メタプログラミングによる高速計算の実現

中里直人 (会津大学)

台坂博(一橋大学), 石川正(KEK), 湯浅富久子 (KEK),
牧野淳一郎(東工大)

2012年3月5日 九州大学

自己紹介

- 中里直人 (なかさと なおひと)
  - 会津大学 准教授 博士(理学)
  - 専門分野 天文シミュレーションとHPC
    - SPH法による銀河形成シミュレーション
    - FPGAボードによる高速計算
    - 専用計算機の研究開発
    - 専用計算機用ソフトウエアの研究開発
    - メニーコア計算機(GRU,GRAPE-DR)による高性能計算
  - http://galaxy.u-aizu.ac.jp/trac/note/

Agenda

- 浮動小数点演算の盲点
- 「自作」計算機
- メタプログラミングによる演算回路設計
- 高速計算とメタプログラミング

浮動小数点演算の罠

- 倍精度演算が広く利用されている
  - IEEE 754規格　仮数部　53 bit(10進数で16桁)
  - 90年代以降の計算機はハードウエアで実装：高速
- それで「必要十分」なのかは、たいていの場合まともに検討されていない
- 21世紀：倍精度が遅い計算機の出現
  - Cell, SSE演算, GPU　単精度演算の1/2の性能
    - GPU (AMD)　1/4 - 1/5のものもある
  - 単精度で十分ならそれでやるほうがいい

倍精度演算は必要十分か？

- そうでもない
  - 条件数が非常に大きい(>10^{16})行列の演算
  - メッシュを再帰的に分割するAMR法
  - ファインマンループの数値積分
  - 単純な例：~ 1.1726 @ 倍精度演算

$$a = 77617.0$$
$$b = 33096.0$$
$$f = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b} = -\frac{54767}{66192}$$

高精度演算の手法

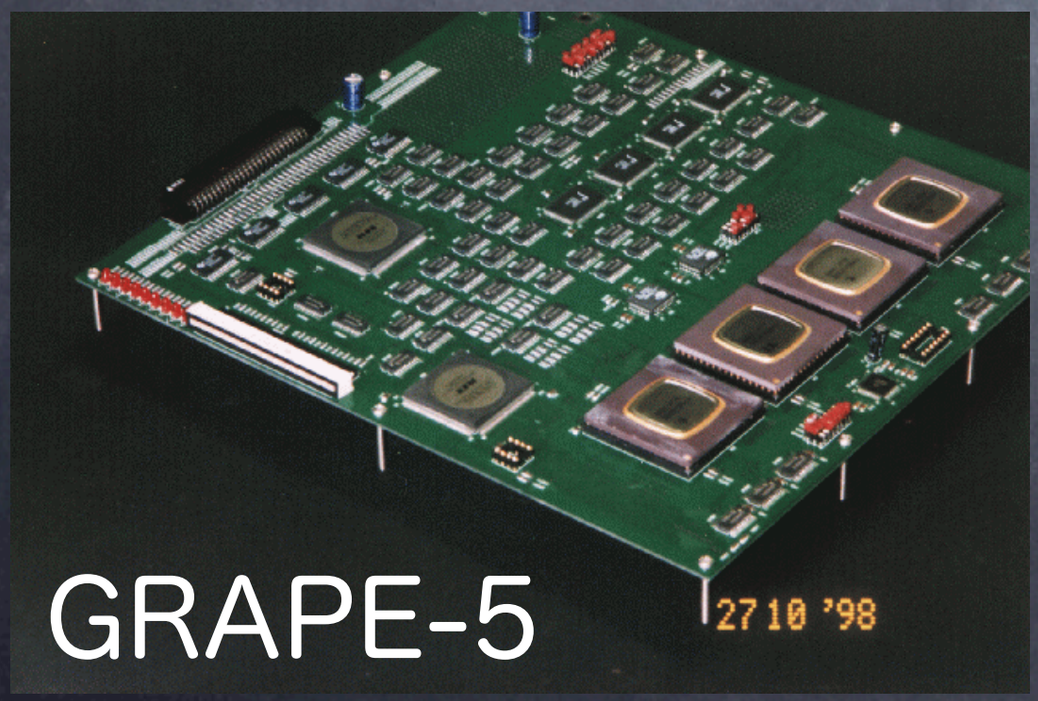
- FP演算器によるソフトウエア実装
  - Knuth 1969 & Dekker 1971のエラーフリー演算
  - QD Library (Hida, Li & Bailey)
  - 四倍精度なら1演算につき約20回のFP演算が必要
    - FMAなしCPUでは1 coreあたり ~ 100 Mflops (mpack 中田(2011))
    - AMD GPU ~ 70 Gflops (中村&中里(2012))
    - NVIDIA GPU ~ 30 Gflops (中田 2011)
    - CPUでも~ 4 Gflops (4 core) (中村&中里(2012))
- INT演算によるソフトウエア実装
  - GNU-MP, FPR, exflib, HMLIB etc...

逆にそんなに桁のいらない場合

- 重力多体問題：GRAPE計算機
  - 力のcancellationがないため

$$f_i = \sum_{j=1}^{N} \frac{m_j (\vec{x_i} - \vec{x_j})}{(|\vec{x_i} - \vec{x_j}|^2 + \epsilon^2)^{1.5}}$$

ここだけ精度が必要



	加算	乗算
GRAPE-1	16/48	5
GRAPE-3	19/48	5
GRAPE-4	64	24
GRAPE-5	32	8
GRAPE-6	64	24

http://jun.artcompsci.org/articles/future_sc/note034.html

演算精度と専用計算機

- CPUは単精度と倍精度のみサポート
  - 他の精度はエミュレーションすることができる
    - 高精度　GNU-MP, FPR, exflib, HMLIB, QD, mpack etc...
      - ただし倍精度と比べて10 - 20倍以上遅くなる
    - 低精度　ない？ GPUで半精度(16 bit)がサポートされる？
- ニッチな精度が必要な場合高速化の余地が大いにある
  - 低精度演算の採用によるGRAPE-1/3/5の成功
  - FPGAを利用した専用計算機
  - 高精度演算のための専用計算機 GRAPE-MP

「自作」計算機

- 必要なものがないなら自分で作る発想
  - 全てのイノベーションの基本
  - 1950年代の「電子」コンピュータ
    - 基本的にどれも「自作」
    - 最終的に使う人達(主に科学者)が、詳細の仕様を決定していた
    - そもそも軍事目的がモチベーションだった
  - その後、多くの商用計算機がつくられ、計算機自体がひとつの研究分野となる
    - programming, computer engineering/scienceの誕生
    - と同時に「自作」の衰退 1960 - 1970年代

「自作」計算機の復権

- 1980年代：再び科学者たちが自分のための専用計算機を「自作」しはじめる
  - Caltech Cosmic Cube (1981)
    - QCD専用計算機：COTSによる初の並列計算機
  - Digital Orrery (1985)
    - 太陽系がカオスかどうかを知りたい
  - GRAPE-1 (1989)
    - 重力多体系を調べため。重力熱振動の検証。
  - DIYマイコンの時代でもある

