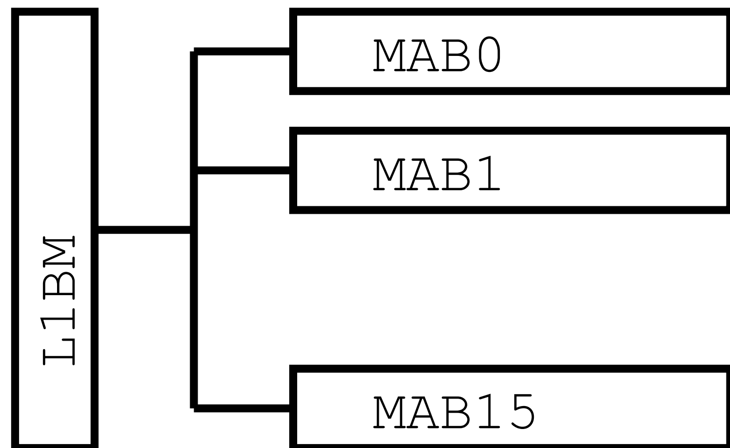
- M レジスタの値でメモリ、レジスタに書き込むかどうかを制御
- ロードストアアーキテクチャではなく、LMx の読み出し結果を直接演算器の入力にできる。また、演算器の出力をメモリに直接格納できる。複数のユニットに格納もできる。
- この機能と、演算結果フィードバック、T レジスタを最大限利用することで、通常のアーキテクチャに比べてレジスタファイルのハードウェア規模、消費電力を大きく削減している。また、スーパースカラー実行制御、OoO 実行のための回路も不要となる。=高い電力性能の実現に大きく貢献

MAB(行列演算ブロック)



- PE4 個が 1 つの行列乗算器を共有
- サイクル毎に行列ベクトル積を実行。
  d= A*b +c
- MAU は行列レジスタをもつ
- 行列レジスタは複数あり、演算中に別の面への書き込みが可能
- 転置しながら格納も可能
- 倍精度、単精度、半精度行列に対応。サイクルあたり演算数は 32, 128, 512
- ベクトル演算 (PE から見ると 64 ビットスカラー演算) モードもある
- 倍精度、単精度、半精度行列乗算器は同一の乗算器の構成変更で実現 (特許取得済): 他社の回路に比べると同一性能で回路規模が半分以下

L1B(レベル1放送ブロック)



- MAB 16 個が 1 つの L1BM(レベル 1 放送メモリ) に接続。
- L1BM から読み出されたデータは放送される (各 PE/MAB での書き込み制御で単一の PE/MAB に送ることもできる)。
- 分配モードも一応ある (あんまり速くならない)
- 各 PE から読み出したデータは、縮約して L1BM に書き込むことができる。単一の PE/MAB を指定した読出しもできる。分配の逆操作 (結合) も一応ある。
- この辺特許取得したが今年で切れた
- PE 間の直接結合はない。

L1B の特色

- 明示的な放送/縮約モードがあるため、複数 PE を使った細粒度並列処理を低オーバーヘッドで実行できる。
- GPU が非常に苦手とする reduction をオーバーヘッドを気にしないで使うことができる。
- 例えば比較的小さな行列乗算を MAB に分散させることができる。総和が遅いと、行列行列積 A*B で B を縦に切る並列化をすると、各 MAB で A は全体が必要になる。総和が速いと、B を横に切る並列化ができ、A を分散させられる。また、単一の行列ベクトル積の並列化ができる。推論 (LLM でも) で非常に効率をあげやすくなる。
- (PE 命令と同期した、4 サイクル毎のデータ転送命令で制御)

L2B(レベル２放送ブロック)

- L1B ８個に対して L2BM(レベル２放送メモリ) が１つ。
- L1B 内と同様な放送・縮約・分配・結合・個別読出しを L1B に対して行う。
- データ幅は広くしている。
- いくつかの L1B 同士の直接データ交換モードがある。
- PE 命令と同期した、４サイクル毎のデータ転送命令で制御。

全体

ポスト富岳向け提案では

- L2B が複数、PCIe インターフェースも複数ある。ある。
- L2B 間は個別転送、放送、縮約が可能 (ポスト富岳向けではさらに近接通信を導入)

PE命令の概要

PE 命令は

- 各演算の動作モード、入力セレクト、各メモリユニットのアドレス、LM では read/write モード、入力セレクト、各メモリユニットの入力マスクレジスタ番号、書き込みマスクレジスタ番号等
- L1BM-PE, L2BM-L1BM の転送モード、転送アドレス等

を指定する。

実行可能コードの(現状の)制限

- ループ回転数等はカーネル関数コール時にきまっている必要がある (ホストコードコンパイル時ではない)。また while ループは実行できない。
- これは現状の制限で、ポスト富岳の時期には撤廃する (ボード上のシーケンサから命令を発行することでこれらを制御する)。

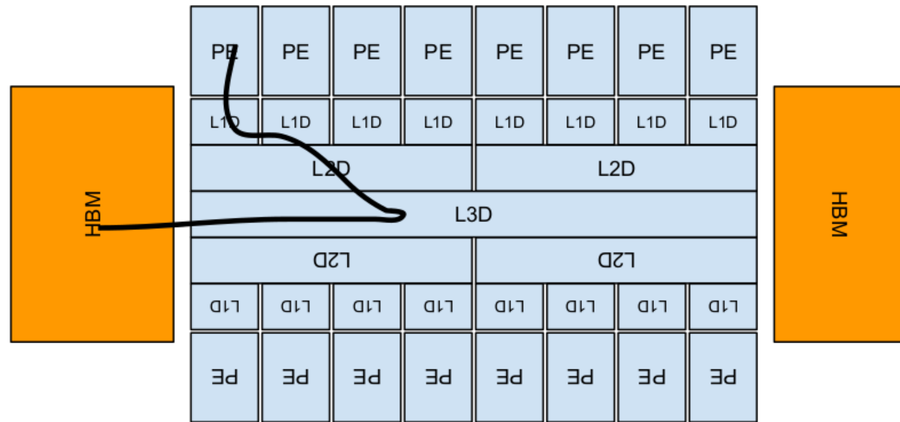
ループ内でストライドアクセス、 a[i-1] のようなシフトも可能。「ローカルメモリの中では」ベクトルプロセッサでできたことがほぼなんでもできる。

