研究報告Ⅱ　システム研究調査の概要と検討状況

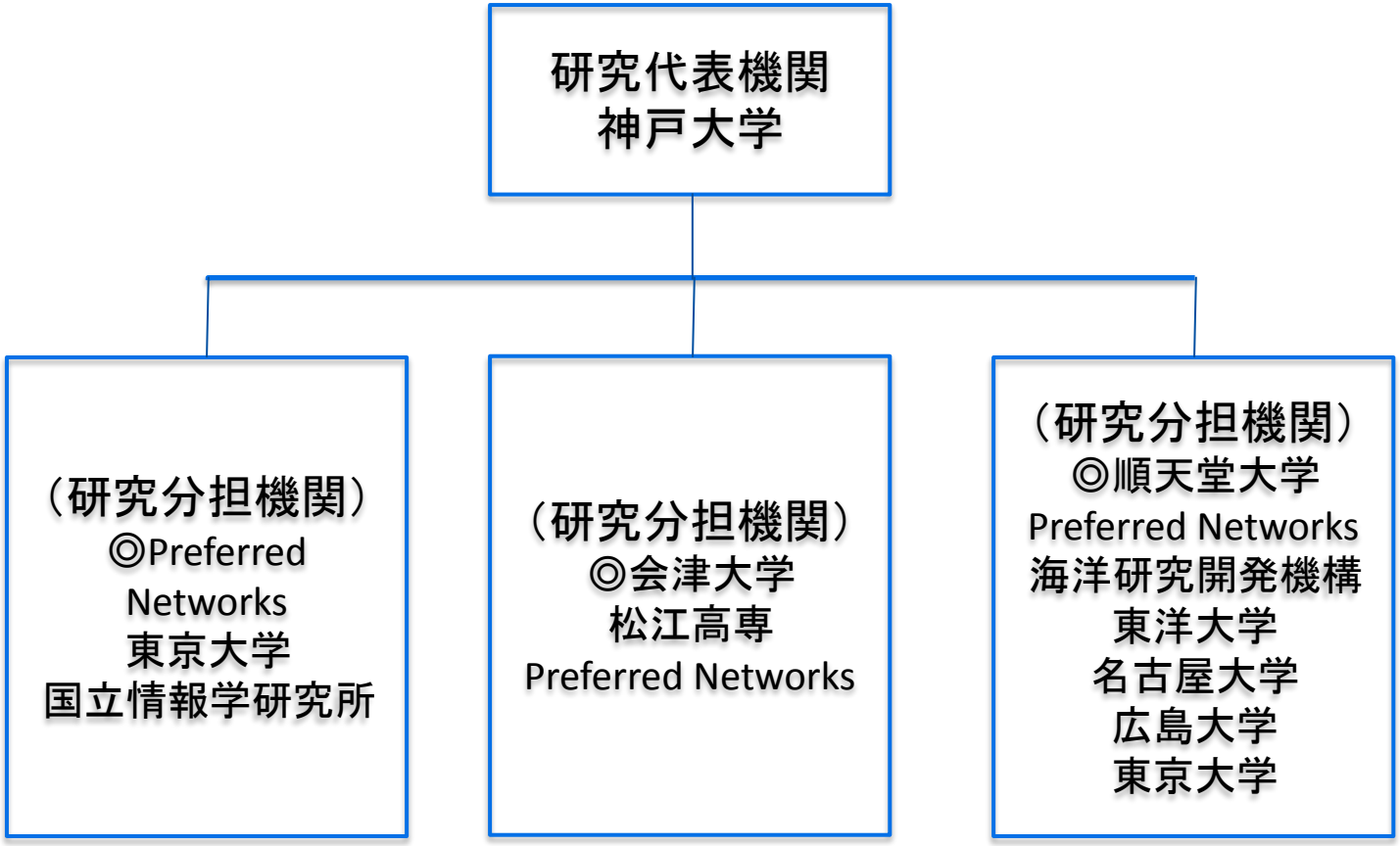
牧野 淳一郎
神戸大学 大学院 理学研究科

次世代計算基盤に係る調査研究に関する合同ワークショップ～フィージビリティスタディ中間報告～
(2023年12月19日)

