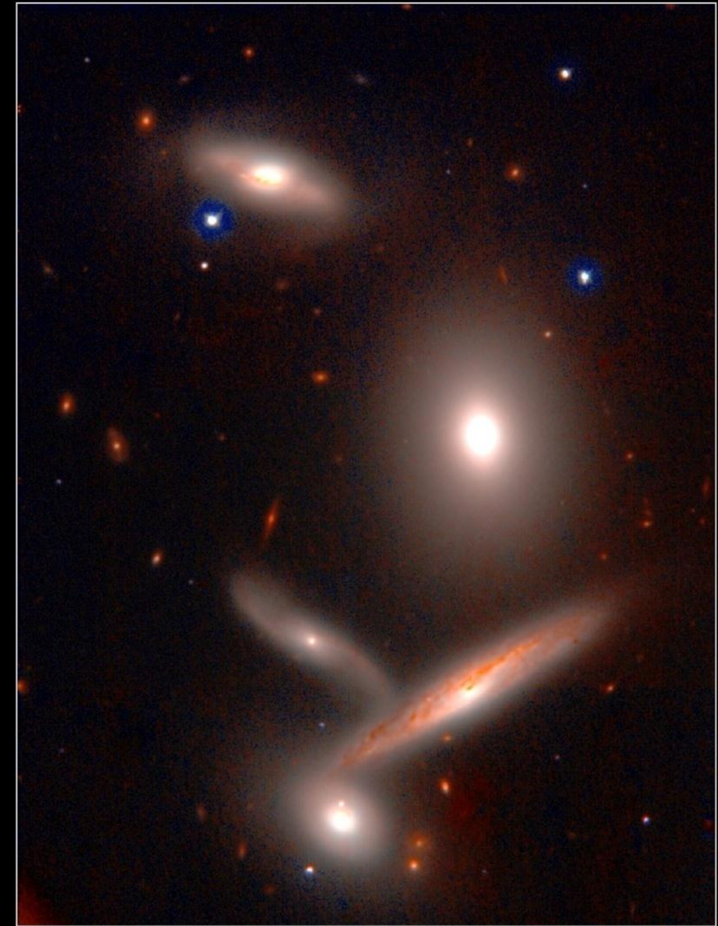
演算精度に応じた高性能計算を実現するコンパイラの提案と実装

会津大学　中里直人

概要

- 問題設定
- アクセラレータの紹介
- 問題特化型のコンパイラ
- 性能評価
  - GRAPE-DRでの性能評価
  - RV770での性能評価
  - 他の応用例
- 発展のアイデア

Grand Challenge problems



Hickson Compact Group 40
Subaru Telescope, National Astronomical Observatory of Japan

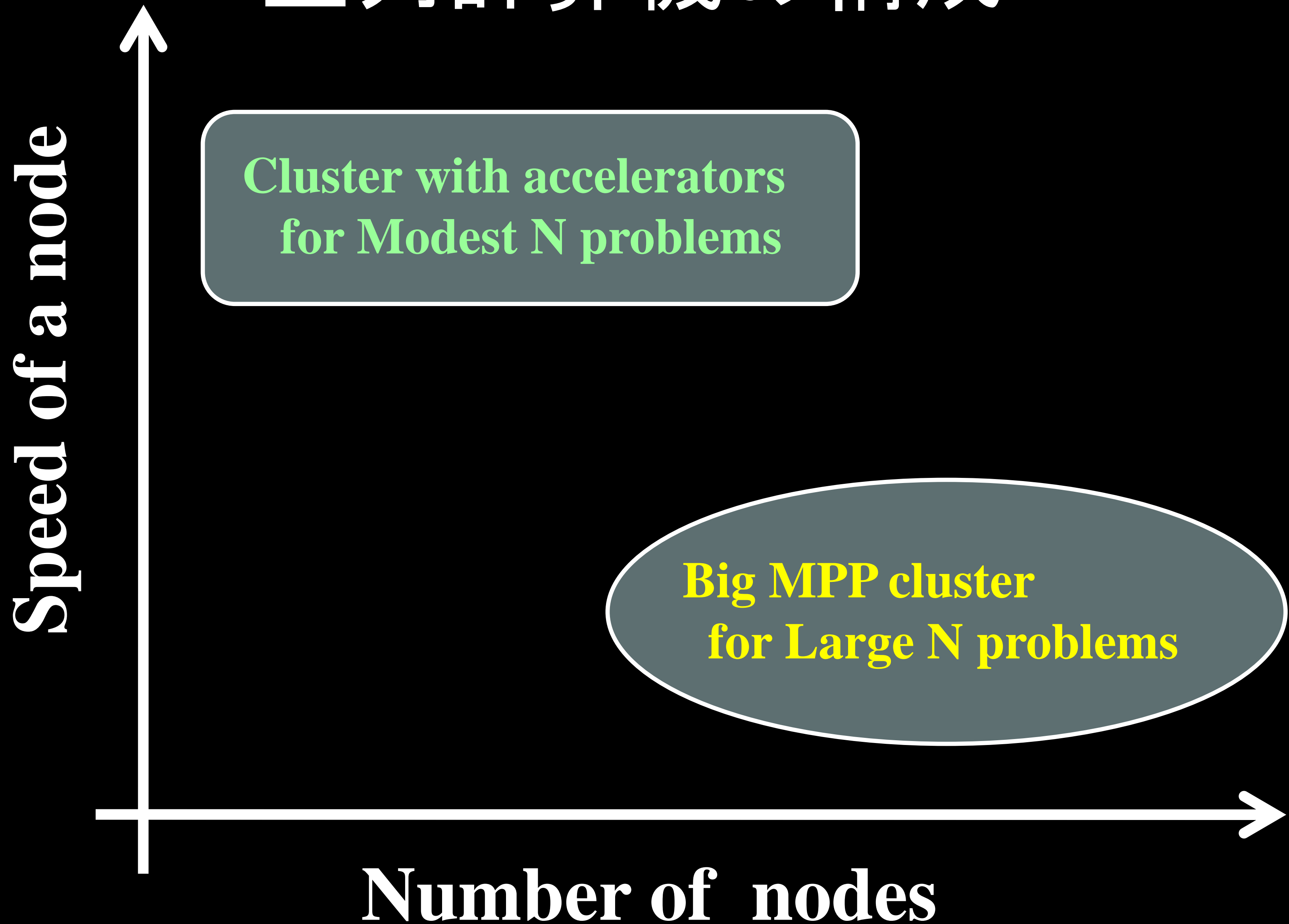
CISCO (J & K′)
January 28, 1999

The APM Galaxy survey
Maddox Sutherland Efstathiou & Loveday

Grand Challenge problems

- Simulations with very huge N
  - One big run with N ~ 10^{9-12}
  - Scalable on a big MPP system
  - Limited by memory size
- Modest N but complex physics
  - Precise modeling of formation of astronomical objects like galaxy, star, solar system.
  - Many runs with N ~ 10^{6-7}
  - Demand a cluster of powerful nodes
    - where accelerators are required and effective!