いまでも「自作」に意味はあるか？

- ニッチな演算精度が必要なら
  - 問題特化による高性能
    - 汎用プロセッサには無駄が多い
  - 問題特化による高効率(場所,電力)
    - 低予算でやりたい仕事ができる
- 一方で　汎用 vs. 専用の競争
  - 専用なら汎用を圧倒する性能がないとすぐに負ける
    - 「牧本ウェーブ」
  - ニッチな演算精度はすぐには汎用化しなさそう

GRAPE的な手法の限界

- LSIの設計コストが莫大になった
  - 億単位の予算がないとcustom ASICは設計できない
  - 予算確保がどんどん困難に
  - GRAPE-DRは準汎用計算機 (GPUとほぼ同じ)
- 設計や検証にかかる時間の問題
  - 最低でも1 - 2年のプロジェクトになる
  - その間に商用の汎用計算機は最低数倍高速化する
- 少人数では不可能になりつつある
  - 過去のGRAPE関連プロジェクトは、どれも実働数名
  - ただし、数名を超えるチームが必要なアーキをやらないのは意図的な戦略

必要リソースの最小化戦略

- FPGAや構造化ASICの採用
  - FGPA：Field Programmable Gray Array
  - 商用品のため、LSIを設計しなくてよい
    - シリコン用マスクを作らなくてもよい
    - Hardware Description Languageによる「プログラミング」が可能
  - ただしシリコン面積あたりの性能は落ちる
    - ネットワーク配線のため
    - クロック周波数がどうしても遅い
- GRAPE-7, GRAPE-MP, GRAPE-8

我々が必要なもの

- やりたいのは粒子シミュレーション
  - 倍精度ではない精度で加速度を計算したい
    - 「必要十分」な演算精度は本来問題依存
  - あるいは高精度演算を速くしたい
- 精度を調整可能な浮動小数点回路を簡単に扱う方法論が必要
  - 解決策：IPコアを買ってくる
    - しかし、高価で、低精度や高精度は既存のIPでは未サポート
  - FPGAベンダーのIPコア
    - そのベンダーのFPGAのみでしか使えない

FPGAによる計算機開発(1)

- 原理的にはどんな「僕の考えた最強のアーキテクチャ」でも実装可能
  - 実際には：
    - FPGAはメモリ少ない
    - メインメモリをつなぐのは大変 or 高価(IPコアを導入)
    - 自律動作て。。。
- GRAPE的に利用するとするなら
  - 演算はパイプライン
  - メモリアクセスが簡単になる
  - 我々は汎用計算機は必要としていない

FPGAによる計算機開発(2)

- アーキテクチャを決めたら
  - 我々が必要としている部品は：
    - パイプライン用の浮動小数点演算回路
    - レジスタファイル
    - パイプラインを制御する状態機械
    - メモリアクセスのための制御回路
- ここまで決まっていても開発は容易ではない
  - HDLだけを書けばいいわけではない
  - 論理合成と配置配線、そして検証に時間がかかる
  - 優秀な大学院生が専任で最低半年の仕事量になる

演算回路設計は部品設計

- 浮動小数点の加算に必要な部品
  - 整数比較
  - 右左シフタ
  - マルチプレクサ
  - 整数加算機
  - プライオリティエンコーダ
  - パイプラインレジスタ
- 全てをパラメータ化する必要がある
  - さらにそれぞれの「ソフト」エミュレーション

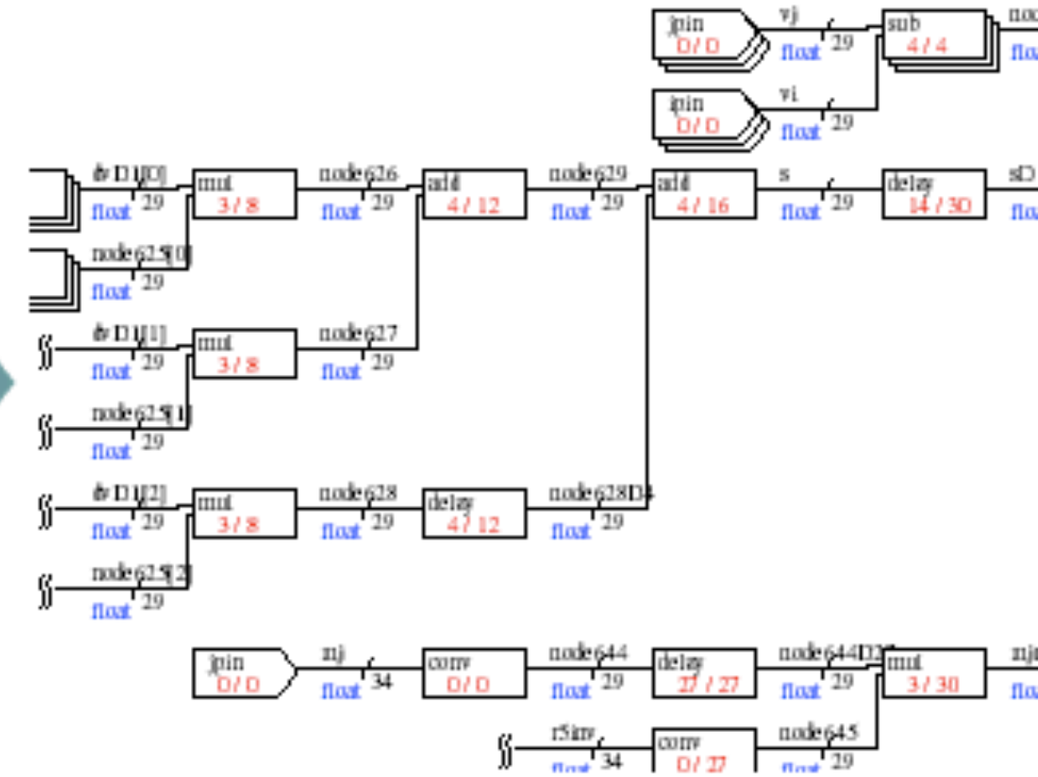
解決策：PGPGとPGR

- FPGAを使ったPROGRAPE-1
  - Hamada, Fukushige, Kawai, & Makino (2000)
  - FPGAの採用により「専用」計算機ではなくなった
  - でもどうやって「計算」を定義するのか

- 2000年頃 PGPGが提案された(らしい)
  - 福重が提案。福重, 濱田が実装。
  - C言語で精度可変の回路を生成
    - PGPG by Hamada, Fukushige, Makino (2005)
    - PGR by Hamada & Nakasato (2007) (未踏スーパークリエーター認定)

PGPG2の例

```
int10       jshift  (int)ipin; // (the library handle
int64       acc[3]  fout;
int32       jerk[3] fout;

// force calculation
dx     = (float34.23)(xj - xi);
r2     = dx * dx + epsi2;
r5inv = (float34.23)pg_pow(r2, -5, 2, 9);
r3inv = r5inv * r2;
r1inv = r3inv * r2;
acc   += (int64)((mj * r3inv * dx) << fshift);

// jerk calculation
dv       = (float29.18)(vj - vi);
s        = (dv * (float29.18)dx);
mjr5inv = (float29.18)mj * (float29.18)r5inv;
j1       = s * mjr5inv * (float29.18)dx;
j1a      = j1 << 1;
```



