

# Goose ソフトウェアパッケージ ユーザガイド

for  version 1.0.1 $\alpha$

最終更新：2009 年 9 月 28 日

**K&F**

Computing Research Co.

株式会社 K & F Computing Research  
E-mail: support@kfcr.jp

# 目次

<b>1</b>	<b>本文書の概要</b>	<b>4</b>
<b>2</b>	<b>インストール</b>	<b>5</b>
2.1	準備 . . . . .	5
2.2	パッケージの展開と環境変数の設定 . . . . .	6
2.3	インストールスクリプトの実行と動作チェック . . . . .	7
<b>3</b>	<b>基本的な使用方法</b>	<b>9</b>
3.1	Goose の扱うプログラム記述 . . . . .	9
3.2	プログラムの開発手順 . . . . .	10
3.3	チュートリアル 1 – 数値積分 . . . . .	11
3.4	チュートリアル 2 – 多粒子間重力相互作用 . . . . .	14
3.5	サンプルプログラム . . . . .	17
<b>4</b>	<b>Goose 疑似命令</b>	<b>18</b>
4.1	Goose for 疑似命令詳細 . . . . .	18
4.1.1	変数の入出力型 . . . . .	19
4.1.2	オプション引数 . . . . .	20
4.2	Goose func 疑似命令詳細 . . . . .	22
<b>5</b>	<b>コンパイル可能な C 言語記述</b>	<b>24</b>
5.1	代入文 . . . . .	24
5.2	if 文 . . . . .	24
5.3	条件演算子 . . . . .	24
5.4	for 文 . . . . .	25
5.5	return 文 . . . . .	25
5.6	コンパイル不可能な記述の処置 . . . . .	26
<b>6</b>	<b>Goose 詳細</b>	<b>27</b>
6.1	goosecc コマンドライン引数 . . . . .	27
6.2	goosecc の内部動作 . . . . .	27
6.3	ハードウェアアクセラレータのアーキテクチャ . . . . .	28
6.3.1	GRAPE-DR . . . . .	29
<b>7</b>	<b>利用許諾</b>	<b>34</b>
<b>8</b>	<b>謝辞</b>	<b>34</b>



# 1 本文書の概要

この文書では Goose ソフトウェアパッケージの使用方法を説明します。Goose は株式会社 K&F Computing Research 開発の、SIMD 型計算機向けソフトウェア開発環境です。C 言語などの高級言語で記述された PC 上で動作するプログラムを、ソースコードへの変更を極力抑えたまま、SIMD 型ハードウェアアクセラレータ上で動作させることを目的としています。現段階 (version 1.0.1 $\alpha$ ) では、高級言語として C 言語を、ハードウェアアクセラレータとして当社 GRAPE-DR をサポートしています。近日中に Fortran、GPU (AMD, NVIDIA)、SSE テクノロジー、GRAPE-7 への対応を予定しています。

Goose はあえて高級言語の言語仕様を完全にはサポートしない、ドメイン特化 (domain specific) 型の開発環境です。SIMD 型アクセラレータ上での実行に適した文法記述のみを処理し、それ以外の記述は通常のコンパイラ (gcc 等) へ渡して PC 向けにコンパイルします。この切り分けによって、PC 上で動作する既存のプログラムをアクセラレータへ移植する際に必要となるソースコードの変更を、最小限に抑えられます。

以降、第 2 章では Goose ソフトウェアパッケージ (以降「本パッケージ」と呼びます) のインストール方法を説明します。第 3 章では基本的な使用方法を、具体例を用いたチュートリアル形式で説明します。第 4 章では Goose 疑似命令の詳細を説明します。第 5 章では Goose がサポートする C 言語記述について説明します。第 6 章では Goose コンパイラやハードウェアアクセラレータの詳細について説明します。

なお以降では、本パッケージのルートディレクトリ (/パッケージを展開したディレクトリ/goosepkg バージョン番号/) を \$goosepkg と表記します。

## 2 インストール

### 2.1 準備

本パッケージは以下のソフトウェアに依存しています。インストール作業の前に、これらの動作環境を整えて下さい。

- C コンパイラ (gcc version 4.1.0 以降を推奨)  
<http://gcc.gnu.org/>
- Ruby (version 1.8.5 以降を推奨)  
<http://www.ruby-lang.org/>
- Perl (version 5.8.8 以降を推奨)  
<http://www.perl.com/>
- LSUMP (中間言語コンパイラ。会津大学中里直人氏開発)  
<http://www.kfcr.jp/goose.html>
- VSM (GRAPE-DR 用アセンブラ。国立天文台牧野淳一郎氏開発)  
<http://www.kfcr.jp/goose.html>
- GRAPE Software Package (version 1.1.0 以降を推奨)  
<http://www.kfcr.jp/goose.html>

## 2.2 パッケージの展開と環境変数の設定

ソフトウェアパッケージ `goosepkgnnn.tar.gz` を展開してください (`nnn` はバージョン番号)。パッケージには以下のファイルが含まれています:

<code>00readme-j</code>	パッケージ概要。
<code>00readme</code>	パッケージ概要英語版。
<code>doc/</code>	ユーザガイド、その他の文書。
<code>scripts/</code>	パッケージ管理ユーティリティ。
<code>bin/goosecc</code>	Goose 疑似命令を埋め込んだ C 言語のソースコードから、ハードウェアアクセラレータ用の実行ファイルを生成するツール。
<code>include/</code>	ヘッダファイル。
<code>lib/</code>	ライブラリ。
<code>singutil/</code>	GRAPE-DR 制御用ライブラリ。
<code>sample/</code>	Goose でコンパイル可能なアプリケーションプログラムの例。
<code>init/</code>	<code>sample/</code> 内のプログラムが使用するデータファイル。
<code>misc/</code>	環境変数設定のサンプル、 <code>init/</code> 内のデータを生成するプログラム等。

以下の環境変数を設定してください。

<code>LSUMPPATH</code>	: <code>LSUMP</code> のインストールされているパス。
<code>VSMPATH</code>	: <code>VSM</code> のインストールされているパス。
<code>GRAPEPKGPATH</code>	: <code>GRAPE Software Package</code> のインストールされているパス。

例 `sh` を使用する場合:

```
kawai@localhost>export LSUMPPATH=/home/kawai/src/lsumppkg
kawai@localhost>export VSMPATH=/home/kawai/src/vsm
kawai@localhost>export GRAPEPKGPATH=/home/kawai/src/grapepkg
```

例 `csh` を使用する場合:

```
kawai@localhost>setenv LSUMPPATH /home/kawai/src/lsumppkg
kawai@localhost>setenv VSMPATH /home/kawai/src/vsm
kawai@localhost>setenv GRAPEPKGPATH /home/kawai/src/grapepkg
```

以上の環境変数設定を行う `sh` 用、`csh` 用スクリプトのサンプルが、

```
$goosepkg/misc/rc/genenv.sh
$goosepkg/misc/rc/genenv.csh
```

に用意されています。自身の環境に合わせて修正のうえお使い下さい。また、以下の環境変数も使用できますが、必ずしも設定する必要はありません。

<code>CC</code>	: 使用する C コンパイラを指定します。
<code>CFLAGS</code>	: C コンパイラへ渡す引数を指定します。

## 2.3 インストールスクリプトの実行と動作チェック

ディレクトリ `$goosepkg` へ移動し、コマンド `$goosepkg/script/install` を実行してください。各種ライブラリや、サンプルコードの実行に必要なデータなどが生成されます。

```
kawai@localhost>./script/install
cc -O3 -c singutil.c -I/home/kawai/grapepkg/include \
-L/home/kawai/grapepkg/lib -fopenmp
ar -r libsing.a singutil.o
ar: creating libsing.a
ranlib libsing.a
cc      -c -o mkdist.o mkdist.c
cc      -c -o util.o util.c
cc -o mkdist mkdist.o util.o -lm
n: 1024  output file: pl1k
distribution: Plummer

...

done
```