並列計算機の構成



Many-core Accelerators

- Cell, ClearSpeed, GPU etc.
  - have FP units as many as 32 – 1000 or more
  - Number of FP units is continuously rising…
    - Driven by demand for high performance gaming!
    - 2 x growth with every generation (~1.5 yr or so)

Latest Cypress GPU (ATi)
1600 FP units (single precision)
Running at 850 MHz
1 GB
16x PCI-E gen2
Consume ~ 200W

TOP500 List (Nov. 2009)

Two systems use accelerators out of top 5 systems

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}
1	Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00
2	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2009 IBM	122400	1042.00	1375.78
3	National Institute for Computational Sciences/University of Tennessee United States	Kraken XT5 - Cray XT5-HE Opteron Six Core 2.6 GHz / 2009 Cray Inc.	98928	831.70	1028.85
4	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2009 IBM	294912	825.50	1002.70
5	National SuperComputer Center in Tianjin/NUDT China	Tianhe-1 - NUDT TH-1 Cluster, Xeon E5540/E5450, ATI Radeon HD 4870 2, Infiniband / 2009 NUDT	71680	563.10	1206.19

PowerXCell 8i

Radeon HD4870

Green500 List

All top systems use accelerators

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)	TOP500 Rank*
1	722.98	Forschungszentrum Juelich (FZJ)	QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus	59.49	110
1	722.98	Universitaet Regensburg	QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus	59.49	111
1	722.98	Universitaet Wuppertal	QPACE SFB TR Cluster, PowerXCell 8i, 3.2 GHz, 3D-Torus	59.49	112
4	458.33	DOE/NNSA/LANL	Blade[illegible] Powe[illegible] GHz, Infiniband	276	29
4	458.33	IBM Poughkeepsie Benchmarking Center	BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Infiniband	138	78
6	444.25	DOE/NNSA/LANL	BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband	2345.5	2
7	428.91	National Astronomical Observatory of Japan	GRAPE-DR	51.2	445
8	379.24	National SuperComputer Center in Tianjin/NUDT	Radeon HD4870	1484.8	5
9	378.77	King Abdullah University of Science and Technology	Blue Gene/P Solution	504	18

PowerXCell 8i

アクセラレータでの高性能演算

- メニーコアアクセラレータとは
  - 100個以上の演算器が並列に動作
    - ベクトル演算あるいは並列演算
  - 単精度性能が非常に高速 ~ 2.7 Tflops
    - 倍精度の性能は単精度性能の0.1 – 0.5倍
  - 現状では、どれも自律的には動作できない
    - ホスト計算機から制御される。
    - 別のメモリ空間をもつのでデータ転送が必要
  - 演算性能とメモリ性能のギャップが大きい
    - 2.7 Tflops vs. 150 GB s^{-1}
  - 複雑なメモリ階層: 明示的な割り当て必須

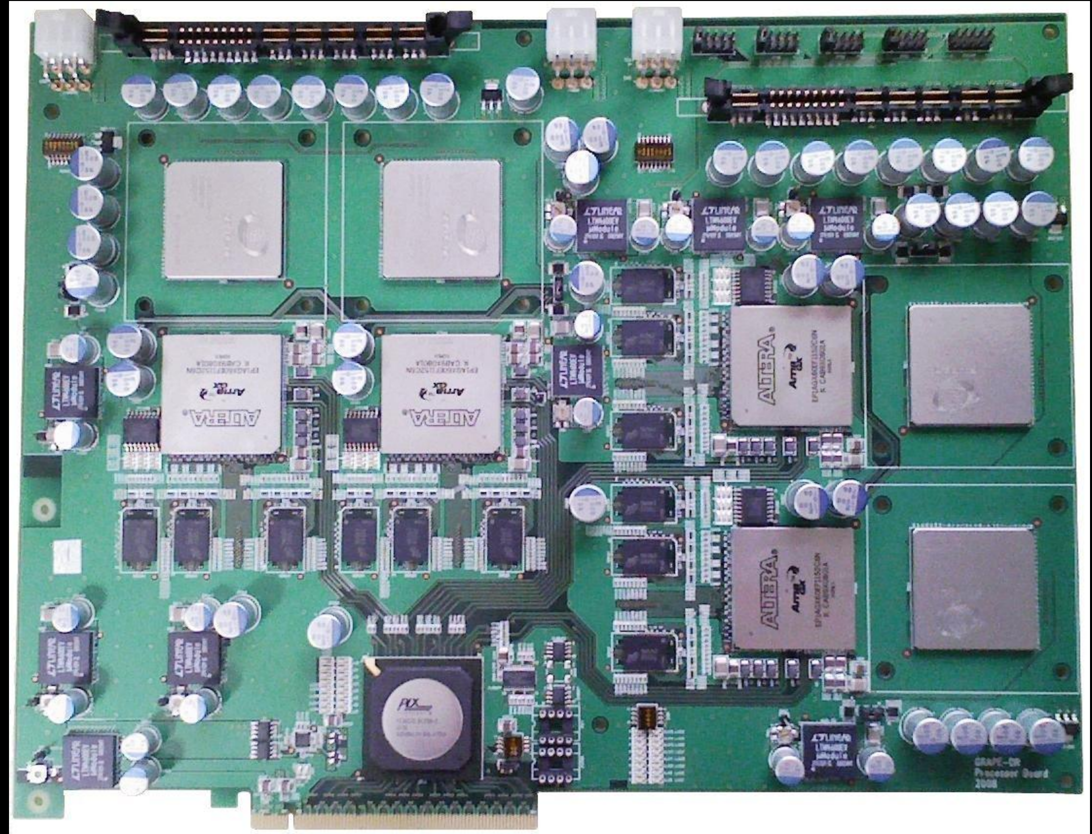
Challenges

- How to program many-core systems?
  - Like a vector-processor but not exactly same
  - Many programming models/APIs for rapidly changing architectures
- Memory wall
  - at the local memory
    - 2.7 Tflops vs. 153 GB s^{-1}
  - at I/O the accelerators
    - Only 16 GB s^{-1}
    - External I/O in cluster configuration is more severe

Programming efforts require

- on how we I/O to/from accelerators
  - Mainly programming for CPU
    - relatively easy
- on how we use FP units
- on how we use internal memories
  - Programming for GPU
    - strongly dependent on a given architecture
    - where we need to optimize
- on how we program a cluster of GPU
  - no definitive answer

