A Compiler for High Performance Computing with Many-core Accelerators

N.Nakasato (University of Aizu, Japan)

J.Makino (National Astronomical Observatory, Japan)

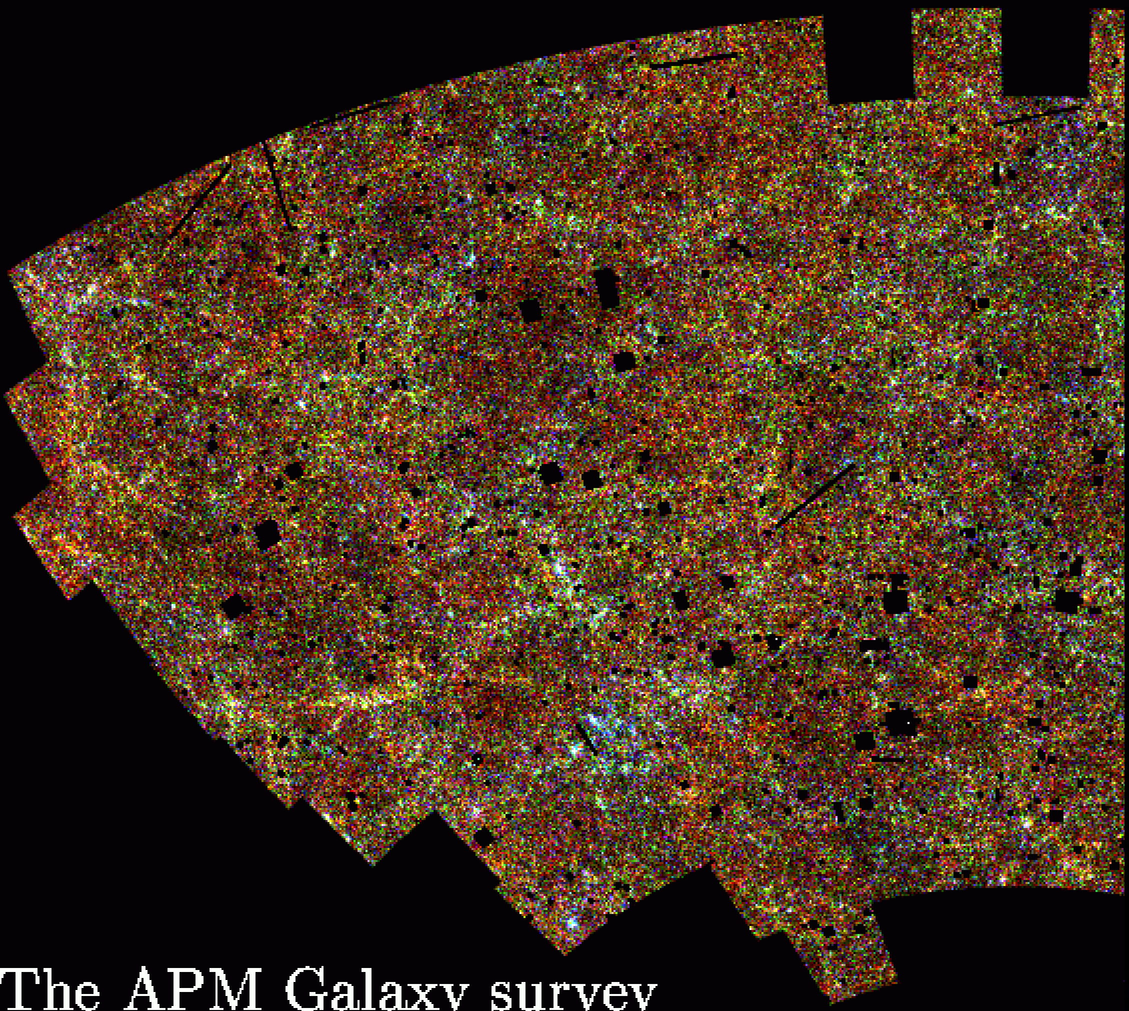
Agenda

- Problem description
  - Astronomical Particle Simulations
- Our Approach
- Performance evaluation
- Summary