以上でインストールは完了です。インストールが正常に行われていれば、本パッケージ付属の各種のサンプルプログラムをコンパイルできるはずです。例えば重力多体シミュレーションを行うプログラム `s9` をコンパイルするには、ディレクトリ `$goosepkg/sample/s9/` へ移動し、コマンド `make all` を実行します。

```
kawai@localhost>pwd
/home/kawai/src/goosepkg/sample/s9
kawai@localhost>make all
../../bin/goosecc -v3 --goose-arch gdr -o s9 sticky9.c -lm -O3
$LSUMPPATH                : /home/kawai/src/lsump
$VSMPATH                  : /home/kawai/src/vsm
$GRAPEPKGPATH              : /home/kawai/src/grapepkg

...

=====
executable file generated successfully.
=====
cc -O3 sticky9.c -o s9_host -lm -fopenmp
kawai@localhost>ls
Makefile  goosetmp  inputpara9  s9  s9_host  sticky9.c
```

コンパイルが正常に完了すると、二つの実行ファイル `s9` および `s9_host` が生成されます。前者はハードウェアアクセラレータを使用して計算を行うプログラム、後者はホスト計算機だけで計算を行うプログラムです。



## 3 基本的な使用方法

### 3.1 Goose の扱うプログラム記述

Goose は C 言語などの高級言語で記述されたプログラムをハードウェアアクセラレータ向けのコードにコンパイルしますが、すべての記述をコンパイル対象とするわけではありません。SIMD 型アクセラレータ上での実行に適した記述のみを処理の対象とします。それ以外の記述は通常のコンパイラ (gcc 等) へ渡して、ホスト計算機向けにコンパイルします。ここでいう「SIMD 型アクセラレータ上での実行に適した記述」とは、

- (a) 並列性が高く、
- (b) 外部 (プロセッサチップ外部のメモリやホスト計算機) との通信量が計算量に対して相対的に少ない、

という条件を満たす計算に関する記述を意味します。

Goose が処理対象とするのは、計算結果  $\vec{r}_i$  が入力パラメタ  $\vec{x}_i$  を用いて

$$\vec{r}_i = f(\vec{x}_i) \quad (1)$$

という形に表せ、 $\vec{r}_i$  を求める計算を異なる  $i$  に対して並列に行えるタイプの計算です。そのような計算のなかでも特に、

$$\vec{r}_i = \sum_j f(\vec{x}_i, \vec{y}_j) \quad (2)$$

という形に表せるものの処理に最適化されています。式 (1) の形に表せる計算は、 $i$  の総数が十分に大きければ「(a) 並列度が高」という条件を満たします。さらにそれが式 (2) の形に表せる場合には、「(b) 外部との通信量が計算量に対して相対的に少ない」という条件をも満たします。これは多数の  $i$  に対する計算を、同一の  $\vec{y}_j$  を使用して行えるからです。そのため  $i$  ごとに別個の  $\vec{y}_i$  が必要な場合に比べ、通信量が少なくてすみます。式 (1) の形には表せるが、式 (2) の形には表せないタイプの計算も扱えますが、そのような計算に対しては、ハードウェアアクセラレータの持つ演算能力を十分に活かしきれない場合があります。

なお、式 (1)、式 (2) における  $i$  や  $j$  に、具体的にどの程度の並列度が必要とされるかは、ハードウェアアクセラレータのアーキテクチャに依存します。GRAPE-DR で式 (2) 型の計算を行う場合には、 $i$  の並列度が 128 以上、 $j$  の並列度が 16 以上あれば、ハードウェアの持つすべてのプロセッサ要素を利用して計算を行えます (節 6.3.1 参照)。並列度がそれよりも低い場合には、一部のプロセッサ要素は計算に利用されず、遊んでしまうこ

とになります。

例：多数の質点間に働く重力相互作用の計算は、式 (2) の形で表現できます。質点  $i$  に他のすべての質点から働く重力の総和  $\vec{f}_i$  は、質点の空間座標  $\vec{x}_i$  と質量  $m_i$  を用いて下式で表せます ( $G$  は万有引力定数)。

$$\vec{f}_i = \sum_{j \neq i} \frac{G m_i m_j (\vec{x}_j - \vec{x}_i)}{|\vec{x}_j - \vec{x}_i|^3} \quad (3)$$

C 言語を用いると、式 (1) 型の計算は様々な記述で表現できます。しかし簡単のため、Goose ではこの形式の計算を for 文によるループで表現した記述のみを扱います。また、式 (2) 型の場合は、入れ子になった 2 重 for ループで表現した記述のみを扱います。他の方法、例えば while 文を用いた記述は、Goose の処理対象にはなりません。

例：式 (3) は for 文による 2 重ループで表現できます。

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        if (i != j) {
            dx[0] = x[j][0] - x[i][0];
            dx[1] = x[j][1] - x[i][1];
            dx[2] = x[j][2] - x[i][2];
            r2 = dx[0]*dx[0] + dx[1]*dx[1] + dx[2]*dx[2];
            mmr3 = G * m[i] * m[j] / (sqrt(r2) * r2);
            f[i][0] += mmr3 * dx[0];
            f[i][1] += mmr3 * dx[1];
            f[i][2] += mmr3 * dx[2];
        }
    }
}
```

## 3.2 プログラムの開発手順

ユーザが直接使用するコマンドは、原則的には `$goosepkg/bin/goosecc` のみです。`goosecc` は C 言語のソースコードから、ハードウェアアクセラレータ用の実行ファイルを生成します。

ユーザは以下の手順でアプリケーションプログラムの開発を行います：

- (手順 1) アプリケーションプログラムを C 言語で記述し、ホスト 計算機 (PC) 上で動作確認を行います。この際、SIMD 型アクセラレータでの実行に適していると予想される箇所 (節 3.1 参照) は、while 文や do while 文ではなく for 文を使用して記述します。計算結果が 2 重ループのカウンタ  $i, j$  に関して対称な場合 (作用反作用則を満たす粒子間中心力など) には、その性質を利用して計算量を減らす工夫を行うことが

あります (下記の例)。しかしこのような記述は Goose では扱えませんので避けて下さい。

例：作用反作用則を利用して式 (3) の計算量を減らす記述。Goose では扱えない。