本発表の構成

- 調査研究の実施体制
- 調査研究の概要
  - 提案調査研究の概要
- GPUアーキテクチャの限界とその次
- MN-Coreアーキテクチャとそのソフトウェア
- まとめ

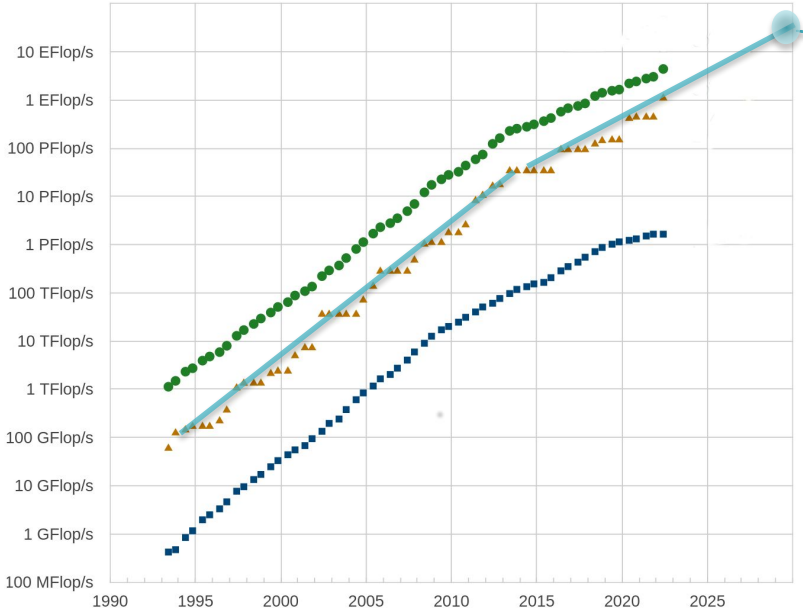
調査研究の実施体制


研究代表機関
神戸大学
（研究分担機関）
◎Preferred Networks
東京大学
国立情報学研究所
（研究分担機関）
◎会津大学
松江高専
Preferred Networks
（研究分担機関）
◎順天堂大学
Preferred Networks
海洋研究開発機構
東洋大学
名古屋大学
広島大学
東京大学

提案調査研究の概要

- HPCシステムのトレンド
- 提案調査研究の概要

HPCシステムのトレンド



2028-30年のHPL Top #1: 20-50EF

技術的にはかなり困難：半導体技術の進歩だけによる低消費電力化は4倍程度。設計の革新による低消費電力化が必須

半導体技術の進歩だけでは
10EF以下にとどまる可能性もある

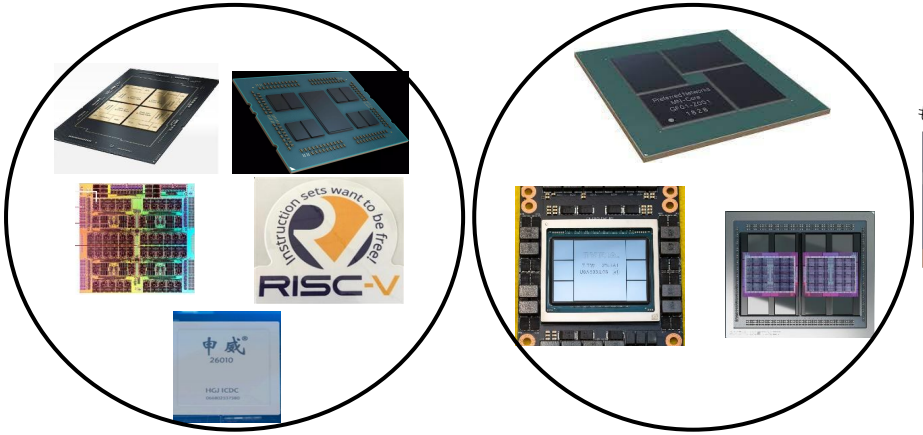
演算性能よりも大きな問題：実効性能、使いやすさ　多くのHPCアプリケーションの実行効率:5-10%
ーピーク演算性能だけ高くてもダメ