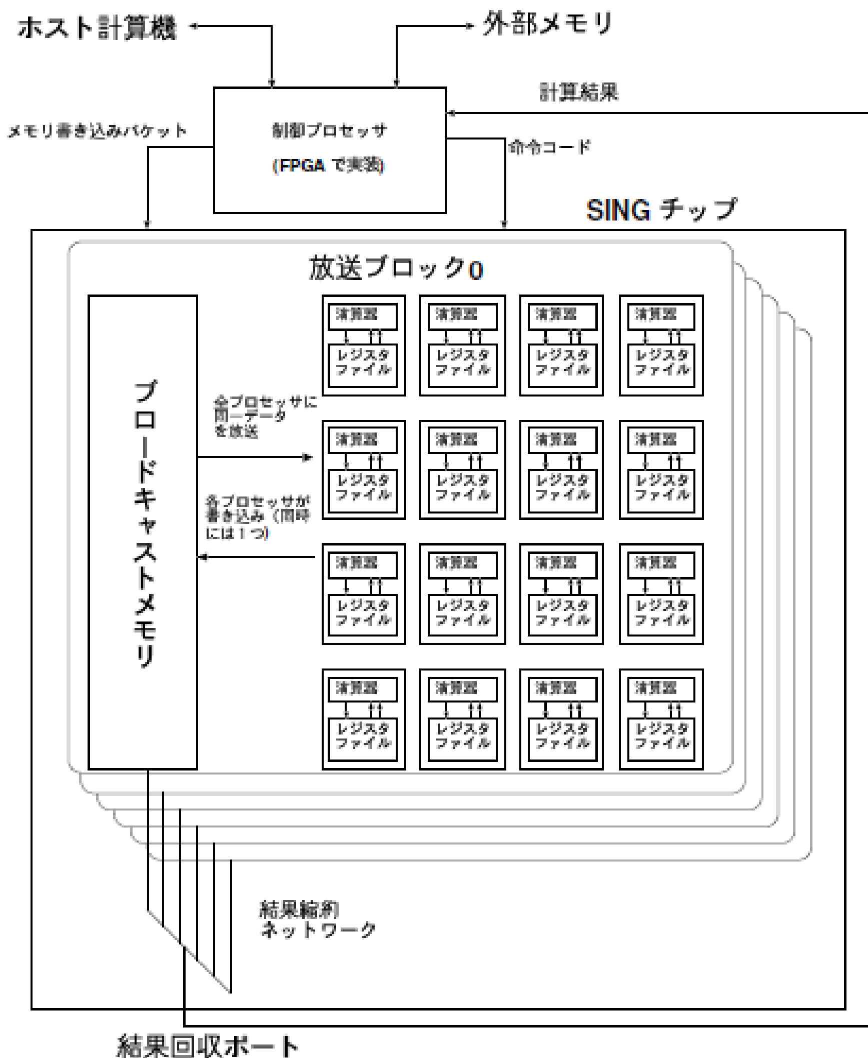
GRAPE-DR



One Chip:
512 PEs
Running at 400 MHz
8x PCI-E gen1
288 MB
Consume ~ 50 W

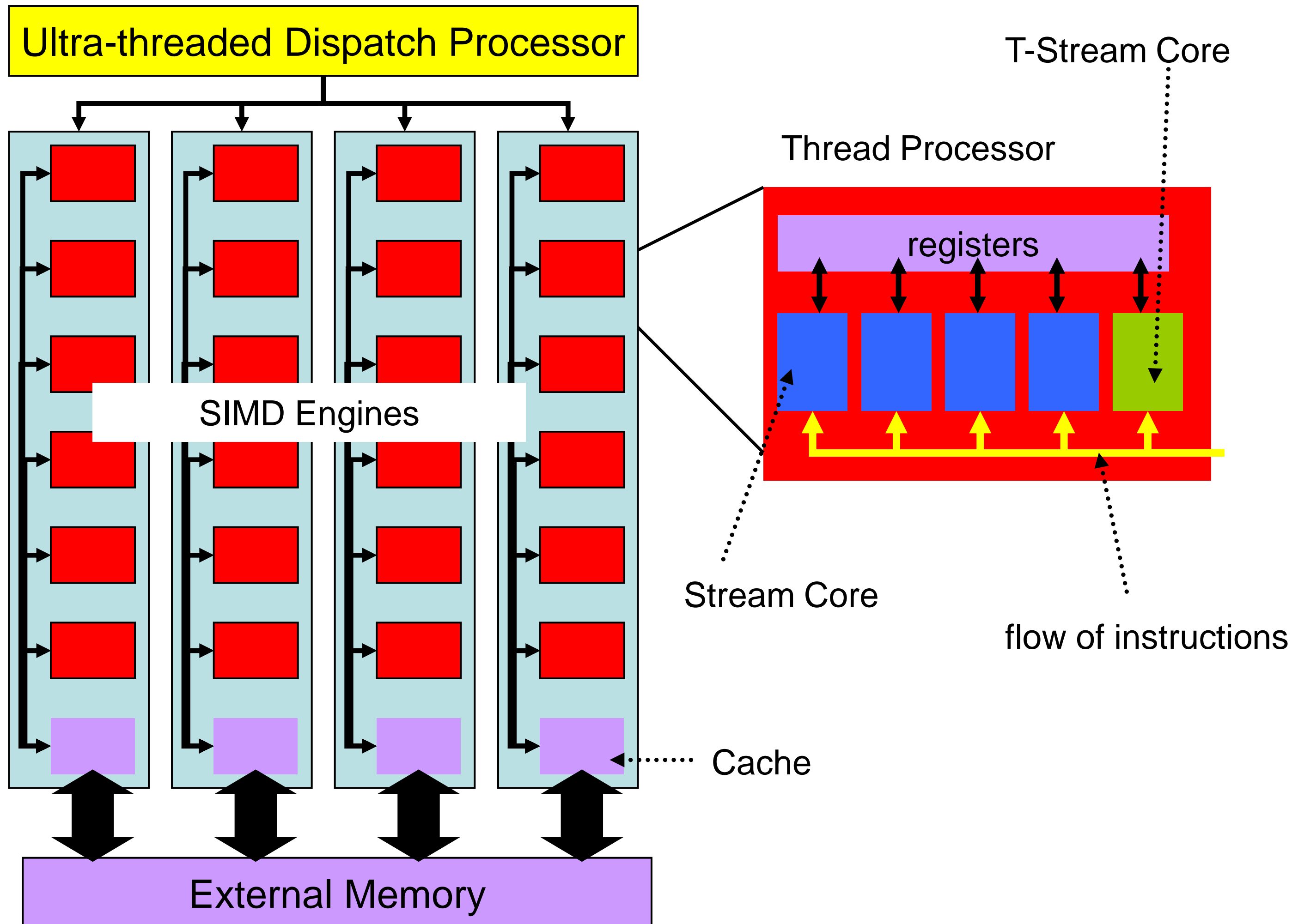
Ranked at 445th on TOP500
Ranked at 7th on Green500

