Goose ソフトウェアパッケージ
ユーザガイド

for goose version 1.3.3

最終更新：2010 年 3 月 17 日

K&F Computing Research Co.

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

目 次

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。現時点では、ソースコード記述言語として C 言語を、ハードウェアアクセラレータとして当社 GRAPE-DR と AMD、NVIDIA、両社の GPU をサポートしています。今後 Fortran、OpenCL フレームワーク、Intel SSE テクノロジ、当社 GRAPE-7 へ順次対応の予定です。

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 動作環境

本パッケージは 64 ビット Linux OS (x86_64) 上で動作します。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/`
- ATI Stream SDK (version 1.3 以降を推奨)
  `http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx`
- CUDA 開発環境 (version 2.1 以降を推奨)
  `http://www.nvidia.com/object/cuda_home_new_jp.html`
- GRAPE Software Package (version 1.1.0 以降を推奨)
  `http://www.kfcr.jp/goose.html`
- 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`

対象とするアクセラレータによっては、上記ソフトウェアのうち一部は不要です。例えばアクセラレータとして NVIDIA 社の GPU のみを使用する場合には、ATI Stream SDK や GRAPE Software Package は不要です。アクセラレータと動作に必要なソフトウェアの対応関係を下表に示します。

	アクセラレータ		
	GRAPE-DR	AMD	NVIDIA
ruby	○	○	○
gcc	○	○	○
ATI Stream SDK		○	
CUDA			○
GRAPE Software Package	○		
LSUMP	○	○	
VSM	○		

2.2 パッケージの展開

ソフトウェアパッケージ goosepkg$M.m.r$.tar.gz を展開してください ($M.m.r$ 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:

00readme-j	パッケージ概要。
00readme	パッケージ概要英語版。
doc/	ユーザガイド、その他の文書。
scripts/	パッケージ管理ユーティリティ。
bin/goosecc	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。
goosec2ir	C 言語を Goose 内部表現へ変換するツール。goosecc が内部で使用します。
include/	ヘッダファイル。Goose が内部で使用します。
lib/	ライブラリ。Goose が内部で使用します。
singutil/	GRAPE-DR 制御用ライブラリ。Goose が内部で使用します。
gcalutil/	AMD 社製 GPU 制御用ライブラリへの API ラッパー。Goose が内部で使用します。
sample/	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。
init/	sample/ 内のプログラムが使用するデータファイル。
misc/	環境変数設定や Makefile のサンプル、init/ 内のデータを生成するプログラム等。

2.3 環境変数の設定

以下の環境変数を設定してください。対象とするアクセラレータによっては、一部設定不要な変数があります。例えばアクセラレータとして AMD 社の GPU のみを使用する場

合には、VSMPATH、GRAPEPKGPATH、CUDAPATH、NVCC 等の設定は不要です。

LSUMPPATH : LSUMP のインストールされているパス。

VSMPATH : VSM のインストールされているパス。

GRAPEPKGPATH : GRAPE Software Package のインストールされているパス。

CUDAPATH : CUDA のインストールされているパス。
デフォルト値は /usr/local/cuda

NVCC : CUDA コンパイラ nvcc のパス。
デフォルト値は /usr/local/cuda/bin/nvcc

ATICALPATH : ATI CAL のインストールされているパス。
デフォルト値は /usr/local/atical

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
kawai@localhost>export CUDAPATH=/usr/local/cuda
kawai@localhost>export NVCC=/usr/local/cuda/bin/nvcc
kawai@localhost>export ATICALPATH=/usr/local/atical
```

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
kawai@localhost>setenv CUDAPATH /usr/local/cuda
kawai@localhost>setenv NVCC /usr/local/cuda/bin/nvcc
kawai@localhost>setenv ATICALPATH /usr/local/atical
```

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/genv.sh
$goosepkg/misc/rc/genv.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。

CC : 使用する C コンパイラを指定します。ただし現在のところ、
Goose の動作確認は gcc のみで行っています。

CFLAGS : C コンパイラへ渡す引数を指定します。

2.4 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

...

-----------------------------------------
Installation of singutil : success.
Installation of gcalutil : success.
-----------------------------------------
done
```

以上でインストールは完了です。なお singutil, gcalutil のコンパイルに失敗する場合がありますが、これらはそれぞれ GRAPE-DR、AMD 社製 GPU にのみ必要なユティリティです。当該アクセラレータを使用しない場合には、インストールに失敗しても問題ありません。

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 を実行します。

```
kawai@localhost>pwd
/home/kawai/src/goosepkg/sample/s9
kawai@localhost>make
cc -O3 sticky9.c -o sticky9_host -lm -fopenmp
../../bin/goosecc -O3 -v2 --goose-arch gdr -o sticky9_gdr sticky9.c -lm
Info    : $LSUMPPATH                                : /home/kawai/src/lsump
Info    : $VSMPATH                                  : /home/kawai/vsm
Info    : $GRAPEPKGPATH                             : /home/kawai/grapepkg
Info    : Architecture                              : gdr
...