アプリケーションの新しい方向：AIの活用、低精度演算の利用
アプリケーションが何を実現できるかのほうが重要

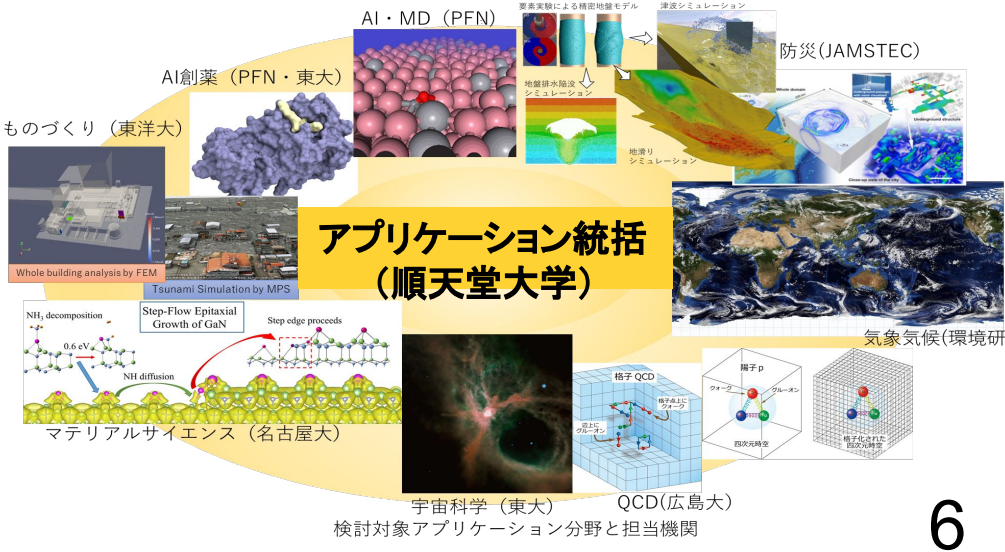
本調査研究の目標

- 国内で開発されるアクセラレータを含む複数のハードウェアアーキテクチャの組合せに対して
  - 演算性能、電力性能比、I/O性能、コスト、生産性、商用展開・技術展開等の指標を
  - 典型的かつ重要と考えられるアプリケーションタイプを対象に

  評価することで、総合的にハードウェアアーキテクチャを評価



CPU・アクセラレータの最適な組み合わせを評価



実際のところ、どんなオプションがあるか？

メインとなるプロセッサは？

- CPU?
- GPU?
- どっちでもないなにか？

もうちょっと基本的な問題：CPUとGPU(とそれ以外)は

- なにが
- なぜ
- なんのために

違うのか

CPUとGPUは何が違うのか？

意外に簡単ではない。
どちらもメニーコア、コア内SIMD演算ユニット、階層キャッシュ。

細かく見ると、
CPU：スーパースカラー、OoO、コヒーレントキャッシュ、少ないハードウェアスレッド、
少数のアーキテクチャレジスタ
GPU：非コヒーレントなキャッシュ、多数のスレッド、膨大なレジスタ

キャッシュコヒーレンシ：消費電力に大きく影響

レジスタ数とスレッド数：（特に主記憶の)レイテンシ隠蔽に効果的

GPUはCPUに比べて

キャッシュコヒーレンシを断念した
多数のハードウェアスレッドを導入して、OoO実行をほぼ不要にした
　　大きく電力性能が改善され、シリコン面積あたり性能も向上
　　演算リミットなアプリケーションでCPUより高い性能