高級言語による記述から、ハードウェア記述言語 VHDL で記述されたパ イプライン回路を自動生成します。典型的なケースでは、高級言語による記述 10〜100 行をもとに VHDL による記述 1000〜10000 行が生成されます。

http://www.kfcr.jp/pgpg2.html

HDLによる回路設計

- ブロック図を作る (易)
- 各ブロックのHDLによる回路設計 (難)
  - HDLで動作させたい仕様の決定 (易)
  - それを再現するソフトウエアの実装(多少難)
    - ソフトウエアに対応するHDL記述を作成 (難)
    - ソフトとHDLの動作を比較(テストベンチ)
      - 難ではないが時間がかかる：全てのパターンを試せるか？
- 全てのブロックを接続 (難)
- ボードの回路設計 (困難+時間かかる)

Description vs. Programming

- HDLによる「プログラミング」は、実はプログラミングではない
  - FPGAのPは、hardware descriptionをメモリにprogram(書き込む)という意味らしい
- HDL自体には
  - 変数がない: すべては信号(signal)
  - フリップフロップが変数のようなもの
  - 信号はつねに行き交っている(並列動作)
  - ただし「プログラミング」的な要素も含んでいる
    - 「パラメータ化」が可能：例えば浮動小数点加算機の仮数部を可変に

VHDLによる左シフト

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity int_lshift_27_8_0 is
  port (
    ss : in std_logic_vector(7 downto 0);
    i : in std_logic_vector(26 downto 0);
    o : out std_logic_vector(26 downto 0)
  );
end int_lshift_27_8_0;

architecture source of int_lshift_27_8_0 is
begin
  with ss select
      o <= i(26 downto 0) when "00000000", -- 0
           i(25 downto 0)&"0" when "00000001", -- 1
           i(24 downto 0)&"00" when "00000010", -- 2
           i(23 downto 0)&"000" when "00000011", -- 3
           i(22 downto 0)&"0000" when "00000100", -- 4
           i(21 downto 0)&"00000" when "00000101", -- 5
           i(20 downto 0)&"000000" when "00000110", -- 6
           i(19 downto 0)&"0000000" when "00000111", -- 7
           i(18 downto 0)&"00000000" when "00001000", -- 8
           i(17 downto 0)&"000000000" when "00001001", -- 9
           i(16 downto 0)&"0000000000" when "00001010", -- 10
           i(15 downto 0)&"00000000000" when "00001011", -- 11
           i(14 downto 0)&"000000000000" when "00001100", -- 12
           i(13 downto 0)&"0000000000000" when "00001101", -- 13
           i(12 downto 0)&"00000000000000" when "00001110", -- 14
           i(11 downto 0)&"000000000000000" when "00001111", -- 15
           i(10 downto 0)&"0000000000000000" when "00010000", -- 16
           i(9 downto 0)&"00000000000000000" when "00010001", -- 17
           i(8 downto 0)&"000000000000000000" when "00010010", -- 18
           i(7 downto 0)&"0000000000000000000" when "00010011", -- 19
           i(6 downto 0)&"00000000000000000000" when "00010100", -- 20
           i(5 downto 0)&"000000000000000000000" when "00010101", -- 21
           i(4 downto 0)&"0000000000000000000000" when "00010110", -- 22
           i(3 downto 0)&"00000000000000000000000" when "00010111", -- 23
           i(2 downto 0)&"000000000000000000000000" when "00011000", -- 24
           i(1 downto 0)&"0000000000000000000000000" when "00011001", -- 25
           i(0)&"00000000000000000000000000" when others; -- 26
end source;
```

VHDLによる左シフト

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity int_lshift_27_8_0 is
  port (
    ss : in std_logic_vector(7 downto 0);
    i : in std_logic_vector(26 downto 0);
    o : out std_logic_vector(26 downto 0)
```

```
with ss select
    o <= i(26 downto 0) when "00000000", -- 0
         i(25 downto 0)&"0" when "00000001", -- 1
         i(24 downto 0)&"00" when "00000010", -- 2
         i(23 downto 0)&"000" when "00000011", -- 3
         i(22 downto 0)&"0000" when "00000100", -- 4
         i(21 downto 0)&"00000" when "00000101", -- 5
         i(20 downto 0)&"000000" when "00000110", -- 6
         i(19 downto 0)&"0000000" when "00000111", -- 7
```

```
         i(6 downto 0)&"0000000000000000000" when "00010100", -- 20
         i(5 downto 0)&"00000000000000000000" when "00010101", -- 21
         i(4 downto 0)&"000000000000000000000" when "00010110", -- 22
         i(3 downto 0)&"0000000000000000000000" when "00010111", -- 23
         i(2 downto 0)&"00000000000000000000000" when "00011000", -- 24
         i(1 downto 0)&"000000000000000000000000" when "00011001", -- 25
         i(0)&"0000000000000000000000000" when others; -- 26
end source;
```

VHDLとHTML

- VHDLの面倒なところ
  - 記述が冗長
  - 同じようなことを繰り返し記述する必要がある
  - 「レジスタ」の暗黙的な宣言
  - VHDLでの繰り返し処理のあつかい
    - VHDLの中に別の言語が埋め込まれている。変数あり(!)
- HTMLの面倒な(略)を解決するには
  - テンプレートエンジン
    - 例：決まり切ったFormを自動生成

テンプレートエンジンの活用

- PGRを実装した経験から
  - Cでべた書きは嫌。m4マクロはどうか？
  - Rubyでどうにかならないか？
- そこでテンプレートエンジンを採用
  - ある記述からカスタマイズされたHTMLを生成
  - HTMLにない「ループ」や条件分岐が可能
  - 2006年頃。。。Ruby on Railsが流行だした
    - RoRのキーテクノロジはメタプログラミング
  - RoRは使ったことがあった：eRuby

静的な部分と動的な部分

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity int_lshift_27_8_0 is
  port (
    ss : in std_logic_vector(7 downto 0);
    i : in std_logic_vector(26 downto 0);
    o : out std_logic_vector(26 downto 0)
  );
end int_lshift_27_8_0;

architecture source of int_lshift_27_8_0 is
begin
  with ss select
      o <= i(26 downto 0) when "00000000", -- 0
           i(25 downto 0)&"0" when "00000001", -- 1
           i(24 downto 0)&"00" when "00000010", -- 2
           i(23 downto 0)&"000" when "00000011", -- 3
           i(22 downto 0)&"0000" when "00000100", -- 4
           i(21 downto 0)&"00000" when "00000101", -- 5
           i(20 downto 0)&"000000" when "00000110", -- 6
           i(19 downto 0)&"0000000" when "00000111", -- 7
           i(18 downto 0)&"00000000" when "00001000", -- 8
           i(17 downto 0)&"000000000" when "00001001", -- 9
           i(16 downto 0)&"0000000000" when "00001010", -- 10
           i(15 downto 0)&"00000000000" when "00001011", -- 11
           i(14 downto 0)&"000000000000" when "00001100", -- 12
           i(13 downto 0)&"0000000000000" when "00001101", -- 13
           i(12 downto 0)&"00000000000000" when "00001110", -- 14
           i(11 downto 0)&"000000000000000" when "00001111", -- 15
           i(10 downto 0)&"0000000000000000" when "00010000", -- 16
           i(9 downto 0)&"00000000000000000" when "00010001", -- 17
           i(8 downto 0)&"000000000000000000" when "00010010", -- 18
           i(7 downto 0)&"0000000000000000000" when "00010011", -- 19
           i(6 downto 0)&"00000000000000000000" when "00010100", -- 20
           i(5 downto 0)&"000000000000000000000" when "00010101", -- 21
           i(4 downto 0)&"0000000000000000000000" when "00010110", -- 22
           i(3 downto 0)&"00000000000000000000000" when "00010111", -- 23
           i(2 downto 0)&"000000000000000000000000" when "00011000", -- 24
           i(1 downto 0)&"0000000000000000000000000" when "00011001", -- 25
           i(0)&"00000000000000000000000000" when others; -- 26
end source;
```

eRubyによるHDL生成