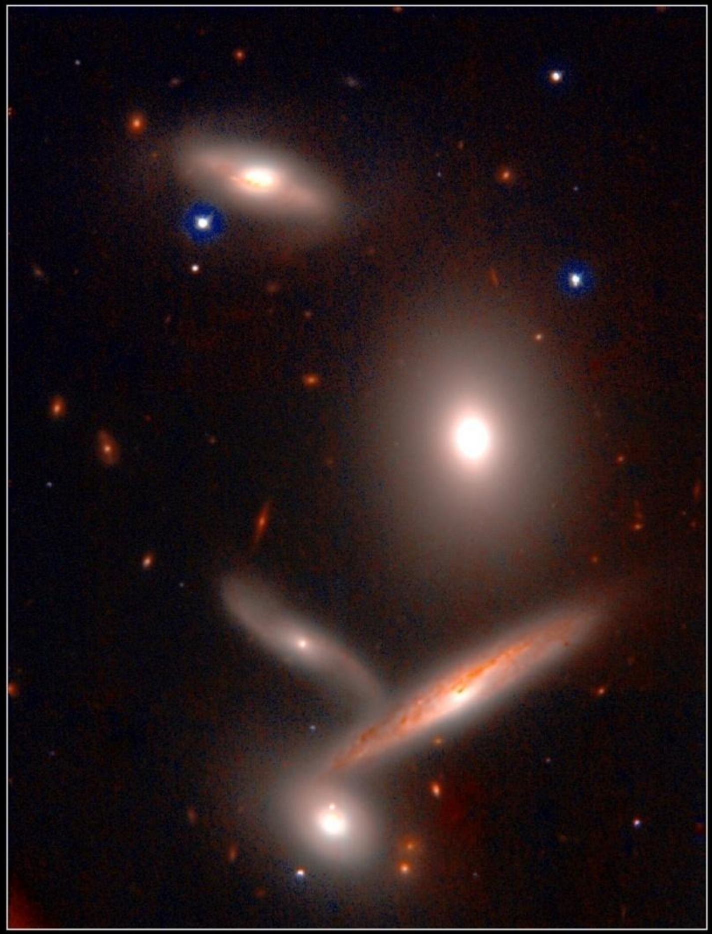
Astronomical Particle Simulation

- Simulate evolution of the universe
  - As a collection of particles
  - Depending on scale, each particle represents
    - Galaxy
    - Star
    - Asteroid
    - Gas blob etc.
  - Particles are interacting
    - Mainly by gravity
      - Long-range force