```
for (i = 0; i < n; i++) {
    for (j = i+1; j < n; j++) {
        dx[0] = x[j][0] - x[i][0];
        dx[1] = x[j][1] - x[i][1];
        dx[2] = x[j][2] - x[i][2];
        r2 = dx[0]*dx[0] + dx[1]*dx[1] + dx[2]*dx[2];
        mmr3 = G * m[i] * m[j] / (sqrt(r2) * r2);
        f[i][0] += mmr3 * dx[0];
        f[i][1] += mmr3 * dx[1];
        f[i][2] += mmr3 * dx[2];
        f[j][0] -= mmr3 * dx[0];
        f[j][1] -= mmr3 * dx[1];
        f[j][2] -= mmr3 * dx[2];
    }
}
```

(手順 2) 可能ならば、手順 1 で記述した for 文に OpenMP 疑似命令 (`#pragma omp parallel for`) を付加し、スレッド並列化を行います。この過程は必須ではありませんが、並列動作時の演算結果の正しさや性能向上の程度を確認する手法として効果的です。OpenMP の使用法や詳細については <http://openmp.org/> などを参照してください。

(手順 3) 手順 1 で記述した for 文に、新たに Goose 疑似命令 (`#pragma goose parallel for`) を付加し、`goosecc` を用いてコンパイルすると、実行ファイルが生成されます。この実行ファイルを実行すると、Goose 疑似命令を付加した for 文がハードウェアアクセラレータ上で実行されます。それ以外の部分はホスト計算機上で実行されます。

### 3.3 チュートリアル 1 – 数値積分

本節と次節では、Goose 疑似命令の記述方法とプログラムのコンパイル手順を、具体例を用いて説明します。

本節ではサンプルプログラムとして `$goosepkg/sample/midpoint/midpoint.c` を使用します。このプログラムは中点法による数値積分で  $\int_0^1 4/(1+x^2)dx$  を計算します (解析解は  $\pi$ )。このプログラムの中核は以下の for ループです。

```
for(i=0;i<n;i++) {
    x = (i+0.5)*dx;
    sum += integrand(x)*dx;
}
```

積分区間  $[0, 1]$  を  $n$  分割し、各区間における被積分関数の値と  $dx$  の積を求め、それを変

数 `sum` へ積算します。被積分関数は C 言語の関数 `integrand()` としてプログラム冒頭で定義されています。

このプログラムを、通常の C コンパイラを用いてホスト 計算機向けにコンパイルしてみましょう。

```
kawai@localhost>cd $goosepkg/sample/midpoint/  
kawai@localhost>cc midpoint.c -o midpoint_host
```

生成された実行ファイル `midpoint_host` を実行すると、以下の出力が得られます。

```
kawai@localhost>./midpoint_host  
n:16384 sum:3.14159265390022e+00 sum/M_PI-1.0:9.881305e-11
```

ホスト 計算機上での動作を確認できたので、次はこのプログラムをハードウェアアクセラレータ向けにコンパイルすることを考えましょう。前述の `for` 文の直前に、Goose 疑似命令 `#pragma goose parallel for` が付加されていることに着目してください (以降この疑似命令を「Goose `for` 疑似命令」と呼びます)。

```
#pragma goose parallel for icnt(i) result(sum) ip(i) shared_ro(dx)  
for(i=0;i<n;i++) {  
    x = (i+0.5)*dx;  
    sum += integrand(x)*dx;  
}
```

プログラム内の、Goose `for` 疑似命令を付加した箇所が、ハードウェアアクセラレータ上での実行対象となります。他の部分はホスト 計算機上で実行されます。

Goose `for` 疑似命令をもう少し詳しく見てゆきましょう。疑似命令への引数として、`icnt(i)` などが与えられています:

- 引数 `icnt(i)` は、変数 `i` を `for` 文のループカウンタとして使用することを示しています。
- 引数 `result(sum)` は、変数 `sum` の値を計算結果としてアクセラレータからホスト 計算機へ回収する必要があることを示しています。
- 引数 `ip(i)` は、計算の入力パラメタとして変数 `i` の値をホスト 計算機からアクセラレータへ送信する必要があることを示しています。
- 引数 `shared_ro(dx)` も `ip()` と同様に、変数 `dx` の値をアクセラレータへ送信する必要があることを示しています。引数 `ip()` との違いは、`shared_ro()` で指定した変数 `dx` の値はループ内で不変であるのに対し、`ip()` で指定した `i` の値はカウンタ `i` に依存してループ内で変化することです。

後述 (節 4.1.1) するように、実際にはこれらの引数すべてが必須というわけではありません。しかし Goose の使い方に慣れるまでは、すべてのループカウンタと、`private` 型を

除くすべての変数の「入出力型」(節 4.1.1) を明示することをお薦めします。

次に被積分関数の定義に着目してください。

```
#pragma goose func
double integrand(double x)
{
    double s;
    s = 4.0/(1.0+x*x);
    return s;
}
```

関数定義に Goose 疑似命令 `#pragma goose func` が付加されています (以降この疑似命令を「Goose func 疑似命令」と呼びます)。この疑似命令を付加すると、Goose for 疑似命令を付加した for 文の内部から関数 `integrand()` を呼び出せるようになります。なお、Goose func 疑似命令を付加するためには、関数は引数を一つだけとり、なおかつ `return` 文を用いて明示的に返り値を返す必要があります。`return` 文の無い関数や、返り値の型として `void` 型を持つ関数は、Goose では使用できません。

以上で Goose 疑似命令を理解できたので、実際にコンパイルを行きましょう。コンパイルには `$goosepkg/bin/goosecc` を使用します。

ディレクトリ `$goosepkg/sample/midpoint/` 内で、以下のようにコマンドライン引数を与えて `goosecc` を実行してください (`goosecc` のコマンドライン引数詳細については節 6.1 を参照してください)。

```
kawai@localhost>../bin/goosecc midpoint.c -o midpoint
=====
processing midpoint.c.
=====
Info      : Goose::GforHandler : No j-loop found. The entire i-loop \
body is used as a kernel.
Info      : Goose::GforHandler : Recognized as an ip var : i
Info      : Goose::GforHandler : Recognized as a shared_ro var : dx
...
=====
executable file generated successfully.
=====
```

コンパイルに成功すると、実行ファイル `midpoint` が生成されます。ハードウェアアクセラレータを稼働させた状態で `midpoint` を実行すると、前述の Goose for 疑似命令を付加した for ループがアクセラレータ上で実行されます。他の部分は `midpoint_host` と同様にホスト 計算機上で実行されます。実行の結果、`midpoint_host` と同様の出力が得られます。

### 3.4 チュートリアル 2 – 多粒子間重力相互作用

本節で使用するサンプルプログラムは\$goosepkg/sample/gravity/gravity.c です。このプログラムは多粒子系において、質点間の重力相互作用によって各質点を得る加速度  $a_i$  を計算します。本来の重力相互作用は式 (3) の形で表現されますが、ここでは万有引力定数を 1 にスケールリングし、また数値計算上の問題を回避するためにソフト ニングパラメタ  $\epsilon$  を導入します。すると計算すべき式は

$$\vec{a}_i = \sum_{j \neq i} \frac{m_j (\vec{x}_j - \vec{x}_i)}{(|\vec{x}_j - \vec{x}_i|^2 + \epsilon^2)^{3/2}} \quad (4)$$

と表されます。サンプルプログラム中の 2 重 for ループはこの式を表現しています。