```
architecture source of <%= @name %> is
begin
% s = n-1
   with ss select
%(0..(n-1)).each { |i|
%   u = s - i
%   l = u - n + 1
%   if l < 0 then
%     z = 0.to_b(-l)
%     zero = "&\"#{z}\""
%     l = 0
%   else
%     zero = ''
%   end
%   case i
%   when 0
      o <= i(<%= u %> downto <%= l %>) when "<%= i.to_b(@nexp) %>", -- <%= i %>
%   when n-1
          i(<%= u %>)<%= zero %> when others; -- <%= i %>
%   else
          i(<%= u %> downto <%= l %>)<%= zero %> when "<%= i.to_b(@nexp) %>", -- <%= i %>
%   end
%}
end source;
```

eRubyによるメタプログラミング

(VHDL + 埋め込みRuby)

eRubyによるHDL生成

```
architecture source of <%= @name %> is
begin
% s = n-1
  with ss select
%(0..(n-1)).each { |i|
%   u = s - i
%   l = u - n + 1
%   if l < 0 then
%     z = 0.to_b(-l)
%     zero = "&\"#{z}\""
```

eRubyによるメタプログラミング

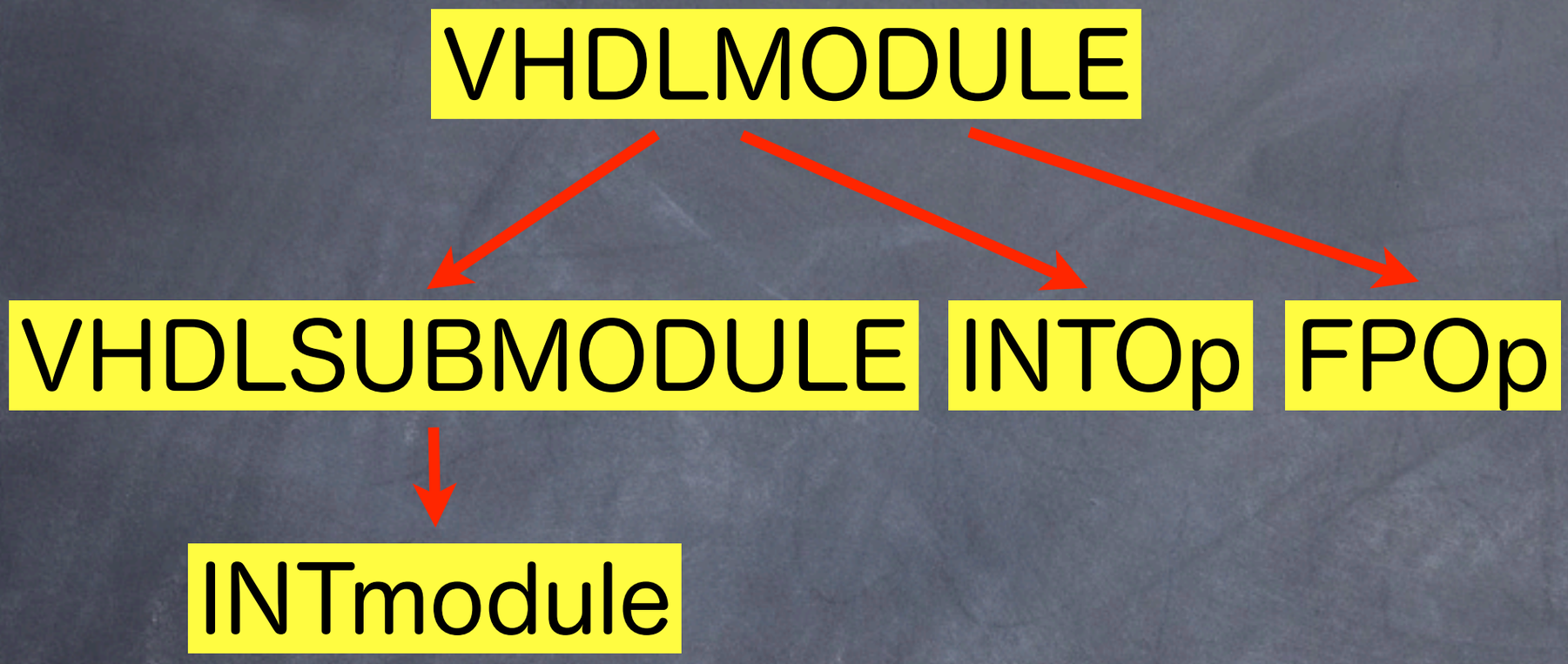
(VHDL + 埋め込みRuby)

```
with ss select
     o <= i(26 downto 0) when "00000000", -- 0
          i(25 downto 0)&"0" when "00000001", -- 1
          i(24 downto 0)&"00" when "00000010", -- 2
          i(23 downto 0)&"000" when "00000011", -- 3
          i(22 downto 0)&"0000" when "00000100", -- 4
          i(21 downto 0)&"00000" when "00000101", -- 5
          i(20 downto 0)&"000000" when "00000110", -- 6
          i(19 downto 0)&"0000000" when "00000111", -- 7
```

eRubyによるメタプログラミング

- HDLで回路を設計するには
  - ソフトウエアによる動作定義 (C言語)
  - それに対応するHDL記述
  - シミュレーション
    - 二つの動作を比較するには
    - C言語プログラムの実行結果→テストベンチ生成→シミュレーション→結果の比較
- できるだけ自動化したい
  - 全てをメタプログラミングにより生成
  - GHDLによる自動シミュレーション

オブジェクト指向の活用



たいしたことはやっていない(設計古い)

数えるとclassは53個あった

eRuby埋め込みソースを処理

パイプライン生成(回路間の接続)もRubyで

float_mul.vhd.erb

全118行

```
% defineIO :in,  :x,   @nfloat
% defineIO :in,  :y,   @nfloat
% defineIO :out, :z,   @nfloat
% defineIO :in,  :clk, 1

% nfloat = @nfloat
% nman = @nman
% nexp = @nexp
%
% width_nf = "#{nfloat-1} downto 0"
% width_nm = "#{nman-1} downto 0"
% width_ne = "#{nexp-1} downto 0"

