DIY(handmade) HPC

中里直人(会津大学)&台坂博(一橋大学)

牧野淳一郎(国立天文台)
石川正, 湯浅富久子(KEK)

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ボードによる専用計算
    - 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
    - メニーコア計算機による高性能計算
  - Web http://galaxy.u-aizu.ac.jp/trac/note/
  - Twitter @dadeba

概要

- Do It YourselfなHigh Performance Computingの実践について紹介
  - プロセッサも重要だがネットワークも重要
  - まさにThe network is the computer.

- これまでの経緯
- ネットワークカードのDIY
- プロセッサのDIY
- HPCのためのソフトウエアをDIY

DIY計算機の簡単な歴史(1)

- 最初期の電子計算機 (1940-50年代)
  - いくつかあるがどれもDIYに近い
  - Research Projects (商用は皆無)
  - 最終的に使う人達(主に科学者)が、かなり詳細な仕様まで決定していた
  - このころprogrammingや、computer science/engineeringは存在しなかった
  - 日本での状況は、「新装版計算機屋かく戦えり」遠藤諭(アスキー) に詳しい
    - 計算機開発の心理: 少数精鋭

DIY計算機の簡単な歴史(2)

- 計算機の複雑化 (1960 - 70年代)
  - 計算機の有用性のため、需要が爆発的に増加
    - より速い計算機の需要
      - Computer Engineeringの分化
    - より使いやすい計算機の需要
      - Computer Science, Software Engineeringの分化
  - 商用メーカー間の競争が激化
  - 計算機開発やプログラムの作成が、本当のユーザーの手を離れる
    - どちらも大規模で複雑になりすぎたため
    - DIYの終焉
    - ハッカーの時代 (“Hackers” Steven Levy参照)

DIY計算機の簡単な歴史(3)

- DIY計算機の復権 (1980年代)
  - 科学者達が自分のやりたいこと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: 商用品による初の並列計算機
      - Birth of Hypercube
  - Digital Orrery (1985) (MIT)
    - 太陽系がカオスかどうかを調べるため
  - GRAPE-1 (1989) (東大)
    - 重力多体問題専用計算機
    - 「スーパーコンピュータを20万円で創る」**伊藤智義著 (千葉大教授)参照**
- 一方でDIYマイコンの時代 (e.g. Apple I)

DIY計算機の簡単な歴史(4)

- 最近のDIY計算機(高度化し完全DIYは不可能)
  - QCD用計算機は今でもDIY的
    - QCDOC(コロンビア大/理研) → BlueGene/L
    - QPACE (PowerXCell8iとFPGAの組み合わせ)
  - GRAPEの発展 (東大/理研)
    - GRAPE-1からGRAPE-6, GRAPE-DR
    - FPGAによるGRAPE (PROGRAPE, GRAPE-7)
    - MD-GRAPE-1からMD-GRAPE-3
  - Anton (D.E.Shaw Research)
    - MD専用の計算機：SC2009で話題に
  - Green Flash (LBNL)
    - 気候モデルに特化した計算機 (開発中)

なぜDIY HPCなのか？

- 問題特化による高性能
  - 汎用プロセッサには無駄が多い
    - シリコンのうち演算器の占める割合が低い
- 問題特化による高効率
  - Cost, energy performanceに優れる
  - 低予算でやりたい仕事ができる

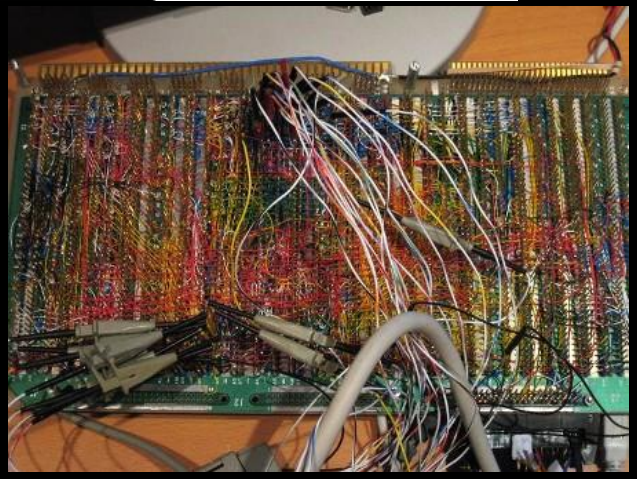
ただし常に汎用 vs. 専用の競争がある

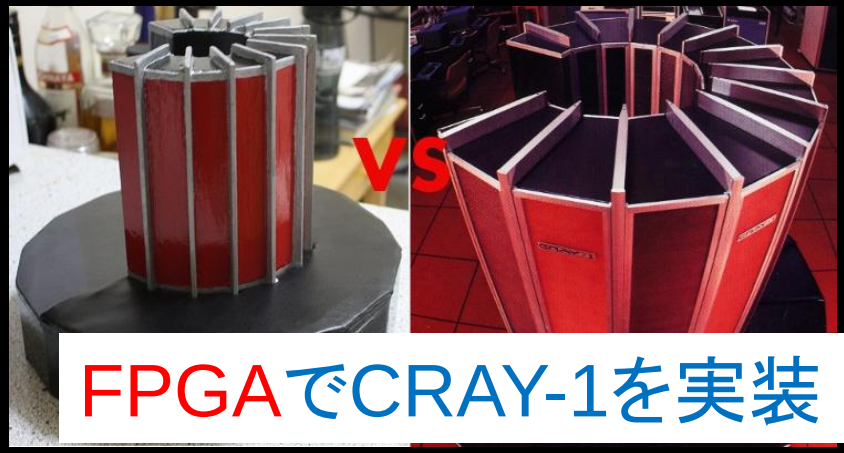
(牧本ウェーブ)