```
#pragma goose parallel for icnt(i) jcnt(j) \
ip(x[i][0..2]) jp(x[j][0..2],m[j]) result(a[i][0..2],pot[i])\
precision ("double")
for(i=0;i<n;i++) {
    for(k=0;k<3;k++) a[i][k] = 0.0;
    pot[i] = 0.0;
    for(j=0;j<n;j++) {
        for(k=0;k<3;k++) dx[k] = x[j][k] - x[i][k];
        r2 = dx[0]*dx[0] + dx[1]*dx[1] + dx[2]*dx[2] + eps2;
        rinv = rsqrt(r2);
        mrinv = m[j]*rinv;
        mr3inv = mrinv*rinv*rinv;
        for(k=0;k<3;k++) a[i][k] += mr3inv * dx[k];
        pot[i] += -mrinv;
    }
}
get_cputime(&lt,&st);
total_time += lt;
for(i=0;i<n;i++) pot[i] += m[i]/sqrt(eps2);
```

本サンプルのように計算対象が式 (2) の形式をとる場合には、プログラム上では 2 重の for ループを用いて記述します (節 3.1 参照)。プログラム内の、Goose for 疑似命令を付加した 2 重ループが、ハードウェアアクセラレータ上での実行対象となります。他の部分はホスト 計算機上で実行されます。

Goose for 疑似命令への引数として、以下のものが使用されています (一部は節 3.3 説明したものも含まれます)。

- 引数 icnt(i) は、変数 i を外側の for ループのカウンタとして使用することを示しています。同様に、引数 jcnt(j) は変数 j を外側の for ループのカウンタとして使用することを示しています。
- 引数 ip(x[i][0..2]) は、計算の入力パラメタとして変数 x[i][0..2] の値をホスト 計算機からアクセラレータへ送信する必要があることを示しています。ここ

で配列インデックス中の 0..2 は 0, 1, 2 の 3 個のインデックスを表します。つまり `x[i][0..2]` は `x[i][0]`, `x[i][1]`, `x[i][2]` と同値です。この記法は Goose 疑似命令文中でのみ使用できます。

- 引数 `jp(x[j][0..2],m[j])` は、計算の入力パラメタとして変数 `x[j][0..2]` と `m[j]` の値を、ホスト 計算機からアクセラレータへ送信する必要があることを示しています。引数 `ip()` との違いは、`jp()` で指定した変数の値は内側のループ内で変化するのに対し、`ip()` で指定した変数の値は内側のループ内では不変であることです。
- 引数 `result(a[i][0..2],pot[i])` は、変数 `a[i][0..2]` と `pot[i]` の値を計算結果としてアクセラレータからホスト 計算機へ回収する必要があることを示しています。
- 引数 `precision("double")` は、変数の数値フォーマットとして 64-bit 浮動小数点形式を用いることを示しています。C 言語での変数の型宣言は無視されます。指定が無い場合には 64-bit 浮動小数点形式を用います。指定できる数値フォーマットについては節 4.1.2 を参照してください。

以下の点に注意してください。

- Goose func 疑似命令を付加した関数と同名の関数が Goose の組み込み関数としてあらかじめ用意されている場合には、関数の再定義は行われずに、組み込み関数が使われます (節 4.2)。例えば `rsqrt()` は組み込み関数として用意されているため、プログラム冒頭の定義:

```
double rsqrt(double r2)
{
    return 1.0/sqrt(r2);
}
```

は使用されません。

- 自己代入演算子 `-=` は使用できません。 `+=` で代用してください。

例 : `pot[i] += -mrinv`

- 2 重 for ループのさらに内側にある for ループはアンロールされます。例えば

```
for(k=0;k<3;k++) dx[k] = x[j][k] - x[i][k];
```

という記述は、

```
dx[0] = x[j][0] - x[i][0];  
dx[1] = x[j][1] - x[i][1];  
dx[2] = x[j][2] - x[i][2];
```

と同値です。なおループ範囲が有限でない for 文や、ループ範囲がコンパイル時に確定しない for 文は、コンパイル時に警告メッセージが出力され、無視されます (節 5.4)。

- プログラム中で数学関数 `sqrt()` を使用しているため、コンパイルにはコマンドラインオプション `-lm` が必要です。

```
kawai@localhost>cd $goosepkg/sample/gravity/  
kawai@localhost>../bin/goosecc gravity.c -o gravity -lm
```



### 3.5 サンプルプログラム

ディレクトリ `$goosepkg/sample/` 内に、`goosecc` でコンパイル可能なアプリケーションプログラムの記述例があります。Goose 疑似命令の使い方などの参考にしてください。

2 重 for ループ (式 (2) 型)	
<code>gravity/</code>	重力相互作用を計算します (節 3.4 のチュートリアルで使用)。
<code>gravity_cutoff/</code>	P <sup>3</sup> M 法向けカットオフの入った重力相互作用を 周期境界条件下で計算します。
<code>hermite/</code>	重力相互作用と、その時間微分を計算します。
<code>tree/</code>	Barnes-Hut ツリー法を用いて重力相互作用を計算します。
<code>s9/</code>	重力多体シミュレーションを行います (leap-frog 公式、共通時間刻み)。
<code>s8/</code>	重力多体シミュレーションを行います (4 次の Hermite 公式、独立時間刻み)。
<code>vdw/</code>	ファンデルワールス力を計算します (Lennard-Jones ポテンシャル)。
<code>sph/</code>	SPH 法における粒子への加速度を計算します (Spline カーネル、人工粘性無し)。
1 重 for ループ (式 (1) 型)	
<code>midpoint/</code>	中点法を用いて $\int_0^1 4/(1+x^2)dx$ を計算します (節 3.3 のチュートリアルで使用)。
<code>mesh1d/</code>	1 次元空間の格子点各点において、隣接する点の値との間で 重み付き平均をとります。

各ディレクトリ内でコマンド `make all` を実行すると、ハードウェアアクセラレータ向けの実行ファイルが生成されます。アクセラレータを使用せずにホスト計算機のみで計算を行う実行ファイルも、併せて生成されます。

## 4 Goose 疑似命令

Goose の疑似命令は以下の 2 種類に大別されます:

**Goose for 疑似命令** : `#pragma goose parallel for` [*optional-arguments*]

for 文の直前に記述すると、その for 文をハードウェアアクセラレータ用のコードに置き換えます。

**Goose func 疑似命令** : `#pragma goose func`

関数定義の直前に記述すると、Goose for 疑似命令を付加した for 文の内部からその関数を呼び出せるようになります。

記法上の注意: Goose 疑似命令は 1 行で記述してください。複数行にまたがる場合には、バックスラッシュを使用して改行コードをエスケープしてください。

```
例 : #pragma goose \  
      parallel for
```

### 4.1 Goose for 疑似命令詳細

Goose for 疑似命令は、その直後の for 文をハードウェアアクセラレータ用のコードに置き換えます。典型的には、以下に示すような 2 重 for ループを置換対象としますが、1 重ループも置換できます。3 重以上のループの場合には、内側のループはアンロールされます。