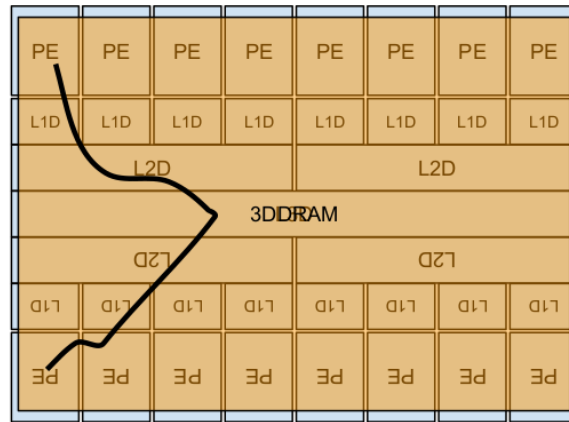
MN-Core の電力性能、電力あたりメモリバンド幅が高い理由

- チップ内メモリが分散して PE(演算素子) 直結となっているため、演算に必要なデータ移動距離 (物理的な長さ) が短く、またキャッシュがないので有効でないデータ移動も少ない
- DRAM も本質的に同じで、3D 積層による分散アーキテクチャのため物理的なデータ移動距離が短い。

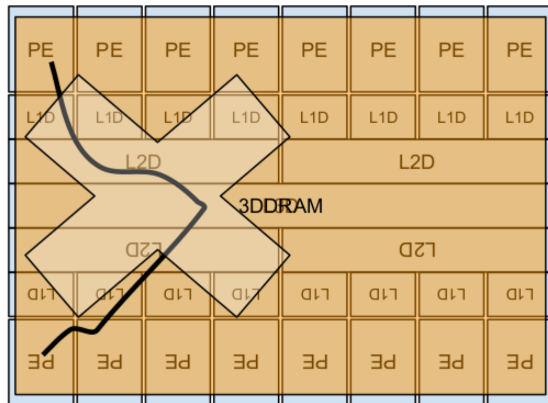
3D DRAM の有効活用にはアーキテクチャの転換が必要な理由

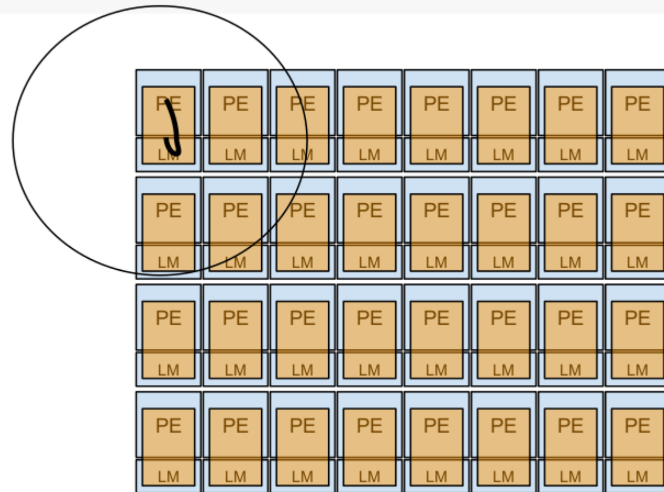
本質的な理由: 階層キャッシュによる共有メモリアーキテクチャはそれ自体がメモリアクセスのボトルネックを持つ。3D でもデータ移動距離があんまりかわらない。





共有メモリと分散メモリの違い





物理的なデータ移動距離を１桁以上短くできる。

メモリアクセスのエネルギー

- DRAM セルの容量 : $10 \sim 20$ fF $\rightarrow$ アクセスの消費電力は 10fJ 程度
- 1 ゲートの消費電力: 1 ゲートあたり $0.1 \sim 1$ fJ 程度
- 配線の寄生容量: 1cm あたり 2pF: アクセスの消費電力は 1pJ 程度。現状では 10pJ に近い。

つまり:

- 現在の DRAM およびロジック製造技術でも、メモリアクセスエネルギーはもう 2-3 桁下げる余地がある (配線長を 1mm 以下にすれば)
- Memory Wall(Wulf and McKee 1995, Wilkes 1995) は 3 次元実装と分散メモリアーキテクチャで本当に解消する。BF=4 の計算機も将来にわたって可能。

冷却の問題

以下のような主張をみることがある

- HBM ですでにそうであるように、3D 積層構造では冷却が最大の問題
- 横につむのをやめて、縦構造(細いメモリダイを一杯並べる) にしないといけない

実際には

- 現状の HBM で熱抵抗が大きい層はマイクロバンプ接合を樹脂で埋めているところ。厚さ 30$\mu\mathrm{m}$ 程度あり熱伝導率が小さい。ハイブリッドボンディングではこの層がないため、1 層あたりの熱抵抗が 1 桁下がる
- このため、 $1\mathrm{W}/\mathrm{mm}^2$ 程度の発熱でも温度上昇は 1K/層 程度。16 層程度までは大きな問題ではない。

DRAM 3D実装技術の現状

	HBM	Custom HBM	Bufferless HBM	Customized Bufferless	Full Custom CoW(etc)	Full Custom WoW
信号数	2K	2K	2K	4-8K?	> 10K	> 100K
電力 pj/bit	3-4	2-3?	2-3?	1-2?	~ 1	0.25-1
時期	now	2026?	2027?	2026?	now?	now?
NRE	-	非常に高い	非常に高い	高い	高い	安い可能性
量産コスト	高	高	高	高	中	低
容量	大	大	大	中	小 - 中	小 - 中
ロジック	最先端	最先端	最先端	微妙	古い	古い

- HBM は世代が進んでも大きくアクセスエネルギーが下がらない。このため、多様なソリューションが検討されている。
- 原理的にアクセスエネルギーを最小にできるのは、信号数を増やせる WoW
- 台湾ベンダは WoW による 3D 積層に積極的になってきた。現状で 1z 世代、ポスト富岳の時期にはその次くらいが利用可能。

中国の動向

- 中国では、AMSL の装置の輸入・稼働ができなくなっているため、微細化が止まっている。また、HBM の輸入も困難になりつつある。
- これに対する対策として、
  - DRAM ファブへの巨額投資。YMTC CXMT。CXMT は 1z 世代
  - HB による 3D 積層

  の開発が急速に進んでいる。
- 2.5D の HBM に比べた 3D 積層 DRAM の優位性は非常に大きいため、今後数年で AI および HPC で中国が優位に立つ可能性がある。
- (微細化による電力削減効果は小さくなっているため。ロジックでも 3D 積層化で電力削減、速度向上が実現できる)

アーキテクチャ調査研究で得られた特に重要な知見

- HBMx を使っても、「プロセッサチップの横に共有メモリを提供する DRAM がある階層キャッシュアーキテクチャ」自体が電力性能向上への足枷になっている。
- DRAM 3D 積層は、製造技術の側から電力性能向上を実現する解を与える。これの有効利用には、物理的な共有メモリから分散メモリへのアーキテクチャの変革が必須になる。
- 3DDRAM と分散メモリアーキテクチャ化により、現在の技術でも 1 桁、将来的には 2 桁以上 DRAM のアクセスエネルギーは下がる。
- MN-Core アーキテクチャはこの変革に容易に対応できる。GPU、他の AI むけプロセッサは未知数である。

