GRAPE-DR 詳解

国立天文台
理論研究部/天文シミュレーションプロジェクト (CfCA)
牧野淳一郎

今日の話の構成

1. GRAPE-DR 概要
2. ハードウェア詳細
3. アセンブラ
4. (コンパイラ)

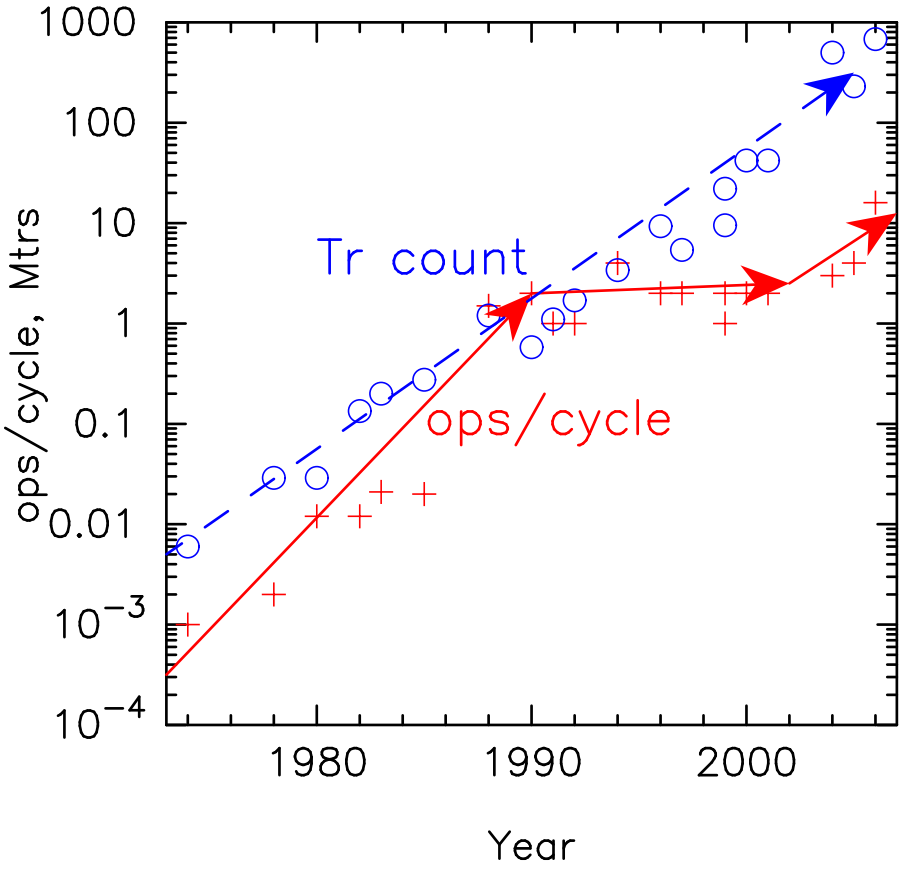
GRAPE-DR の基本的な考え

- 応用に特化し、多数の演算器を 1 チップに集積、並列動作させて高い性能を得た専用計算機の特徴を生かす
- しかし広い応用範囲を実現する

そんなことができるか？が問題

トランジスタは沢山ある

マイクロプロセッサの「進歩」



- **トランジスタ: 15 年で 1000 倍**
- **演算器の数: 同じ期間に 8 倍**
- **100 倍分がどこかに消えた**
- **最も良かった時でもチップ上の演算器の割合は 5% くらい**

多数の演算器を詰め込む方法

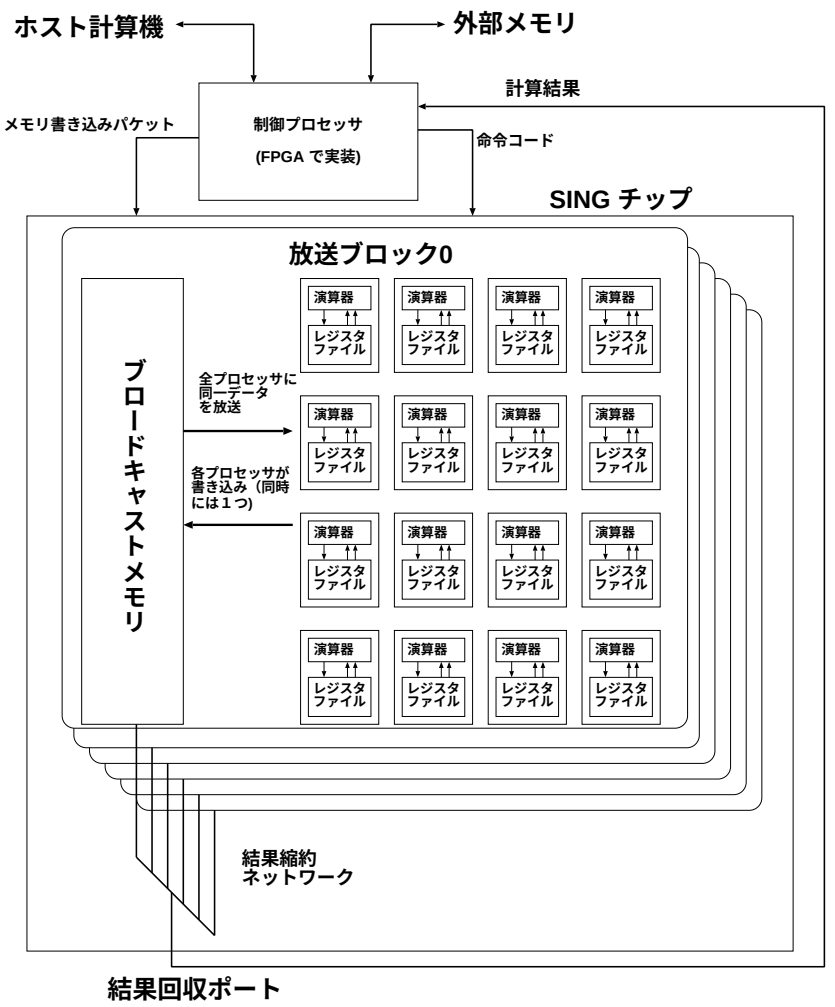
境界条件: メモリバンド幅は増やしたくない (システムコストはほぼメモリバンド幅で決まる)
可能な方策

1. GRAPE 的専用パイプラインプロセッサ
2. 再構成可能プロセッサ
3. SIMD 並列プロセッサ

SIMD 並列処理

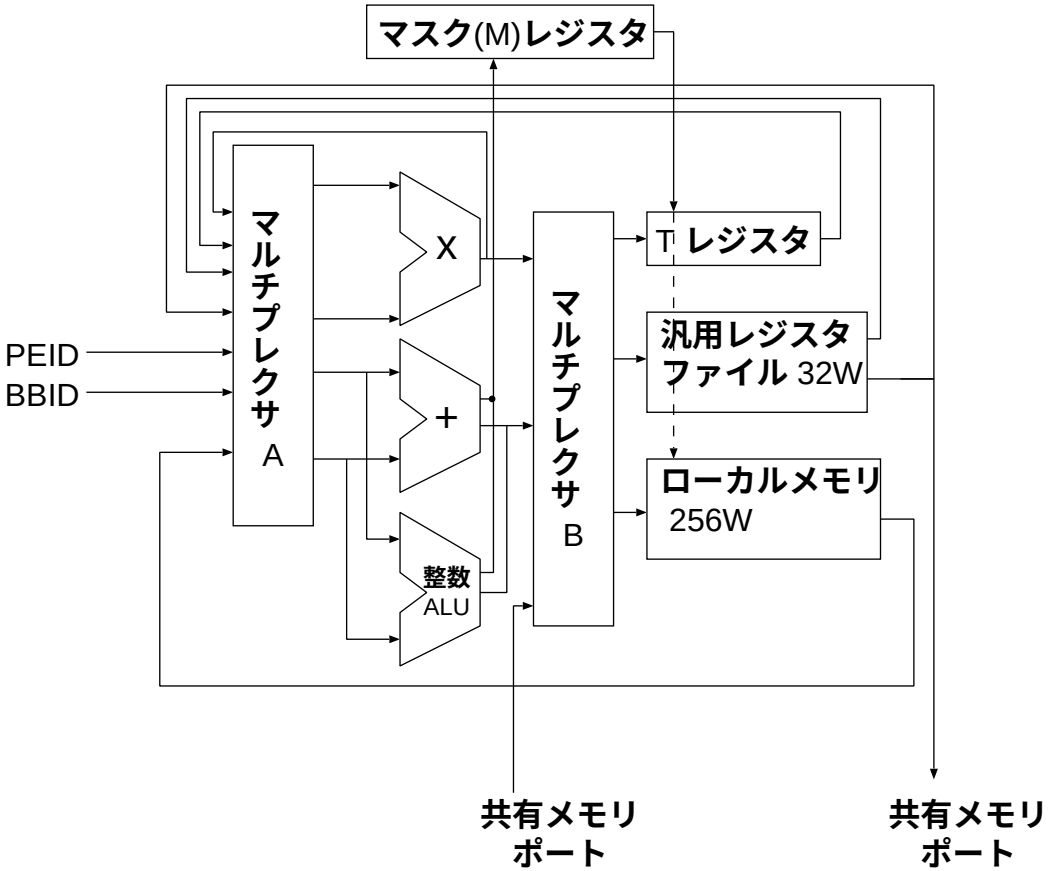
- パイプラインプロセッサをやめにして、「プログラム可能なプロセッサ」を沢山載せる。
- SIMD (Single Instruction Multiple Data): 全プロセッサが同じ命令を実行
- 基本的には、全プロセッサがソフトウェアで GRAPE をエミュレーションする。