- そこに「計算機」があるから
  - 計算機を一から作ってみたいという願望

趣味としてのDIY計算機

CPUとOSを自作





FPGAでCRAY-1を実装

http://www.stevechamberlin.com/cpu/category/bmow1/

http://chrisfenton.com/homebrew-cray-1a/

DIY HPC計算機開発の実践

- 並列計算機で最もコストがかかるのはインターコネクト！
  - お金も電気もかかり、性能にクリティカル
  - インターコネクト用ネットワークカードの開発
- 高精度な計算がしたい！
  - 汎用計算機は倍精度(64bit)まで
  - 四倍精度専用プロセッサの開発
- GPUの性能を極限まで引き出す！
  - 問題特化型コンパイラの開発
  - ソフトウエアのDIY

四倍精度演算専用プロセッサの開発と応用

中里直人(会津大学)
台坂博(一橋大学)
牧野淳一郎(国立天文台)
石川正, 湯浅富久子(KEK)

概要

- 四倍精度演算とは?
- 四倍精度演算プロセッサのアーキテクチャ
- GRAPE-MPチップの開発
- GRAPE-MPシステムについて
- 今後の応用例

数値計算の現状

- 数値計算では倍精度が利用されている
  - 仮数部 53 bit, 指数部 11 bit, 符号部 1bit
  - 「たいてい」は倍精度で十分
    - 多くのアルゴリズムは倍精度であれば安定
  - プログラムが容易
  - 倍精度まではハードウエア実装
    - 半導体プロセスの進化にしたがって高速化してきた
    - 最新のCPU ~ 96 Gflop/s (12 cores)
    - 最新のGPU ~ 544 Gflop/s (320 cores)
  - ただし単精度以下でも十分な問題はある
    - 例：重力多体問題 (GRAPEの成功の理由)

高精度演算の必要性

- 倍精度では十分ではない問題
  - 条件数が非常に大きい(>10^{16})行列
  - メッシュを再帰的に分割するAMR
    - 分割数が50以上となると倍精度では不足
  - ファインマンループの数値積分
    - 二重指数関数型積分公式
    - ε算法
  - 精度の足りない例：~1.1726@倍精度

$$a = 77617.0$$

$$b = 33096.0$$

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

高精度演算の問題点

- 遅い！
  - 整数演算/浮動小数点エミュレーションどちらもCPUでは遅い。
    - マイクロベンチマーク

	加算秒数	乗算秒数	加算 lat.	乗算 lat.	加算性能	乗算性能	clock	MA
Core2	4.545	5.829	40.6	52.2	59	46	2.4	Core MA
Xeon	4.502	6.002	39.0	52.2	60	45	2.33	Core MA
Core i7	3.617	4.621	36.0	46.0	73	58	2.67	Nehalem
Opteron	3.830	4.049	31.3	33.1	70	66	2.2	K8

表 2　DD 加算と乗算の各種 x86 アーキテクチャCPU における演算性能: 性能のコラムの単位は MFLOPS であり, clock 周波数の単位は GHz である. 最後のコラムはマイクロアーキテクチャを示す.

    - Mpackによるベンチマーク (中田 2010)
      - DGEMM by DD(32桁)　　~ 141 Mflop/s = ~ 1.3%
      - DGEMM by GMP (308桁) ~ 14.4 Mflop/s

