



Compilation : le Middle End

Patrick Carribault
patrick.carribault@cea.fr



Plan du cours

- Modification du compilateur
 - Plugin
 - Événement
 - *Pass manager*
- Manipulation du code
 - Structures GIMPLE
 - Structures CFG
- Gestion des directives OpenMP dans GCC
 - Introduction à OpenMP
 - Transformation manuelle
 - Implémentation dans le compilateur GCC



Plan du cours

- Modification du compilateur
 - Plugin
 - Événement
 - *Pass manager*
- Manipulation du code
 - Structures GIMPLE
 - Structures CFG
- Gestion des directives OpenMP dans GCC
 - Introduction à OpenMP
 - Transformation manuelle
 - Implémentation dans le compilateur GCC



Modification du compilateur

- Catégories de modifications
 - Corrections de bugs
 - Ajout de fonctionnalités
- Ajout de fonctionnalités
 - Nouveau langage en entrée
 - Nouvelle architecture cible
 - Nouvelle passe (analyse/transformation/optimisation)
- Comment faire des modifications dans le compilateur GCC ?
 - Nouveau Front-end
 - Nouvelle description d'architecture (*machine description*)
 - Nouvelle passe
- Comment ajouter une nouvelle passe ?
 - Ajout direct dans le cœur du compilateur
 - Programmation d'un plugin externe



Description d'un plugin

- Plugin
 - Bout de code chargé par le compilateur au moment de la compilation d'un fichier
 - Sous forme de bibliothèque dynamique
 - Interaction avec le cœur du compilateur

- Contenu minimal d'un plugin
 - Initialisation : fonction prédéfinie qui retourne 0 si tout se passe bien
 - Licence GPL : déclaration d'une variable globale prédéfinie

```
int plugin_is_GPL_compatible ;
```

- Etapes
 1. Compilation du plugin en une bibliothèque dynamique
 2. Exécution
 - Lors de la compilation d'un fichier : renseigné l'utilisation d'un plugin
 - Possibilité de mettre des arguments



Initialisation d'un plugin

- Fonction d'initialisation du plugin
 - ```
int plugin_init (
 struct plugin_name_args *plugin_info,
 struct plugin_gcc_version *version
);
```
- **Chaque plugin doit implémenter cette fonction**
- Point d'entrée
  - Fonction `main` dans le cœur du compilateur
  - Lors de la phase d'initialisation des plugins, le compilateur appelle la fonction `plugin_init` de tous les plugins (de façon séquentielle)
- Où est défini ce prototype ?
  - Dans le *header* `gcc-plugin.h`



# Plugin minimal

---

```
#include <gcc-plugin.h>

int plugin_is_GPL_compatible ;

int
plugin_init (struct plugin_name_args *plugin_info,
 struct plugin_gcc_version *version)
{
 printf("Initialization of my plugin\n") ;

 return 0 ;
}
```



# Compilation d'un plugin

---

- Etape 1 : compilation séparée des fichiers appartenant au plugins
  - Besoin du chemin où sont stocker les *headers* servant au plugin (comme `gcc-plugin.h`)
  - Commande : `gcc -print-file-name=plugin`
    - Donne le répertoire de base pour les fichiers qui concernent les plugins
    - Besoin d'ajouter le sous-répertoire `include` pour la recherche de header (option `-I` pour le compilateur)
- Etape 2 : *link* de ces fichiers pour créer une bibliothèque dynamique
  - Utilisation de l'option `-shared`
  - Extension par convention : `.so`





# Compilation d'un plugin

---

```
carribaultp$ gcc -print-file-name=plugin
/media/sf_PartageVM/GCC/gcc_install/lib/gcc/i686-pc-linux-gnu/4.7.1/plugin
carribaultp$ gcc -I/media/sf_PartageVM/GCC/gcc_install/lib/gcc/i686-pc-linux-gnu/4.7.1/plugin/include -c plugin.c
carribaultp$ gcc -shared -o plugin.so plugin.o
```

```
carribaultp$ gcc -print-file-name=plugin
/media/sf_PartageVM/GCC/gcc_install/lib/gcc/i686-pc-linux-gnu/4.7.1/plugin
carribaultp$ gcc -I`gcc -print-file-name=plugin`/include -c plugin.c
carribaultp$ gcc -shared -o plugin.so plugin.o
```