=========================================================
 An executable file sticky9_amd generated successfully.
=========================================================
kawai@localhost>ls
goosetmp      inputpara9   Makefile        sticky9.c  sticky9_amd
sticky9_host  sticky9_gdr  sticky9_nvidia
```

コンパイルが正常に完了すると、最大 4 つの実行ファイル s9_host, s9_gdr, s9_amd, s9_nvidia が生成されます。これらは順に、ホスト計算機だけで計算を行う実行ファイル、GRAPE-DR を使用して計算を行う実行ファイル、AMD 社製の GPU を使用して計算を行う実行ファイル、NVIDIA 社製の GPU を使用して計算を行う実行ファイルです。特定ハードウェア向けの環境がインストールされていない場合には、対応する実行ファイルは生成されません。例えば CUDA がインストールされていない環境では、s9_nvidia は生成されません。

NVIDIA 社製 GPU に固有の注意 : サンプルコードはすべて倍精度で演算を行うように記述されていますが、NVIDIA 社製 GPU のうち旧世代の製品には、倍精度演算を扱えないものがあります。これらの機種を使用する場合にはサンプルコード中の Goose for 疑似命令のオプション引数 precision("double") を precision("single") へ変更し、演算を単精度で行うよう指定してください (節 4.2 参照)。またサンプルコードの makefile ($goosepkg/misc/sample.mk) では、nvcc に対して GPU の世代を指示するコンパイルオプション -arch=sm_13 が指定されています。このオプションを削除してください。

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

$$\vec{r_i} = f(\vec{x_i}) \tag{1}$$

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、

$$\vec{r_i} = \sum_j f(\vec{x_i}, \vec{y_j}) \tag{2}$$

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 の総数が充分に大きければ「(a) 並列度が高」い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充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 6.4.1 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)。

$$\vec{f_i} = \sum_{j \neq i} \frac{G m_i m_j (\vec{x_j} - \vec{x_i})}{|\vec{x_j} - \vec{x_i}|^3} \tag{3}$$

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 π)。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 loopcounter(i) result(sum)
  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、loopcounter(i) などが与えられています:

- 引数 loopcounter(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。なおループカウンタ変数はデフォルト値 i を持ちます。ループカウンタとして i を使用している場合には、loopcounter 宣言を省略できます。
- 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。

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いましょう。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。

```
kawai@localhost>../../bin/goosecc midpoint.c -o midpoint --goose-arch amd
==========================
 Processing 'midpoint.c'.
==========================
/home/kawai/src/goosepkg1.3.3/bin/goosec2ir  -i midpoint.c \
-o ./goosetmp -a amd -p f0 -v2
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
...