<%= generate_header %>

<%= generate_entity %>

architecture source of <%= @name %> is

<%= generate_components %>

signal signx, signy, signz, sz : std_logic;
signal manx,  many,  manz,  mz : std_logic_vector(<%= nman %> downto 0);
signal expx,  expy,  expz,  ez : std_logic_vector(<%= width_ne %>);
signal zerox, zeroy, zero, zero_reg1 : std_logic;
....
```

float_mul.vhd : 32 bit の場合

生成されるソース169行

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity fp_mul_32_23_8_4 is
  port (
    x : in std_logic_vector(31 downto 0);
    y : in std_logic_vector(31 downto 0);
    z : out std_logic_vector(31 downto 0);
    clk : in std_logic
  );
end fp_mul_32_23_8_4;

architecture source of fp_mul_32_23_8_4 is
component extract_32_23_8
  port (
    x : in std_logic_vector(31 downto 0);
    s : out std_logic;
    m : out std_logic_vector(23 downto 0);
    e : out std_logic_vector(7 downto 0)
  );
end component;
component rounding_32_23_8
  port (
    m : in std_logic_vector(23 downto 0);
    e : in std_logic_vector(7 downto 0);
....
```

パイプライン記述からの自動生成

- パイプラインを以下のように記述
  - RubyでパースしてVHDL, C言語, APIのソース, Makefileなどを自動生成

```
/VARI  xi, yi, zi, e2;
/VARJ  xj, yj, zj, mj;
/VARF  fx, fy, fz;
dx = xi - xj;
dy = yi - yj;
dz = zi - zj;
r2 = dx*dx + dy*dy + dz*dz + e2;
r1i = powm12(r2);
r3i = r1i*r1i*r1i;
ff = mj*r3i;
fx += dx*ff;
fy += dy*ff;
fz += dz*ff;
```

13行

$$f_i = \sum_{j=1}^{N} \frac{m_j(\vec{x_i} - \vec{x_j})}{(|\vec{x_i} - \vec{x_j}|^2 + \epsilon^2)^{1.5}}$$

計4620行を生成
論理合成が必要

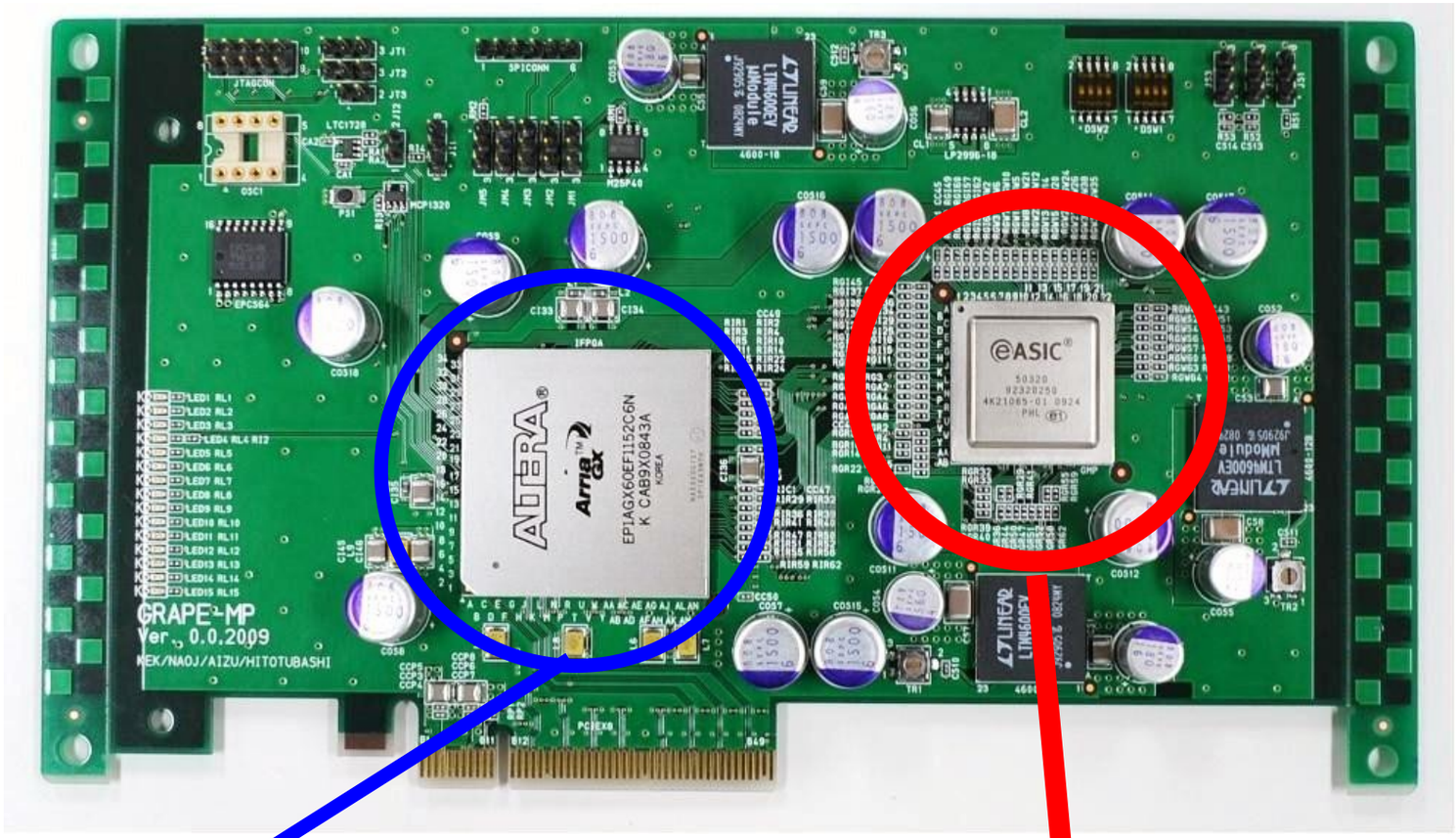
eRubyによるメタプログラミング

- HTMLではなくVHDLの生成に適用
  - ポイントはHTMLと同様に冗長な記述の自動生成とパラメータ化
- 既存のソースを徐々にパラメータ化
- Rubyを使うことで色々と便利にできた
- 精度可変浮動小数点演算回路のライブラリを実現
  - eASICによるGRAPE-MP開発への応用

GRAPE-MPの概要

- 四倍精度演算器をハードウエアで実装
  - 独自形式の128 bit 浮動小数点フォーマット
  - GRAPE-DRアーキテクチャを踏襲：SIMD計算器
    - 省メモリアーキテクチャ: 演算密度の高い演算向け
    - PCI-Expressによりホスト計算機と接続して利用
- Structured ASIC(eASIC)の採用
  - eASICとFPGAは演算粒度は同程度(どちらもLUTを利用)だが、FPGAではLUT間の配線が再構成可能なのに対し、eASICでは配線層が固定されてる。原理的に性能あたりの単価はeASICのほうが安い。一方FPGAではチップの開発コストは必要ない。