# Exécution d'un plugin

---

- Pas d'exécution directe d'un plugin
  - Fonction `main` dans le compilateur
  - Plugin contrôlé par le compilateur
  - Le compilateur connaît un point d'entrée pour le plugin (fonction d'initialisation avec un prototype forcé)
- Option pour renseigner un plugin à utiliser lors de la compilation
  - `-fplugin=name`
  - L'argument `name` est le nom de la bibliothèque dynamique contenant le plugin (e.g., `plugin.so`)
  - Possibilité d'utiliser plusieurs plugins !



# Exécution d'un plugin

---

```
#include <gcc-plugin.h>

int plugin_is_GPL_compatible ;

int
plugin_init (struct plugin_name_args *plugin_info,
 struct plugin_gcc_version *version)
{
 printf("Initialization of my plugin\n") ;

 return 0 ;
}
```



# Exécution d'un plugin

---

```
carribaultp$ cat test.c
#include <stdio.h>

void f() {
printf("In f\n") ;
}

void g() {
printf("In g\n") ;
}
carribaultp$ gcc -fplugin=./plugin.so -c test.c
Initialization of my plugin
```



# Structures d'initialisation

---

- Rappel : fonction d'initialisation du plugin

```
int plugin_init (
 struct plugin_name_args *plugin_info,
 struct plugin_gcc_version *version
);
```

- Deux arguments en entrée de la fonction
  - Arguments fournis par le compilateur
  - Correspond à deux pointeurs sur des structures
  - Information sur le contexte d'exécution
  - Définition dans le *header* `gcc-plugin.h`



# Information sur GCC

---

- Second argument : informations sur le compilateur qui exécute ce plugin
- Détail de cette structure

```
struct plugin_gcc_version
{
 const char *basever;
 const char *datestamp;
 const char *devphase;
 const char *revision;
 const char *configuration_arguments;
};
```



# Information sur GCC

```
#include <gcc-plugin.h>

int plugin_is_GPL_compatible ;

int plugin_init (struct plugin_name_args *plugin_info,
 struct plugin_gcc_version *version) {
 printf("Plugin initialization:\n") ;
 printf("\tbasever = %s\n", version->basever) ;
 printf("\tdatestamp = %s\n", version->datestamp) ;
 printf("\tdevphase = %s\n", version->devphase) ;
 printf("\trevision = %s\n", version->revision) ;
 printf("\tconfig = %s\n", version->configuration_arguments) ;
 return 0 ;
}
```

```
carribaultp$ gcc -fplugin=./plugin.so -c test.c
```

Plugin initialization:

basever = 4.7.1

datestamp = 20120614

devphase =

revision =

config = /media/sf\_PartageVM/GCC/gcc-4.7.1/configure --disable-bootstrap --prefix=/media/sf\_PartageVM/GCC/gcc\_install/ --enable-languages=c,c++,fortran



# Information sur le plugin

---

- Premier argument : informations sur le contexte d'exécution du plugin
- Détail de cette structure

```
struct plugin_name_args {
 char *base_name; /* Short name of the plugin
 (filename without .so suffix). */
 const char *full_name; /* Path to the plugin as specified
 with -fplugin=. */
 int argc; /* Number of arguments specified with
 -fplugin-arg-.... */
 struct plugin_argument *argv; /* Array of ARGV
 key-value pairs. */
 const char *version; /* Version string provided by
 plugin. */
 const char *help; /* Help string provided by plugin. */
}
```





# Information sur le plugin

---

- Le champ `argv` représente les arguments donnés au plugin lors de l'exécution de la compilation
  - Option : `-fplugin-arg-name-key1 [=value1]`
  - Nom du plugin (sans le chemin, ni l'extension `.so`) : `name`
  - Nom de l'argument : `key1`
  - Valeur de l'argument (optionnelle) : `value1`

- Structure pour accéder aux arguments

```
struct plugin_argument {
 char *key; /* key of the argument. */
 char *value; /* value is optional and
 can be NULL. */
};
```



