Goose Software Package User's Guide

for gooSe version 1.0.1α

Last modified at : Sep. 28, 2009

K&F Computing Research Co.

K & F Computing Research Co.
E-mail: support@kfcr.jp

Contents

1 Abstract

This document describes usage of the Goose Software Package.

Goose is an environment for software development that integrates compilers and other utilities. It helps porting a program described with a programing language such as C, from PC to a hardware accelerator that works in SIMD fashion. It is designed to minimize the amount of modification to the original source code. The current version (version $1.0.1\alpha$) support C language and GRAPE-DR. In the near future, it is planned to support Fortran, GPU (both AMD and NVIDIA), Intel SSE technology, and GRAPE-7.

Goose is a kind of domain-specific developing environment, that is, it is not designed to support the whole grammar and specification of the programing language. It handles only descriptions which are suitable for the hardware accelerator. Other descriptions are passed on to a conventional compiler, such as `gcc`, to generate an executable for PC, which serves as the host computer of the accelerator. This approach would minimize the necessary amount of modification to the source code.

In section 2, installation procedure of the Goose Software Package (hereafter "this package") is described. In section 3, basic usage of Goose is described via tutorials. In section 4, all Goose directives are listed. In section 5, C-language grammar which can be handled by Goose are shown. Section 6 is devoted to detailed description of Goose compiler inside and architectures of the hardware accelerators.

In the rest of this document, the topmost directory of this package, *i.e.*, /*path_to_the_directory_at_which_you_unpacked_this_package*/`goosepkg`*the_version_number*/, is denoted as `$goosepkg`.

2 Installation

2.1 Required Environment

Goose runs on 64-bit Linux (x86_64). Goose internally uses the following softwares. They need to be installed beforehand.

- C compiler (gcc version 4.1.0 or higher recommended.)
  `http://gcc.gnu.org/`
- Ruby (version 1.8.5 or higher recommended.)
  `http://www.ruby-lang.org/`
- Perl (version 5.8.8 or higher recommended.)
  `http://www.perl.com/`
- LSUMP (A compiler for the Goose intermediate-representation. Developed by Naohito Nakasato, University of Aizu.)
  `http://www.kfcr.jp/goose.html`
- VSM (An assembler for GRAPE-DR. Developed by Junichiro Makino, National Astronomical Observatory of Japan.)
  `http://www.kfcr.jp/goose.html`
- GRAPE Software Package (version 1.1.0 or higher recommended.)
  `http://www.kfcr.jp/goose.html`

2.2 Unpacking and Setting up Environment Variables

Unpack the software package `goosepkg`nnn`.tar.gz`, where nnn is the version number. The package includes the following items:

00readme	Summary of the package.
00readme-j	00readme in Japanese.
doc/	User's guide and other documents.
scripts/	install and backup scripts.
bin/goosecc	A tool to build executables for hardware accelerators.
include/	Header files.
lib/	Libraries.
singutil/	GRAPE-DR control library.
sample/	Examples of application programs which can be compiled with the Goose C Compiler.
init/	Data files used by programs in sample/.
misc/	Templates for environment variable set up, a program to generate sample data, etc.

Set the environment variables listed below:

LSUMPPATH	: The path to the directory at which LSUMP is installed.
VSMPATH	: The path to the directory at which VSM is installed.
GRAPEPKGPATH	: The path to the directory at which GRAPE Software Package is installed.

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

Example for csh:
kawai@localhost>setenv LSUMPPATH /home/kawai/src/lsumppkg
kawai@localhost>setenv VSMPATH /home/kawai/src/vsm
kawai@localhost>setenv GRAPEPKGPATH /home/kawai/src/grapepkg
```

You can find template files for above mentioned setting at

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

You may copy one of them and modify it to suit your environment. The environment variables below are also available, but not mandatory.

CC	: Specify the C compiler to be used.
CFLAGS	: Specify arguments passed on to the C compiler.

2.3 Running the Install Script

Change directory to `$goosepkg` and run `$goosepkg/script/install`. It will generate libraries, utilities, and data used by sample programs.

```
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
```

This is all what you need for the installation. If the package is successfully installed, all sample programs in `$goosepkg/sample/` directory should be compiled. For example, you can change directory to `$goosepkg/sample/s9/` and run `make all` to compile a program `s9`, which

performs a gravitational many-body simulation.