システムソフトウェア・ライブラリ調査研究の概要

当初の調査内容

- DSL・フレームワークによる高効率コードの自動生成に重点をおく。
- 深層学習は Pytorch, JAX 等からのコード生成。
- 他のアプリケーションでも同様の方向での高い効率と開発の容易さを実現する可能性の評価に重点をおく。

実際の調査研究

- 深層学習については当初の方針通り
- 一般の HPC アプリケーションについては、
  - DSL が適しているもの
  - 行列演算ライブラリ、FFT ライブラリ等のライブラリ利用が中心になるもの
  - どちらでもなく、汎用言語でのアプリケーションカーネルの記述が必要なもの

  に対してそれぞれ対応する。

このため、
(1)DSL 性能評価、 (2) ライブラリ性能評価、 (3) 汎用言語の設計・評価
をそれぞれ進める。

アプリケーションとアプローチの対応

アプリケーション	タイプ	
創薬と深層学習応用	粒子・深層学習	DSL
ゲノム科学	ツリー探索	汎用環境
地震と構造物	FEM(EbE)	汎用環境
気象・気候	規則格子差分	汎用環境
ものづくり	疎/密行列	ライブラリ
マテリアルサイエンス	密行列	ライブラリ
素粒子・原子核	規則格子差分	汎用環境
宇宙・惑星科学	粒子	DSL

- 規則格子向けの DSL を当初検討していたが、OpenACC ライクな言語環境で性能的には十分であり、記述の柔軟性が高いと判断した。
- ツリー探索や EbE 法は DSL よりも低レベルの記述が現実的である

システムソフトウェア・ライブラリ調査研究の主要な成果 (1) DSL

- 粒子法については、オフロードモデルを想定した。これは現在 GPU を使っているコードのほとんどが、粒子間相互作用計算のみを GPU で行うオフロードモデルであること、またシステム提案が CPU 側の演算能力、バンド幅が十分あり、オフロードモデルで高い効率を期待できることによる。(性能評価はアプリケーションのところで)
- 粒子間相互作用について、高レベルの記述から最適化されたカーネルのアセンブリコード (理論限界の性能がでる) を生成するコンパイラと、ホストとのデータ通信等の関数を自動生成する DSL を開発した。これは理研 RCCS 粒子系シミュレータ開発チームで開発していた FDPS/PIKG がベースである。
- 深層学習については、PFN 社内で開発してきた PyTorch からのコード生成で十分な性能がでることを確認できた。

FDPS/PIKG

粒子法については、理研-神戸大学で開発してきたフレームワーク FDPS の相互作用計算カーネルコンパイラ PIKG を、MN-Core の最適化したコードを生成する。

```
rij = EPI.pos - EPJ.pos
r2 = rij * rij + eps2
r_inv  = rsqrt(r2)
r2_inv = r_inv * r_inv
mr_inv  = EPJ.mass * r_inv
mr3_inv = r2_inv * mr_inv
FORCE.acc -= mr3_inv * rij
FORCE.pot -= mr_inv
```

左のコードから最適化した命令列を生成する。MN-Core での命令数 (実行時間そのもの) は反復あたり 23 命令であり、人間がアセンブリ言語を限界まで最適化したものと同じ命令数となった。

カーネル部分だけの実行効率はベクトル性能ピークに対して 65.2% (行列乗算ピーク性能に対して 16.3%) となる。

コード全体の実行効率についてはアプリケーション調査研究のところで触れる。

システムソフトウェア・ライブラリ調査研究の主要な成果 (2) ライブラリ

- 基本的な数値ライブラリとして、密行列向けの LAPACK と、FFT ライブラリを開発中である。どちらも、PE/MAB 単位の中では 80% から 95% 程度の実行効率が得られている。
- 特に、FFT でも、行列乗算ユニットを有効に使って 2 段 (倍精度) や 3 段 (単精度) のバタフライ演算を一度に行なうことで、ベクトル演算の場合よりも高速化できるため、ベクトル演算で性能が落ちるという欠点が大きな問題になりにくいこと、むしろ、ローカルメモリのバンド幅が高いため、GPU や CPU よりも高い実行効率が得られることがわかった。
- 最終報告では PE 間通信も考慮した性能推定を与える。多くの場合に通信は隠蔽でき、また提案システムは B/F=0.2 と高い DRAM バンド幅と大きなオンチップメモリを持つため、大きな FFT でも高い実行効率が得られることがわかった。

システムソフトウェア・ライブラリ調査研究の主要な成果 (3) 汎用言語環境

- Cuda/OpenCL 風言語、OpenACC 風コンパイラ指示によるアクセラレータへのオフロードの両方の検討を進めてきた。
- それぞれについて、これまでの検討で得られた知見をまとめる。

Cuda/OpenCL 風環境

どのような実装がよいかの比較検討のため、複数の実装を試みた。どれも LLVM ベースである。

- 会津大学によるもの
- 神戸大学の作成したものおよびそれをベースに外部委託開発したもの
- PFN 社内で独自開発したもの

これらの比較検討により多くの重要な知見が得られた。

重要な知見(1)

LLVM-IR から MN-Core アセンブリ言語 (vsm) への変換で十分に最適化されたコードをだすことはそれほど難しくない。

- MN-Core では固定長ベクトル命令を採用することで、命令スケジューリングを不要にしている (直前の命令の結果を使える)。LLVM は固定長ベクトル命令 (SIMD 命令) に対応しているので命令生成できる。
- 条件分岐はマスク実行に変換する必要があるが、この機能を LLVM がもっている (SIMD 命令用のマスク付きコード生成ができる)。
- 現状ではアドレス計算はコンパイル時に解決され、(ループは展開される) ロード・ストア命令は次に述べるように消去されるため、生成される命令は基本的に演算命令だけになる。(袖交換等は別)
- このため、生成されるカーネルコードの実行効率は、ベクトル性能に対して最低でも 50%、典型的には 70% 前後になる。

重要な知見(2)

通常のコンパイラバックエンドにはない特異な処理が必要である。

- LLVM はロード・ストアアーキテクチャを仮定しているが、MN-Core はローカルメモリもオペランドにとれるため、ロード・ストア命令は不要である。このため、オペランドの書換えによってロード・ストアを消去する必要がある。
- MN-Core は複数のレジスタファイルとローカルメモリが命令セットから見える。このため、変数を適切に複数のユニットにわりつけてハザードを解決する必要がある。これはグラフ彩色問題に帰着でき、実用的な時間で十分良い解を求められる。