============================================================
 An executable file midpoint_amd generated successfully.
============================================================
```

ここで `--goose-arch` はアクセラレータの種別を指定する引数です。GRAPE-DR, AMD 社製 GPU, NVIDIA 社製 GPU はそれぞれ `gdr`, `amd`, `nvidia` で指定します。`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。

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が得る加速度 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 ϵ を導入します。すると計算すべき式は

$$\vec{a_i} = \sum_{j \neq i} \frac{m_j(\vec{x_j} - \vec{x_i})}{(|\vec{x_j} - \vec{x_i}|^2 + \epsilon^2)^{3/2}} \tag{4}$$

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 precision ("double") loopcounter(i, 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 説明したものも含みます)。

- 引数 `loopcounter(i, j)` は、変数 `i` と `j` を 2 重 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。変数宣言の順序に意味はありません。goosecc は Goose for 疑似命令の直後に記述された for ループを外側のループとして識別します。この for ループのカウンタが loopcounter 宣言に含まれていない場合には、コンパイルをエラー終了します。

外側の for ループが見つかると、goosecc はそのループカウンタを loopcounter 宣言のリストから取り除き、残った変数を内側の for ループ用カウンタとみなします。次に外側ループの内部を精査し、内側用カウンタを持った for ループを探します。当該ループをひとつ見つけると、それを内側のループとして識別します。見つからない場合や、複数見つかった場合には、コンパイルをエラー終了します。

なお外側、内側のループカウンタ変数は、それぞれデフォルト値 i と j を持ちます。ループカウンタとしてこれらの変数を使用している場合には、loopcounter 宣言を省略できます。

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

- 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。precision 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、組み込み関数が使用されます (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。

- 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、コンパイル時にエラーとなります (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)	
gravity/	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)。
gravity_cutoff/	P^3M 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。
hermite/	重力相互作用と、その時間微分を計算します。
gravperf/	重力相互作用計算の性能評価を行います。重力を受ける粒子、 及ぼす粒子の粒子数を独立に指定して計算を行えます。
pairwise/	2 粒子間に働く重力相互作用計算の精度評価を行います。
tree/	Barnes-Hut ツリー法を用いて重力相互作用を計算します。
s9/	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)。
s8/	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)。
vdw/	ファンデルワールス力を計算します (Lennard-Jones ポテンシャル)。
sph/	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)	
singleloop/	1 重 for ループの単純なテスト計算 (整数値の積算) をおこないます。
midpoint/	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)。
mesh1d/	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 を実行すると、各種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 (節 2.4 参照)。

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 `precision` オプション引数で指定した型) 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。

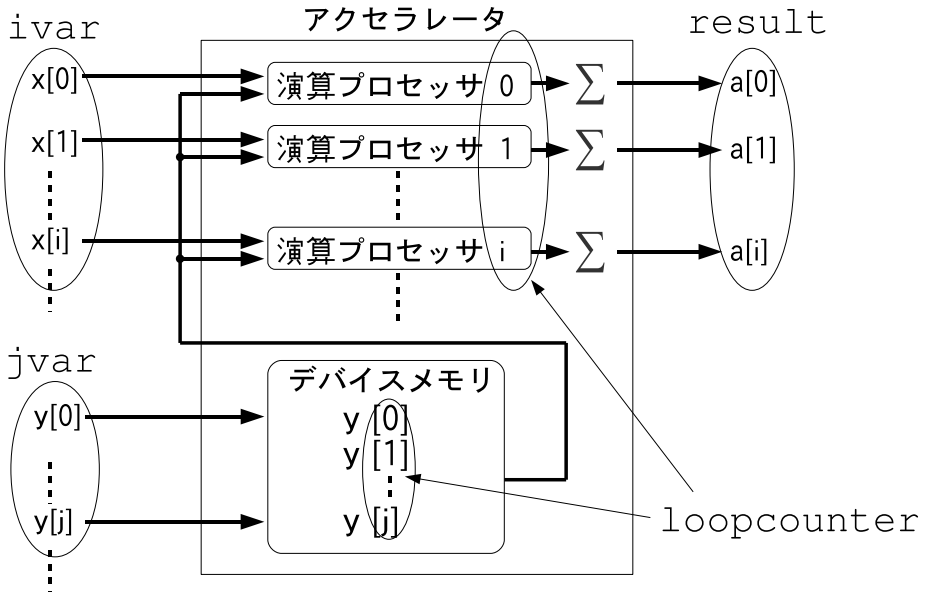


図 : 変数の入出力型の例。図中には入力用の型 ivar, jvar, 出力用の型 result が示されている。これらの他に、必要に応じて cvar 型, private 型が定義される。

入出力型一覧

型名	用法
ivar (別名 ip)	計算の入力パラメタのうち、外側の for ループのカウンタ値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。
jvar (別名 jp)	計算の入力パラメタのうち、内側の for ループのカウンタ値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。
cvar (別名 shared_ro)	計算の入力パラメタのうち、for ループのカウンタ値 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。

入出力型の自動推論

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、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。

- i でインデクスされた変数は ivar 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 jvar 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、cvar 型と認識されます。

ほとんどの場合、入出力型をユーザが明示する必要はありません。ただし、i でインデクスされていない変数を計算結果として使用する場合には、これを result 型として明示する必要があります (明示しない場合には自動推論によって private 型と判定されてしまいます)。節 3.3 チュートリアル 1 の変数 sum は、このような場合の一例です。入出力型の指定方法については節 4.1.2 を参照してください。

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
`loopcounter(`*var0*`,[`*var1*`])`	for 文のループカウンタとして変数 *var0* と *var1* を用いるよう指定します。指定がない場合には変数 `i`, `j` を使用します。
`result(`*var, ...*`)`	変数 *var* の入出力型を `result` 型に指定します。
`ivar(`*var, ...*`)` 別名 `ip(`*var, ...*`)`	変数 *var* の入出力型を `ivar` 型に指定します。
`jvar(`*var, ...*`)` 別名 `jp(`*var, ...*`)`	変数 *var* の入出力型を `jvar` 型に指定します。
`cvar(`*var, ...*`)` 別名 `shared_ro(`*var, ...*`)`	変数 *var* の入出力型を `cvar` 型に指定します。
`private(`*var, ...*`)`	変数 *var* の入出力型を `private` 型に指定します。
`precision("`*prec*`")`	変数の数値フォーマットとして、"*prec*" を指定します。指定できる値は "`double`" (64-bit 浮動小数点) と "`double-single`" (加減算 64-bit 浮動小数点、乗算 32-bit 浮動小数点 [1] [2])、"`single`" (32-bit 浮動小数点) です。"`quadruple`" (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 "`double`" を使用します。なお "`double`" 型はすべてのアクセラレータで使用できますが、それ以外の型については現段階では未対応の機種があります。対応状況については下表及び `goosecc` 実行時のメッセージ出力を参照してください。
`asmfile("`*filename*`")`	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 (*e.g.,* `foo_0.vsm`) 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。

	アクセラレータ		
precision	GRAPE-DR	AMD	NVIDIA
single	-	○	○
double-single	○	△	○
double	○	○	○
quadruple	△	△	-

○:対応済　△:対応予定

(続く)

オプション引数一覧 (続き):

書式	機能
njp_write(*var*)	指定すると、jvar 型データのうち、インデクス j の値が *var* 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 jvar 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 (`$goosepkg/sample/s8/` 参照)。指定がない場合にはすべての jvar 型データを送信します。
nip_pack(*uint*)	ハードウェアアクセラレータのプロセッサ要素 1 個が処理する演算量を指定します。指定のない場合には、ひとつのプロセッサ要素が *nvec* 個の i に対する計算を実行しますが (ここで *nvec* はベクトルプロセッサのループ長)、指定すると、$uint \times nvec$ 個の 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 : `result(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;
     ivar(x[0]);                  // 配列型変数の要素
     ivar(x[i]);                  // 配列型変数のインデクスに変数を使用
     ivar(x[index[i]]);           // 配列のインデクスに他の配列の要素を使
用
     jvar(*(x+j));                // ポインタのデリファレンス
     jvar(a_struct.a_member);     // 構造体メンバの参照
     jvar(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 : `result(a[i][0..2])`

  という記述は

  `result(a[i][0], a[i][1], a[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のアドレス空間は独立しているため、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;
     ivar(&x);                // NG
     ivar(x);                 // OK
     ivar(y);                 // NG
     ivar(y[0]);              // OK
     ivar(y[i]);              // OK
     ivar(*(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。
- NVIDIA 社製 GPU 向けには、この他に標準数学ライブラリのサポートするすべての関数が組み込み関数として用意されています ($e.g.$ `sqrt(x)`, `cos(x)`, `sin(x)` 等)。

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。それ以外の記述を見つけると、goosecc はコンパイルをエラー終了します。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 $var \mathrel{+}= inc\_val$ 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とトラブルシューティング

goosecc は C 言語の言語仕様で定義されている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。コンパイル不可能な記述を見つけると、警告あるいはエラーメッセージを出力し、記述の内容によってはコンパイル自体を中止します。本節ではコンパイル不可能な記述の典型的な例とその理由、対処法を説明します。

5.6.1 未定義の関数

Goose func 疑似命令の付加されていない関数が Goose for 疑似命令を付加した for 文の内部から使用されていた場合、goosecc はコンパイルをエラー終了します。

例 :