GRAPE-DR における SIMD



- 非常に多数のプロセッサエレメント (PE) を 1 チップに集積
- PE = 演算器 + レジスタファイル (メモリをもたない)
- チップ内に小規模な共有メモリ (PE にデータをブロードキャスト)。これを共有する PE をブロードキャストブロック (BB) と呼ぶ。
- 制御プロセッサ、外部メモリへのインターフェースを持つ

PE の構造

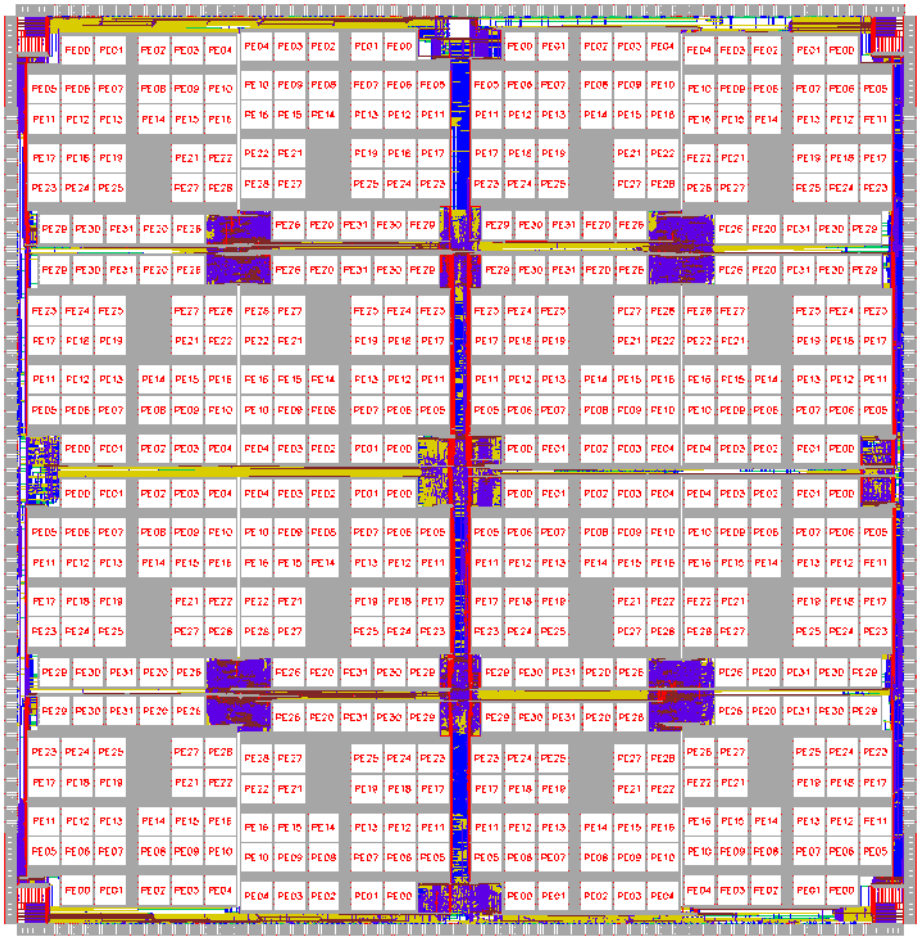


- 浮動小数点演算器
- 整数演算器
- レジスタ
- メモリ 256 語
  (K とか M ではない。)