多数のハードウェアスレッドを導入した
　　⇒記憶レイテンシ隠蔽を容易に実現し、実行効率を(ある程度)向上

GDDR、HBM等の高バンド幅DRAMを採用してメモリバンド幅をあげた
　　⇒記憶バンド幅リミットのアプリケーションでCPUより高い性能

じゃあもうGPUでいいのか？

コンピュータアーキテクチャの歴史からはそろそろ次がでてくるはず
1975まで：スカラー計算機（CDC6600とか)
1976-1992: 共有メモリベクトル並列(Cray-1、日本のベクトルスーパーとか)
1993-2007: 分散メモリマイクロプロセッサ(Cray T3Dとか)
2008-202x?: GPGPU

大体15年おきに変わることがわかる。

なぜ変わるか？：それまでのアーキテクチャが半導体技術の進歩(変化)に対応できなくなるから。

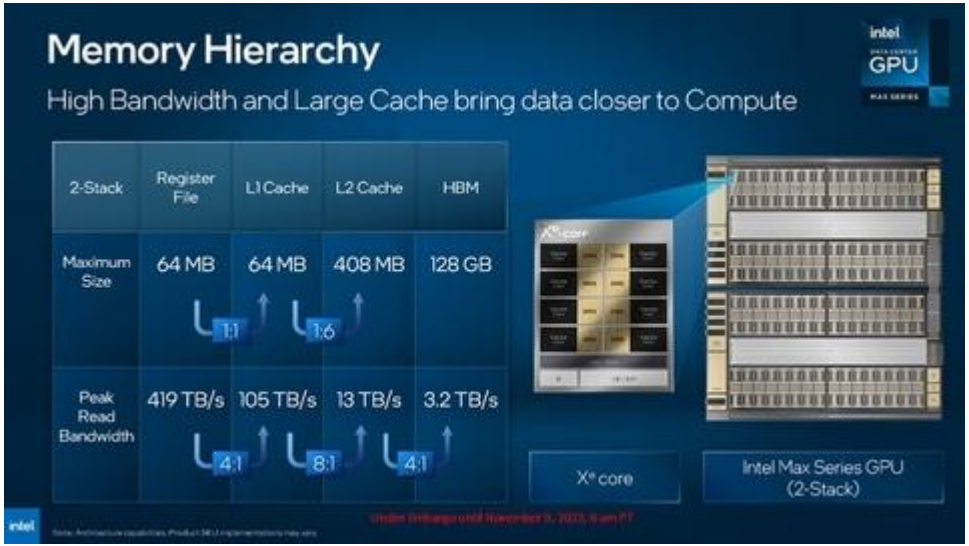
半導体技術とアーキテクチャ

スカラーからベクトル：半導体メモリと、マルチコアが可能な集積度
ベクトルからマイクロプロセッサ並列：1チップ1コアになるとmemory wall出現
CPUからGPU：メニーコアになるとコヒーレントキャッシュとOoOが制約に

GPUからその次のなにか：　制約はなにか？

明らかに現在制約になっているもの：共有メモリバンド幅

Ponte Vecchio の「教訓」



演算性能に対して主記憶、L2$バンド幅は決して高くない。L2$でB/F=0.25、主記憶で0.06

それでも、電力性能は H100, MI250X より大きく劣る。これらの L2$ バンド幅は 0.1 程度。

演算性能の向上に共有メモリ（キャッシュも）バンド幅がおいつかなくなっている
共有メモリベクトルプロセッサと同じ。システムレベルで起こったことがチップレベルで繰り返す