```
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
```

On successful build, you will see two executables, namely, `s9` and `s9_host`. The former performs the simulation using a hardware accelerator. The latter performs the simulation purely on the host computer.

3 Basic Usage

3.1 Programs Handled by Goose

Although Goose handles a program written in C, it does not recognize all grammars defined by the language specification. It handles only descriptions suitable for SIMD-type accelerators. The rest of the program is passed on to a conventional C compiler, such as gcc, and compiled for run on the host computer. Here, a description "suitable for SIMD-type accelerators" is defined as the one which satisfy the followings:

(a) Can be highly parallelized, and,

(b) The required amount of data transfer among the accelerator and the external memory (or the host computer) is small, relatively to the amount of the calculation to be performed on the accelerator.

Goose handles calculations which can be expressed as:

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

where $\vec{x_i}$ are input parameters and $\vec{r_i}$ are calculation results. Results of calculations for different i must not depend each other, and they must be able to run in parallel. Among the above-mentioned calculations, Goose is especially optimized for the ones which can be expressed as:

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

A calculation described in the form of equation (1) satisfies the condition (a), if the different number of i is large enough. If it also can be described in the form of equation (2), it satisfies the condition (b), too. This is because the calculation for various different i-s can be performed using a shared set of $\vec{y_j}$. Therefore, the amount of data necessary to be supplied is smaller, compared to a case different $\vec{y_j}$-s are required for different i-s. Goose also can handle a calculation of the form of equation (1) but not of equation (2). In such a case, however, the potential performance of the accelerator may not fully be utilized.

The necessary number of i and j depends on the architecture of the accelerator. In order to perform a calculation using all processor elements of a GRAPE-DR, for example, the number of i and j should be larger or equal to 128 and 16, respectively (*cf.* section 6.3.1). Otherwise some processor elements are not utilized and remain idle during the calculation.

> Example : Calculation of gravitational interactions among point masses can be expressed by equation (2). The total force $\vec{f_i}$ on a point mass i from all other point masses is given by:
>
> $$\vec{f_i} = \sum_{j \neq i} \frac{G m_i m_j (\vec{x_j} - \vec{x_i})}{|\vec{x_j} - \vec{x_i}|^3} \tag{3}$$
>
> where G is the gravitational constant, $\vec{x_i}$ and m_i are position and mass of a particle i.

In C language, equation (1) can be expressed in various way. For simplicity, however, Goose recognize an expression by 'for' statement only. In the case of equation (2), it recognize an

expression by a nested 'for' statement only. Expressions other than 'for' statement, such as 'while' statement or 'goto' statement, are not recognized.

Example : Equation (3) can be described by a nested 'for' statement:

```
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 Design Flow of a Program

Basically, user write a program in C language and process it with a command `$goosepkg/bin/goosecc`. The command generates an executable for a hardware accelerator. The design flow of an application program is as follows:

(step 1) Write an application program in C language, and test it on the host computer (PC). A code fragment which is expected to run efficiently on the hardware accelerator (*cf.* section 3.1) should be described with 'for' statements. Do not use 'while' or 'do while' statements.

In the case of calculation which is symmetric for the inner/outer loops of the nested 'for' statement (*e.g.* particle interactions which satisfy the law of reciprocal action), the calculation is often described in the following manner, in order to save the amount of calculation. However, such a description should be avoided since Goose cannot handle

it.

Example : A description of equation (3) in C language. Optimized to save the amount of calculation using the symmetric nature of gravity. Such an optimization **should be avoided** since Goose cannot handle the inner 'for' statement with non-constant loop range.

```
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];
  }
}
```

(step 2) Attach an OpenMP directive (`#pragma omp parallel for`) to the 'for' statement mentioned in (step 1). Then test the program in a multi-threaded environment. This step may be skipped, although it is recommended as a method to check sanity of the 'for' statement when it is run in parallel. It is also useful to estimate the speed up factor. For the usage of OpenMP, refer to other documents such as `http://openmp.org/`.

(step 3) Attach a Goose directive (`#pragma goose parallel for`) to the 'for' statement mentioned in (step 1) and (step 2). Then compile the program with `goosecc` to obtain an executable. The executable generated performs the 'for' statement on the hardware accelerator. Other parts are performed on the host computer.

3.3 Tutorial 1 – A Numerical Integration

In this and next sections, basic usage of Goose is shown via two programming examples.

In this section, we describe Goose directives using a sample program `$goosepkg/sample/midpoint/midpoint.c`. This program numerically calculates $\int_0^1 4/(1+x^2)dx$, which should analytically be equal to π. The primary part of this program is a 'for' statement quoted below:

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

The domain $[0, 1]$ is divided into `n` segments. At each center of the segment, the code

multiplies the value of the integrand and dx, and sums it up to obtain sum. The integrand is defined as a C function, integrand().

You can compile the program for the host computer, using a conventional C compiler:

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

The generated executable, midpoint_host, outputs the following result:

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

Now we confirmed that the program works on the host computer as we expected. The next step is to put the above mentioned 'for' statement on a hardware accelerator. Note that the 'for' statement is preceded by a Goose directive, #pragma goose parallel for, hereafter we call this directive as a "Goose-for directive".

```
#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;
  }
```

A Goose-for directive directs the Goose C compiler to put the 'for' statement on a hardware accelerator. Other parts of the program is performed on the host computer.

If you look inside the Goose 'for' directive, you would notice that some arguments, such as icnt(i), are given for the directive:

- The argument icnt(i) denotes that a variable i is used as the loop counter of the 'for' statement.
- The argument result(sum) denotes that a variable sum need to be sent back from the accelerator to the host computer, as the calculation result.
- The argument ip(i) denotes that a variable i need to be sent from the host computer to the accelerator, as an input parameter of the calculation.
- Similar to ip(), the argument shared_ro(dx) also denotes that a variable dx need to be sent to the accelerator. The difference from the argument ip() is that dx, a variable specified by shared_ro(), does not change its value inside the loop, while i, a variable specified by ip(), does.