プロセッサチップとプロトタイプボード

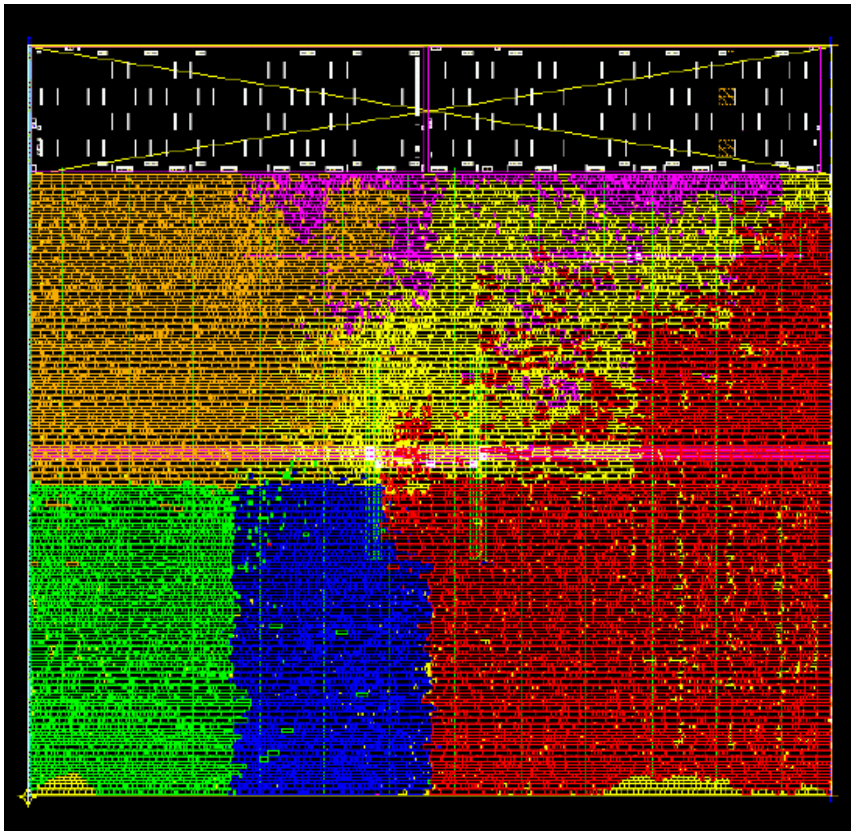
チップ写真

レイアウト



- 32PE(要素プロセッサ)のブロックが 16 個
- 空いているところは共有メモリ
- チップサイズは 18mm 角

PE レイアウト



0.7mm by 0.7mm

Black: Local Memory

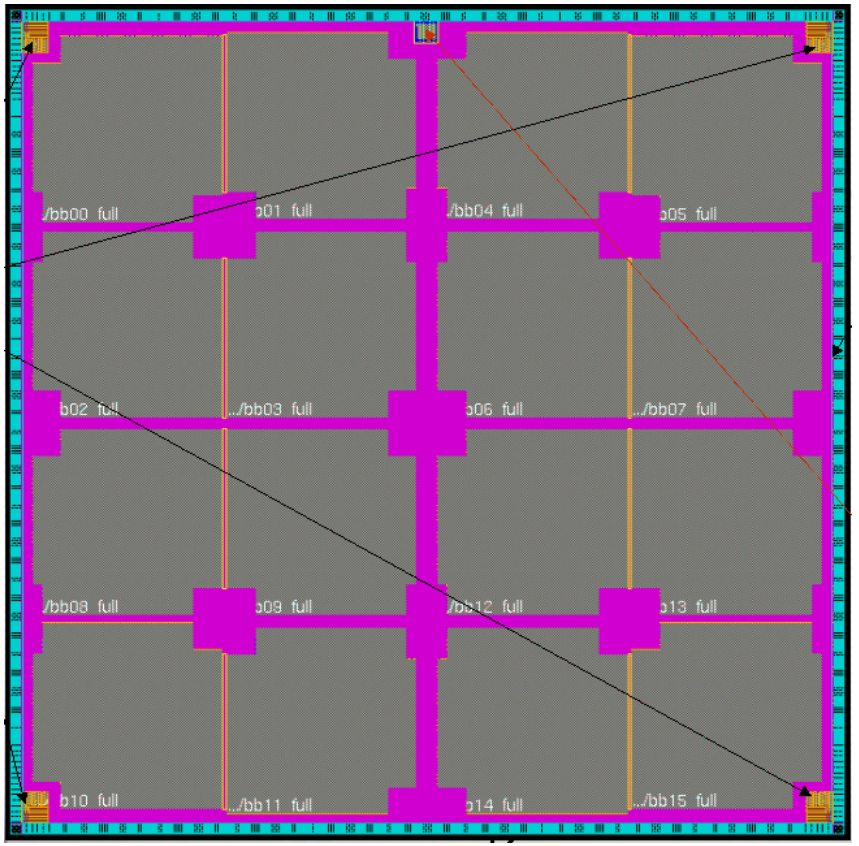
Red: Reg. File

Orange: FMUL

Green: FADD

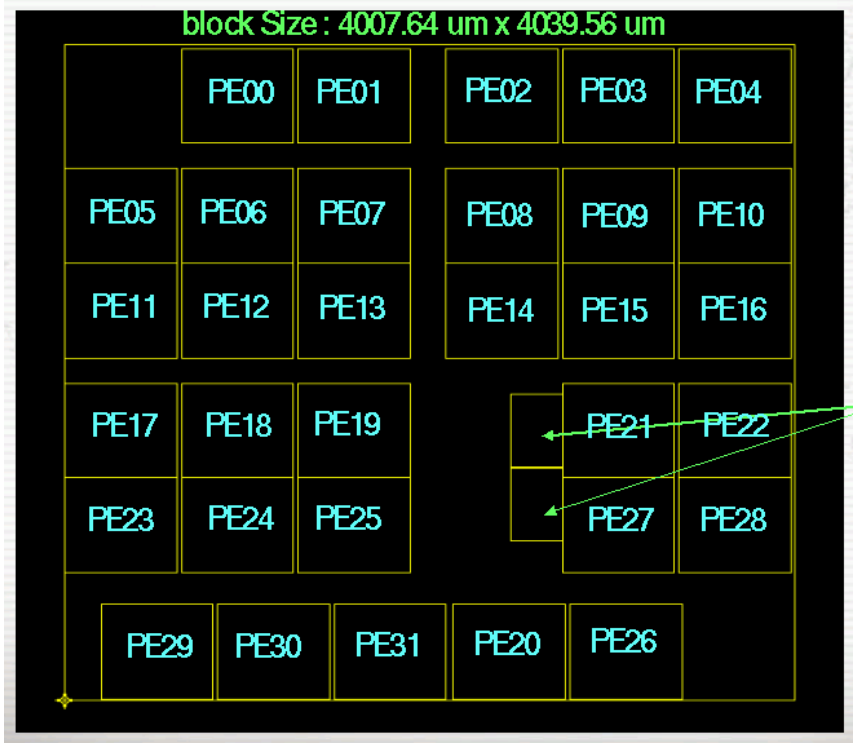
Blue: IALU

チップ全体フロアプラン



- 特におかしいとこなし
- サイズは大きい。
  17mm 角

放送ブロックフロアプラン



- 特におかしいとこなし

アプリケーションに対する考え方

- Memory Wall が問題にならないような
  アプリケーションのみを対象にする
- 3 つの型に特化
  - 散乱実験型
  - 粒子間相互作用型
  - 密行列型
- 可能ならばアプリケーションを書き換える

散乱実験型

- 多数の PE が、独立にイベントを計算
  - イベント間の相互作用はない、または非常に少ない
    - 光線追跡：レンズ自動設計
    - アナログ回路シミュレーション：回路自動設計
    - 放射線伝播のモンテカルロ計算：検出器設計
- “Embarassingly Parallel” とほぼ対応
- 古典的 SIMD 機と同様の振る舞い:
  - Goodyear MPP, ICL DAP, TMC CM-1/2, Maspar MP-1/2
  - 極端に少ないメモリ
  - PE 間通信が遅い
- 計算速度と通信速度の比:
  - 散乱実験の計算がどれだけ複雑かで決まる

粒子間相互作用型

$$f_i = \sum_j f(x_i, x_j)$$

- 他の「粒子」との「相互作用」を縮約。
  - 全ての相互作用を並列に計算可能
  - 同じ「粒子」のための計算結果を高速に縮約する必要
- 計算手順
  - PE に相互作用を受ける粒子をロード
  - 相互作用を及ぼす粒子をロード
  - 計算機終了したら結果を縮約しながら回収
  - 計算速度とチップ外への通信速度の比:
    相互作用を及ぼす粒子数に比例

密行列型

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

- 計算手順
  - 行列が PE に収まるところまで分割。それから
  - 行列 A の部分行列を PE にロード
  - B の 1 列を分解して各グループにロード
  - 各 PE で B の部分列と A の部分行列の積を計算
  - 計算が終わったものから順次回収。グループ間で合計
- 計算速度・通信速度の比はチップ全体にロードできる行列のサイズに依存
  - メモリサイズの平方根に比例して通信速度を落とせる

計算・通信比のまとめ

- 散乱実験型: アプリケーション依存
- 粒子間相互作用型: 粒子数依存
- 密行列型: オンチップメモリサイズ依存
- 設計におけるトレードオフ:
  - なるべくアプリケーション範囲を広く
    - メモリを多く、バンド幅を広く $\rightarrow$ コスト増
  - コストを圧迫しないようにバランスを考える必要あり
- 実際の設計では密行列型がもっとも厳しい

プログラミング環境

- 加速ボードのメーカーが必ずいうこと:
  - アプリケーションプログラムから加速ボード側で実行できる部分を自動検出、変換できます
  - ソースコードに手を加えずに性能向上可能です
- 30 年前から同じことをいわれてきた
  - 今更騙される人はいない
- より現実的、効果的なアプローチ
  - 加速ボードでは単純なカーネルルーチンだけを実行
  - 他の部分はホスト上のプログラム。
    これは「基本的には」改変なし

GRAPE-DRで提供する環境

- カーネルルーチン
  - 行列乗算、対角化等の行列演算ルーチン
    - BLAS, LAPACK のサブルーチンの形に
  - 粒子間相互作用計算ルーチン
    - 単純な粒子間相互作用程度はアセンブラでも書ける
  - 二電子積分、交換積分などグランドチャレンジに必要なルーチン
- アセンブラ
  - 昔はみんなこれで書いていた
- PE 上でのコードに対して、「高級言語」を提供
  - PGDL (理研 濱田、中里、FPGA 再構成計算用コンパイラ)
  - SPH 法のための専用パイプライン (演算器150) の合成に初めて成功