```
例 : #pragma goose parallel for  
    for (i = 0; i < ni; i++) {      // 外側の 'for'。  
        for (j = 0; j < nj; j++) {  // 内側の 'for'。  
            for (k = 0; k < 3; k++) { // この 'for' はアンロールされる。  
                dx[k] = x[j][k] - x[i][k];  
            }  
            r2 = dx[0]*dx[0] + dx[1]*dx[1] + dx[2]*dx[2] + eps2;  
            rinv = rsqrt(r2);  
            mf = m[j]*rinv*rinv*rinv;  
            for (k = 0; k < 3; k++) { // この 'for' はアンロールされる。  
                a[i][k] += mf * dx[k];  
            }  
            p[i] -= m[j] * rinv;  
        }  
    }
```

アクセラレータ上の計算に使用される変数の型はすべて double 型 (あるいは後述の `#pragma goose parallel for precision` で指定した型) となります。C 言語での型宣言は無視されます。

### 4.1.1 変数の入出力型

Goose for 疑似命令 を付加した for ループ内で使用する変数は、ハードウェアアクセラレータとホスト 計算機との間で受渡しを行う必要があります。必要とされる受渡しの方法に応じて、各変数には「入出力型」という属性が定義されます。

#### 入出力型一覧

型名	用法
ip	計算の入力パラメタのうち、 $i$ に依存する変数です (式 (2) の $\vec{x}_i$ )。 ホスト 計算機からアクセラレータへ送信されます。
jp	計算の入力パラメタのうち、 $j$ に依存する変数です (式 (2) の $\vec{x}_j$ )。 ホスト 計算機からアクセラレータへ送信されます。
shared_ro	計算の入力パラメタのうち、 $i$ や $j$ に依らない変数です。 ホスト 計算機からアクセラレータへ送信されます。
private	計算中に一時的に使用される中間変数です。 アクセラレータ上だけで使用され、ホスト 計算機へは回収されません。
result	計算結果です (式 (2) の $\vec{r}_i$ )。 アクセラレータからホスト 計算機へ回収されます。

変数の入出力型は以下の規則に従って goosecc により自動推論されますが、後述 (節 4.1.2) のように Goose for 疑似命令への引数によって明示的に指定することも可能です。

#### 入出力型の推論規則:

外側と内側の for ループのループカウンタを、それぞれ  $i$ 、 $j$  とするとき、

- $i$  でインデクスされた変数は ip 型と認識されます。ただしその変数が左辺値として使用されている場合には、result 型と認識されます。
- $j$  でインデクスされた変数は jp 型と認識されます。
- 以上に該当しない変数のうち、左辺値として使用されているものは、private 型と認識されます。ただし Goose for 疑似命令への引数によって明示的に指定すると (節 4.1.2)、result 型として認識させることが可能です。プログラム `$goosepkg/sample/midpoint/midpoint.c` 中の変数 `sum` はそのような使われかたをしています (節 3.3 参照)。
- 右辺値としてしか使用されていない変数は、shared\_ro 型と認識されます。

### 4.1.2 オプション引数

Goose for 疑似命令 は、変数の入出力型指定を含む多くのオプション引数をとることができます。以下にオプションの一覧を示します。

オプション引数一覧:

書式	機能
<code>icnt(var)</code>	外側の for 文のループカウンタとして変数 <i>var</i> を指定します。指定がない場合には変数 <code>i</code> を使用します。
<code>jcnt(var)</code>	内側の for 文のループカウンタとして変数 <i>var</i> を指定します。指定がない場合には変数 <code>j</code> を使用します。
<code>result(var, ...)</code>	変数 <i>var</i> の入出力型を <code>result</code> 型に指定します。
<code>ip(var, ...)</code>	変数 <i>var</i> の入出力型を <code>ip</code> 型に指定します。
<code>jp(var, ...)</code>	変数 <i>var</i> の入出力型を <code>jp</code> 型に指定します。
<code>private(var, ...)</code>	変数 <i>var</i> の入出力型を <code>private</code> 型に指定します。
<code>shared_ro(var, ...)</code>	変数 <i>var</i> の入出力型を <code>shared_ro</code> 型に指定します。
<code>precision("prec")</code>	変数の数値フォーマットとして、" <i>prec</i> " を指定します。指定できる値は " <code>double</code> " (64-bit 浮動小数点) と " <code>double-single</code> " (加減算 64-bit 浮動小数点、乗算 32-bit 浮動小数点) です。" <code>single</code> " (32-bit 浮動小数点) および " <code>quadruple</code> " (128-bit 浮動小数点) にも対応の予定です。指定がない場合には " <code>double</code> " を使用します。
<code>asmfile("filename")</code>	アセンブラに渡すアセンブリコードを指定します。goosecc が出力したコード ( <i>e.g.</i> , <code>foo_0.vsm</code> ) の代わりに、ハンドチューニングしたアセンブリコードを使用する場合に指定します。

(続く)

## オプション引数一覧 (続き):

書式	機能
<code>njp_write(<i>var</i>)</code>	指定すると、jp 型データのうち、インデクス <i>j</i> の値が <i>var</i> 未満のものだけをアクセラレータに送信するようになります。複数回の計算にまたがって jp 型データの一部を使いまわせる場合に、データ転送量を減らすことができます ( <code>\$goosepkg/sample/s8/</code> 参照)。指定がない場合にはすべての jp 型データを送信します。
<code>nip_pack(<i>uint</i>)</code>	ハードウェアアクセラレータのプロセッサ要素 1 個が処理する演算量を指定します。指定のない場合には、ひとつのプロセッサ要素が <i>nvec</i> 個の <i>i</i> に対する計算を実行します (ここで <i>nvec</i> はベクトルプロセッサのループ長) が、指定すると、 <i>uint</i> × <i>nvec</i> 個の <i>i</i> に対する計算を実行するようになります。2~5 程度の値を指定すると、ホスト 計算機との通信オーバーヘッドが隠蔽され、実効性能が向上する場合があります。

入出力型指定に関する注意:

- 複数の変数の入出力型を指定するには、カンマで区切って列挙します。

例 : `ip(a, b, c)`

- 入出力型指定は、より正確には、変数でなく、変数によって参照される値に対して行います。従って例えば、ある変数配列のインデクスに別の変数を使用した表現や、ポインタ変数のデリファレンス、構造体のメンバ参照なども指定できます。

```
例 : double x[255];
      int index[255], j;
      struct a_struct_t a_struct, *a_struct_p;
      ip(x[0]);                // 配列型変数の要素
      ip(x[i]);                // 配列型変数のインデクスに変数を使用
      ip(x[index[i]]);         // 配列のインデクスに他の配列の要素を使用
      jp(*(x+j));              // ポインタのデリファレンス
      jp(a_struct.a_member);    // 構造体メンバの参照
      jp(a_struct_p->a_member); // 構造体へのポインタからのメンバ参照
```

- 配列型の変数を指定する際、配列の範囲を表す演算子 `...` を使用できます。

例 : `ip(x[i][0..2])`

という記述は

`ip(x[i][0], x[i][1], x[i][2])`

と同値です。

- ハードウェアアクセラレータとホスト 計算機はアドレス空間を共有していないため、アドレスの受渡しは行えません。従ってアドレス参照に対する入出力型の指定は行えません。

```
例 : double x, y[255];
      int i;
      ip(&x);                  // NG
      ip(x);                   // OK
      ip(y);                   // NG
      ip(y[0]);                // OK
      ip(y[i]);                // OK
      ip(*(y + i));            // OK
```

## 4.2 Goose func 疑似命令詳細

Goose func 疑似命令 `#pragma goose func` を関数定義の直前に記述すると、Goose for 疑似命令を付加した `for` 文の内部からその関数を呼び出せるようになります。関数は引数を一つだけとり、なおかつ `return` 文を用いて明示的に返り値を返す必要があります (`void` 型を返すことはできません)。引数および返り値の型はすべて `double` 型 (あるいは

#pragma goose parallel for precision で指定した型) となります。C 言語での型宣言は無視されます。

```
例 : #pragma goose func
      static double rsqrt(double r2)
      {
          int i;
          double x0 = 1.0;
          for (i = 0; i < 4; i++) {
              x0 = 0.5*x0*(3.0 - r2*x0*x0);
          }
          return x0;
      }
```

なお Goose func 疑似命令を付加した関数と同名の関数が Goose の組み込み関数としてあらかじめ定義されている場合には、関数の再定義は行われずに、組み込み関数が使用されます。組み込み関数には以下のものがあります:

- `rsqrt(x)`: 引数  $x$  の逆平方根 ( $1/\sqrt{x}$ ) を返します。

## 5 コンパイル可能な C 言語記述

goosecc はすべての C 言語記述をコンパイルできるわけではありません。本章ではコンパイル可能な記述について説明します。

### 5.1 代入文

ほとんどの代入文はコンパイル可能です。