Grand Challenge Problems



The APM Galaxy survey
Maddox Sutherland Efstathiou & Loveday



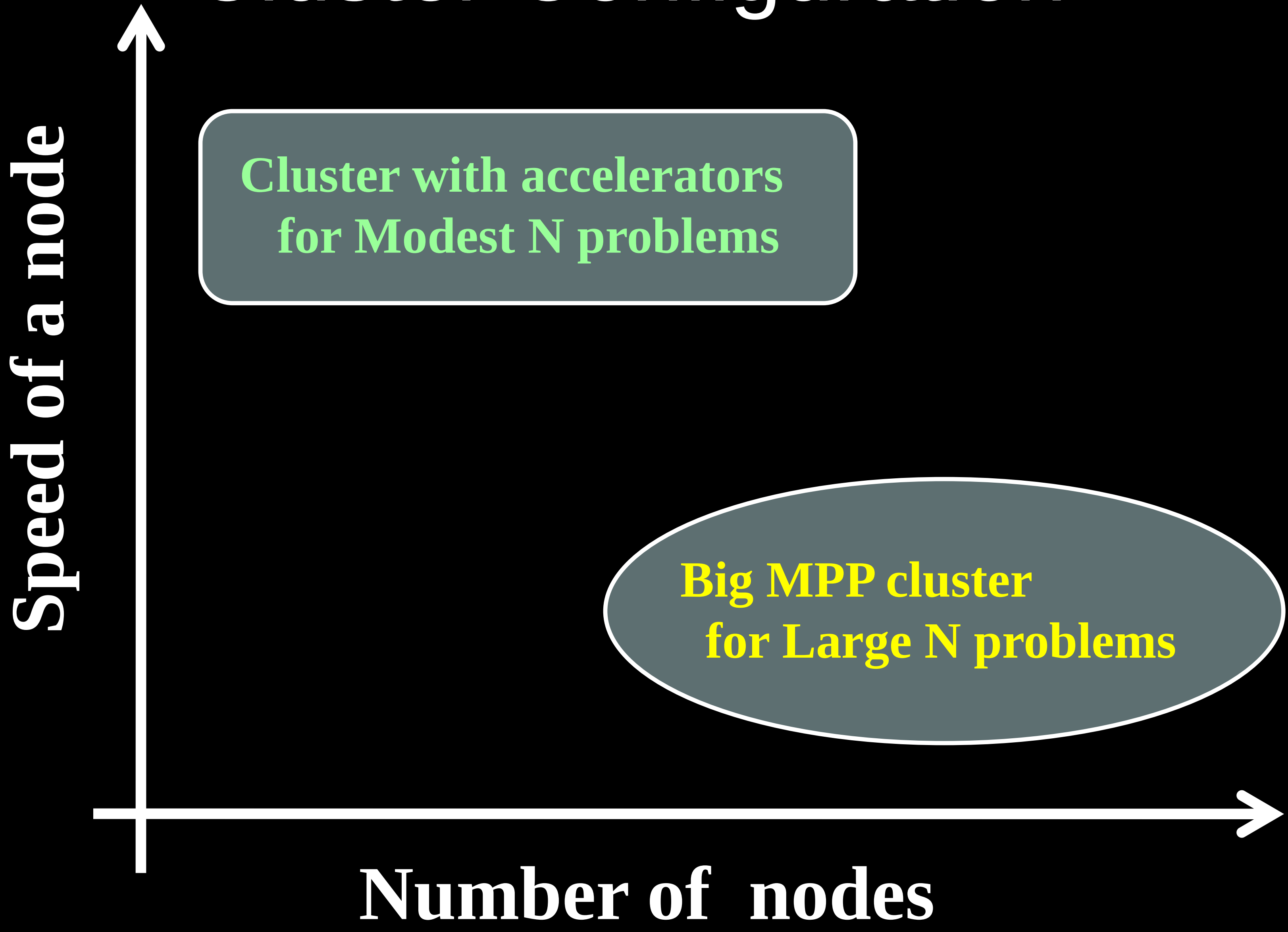
Hickson Compact Group 40
Subaru Telescope, National Astronomical Observatory of Japan
CISCO (J & K′)
January 28, 1999

Grand Challenge Problems

- Simulations with very huge N
  - How is mass distributed in the Universe?
    - One big run with $N \sim 10^{9\text{-}12}$
  - Scalable on a simple big MPP system
    - Limited by memory size

- Modest N but complex physics
  - Precise modeling of formation of astronomical objects like galaxy, star, solar system.
  - Need many runs with $N \sim 10^{6\text{-}7}$

Cluster Configuration



Numerical Modeling

- Solve ODE for many particles

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

where f is gravity, hydro force etc…

- Two main problems
  - How to integrate the ODE?
  - How to compute RHS of ODE?
    - We will use accelerators for this part

A simple way to compute RHS

- Compute force summation as

```
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.

  - Each s[i] can be computed independently
    - Massively parallel if N is large
    - Given i & j, each f(x[i],x[j]) can be computed independently if f() is complex

Unrolling (vectrization)

- Parallel nature enable us to unroll the outer-loop in n-ways