As we describe later in section 4.1.1, some of these arguments can be omitted. We recommend, however, that you explicitly specify the loop counter and the I/O type (*cf.* section 4.1.1) of all variables except private ones, at least until you get familiar with behavior of the Goose compiler.

Pay your attention to the definition of the integrand :

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

The function is preceded by a Goose directive, `#pragma goose func`, hereafter we call this directive as a "Goose-func directive". The directive should be attached to a function, if the function is used inside a 'for' loop, which a Goose-for directive is attached to. There are some restrictions for a function to be attached by a Goose-func directive: The function should have exactly one argument; It should explicitly return a value by a `return` statement; A Goose-func directive cannot be attached to a function of type `void`, or one without `return` statement.

Now we understood all the Goose directives in the program. Next we compile the program to build an executable for a hardware accelerator. For the compilation, we use a command `$goosepkg/bin/goosecc`. Change directory to `$goosepkg/sample/midpoint/`, and run `goosecc` with arguments as shown below. (see section 6.1 for a complete list of the command-line arguments):

```
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.
=========================================
```

On successful compilation, an executable `midpoint` will be generated. When you set up a hardware accelerator and run `midpoint`, the 'for' statement attached by a Goose-for directive runs on the accelerator. Other parts of the program are performed on the host computer. The outputs should be the same as that of `midpoint_host`.

3.4 Tutorial 2 – Gravitational Interactions among Point-Masses

In this section, we use a sample program `$goosepkg/sample/gravity/gravity.c`. This program calculates acceleration of particles a_i in a many-body system, in which point-mass particles are interacting via gravity. The gravitational force is originally described by equation (3). Here, we adjust the spatial and mass scaling so that the gravitational constant G equals unity, and introduce a "softening parameter", ϵ, in order to avoid numerical inaccuracy:

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

In the sample program, you can find a nested 'for' statement, that describes the equation above.

```
#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);
```

Equation (4) is in the form of equation (2). In such a case, the program should be described using a nested 'for' statement (*cf.* section 3.1). A Goose-for directive directs the Goose C compiler to put the nested 'for' statement on a hardware accelerator. Other parts of the program is performed on the host computer.

The followings are arguments to the Goose-for directive (some of them are the ones described in section 3.3):

- The argument `icnt(i)` denotes that a variable `i` is used as the loop counter of the outer 'for' statement, while `jcnt(j)` denotes `j` is the counter of the inner 'for'.
- The argument `ip(x[i][0..2])` denotes that a variable `x[i][0..2]` need to be sent from the host computer to the accelerator, as an input parameter of the calculation. Here, `0..2` is interpreted as three indices, `0`, `1`, and `2`. That is, an expression `x[i][0..2]` is equivalent to `x[i][0]`, `x[i][1]`, `x[i][2]`. The two-dot notation, '`..`', is valid only in the Goose directives.
- The argument `jp(x[j][0..2],m[j])` denotes that variables `x[j][0..2]` and `m[j]` need to be sent from the host computer to the accelerator, as input parameters of the calculation. The difference from the argument `ip()` is that variables specified by `jp()` change their value inside the inner loop, while variables specified by `ip()` do not.
- The argument `result(a[i][0..2],pot[i])` denotes that variables `a[i][0..2]` and `pot[i]` need to be sent back from the accelerator to the host computer, as the calculation result.

- The argument `precision("double")` denotes that variables are expressed in 64-bit floating-point format. Types specified in variable declarations of C language, if any, would be ignored. valid values for the `precision()` argument are listed in section 4.1.2.

Notice:

- If the name of a function attached by a Goose-func directive is predefined as Goose-precompiled function, the user definition is ignored (*cf.* section 4.2). For example, `rsqrt()` is one of a Goose-precompiled function. Thus, the definition at the top of the sample program:

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

  is skipped by `goosecc`.

- A self-assignment operator -= is not recognized. Use += instead.

```
Example : pot[i] += -mrinv
```

- 'For' statements nested further inside a nested 'for' loop, would be unrolled. For example,

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

  is equivalent to

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

  `Goosecc` does not handle a 'for' statement whose loop-range is infinite or not determined at the compilation time. Such a statement is ignored, and `goosecc` warns about it (*cf.* section 5.4).

- Since the sample program uses a mathematical function `sqrt()`, a command-line option `-lm` is required.

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

3.5 Sample Programs

A directory `$goosepkg/sample/` contains examples of application programs which can be compiled by `goosecc`. These would be useful as reference designs to help understanding of the usage of the Goose directives.

Nested 'for' statement (equation (2))	
gravity/	Calculate gravitational interactions (used in the tutorial in section 3.4).
gravity_cutoff/	Calculate gravitational interactions with P^3M cutoff, under a periodic boundary condition.
hermite/	Calculate gravitational interactions and its time derivatives.
tree/	Calculate gravitational interactions using the Barnes-Hut Tree algorithm.
s9/	Performs a gravitational many-body simulation (the leaf-frog integrator, shared timestep).
s8/	Performs a gravitational many-body simulation (the Hermite integrator of the 4th order, individual timestep).
vdw/	Calculate van der Waals interactions (the Lennard-Jones potential).
sph/	Calculate the accelerations for SPH particles (Spline kernel, no artificial viscosity).