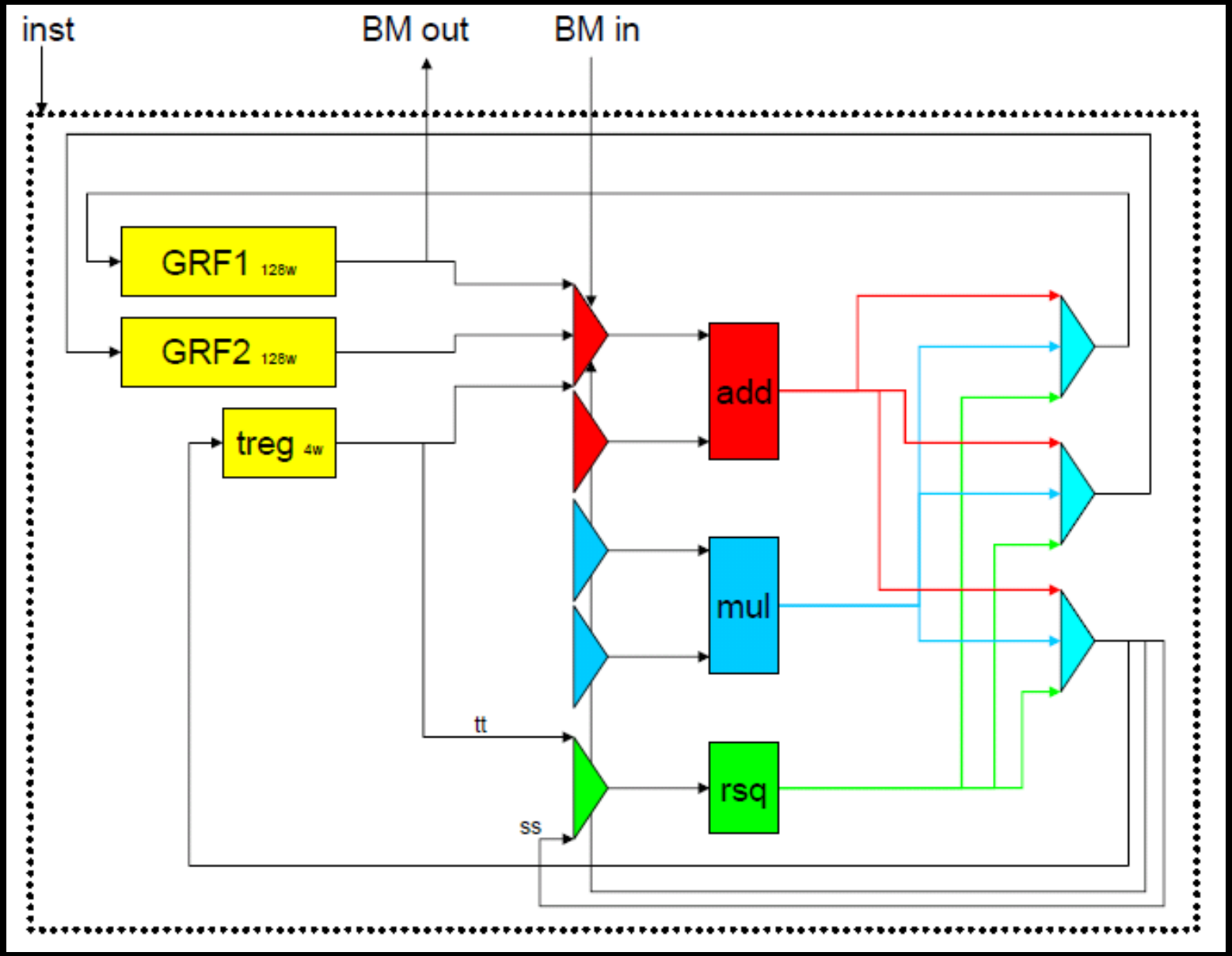
GRAPE-MPとは

- GRAPE-DRのアーキテクチャにもとづき、演算器を四倍精度とした専用プロセッサ
  - GRAPE-DRとは
    - HPC専用に拡張されたプログラマブルなGRAPE
    - SIMD型プロセッサ (512 PE)
    - 省メモリ帯域設計
    - 縮約専用回路
    - DGEMMや重力計算に向いた設計
  - GRAPE-MP
    - 128 bit浮動小数点演算に特化したプログラマブルSIMD型プロセッサ
    - 縮約回路なし。外部メモリなし。

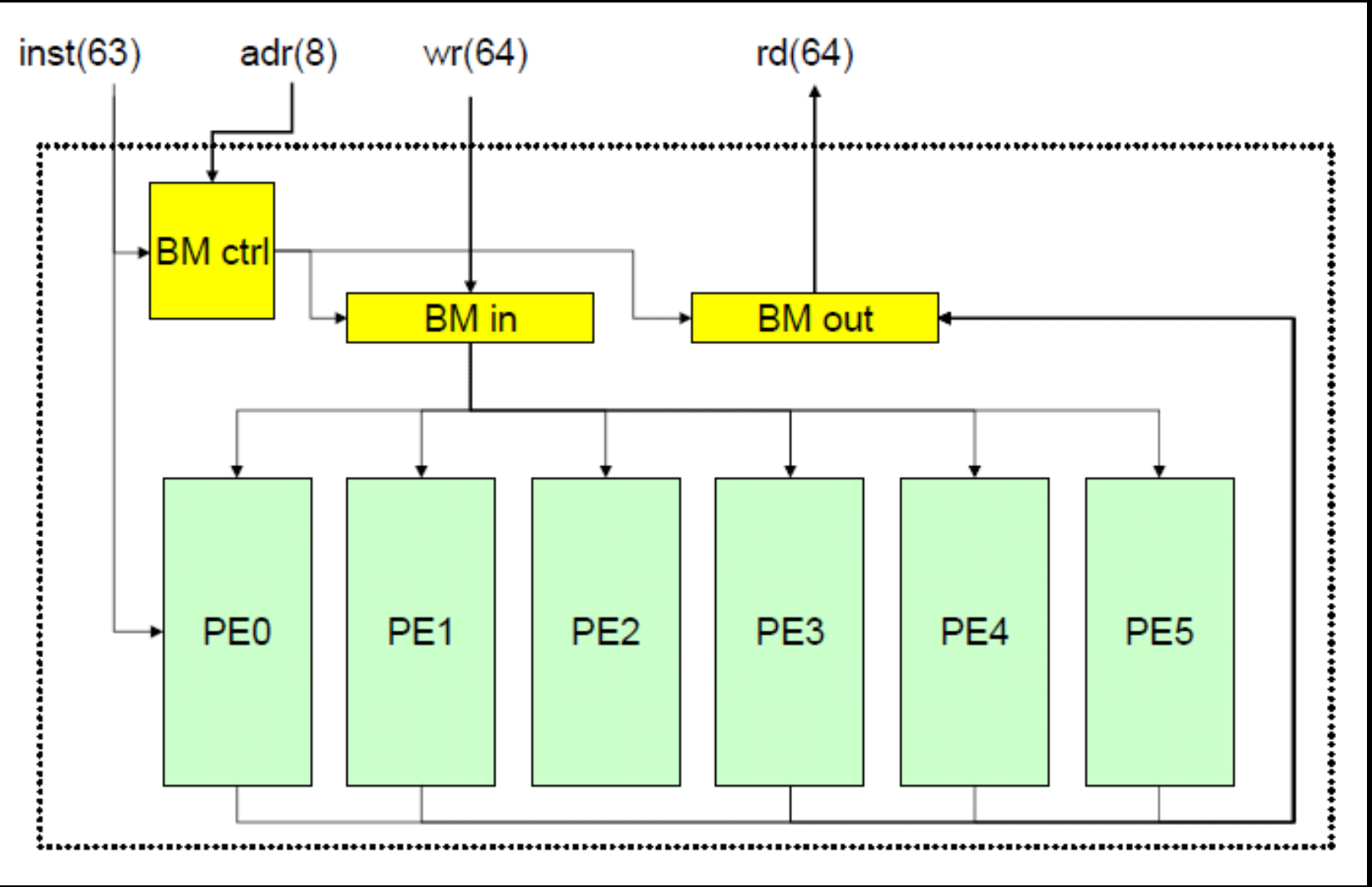
GRAPE-MPができるまで(1)

- 2008年8月
  - eASICでのプロセッサ開発の検討 (牧野)
- 2008年10月
  - 三つの設計案を検討 (中里, 牧野)
    - GRAPE-DRを拡張 (捨てられているビットを保持する)
      - DDエミュレーションの高速化
    - 整数のALUをたくさん並べる
      - 整数エミュレーション専用プロセッサ
    - 128bitの演算器のはいったGRAPE-DR
      - 四倍精度専用プロセッサ
  - 第三案を採用 (12月頃に決定)

PEブロック図



chipブロック図


inst(63)
adr(8)
wr(64)
rd(64)
BM ctrl
BM in
BM out
PE0
PE1
PE2
PE3
PE4
PE5

GRAPE-MPができるまで(2)