重要な知見(3)

大規模 SIMD アーキテクチャ一般の制約に加えて、現在の MN-Core の実装の持つ制約がある。特に

- 現在のところ実行時にループ反復回数を指定できない
- while ループを実装できない

これらは、機械学習では学習でも推論でも基本的に不要なため実装されておらず、現在の PFN 内製の命令生成系は ONNX 等のネットワーク記述からループが完全にアンロールされた命令列を生成する。
ポスト富岳向けシステムでは簡単なシーケンサを実装して実行時に回数を指定するループや while ループを可能にする。
MN-Core の前身である GRAPE-DR ではループ機能がオンボード FPGA によって実装されており、同様な実装を想定している。

OpenACC 風環境

- OpenCL に変換するものと、直接 vsm を生成するものの２バージョンを開発・評価中である。
- ある程度の複雑なコードをコンパイル・実行可能になっている。
- 分散メモリなので、プラグマが OpenACC というより HPF 的なものになり、領域分割の方法を指定する。

アプリケーション調査研究の概要

当初の調査内容

- 重要なアプリケーション、アプリケーションカーネルの抽出、 次世代システム向けアルゴリズムの検討と性能評価を合わせて行う。伝統的な HPC アプリケーションだけでなく、AI 応用、OSS 等についても十分な検討を行う。
- 抽出したアプリケーション/アルゴリズムに対して、演算性能、メモリアクセス性能については想定するアーキテクチャでの演算カーネルの性能評価、それに基づいたアーキテクチャ、アルゴリズム双方の改良を行う。ネットワークについても同様に、想定アーキテクチャのレイテンシも考慮した性能評価を行う。

アプリケーションとアプローチの対応(再掲)

アプリケーション	タイプ	
創薬と深層学習応用	粒子・深層学習	DSL
ゲノム科学	ツリー探索	汎用環境
地震と構造物	FEM(EbE)	汎用環境
気象・気候	規則格子差分	汎用環境
ものづくり	疎/密行列	ライブラリ
マテリアルサイエンス	密行列	ライブラリ
素粒子・原子核	規則格子差分	汎用環境
宇宙・惑星科学	粒子	DSL

- 規則格子向けの DSL を当初検討していたが、OpenACC ライクな言語環境で性能的には十分であり、記述の柔軟性が高いと判断した。
- ツリー探索や EbE 法は DSL よりも低レベルの記述が現実的である

創薬と深層学習応用アプリケーションの検討（株式会社Preferred Networks）

深層学習については今後重要となる LLM 推論について性能推定をおこなった。
提案システムの FP8 性能に対する B/F 値が高いため、LLM 推論でも 高い実行効率が得られる。
創薬で重要なアプリケーションは今後は AlphaFold や Matlantis のような機械学習ベースのものの役割が増えていくと考えられるが、これまで重要であった古典MDの評価も行った。 LAMMPS を例に短距離相互作用の実装と性能評価をおこなったが、間接アクセスを必要とするデータ構造が MN-Core 向きではないことがわかった。適切なデータ構造への変換は容易であり、その変換を行った結果であるFDPSの評価を宇宙アプリケーションのところで述べる。

ゲノム科学アプリケーションの検討 （国立大学法人東京大学）

NGS の生成するショートリードのマッピングを想定して評価を進めている。現在広く使われている (富岳でもターゲットアプリケーション) BWA-MEM は 1 塩基毎に大きなテーブルのアクセスを複数回行うため、キャッシュが有効に機能せず処理が遅い。このため、ツリー探索を行う新しいアルゴリズムを評価した。ローカルメモリを有効に使うようにアルゴリズムを書き換えることで、高い性能が得られるとわかった。現在 GPU クラスタで数時間かかっている ヒトゲノム x35 データの処理時間を数分まで短縮できる見込みが得られた。

地震と構造物シミュレーションアプリケーションの検討 （国立研究開発法人海洋研究開発機構）

地震波伝搬シミュレーションで使われている EbE 法による有限要素法カーネルについて実行効率の評価を進めている。 EbE 法ではある程度メモリバンド幅が低い場合でも高い実行効率が得られるため、 A64fx や A100 と同様の理論ピークに対して 10% 程度ないしそれより高い実行効率が期待できる。
実際にベンチマークコードを机上検討した結果では、MN-Core では EbE 法が必要とする主記憶への間接アクセスを高速に実行できるため、主記憶アクセス時間が計算時間に隠蔽可能であるとわかった。このため、実行効率はベクトル性能に対して 80% 程度 (積和にならない部分があるため)、理論ピークの 20% 程度が期待できる。

気象・気候シミュレーションアプリケーションの検討 （国立研究開発法人国立環境研究所）

NICAM ベースのベンチマークコードである IcoAtmosBenchmark について、現行の PFN 開発環境での MN-Core への実装と性能評価をおこなった。ベクトル演算での理論ピーク性能 8TF に対して、3 つのカーネルで 4.178TF, 1.9TF, 3.922 TF と 25-50% の効率が実現できた。これはもちろんデータがオンチップ SRAM にはいる場合であるが、ポスト富岳提案では仮に DRAM にデータがあってもそれを SRAM にロードする時間はほぼ無視できるようになるため、同等の効率が期待できる。興味深いことは、全てのベンチマークカーネルで A64fx よりも高い実行効率が実現できていることであり、大きなオンチップメモリをソフトウェアが有効利用できる提案アーキテクチャの特性があらわれている。

ものづくりアプリケーションの検討 （学校法人東洋大学）

昨年度までで、有限要素法の部分領域ソルバに対してLocal Schur Compliment(LSC)アプローチを適用し、これにより反復において疎行列ベクトル積や行列分解を行うのではなく、対称密行列ベクトル積演算だけを行う方法を開発した。これにより、計算時間のほぼ全部となるDDM 反復の間の演算は固定係数の行列ベクトル積で。行列側の要求 B/F は2となる。
この場合、この B/F にみあう実行効率が期待できる。

マテリアルサイエンス応用アプリケーションの検討（株式会社Quemix）

RSDFT に対して評価を進めている。基本的に密行列処理で計算量が大きいため、
50% 程度の実行効率が期待できる。

素粒子・原子核物理応用アプリケーションの検討（国立大学法人広島大学）

CCS QCD Solver Benchmark をベースに評価を進めてきている。現行のMN-Coreでは $16^3 \times 32$ の格子を1カードで持つ時、全データがオンチップSRAM にはいり、2000 回反復での実行時間が 1.6 秒となる。実行効率は15% 程度の見込みである。