GRAPE-MPボード



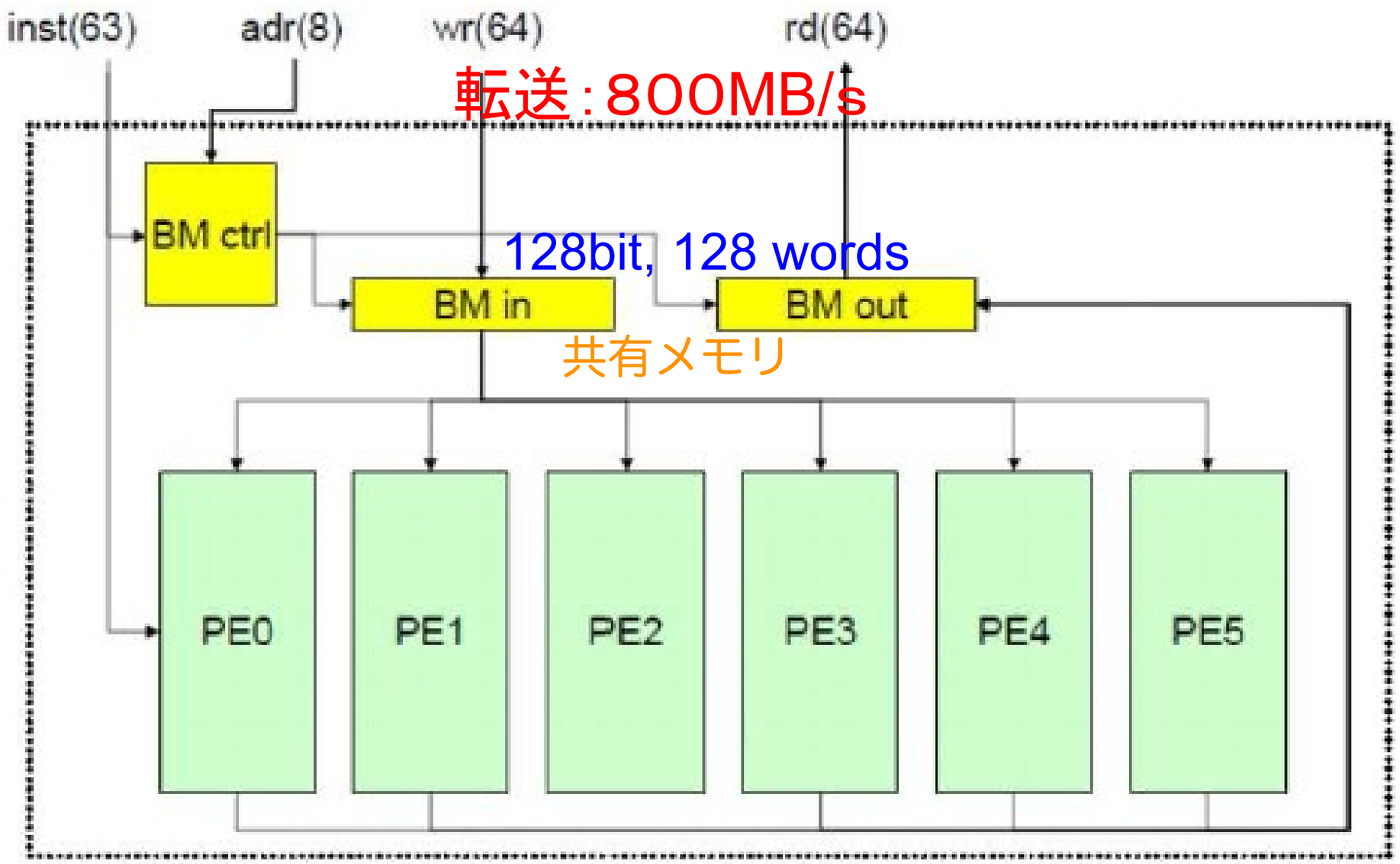
ホスト計算機からFPGAを介してGRAPE-MPを制御する

FPGAにPCI-Express x4 Gen.1を搭載

FPGAのメモリに「プログラム」を保持

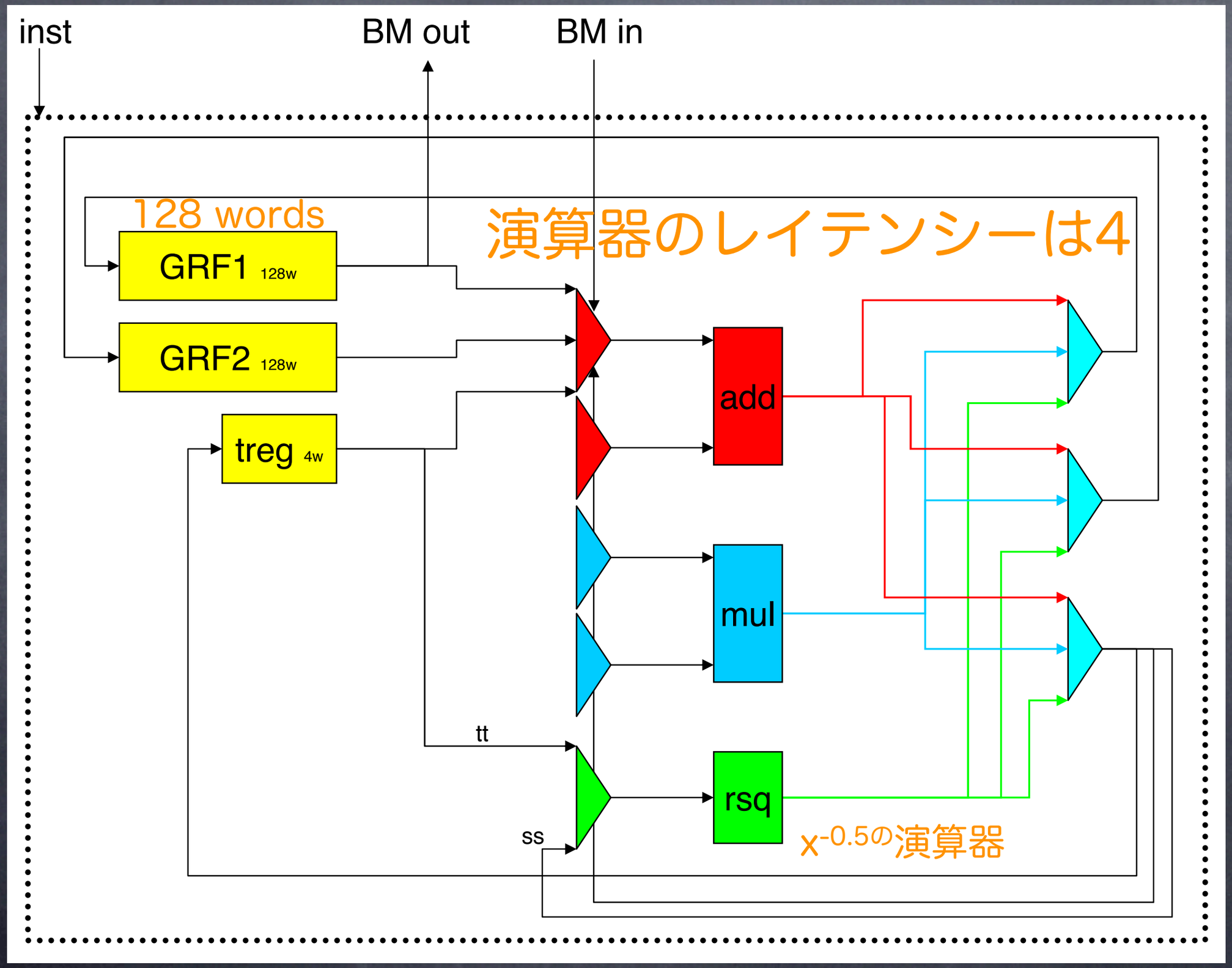
GRAPE-MPチップ

GRAPE-MP チップのブロック図



- 2 演算 x 100MHz x 6 PE = 1.2 Gflops
- 4 ”論理”pipelines x 6 PE = 24 pipelines /chip