実際のチップ内ネットワーク

PE は放送メモリとだけ接続

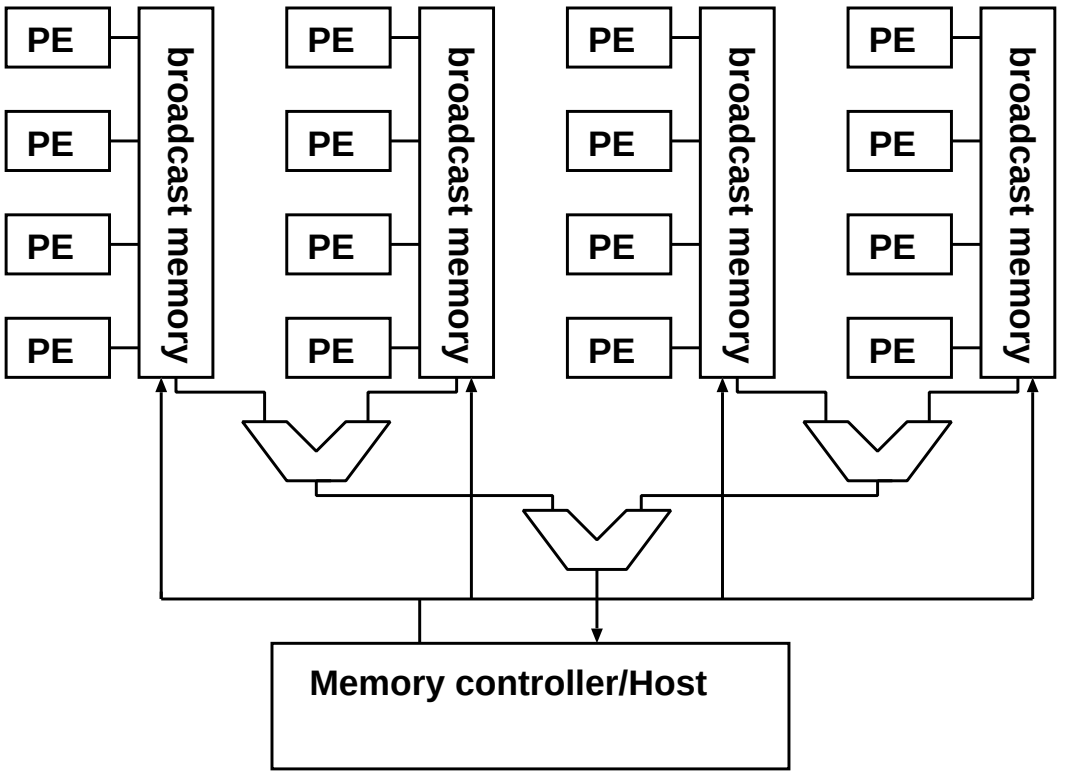
放送メモリからそれにつながった PE には

- 放送
- ランダム読み書き

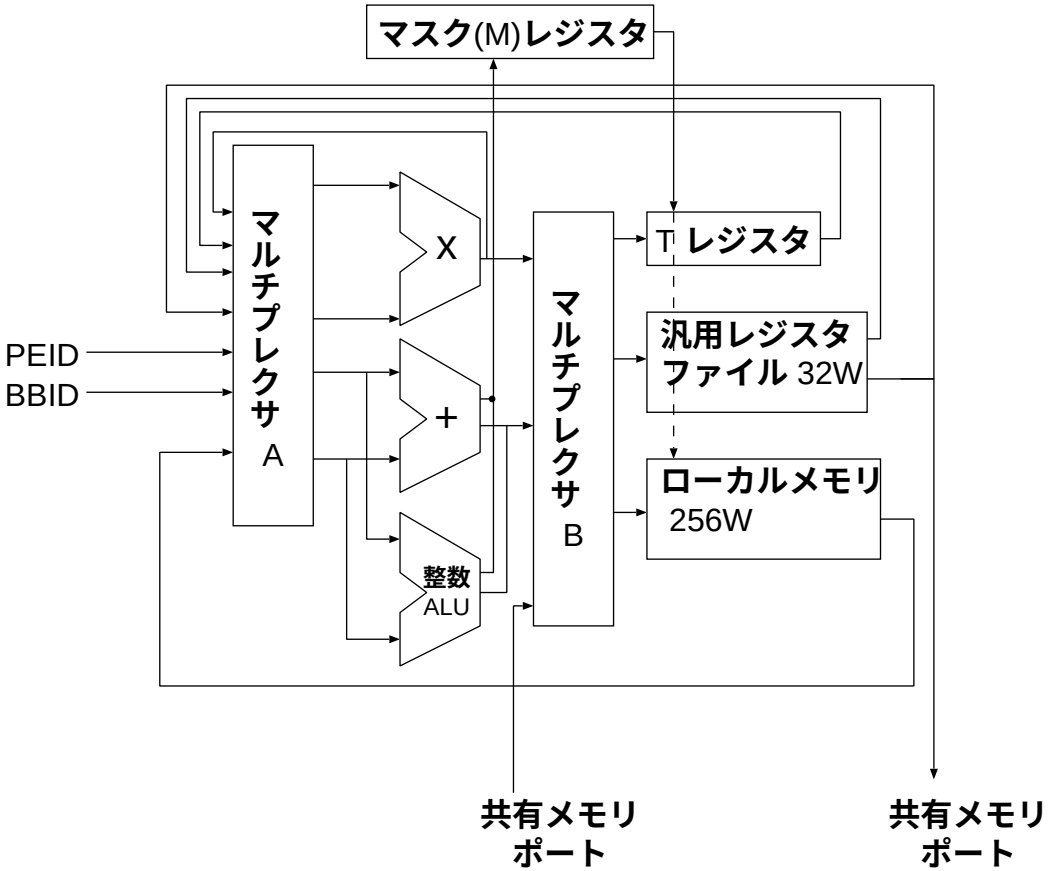
チップ外ポートから放送メモリには

- 放送
- 縮約 (総和など) しながら読み出し
- ランダム読み書き

これで必要なことは全てできる。



PE の構造



- 浮動小数点演算器
- 整数演算器
- レジスタ
- メモリ (256 語),K とか M ではない。

PEの詳細

データ形式

単精度浮動小数点: 36 ビット (符号 1、指数 11、仮数 24)
倍精度浮動小数点: 72 ビット (符号 1、指数 11、仮数 60)
36/72 ビット固定小数点数

演算命令

乗算は単精度のみ (倍精度のための部分積をサポート) 倍精度乗算を 2 サイクルでするために 25×50 ビットの乗算器
整数演算、加減算は倍精度のみ (メモリ/レジスタからの読出し/格納時に単・倍変換ができる)
特殊な浮動小数点命令: 仮数を正規化しないまま演算を続ける。これにより、演算順序によらないで結果が同じになることを保証する (GRAPE-6 の積算と同様)

PEの詳細(続き)

- パイプラインは8ステージ。
- 基本命令は4データに対するベクトル命令。4サイクルに1回しか命令ははいらない。
- T レジスタについてのみ直前の命令の実行結果を利用可能。
- T レジスタはアドレスレジスタになる (間接アクセスが可能)