- 2008年12月 – 2009年1月
  - eASICでデバイスを作ることを決定
  - 設計などにとりかかる (中里)
    - C言語でのエミュレーションプログラムの作成
      - 問題:128bit整数演算をどうするか？ (GMPを利用)
    - HDLでの演算回路およびプロセッサの設計・テスト
- 2009年1月 – 4月
  - 1月17日「GRAPE-MP」という名前がつく
  - 引き続きアーキテクチャと回路設計 (中里)
    - 他, システムソフトウエアの作成 (アセンブラ等)
  - 設計担当会社との打ち合わせ (中里,牧野)
  - 4月28日 Final Design Review

四倍精度演算器の設計 (1)

- 我々が開発したFP回路生成フレームワーク
  - 指数部, 仮数部等を指定してFP演算器を生成
    - CORE Generator/Mega Functionと同等
      - ただしこれらは必要最低限のflexibility
      - 128bitのFP演算器などは生成できない
    - 自前で設計することで、すべてflexible！
    - ただし、FP乗算器は一部再設計した
      - bit数が大きいため
  - メタプログラミングはやっぱり重要
    - RubyによりCのソースとVHDLを生成
      - Cのソースは FP演算のエミュレーションコード
    - GRAPE的なパイプライン生成も含む

四倍精度演算器の設計 (2)

- 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での繰り返し処理等を覚えるのが大変

- HTMLの面倒なところを改善するのに
  - HTML用のテンプレートエンジン
    - Formの繰り返し記述をソースから生成
    - VHDLの繰り返し記述に応用できる？

四倍精度演算器の設計 (3)

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

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(8 downto 0)&"00000000000000000" when "00010010", -- 18
         i(7 downto 0)&"00000000000000000" when "00010011", -- 19
         i(6 downto 0)&"00000000000000000" when "00010100", -- 20
         i(5 downto 0)&"00000000000000000" when "00010101", -- 21
         i(4 downto 0)&"00000000000000000" when "00010110", -- 22
         i(3 downto 0)&"00000000000000000" when "00010111", -- 23
         i(2 downto 0)&"00000000000000000" when "00011000", -- 24
         i(1 downto 0)&"00000000000000000" when "00011001", -- 25
         i(0)&"00000000000000000" when others; -- 26
end source;
```

四倍精度演算器の設計 (4)

```
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)

GRAPE-MPができるまで(3)

- 2009年4月-
  - 4月30日テープアウト
    - イノテック → eASIC
  - 7月下旬 チップが届く
  - 7月 – 12月
    - GRAPE-MPボードの設計 (台坂)
    - 9月- FPGA回路の設計開始 (台坂, 中里)
  - アセンブラ, コンパイラの整備など (中里)
- 20010年4月 -
  - GRAPE-MPボードが届く
  - FPGA回路の設計 (台坂)

GRAPE-MPボード

- FPGAとGRAPE-MPチップからなる

GRAPE-MPアーキテクチャ

- PEが6個からなるSIMDプロセッサ
  - 4 cycleのSIMTベクトル処理
  - 実効ベクトル長は24
- PEは四倍精度演算器を持つ
  - 仮数部 116 bit 指数部 11 bit の演算器
  - 逆数平方根の初期値回路 (26 bit精度)
    - フル精度の除算と平方根はニュートン法でソフトウエアにより処理。3回の繰り返し
  - レジスタは256語
  - Forwarding pathあり
- PEはブロードキャストメモリ(BM)と接続

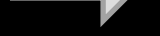
GRAPE-MPのソフトウエア(1)

- C/C++言語によるシミュレータ
  - 演算器レベル
    - GMPによる128bit整数演算
  - PEレベル
    - 演算命令やレジスタのシミュレーション
  - チップレベル (BM等を含めたシミュレーション)
    - アセンブラが生成した命令を読み込んで計算
    - デバッグ命令
    - 変数変換のライブラリ
      - QDとGRAPE-MPフォーマットの変換しょりなど

GRAPE-MPのソフトウエア(2)

- アセンブラ
  - ソースコードを命令(63bit)に変換
  - 読み書きの衝突は検出
  - 他のエラー処理は今はなし

```
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
```

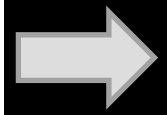


```
1006600214000003        0001000000000110011000000000001000010100
1106600214800007        0001000100000110011000000000001000010100
120660021500000b        0001001000000110011000000000001000010101
130660021580000f        0001001100000110011000000000001000010101
1406600216000013        0001010000000110011000000000001000010110
1506600216800017        0001010100000110011000000000001000010110
160660021700001b        0001011000000110011000000000001000010111
170660021780001f        0001011100000110011000000000001000010111
1806600218000023        0001100000000110011000000000001000011000
1906600218800027        0001100100000110011000000000001000011000
1a0660021900002b        0001101000000110011000000000001000011001
1b0660021980002f        0001101100000110011000000000001000011001
7a24000245001   000000000000011110100010010000000000000000001001000
7a24000255201   000000000000011110100010010000000000000000001001010
7a24000265401   000000000000011110100010010000000000000000001001100
7a24000275601   000000000000011110100010010000000000000000001001110
3e24000005801   000000000000011111000100100000000000000000000000000
3e24040005a01   000000000000011111000100100000010000000000000000000
3e24080005c01   000000000000011111000100100000100000000000000000000
3e240c0005e01   000000000000011111000100100000110000000000000000000
7802000200091   000000000000011110000000010000000000000001000000000
7802000210095   000000000000011110000000010000000000000001000010000
7802000220099   000000000000011110000000010000000000000001000100000
780200023009d   000000000000011110000000010000000000000001000110000
3e24000006001   000000000000011111000100100000000000000000000000000
3e24040006201   000000000000011111000100100000010000000000000000000
3e24080006401   000000000000011111000100100000100000000000000000000
3e240c0006601   000000000000011111000100100000110000000000000000000
1e02000000081   000000000000011111000000010000000000000000000000000
1e02040000085   000000000000011111000000010000010000000000000000000
```