宇宙・惑星科学応用アプリケーションの検討（国立大学法人東京大学）

FDPS をベースに、先に述べた DSL によるカーネルを使った実コードでの評価を行った。
カーネル部分の、ベクトル性能ピークに対する実行効率 65.2% に対して、コード全体の効率は 4.0% と 1/15 となった。この理由は、

- ホスト CPU が 6 コアと遅く、ホスト実行時間が全体の 60%である。つまり、2.5 倍遅くなっている。これは提案システムではずっと強力なコアを 128 個であり、アクセラレータの性能向上を考慮しても大きく改善される。
- 現在の MN-Core コード生成系が固定回数ループしか実行できないため、実際の引数に対してあらかじめ用意した引数パターンの中からあうものを呼ぶ、というアプローチであり、平均して計算量が 3-4 倍に増えている。

  これは、実行時に回数指定可能になり、またループの構造を変えることでオーバーヘッドを有効な計算時間の 20% 程度にできると期待している。
- 通信が完全には隠蔽できていない。

  これは、通信性能が相対的に大きく改善するため問題にならない。

これらを修正することで 10 倍程度の効率向上が可能であり、40% 程度の効率が期待できる。行列乗算ピークに対しては 10% 程度となる。
将来的には、行列演算ユニットの有効利用を進める (FMM 等の行列ユニットを有効に使えるアルゴリズムにする) ことで実行効率、計算速度をさらに改善する。

アプリケーションまとめ

- 個々のアプリケーションの数値はここでは省略する。
- 実行効率 10% 程度から 50% 程度まで様々だが、多くのアプリケーションで性能は DRAM バンド幅リミットではなく、演算器のベクトル演算性能が行列演算性能に対して低いことによる。
- FP64 や FP32 のベクトル演算性能をあげることはピーク電力を増やさずに可能なので、 HPC むけにはそのようなアーキテクチャが望ましいといえる。

アプリケーションポーティングの大変さと実行効率

既に述べたように、アプリケーションポーティングの内容はアプリケーションのタイプ等で 3 つに分かれる。

1. DSL 型: 深層学習での PyTorch のようにアクセラレータで実行されるコード全体 (といってもも行列行列積が主体) を DSL で書けるもの、あるいは粒子法のように、計算量のほとんどを占める部分 (粒子法では相互作用計算) をオフロードするもの。
2. ライブラリ型: DSL 型と同様に計算量の大半を占めるところをオフロードするが、その部分が行列の対角化や分解、FFT、疎行列ベクトル積等のライブラリコールで実現できるもの。
3. それ以外: さらに規則格子差分法とそれ以外に分かれる。

実行効率とポーティングの大変さ

1. DSL 型: 効率は高く、ポーティングはそれほど大変ではない。
2. ライブラリ型: 効率は高く、ポーティングはそれほど大変ではない。
3. それ以外:
    - 規則格子型: 効率は高く、ポーティングはそこそこ大変
    - それ以外: 効率も大変さもアプリケーションによる。

DSL型・ライブラリ型

- 深層学習では、PyTorch で書かれたものが CPU でも GPU でも他の AI アクセラレータでも動く。アプリケーションを書く側からみるとポーティングは特に大変ではない (各アーキテクチャ向けのコンパイラ開発は大変)
- 粒子のオフロードでも、DSL の相互作用計算部分の「コンパイラ」があればよい。実際に FDPS/PIKG では GPU と MN-Core で同一のコードが動作する。
- ライブラリ型も同様で、場合によってはライブラリのインターフェース API が違うがその書換え程度で動作する。

このようなタイプはかなり多いと考えられる。

それ以外:規則格子差分法

- 規則格子差分法は、工学応用はあまり多くないかもしれない (が、FDTD 法等重要なアプリケーションもある) が、素粒子、宇宙、気象の大規模シミュレーションでは広く使われている。
- 細かいバリエーションがあるので DSL を定義するよりも OpenACC (ないし XcalableMP/HPF) のような配列オリエンテッドな並列言語での記述が適切な抽象度であると考えられる。

それ以外: それ以外

- それ以外では、間接参照が発生するものが多い。が、間接参照を PE の中に閉じられる、あるいは前処理で対応可能なものが多い。以下にその例をあげる。
- ゲノム解析では、ツリー構造を辿るため、そのままのアルゴリズムでは反復回数が不定で実行時に判定が必要である。但し、通信リミットであることがわかっていると、固定回数反復で、終了していないものはホストで処理で十分であり、コードは PE 内で閉じるため特に困難はない。
- EbE 法では、計算実行部分はコンパイラで問題なくコード生成可能だが、間接参照部分には前処理が必要になる。前処理によって通信コードを生成すれば問題なく実行できる。

まとめ

- 神戸大学チームでは、2.5D の HBM が性能の限界にきていること、 3D 積層 DRAM が急速に発展していることから、 DRAM ダイをプロセッサダイにスタックし、高いメモリバンド幅を実現するアーキテクチャについて、その実現性をハードウェア、システムソフトウェア、アプリケーションの観点から検討した。
- 性能については、積層 DRAM の採用により DRAM アクセスの消費エネルギーを少なくとも１桁、将来的には２桁以上減らせることがわかった。逆にいうと、同じ電力で DRAM バンド幅を 1-2 桁増やせることがわかった。
- コンパイラ開発には大きな困難はない。実行効率は、ローカルメモリ、DRAM とも高いバンド幅をもつために高くなる。
- アプリケーションは、多くは GPU むけと同様なオフロードで実現でき、また、ライブラリの利用や、 OpenACC のようなプラグマベースで実現できる。

パネルディスカッションへのコメント

1) 今回の次世代計算基盤に係る調査研究 (FS) の良かった点（どう機能したか）・改善すべき点

2) 今後の次世代計算基盤開発・整備の在り方（提言）FS の在り方、更新期間、設置台数、構成（富士山型・八ヶ岳型）、アーキ・システムソフト・システム設計・アプリ・利用者、人材育成からの観点等 •••

今回の次世代計算基盤に係る調査研究(FS)の良かった点（どう機能したか）・改善すべき点

- 「京」以降の国策スパコン開発の根本的な問題: アーキテクチャの進歩を妨げている
  - 「京」: アクセラレータの時代になることは予見できていたにもかかわらずスカラー CPU を採用
  - 富岳: 汎用スカラー CPU の時代では既になかったにもかかわらずスカラー CPU を採用
  - ポスト富岳: ？
- 利用拡大につながらない理由の根本はここで、過去のアーキテクチャであるためにコストパフォーマンスが悪い。