```
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])
```

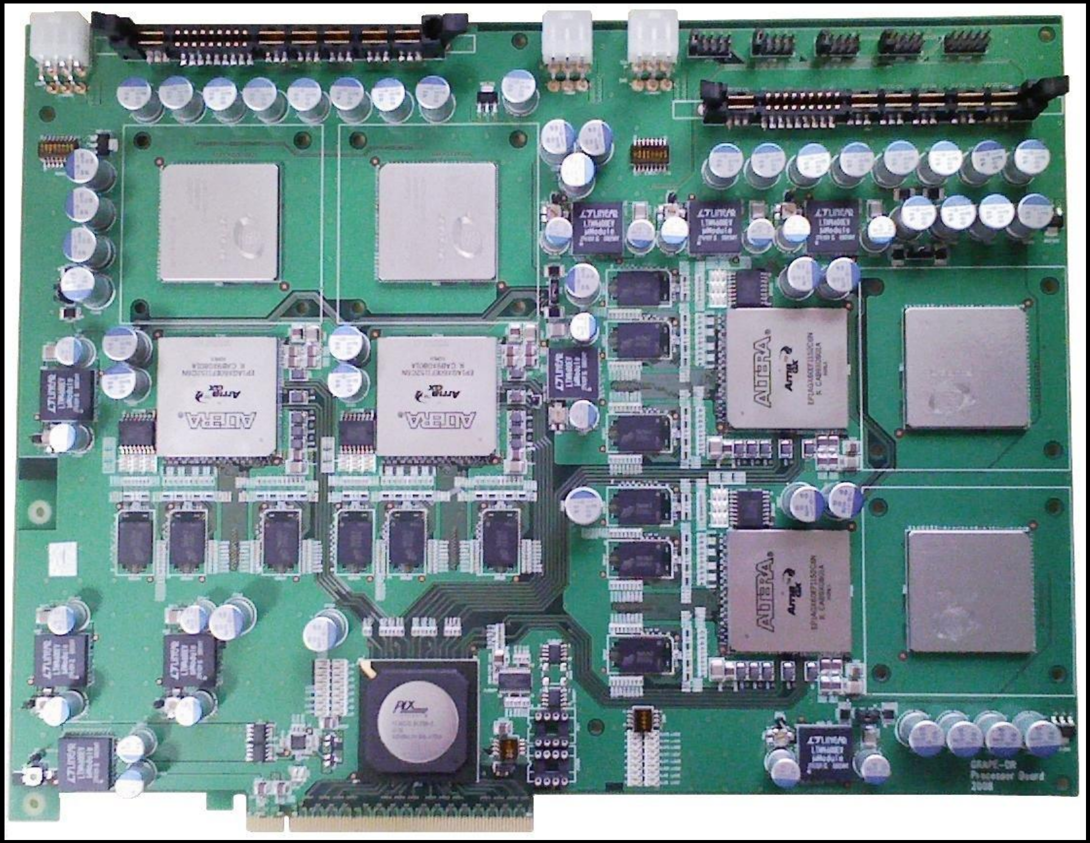
  - Two types of variables
    - x[i] and s[i] are unchanged during j-loop
    - x[j] is shared at each iteration
  - Map computation for each x[i] to PE on accelerators

Using Many-core Accelerators

- To use accelerators, need two programs
  - A program running on host
  - A program running on accelerators
    - Compute kernel
- Example
  - C for CUDA / Brook+
    - Host program in C++
    - Compute kernel in extended C++
      - Function with appropriate keyword
      - Separate source code

GRAPE-DR

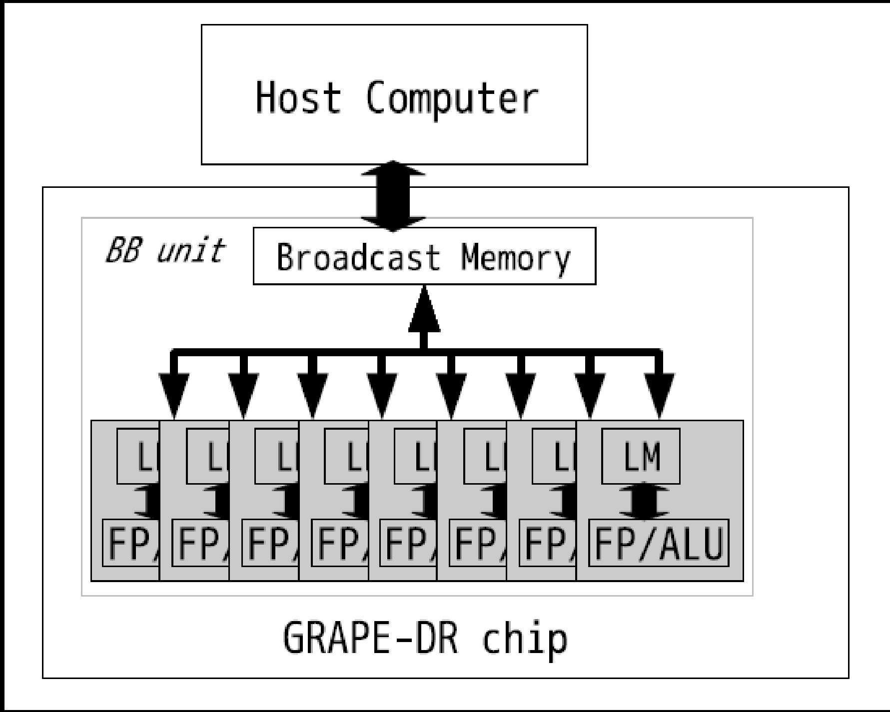
One Chip:
512 PEs
Running at 400 MHz
8x PCI-E gen1
288 MB
Consume ~ 50 W



Ranked at 277th on TOP500
Ranked at 5th on Green500

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

Our Approach

- Ask user to specify
  - Which part of a code is in parallel
  - In addition, what nature of each variables
  - Write that information in DSL

- Then, our compiler generates an code by using predetermined optimization techniques
  - This is dependent on a problem
  - Current one is only for the particle summation

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 …
```