GRAPE-MPのソフトウエア(3)

- コンパイラ(18行 → 63行のアセンブリ言語)

```
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;
ay += af*dy;
az += af*dz;
```



```
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
add ts ra12v tt
mul ts bm126 ra120v
rsq tt tt
 ....
```

GRAPE-MPの応用

- 現在までに実機で以下のカーネルが動作
  - ファイマンループ積分

$$
\begin{aligned}
I &= \int_0^1 dx \int_0^{1-x} dy \int_0^{1-x-y} dz F(x,y,z), \\
F(x,y,z) &= D(x,y,z)^{-2} \\
D &= -xys - tz(1-x-y-z) + (x+y)\lambda^2 \\
&\quad +(1-x-y-z)(1-x-y)m_e^2 \\
&\quad +z(1-x-y)m_f^2. \qquad (2)
\end{aligned}
$$

  - 重力とポテンシャルの計算

$$
\boldsymbol{f}_i = \sum_{j=1}^{N} \frac{m_j(\boldsymbol{x}_i - \boldsymbol{x}_j)}{(|\boldsymbol{x}_i - \boldsymbol{x}_j|^2 + \epsilon^2)^{3/2}},
$$

GRAPE-MPの開発とは

- 計算機を(ほぼ)full scratchで作る体験
  - アーキテクチャ定義
  - 回路の論理設計
  - シミュレーションコードの作成
  - アセンブラの設計と実装
  - システムソフトウエアの実装
  - コンパイラの実装
- 私がやってないこと
  - チップの物理設計, チップの作成
  - ボードの設計や制御回路の設計 (台坂さん)

究極のメニーコアアクセラレータプログラミング

会津大学　中里直人

アクセラレータのプログラミング(1)

- 明示的なベクトル・並列処理が必要
  - 拡張されたC言語 (C for CUDA, OpenCL)
  - アセンブリ言語 (IL, PTX, DRアセンブリ)
- 最適な利用方法は経験的に得る必要あり
  - データ構造の最適化：SoA or AoS
  - データ移動の最適化
  - メモリ階層の利用
  - 様々な制限の回避

アクセラレータのプログラミング(2)

- C for CUDA (NVIDIA)
  - 拡張されたC言語によりプログラミング
  - 並列計算する部分を特殊な関数として定義
  - 高パフォーマンスを得るには
    - 共有メモリをキャッシュとして利用する必要あり
- OpenCL (NVIDIA, AMD, Apple)
  - GPU, CPU, DSPなどを統一してプログラム可能
  - プログラミングモデルはC for CUDAと同等
  - 問題点
    - 異なるアーキテクチャを唯一の抽象化で扱える？

アクセラレータのプログラミング(3)

- DIYした専用プロセッサの場合
  - 開発者(研究者)がソフトも自前で用意する
    - たいていは予算の問題で外注できない
    - ある意味自由だが、人的資源は必要
  - GRAPE-DRの場合
    - システムソフトウエア (小池君,牧野さんら)
    - アセンブラ (牧野さん)
    - コンパイラ (中里)
    - アプリケーション (利用者がそれぞれ工夫)
  - 問題特化コンパイラの自作はそれほど難しくない
    - オープンソース(LLVMなど)のおかげ

GPUの性能を極限まで引き出す

- アセンブリ言語での直接プログラミング
  - 拡張C言語で書く場合も、アーキテクチャを意識してコードを書かないと性能は出ない
    - まどろっこしい
    - ただしアセンブリ言語では「面倒」も多い
    - GPUに限らずCPUでも同様
- 「面倒」を避ける一つの方法
  - メタプログラミング
    - プログラムでプログラムすること
    - LISP(Scheme)のマクロが有名

ILによるプログラム例

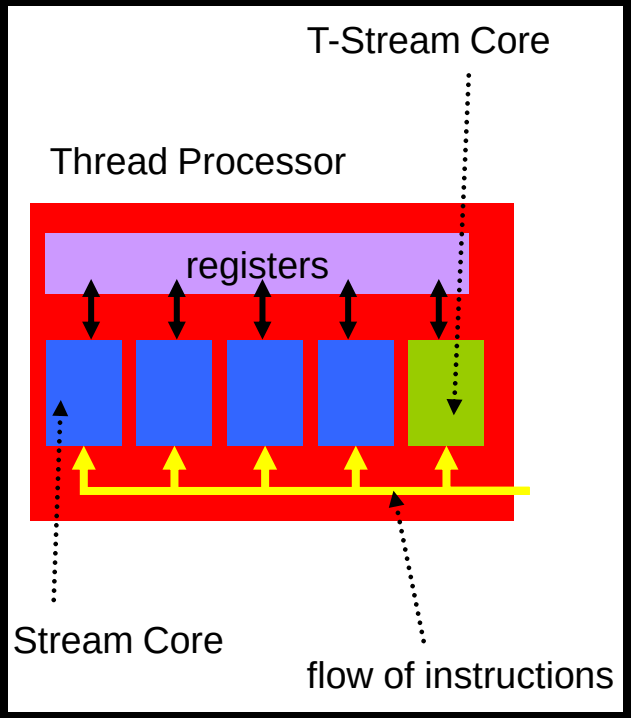
```
il_ps_2_0
dcl_input_interp(linear) v0.xy
dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)
dcl_output_generic o0

dcl_cb cb10[1]
dcl_literal l9, 3.0, 3.0, 3.0, 3.0
dcl_literal l10, 0.0, 0.0, 0.0, 0.0
dcl_literal l11, 0.0, 0.0, 1.0, 0x1
dcl_literal l12, 0.0, 0.0, 1.0, 128.0

mov r400, l10
mov r200.xy, l11.xy
mov r201.w, cb10[0].w
mov r201.z, cb10[0].z
ixor r201.x, r201.x, r201.x

sample_resource(0)_sampler(0) r100, v0.xy
whileloop
  sample_resource(0)_sampler(4) r300, r200.xy
  sub r17.xyz, r300.xyz, r100.xyz
  dp3 r6, r17, r17
  add r6, r6, l9
  rsq r7, r6
  mul r8, r7, r7
  mul r8, r8, r7
  mul r9, r8, r300.w
  mad r400, r9, r17, r400
  iadd r201.x, r201.x, l11.w
  ige r201.y, r201.x, r201.w
  break_logicalnz r201.y
  add r200.x, r200.x, l11.z
  eq r202.x, r200.x, l12.w
  if_logicalnz r202.x
    add r200.0y, r200.0y, l11.z
  endif
endloop
mov o0, r400
ret_dyn
endmain
end
```

$$\boldsymbol{f}_i = \sum_{j=1}^{N} \frac{m_j(\boldsymbol{x}_i - \boldsymbol{x}_j)}{(|\boldsymbol{x}_i - \boldsymbol{x}_j|^2 + \epsilon^2)^{3/2}},$$