ではGPUの次は？

「共有メモリベクトルプロセッサと同じ」問題なら同じ解決があるはず
＝「分散メモリ」への移行

チップ内分散メモリでなぜ電力が減るか？
チップ内でもデータの横方向の移動の電力が無視できなくなるから。

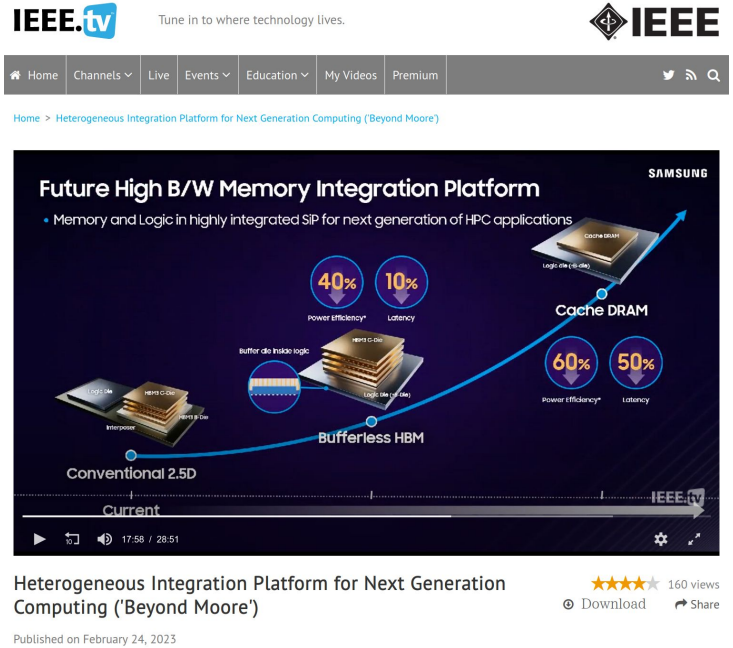
電圧1V なら、基板上でもチップ上でも 1 bit を 1cm 動かすと大体 1pJ消費する

8ビットデータを1cm 移動させると8pJ、B/F=4 で演算毎に1cm動くと30GF/W が限界になる。

GPUの共有L2$バンド幅が上がらない（上げると電力的に破綻する）のはこれが理由

チップ内分散メモリとは？

- 実際に最近のGPUでない高性能プロセッサはコア毎にキャッシュじゃないローカルメモリを持つものがほとんど。PEZY-SC, Sunway, MN-Core 等
- とはいえ、これはDRAMに比べるとずっと容量が小さいオンチップSRAMなので、どんな問題でも高性能というわけではなかった



論理的には、「容量の大きなオンチップDRAM」が必要
最近までそんなものはなかった
ようやく姿がみえてきた=3D実装DRAM（図は Samsung)

3D実装DRAM

- HBM: DRAM自体は3次元実装。12枚とかを重ねて、一番下に「logic die」
- この logic die はインターフェースだけ。CPUとかGPUはその横に
- このため、パッケージ内にDRAMがはいってGDDR等より配線短いが1-2cm ある
- 本当にDRAMをプロセッサの上に載せて、DRAM全面から信号出せば配線を1mm程度にできる