A single 'for' statement (equation (1))	
midpoint/	Numerically calculate $\int_0^1 4/(1+x^2)dx$ using the Midpoint method (used in the tutorial in section 3.3).
mesh1d/	On each grid-point in a one-dimensional domain space, calculate a weighted average of a given function.

In each directory, type `make all` to generate an executable for a hardware accelerator. An executable that performs the calculation without the accelerator, is also generated.

4 Goose Directives

Goose recognizes two directives below:

Goose-for directive : `#pragma goose parallel for` [*optional-arguments*]
Inserted just before a 'for' statement to make the statement run on a hardware accelerator.

Goose-func directive : `#pragma goose func`
Inserted just before a function definition, so that the function can be used inside a 'for' statement which a Goose-for directive is attached to.

> Notice : Goose directive must be written in a single line. Otherwise the line-feed character must be escaped by a backslash.
>
> ```
> Expample : #pragma goose \
>            parallel for
> ```

4.1 Goose-for Directive

A Goose-for directive is inserted just before a 'for' statement. It directs the Goose to make the statement run on a hardware accelerator. It is usually attached to a nested 'for' statement, although it can be attached to a single 'for'. 'For' statements further inside the nested 'for' loop, would be unrolled.

```
Example :
  #pragma goose parallel for
    for (i = 0; i < ni; i++) {        // outer 'for'.
      for (j = 0; j < nj; j++) {      // inner 'for'.
        for (k = 0; k < 3; k++) {     // 'for' to be unrolled.
          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++) {     // 'for' to be unrolled.
          a[i][k] += mf * dx[k];
        }
        p[i] -= m[j] * rinv;
      }
    }
```

Types of all variables used by the accelerator are `double`, or the one specified by `#pragma goose parallel for precision` (*cf.* section 4.1.2). Types specified in variable declarations of C language, if any, would be ignored.

4.1.1 I/O Type of Variables

The host computer and the hardware accelerator communicate the values of variables used inside a 'for' statement attached by a Goose-for directive. Depending on the communication required, each variable is assigned an "I/O type" attribute.

I/O Types

type	description
ip	Input parameters which depend on i, that is, $\vec{x_i}$ in equation (2). Sent from the host computer to the accelerator.
jp	Input parameters which depend on j, that is, $\vec{x_j}$ in equation (2). Sent from the host computer to the accelerator.
shared_ro	Input parameters which depend neither on i nor j. Sent from the host computer to the accelerator.
private	Intermediate variables temporarily used during the calculation. Used only on the accelerator, and not sent back to the host computer.
result	Results of the calculation, that is, $\vec{r_i}$ in equation (2). Sent back from the accelerator to the host computer.

The I/O type of a variable is automatically inferred by `goosecc` following the rules shown below. However, you can explicitly specify the I/O type by giving an argument to the Goose-for directive (*cf.* section 4.1.2).

I/O Type Inference Rules:

Letting the loop counter of the outer and inner 'for' being `i` and `j`,

- A variable indexed by `i` is inferred as a variable of I/O type `ip`. If the variable is used as an lvalue, however, it is inferred as a `result` type.
- A variable indexed by `j` is inferred as a `jp` type.
- A variable used as an lvalue is inferred as a `private` type, if none of above mentioned conditions are met. It can be recognized as a `result` type, if explicitly specified by an argument to the Goose-for directive. A variable `sum` in `$goosepkg/sample/midpoint/midpoint.c` is such an example (*cf.* section 3.3).
- A variable never used as an lvalue is inferred as a `shared_ro` type.

4.1.2 Optional Arguments

A Goose-for directive may take various optional arguments including I/O type specifiers. Below is the complete list of valid arguments.

Optional Arguments:

synopsis	description
`icnt(`*var*`)`	Use a variable *var* as the loop counter of the outer 'for' statement. Default value is `i`.
`jcnt(`*var*`)`	Use a variable *var* as the loop counter of the inner 'for' statement. Default value is `j`.
`result(`*var, ...*`)`	Set the I/O type of a variable *var* to `result`.
`ip(`*var, ...*`)`	Set the I/O type of a variable *var* to `ip`.
`jp(`*var, ...*`)`	Set the I/O type of a variable *var* to `jp`.
`private(`*var, ...*`)`	Set the I/O type of a variable *var* to `private`.
`shared_ro(`*var, ...*`)`	Set the I/O type of a variable *var* to `shared_ro`.
`precision("`*prec*`")`	Set the numerical format of variables to "*prec*". Valid values as "*prec*" are: "`double`" for 64-bit floating-point and "`double-single`" for 64-bit floating-point addition and 32-bit floating-point multiplication. Two more values, "`single`" (32-bit floating-point) and "`quadruple`" (128-bit floating-point) will be supported soon. The default value is "`double`".
`asmfile("`*filename*`")`	Specify an assembly code passed on to the assembler. If given, the code is used to build an executable, instead of the one generated by `goosecc` (*e.g.,* `foo_0.vsm` *cf.* section 6.2). This argument is used when you need to hand-tune the assembly code.