# Information sur le plugin

```
#include <gcc-plugin.h>

int plugin_is_GPL_compatible ;

int plugin_init (struct plugin_name_args *plugin_info,
 struct plugin_gcc_version *version) {
 int i ;
 printf("Plugin initialization:\n") ;
 printf("\tbase_name = %s\n", plugin_info->base_name) ;
 printf("\tfull_name = %s\n", plugin_info->full_name) ;
 printf("\targc = %d\n", plugin_info->argc) ;
 for (i = 0 ; i < plugin_info->argc ; i++) {
 printf("\t\tArg %d: %s = %s\n", i,
 plugin_info->argv[i].key,
 plugin_info->argv[i].value) ;
 }
 printf("\tversion = %s\n", plugin_info->version) ;
 printf("\thelp = %s\n", plugin_info->help) ;
 return 0 ;
}
```



# Information sur le plugin

```
carribaultp$ gcc -fplugin=./plugin.so -c test.c
```

```
Plugin initialization:
```

```
 base_name = plugin
 full_name = ./plugin.so
 argc = 0
 version = (null)
 help = (null)
```

```
carribaultp$ gcc -fplugin=./plugin.so -fplugin-arg-plugin-mon_arg1=toto -c test.c
```

```
cc1: error: plugin plugin should be specified before -fplugin-arg-plugin-mon_arg1=toto in the command line
```

```
Plugin initialization:
```

```
 base_name = plugin
 full_name = ./plugin.so
 argc = 0
 version = (null)
 help = (null)
```

```
carribaultp$ gcc -fplugin=./plugin.so -fplugin-arg-plugin-mon_arg1=toto -c test.c
```

```
Plugin initialization:
```

```
 base_name = plugin
 full_name = ./plugin.so
 argc = 1
 Arg 0: mon_arg1 = toto
 version = (null)
 help = (null)
```

```
carribaultp$ gcc -fplugin=./plugin.so -fplugin-arg-plugin-mon_arg1=toto -fplugin-arg-plugin-val=2 -c test.c
```

```
Plugin initialization:
```

```
 base_name = plugin
 full_name = ./plugin.so
 argc = 2
 Arg 0: mon_arg1 = toto
 Arg 1: val = 2
 version = (null)
 help = (null)
```



# Plan du cours

---

- Modification du compilateur
  - Plugin
  - Événement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG
- Gestion des directives OpenMP dans GCC
  - Introduction à OpenMP
  - Transformation manuelle
  - Implémentation dans le compilateur GCC



# Evènement d'un plugin

---

- Pour le moment, le plugin s'initialise
  - Dans cette fonction, il faut donner des infos au compilateur sur le comportement de notre plugin
- Programmation évènementielle avec des *callbacks*
  - Enregistrement d'évènements à capturer par le plugin
  - Appel d'une fonction pour l'enregistrement avec un type d'évènement
  - Ajout d'un pointeur de fonction pour désigner la fonction que le compilateur doit appeler lorsque l'évènement se produit
- Liste des évènements dans le fichier `plugin.def`



# Liste exhaustive

---

- PLUGIN\_PASS\_MANAGER\_SETUP,
- PLUGIN\_FINISH\_TYPE,
- PLUGIN\_FINISH\_DECL,
- PLUGIN\_FINISH\_UNIT,
- PLUGIN\_PRE\_GENERICIZE,
- PLUGIN\_FINISH,
- PLUGIN\_INFO,
- PLUGIN\_GGC\_START,
- PLUGIN\_GGC\_MARKING,
- PLUGIN\_GGC\_END,
- PLUGIN\_REGISTER\_GGC\_ROOTS,
- PLUGIN\_REGISTER\_GGC\_CACHES,
- PLUGIN\_ATTRIBUTES,
- PLUGIN\_START\_UNIT,
- PLUGIN\_PRAGMAS,
- PLUGIN\_ALL\_PASSES\_START,
- PLUGIN\_ALL\_PASSES\_END,
- PLUGIN\_ALL\_IPA\_PASSES\_START,
- PLUGIN\_ALL\_IPA\_PASSES\_END,
- PLUGIN\_OVERRIDE\_GATE,
- PLUGIN\_PASS\_EXECUTION,
- PLUGIN\_EARLY\_GIMPLE\_PASSES\_START,
- PLUGIN\_EARLY\_GIMPLE\_PASSES\_END,
- PLUGIN\_NEW\_PASS,
- PLUGIN\_EVENT\_FIRST\_DYNAMIC