サポートする命令等は基本的にはSIMD 計算機、例えば CM-2, MasPar MP-1 なんかとあまり変わらない。但し、 PE がはるかに強力になっている。

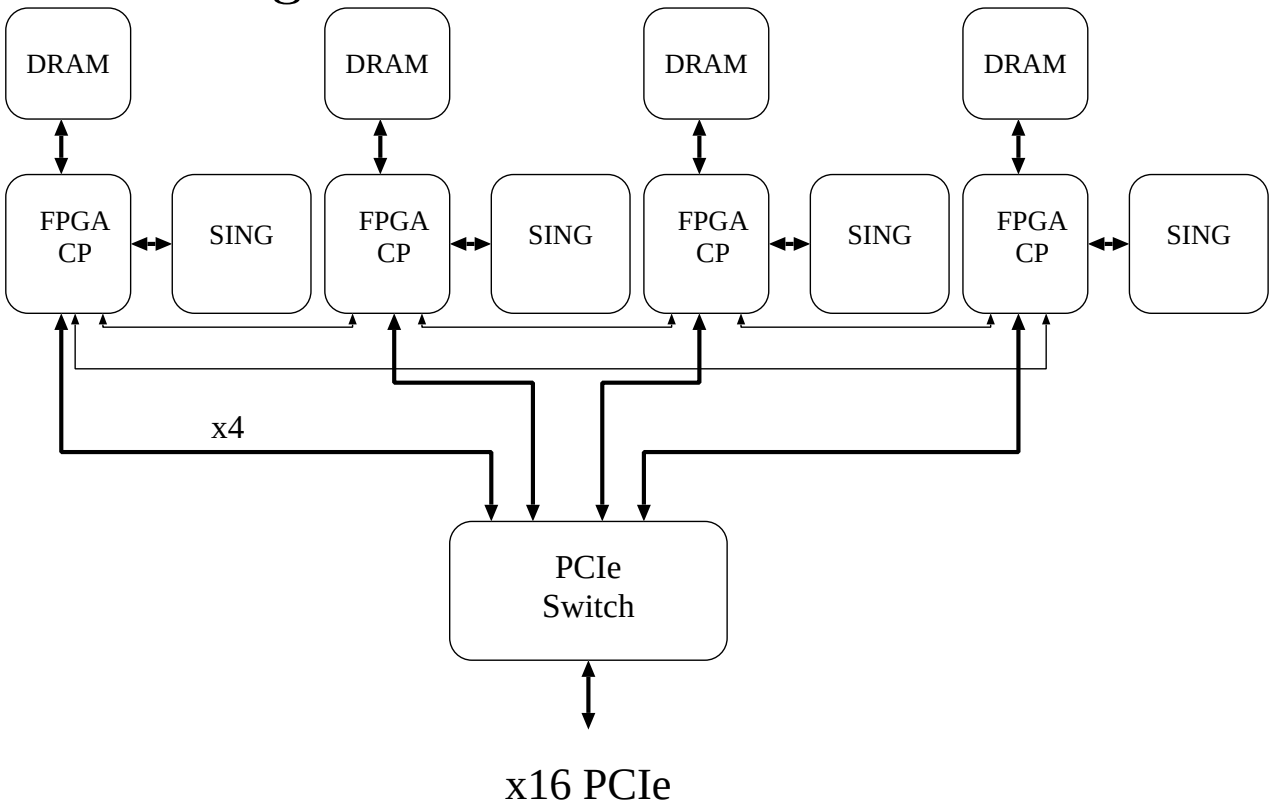
どうやってプログラムするか？という話

- 今までの GRAPE とは全然違う: プログラムを書く必要がある
- 古典的な SIMD マシンともプログラミングモデルが違う
  - GDR の部分は「付加プロセッサ」:そこでプログラム全体が走るわけではない
  - 通信モデルが違う
  - 制御プロセッサはハードウェアレベルで変更可能 (FPGA で実装)

制御プロセッサの中身を決める = プログラミングモデルを決める
ここをどうするかが問題
「プログラミング」

ハードウェアのイメージ

SING: Sing Is Not GRAPE、チップの開発コードネーム



FPGA 間は 750MB/s 双方向リング

ボードの動作の話の前にチップ内の話を。

機械語の構造

- 基本的には水平型マイクロコードそのもの
- 但し、固定長ベクトル命令 (長さ 4)。スカラ量の操作の時でも 4 サイクルかける
- 命令に必要なバンド幅を圧縮するのと、直前の命令の結果を利用可能にすることで命令スケジューリングの必要性を減らすため。

データフォーマット

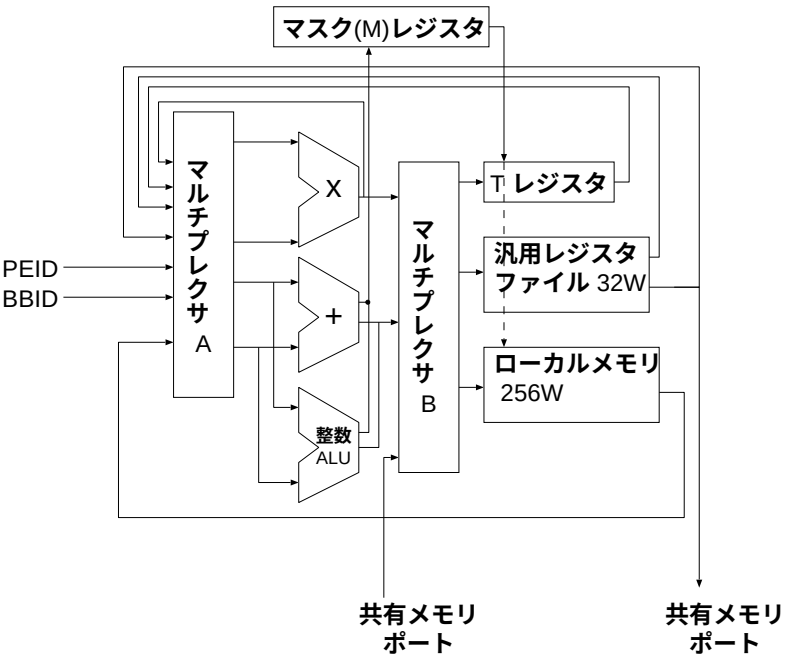
浮動小数点

- **短語 36 ビット、長語 72 ビット**
- **仮数 24/60 ビット** unbiased forced-1 rounding
- **指数 11 ビット**
- **符号ビット**
- **「非正規化数」をサポート。これは** hidden bit **を補わない。**

整数: 長、短語サポート

制御フィールド

- ループ長
- M レジスタ制御 (マスク)
- T レジスタ制御 (補助レジスタ)
- レジスタファイル制御
- ローカルメモリ制御
- FMUL 制御
- FADDSUB 制御
- IALU 制御
- BM 制御



ループ長 (LL)

LL フィールドはベクトル命令の繰り返し数を指定する。今回の実装ではフィールド長は 3 ビットで、最大値は 4 とする。それ以上を指定した時の動作は保証しない。0 の時には実効的の NOP (no operation) となる。

M レジスタ制御 (MRC)

MRC フィールドは以下の指定を行う。

```
IMR[0:2]  書き込み制御を  M レジスタのどれから取るか。 0 は ALL 1
OMR[0:2]  演算器の出力を M レジスタのどれに書くか。 0 は 書かない
IFSEL     FADD と IALU のどちらのフラグ出力を取るか
```

少しわかりにくいが、M レジスタの出力セレクトが IMR である。これによる書き込み指定は、M レジスタ自身を含めて全てのストレージ書き込みに適用される (他の指定との AND) になる。
合計 7 ビット。

T レジスタ制御

T レジスタの制御コードは以下の通り

WRITE	書き込むかどうか。 1 なら書き込む。
SHORTSTOP	書き込み (入力) データをそのまま出力に回す。
ISEL[0:1]	入力セレクト 00: FMUL 01: FADDSUB/IALU 10: BM

合計 4 ビット。
MRC の項で述べたように、マスクレジスタは MRC で指定されたものが適用される。また、アドレス初期値は読み出し、書き込みともに必ず 0 なので指定の必要はない。
なお、 shortstop 動作の時にはマスクレジスタ指定は無視され、マスクレジスタの指定にかかわらず前の命令での T レジスタ入力ポートへの入力データがそのまま出力ポートにでる。

アドレスに関する規約

レジスタファイルのアドレス指定は全て短語アドレスで行う。長語アクセスの時には LSB は無視される (non-aligned access はサポートしない)。これはローカルメモリ、放送メモリでも同様である。

レジスタファイル制御 (RFC)

レジスタファイルの制御コードは以下の通り。 合計 27 ビット。

WRITE	書き込むかどうか。 1 なら書き込む。
ISEL[0:1]	入力セレクト 00: FMUL 01: FADDSUB/IALU 10: BM
WADR[0:5]	書き込みアドレス初期値
WADRI	書き込みアドレスをインクリメントする (1)/しない
WWL	書き込みワード長 1:長語、 0:短語
RADRA[0:5]	読み出しAアドレス初期値
RADRIA	読み出しAアドレスをインクリメントする (1)/しない
RWLA	読み出しAワード長 1:長語、 0:短語
RADRB[0:5]	読み出しBアドレス初期値
RADRIB	読み出しBアドレスをインクリメントする (1)/しない
RWLB	読み出しBワード長 1:長語、 0:短語

ローカルメモリ制御 (LMC)

ローカルメモリの制御コードは以下の通り

WRITE	書き込むかどうか。 1 なら書き込む。
ISEL[0:1]	入力セレクト 00: FMUL 01: FADDSUB/IALU 10: BM
ADR[0:8]	アドレス初期値
ADRI[0:3]	アドレスインクリメントの値
TREGADR	アドレスを TREG から取る
WL	ワード長 1:長語、 0:短語

書き込みがある命令のあとの 2 命令は、読み出したデータは保証されない。
ADRI は 1 の時に連続アクセスになる、つまり長語ならアドレス増分は ADRI で指定した値の 2 倍になるものとする。
これは 18 ビット。

FMUL 制御(FMC)