N体計算の最適化(1)

- VLIW命令の解析
  - CAL APIによりVLIW命令を取得可能
  - VLIWのスロットが多く埋まるほど性能が高い

slot	命令数	割合(%)
1	5	26
2	5	26
3	2	11
4	4	21
5	3	16
計	**19**	

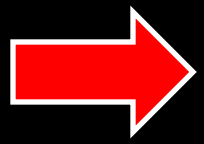


  - おおよそ50%は効率が2割以下

N体計算の最適化(2)

- 最適化結果

slot	命令数	割合(%)
1	5	26
2	5	26
3	2	11
4	4	21
5	3	16
計	**19**	



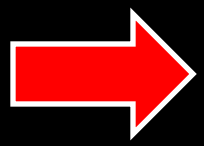
slot	命令数	割合(%)
1	12	15
2	6	7
3	4	5
4	16	20
5	43	53
計	**81**	

  - ベクトル化とループアンローリング
  - VLIW命令の実行効率が大幅に増加
  - 演算性能は約4倍(~ 800 Gflops)

N体計算の最適化(3)

- ループ制御構造の最適化

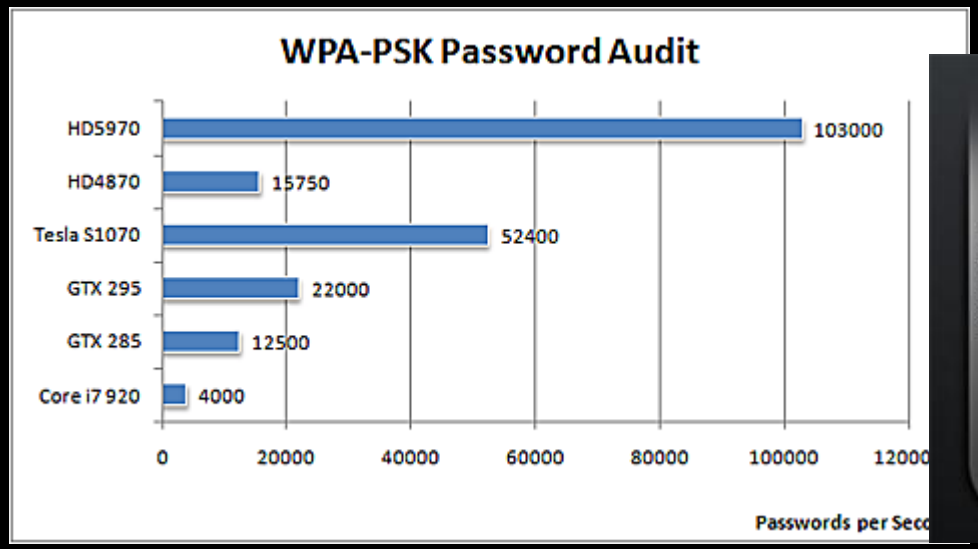
slot	命令数	割合(%)
1	12	15
2	6	7
3	4	5
4	16	20
5	43	53
計	**81**	

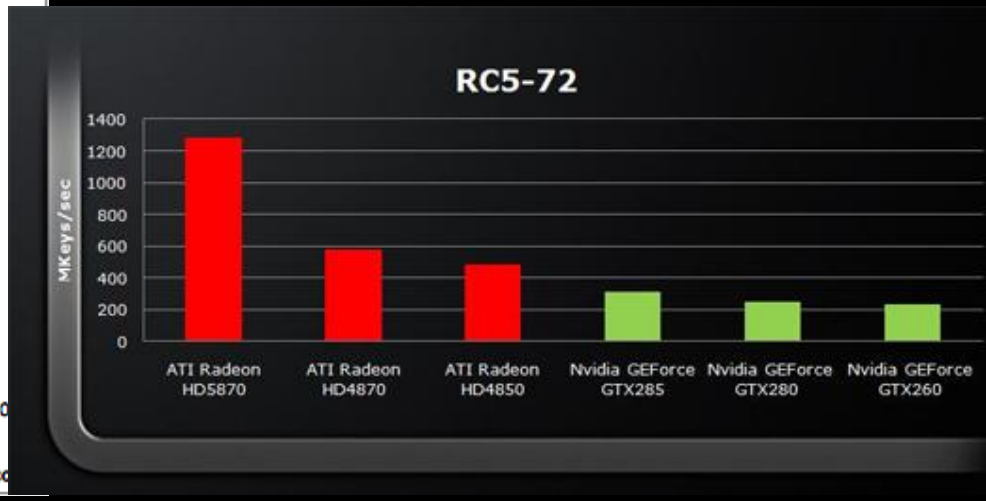


slot	命令数	割合(%)
1	1	2
2	2	3
3	4	6
4	16	24
5	43	65
計	**66**	

  - 命令数の削減と効率の更なる向上
  - さらに20%ほど性能が向上

整数演算の例





PRNG	$\times 10^9$ per second			$\times 10^7$ per second		Speed-up factor
	5850	4870	4850	CPU	CPU_2	
GGL	8.37	5.05	4.21	1.23	0.80	681
XOR128	8.45	6.29	4.52	1.86	1.20	455
RANECU	4.98	3.32	2.66	1.21	0.79	411
RANLUX3P	1.08	1.02	0.63	0.50	0.33	216
RANLUX4P	1.02	0.86	0.58	0.32	0.21	322
MT19937	0.50	0.62	0.36	1.07	0.69	47
RANMAR	0.18	0.23	0.14	1.69	1.10	11