GRAPE-DRの構造



- 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

GPU:RV770の構造

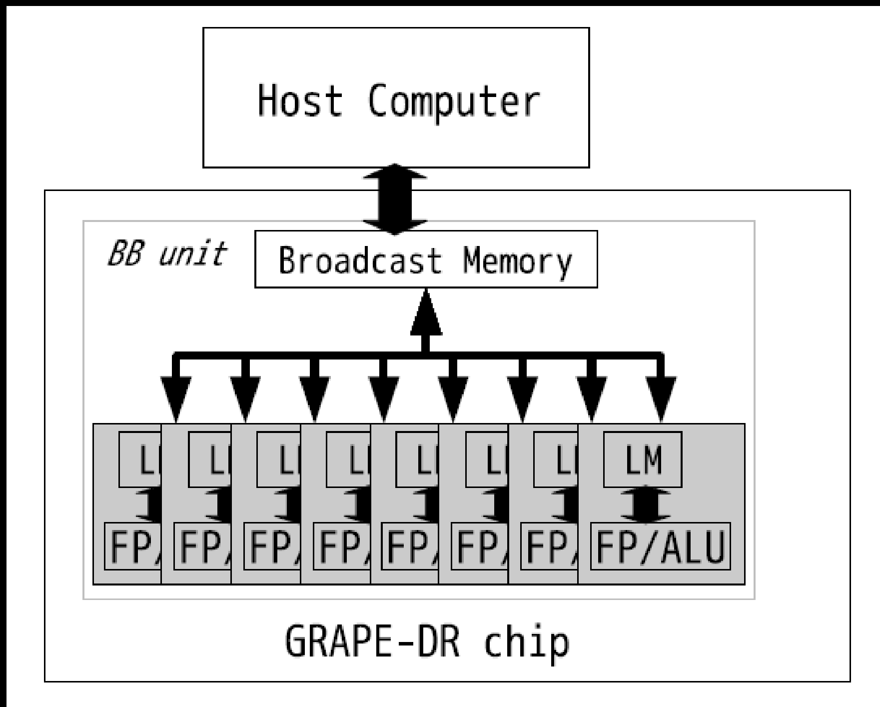


GRAPE-DRとGPUの比較

- 共通点
  - DP性能が200 GFLOPSを超える
  - SPとDPでリソース共有
  - レジスタが多い: 72/256 DP words
- 相違点
  - 演算コア数：320 vs. 512
    - グループ数: 20 SIMD engines vs. 16 BB units
  - Cypressの演算コアはVLIWプロセッサで複雑
  - GRAPE-DRはBMとreduction networkを持つ
    - Cypressではテクスチャフェッチユニットが効果的
  - 外部メモリのアクセス速度

Many-core Accelerators

- Both GRAPE-DR and R700 GPU
  - DP performance > 200 GFLOPS
  - Have many local registers : 72/256 words
  - Resource sharing in SP and DP units



But different in

- **R700 has more complex VLIW stream cores**
- **R700 has no BM**
- **R700 has faster memory I/O**
- **DR has reduction network for efficient summation**

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 (Brook+, C for CUDA)
  - アセンブリ言語(IL , PTX, DRアセンブリ)
- 最適な利用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
  - メモリ階層の利用方法
  - 様々な制限

アクセラレータのプログラミング(2)

- C for CUDA (NVIDA), Brook+(ATI)
  - 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
    - スレッド数を考慮したプログラミング
    - 共有メモリをキャッシュとして利用する必要あり
- OpenCL
  - GPU, CPU, DSPなどを統一して利用可能
  - プログラミングモデルは上記の環境と同じ
  - 問題点
    - 異なるアーキテクチャを唯一の抽象化で扱える？

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

我々のやりたい計算の一例

- 以下のような常微分方程式を解く

$$\frac{d\overrightarrow{v_i}}{dt} = \sum_{j=1}^{N} \vec{f}(\vec{r}_i - \vec{r}_j)$$

where f is gravity, hydro force etc…

- GRAPE-DRは右辺の計算に適した構造
  - GRAPE-DR用のコンパイラの開発
  - その問題特化型コンパイラのGPUへの援用

A simple way to compute RHS

- N粒子に働く力の総和演算を計算

```
for i = 0 to N-1
  s[i] = 0
  for j = 0 to N-1
    s[i] += f(x[i], x[j])
```

Fig. 1. A simple nested loop to computer a general force calculation.

  - それぞれのs[i]は並列に計算できる
    - Massively parallel if N is large
  - さらに, 与えられたiに対して、異なるjとの関数f(x[i],x[j])も並列に計算できる

Unrolling (vectrization)

- 並列計算可能なのでループを展開できる

```
for i = 0 to N-1 each 4
  s[i] = s[i+1] = s[i+2] = s[i+3] = 0
  for j = 0 to N-1
    s[i]   += f(x[i],   x[j])
    s[i+1] += f(x[i+1], x[j])
    s[i+2] += f(x[i+2], x[j])
    s[i+3] += f(x[i+3], x[j])
```

- Two types of variables
  - x[i] and s[i] are unchanged during j-loop
  - x[j] is shared at each iteration
- それぞれのx[i]の計算をアクセラレータの異なるプロセッサに割り当てる

GPUでの最適計算手法

```
for i = 0 to N-1
  acc[i] = 0
  for j = 0 to N-1
    acc[i] += f(x[i], x[j])
```

~ 300 Gflops

```
for i = 0 to N-1 each 4
  acc[i] = acc[i+1] = acc[i+2] = acc[i+3] = 0
  for j = 0 to N-1
    acc[i]   += f(x[i],   x[j])
    acc[i+1] += f(x[i+1], x[j])
    acc[i+2] += f(x[i+2], x[j])
    acc[i+3] += f(x[i+3], x[j])
```

~ 500 Gflops

```
for i = 0 to N-1 each 4
  acc[i] = acc[i+1] = acc[i+2] = acc[i+3] = 0
  for j = 0 to N-1 each 4
    for k = 0 to 3
      acc[i  ] += f(x[i  ], x[j+k])
      acc[i+1] += f(x[i+1], x[j+k])
      acc[i+2] += f(x[i+2], x[j+k])
      acc[i+3] += f(x[i+3], x[j+k])
```