# Evènements intéressants

---

- **PLUGIN\_PASS\_MANAGER\_SETUP**
  - Permet d'interagir avec le *pass manager* pour ajouter une nouvelle passe
- **PLUGIN\_START\_UNIT**
  - Utile pour initialiser des données (e.g., ouverture de fichiers) au début de la compilation d'un fichier
- **PLUGIN\_FINISH** ou **PLUGIN\_FINISH\_UNIT**
  - Utile pour finaliser des données (e.g., fermeture de fichiers) à la fin de la compilation d'un fichier
- **PLUGIN\_PRAGMAS**
  - Ajout de la reconnaissance d'une directive (`#pragma` en C/C++)



# Enregistrement

---

- Fonction pour enregistrer un évènement :

```
void register_callback (const char *plugin_name,
 int event,
 plugin_callback_func callback,
 void *user_data);
```
- Arguments
  - `plugin_name` : nom du plugin sans le chemin ni l'extension
    - Utilisation de `plugin_info->base_name` pour le plugin courant
  - `event` : évènement à enregistrer
  - `callback` : fonction appelée lorsque cet évènement apparaît
  - `user_data` : données utilisateurs utiles pour le callback
- Selon les évènements,
  - `callback` peut être NULL
  - `user_data` peut être NULL
- Prototype de la fonction de call back

```
void (*plugin_callback_func) (void *gcc_data, void *user_data);
```





# Exemple de *callback*

```
#include <gcc-plugin.h>

int plugin_is_GPL_compatible ;

void callback_start_unit (void *gcc_data, void *user_data) {
 printf("Callback start unit\n") ;
}

void callback_finish_unit (void *gcc_data, void *user_data) {
 printf("Callback finish unit\n") ;
}

void callback_finish (void *gcc_data, void *user_data) {
 printf("Callback finish\n") ;
}

int plugin_init (struct plugin_name_args *plugin_info,
 struct plugin_gcc_version *version) {

 register_callback (plugin_info->base_name,
 PLUGIN_START_UNIT,
 callback_start_unit,
 NULL);

 register_callback (plugin_info->base_name,
 PLUGIN_FINISH_UNIT,
 callback_finish_unit,
 NULL);

 register_callback (plugin_info->base_name,
 PLUGIN_FINISH,
 callback_finish,
 NULL);

 return 0 ;
}
```



# Exemple de *callback*

---

```
carribaultp$ gcc -fplugin=./plugin.so -c test.c
Callback start unit
Callback finish unit
Callback finish
carribaultp$ gcc -fplugin=./plugin.so -c test.c test2.c
Callback start unit
Callback finish unit
Callback finish
Callback start unit
Callback finish unit
Callback finish
```



# Plan du cours

---

- Modification du compilateur
  - Plugin
  - Événement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG
- Gestion des directives OpenMP dans GCC
  - Introduction à OpenMP
  - Transformation manuelle
  - Implémentation dans le compilateur GCC



# Création d'une nouvelle passe

---

- Etapes pour ajouter une nouvelle passe
  1. Définition d'une nouvelle passe
  2. Insertion dans le *pass manager*
  3. Enregistrement de l'évènement associé pour le plugin



# 1 - Définition d'une passe

---

- Structure d'une passe
  - Définie dans `tree-pass.h`
  - Localisée dans répertoire `include` des plugins
- Nom : `struct opt_pass`
- Champs intéressants :
  - Type de passes (voir ci-après)  
`enum opt_pass_type type`
  - Nom de la passe  
`const char *name;`
  - Fonction pour la décision d'exécution  
`bool (*gate) (void) ;`
  - Fonction d'exécution de la passe  
`unsigned int (*execute) (void) ;`



# 1 - Définition d'une passe

---

- Type de passe (différences ?)

```
enum opt_pass_type {
 GIMPLE_PASS,
 RTL_PASS,
 SIMPLE_IPA_PASS,
 IPA_PASS
};
```

- Fonction pour la décision d'exécution

- Retourne un booléen
- Permet d'activer la passe seulement dans un contexte particulier (niveau d'optimisation, option, ...)