(the list continues to the next page...)

Optional Arguments (...continued from the previous page):

synopsis	description
`njp_write(`var`)`	When given, only the first var variables of `jp` type is sent to the accelerator. Variables with index `j` larger than, or equal to var are not sent to the accelerator. Their values remain unchanged from the previous calculation and reused. By doing this, the amount of data transfer would be reduced ($cf.$ `$goosepkg/sample/s8/`). All `jp`-type variables are sent to the accelerator, if this argument is not given.
`nip_pack(`$uint$`)`	Set the amount of calculation processed by a single processor element (PE). If this argument is not given, one PE processes calculations for $nvec$ indices (`i`-s) (Here, $nvec$ is the vector loop length of the PE). When given, one PE processes $uint \times nvec$ indices. Effective performance of a calculation may be improved when $uint$ is set to around 2$\sim$5, since the communication overhead between the host computer and the accelerator is hidden.

Notice:

- In order to specify the I/O types of multiple variables, give a comma-separated list of variables to the type specifier.

  ```
  Example :  ip(a, b, c)
  ```

- Strictly speaking, an I/O-type attribute is assigned not to a variable but to a value referenced by a variable. Therefore, the attribute is assigned to, for example, an array element, dereference of a pointer, and a member of a structure.

  ```
  Example :
       double x[255];
       int index[255], j;
       struct a_struct_t a_struct, *a_struct_p;
       ip(x[0]);                  // array element.
       ip(x[i]);                  // array element with variable index.
       ip(x[index[i]]);           // array element with index expressed
                                  // using another array element.
       jp(*(x+j));                // dereference of a pointer.
       jp(a_struct.a_member);     // a member of a structure.
       jp(a_struct_p->a_member); // a member of a structure refered
                                  // by a pointer.
  ```

- A range of an array can be specified by a two-dot notation, '..'.

  ```
  Example : ip(x[i][0..2])
  ```

  is equivalent to the below.

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

- The hardware accelerator and the host computer do not share any address space. Therefore, the address of variables cannot be passed on to each other. The I/O-type specifier is not valid for reference to an address.

  ```
  Example:
       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 Directive

Goose-func directive, `#pragma goose func`, should be attached to a function, if the function is used inside a 'for' loop, which a Goose-for directive is attached to. There are some restrictions for a function to be attached by a Goose-func directive: The function should have

exactly one argument; It should explicitly return a value by a `return` statement; A Goose-func directive cannot be attached to a function of type `void`, or one without `return` statement.

The value of argument and the returning value are treated as `double`, or the numerical format specified by `#pragma goose parallel for precision` (*cf.* section 4.1.2). Types specified in variable declarations of C language, if any, would be ignored.

Example :

```
#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;
}
```

If the name of a function attached by a Goose-func directive is predefined as Goose-precompiled function, the user definition is ignored and the precompiled one is used. Below is the complete list of Goose-precompiled functions:

- `rsqrt(x)` : Returns the inverse square root of the argument `x`, that is, $1/\sqrt{x}$.

5 C-Language Grammar Handled by Goose

Although goosecc handles a program written in C, it does not recognize all grammars defined by the language specification. In this section, C-language grammars which can be handled by goosecc are described.

5.1 Assign Statement

Goosecc can compile most of assign statements.

```
Example :
  r2 = dx[0]*dx[0] + dx[1]*dx[1] + dx[2]*dx[2];
  a[i][k] += mf * dx[k]; // a self assignment.
  double eps = 0.01;     // a variable declaration with an initializer.
  rinv = rsqrt(r2);      // an assignment of a returning value of a function.
```

An assign expression can be evaluated as a value.

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

5.2 If Statement

An 'if' statement can be compiled if its conditional expression matches the following form:

$$var0 \;\; rel\_op \;\; var1$$

Here, $var0$ and $var1$ are variables or number literals,
rel_op is a relational operator one of : $<, <=, >, >=, ==$, and $!=$.

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

The statements can be nested.

```
Example :
  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 Conditional Operator

A conditional operator can be compiled if its conditional expression matches the following form:

var0 rel_op var1

Here, *var0* and *var1* are variables or number literals,
rel_op is a relational operator one of : $<, <=, >, >=, ==$, and $!=$.