~ 700 Gflops

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;
```

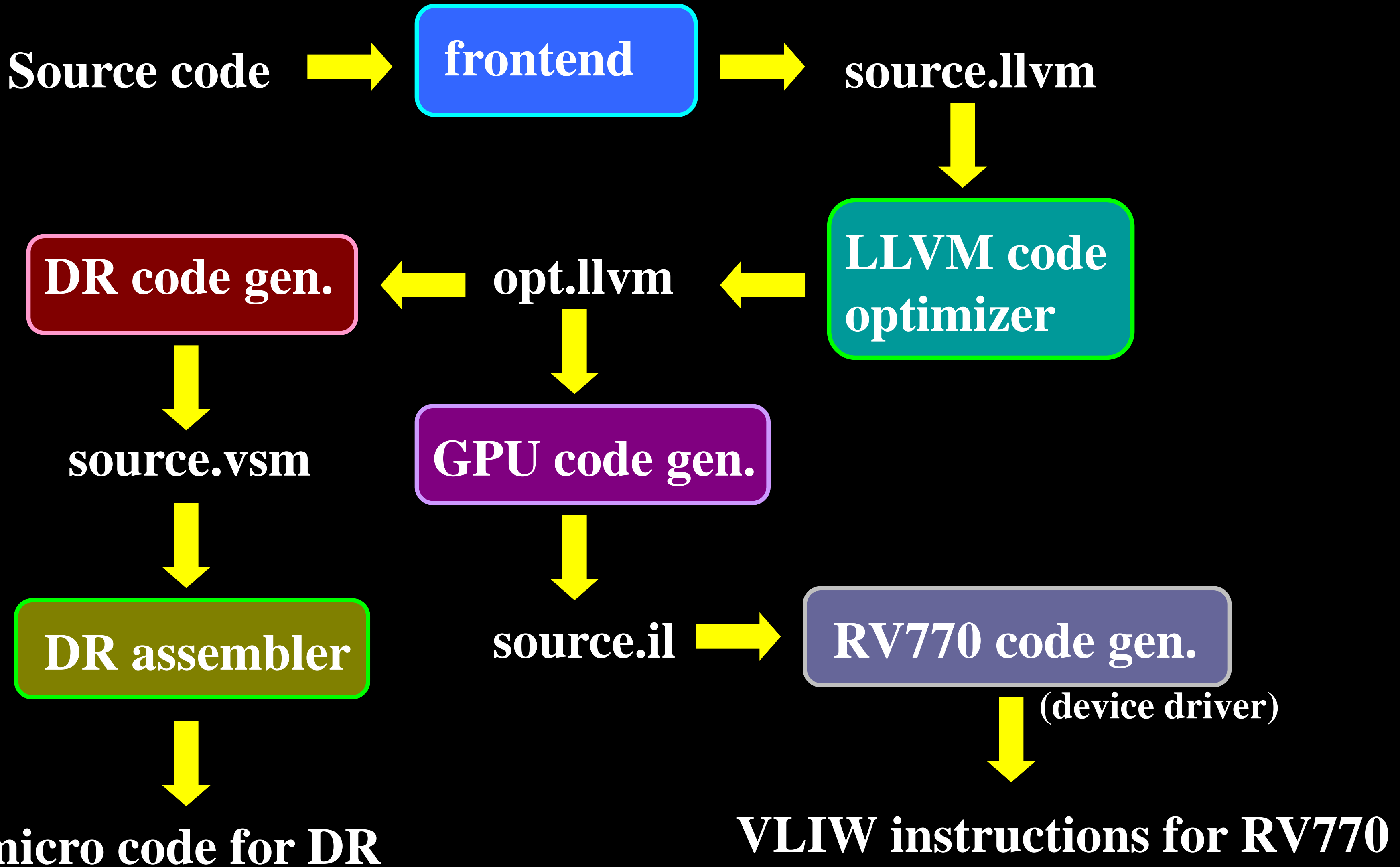
Usage Model (3)

- 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

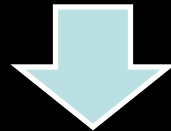


http://galaxy.u-aizu.ac.jp/trac/note/

Example : N-body

- Simple softened gravity

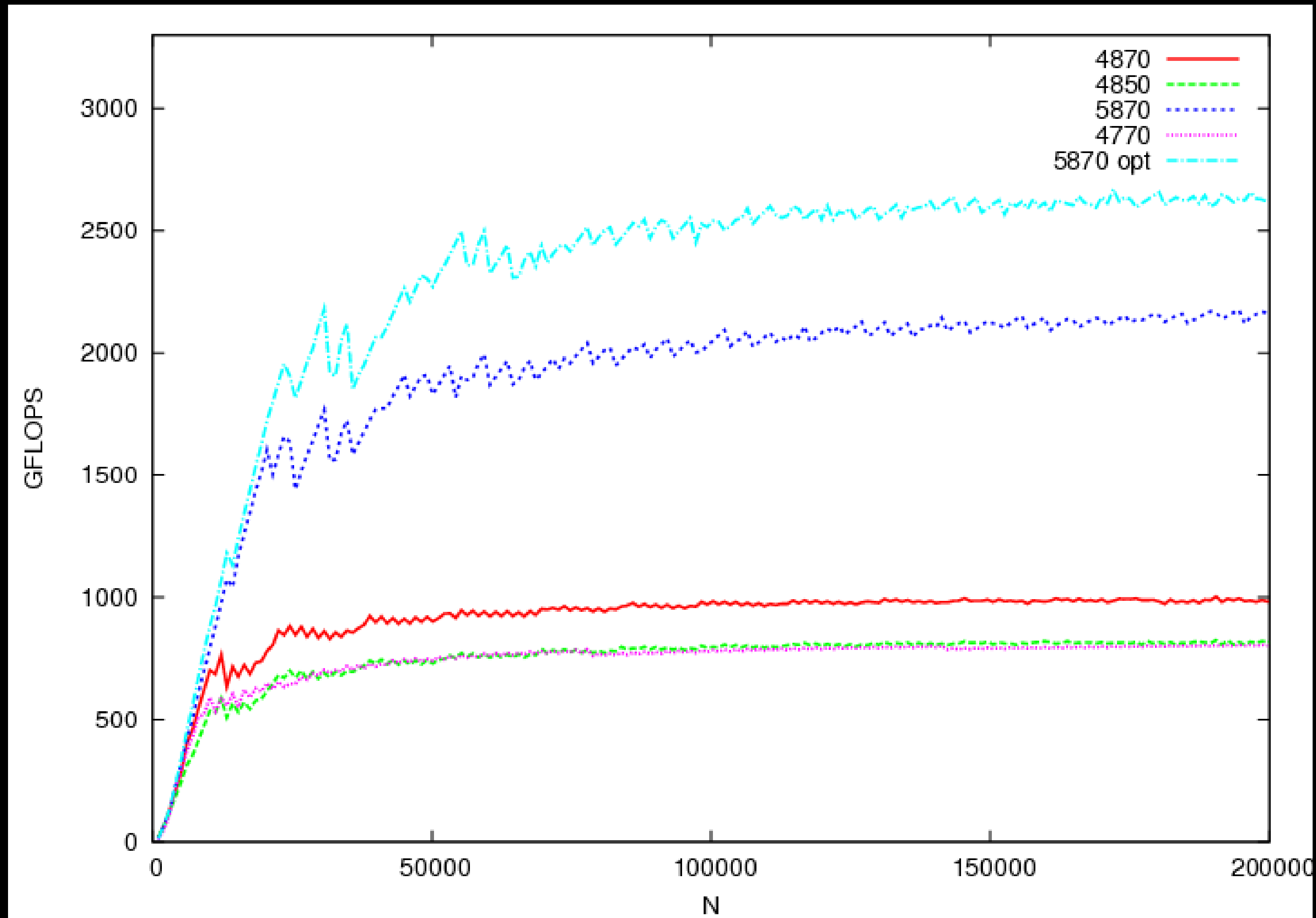
$$\boldsymbol{f}_i = \sum_{j=1}^{N} \frac{m_j(\boldsymbol{x}_i - \boldsymbol{x}_j)}{(|\boldsymbol{x}_i - \boldsymbol{x}_j|^2 + \epsilon^2)^{3/2}},$$