Where the part to be able to compute in parallel

Usage Model (2)

- User write a source in DSL such as

```
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;
```

  - Our compiler generates optimized machine code for GPU / GRAPE-DR

Usage Model (3)

- And also generates APIs as library to send/receive data and control the accelerator

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

Where a user replaces the nested loop with call to APIs and link the code with the generated library

Features

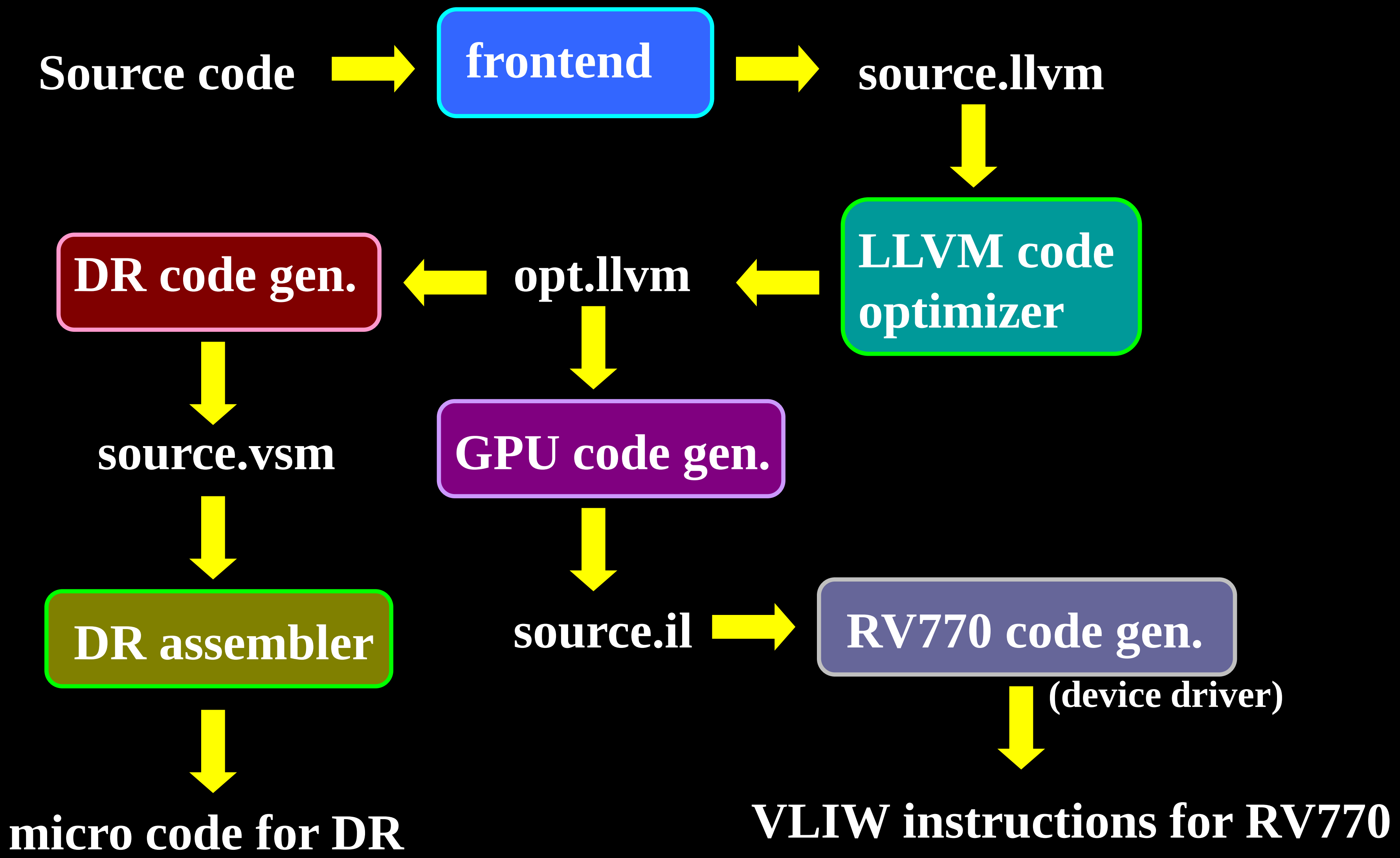
- Accelerates force summation loop
- Support two accelerators
  - R700 architecture GPU
  - GRAPE-DR
    - Developed by JM etal.
- Precision controllable
  - Single, Double, & Quadruple precision
    - QP through DD emulation techniques
  - Partially support mixed precision