命令セット
整数演算もなんでも可能
演算器は32bit
理論性能 ~ 1.35 TIPS

アクセラレータのプログラミング

- 困難なのは並列プログラミングだから
  - しかも複数レベルの「並列プログラミング」が必要
    - 演算器 (SIMD 並列度 4)
    - プロセッサ (SIMT 並列度 32-64)
    - CPUとGPUの並列 (非同期プログラミング)
    - GPUクラスターの並列

- さらにI/Oの問題が性能の大きな制約
  - GPU chip ~ 150 -200 GB/sec
  - CPUとGPU ~ 16 GB/sec (PCIex16 gen2)
  - GPUクラスター: レイテンシと帯域

提案手法

- ユーザーは以下をDSLで記述する
  - 並列計算する部分
  - 入力変数の性質の指定

- 提案コンパイラは、指定された変数の性質に基づいてアクセラレータ用コードを生成する
  - 経験的な最適計算手法の適用
    - これは問題に依存する：今回は総和演算
    - さらにアクセラレータにも依存する

Usage Model (1)

- Original source code of particle simulations

```
… initialization …
while(t <= t_end) {
   … predict …
  for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
      f[i] += force(x[i], x[j]);
    }
  }
  … update …
  t = t + dt;
}
… finalization …
```

並列計算する力の
総和演算の部分

Usage Model (2)

- ユーザーは以下のようなソースを記述

```
LMEM xi, yi, zi, e2;
BMEM xj, yj, zj, mj;
RMEM ax, ay, az;

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

r1i = rsqrt(dx**2 + dy**2 + dz**2 + e2);
af = mj*r1i**3;

ax += af*dx;
ay += af*dy;
az += af*dz;
```

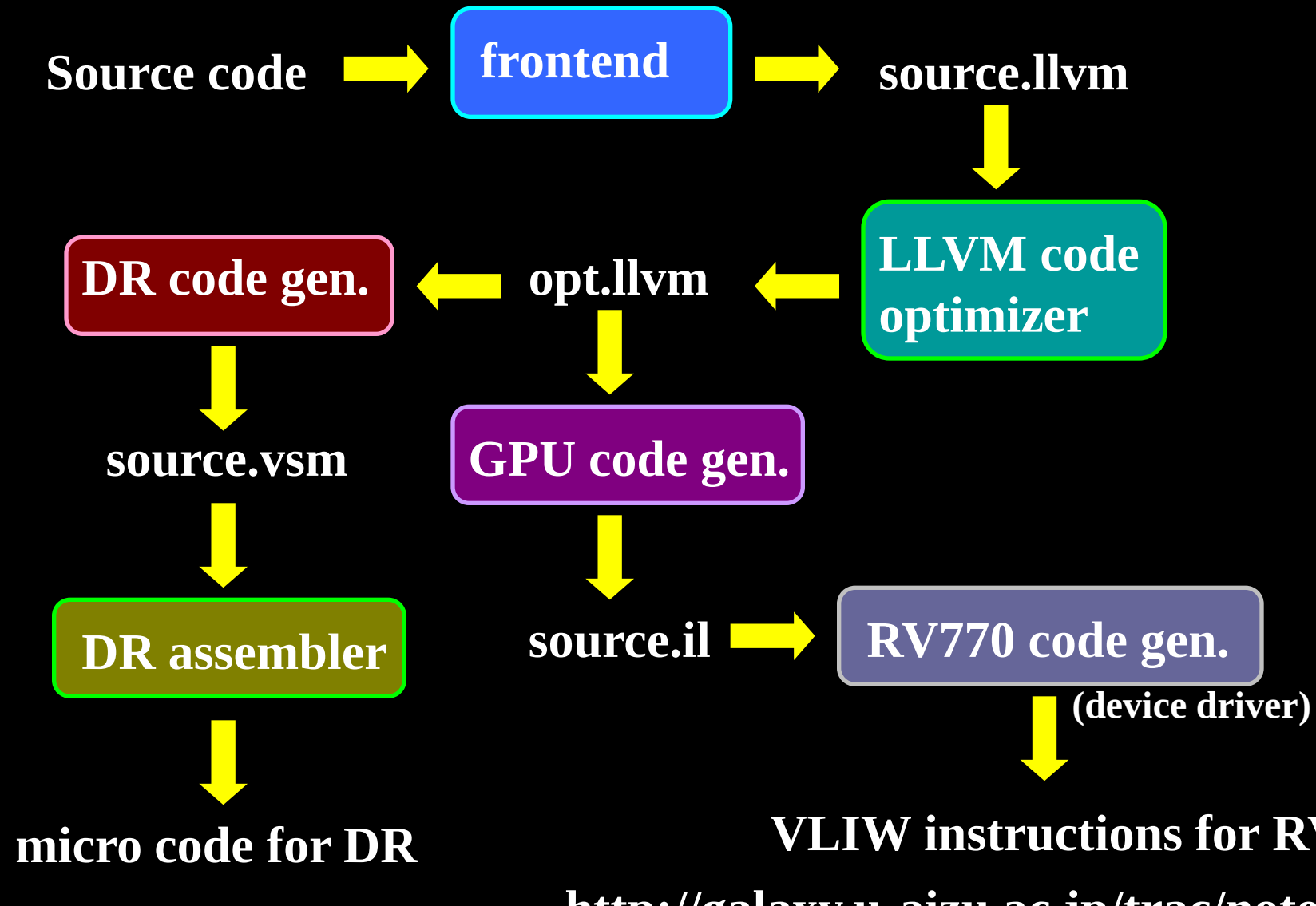
Usage Model (3)