```
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;
```

Performance of $O(N^2)$ algorithm

On a recent GPU ~ 2.6 Tflops



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

$$a = 77617.0$$

$$b = 33096.0$$

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

FP演算でエミュレーション

- 四倍精度(DD)演算の場合
  - 変数　2つの倍精度変数で表現
  - 精度　仮数部　106 bit, 指数部 11 bit
  - 加算　20回の倍精度演算
    - 演算密度 5.0演算/1語読み出し
  - 乗算　23回の倍精度演算
    - 演算密度 5.7演算/1語読み出し

  - 演算密度が高いためメニーコアアクセラレータでの計算にむいている
    - キャッシュありの現代のCPUにも向いている？

DD演算のCPUでの性能

- CPUでの演算性能まとめ
  - 加算の場合で60 – 70 Mflops
  - 乗算の場合で45 – 66 Mflops
  - 演算器のレイテンシがボトルネック
    - x86アーキテクチャでは論理レジスタが少ないため、ループアンローリングは効かない

	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.

DD演算のアクセラレータでの性能

- GPUとGRAPE-DRでは高性能が予想される
  - 演算密度が高いので向いている
  - そもそも倍精度演算性能がCPUより高い
    - 500 Gflops vs. 20 – 30 Gflops
  - 演算器が多くそれぞれに専用レジスタがある
    - 全レジスタ数が圧倒的に多い
    - 6 万語 vs. 128語 (4 core SSEレジスタの場合)
    - レジスタ数が多いのでハードウエアでループアンローリングをやっていることと同等

GRAPE-DRでのDD演算

- DD演算のアルゴリズムを実装した
  - DR用アセンブリ言語で記述
  - 加算 21 step
  - 乗算 41 step
    - 乗算器が 50bit x 25bit であるため
  - 除算 199 step, 逆数平方根 279 step
    - いずれも倍精度の初期値計算とニュートン法
- 理論的な性能は (380 MHz動作時)
  - 加算と乗算それぞれ9.3と4.7Gflops

RV770でのDD計算(1)

- IL(仮想アセンブリ言語)により実装
  - ILは3 operandsの命令体系
  - ILはVLIWの機械語に翻訳される
    - 以下VLIW命令数での結果
  - 加算 21 step
  - 乗算 25 step
  - 除算 53 step
  - 性能予測
    - 750 MHz時 秒間1.2×10^{11}個の VLIW命令 (RV770)
    - 加算, 乗算, 除算 : 5.7, 5.2, 2.3 Gflops

RV770でのDD計算(2)

- 単独演算でのVLIWスロットの分布

命令	5 slots	4 slots	3 slots	2 slots	1 slots	計
加算	2	6	0	13	0	21
乗算	5	9	1	10	0	25
除算	7	25	1	20	0	53

  – 演算器の利用率が低いため、演算性能が低めになっている
  – 演算が連続するとスロットがより埋まるため、演算性能が向上すると予測される

実性能の評価(1)

- ファインマンループの積分
  - 素粒子衝突実験の検証に必要とされる
  - 情報落ちが発生するため倍精度では困難
  - 多重積分を100万組のパラメータについて計算
    - 一例では 5.5x10^{16} FP operations
  - 二重指数型積分法により級数となる

$$D(x,y,z) = -xys + (x+y)(1-x-y-z)t + xM_a^{\ 2} + yM_b^{\ 2}$$

$$-(x+y)(1-x-y-z)m_e^2 + (1-x-y-z)m_b^2$$

$$+ zm_a^2 - z(x+y)m_f^2$$

$$R = \sum \frac{G(z)}{D^2(z)}$$

Feynman-loop integral

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

```
LMEM  xx, yy, cnt4;
BMEM  x30_1, gw30;
RMEM  res;
CONST tt, ramda, fme, fmf, s, one;

zz = x30_1*cnt4;
d = -xx*yy*s-tt*zz*(one-xx-yy-zz)+(xx+yy)*ramda**2 +
     (one-xx-yy-zz)*(one-xx-yy)*fme**2+zz*(one-xx-yy)*fmf**2;
res += gw30/d**2;
```

実性能の評価(2)

- GRAPE-DRの場合
  - micro codeは1079 step
- Cypress/RV770の場合
  - VLIW命令は 319 step
    - 81%は、4または5 slotsが埋まっている
    - 命令融合の効果を確認
  - 利用レジスタ数は39個
    - 性能向上の余地がある

実性能の評価(3)

- CPU, GPU, GRAPE-DRにおいて
  - 級数の項数を変化させて、実機で計算した

	$N = 256$	$N = 512$	$N = 1024$	$N = 2048$	clock
GRAPE-DR	0.21	1.21	7.83	55.1	380
Cypress	0.05	0.29	1.94	14.2	850
Core i7	7.39	59.0	472	—	2670

(経過時間 sec)

  - 演算量を $28N^3$ で評価すると
    - CPU ~ 64 QD-Mflops
    - GPU ~ 13 – 17 QD-Gflops
      - I/Oが高速のためN依存が小さい
    - GRAPE-DR ~ 4 QD-Gflops

Mixed Precisionの例(1)

- 高精度積分では以下の部分で高精度が必要

```
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;
```

Mixed Precisionの例(2)

- 以下の記述を追加する