```
#pragma goose parallel for
for (i = 0; i < ni; i++) {
  for (j = 0; j < nj; j++) {
    printf("just for debugging purpose.\n");  // NG
  }
}
```

5.6.2 ivar 型変数、jvar 型変数、cvar 型変数への代入

ivar 型変数、jvar 型変数、cvar 型変数は読み出し専用ですので、左辺値にはなり得ません。goosecc はこのような変数を見つけるとコンパイルをエラー終了します。

例 :

```
#pragma goose parallel for ivar(x[i]) jvar(x[j]) cvar(eps)
for (i = 0; i < ni; i++) {
  for (j = 0; j < nj; j++) {
    x[j] = some_value;         // NG
    x[i] = other_value;        // NG
    eps  = yet_another_value; // NG
  }
}
```

5.6.3 result 型変数の参照

result 型変数は書き込み専用ですので、右辺値にはなり得ません。goosecc はこのような変数を見つけるとコンパイルをエラー終了します。ただし例外的に、積算演算子 (+= および −=) だけは使用できます。

例 :

```
#pragma goose parallel for result(r[i])
for (i = 0; i < ni; i++) {
  for (j = 0; j < nj; j++) {
    r[i] = some_value * (r[i] + other_value);   // NG
    r[i] = r[i] + some_value;                   // NG
    r[i] += some_value;                         // OK
    r[i] -= some_value;                         // OK
  }
}
```

5.6.4 2 重ループ中の result 型変数への単純代入

2 重ループ中の result 型変数に対しては単純な代入演算子 (=) は使用できません。積算演算子 (+= および −=) だけを使用できます。goosecc はこのような変数を見つけるとコンパイルをエラー終了します。このような記述は通常は冗長であり、2 重ループの外側にくくり出すことができるはずです。

例 :

```
#pragma goose parallel for result(r[i])
for (i = 0; i < ni; i++) {
  for (j = 0; j < nj; j++) {
    r[i] = some_value;       // NG
  }
  r[i] = some_value;         // OK
}
```

1 重ループ中の result 型変数に対してはこの制限はありません。

例 :

```
#pragma goose parallel for result(r[i])
for (i = 0; i < ni; i++) {
  r[i] = some_value;         // OK
}
```

5.6.5 result 型変数の初期値設定

入出力型 result を持つ変数の初期値は Goose for 疑似命令の直前で自動的に 0.0 に初期化されます。事前に他の値を設定しても、その値は上書きされます。これはホスト計算機からアクセラレータへの不必要な転送 (初期値の送信) をおこなわないために意図的に設定された制約です。result 型変数に初期値を与える記述:

例 :

```
for (i = 0; i < ni; i++) {
  r[i] = init_value; // 0.0 で上書きされる。
}
#pragma goose parallel for
for (i = 0; i < ni; i++) {
  for (j = 0; j < nj; j++) {
    r[i] += some_value;
  }
}
```

は、以下の記述で代用できます。

```
#pragma goose parallel for
for (i = 0; i < ni; i++) {
  for (j = 0; j < nj; j++) {
    r[i] += some_value;
  }
}
for (i = 0; i < ni; i++) {
  r[i] += init_value;
}
```

5.6.6 for ループ外部での private 型変数の初期化

入出力型 private を持つ変数の初期値は for ループの外部では設定できません。ループ内部でのみ設定できます。ループ内部で初期値が設定されていない場合、不定値をとります。現段階では goosecc はこのような記述に対して警告メッセージを出力せずにコンパイルを続行しますので、特に注意が必要です。近日中のバージョンアップで goosecc はこのような記述を検出し、警告メッセージを出力するよう改良される予定です。

```
some_pvar = init_value;  // 無効。
#pragma goose parallel for
for (i = 0; i < ni; i++) {
  for (j = 0; j < nj; j++) {
    some_pvar += some_value;
    r[i] += some_pvar;  // NG (some_pvar が初期化されていない)
  }
}

some_pvar = init_value;  // 無効。
#pragma goose parallel for
for (i = 0; i < ni; i++) {
  for (j = 0; j < nj; j++) {
    if (cond) {
      some_pvar = some_value;
    }
    r[i] += some_pvar; // NG (cond が偽のとき some_pvar が不定)
  }
}
```

5.6.7 2 つのループカウンタに依存する変数

外側と内側、両方の for ループのカウンタに依存する変数は、現段階では扱うことが出来ません。goosecc はこのような変数を見つけるとコンパイルをエラー終了します。

例 :
```
#pragma goose parallel for
for (i = 0; i < ni; i++) {
  for (j = 0; j < nj; j++) {
    r[i] += x[i][j]; // NG
  }
}
```

近日中のバージョンアップで goosecc はこのような変数を扱えるよう拡張される予定です。ただし拡張後も、このような変数の使用は最小限にとどめることが、実効性能の観点からは望まれます。

5.6.8 GRAPE-DR に固有の注意

GRAPE-DR を用いる場合には、Goose for 疑似命令付加した 2 重ループのうち、内側のループの範囲に注意が必要です。

内側のループのカウンタを j とするとき、j の変化する範囲は、16 の倍数に切り上げられます。これは jvar 型のデータが GRAPE-DR チップ内部の全 16 個の放送ブロックへ分配され、全放送ブロックが常に同じ個数の jvar 型データを処理しようとするために生じる制約です (節 6.4.1 参照)。

例えば以下の記述:

```
#pragma goose parallel for precision("double")
  for (i = 0; i < ni; i++) {
    for (j = 0; j < nj; j++) {
      sum[i] += x[j];
    }
  }
```

は nj が 16 の倍数の場合には正しく動作しますが、そうでない場合には sum[i] に x[nj], x[nj+1], ... , x[nj1-1] の値が余分に積算されてしまいます (ここで nj1 は 16 の倍数に切り上げられた nj の値を表します)。この問題を回避するため、Goose は jvar 型変数配列の末尾の端数分 x[nj], x[nj+1], ... , x[nj1-1] に、自動的に 0.0 を代入します。この機構によって、上記の記述は nj が 16 の倍数で無い場合でも正しい計算結果を返します。

しかしながら、上述の機構では問題を回避できない場合があります。例えば以下の記述は、nj が 16 の倍数で無い場合には、正しい計算結果を得られません。

```
#pragma goose parallel for precision("double")
  for (i = 0; i < ni; i++) {
    for (j = 0; j < nj; j++) {
      sum[i] += x[j] + y[i];
    }
  }
```

この場合には、x[j] の端数分は 0.0 に初期化されているため計算結果に影響を与えませんが、y[i] が nj1 - nj + 1 回余分に積算されてしまうことになります。このような記述でも正しい演算結果を得るには、以下のように新たに varj 型変数 filter を導入するなどの配慮が必要です。

```
  for (j = 0; j < nj; j++) {
    filter[j] = 1.0;
  }
  for (j = nj; j < nj1; j++) {
    filter[j] = 0.0;
  }
  x[nj] = 0.0;
#pragma goose parallel for precision("double")
  for (i = 0; i < ni; i++) {
    for (j = 0; j < nj; j++) {
      sum[i] += (x[j] + y[i]) * filter[j];
    }
  }
```

しかしながら元のコードを良く見直してみると、記述が冗長であり、そもそも 2 重ループは不要であることが分かります。以下のように書き直せば、filter 変数を導入しなく

ても正しい計算結果を得られるうえに、計算時間も短縮できます。