浮動小数点乗算器の制御コードは以下の通り。

SHIFT50A	ポートAの仮数を50ビット上にシフト
ROUND50A	ポートAの仮数を(シフト後)50ビットに丸める
ROUNDA	ポートAの仮数を(シフト後)25ビットに丸める
NORMALA	ポートAの入力を正規化数とみなす
SHIFT25B	ポートBの仮数を25ビット上にシフト
SHIFT50B	ポートBの仮数を50ビット上にシフト
ROUNDB	ポートBの仮数を(シフト後)25ビットに丸める
NORMALB	ポートBの入力を正規化数とみなす
ROUND	出力を丸める
NORMALO	出力を正規化する
ISELA[0:1]	Aポート入力セレクト
	00: レジスタファイル A ポート
	01: レジスタファイル B ポート
	10: T レジスタ
	11: ローカルメモリ
ISELB[0:1]	Bポート入力セレクト
	00: レジスタファイル A ポート
	01: レジスタファイル B ポート
	10: T レジスタ
	11: ローカルメモリ

14 ビット。

FADDSUB 制御

浮動小数点加減算器の制御コードは以下の通り。

```
NORMALA      ポート A の入力を正規化数とみなす
NORMALB      ポート B の入力を正規化数とみなす
SIGNB        B の符号を反転する (減算)
ROUND        出力を 25 ビットに丸める
NORMALO      出力を正規化する
ISELA[0:2]   A ポート入力セレクト
             000: レジスタファイル A ポート
             001: レジスタファイル B ポート
             010: T レジスタ
             011: ローカルメモリ
             100: PE 番号 (固定)
             101: BM 番号 (固定)
             110: 乗算器フィードバック
             111: 未定義
ISELB[0:1]   B ポート入力セレクト
             00: レジスタファイル A ポート
             01: レジスタファイル B ポート
             10: T レジスタ
             11: ローカルメモリ
```

10 ビット。

FADDSUB はフラグ出力を持つ。これは、演算結果が正である時にセットされる (符号ビットを反転したものがそのまま使われる)。
ISELA の乗算器フィードバックは直前の乗算器の演算結果がそのまま入る。フィードバックパスの場合には、マスクやベクトル長とは無関係に必ず前の命令での乗算器の 4 サイクル分の出力結果がそのまま加算器の入力となる。

IALU 制御

整数 ALU の制御コードは以下の通り

IALUOP[0:4] ALU 自体の命令コード
UNSIGNED　　符号なし演算を指定(1 の時に)

6 ビット。
フラグ生成は以下のルールに従う

演算が A-B, A+B, A+1, A-1　の時: 結果が正ならフラグがセットされる
それ以外の時: 結果が all 0 ならフラグがセットされる

整数 ALU 制御コード

opcode	operation
0x0	A+B
0x1	A-B
0x2	A+1
0x3	A-1
0x4	not A
0x5	A and B
0x6	A or B
0x7	A xor B
0x8	max (A,B)
0x9	min (A,B)
0xA	A
0xB	B
0xC	A lshiftl B
0xD	A lshiftr B
0xE	A bshiftl B
0xF	A bshiftr B
0x10	logical not A
0x1F	immidiate

FADDSUB/ALU 出力セレクト

FADDSUB と IALU の出力はどちらか一方だけがメモリ等のマルチプレクサに入るという仕様なので、まず手前で選ぶ必要がる。これをこのフィールドで指定する。

FSEL　　出力選択。1 なら FADDSUB の出力を取る。 0 なら IALU

BM 制御(BMC)

BM の制御コードは以下の通り。これは PE 側からのアクセス

WRITE	書き込むかどうか。 1 なら書き込む。
ADR[0:10]	アドレス初期値
PEADR[0:4]	BM 書き込みの時にアクセスする PE 番号
WL	ワード長 1:長語、 0:短語

書き込みがある命令のあとの 2 命令は、読み出しデータは保証されない。これは 18 ビット。

IDP/BM 命令

BM へのチップ外からの書き込み。これはアドレス、データ長を先頭ワードで指定する可変長パケット。
先頭ワードの形式は以下の通り

LEN[0:7]	データ語数
ADDR[0:12]	BM の先頭アドレス
BBN[0:4]	BB 番号の指定
BBNM[0:4]	番号のマスク
SEQ[0]	シーケンシャルライトフラグ

合計 32 ビット。
LEN は長語単位の語数指定であり、 0 は 256 語と解釈される。BBN は実際に書く BB の番号を指定する。放送やマルチキャストを可能にするために、 BBNM で指定されたビット位置だけを比較し、BBNM が 0 であるビット位置は無視する。
SEQ は同一データを放送 (またはマルチキャスト) するか、 BB 毎に違うデータを送るかを制御する。 SEQ=0 の時は放送である。 SEQ=1 の時には LEN で指定された語数のデータをアクティブな (下の放送ブロックマスクレジスタがセットされていない) BB の数だけ送り、送られたものが順番に BB0 から書かれる。この時には BBN, BBNM は無視される。

RRN 命令

RRN 命令は、IDP データパケットのアドレス 0x1000 への書き込みとして指定される。命令のフィールドは以下の通り。基本的に、浮動小数点加減算器と整数 ALU ができることは全て指定可能。

ADR[0:12]	アドレス初期値
N [0:7]	語数
BBADR[0:4]	アクセスする BB 番号
REDUC	縮約モードかどうか。 1 なら縮約、 0 なら PE 選択。
WL	ワード長 1:長語、 0:短語
FSEL	出力選択。1 なら FADDSUB の出力を取る。 0 なら IALU
NORMALA	ポート A の入力を正規化数とみなす
NORMALB	ポート B の入力を正規化数とみなす
SIGNB	B の符号を反転する (減算)
ROUND	出力を 25 ビットに丸める
NORMALO	出力を正規化する
IALUOP[0:4]	ALU 自体の命令コード
UNSIGNED	符号なし演算を指定 (1 の時に)
ODPOE	ODP から出力する
SREGEN	ステータスレジスタに書き込む

どうやって使うか

108 ビットのマイクロコードを書く

```
DUM l m m m t t t t r r  r r r  r r r  r r r l l  l l l l f f f f f f f f f f f f f f f f  i i f b  b  b b
DUM l _ _ _ _ _ _ _ _ _  _ _ _  _ _ _  _ _ _ _ _  _ _ _ _ m m m m m m m m m m m m a a a a a a a a  a a s m  m  m m
DUM : i o i w l s i w i  w w w  r r r  r r r w i  a a t w u u u u u u u u u u u u d d d d d d d d  l l e _  _  _ _
DUM : m m f r m h s r s  a a w  a a w  a a w r s  d d r l l l l l l l l l l l l l d d d d d d d d  u u l w  a  p w
DUM : r r s i a o e i e  d d l  d d l  d d l i e  r r e : _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _  _ _ : r  d  e l
DUM : : : e t d r l t l  r r :  r r a  r r b t l  : i g : s s r n s s r n r n i i n n s r n i i  i u : i  r  a :
DUM : : : l e r t : e :  : i :  a i :  b i : e :  : : a : h h o o h h o o o o s s o o i o o s s  a n : t  :  d :
DUM : : : : : : s : : :  : : :  : a :  : b : : :  : : d : i i u r i i u r u r e e r r g u r e e  l s : e  :  r :
DUM : : : : : : t : : :  : : :  : : :  : : : : :  : : r : f f n m f f n m n m l l m m n n m l l  u i : :  :  : :
DUM : : : : : : o : : :  : : :  : : :  : : : : :  : : : : t t d a t t d a d a a b a a b d a a b  o g : :  :  : :
DUM : : : : : : p : : :  : : :  : : :  : : : : :  : : : : 2 5 a l 2 5 b l : l : : l l : : l : :  p n : :  :  : :
DUM : : : : : : : : : :  : : :  : : :  : : : : :  : : : : 5 0 : a 5 0 : b : o : : a b : : o : :  : e : :  :  : :
DUM : : : : : : : : : :  : : :  : : :  : : : : :  : : : : a a : : b b : : : : : : : : : : : :  : d : :  :  : :
ISP 1 0 0 0 0 0 0 0 1 1  0 0 1  0 0 0  0 0 0 0 2  2 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  4 3  A 0 0 0  0  0 1
ISP 1 0 0 0 0 0 0 0 1 1  0 1 1  0 1 1  0 0 0 0 2  2 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0  2 0 0 0  0  0 1
ISP 1 0 0 0 0 0 0 0 1 1  2 0 1  0 0 0  0 0 0 0 2  2 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  5 3  A 0 0 0  0  0 1
ISP 1 0 0 0 0 0 0 0 1 1  4 1 1  0 1 1  2 1 1 0 2  0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 3  0 0 0 0  0  0 1
ISP 1 0 0 0 0 0 0 0 0 0  0 0 1  4 1 1  0 0 0 0 0  0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 3  A 0 0 1  0  0 1
DUM
DUM IDP header format: IDP len addr bbn bbnmask, all in hex
DUM RRN format
DUM ADDR N BBADR REDUC WL FSEL NA NB SB RND NO OP UN ODP SREGEN
IDP 1 1000  0 0
RRN 0    1  0    0    1   0   0  0   0  0  0  0   0  1    1
IDP 1 1000  0 0
RRN 0    1  0    1    1   0   0  0   0  0  0  0   0  1    1
```