- 本システムはデータ転送やアクセラレータ管理用APIをライブラリとして生成

```
… initialization…
while(t <= t_end) {
   … predict ..
  send_data(n, x);
  execute_kernel(n);
  receive_data(n, f);
  … update …
  t = t + dt;
}
… finalization …
```

ユーザーは二重ループによる総和演算を生成されたAPIの呼び出しに置き換える

Compiler Flow



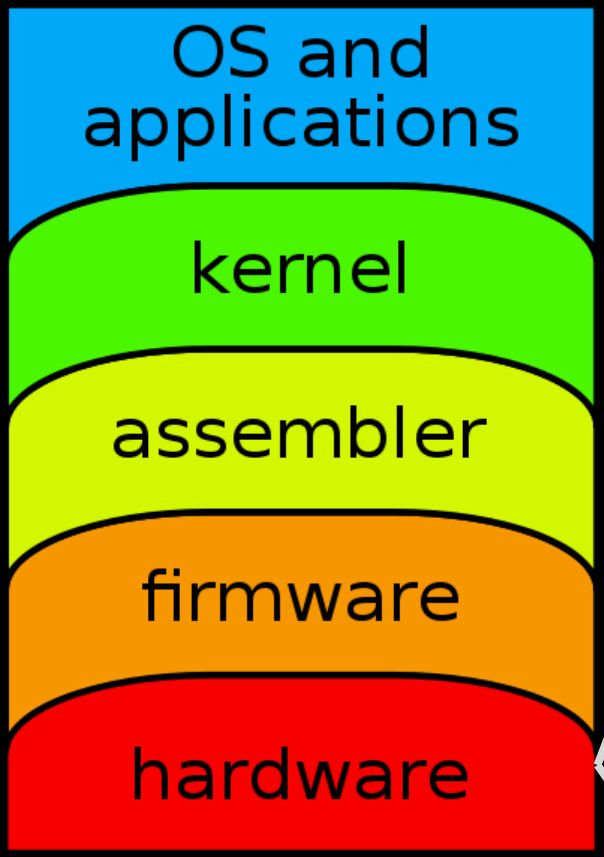
http://galaxy.u-aizu.ac.jp/trac/note/

まとめ

- なんでもDIYするのは時間の無駄?
  - そうでもないですよ
    - 明確な目標があり、そこたどりつく最短ルート
      - 実験物理屋の発想
    - 他にやる人がいないなら自分でやるしかない
    - それが楽しいならばなおよい
- 分業思考に陥ってませんか？
  - 分業で効率を求めたからといって、高パフォーマンスになるかどうかは自明ではない
  - 垂直統合 vs. 水平統合
    - 日本の家電メーカー vs. アメリカのCE業界
    - iPhone vs. Android

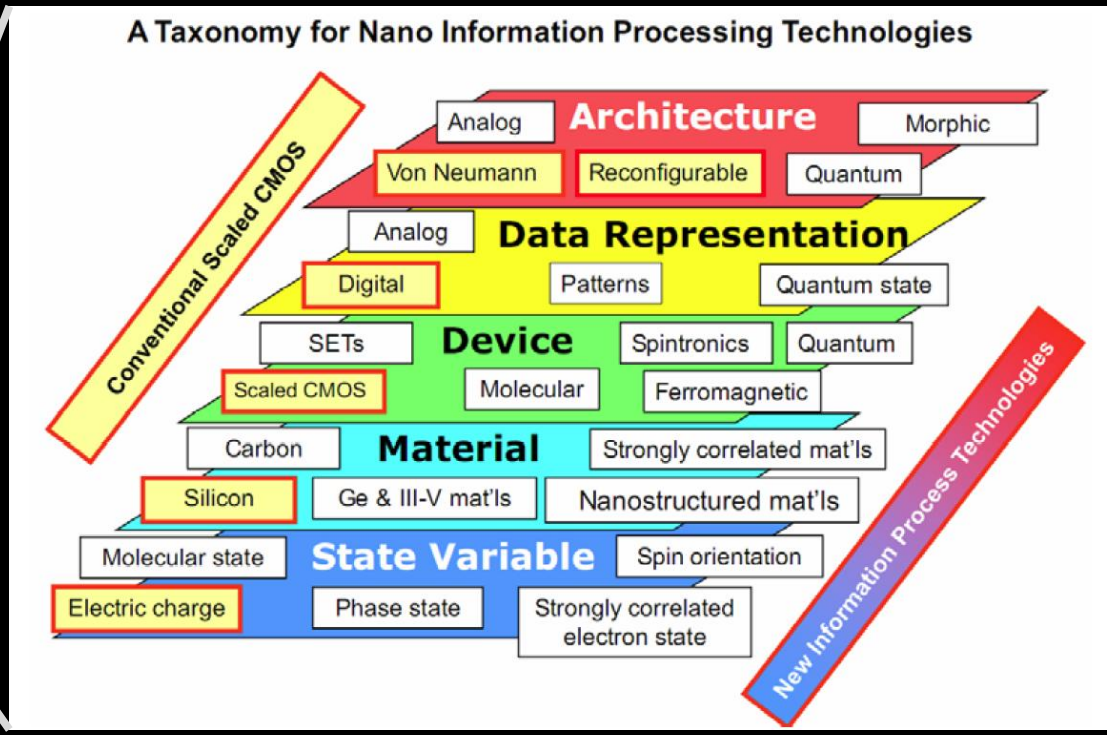
分業(レイヤー)思考を突き抜ける

計算機の抽象レイヤー構造



我々がDIYしている部分

計算デバイスのレイヤー構造



DIY精神の成功例

- “Parallel Computing Works“ Fox, Williams, Messina 1994, 20章より
  - Caltech Cosmic Cube(CCC)を作ったFoxらによる、並列計算プロジェクトC3Pについてまとめた大著
- 逸話：CCCの実働者であったOttoは、QCDの物理を理解していただけでなく、その定式化、そしてCCCの実現に必要な計算機アーキテクチャやハードウエアも理解し、CCCを実現した --- 計算科学者の誕生

- 詳しくは私訳(http://galaxy.u-aizu.ac.jp/trac/note/blog/)を参照してください