電力性能を現在のGPUを超えてあげつつ、メモリバンド幅を向上させるには必須の技術

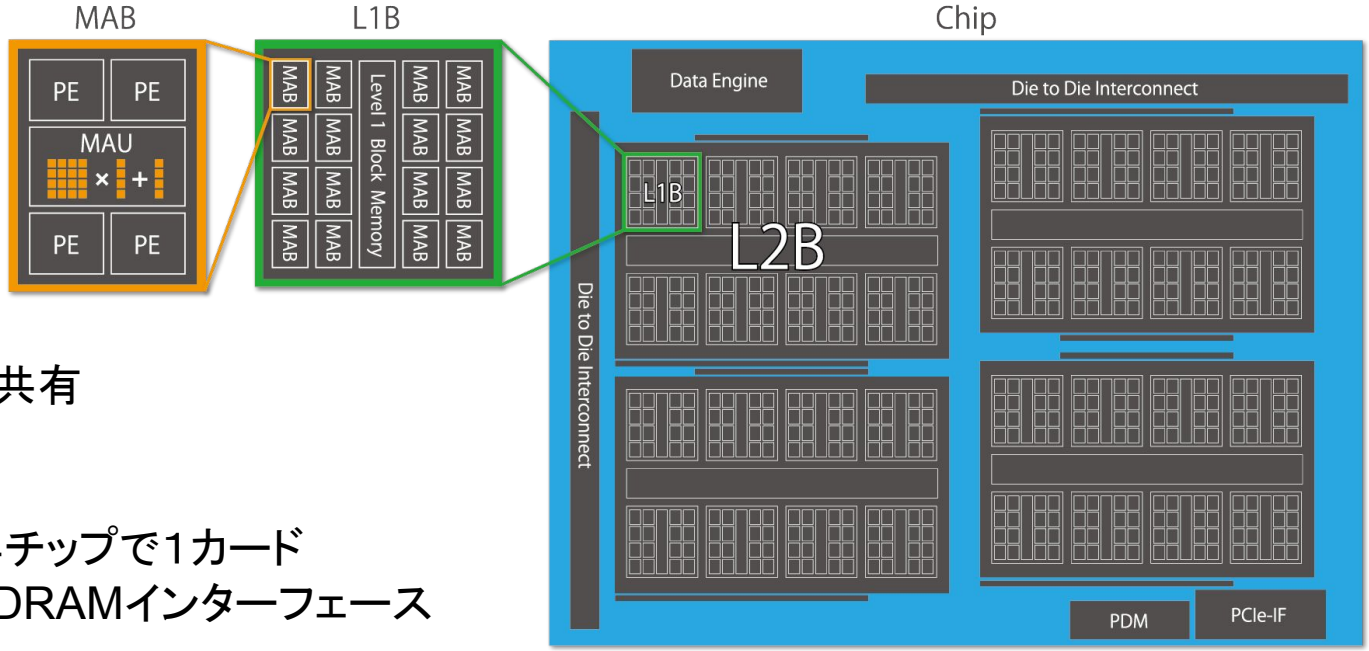
神戸大学・PFN チームでは当初から検討。最近CPU・GPU各社が検討しているという情報が流れてきている

MN-Core アーキテクチャ

- 大規模SIMD
- PE毎に大きなローカルメモリ
- キャッシュではなくて明示的にデータ移動を制御できるチップ内ネットワーク

- これまでのMN-Core:　DRAMは普通に実装、B/F 低い
  ローカルメモリとの間はDMA的なコマンドでデータ移動
- ポスト富岳での検討:　3D実装DRAM、B/F「高い」（といっても0.1くらい、、、）
  （遅いが）ローカルメモリの拡張として見える

MN-Core アーキテクチャ

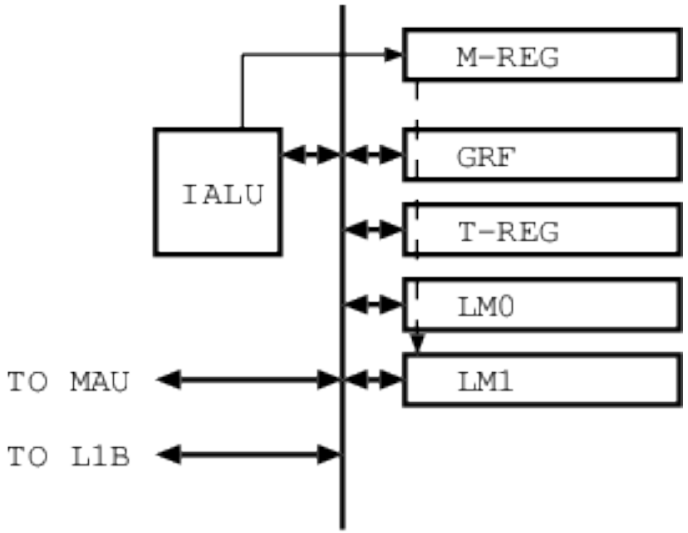


PE4個が行列乗算器を共有
PE64個が「L1B」
「L1B」8個が「L1B」
「L2B」4個が1チップ。4チップで1カード
PCIeインターフェース、DRAMインターフェース