今回の次世代計算基盤に係る調査研究(FS)の良かった点（どう機能したか）・改善すべき点(続き)

- これが問題として認識されていないことがまず問題である。
- 「これが最善だった」「これしかなかった」という主張はいくらでもできるが、だからといって問題がなくなるわけでではない。

今回良かった点

- PFN の中でも、HPC のプログラムは PyTorch で書けるものではないらしいということが多少理解されるようになった。

今後の次世代計算基盤開発・整備の在り方（提言）

- (少なくとも数値風洞がそうであったように) アーキテクチャの進歩、革新につながるものでなければならない。
- 何故今まではそうならなかったか、どうすればそれができるか？に答をださないと、同じことの繰り返しになる。
- (牧野個人としては何かいっても何もおきないという気もするので、別のアプローチもとっている)

予備スライド

LLM からくるDRAM バンド幅向上の要請とその困難

典型的な transformer では、メモリアクセスはバッチあたりで

$$10C^2 + BLC/4 \tag{1}$$

計算量は

$$[20C^2 + LC/4)]B \tag{2}$$

C: チャネル数、8192 くらい。B: バッチサイズ、L: コンテクスト長(文章の長さ)、あといくつか仮定あり。
モデルパラメータにほとんど依存しないで、

$$\mathrm{B/F} = \begin{cases} 1/(2B) & (\ L \ll C/B) \\ 1 & (L \gg C/B) \end{cases} \tag{3}$$

LLM からくるDRAM バンド幅向上の要請とその困難(続き)

つまり、

- コンテクスト長が小さいうちは、バッチサイズを大きくすることで要求 B/F を小さくし、効率をあげることができる。
- しかし、$L \sim C/B$ くらいから B/F 要求が大きくなるので、大きなバッチサイズを要求するシステムでは長いコンテクストは扱えない。
- とにかくハードウェアのメモリバンド幅をあげるしかなくなっている。
- 近年の GPU の消費電力増加の大きな要因になっている。

大規模 SIMD アーキテクチャでは基本的な制御構造はどのように実現されるか？どのようなメリット・デメリットがあるか。

- 3 つの「制御構造」: 順次、選択、反復
- 「順次」は特に考えることはない
- 選択: 通常の if 文。SIMD マシンでは、
  - 両方を実行する
  - それぞれの条件が成り立っていないプロセッサではマスクして命令実行しない

反復

- 反復: for や while。古典的な大規模 SIMD マシンでは、
  - 全てのプロセッサで終了条件が満たされるまで繰り返す
  - 終了条件が満たされたプロセッサではマスクして命令実行しない

現状の MN-Core での選択

選択

- 実際にマスクレジスタで実現。AVX512 や SVE の predicate と同じ (古典的ベクトルプロセッサの vector mask register と同様)
- 両方実行することによる効率低下は、通常の分岐であればSIMD 化されたループでの AVX512 や SVE の効率低下と大きく変わらない。

現状の MN-Core での反復

- 現状では、実行コードの一部を反復する機能はない。
- これは、機械学習ではほとんどの場合ループ等も実行前に反復回数がわかるものであり、アンロールした命令生成が可能だから。
- 古典的 SIMD プロセッサの多くがやっていたように、専用のシーケンサで反復制御を行なうことはできる。通信の実現のため実装の方向で検討中。
- 実際、アーキテクチャ的に MN-Core の原型である GRAPE-DR では外付け FPGA からの命令供給による固定回数ループを実現している。 MN-Core ではこれまでやってなかったのは深層学習では不要だったからにすぎない。

制御構造についてのポイント

- 選択、反復どちらも実現可能である。
- 理想的なスカラープロセッサ (分岐のペナルティが 0 の、、、) に比べると必ず性能低下はある
- 選択については、ベクトルプロセッサや AVX512、SVE 命令での SIMD 化で起こる効率低下と同程度。とはいえ、簡単な分岐ではスカラープロセッサでも両方計算して条件によって選択するほうが分岐で実装するよりも速く、事実上ペナルティはない。
- (回数不明の) 反復では、性能ペナルティはそれなりに大きくなりえる。但しこれは実装による。

MN-Core では何が「できない」か？

- 制御構造については基本的には制限はない。
- メモリアクセスについてはハードウェアでは各 PE は自分のローカルメモリしかアドレッシングできない。古典的 SIMD マシンのようなルーティングネットワークをもたない。

これは制限になるか？

- 実際には、現在のチップ内ネットワークのソフトウェア制御で PE 間のランダムな通信 (put/get) をサポートできる。
- 通信速度はもちろん通信パターンに依存する。
- 実行時にデータ依存でループ回転数を制御するメカニズムは必要

古典的大規模 SIMD マシンと同等のことが可能。HPF (ほぼ CM-Fortran) が実行できる。

コンパイラによるコード生成の例—OpenCL(風)

実行コードの例(開発中の OpenCL): ソース(1)

```
__kernel void gravity(global REAL gxi[][3], global REAL gxj[][3],
                      global REAL * gmj,    global REAL gf[][3],
                      global REAL * geps2,  int ni,  int nj)
{
    private double xi[NWORK][3];
    private double xj[NWORK][3];
    private double mj[NWORK];
    double eps2;
    private double f[NWORK][3];
    dist(xi, gxi, ni*3);
    dist(xj, gxj, nj*3);
    dist(mj, gmj, nj);
    dist(&eps2, geps2, 1);
```

ここまでは宣言と、DRAM からのデータ転送関数

実行コードの例(開発中の OpenCL):ソース(2)

実行コードと DRAM への逆転送

```
    for (int j=0;j<nj;j++){
        double r[3];
#pragma clang loop vectorize_predicate(enable)
        for (int i=0; i<ni; i++){
            for (int k=0;k<3;k++)  r[k] = xj[j][k]-xi[i][k];
            double r2 = r[0]*r[0]+ r[1]*r[1]+ r[2]*r[2] + eps2;
            double rinv = sqrt(1.0/r2);
            double mr3inv = mj[j]* rinv* rinv* rinv;
            for (int k=0;k<3;k++) f[i][k] += mr3inv*r[k];
        }
    }
    col(gf, f, ni*3);
}
```