```
IMPLICIT REAL8;
LMEM  xi, yi, zi, e2;
BMEM  xj, yj, zj, mj;
RMEM  ax, ay, az;
REAL16 xi, yi, zi, xj, yj, zj, ax, ay, az;
```

- Performance of the Hermite scheme
  - 4-th order integration scheme
  - 6.31 GFLOPS with QP
  - 27.8 GFLOPS with mixed precision (4x gain)
    - With negligible integration error compared to QP

提案システムとその発展

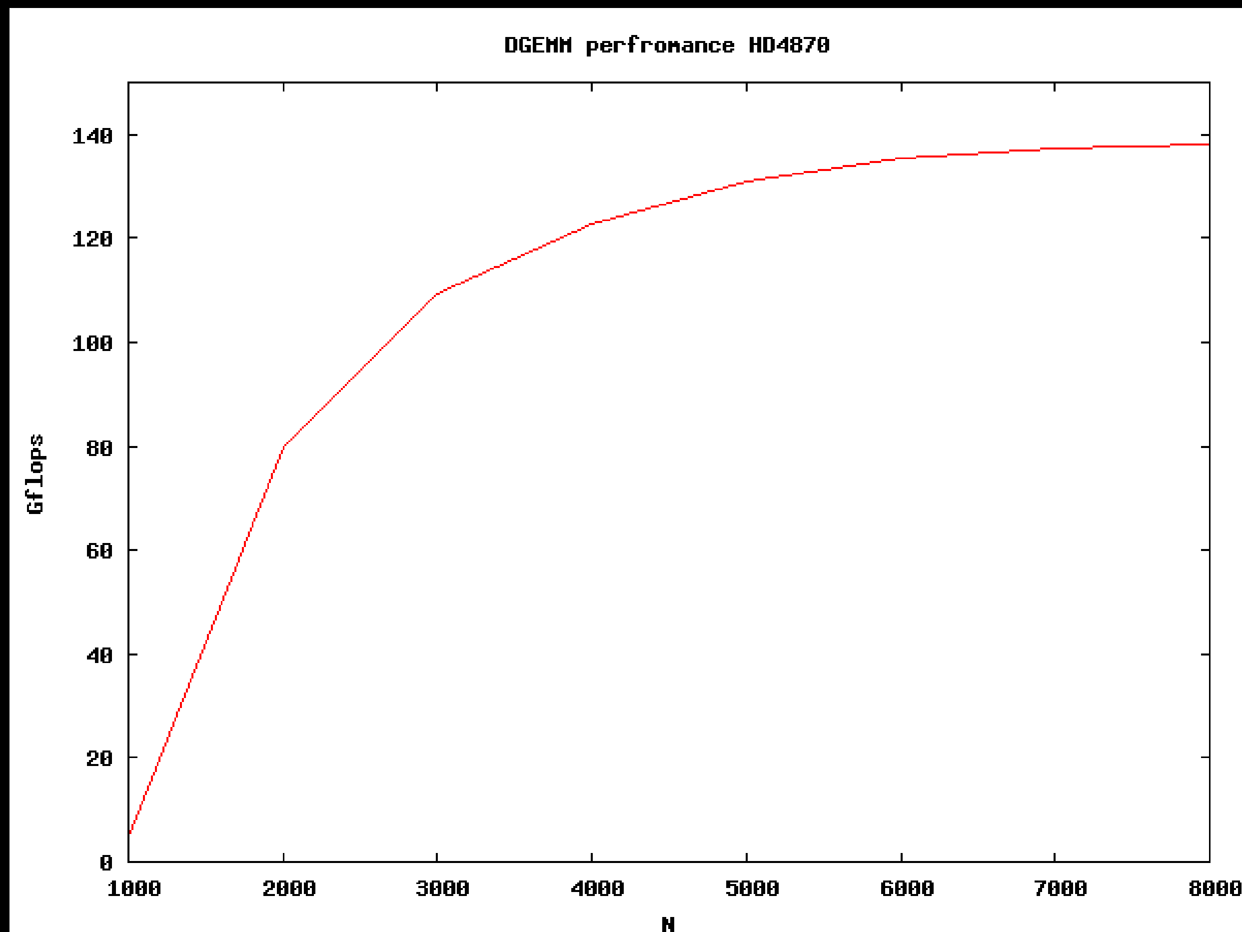
- GRAPE-DRとGPUで総和演算を並列に計算するためのDSLコンパイラ
  - DSLのソースより最適なアクセラレータ用コードとデータ転送用APIを生成する
    - ユーザーはAPI経由でアクセラレータを利用
  - オプション指定により単, 倍, 四倍精度に対応
    - 一部混合演算をサポート
- 問題に応じて同様のシステムを実装可能
  - コード生成部はほぼ流用可能
  - どのような入力変数の性質があるか？

アクセラレータのまとめ(1)

- 演算密度が高いなら高性能
- $O(N^3)$またはそれ以上の計算
  - 非常に高性能(なはず)
  - QDによる積分 ~ 15 QD-GFLOPS
    - 倍精度で300 GFLOPS相当
- $O(N^2)$の計算 : direct summation
  - 単精度2.6 TFLOPS相当
  - ほぼ100%の演算効率

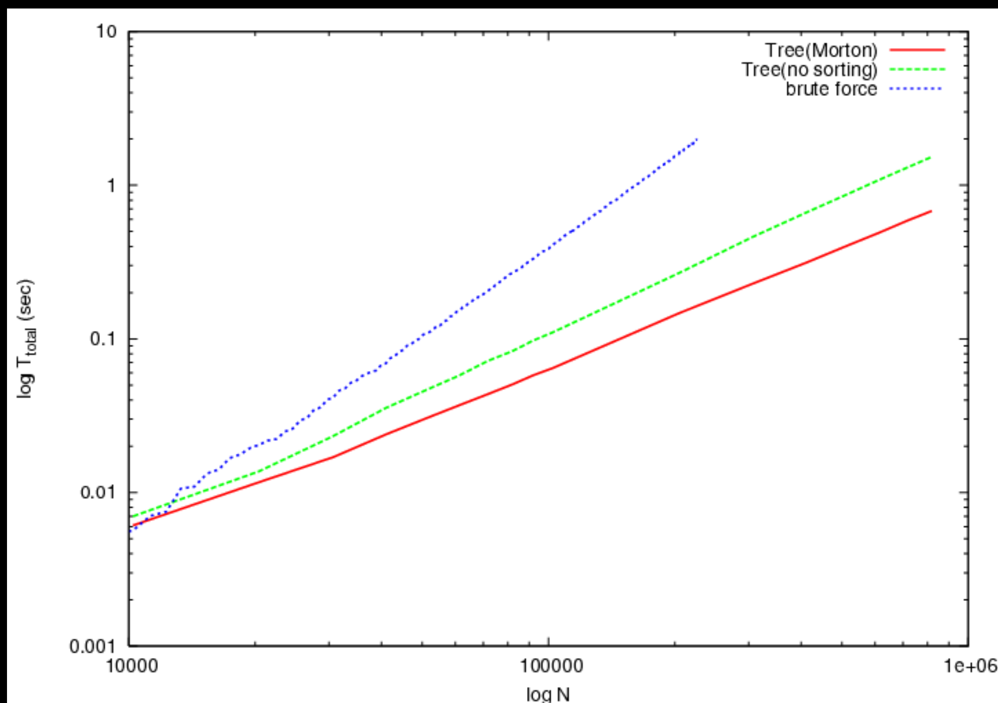
アクセラレータのまとめ(2)

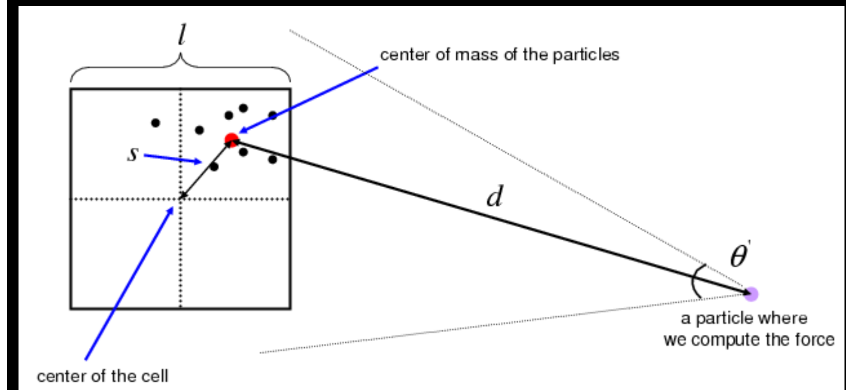
- $O(N^{1.5})$の計算: 行列乗算
  - 約60パーセントの効率



アクセラレータのまとめ(3)

- O(N log N)の計算: 短距離力
  - Oct-treeをGPUで実装
    - Gravity, SPHの実装ができた
  - 効率(~ 1 %)は落ちるがCPUより高速





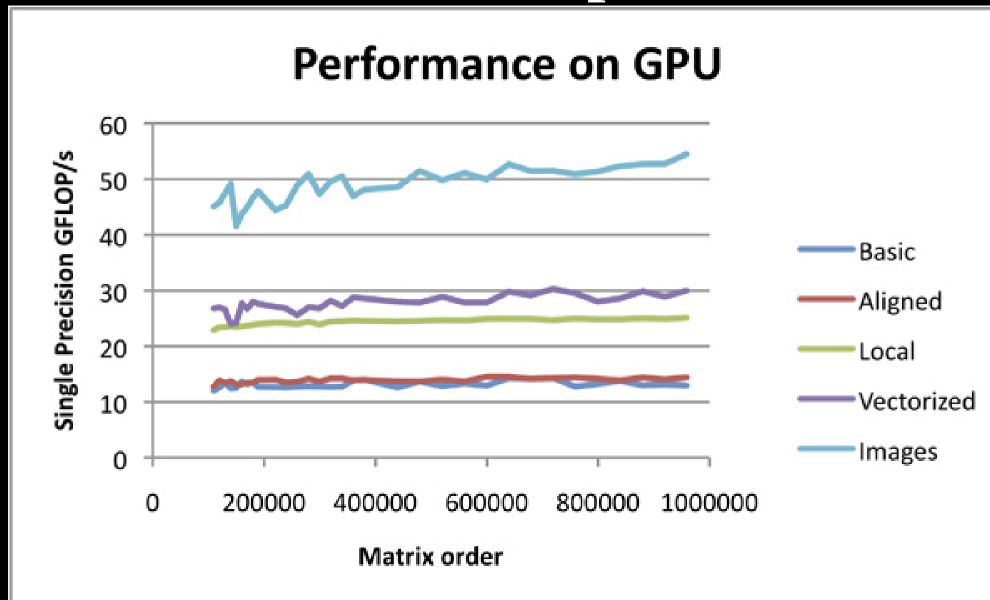
アクセラレータのまとめ(4)

- O(N)の計算: 流体計算など
  - 今後の課題
    - 原理的には非常に難しい
    - 複雑なschemeなら演算密度は高い
    - cacheはほぼ効かない
  - コンパイラ拡張の良い候補

アクセラレータのまとめ(5)

- GPUのcacheについて
  - Cypress : read cache有用 write cacheなし
  - Fermi : read cache? write cacheを追加

OpenCL™ Optimization Case Study: Diagonal Sparse Matrix Vector Multiplication



http://developer.amd.com/documentation/articles/Pages/OpenCL-Optimization-Case-Study.aspx

短距離力用DSL(1)

- Domain Specific Language
  - 問題に特化したプログラミング言語
  - 我々のコンパイラもDSL
  - なぜ有効か？
    - 多くの問題は「計算の枠組み」は同じ
      - 二重ループによる計算, tree法による計算
    - 「なにを計算するのか」が異なる
      - 重力, SPH, 分子間力などなど
  - 「計算の枠組み」を固定して
  - 「なにを計算するのか」だけをプログラムさせる

短距離力用DSL(2)

- 力の計算の近似や短距離ではtree法による演算量の削減が効果的:$O(N^2) \rightarrow O(N \log N)$
  - treeをたどるやりかたは同一
  - 入出力と計算のみが異なる