```
  for (j = 0; j < nj; j++) {
    tmp += x[j];
  }
#pragma goose parallel for precision("double")
  for (i = 0; i < ni; i++) {
    sum[i] = y[i] * nj + tmp;
  }
```

なお 2 重ループのうち外側のループの範囲については、このような制約はありません。

5.6.9 AMD 社製 GPU に固有の注意

AMD 社製 GPU を用いる場合には、Goose for 疑似命令付加した 2 重ループのうち、内側のループの範囲に注意が必要です。

内側のループのカウンタを j とするとき、j の変化する範囲は、倍精度演算時には 2 の、単精度演算時には 4 の倍数に切り上げられます。これは GPU 内の演算をベクトルレジスタのループ長 (倍精度 2, 単精度 4) へ切り上げることによって実効性能を向上させるために、意図的に設定された制約です。

ループの範囲が切り上げられた場合にも正しい演算結果を得られるよう、記述に配慮が必要です。GRAPE-DR にも同様の制約があり (ただし GRAPE-DR の場合には 2 や 4 ではなく、16 の倍数に切り上げられます)、その対処法に関する説明が前節「GRAPE-DR に固有の注意」にありますので参照してください。

なお外側のループの範囲については、このような制約はありません。

5.6.10 NVIDIA 社製 GPU に固有の注意

NVIDIA 社製 GPU のうち旧世代の製品には、倍精度演算を扱えないものがあります。これらの製品上では、倍精度以上の精度で演算を行うコードは正しく動作しません。Goose for 疑似命令へのオプション引数 `precision("single")` によって、演算を単精度で行うよう指定してください。

また倍精度演算を扱える製品であっても、倍精度以上の演算を行うためには、コンパイルオプション `-arch=` によって GPU の世代を明示する必要があります。

例 : `goosecc --goose-arch nvidia test.c -arch=sm_13`

このオプションを `goosecc` に与えると、`goosecc` は内部的にこれを `nvcc` へ渡します。`-arch=` で指定可能な世代名については NVIDIA 社の提供するドキュメントを参照してください。例えば CUDA version 2.3 の場合には「The CUDA Compiler Driver NVCC」(`nvcc_2.3.pdf`) に`-arch=` に関する記載があります。

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。
-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。
-Wld "*options*"	リンカへ渡すオプションを明示的に指定します。
–verbose [$= level$] -v [*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。 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。
–version -version	goosecc のバージョン情報を出力します。
–help -h	ヘルプメッセージを出力します。

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 の定義する定数

goosecc は以下の定数を定義します。定数はソースコード中で参照できます。

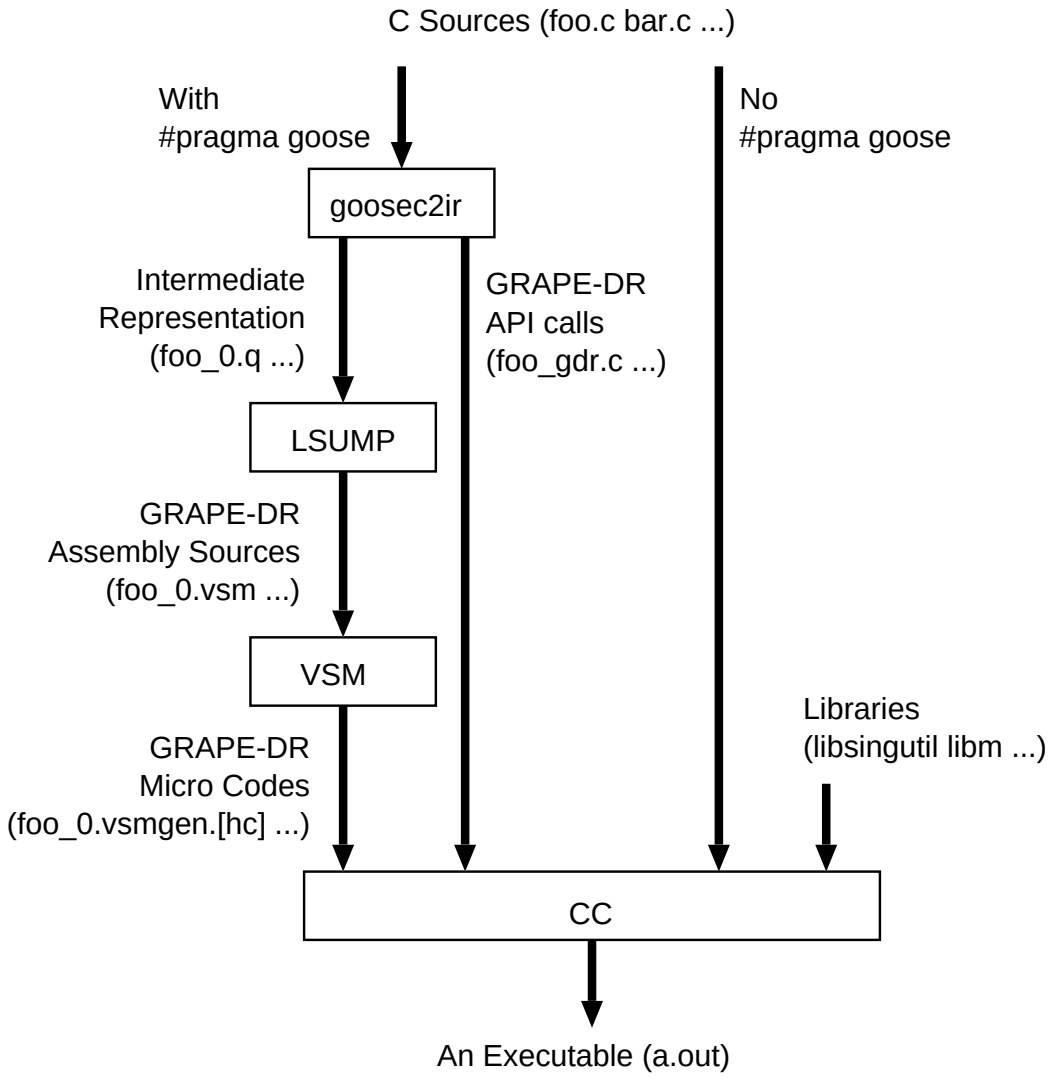
定数名	値
__GOOSECC__	1
__GOOSECC_VERSION__	バージョン番号 ($e.g.$ バージョン 1.2.3 の値は 0x010203)
__GOOSECC_ARCH_GDR__	1 (アーキテクチャ gdr を指定した場合のみ定義される)
__GOOSECC_ARCH_AMD__	1 (アーキテクチャ amd を指定した場合のみ定義される)
__GOOSECC_ARCH_NVIDIA__	1 (アーキテクチャ nvidia を指定した場合のみ定義される)

6.3 goosecc の内部動作

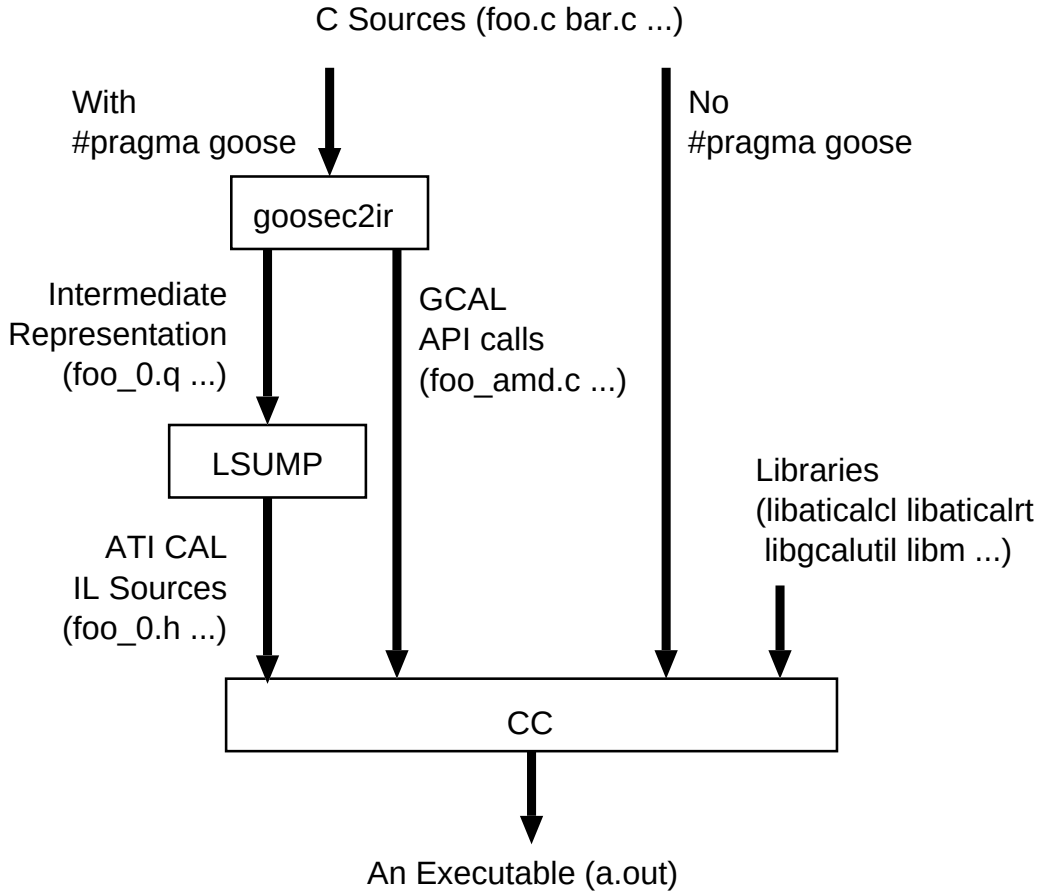