LLVM-IR 出力(一部)

```
vector.body:                                      ; preds = %vector.body, %vector.ph
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %vector.body ], !dbg !30
  %vec.ind = phi <4 x i64> [ <i64 0, i64 1, i64 2, i64 3>, %vector.ph ], [ %vec.ind.next, %vector.body ]
  %2 = icmp ule <4 x i64> %vec.ind, %broadcast.splat
  %3 = getelementptr inbounds [16 x [3 x double]], ptr %xi, i64 0, <4 x i64> %vec.ind, i64 0
  %wide.masked.gather = call <4 x double> @llvm.masked.gather.v4f64.v4p0(<4 x ptr> %3, i32 8, <4 x i1> %2, <4 x double> poison), !dbg !31
  %4 = fsub <4 x double> %broadcast.splat97, %wide.masked.gather, !dbg !32
  %5 = getelementptr inbounds [16 x [3 x double]], ptr %xi, i64 0, <4 x i64> %vec.ind, i64 1, !dbg !32
  %wide.masked.gather98 = call <4 x double> @llvm.masked.gather.v4f64.v4p0(<4 x ptr> %5, i32 8, <4 x i1> %2, <4 x double> poison), !dbg !
  %6 = fsub <4 x double> %broadcast.splat100, %wide.masked.gather98, !dbg !32
  %7 = getelementptr inbounds [16 x [3 x double]], ptr %xi, i64 0, <4 x i64> %vec.ind, i64 2, !dbg !32
  %wide.masked.gather101 = call <4 x double> @llvm.masked.gather.v4f64.v4p0(<4 x ptr> %7, i32 8, <4 x i1> %2, <4 x double> poison), !dbg
  %8 = fsub <4 x double> %broadcast.splat103, %wide.masked.gather101, !dbg !32
  %9 = fmul <4 x double> %6, %6, !dbg !33
  %10 = call <4 x double> @llvm.fmuladd.v4f64(<4 x double> %4, <4 x double> %4, <4 x double> %9), !dbg !34
```

アセンブラ出力(中間コード)

ループ1回目

```
fsubd xj[0]s0v  xi[0]s3v  %4_s1v
fsubd xj[1]s0v  xi[1]s3v  %6_s1v
fsubd xj[2]s0v  xi[2]s3v  %8_s1v
fmuld %6_s1v  %6_s1v  %9_s1v
fmuladdd %4_s1v %4_s1v %9_s1v  %10_s1v
fmuladdd %8_s1v %8_s1v %10_s1v  %11_s1v
faddd eps2[0]s0v  %11_s1v  %12_s1v
fdivd CONST_double_1.000000e+00  %12_s1v  %13_s1v
call sqrt_d %13_s1v  %14_s1v
fmuld mj[0]s0v  %14_s1v  %15_s1v
fmuld %14_s1v  %15_s1v  %16_s1v
fmuld %14_s1v  %16_s1v  %17_s1v
fmuladdd %17_s1v %4_s1v f[0]s3v  f[0]s3v
fmuladdd %17_s1v %6_s1v f[1]s3v  f[1]s3v
fmuladdd %17_s1v %8_s1v f[2]s3v  f[2]s3v
```

アセンブラ出力(最適化した変数割り当て)

```
xj[]:M
xi[]:R
%4_s1v:N
%6_s1v:N
%8_s1v:M
%9_s1v:R
%10_s1v:N
%11_s1v:R
eps2[]:M
%12_s1v:R
%rsqrt.x_s1v:R
CONST_double_0.5:M
%rsqrt.ahalf_s1v:R
%rsqrt.xx_s1v:R
%rsqrt.axx_s1v:R
CONST_double_1.5:R
%rsqrt.dx_s1v:R
mj[]:M
%14_s1v:N
%15_s1v:M
%16_s1v:M
%17_s1v:M
f[]:R
```

R, M, N はそれぞれレジスタ、ローカルメモリ 0、ローカルメモリ 1
最適化の原理: 効率低下につながるデータ移動命令や nop が減るように各変数のわりつけを変える。
SA で最適化。

アセンブラ出力(最適化後)

```
0009: fsubd in=xj[0]s0v xi[0]s3v out=%4_s1v nop=0
0010: fsubd in=xj[1]s0v xi[1]s3v out=%6_s1v t nop=0
0011: fsubd in=xj[2]s0v xi[2]s3v out=%8_s1v nop=0
0012: fmuld in=t t out=%9_s1v nop=0
0013: fmuladdd in=%4_s1v %4_s1v fb out=%10_s1v nop=0
0014: fmuladdd in=%8_s1v %8_s1v fb out=%11_s1v t nop=0
0015: faddd in=eps2[0]s0v fb out=%12_s1v nop=0
0016: rsqrt in=fb out=%rsqrt.x_s1v nop=0
0017: fmuld in=%12_s1v CONST_double_0.5 out=%rsqrt.ahalf_s1v t nop=0
0018: fmuld in=%rsqrt.x_s1v %rsqrt.x_s1v out=%rsqrt.xx_s1v nop=0
0019: fmuld in=fb t out=%rsqrt.axx_s1v t nop=0
0020: fsubd in=CONST_double_1.5 fb out=%rsqrt.dx_s1v nop=0
0021: fmuld in=fb %rsqrt.x_s1v out=%rsqrt.x_s1v t nop=0
0022: fmuld in=fb fb out=%rsqrt.xx_s1v t nop=0
0023: fmuld in=fb %rsqrt.ahalf_s1v out=%rsqrt.axx_s1v nop=0
0024: fsubd in=CONST_double_1.5 fb out=%rsqrt.dx_s1v t nop=0
0025: fmuld in=fb %rsqrt.x_s1v out=%rsqrt.x_s1v t nop=0
```

```
0026: fmuld in=mj[0]s0v %14_s1v out=%15_s1v t nop=0
0027: fmuld in=%14_s1v fb out=%16_s1v nop=0
0028: fmuld in=%14_s1v fb out=%17_s1v t nop=0
0029: fmuladdd in=fb %4_s1v f[0]s3v out=f[0]s3v nop=0
0030: fmuladdd in=t %6_s1v f[1]s3v out=f[1]s3v nop=0
0031: fmuladdd in=t %8_s1v f[2]s3v out=f[2]s3v nop=0
```

コンパイラその他

- OpenCL(風) のコード生成は、C/C++, Fortran の両方から可能。カーネル部分は原理的限界までの最適化ができている。
- OpenACC コンパイラはだいぶ目処がたってきた。XcodeML 経由で OpenCL 方言にコンパイル。

メモリ割り付け最適化の現状

- SA で原理的な限界まで最適化可能。コンパイル時間は短い。
- 直前の命令の結果を (特殊レジスタ指定で) 必ず利用可能、メモリをオペランドにとれる、というアーキテクチャでのメモリ割り付けの最適化は可能である。
- 原理的には、このアプローチは大規模 SIMD でなくても適用可能である。また、演算器レイテンシの段数だけ機械的にアンロールすることを前提とするならベクトル命令をサポートしなくても適用可能である。
- この結果は、マルチポートレジスタではなく「シングルポートレジスタを複数」用意して、命令セットから明示的にアクセス可能にすることで、マルチポートレジスタより大幅に消費電力を下げつつ実行効率をあげることが可能であることを示唆する。