```
例 : r2 = dx[0]*dx[0] + dx[1]*dx[1] + dx[2]*dx[2];  
      a[i][k] += mf * dx[k]; // 自己代入。  
      double eps = 0.01;    // 初期値つき変数宣言。  
      rinv = rsqrt(r2);      // 関数の返り値の代入。
```

代入表現を値として評価することも可能です。

```
例 : x = y = z = 0.0;  
      rinv = rsqrt(r2 = dx[0]*dx[0] + dx[1]*dx[1] + dx[2]*dx[2]);
```

### 5.2 if 文

if 文は、条件式が以下の形式にマッチするもののみコンパイル可能です。

*var0 rel\_op var1*

ここで *var0*, *var1* は変数あるいは数値リテラル、*rel\_op* は比較演算子 `<`, `<=`, `>`, `>=`, `==`, `!=` のいずれかです。

```
例 : if (i != j) {  
      f[i] += fij;  
    }
```

ネストも可能です。

```
例 : if (q > 2.0) {  
      wkernel = 0.0;  
    }  
    else if (q > 1.0) {  
      wkernel = 0.25*(2.0-q)*(2.0-q)*(2.0-q);  
    }  
    else {  
      wkernel = 1.0-1.5*q*q+0.75*q*q*q;  
    }
```

### 5.3 条件演算子

条件演算子は、if 文と同様に、条件式が以下の形式にマッチするもののみコンパイル可能です。



*var0 rel\_op var1*

ここで *var0*, *var1* は変数あるいは数値リテラル、*rel\_op* は比較演算子 *<*, *<=*, *>*, *>=*, *==*, *!=* のいずれかです。

例: `wkernel = q > 2.0 ? 0.0 : 0.25*(2.0-q)*(2.0-q)*(2.0-q);`

ネスト も可能です。

例: `wkernel = q > 2.0 ? 0.0 :  
q > 1.0 ? 0.25*(2.0-q)*(2.0-q)*(2.0-q) :  
1.0-1.5*q*q+0.75*q*q*q;`

## 5.4 for 文

Goose for 疑似命令 を付加した 2 重 for ループのさらに内側にある for 文、および Goose func 疑似命令を付加した関数定義内の for 文は、アンロールされます。ループがネストしている場合には、再帰的にすべてのループがアンロールされます。

ループ範囲とループカウンタの増分が定数の場合のみアンロール可能です。それ以外の場合にはコンパイル中に警告メッセージが出力され、for 文全体が無視されます。より厳密には、以下の形式にマッチする for 文のみがアンロールの対象となります。

`for (var = init_val ; var rel_op final_val ; var = var + inc_val)`

ここで *var* は変数、*init\_val*, *final\_val*, *inc\_val* は整数リテラル、*rel\_op* は比較演算子 *<* または *<=* です。式 `var = var + inc_val` を `var += inc_val` と表記することも可能です。*inc\_val* が 1 の場合には、`var++` と表記することも可能です。

例 : `for (k = 0; k < 3; k++) {  
 a[i][k] += mf * dx[k];  
}`

この for 文は、以下の記述にアンロールされます。

```
a[i][0] += mf * dx[0];  
a[i][1] += mf * dx[1];  
a[i][2] += mf * dx[2];
```

## 5.5 return 文

Goose func 疑似命令を付加した関数定義内で使用できます。Goose for 疑似命令を付加した for 文内では使用できません。

## 5.6 コンパイル不可能な記述の処置

goosecc はコンパイル不可能な記述を見つけると警告メッセージを出力しますが、コンパイル自体は続行します。コンパイル不可能な記述は単に無視します。そのため、例えばホスト 計算機上でのデバッグに使用した `printf()` 関数呼び出しがプログラム中に残っていても、ハードウェアアクセラレータ上の計算に影響を与えることはありません。

## 6 Goose 詳細

### 6.1 goosecc コマンドライン引数

goosecc の書式を以下に示します。