goosecc は内部的にいくつかのコマンドを呼び出してコンパイルを行います。通常、ユーザがこれらの呼び出しを意識する必要はありませんので、本節では動作の概要のみを簡単に説明します。

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 疑似命令が見つかると、そのファイルを C 言語フロントエンド (goosec2ir) へ渡します。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 (あるいは CUDA 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。

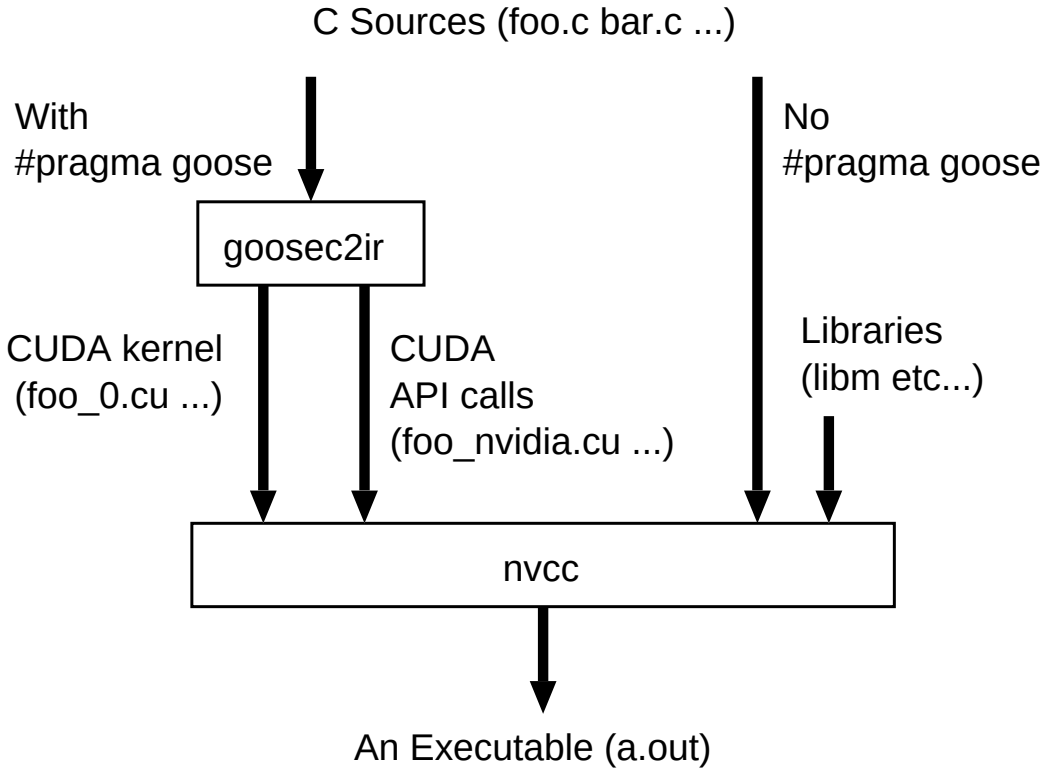
goosec2ir の出力は、対象とするアクセラレータ (--goose-arch で指定) に応じて各種の外部プログラムへ渡されます。外部プログラムの名称とその役割について、下図にまとめます:



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)。GRAPE-DR を対象とする場合には、`goosec2ir` は中間言語コンパイラ (`LSUMP`) 向けの記述を出力します。`LSUMP` はこれをアセンブリ (`vsm`) 記述へ変換します。アセンブリ記述はアセンブラ (`VSM`) によって処理され、C 言語でラップされたマイクロコード記述へと変換されます。この記述は 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) によって、Goose の生成するラッパーファイル (`foo_gdr.c`)、および GRAPE-DR 制御用ライブラリ `libsing.a` とともにコンパイル、リンクされ、実行ファイルが生成されます。



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 AMD 社製 GPU を使用する場合)。AMD 社製の GPU を対象とする場合には、`goosec2ir` は中間言語コンパイラ (`LSUMP`) 向けの記述を出力します。これを `LSUMP` が ATI IL 記述へと変換します。IL 記述は Goose の生成するラッパーファイル (`foo_amd.c`) にヘッダファイルとして挿入されます。C コンパイラはこのラッパーファイルを ATI CAL ライブラリの API ラッパー `libgcal`、および ATI CAL ライブラリ本体 `libaticalrt`, `libaticalcl` とともにコンパイル、リンクし、実行ファイルを生成します。IL 記述は実行ファイルの起動時に、CAL ランタイムライブラリによってアセンブリへと変換されます。



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 NVIDIA 社製 GPU を使用する場合)。NVIDIA 社製の GPU を対象とする場合には、goosec2ir は CUDA C コンパイラ nvcc 向けの記述を出力します。これを nvcc が Goose の生成するラッパーファイル (foo_nvidia.c) とともにコンパイル、リンクし、実行ファイルを生成します。

6.4 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と、そのアーキテクチャ向けに Goose がどのような最適化を行うかについて説明します。ただしアーキテクチャの説明は基礎的なものにとどめます。詳細に関しては各アクセラレータの関連文書を参照してください。

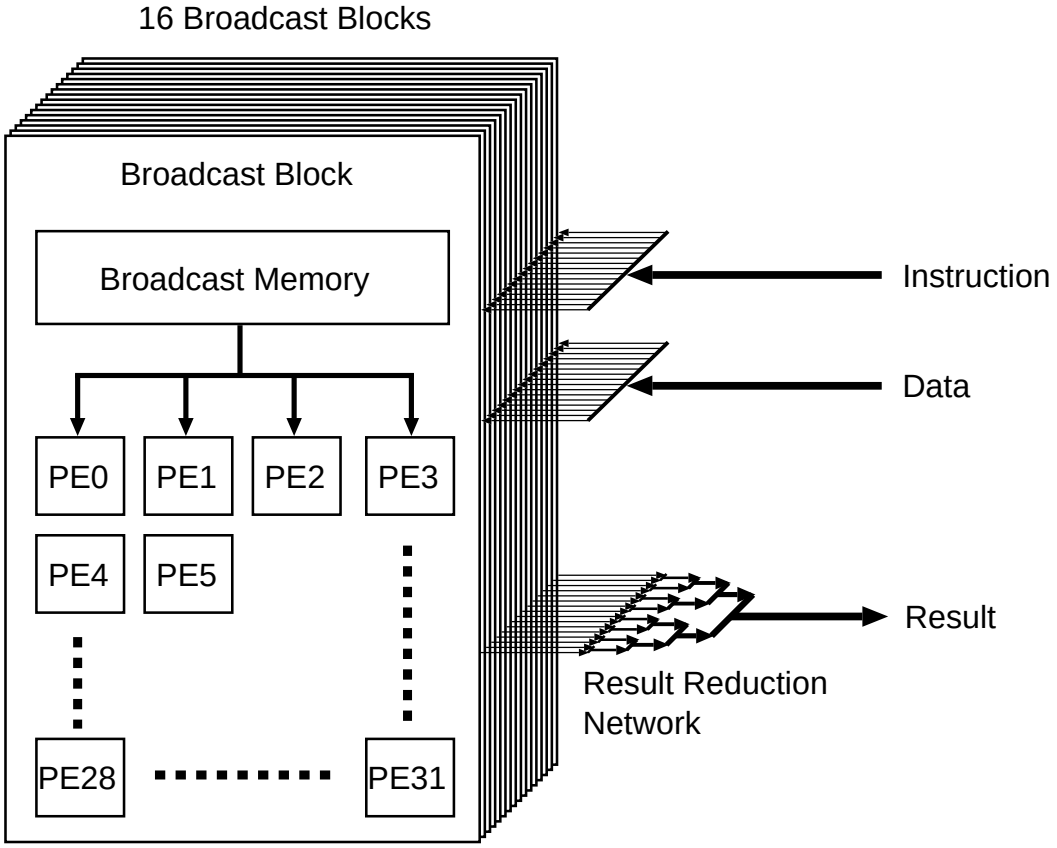
6.4.1 GRAPE-DR

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 (model460 ほか) と、4 個搭載したモデル (model2000 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 precision ("double") loopcounter(i, 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 $\sim$ 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] を求め終えるまで、(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 loopcounter(i) result(sum)