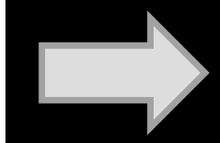
GRAPE-MPの演算ユニット(PE)



GRAPE-MPアセンブラ

- プログラミング用にアセンブラを実装
  - 三つ組で書いたコードをマイクロコードに変換
  - 全ての命令はベクトル長4として扱われる
  - Rubyで実装

```
sub bm16v ra0v rb40v
sub bm20v ra4v rb44v
sub bm24v ra8v rb48v
mul rb40v rb40v ra36v
mul rb44v rb44v tt
add ra36v ts ra32v
mul rb48v rb48v tt
add ra32v ts tt
```

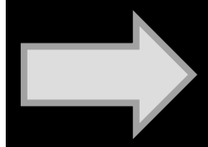


```
1006600214000003        0001000000000110011000000000001000010100000000000000
1106600214800007        0001000100000110011000000000001000010100100000000000
1206600021500000b        0001001000000110011000000000001000010101000000000000
1306600021580000f        0001001100000110011000000000001000010101100000000000
1406600216000013        0001010000000110011000000000001000010110000000000000
1506600216800017        0001010100000110011000000000001000010110100000000000
1606600021700001b        0001011000000110011000000000001000010111000000000000
1706600021780001f        0001011100000110011000000000001000010111100000000000
1806600218000023        0001100000000110011000000000001000011000000000000000
1906600218800027        0001100100000110011000000000001000011000100000000000
1a0660021900002b        0001101000000110011000000000001000011001000000000000
1b0660021980002f        0001101100000110011000000000001000011001100000000000
7a24000245001   00000000000001111010001001000000000000000000001001000000000000
7a24000255201   00000000000001111010001001000000000000000000001001010000000000
7a24000265401   00000000000001111010001001000000000000000000001001100000000000
7a24000275601   00000000000001111010001001000000000000000000001001110000000000
3e24000005801   00000000000000111110001001000000000000000000000000000000000000
3e24040005a01   00000000000000111110001001000000010000000000000000000000000000
3e24080005c01   00000000000000111110001001000000100000000000000000000000000000
3e240c0005e01   00000000000000111110001001000000110000000000000000000000000000
7802000200091   00000000000000111100000000010000000000000000001000000000000000
7802000210095   00000000000000111100000000010000000000000000001000010000000000
7802000220099   00000000000000111100000000010000000000000000001000100000000000
780200023009d   00000000000000111100000000010000000000000000001000110000000000
3e24000006001   00000000000000111110001001000000000000000000000000000000000000
3e24040006201   00000000000000111110001001000000010000000000000000000000000000
3e24080006401   00000000000000111110001001000000100000000000000000000000000000
3e240c0006601   00000000000000111110001001000000110000000000000000000000000000
1e02000000081   00000000000000011110000000100000000000000000000000000000000000
1e02040000085   00000000000000011110000000100000010000000000000000000000000000
```

LSUMPプログラミングシステム

- 総和計算の並列化用DSL
  - GRAPE-MP, GRAPE-DR, GPUなどに対応
  - 単精度、倍精度、四倍精度をサポート

```
VARI xi, yi, zi, e2;
VARJ xj, yj, zj, mj;
VARF ax, ay, az, pt;

dx = xj - xi;
dy = yj - yi;
dz = zj - zi;

r1i = rsqrt(dx**2 + dy**2 + dz**2 + e2);

pf = mj*r1i;
pt += pf;

af = pf*r1i**2;

ax += af*dx;
```



```
bm_in bm12v ra12v pe0
bm_in bm8v ra8v pe0
bm_in bm4v ra4v pe0
bm_in bm0v ra0v pe0
mov zz ra16v
mov zz ra28v
mov zz ra24v
mov zz ra20v
sub bm16v ra0v rb40v
sub bm20v ra4v rb44v
sub bm24v ra8v rb48v
mul rb40v rb40v ra36v
mul rb44v rb44v tt
add ra36v ts ra32v
mul rb48v rb48v tt
add ra32v ts tt
```

GRAPE-MPまとめ

- 四倍精度演算用SIMD型計算機
  - 世界で初めての高精度専用計算機
  - 1ボードあたり1.2 Gflopsの性能
- 総和演算では有効に利用可能
  - 複数ボードでの並列化もスケールする
- 詳しくは下記論文とスライド参照
  - http://dx.doi.org/10.1016/j.procs.2011.04.093
  - http://galaxy.u-aizu.ac.jp/trac/note/attachment/wiki/Talks/MPCOMP_Nakasato.pdf

高速計算とメタプログラミング

- 伝統的な高速計算(HPC) = Fortran
  - この言語自体が数値計算のために設計された
  - 最適化がやりやすい言語仕様
- 現代の高速計算 = 並列計算
  - プログラミングが本質的に困難
  - 自動並列化は絶望的
    - FORTRANのベクトル化だけはうまくいっていたけれど。。。
  - 上から下まで異なる粒度の並列化
    - SSE/AVX, スレッド, CPU-GPU, MPI....
  - 記述量を減らす意味でメタプログラミング重要

Domain Specific Language

- これもメタプログラミング
  - HPCの世界でも近年色々やられはじめている
- LSUMP (LLVM-SUMP)
  - Nakasato & Makino (2009)
  - 粒子法シミュレーションに特化したDSL
  - 元々GRAPE-DR用のコンパイラとして設計
    - GRAPE-DRは粒子シミュレーションに向いた準汎用計算機
  - 後にGPUにも対応。
  - 単、倍、四倍精度演算のカーネルを生成可能

メタプログラミングと言語

- メタプログラミングによる記述量の削減は、並列計算必須のHPCにますます欠かせなくなる
  - 予想：Fortranのような言語はますます廃れる
  - C++の重要性が増す：OOPではなくGPの面で
  - 例1 Eigen (C++行列配列ライブラリ)
    - Template meta programmingでSSE/AVX対応
  - 例2 PETSc (C++ HPC用並列ライブラリ)
    - Template meta programmingで並列処理に対応
    - 演算子オーバーロードを利用したターゲットごとのコード生成