どうやって使うか

108 ビットのマイクロコードを書く

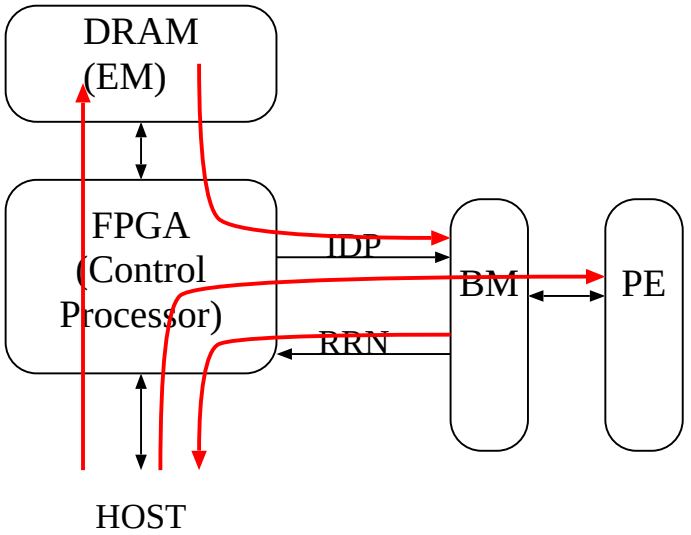
```
DUM l m m m t t t t r r  r r r  r r r  r r r l l   l l l l f f f f f f f f f f f f f f f f f f  i i f b   b  b b
DUM l _ _ _ _ _ _ _ _ _  _ _ _  _ _ _  _ _ _ _ _ _   _ _ _ _ m m m m m m m m m m m m a a a a a a a a  a a s m   m  m m
DUM : i o i w l s i w i  w w w  r r r  r r r w i   a a t w u u u u u u u u u u u u d d d d d d d d  l l e _   _  _ _
DUM : m m f r m h s r s  a a w  a a w  a a w r s   d d r l l l l l l l l l l l l l d d d d d d d d  u u l w   a  p w
DUM : r r s i a o e i e  d d l  d d l  d d l i e   r r e : _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _  _ _ : r   d  e l
DUM : : : e t d r l t l  r r :  r r a  r r b t l   : i g : s s r n s s r n r n i i n n s r n i i  i u : i   r  a :
DUM : : : l e r t : e :  : i :  a i :  b i : e :   : : a : h h o o h h o o o o s s o o i o o s s  a n : t   :  d :
DUM : : : : : : s : : :  : : :  : a :  : b : : :   : : d : i i u r i i u r u r e e r r g u r e e  l s : e   :  r :
DUM : : : : : : t : : :  : : :  : : :  : : : : :   : : r : f f n m f f n m n m l l m m n n m l l  u i : :   :  : :
DUM : : : : : : o : : :  : : :  : : :  : : : : :   : : : : t t d a t t d a d a a b a a b d a a b  o g : :   :  : :
DUM : : : : : : p : : :  : : :  : : :  : : : : :   : : : : 2 5 a l 2 5 b l : l : : l l : : l : :  p n : :   :  : :
DUM : : : : : : : : : :  : : :  : : :  : : : : :   : : : : 5 0 : a 5 0 : b : o : : a b : : o : :  : e : :   :  : :
DUM : : : : : : : : : :  : : :  : : :  : : : : :   : : : : a a : : b b : : : : : : : : : : : :  : d : :   :  : :
ISP 1 0 0 0 0 0 0 0 1 1  0 0 1  0 0 0  0 0 0 0 2   2 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  4 3  A 0 0 0   0  0 1
ISP 1 0 0 0 0 0 0 0 1 1  0 1 1  0 1 1  0 0 0 0 2   2 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 0  2 0 0 0   0  0 1
ISP 1 0 0 0 0 0 0 0 1 1  2 0 1  0 0 0  0 0 0 0 2   2 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  5 3  A 0 0 0   0  0 1
ISP 1 0 0 0 0 0 0 0 1 1  4 1 1  0 1 1  2 1 1 0 2   0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 3  0 0 0 0   0  0 1
ISP 1 0 0 0 0 0 0 0 0 0  0 0 1  4 1 1  0 0 0 0 0   0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  0 3  A 0 0 1   0  0 1
DUM
DUM IDP header format: IDP len addr bbn bbnmask, all in hex
DUM RRN format
DUM ADDR N BBADR REDUC WL FSEL NA NB SB RND NO OP UN ODP SREGEN
IDP 1 1000  0 0
RRN 0    1  0    0    1   0   0  0   0  0  0  0   0  1    1
IDP 1 1000  0 0
RRN 0    1  0    1    1   0   0  0   0  0  0  0   0  1    1
```

牧野は 3 行書いて嫌になった。

SING アセンブラ

現行の「アセンブラ」:計算モデル (制御プロセッサの中身) も規定
まずそっちの話を。

計算モデル(現行の)



制御するのは

- PE の動作
- Host → EM
- EM → BM
- Host → PE
- BM → Host

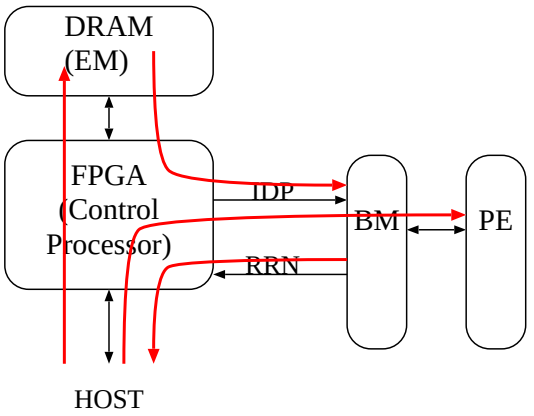
の５個。

歴史的な経緯でホストから直接 PE に書き込みがいったりして、ちょっと変。

簡単な例: 粒子間相互作用

$$f_i = \sum_j f(x_i, x_j)$$

x_j データをオンボードメモリ (EM) に格納
x_i データをオンチップメモリ (LM) に格納
1 ループに必要な個数の x_j データを EM から BM に転送
相互作用の計算・積算
PE から計算結果を回収。必要に応じて縮約しながら。

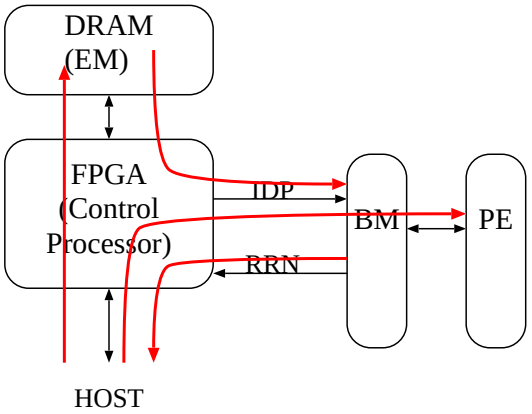


実際の動作は結構複雑
計算している間に次の x_j データ転送と結果回収、および次の x_i データ転送 (必要なら EM にバッファリング) をしたい
但し、これはハードウェアが勝手にすればよくて、ユーザはあんまり関係ない

もう一例: 行列乗算

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

単純な実装
b データをオンボードメモリ (EM) に格納
　a データをオンチップメモリ (LM) に格納
　　b 1 行を EM から BM に転送
　　c を計算　PE から計算結果を回収。必要に応じて縮約しながら。



計算している間に次の b データ転送と結果回収、および次の a データ転送 (必要なら EM にバッファリング) をしたい

実は動作は相互作用計算と全く同じ。内側ループで結果出力するかどうかだけが違う

行列演算の最適化
(通信と計算のオーバーラップ)