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

The operators can be nested.

```
Example :
  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 Statement

A 'For' statement further inside a nested 'for' loop attached by a Goose-for directive, or one inside a function attached by a Goose-func directive, would be unrolled. If the 'for' statement is nested, all the loops are recursively unrolled.

A 'for' statement can be unrolled only if its loop-range and stride is fixed at the compilation time. Otherwise the statement is ignored, and goosecc warns about it. More precisely, a 'for' statement is unrolled, if it matches the following form:

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

Here, *var* is a variable, *init_val*, *final_val*, and *inc_val* are number literals, and *rel_op* is a relational operator one of : $<$ and $<=$. The equation $var = var + inc\_val$ can be written as $var \mathrel{+}= inc\_val$. If *inc_val* is unity, it can be written as $var{+}{+}$, too.

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

This 'for' statement is unrolled as:

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

5.5 Return Statement

A 'return' statement can be used only inside a function attached by a Goose-func directive. It cannot be used inside a 'for' statement attached by a Goose-for directive.

5.6 Statements Not Handled by Goose

When goosecc find a statement that it cannot handle, it warns about it and continue the compilation process. The statement is just ignored. Therefore, for example, a function call to

`printf()` which was introduced just for debugging purpose, may not affect the calculation on the hardware accelerator.

6 Goose Inside

6.1 Goosecc Command-Line Arguments

Synopsis:

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

inputfile(s)... are source files written in C. Valid *options* are as follows:

Command-Line Arguments:

arguments	description (the default value is denoted by [ ].)
–goose-arch $< arch >$	Architecture type of the hardware accelerator. [gdr]
–goose-backannotate	Suppress generation of assembly codes. Used to avoid hand-tuned assembly codes are over written.
-o $<outputfile >$	Name of executable file to be generated. [a.out]
-I $<headerpath>$	Search path for header files.
–verbose [$= level$] -v [*level*]	Be verbose. The level may optionally be given. The higher level gives the more verbose messages. The minimum value is 0, the default is 2.
–help -h	Print help message.
-Wcc "*options*"	Explicitly specify options passed on to the C compiler.

All options not listed above are not recognized by goosecc, and implicitly passed on to the C compiler.

6.2 External Commands Used by goosecc

Goosecc invokes several external commands to build an executable. Here we briefly describe this internal procedure, although a user do not need to take care about it.

At the beginning, goosecc scans files given by command-line arguments and have file extension '.c'. If goosecc finds one or more Goose directives in a file, the file is passed to a C-language front end (goosec2q). Next the output of goosec2q is fed to a compiler for the Goose intermediate-representation (LSUMP), and then processed by an assembler (vsm) and by a C compiler (gcc).

If no Goose directive is found in an input file, goosecc passes it directly to the C compiler. Files with extensions other than '.c' are also passed to the C compiler.

The C compiler takes in all inputs from other programs, and compiles and links them to generate an executable.

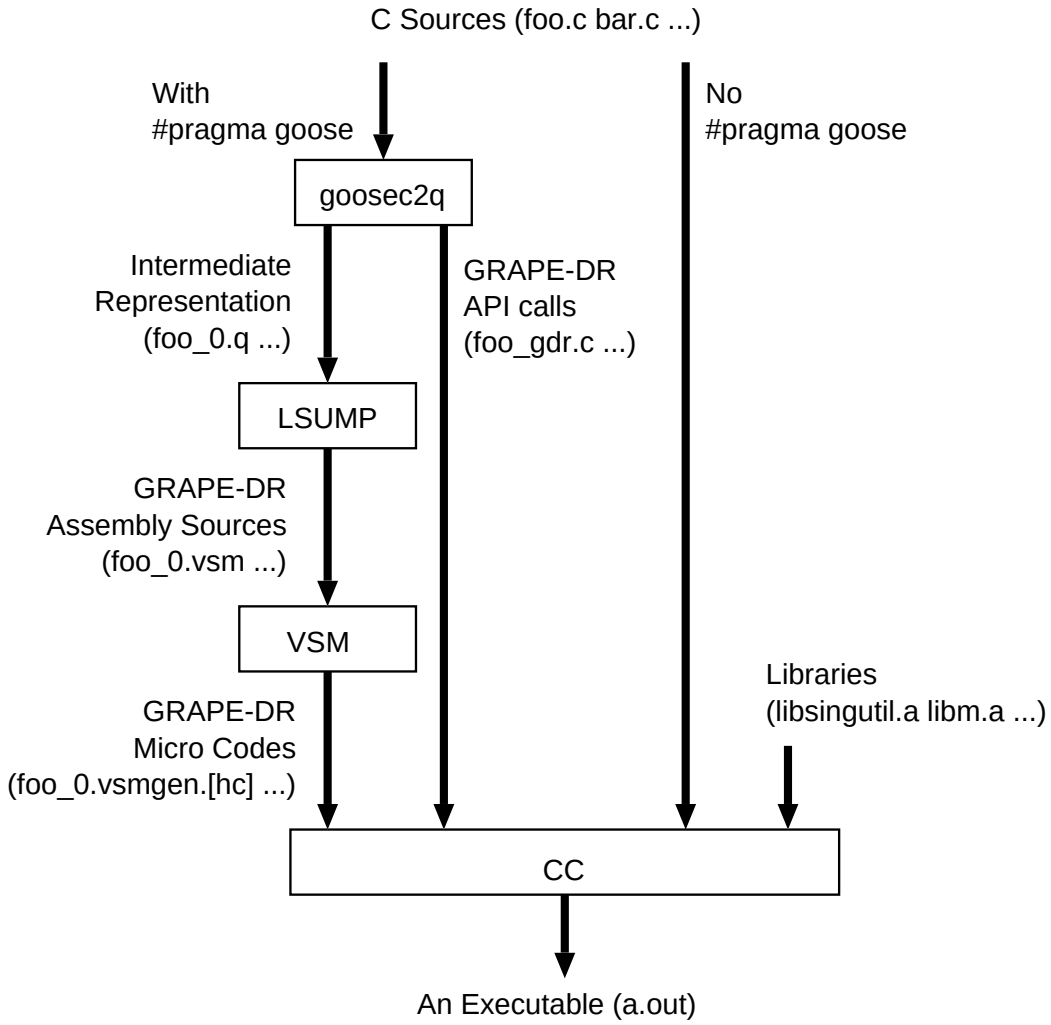

Figure : `Goosecc`-internal compilation procedure
(For the case GRAPE-DR is chosen as the hardware accelerator).

6.3 Architecture of the Hardware Accelerators

In this section, architectures of the hardware accelerators are briefly described. The description would help Goose users to develop efficient program targeting a given accelerator.

6.3.1 GRAPE-DR

GRAPE-DR is a hardware accelerator developed by K&F Computing Research. Co. It houses GRAPE-DR processor chip, a custom LSI developed by the University of Tokyo and National Astronomical Observatory of Japan. A GRAPE-DR model450 board houses one chip per board, while a model1800 houses four.

In a single GRAPE-DR chip, 512 vector-processor elements (hereafter PE) are integrated. Each PE has a multiplier, an adder, register files, and a local memory. It processes an instruction stream in SIMD fashion. The maximum loop length of the vector processor is 4.

Each 32 PEs are grouped into 16 broadcast blocks. Each broadcast block has a broadcast memory, whose contents can be broadcasted to all 32 PEs in the same block.

Outputs of the 16 broadcast blocks are connected each other, via result-reduction tree network. Each block outputs 128 ($= 32$ PEs $\times$ loop length 4) calculation results. Results from all 16 blocks, 2048 (128×16) in total, are reduced into 128 by adders (or logical operators), and written out to outside of the chip.

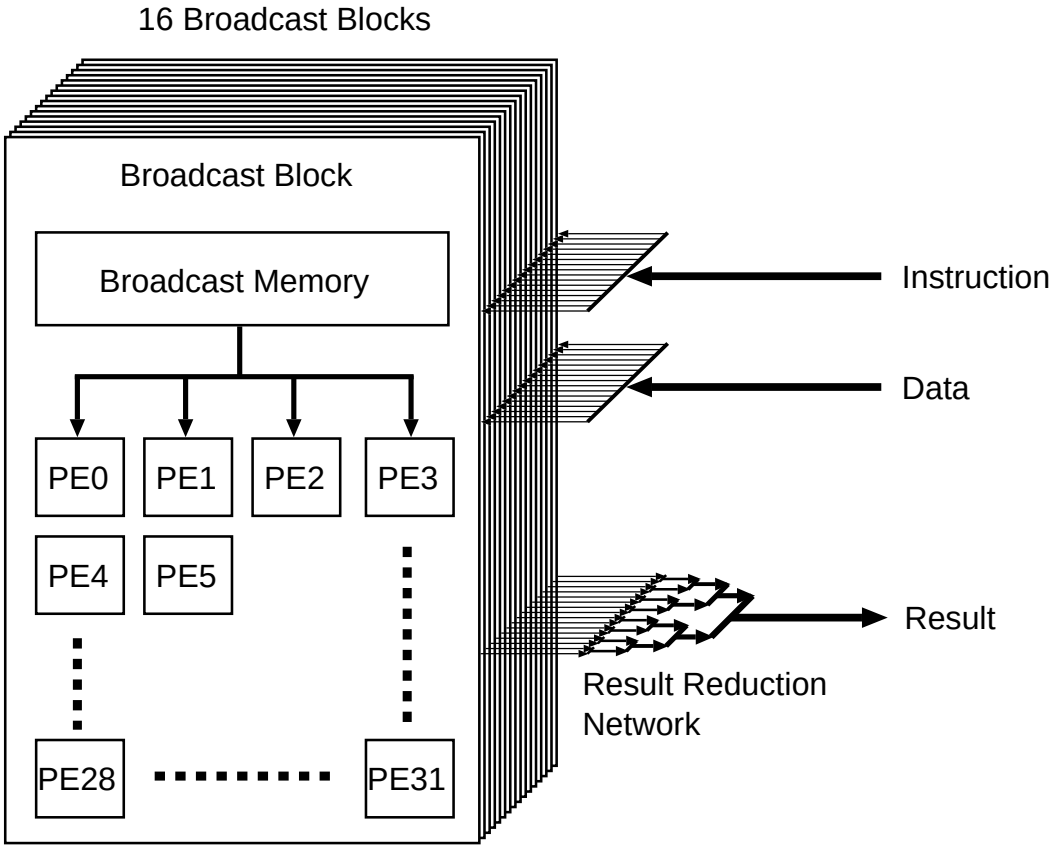

Figure : A block diagram of the GRAPE-DR chip.

A Nested 'for' Statement (equation (2)) on GRAPE-DR:

In the following, we use a nested 'for' statement below (the same one used in section 3.4) to describe a calculation procedure on GRAPE-DR.