モダンな言語によるHPC

- そのものでHPC計算はまだ無理
- DSLの実装言語としては有力
  - 例1：Liszt (偏微分方程式のためのDSL)
    - Scalaで実装されている
    - DSLからCUDA, MPIを含むC++のコードを生成
  - 例2：Paraiso (Euler的流体スキーム用DSL)
    - Haskellで実装されている
    - DSLからCUDA, MPIを含むC++のコードを生成
    - http://www.paraiso-lang.org/

自動チューニング

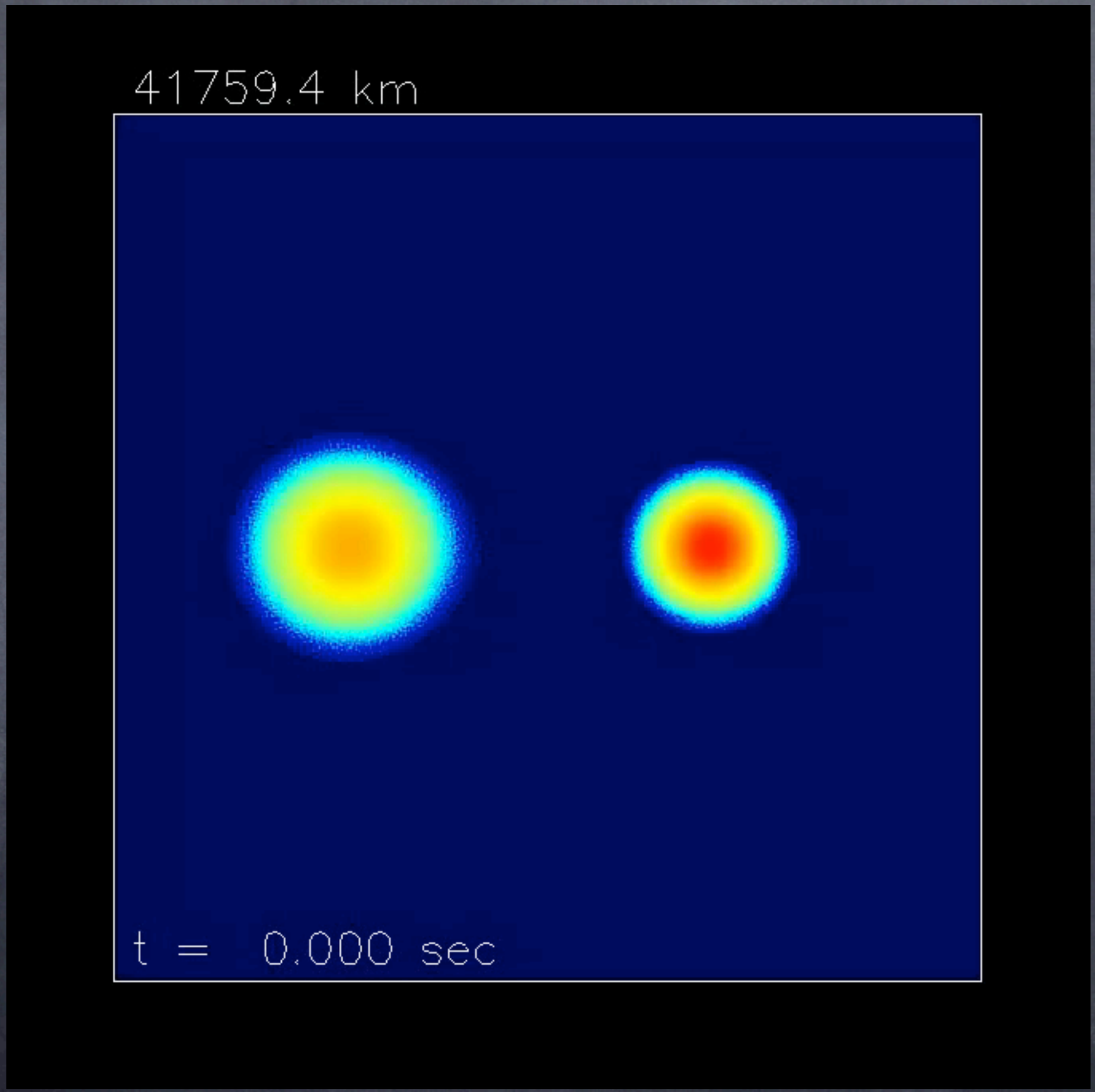
- DSLと対になるのが自動チューニング
  - DGEMMカーネルのRubyによる生成とチューニング

```ruby
if order == 'R' and transa == 'N' and transb == 'N' # RNN
  w << "__local #{vec_type} __a[#{k_width}][#{blk_m}];\n" if use_shared_a
  w << "__local #{vec_type} __b[#{blk_k}][#{blk_n/vec_width}];\n" if use_shared_b
  w << "const int i = get_global_id(1) << #{sblk_mshift};\n"
  w << "const int j = get_global_id(0) << #{shift_hash[sblk_n/vec_width]};\n"
  if use_shared_a
    w << "const int __i = get_local_id(1) << #{sblk_mshift};\n"
    w << "const int __j = get_local_id(0) << #{shift_hash[blk_k/(blk_n/sblk_n*vec_width)]};\n"
  elsif
    w << "const int __i = get_local_id(1) << #{shift_hash[blk_k/(blk_m/sblk_m)]};\n"
    w << "const int __j = get_local_id(0) << #{shift_hash[sblk_n/vec_width]};\n"
  end

  w << "#{vec_type} c[#{sblk_m}][#{sblk_nd}];\n"
  (0...sblk_m).each do |y|
    (0...sblk_nd).each do |x|
      w << "c[#{y}][#{x}] = (#{vec_type})0.0;\n"
    end
  end

  if use_shared_a or use_shared_b
    w << "for (int l = 0; l < k; l += #{blk_k}) {\n"

    if use_shared_a
      (0...sblk_m).each do |y|
        (0...[blk_k/(blk_n/sblk_n*vec_width),1].max).each do |x|
          w << "__a[__j+#{x}][__i+#{y}] = A[(i+#{y})*(lda>>#{vwshift})+((l>>#{vwshift})+__j+#{x})];\n"
        end
      end
    end
    if use_shared_b
      (0...[blk_k/(blk_m/sblk_m),1].max).each do |y|
        (0...sblk_n/vec_width).each do |x|
          w << "__b[__i+#{y}][__j+#{x}] = B[(l+__i+#{y})*(ldb>>#{vwshift})+j+#{x}];\n"
        end
      end
    end
    w << "barrier(CLK_LOCAL_MEM_FENCE);\n"

    w << "for (int p = 0; p < #{blk_k}; p += #{sblk_k}) {\n"

    w << "#{vec_type} a[#{sblk_m}], b[#{sblk_nd}];\n"
    (0...sblk_k).each do |z|
      if z%vec_width == 0
```

gen_dgemm_kernel.rb

松本さん (会津大学D2)

おまけ：白色矮星の合体



まとめ

- ニッチな演算精度の重要性
- 「自作」:ないなら自分でやるしかない
- HDLでの回路設計をRubyで簡単化
  - eRubyによるメタプログラミング
- HPCではメタプログラミングがますます重要になる
  - 自動最適化とフルチューンコードの狭間で
  - Rubyのような動的言語はDSL記述に向く