  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 $\sim$ 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 はを i と dx 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えるまで、(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)。

6.4.2 AMD 社製 GPU

Goose の行う AMD 社製 GPU 向けの最適化は、主として LSUMP によって行われています。LSUMP による最適化の手法については文献 [3] を参照してください。

6.4.3 NVIDIA 社製 GPU

Goose の生成する CUDA C コードは文献 [4] に述べられている手法をベースに、様々な手法を用いて最適化されています (共有メモリへの効率的なアクセス [5]、ループアンローリング、result 型変数のリダクション (グローバルメモリ経由、共有メモリ経由) 等)。

NVIDIA 社製の GPU を効率的に使用する方法については多くの書籍や研究報告があります。詳細についてはこれらを参照してください。

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 `goosec2ir` 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/`)。NVIDIA 社製 GPU 向けの最適化にあたっては、似鳥啓吾氏 (理化学研究所) に N 体計算プログラム Yebisu のソースコードを提供して頂きました。

9 参考文献

[1] K. Nitadori
"New approaches to high-performance N-body simulations — high-order integrator, new parallel algorithm and efficient use of SIMD hardware", doctoral thesis, 2008

[2] E. Gaburov, S. Harfst, S. P. Zwart
"SAPPORO: A way to turn your graphics cards into a GRAPE-6", New Astronomy Vol.14, Issue 7, p630, 2009

[3] K. Fujiwara, N. Nakasato
"Fast Simulations of Gravitational Many-body Problem on RV770 GPU", Extended undergraduate thesis in University of Aizu 2008, 2009,
`http://xxx.yukawa.kyoto-u.ac.jp/abs/0904.3659`

[4] 成見 哲、濱田 剛、小西 史一
"アクセラレータによる粒子法シミュレーションの加速", 情報処理 Vol. 50, No.2, p129, 2009

[5] T. Hamada, T. Iitaka
"The Chamomile Scheme: An Optimized Algorithm for N-body simulations on Programmable Graphics Processing Units",
http://arxiv.org/abs/astro-ph/0703100

10 Goose ソフトウェアパッケージ更新履歴

version	date	description	author(s)
1.3.3	17-Mar-2010	文書の英語版を更新。	AK
1.3.2	05-Mar-2010	[nvidia] ni の小さな領域での性能を強化。	AK
1.2.0	21-Jan-2010	[nvidia] 精度指定子 "double-single" をサポート。 [nvidia] 数学関数をサポート。 [gdr] ループカウンタに i,j 以外を使用できない バグを修正。	AK
1.1.0	21-Dec-2009	AMD, NVIDIA 両社の GPU に対応。	AK
1.0.1α	28-Sep-2009	文書の英語版を追加。	AK
1.0.0α	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)