Our Compiler

- Written in C++
  - Prototype was developed in Ruby

- We use following software/library
  - Boost sprit for the parser
  - Low Level Virtual Machine for the optimizer
  - Google template library for the code generators

Compiler Flow



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

Example 1 : 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;
```

Optimization on GPU

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

~ 300 Gflops

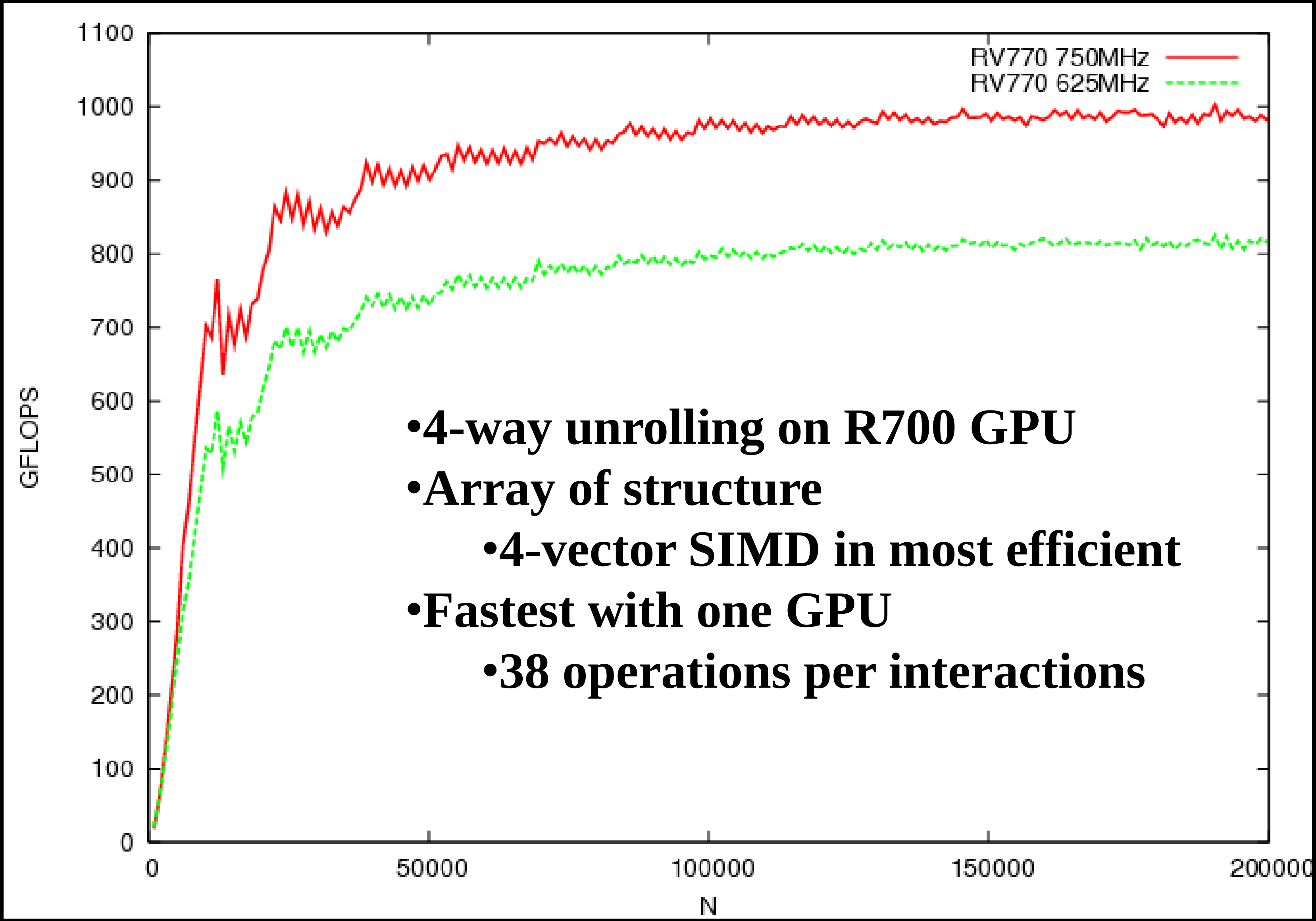
```
for i = 0 to N-1 each 4
  acc[i] = acc[i+1] = acc[i+2] = acc[i+3] = 0
  for j = 0 to N-1
    acc[i]   += f(x[i],   x[j])
    acc[i+1] += f(x[i+1], x[j])
    acc[i+2] += f(x[i+2], x[j])
    acc[i+3] += f(x[i+3], x[j])
```

~ 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

Performance of $O(N^2)$ algorithm



Example 2: 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;
```

Performance of QP operations

- Computation of Feynman-loop integral
  - elapsed time in QP operations

	$N = 256$	$N = 512$	$N = 1024$	$N = 2048$	clock
GRAPE-DR	0.21	1.21	7.83	55.1	380
RV770	0.09	0.66	5.03	39.7	750
Core i7	7.39	59.0	472		2670

  - CPU ~ 80 Mflops
  - R700 GPU ~ 6.43 – 7.57 Gflops
  - GRAPE-DR ~ 2.67 – 5.46 Gflops
- Tow reasons why QP is so fast
  - High compute density
  - DR & R700 are register rich

Example 3: Mixed Precision

- High accuracy integration needs high accuracy in distance and summation

```
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;
```

Mix Precision Example

- Add declaration lines to specify precision of variables

```
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

Comparison

- Our approach is in between two conventional approaches
  - Automatic parallel compiler
    - A user just feed an existing source code
    - But not effective in general
  - Let-users-do-everything-type compiler
    - C for CUDA, OpenCL, Brook+ etc.
    - A user have to specify every details of
      - Memory layout and its movement
      - SIMD operations
      - Threads management on GPU

Conclusion

- Many-core accelerators are effective in astronomical/astrophysical N-body simulations
  - But how to program?
- We have constructed a compiler for many-core accelerators
  - That accelerate force-calculation-loop
  - Features simplicity and controllable precision
- Planed Extension
  - Support O(N log N) method on GPU