```
goosecc [options] inputfile(s)...
```

*inputfile(s)...* は C 言語で記述されたプログラムのソースファイルです。 *options* には以下のものを指定できます。

#### コマンドラインオプション一覧

オプション	意味 ([ ] 内はデフォルト 値)
-goose-arch < arch >	ハードウェアアクセラレータの種別を指定します。[gdr]
-goose-backannotate	アセンブリコードの再生成を抑止します。 アセンブリコードをハンドチューニングした場合に使用します。
-o <outputfile >	生成する実行ファイルのファイル名を指定します。[a.out]
-I <headerpath>	ヘッダファイルの検索パスを指定します。
-verbose [ = level ]	饒舌モードを指定します。饒舌度を指定することも可能です。
-v [ level ]	値が大きいほど饒舌になります。最小値は 0、デフォルト 値は 2 です。
-help	ヘルプメッセージを出力します。
-h	
-Wcc "options"	C コンパイラへ渡すオプションを明示的に指定します。

なお、これら以外のオプションは、goosecc では解釈されずに C コンパイラへと渡されます。

### 6.2 goosecc の内部動作

goosecc は内部的にいくつかのコマンドを呼び出してコンパイルを行います。通常、ユーザがこれらの呼び出しを意識する必要はありませんので、本節では動作の概要のみを簡単に説明します。

goosecc は引数として渡されたファイルのうち、拡張子 .c を持つものについてのみ、ファイルの内部を走査します。内部に Goose 疑似命令が見つかったら、そのファイルを C 言語フロントエンド (goosec2q) へ渡します。goosec2q の出力は中間言語コンパイラ (LSUMP)、アセンブラ (vsm) を経由して C コンパイラ (gcc もしくは環境変数 CC で指定したコンパイラ) へと渡されます。

goosecc に渡されたファイル内に Goose 疑似命令が見つからない場合には、そのファイルはそのまま C コンパイラへ渡されます。 .c 以外の拡張子を持つファイルについては、疑似命令の有無や記述言語にかかわらず、無条件に C コンパイラへ渡されます。

C コンパイラは各プログラムから受け取った入力をコンパイルし、アクセラレータ制御用ライブラリ等とリンクして、実行ファイルを生成します。

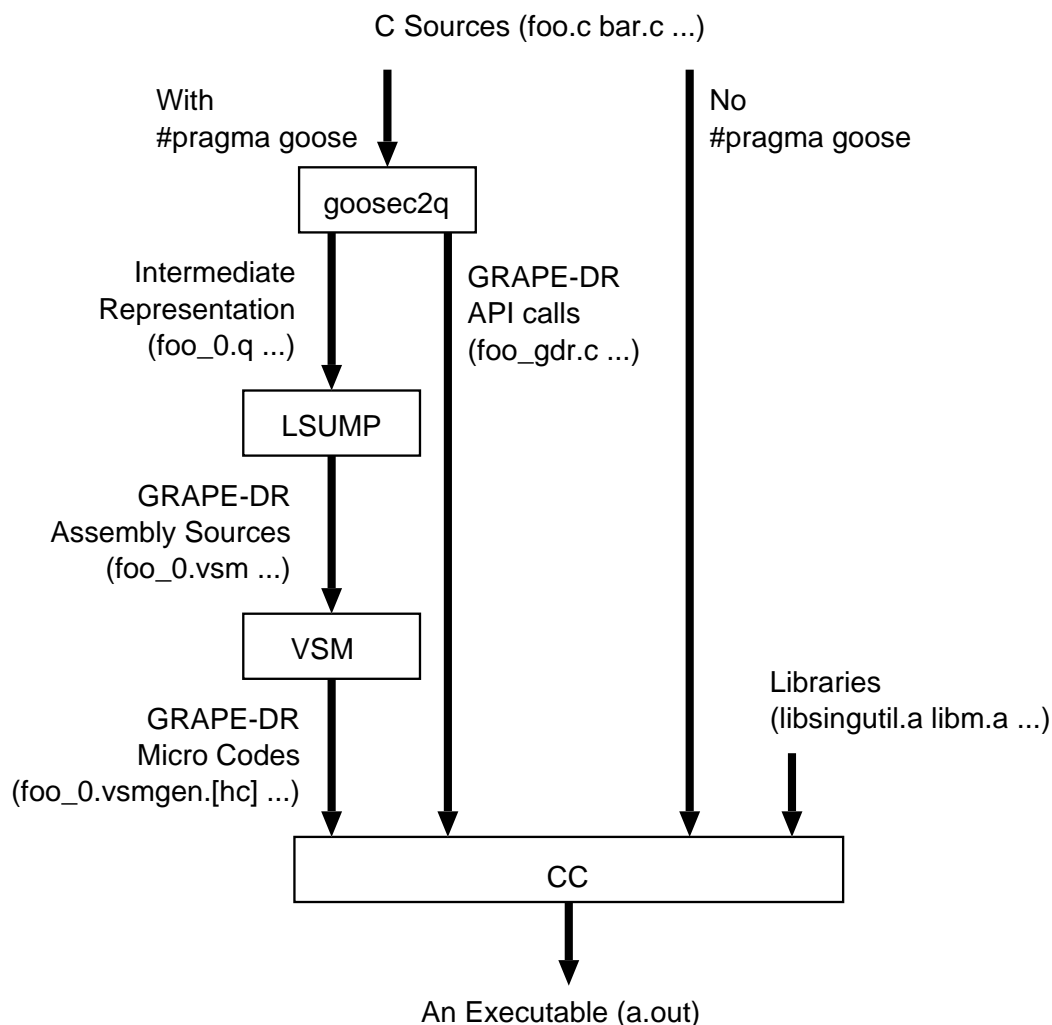


図 : goosecc によるコンパイルの流れ  
(ハードウェアアクセラレータとして GRAPE-DR を使用する場合)。

### 6.3 ハードウェアアクセラレータのアーキテクチャ

本節ではハードウェアアクセラレータのアーキテクチャを説明します。ただし説明内容は、Goose からアクセラレータを使用する際の効率を向上させるために有用と思われる、

基礎的な知識のみに限定します。詳細説明は各アクセラレータの関連文書を参照してください。

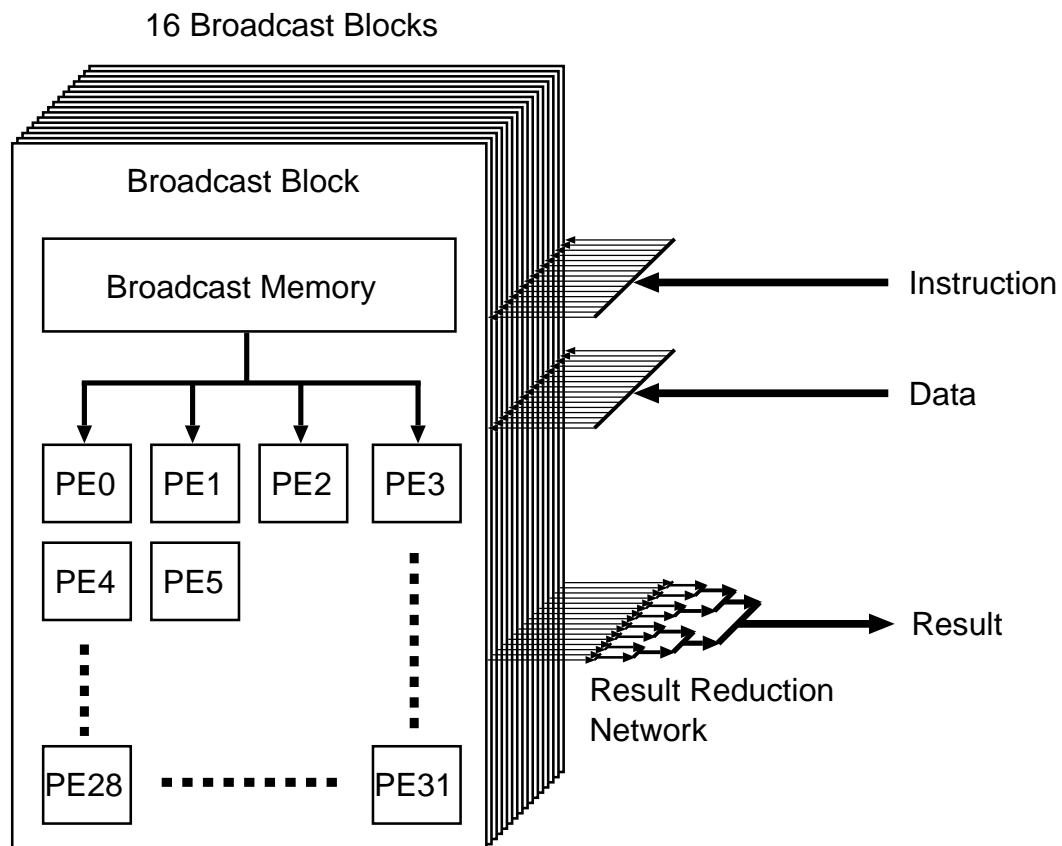
### 6.3.1 GRAPE-DR

GRAPE-DR は GRAPE-DR プロセッサチップ (東京大学、国立天文台開発のカスタム LSI) を搭載した、当社開発のハードウェアアクセラレータです。チップを 1 個搭載したモデル (model450 ほか) と、4 個搭載したモデル (model1800 ほか) があります。

GRAPE-DR チップには 512 個のベクトルプロセッサ (processor element、以降 PE と表記します) が集積されています。各 PE は乗算器と加算器、レジスタファイル、ローカルメモリを持ち、外部から送られてくる命令を SIMD 的に実行します。ベクトル演算のループ長は最大 4 です。

512 個の PE は 32 個ずつ、全 16 個の放送ブロックにまとめられています。各放送ブロックは放送メモリを持ち、同一ブロック内の 32 個の PE に対してデータを放送することが可能です。

16 個の放送ブロックの出力は、ツリー型の結果縮約ネットワークによって相互に接続されています。各放送ブロックからの 128 個 (PE 32 個 × ループ長 4) の計算結果出力は、加算 (あるいは論理演算など) によってトーナメント形式で縮約され、チップ外部へ出力されます。



図：GRAPE-DR チップの内部構成

2 重 for ループ (式 (2) 型) の動作例:

節 3.4 のチュートリアルで示した 2 重 for ループ (以下に再掲) を例に、GRAPE-DR 上での計算の実行手順を説明します。