MN-Core PE アーキテクチャ

汎用レジスタファイル(GRF)
ローカルメモリ(LM0, LM1)
の他に
補助レジスタ(T)
マスクレジスタ

GRF、LM0/1、Tレジスタはすべてオペランドにとれる。また、演算器の出力自体をオペランドにできる



MN-Core ソフトウェア

- OpenCL、OpenACCに近いものが大体普通に動く（まだ開発中ではある）
- 80-90年代のSIMD超並列マシン(Connection Machine や MasPar)のデータ並列言語とほぼ同様（OpenACCというより HPF 的）

- これまでのMN-Core:　DRAMは普通に実装、B/F 低い
  ローカルメモリとの間はDMA的なコマンドでデータ移動
- ポスト富岳での検討:　3D実装DRAM、B/F「高い」（といっても0.1くらい、、、）
  （遅いが）ローカルメモリの拡張として見える

OpenCL（に近いもの）の例

```
subroutine twoddiff(a,b,h, n)
  use type_mncl
  use type_mncl_funcs
  type(integer), value  :: n
  type(gdouble) ::   a(n,n), b(n,n), h
  real * 8 alocal(16,16), blocal(16,16),  hlocal
  type(integer) :: i
  call distgl(alocal, a, n*n)
  call distgl(blocal, b, n*n)
  call bcast2(hlocal, h)
  do j=2,n-1
        do i=2,n-1
        blocal(i,j) = (alocal(i-1,j)+alocal(i+1,j) + alocal(i,j-1) + alocal(i,j+1)
          - 4*alocal(i,j))*hlocal
        enddo
  enddo
  call colgl(b, blocal, n)
end subroutine twoddiff
```

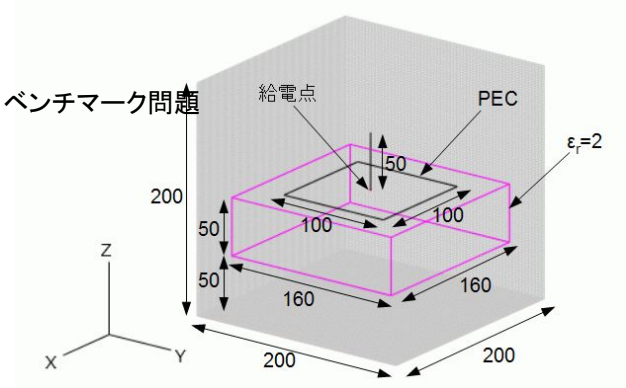
OpenCL（に近いもの）の例

- もちろんC（C++コンパイラ利用）もある
- OpenCL の仕様には Fortran はないが、色々不便なので作る方向で
- といっても LLVM フロントエンド使うだけ
- これは2次元拡散方程式の例。実用コードには袖交換が必要で、これはライブラリ関数を用意する
- この部分以外のコンパイラはほぼ完成。非常に良い最適化ができる。

アプリケーション性能の例

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
- *FDTD法の差分計算はベクトル化に適したアルゴリズムで「メモリアクセスがボトルネックになります」(NEC のページ https://jpn.nec.com/hpc/sxauroratsubasa/Application/ より)
- 6成分の微分方程式を格子(有限差分)で離散化
- ベンチマーク問題



・ベンチマーク結果

システム	性能 (GFlops)	性能 (Gcells/s)	備考
MN-Core 2 820MHz	**655**	**16.6**	**TB, overlap あり**
NVIDIA A100	488.7	11.6	開発元測定
NVIDIA P100	134.7	3.2	開発元測定

PFN 社内でアセンブラにより実装
temporal blocking を導入 (今回 N_TB=6)