ソフトウェア環境のまとめ

- MN-Core のハードウェアとしての汎用性は思っていた以上に高いことが本調査研究で明らかになった。
  - 一般的な制御構造を実現できる
  - PE 間のランダムな通信を実装できる
- 最適化コンパイラの開発は、固定長ベクトル命令、命令から見える大きなレジスタファイル等のためあまり難しくない。実際に、実用コードからほぼ完璧に最適化された命令を生成できている。
- 非常に高速かつ大容量のオンチップメモリがアプリケーションから利用可能なため、規則格子法等アプリケーションの性能は GPU よりずっと高くなる。

コンパイラ出力に関するコメント

- LLVM-IR の時点ではロード・ストア命令があるが、命令生成時に消去できる。
- LLVM-IR の時点ではアドレス計算が非常に複雑だが、実行コードになるところまでで解決する。
- 固定長ベクトル命令でレイテンシを隠蔽できるため、in-order 実行で完全にパイプラインを埋めることができる。このため、コンパイラによる高度な最適化は不要となる。
- 繰り返し回数の非常に少ないループでも、デバイスコード実行前に完全にアンロールされるため、分岐のオーバーヘッドがなく、小さな多次元配列でも性能が落ちない。

通常のアーキテクチャとの違い

- 現在の通常の CPU では、レジスタリネーミングと OoO 実行によって、最内側ループに本来ある並列性を実行時に回復する必要がある。これは、通常はアーキテクチャレジスタの数が不足し、コンパイラによるアンロールだけではパイプラインを埋められないからである (「京」は例外)
- MN-Core ではアーキテクチャレジスタ数の制限がないため、レジスタリネーミングが不要となる。
- さらに、固定長ベクトル命令の採用により、OoO 実行自体が不要になっている。直前の命令の実行結果が必ず利用可能だからである。(SVE や RISC-V vector extension でも実装可能)
- かなり単位の大きな SIMD 実行をすることで、非常に長い命令語を許しており、レジスタのアドレス長等の制限が不要になっている。

アプリケーション性能の例(1)

PFN 社内で OpenFDTD の実装 (アセンブラ) をおこなった例。
OpenFDTD: 広く使われている FDTD (Finite difference Time Domain method) 法による電磁界シミュレータ。電子機器、無線アンテナ等の特性評価・設計に広く使われている

- Cuda 版が存在、CPU 版に比べて圧倒的に高速。
- 6 成分の微分方程式を格子 (有限差分) で離散化
- 要求 B/F は 2.28 (格子点当り 96 バイト、42 演算)。
- 演算数が極端に少なく、袖交換が重い
- temporal blocking を導入 (今回 $N_{TB} = 6$)

	速度 (GF)	速度 (Gcells/s)	備考
MN-Core 2	655	16.6	TB
Nvidia A100	488.7	11.6	開発元データ
Nvidia P100	134.7	3.2	開発元データ

A100 に比べて倍精度ピーク (非行列モード)、メモリバンド幅でかなり低いが、達成できた性能は高い

アプリケーション性能の例(2)

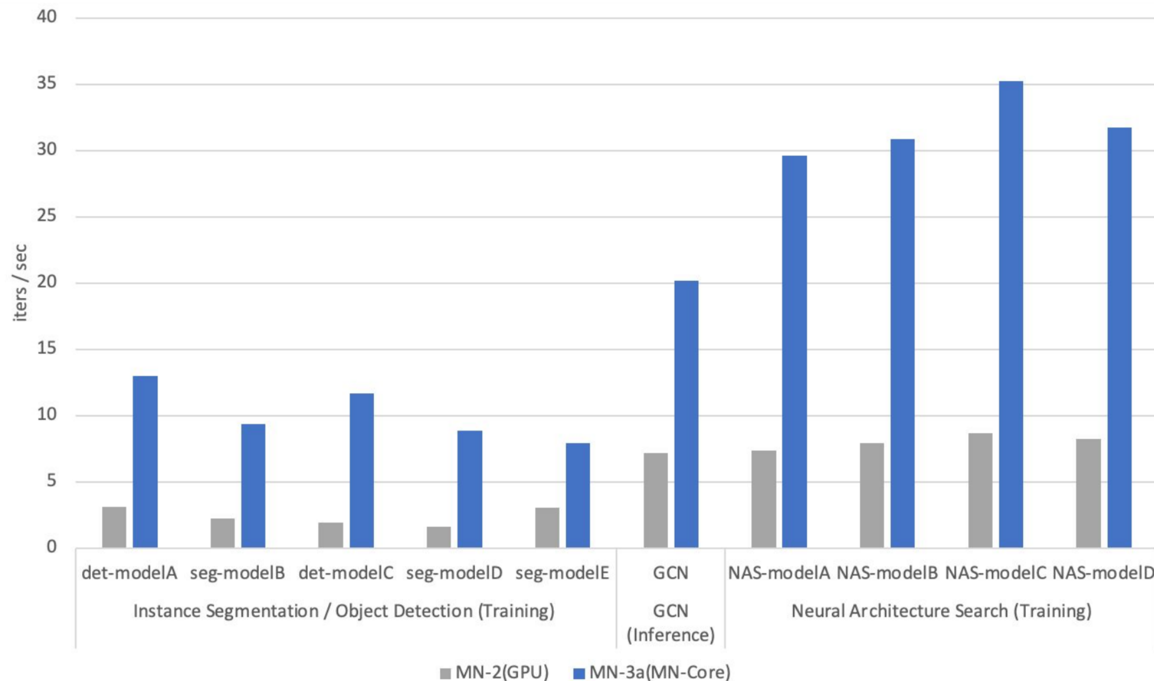
PFN 社内で 姫野ベンチの実装 (アセンブラ) を行った例。
メモリ性能の実アプリケーション (規則格子) に近い状況での評価

- 19 点ステンシル。拡散方程式の差分近似
- 要求 B/F は 3 程度
- 1 点の演算数が多く、袖交換は少ない。
- MN-Core ではオンチップメモリにはいる
- A100/H100 では L2 にのらない。

	速度	備考
MN-Core	6TF	
Nvidia H100	1.2TF	PFN 社内測定
Nvidia A100	600GF	プロメテック社資料

H100 に比べても 5 倍の性能を実現した。

アプリケーション性能の例(3)–機械学習



V100 との比較。絶対性能では高い。ピーク性能に対する比は同等。
(深層学習では GPU の効率も高い)