```
#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);
```

(step 1) The host computer writes `n` variables of I/O type `jp`, namely `x[j][0..2]` and `m[j]`, into the on-board memory of GRAPE-DR.

(step 2) The host computer writes variables of I/O type `ip`, $i.e.$, `x[i][0..2]`, into the local memory of each PE. Here, among `n` sets of `x[i][0..2]` for different `i`-s, only 128 sets are written. They are broadcasted to all 16 broadcast blocks. For each block, the 128 sets are split into 32 pieces, each contains 4 sets ($i.e.$, loop-length of the vector processor), and each 4 sets are written to one of the 32 PEs (PE0 $\sim$ PE31). Note that the same 128 sets are written to all broadcast blocks.

(step 3) Before starting the calculation, 16 `x[j][0..2]` and `m[j]` are read out from the on-board memory and each of them is stored into each broadcast memory. Different `x[j][0..2]` and `m[j]` are written to different blocks.

(step 4) When the calculation starts, each broadcast memory broadcasts `x[j][0..2]` and `m[j]` to its 32 PEs. each PE uses common `x[j][0..2]` and `m[j]` and its own `x[i][0..2]` to execute the calculation. Different PEs in the same block perform calculations of gravitational forces from the same `x[j][0..2], m[j]` to different `x[i][0..2]`. Meanwhile, another 32 PEs in a different broadcast block perform calculations from different `x[j][0..2], m[j]` to the same `x[i][0..2]`.

(step 5) Each of the 16 broadcast blocks outputs calculation results `a[i][0..2]` and `pot[i]` for 128 different `i`-s. Results from 16 blocks, 128 $\times$ 16 in total, are summed up via the result-reduction tree network and reduced to 128 results.

(step 6) Apply (step 2) to (step 5) for another 128 sets of `x[i][0..2]`, until `a[i][0..2]` and `pot[i]` for all `i`-s are obtained.

This procedure utilize all 512 PEs in the chip, if the input has 128 or more i-s and 16 or more j-s. Otherwise some of the PEs are not utilized. They remain idle during the calculation.

A Single 'for' Statement (equation (1)) on GRAPE-DR:

In the following, we use a single 'for' statement below (the same one used in section 3.3) to describe a calculation procedure on GRAPE-DR.