前スライドのアルゴリズムでは b の転送は決して隠蔽されない。
A が (M, K) 行列、 B が (K, N) 行列で、$M \gg K, N \gg K$ の時に
(LINPACK ではこれが主要な計算) 両方隠蔽したい。

A, B を途中でひっくり返すと隠蔽可能。

$A(K, K)$ 分を EM に転送
B 全部を EM 経由で転送しながら $C(K, N)$ を計算
A の残りを転送しながら C の残りを計算

(まだ実装できてない、、、)

アセンブラ定義の概要

- 放送メモリ、 PE ローカルメモリに変数を置くことができる
- PE レジスタはアドレスで直接指定。
- ベクトル変数とスカラー変数がある
- 並列に実行する命令は明示的に指定する。

アセンブラの例

単純な重力計算の例

```
var vector long xi     hlt  flt64to72
var vector long yi     hlt  flt64to72
var vector long zi     hlt  flt64to72
var vector short idxi  hlt  fix32to36ru
bvar long xj           elt  flt64to72
bvar long yj           elt  flt64to72
bvar long zj           elt  flt64to72
bvar long vxj xj
bvar short mj        elt  flt64to36
bvar short eps2      elt  flt64to36
bvar short idxj      elt  fix32to36ru
var short  lmj
var short  leps2
var short  lidxj
var vector long accx rrn  flt72to64 fadd
var vector long accy rrn  flt72to64 fadd
var vector long accz rrn  flt72to64 fadd
var vector long pot  rrn  flt72to64 fadd
```

ここまでで変数宣言。 hlt, elt, rrn は通信タイプ。そのあとのは変換タイプ

アセンブラの例 (続き)

```
loop initialization
vlen 4
uxor $t $t $t
upassa $ti $ti $lr40v
upassa $t $t   $lr48v
upassa $t $t   $lr56v
upassa $t $t   pot
loop body
vlen 3
bm vxj $lr0v
vlen 1
bm mj lmj
bm eps2 leps2
bm idxj lidxj
```

初期化 (ループ前に実行) とループ本体の一部 (放送メモリからの転送)

アセンブラの例(続き)

```
vlen 4
nop
upassa idxi  idxi  $t
uxor  $ti lidxj $t
moi 2
ulnot $ti $ti $t  # mreg 1 indicates i != j
moi 0
nop
fsub $lr0 xi $r6v  $t
fsub $lr2 yi $r10v ;  fmul $ti $ti $t
fsub $lr4 zi $r14v
fmul $r10v $r10v $r18v ; fadd $t leps2 $t
fmul $r14v $r14v ;  fadd $fb $ti  $t
fadd $fb $ti  $r18v $t # rsq is now in r18 t, dx, dy,dz are in  6,1
```

自己相互作用のチェック、座標の引き算と距離の 2 乗の計算

アセンブラの例(続き)

```
ulsr $ti      il"60"  $t $lr22v
ulsr $ti      il"1"   $t
uadd $ti      $lr22v  $t
usub hl"9fd" $ti      $t        # $lr8v は指数の 1.5 倍
ulsl $ti      il"60"  $lr30v
moi 1
uand il"1" $lr22v
moi 0
uand $r18v h"000ffffff" $t
uor  $ti h"3ff000000" $t
fmul $ti f"0.57" $t
fsub f"1.57" $ti $t
mi 1
fmul f"1.414" $ti $t
mi 0
nop
fmul $t $lr30v $t $r22v # Here the result is the initial guess
```

r^{-3} の初期推定値。これは手抜きな 1 次式

アセンブラの例(続き)

```
 fmul $r18v $r18v $r26v $t
 fmul $r18v $ti   $r26v $t
 fmul $ti f"0.5" $r26v # r26v is a**3/2
 fmul $r22v $r22v $t
 fmul $ti $r26v $t
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t $r22v
 fmul $ti $ti $t
 fmul $ti $r26v $t
(反復ちょっと省略)
 fsub f"1.5" $ti $t
 fmul $r22v $ti $t $r22v
 fmul $ti $ti $t
 fmul $ti $r26v $t
 fsub f"0.5" $ti $t
 fmul $r22v $ti $t
 fadd $r22v $ti $t
 fmul lmj $ti $t $r22v
```

ニュートン反復

アセンブラの例 (続き)

```
mi 2
fmul $r6v $ti ; upassa pot pot $lr0v
fmul $r10v $t ;  fadd $fb $lr40v  $lr40v accx
fmul $r14v $t ; fadd $fb $lr48v  $lr48v accy
fmul $r18v $t ; fadd $fb $lr56v $lr56v accz
fadd $fb $lr0v  pot
```

ポテンシャルと加速度の積算
この記述から、インターフェース関数とシミュレータを動かすのに必要なデータを生成する。
これは粒子系用に、データ転送を明示しないもの。 IDP/RRN 動作を記述することもできる。

インターフェース関数

中身はアセンブラ記述から自動生成される。

```
int SING_send_j_particle(struct grape_j_particle_struct *jp,
                         int index_in_EM);
int SING_send_i_particle(struct grape_i_particle_struct *ip,
                         int n);
int SING_get_result(struct grape_result_struct *rp);
void SING_grape_init();
int SING_grape_run(int n);
```

これにもう一層かぶせれば GRAPE-3/5 互換インターフェースはできる。

インターフェース構造体

これももちろん自動生成される。

```
struct grape_j_particle_struct{
  double xj;
  double yj;
  double zj;
  double mj;
  double eps2;
  UINT32 idxj;
};
struct grape_i_particle_struct{
  double xi;
  double yi;
  double zi;
  UINT32 idxi;
};
struct grape_result_struct{
  double accx;
  double accy;
  double accz;
  double pot;
};
```

行列乗算コード (一部)

```
## even loop
bm b10 $lr0v
bm b11 $lr8v
dmul0 $lr0 $lm0v     ;   bm $lr32v c0 0       ; rrn fadd c0 256 flt72t
dmul1 $lr0 $lm0v     ; upassa $fb $t $t       ; idp 0
dmul0 $lr0 $lm256v   ; faddAB $fb $ti $lr48v ; bm $lr40v c1 0
dmul1 $lr0 $lm256v   ; upassa $fb $t $t
dmul0 $lr2 $lm8v     ; faddAB $fb $ti $lr56v ; bm $lr32v c2 1
dmul1 $lr2 $lm8v     ; faddA $fb $lr48v $t
.....
dmul0 $lr14 $lm504v ; faddA $fb $ti $lr32v  ; bm $lr40v c63 31
dmul1 $lr14 $lm504v ; faddA $fb $lr56v $t
faddA $fb $ti $lr40v
nop
```

内積計算、EM から BM への転送、 LM から BM への転送、 BM からの結果出力を全て並行に行うことを、明示的に記述。
BM はデュアルポートだが、PE からのアクセスと RRN 出力でアドレス共通。これらは 2 サイクルに 1 語でよいため。

「並列計算機用言語」としての特徴

- 並列性を記述しない
- 並列度は実行環境任せ
- ハードウェアの違いは呼び出し側で対応 (隠蔽しないのは実行効率を出しやすくするため)
- 記述量は (アセンブラなので多い、というのを別にすると) 少ない

GRAPE-DR アセンブラ記述の利害得失

利点:

- 書いた時点で速度がわかる
- 並行動作を全てプログラマが制御できる
- データ型も自由に制御できる
- データ分配、並列制御、縮約制御等はやってくれる。

欠点

- 実行部書くの面倒くさい

コンパイラ

中里 (2008) LLVM を使った、最適化とかもするもの

```
VARI a, b;
VARJ z;
VARF tt;

function test(x) {
  res = 1 - x**2;
}

dx = (b-a)/100.0;
res = 0.5*test(a) + 0.5*test(b);
x = a;
for(i, 1, 99) {
  x = x + dx;
  res = res + test(x);
}

tt = z*res;
```

コンパイラの動作

- アセンブラに翻訳される
- インターフェース関数等はアセンブラから生成
- そこの使いかたは同じ

まとめ

- GRAPE-DR のプログラミング環境、特に機械語レベルとアセンブラについてまとめた。
- アセンブラから並列動作し、通信等も最適化されたライブラリ関数が生成される。
- 最適化コンパイラもある。