```
procedure treewalk(i, cell)
  if cell has only one particle
    force += f(i, cell)
  else
    if cell is far enough from i
      force += f_multipole(i, cell)
    else
      for i = 0, 7
        if cell->subcell[i] exists
          treewalk(i, cell->subcell[i])
```

Fig. 3. A pseudo code for the force-calculation by traversing the oct-tree

短距離力用DSL(3)

- 重力の場合
  - 入力：座標と質量 (4要素)
  - 出力：重力加速度 (4要素)
- SPHの例

$$\rho_i = \sum m_j W(\vec{r}_i - \vec{r}_j; h), h = \frac{1}{2}(h_i + h_j)$$

$$\rho_i (\nabla \cdot \vec{v}_i) = \sum (\vec{v}_i - \vec{v}_j) \cdot \nabla W(\vec{r}_i - \vec{r}_j; h)$$

$$\rho_i (\nabla \times \vec{v}_i) = \sum (\vec{v}_i - \vec{v}_j) \times \nabla W(\vec{r}_i - \vec{r}_j; h) \ldots$$

入力
座標,質量,速度,半径,圧力,密度
～15要素

$$\frac{d\vec{v}_i}{dt} = -\sum m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \Pi_{ij} \right) \nabla W(\vec{r}_i - \vec{r}_j; h)$$

$$\frac{d\vec{u}_i}{dt} = \sum m_j \left(\frac{P_i}{\rho_i^2} + \frac{1}{2}\Pi_{ij} \right) (\vec{v}_i - \vec{v}_j) \cdot \nabla W(\vec{r}_i - \vec{r}_j; h)$$

出力
9要素

短距離力用DSL(4)

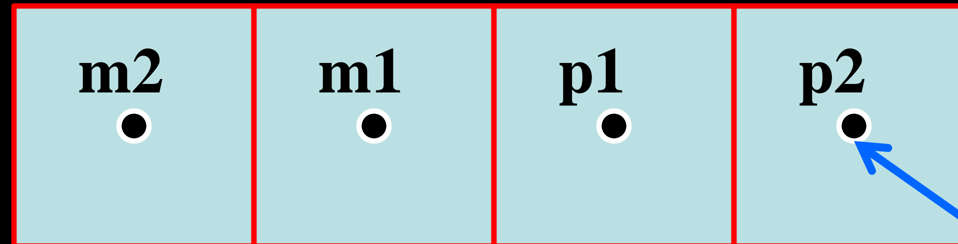
- 変化する部分を赤で示す (in OpenCL)

```
__kernel void tree_gm( __global float4 *pos,  __global float4 *acc_g, __global float *size,
                                   __global int *next, __global int *more, int root, int n)
{
  unsigned int gid = get_global_id(0);
  float4 p = pos[gid];
  float4 acc = (float4)(0.0f, 0.0f, 0.0f, 0.0f);
  int cur = root;
  while(cur != -1) {
    float4 q = pos[cur], dx = q - p;
    float  mj = q.w,  s = size[cur], r2 = dx.x*dx.x + dx.y*dx.y + dx.z*dx.z;
    if (cur < n) {
      if (r2 != 0.0f) {
        r2 += s; // e2
        acc += g(dx, r2, mj);
      }
      cur = next[cur];
    } else {
      if (s < r2) {
        acc += g(dx, r2, mj);
        cur = next[cur];
      } else {
        cur = more[cur];
      }
    }
  }
  acc_g[gid] = acc;
}
```

入力
座標,質量,速度,
半径,圧力,密度… ～15要素

まだ余りよい文法を考案できず

流体計算用DSL




```
FLUX  f_d, f_m, f_e;
GM1  rho_m1, v_m1, p_m1;
GM2  rho_m2, v_m2, p_m2;
GP1  rho_p1, v_p1, p_p1;
GP2  rho_p2, v_p2, p_p2;
....
f_d = ....
```

HPC用のDSL (1)

- 「計算の枠組み」
  - 演算データの入出力と同義である
  - つまり、どのように演算に必要なデータをレジスタまでもってくるか？
    - キャッシュの有効利用
    - GPUならどのように共有メモリを使うか
    - tree構造を使って演算に必要なデータ読み出しというのも、データアクセスの問題に帰着する
  - 難しい：ほぼ全ての最適化の努力はここ
- 「なにを計算するか」
  - レジスタにあるデータを色々と計算する
    - (コンパイラが賢ければ)ある意味簡単

HPC用のDSL (2)

- 計算科学のお仕事
  - 理想：数式　→　プログラム
    - Mathematicaのコードから高速なコードが自動生成できればうれしい！
    - 数値計算ライブラリの利用は理想に近い
  - 現実：数式　→　アルゴリズム → データ入出力 → レジスタ間の演算
    - 汎用プログラミング言語(C, Fortran)で、統一的に実装
  - DSLによりデータ入出力を隠蔽
    - ユーザーの負担が減る(はず)
    - 理想に少し近づく(はず)：ライブラリの自動生成
    - 様々なデータ入出力パターンの研究が必要

まとめ

- メニーコアアクセラレータを有効利用するためのコンパイラの研究開発
  - 粒子シミュレーションへの適用
- 四倍精度演算はGRAPE-DR, GPUで数十倍の高速化が可能
  - メニーコアアクセラレータは効果的
- ある程度面倒な計算を手軽に高速化できる