```

#pragma goose parallel for icnt(i) jcmt(j) \
  ip(x[i][0..2]) jp(x[j][0..2],m[j]) result(a[i][0..2],pot[i])\
precision ("double")
  for(i=0;i<n;i++) {
    for(k=0;k<3;k++) a[i][k] = 0.0;
    pot[i] = 0.0;
    for(j=0;j<n;j++) {
      for(k=0;k<3;k++) dx[k] = x[j][k] - x[i][k];
      r2 = dx[0]*dx[0] + dx[1]*dx[1] + dx[2]*dx[2] + eps2;
      rinv = rsqrt(r2);
      mrinv = m[j]*rinv;
      mr3inv = mrinv*rinv*rinv;
      for(k=0;k<3;k++) a[i][k] += mr3inv * dx[k];
      pot[i] += -mrinv;
    }
  }
  get_cputime(&lt,&st);
  total_time += lt;
  for(i=0;i<n;i++) pot[i] += m[i]/sqrt(eps2);

```

- (手順 1) 入出力型として jp を持つ  $n$  個の変数  $x[j][0..2]$  および  $m[j]$  を、ホスト 計算機 から GRAPE-DR ボード 上の外部メモリに書き込みます。
- (手順 2) 入出力型として ip を持つ変数  $x[i][0..2]$  を、各 PE のローカルメモリに書き込み ます。書き込まれるのは全  $n$  個の  $x[i][0..2]$  のうち、128 個だけです。この 128 個を全 16 個の放送ブロックに放送します。各放送ブロック内では、この 128 個を PE0 ~ PE31 の 32 個の PE それぞれへ 4 個 (つまりベクトルプロセッサのループ 長) ずつ書き込みます。すべての放送ブロックに、同一の  $x[i][0..2]$  が書き込ま れることに注意してください。
- (手順 3) 計算に先立ち、外部メモリから 16 個の  $x[j][0..2]$  および  $m[j]$  を読み出し、各 放送ブロックの放送メモリに書き込んでおきます。各放送ブロックの放送メモリに は、互いに異なる  $x[j][0..2]$ ,  $m[j]$  が書き込まれます。
- (手順 4) 計算を開始すると、各放送ブロックの放送メモリから 32 個の PE へ  $x[j][0..2]$  と  $m[j]$  が放送されます。各 PE はこれらと自身の  $x[i][0..2]$  を用いて 2 重 for ルー プ最内部の計算を実行します。同一の放送ブロック内の PE は、共通の  $x[j][0..2]$ ,  $m[j]$  から互いに異なる  $x[i][0..2]$  に対して及ぼされる重力を計算することになりま す。これに対し、異なる放送ブロック内の 32 個の PE は、互いに異なる  $x[j][0..2]$ ,  $m[j]$  から共通の  $x[i][0..2]$  に対して及ぼされる重力を計算することになります。
- (手順 5) 16 個の放送ブロックからは、それぞれ 128 個の  $i$  に対する  $a[i][0..2]$  と  $pot[i]$  が、計算結果として出力されます。これら  $128 \times 16$  個の計算結果は、ツリー型の 結果縮約ネットワークを通して順次積算され、128 個に縮約されます。

(手順 6) すべての  $i$  に対する  $a[i][0..2]$  と  $pot[i]$  を求め終わるまで、(手順 2) から (手順 5) と同様の手順を、新たな 128 個の  $x[i][0..2]$  について順次繰り返します。

この手順で計算を行う場合、 $i$  に関する並列度が 128 以上、 $j$  に関する並列度が 16 以上あれば、チップ内のすべての PE を利用して計算を行えます。並列度がそれよりも低い場合や、128 や 16 で割り切れない端数部分の処理を行う場合には、一部のプロセッサ要素は計算に利用されず、遊んでしまうことになります。

1 重 for ループ (式 (1) 型) の動作例:

節 3.3 のチュートリアルで示した 1 重 for ループ (以下に再掲) を例に、GRAPE-DR 上での計算の実行手順を説明します。

```
#pragma goose parallel for icnt(i) result(sum) ip(i) shared_ro(dx)
for(i=0;i<n;i++) {
    x = (i+0.5)*dx;
    sum += integrand(x)*dx;
}
```

(手順 1) 入出力型として  $ip$  を持つ変数  $i$  と、型  $shared\_ro$  を持つ変数  $dx$  を、各 PE のローカルメモリに書き込みます。書き込まれるのは全  $n$  個の  $i$  と  $dx$  のうち、128 個だけです。この 128 個を全 16 個の放送ブロックに放送します。各放送ブロック内では、この 128 個を PE0 ~ PE31 の 32 個の PE それぞれへ 4 個 (つまりベクトルプロセッサのループ長) ずつ書き込みます。すべての放送ブロックに、同一の  $i$  と  $dx$  が書き込まれることに注意してください。

(手順 2) 計算を開始すると、PE は  $i$  と  $dx$  を用い for ループ内部の計算を実行します。全 16 個の放送ブロックが同じ計算を実行することになります。

(手順 3) 16 個の放送ブロックからは、それぞれ 128 個の  $i$  に対する計算結果が出力されます。これら  $128 \times 16$  個の計算結果は、ツリー型の結果縮約ネットワークを通して順次積算され、128 個に縮約されます。ホスト 計算機上でこの結果を 16 で除し、最終的な計算結果を得ます。

(手順 4) すべての  $i$  に対する計算結果を求め終わるまで、(手順 1) から (手順 3) と同様の手順を、新たな 128 個の  $i$  について順次繰り返します。

この手順には 2 つの改善の余地があります。

- 全 16 個の放送ブロックが同じ計算を実行しています。本来であれば各放送ブロックには互いに異なる  $i$  を書き込み、 $128 \times 16$  個の  $i$  に対する計算を並列に実行することが可能ですが、実際にはそうになっていません。



- `shared_ro` 型変数は `i` に依存しないため、本来であれば計算開始時に一度だけ、ホスト 計算機からアクセラレータへ送信すれば充分のはずです。しかし実際には `for` ループ内で `ip` 型変数と同様に繰り返し送信を行っています。この無駄な通信を省けば、通信時間を短縮できます。

これらの問題は近日中に修正の予定です。ただし、修正を行った場合であっても、式 (1) 型の計算には通信量に関する本質的な問題は残ります (節 3.1)。

## 7 利用許諾

Goose ソフトウェアパッケージ (以降「本ソフトウェア」と呼びます) の利用を、本ソフトウェアの所有者に対してのみ許可します。本ソフトウェアの再配布を禁じます。本ソフトウェアの改変物についても、本ソフトウェアの利用許諾に従うものとします。

本ソフトウェアの著作権は株式会社 K&F Computing Research に帰属します。本ソフトウェアが依存する外部プログラムやライブラリについては、それぞれに個別の利用許諾と著作権が定められています。

## 8 謝辞

Goose ソフトウェアパッケージの作成にあたり、以下の方々に協力を頂きました。ここに感謝の意を表します:

中里直人氏 (会津大学) には LSUMP を Goose の中間言語コンパイラとして使用することを快諾頂きました。また当社からの数多くの仕様拡張や仕様変更の要求に対応して頂きました。牧野淳一郎氏 (国立天文台) には GRAPE-DR 用アセンブラ VSM の利用を快諾頂きました。勝山哲志氏には Goose のロゴを作成して頂きました。C 言語フロントエンド `goosec2q` の開発には、青木峰郎氏開発のパーサジェネレータ `racc` を使用しました (<http://i.loveruby.net/ja/projects/racc/>)。

## 9 Goose ソフトウェアパッケージ更新履歴

version	date	description	author(s)
1.0.1 $\alpha$	28-Sep-2009	文書の英語版を追加	AK
1.0.0 $\alpha$	17-Sep-2009	初版作成。	A. Kawai, T. Fukushige

お問い合わせおよびバグレポートは下記まで:

株式会社 K&F Computing Research ([support@kfcr.jp](mailto:support@kfcr.jp))