```
#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;
  }
```

(step 1) The host computer writes variables i of I/O type ip and dx of type shared_ro into the local memory of each PE. Among n sets of i and dx, only 128 sets are written. They are broadcasted to all 16 broadcast blocks. For each block, the 128 sets are split into 32 pieces, each consists of 4 sets ($i.e.$, loop-length of the vector processor), and each 4 sets are written to one of the 32 PEs (PE0 $\sim$ PE31). Note that the same 128 sets are written to all broadcast blocks.

(step 2) When the calculation starts, Each PE uses i and dx to execute the calculation. All 16 broadcast block perform the same calculation.

(step 3) Each of the 16 broadcast blocks outputs calculation results for 128 different i-s. Results from 16 blocks, 128 $\times$ 16 in total, are summed up via the result-reduction tree network to 128. On the host computer, the results are divided by 16 and the final results are obtained.

(step 4) Apply (step 1) to (step 3) for another 128 sets of i-s, until results for all i-s are obtained.

This procedure has at least two rooms for improvement:

- All 16 broadcast blocks are performing the same calculation. This behavior can be modified so that different block perform calculation for different i-s. With this modification, calculations for 128 $\times$ 16 different i-s would be parallelized.
- A value of a shared_ro-type variable does not depend on i. Therefore, the host computer need to send it to the accelerator only once per calculation. But actually, it is resent at the beginning of each outer 'for' statement. This useless data transfer can be removed to reduce the communication time.

These two improvements are going to be implemented soon. Even after the improvements, however, calculation described in the form of equation (1) has a potential disadvantage to that of equation (2), due to the amount of data transfer ($cf.$ section 3.1).

7 License

Permission for use of the Goose Software Package (hereafter the "Software") is granted only to owners of a copy of the Software. The Software may not be redistributed. The Software may be modified by the owner, as long as the modified ones subject to this license agreement.

The copyright of the Software belongs to K&F Computing Research. Co. Programs and libraries which the Software relies on have their own licenses, copyrights, and restrictions.

8 Acknowledgement

K&F Computing Research Co. would like to thank the following people for help in development of the Goose Software Package:
Naohito Nakasato (University of Aizu) gave us a permission to use LSUMP as a compiler for the Goose intermediate-representation. Also, he kindly and quickly responded to a lot of our request for modification and extension to LSUMP. Junichiro Makino (National Astronomical Observatory of Japan) gave us a permission to use VSM, an assembler for GRAPE-DR. Satoshi Katsuyama designed the logo of Goose. We used `racc`, a fast and powerful parser generator developed by Mineroh Aoki, for development of `goosec2q`, the C-language front end of Goose (`http://i.loveruby.net/ja/projects/racc/`).

9 Modification History

version	date	description	author(s)
1.0.1α	28-Sep-2009	Document translated from the Japanese version.	A. Kawai

Contact address for questions and bug reports:
K&F Computing Research Co. (`support@kfcr.jp`)