メモリバンド幅・FP64ピーク性能の両方で上回る NVIDIA A100 よりも高い性能を実現＝大幅に高い実行効率を実現

MN-Core アーキテクチャのアプリケーション効率の優位性を実証

アプリケーション性能の例

- ヤコビ反復によるポアソン方程式の求解を行う
- 有限差分法のベンチマークとして広く用いられる
- メモリアクセスがボトルネックになる場合が多い、行列ベクトル積ユニットが使えないという意味でMN-Coreは比較的不得意なアプリケーション

```
do loop=1,nn
      gosa= 0.0
      do k=2,kmax-1
         do j=2,jmax-1
            do i=2,imax-1
               s0=a(I,J,K,1)*p(I+1,J,K) &
                     +a(I,J,K,2)*p(I,J+1,K) &
                     +a(I,J,K,3)*p(I,J,K+1) &
                     +b(I,J,K,1)*(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(I,J,K,2)*(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(I,J,K,3)*(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(I,J,K,1)*p(I-1,J,K) &
                     +c(I,J,K,2)*p(I,J-1,K) &
                     +c(I,J,K,3)*p(I,J,K-1)+wrk1(I,J,K)
               ss=(s0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
               GOSA=GOSA+SS*SS
               wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
            enddo
         enddo
      enddo
```

- 128x128x256サイズの問題をMN-Core上で実装
- データレイアウトと袖交換命令列の最適化により約3TFlopsを実現見込み(**効率としては理論限界に対して40%**)
- PFN社内でアセンブラにより実装

まとめ

- 本調査研究では、現在限界が見えてきているGPUの「次」のアーキテクチャを提案する。
- それにより、CPUやGPUの延長では不可能な高い価格性能比、電力性能を実現する
- ソフトウェア環境は整備中だが、ユーザーがアセンブラでチューニングしなくても高い実行効率が得られる目処がたってきた
- 要求B/Fが高い、陽解法差分法のコードでGPUを上回る性能がでることが実証できた
- CPUと違ってアーキテクチャレジスタ数の制約がないこと、ローカルメモリのバンド幅が高いことが実行効率の向上に貢献している

コンパイルの例(1) ソースコード

```
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];
    }
}
```

コンパイルの例（2）変数割当

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
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

コンパイルの例(3)実行コード

```
fsubd in=xj[0]s0v xi[0]s3v out=%4_s1v
fsubd in=xj[1]s0v xi[1]s3v out=%6_s1v
fsubd in=xj[2]s0v xi[2]s3v out=%8_s1v
fmuld in=t t out=%9_s1v
fmuladdd in=%4_s1v %4_s1v fb out=%10_s1v
fmuladdd in=%8_s1v %8_s1v fb out=%11_s1v t
faddd in=eps2[0]s0v fb out=%12_s1v
rsqrt in=fb out=%rsqrt.x_s1v
fmuld in=%12_s1v CONST_double_0.5 out=%rsqrt.ahalf_s1v t
fmuld in=%rsqrt.x_s1v %rsqrt.x_s1v out=%rsqrt.xx_s1v
fmuld in=fb t out=%rsqrt.axx_s1v t
```

```
fsubd in=CONST_double_1.5 fb out=%rsqrt.dx_s1v
fmuld in=fb %rsqrt.x_s1v out=%rsqrt.x_s1v t
fmuld in=fb fb out=%rsqrt.xx_s1v t
fmuld in=fb %rsqrt.ahalf_s1v out=%rsqrt.axx_s1v
fsubd in=CONST_double_1.5 fb out=%rsqrt.dx_s1v t
fmuld in=fb %rsqrt.x_s1v out=%rsqrt.x_s1v t
fmuld in=mj[0]s0v %14_s1v out=%15_s1v t
fmuld in=%14_s1v fb out=%16_s1v
fmuld in=%14_s1v fb out=%17_s1v t
fmuladdd in=fb %4_s1v f[0]s3v out=f[0]s3v
fmuladdd in=t %6_s1v f[1]s3v out=f[1]s3v
fmuladdd in=t %8_s1v f[2]s3v out=f[2]s3v
```

- メモリ割当の最適化(SAで自動化)により、パイプラインバブルがほぼない(この場合では全くない)コードを生成できる
- 無駄なレジスタアクセスがなく電力消費が小さい