- Fonction d'exécution de la passe

- Corps de la passe proprement dite
- Cette fonction n'est appelée que si la *gate* a répondu VRAI



## 2 – Insertion de la passe

---

- Interaction avec le pass manager
  - Une fois notre passe définie, besoin de l'insérer dans le processus de compilation
  - Besoin également de donner plusieurs infos (e.g., la fréquence de décisions)
- Structure permettant de renseigner les informations sur la passe et son insertion

```
struct register_pass_info {
 struct opt_pass *pass; /* New pass provided by the plugin. */
 const char *reference_pass_name; /* Name of the reference pass
 for hooking up the new pass. */
 int ref_pass_instance_number; /* Insert the pass at the specified
 instance number of the reference pass. */
 /* Do it for every instance if it is 0. */
 enum pass_positioning_ops pos_op; /* how to insert the new pass. */
};
```

- Positionnement de la passe

```
enum pass_positioning_ops {
 PASS_POS_INSERT_AFTER, // Insert after the reference pass.
 PASS_POS_INSERT_BEFORE, // Insert before the reference pass.
 PASS_POS_REPLACE // Replace the reference pass.
};
```



## 3 - Enregistrement

---

- Utilisation de l'évènement  
`PLUGIN_PASS_MANAGER_SETUP`
- Appel à la fonction d'enregistrement  
`register_callback`
  - Fonction de `callback` → `NULL`
    - Fonctions nécessaires pour décider et exécuter la passe sont contenues dans l'instance de la structure `opt_pass`
  - Pointeur `user_data` → pointeur sur une structure pour renseigner les informations sur la passe (adresse sur instance de `register_pass_info`)





# Exemple

```
bool gate_my_pass (void) {
 return true ;
}

unsigned int execute_my_pass (void) {
 printf("Executing my_pass with function %s\n",
 get_name (current_function_decl)) ;
 return 0 ;
}

struct opt_pass my_pass = {
 GIMPLE_PASS, /* type */
 "my_pass", /* name */
 gate_my_pass, /* gate */
 execute_my_pass, /*execute */
 NULL, /* sub */
 NULL, /* next */
 0, /* static_pass_number */
 TV_NONE, /* tv_id */
 PROP_gimple_any, /* properties_required */
 0, /* properties_provided */
 0, /* properties_destroyed */
 0, /* todo_flags_start */
 0 /* todo_flags_finish */
} ;
```

```
int plugin_init (struct plugin_name_args *plugin_info,
 struct plugin_gcc_version *version) {

 struct register_pass_info pass_info;

 pass_info.pass = &my_pass ;
 pass_info.reference_pass_name = "omplower" ;
 pass_info.ref_pass_instance_number = 0 ;
 pass_info.pos_op= PASS_POS_INSERT_BEFORE ;

 register_callback (plugin_info->base_name,
 PLUGIN_PASS_MANAGER_SETUP,
 NULL,
 &pass_info);

 return 0 ;
}
```



# Example

---

```
carribaultp$ cat test.c
#include <stdio.h>
```

```
void f() {
printf("In f()\n") ;
}
```

```
int main() {
printf("Hello\n") ;
f() ;
return 0 ;
}
```

```
carribaultp$ gcc -fplugin=./plugin.so test.c
Executing my_pass with function main
Executing my_pass with function f
```



# Pass Manager

---

- Nécessité de connaître l'ordre des passes exécutées par GCC !
- Besoin de regarder le code du *pass manager*
  - Dans les sources du compilateur
  - Pas disponible dans les *headers* relatifs aux plugins
- *Pass manager*
  - Sous-répertoire `gcc`
  - Fichier `passes.c`
  - Fonction `init_optimization_passes`
- Plusieurs types de passes
  - *Lowering, IPA, All passes, ...*



# Pass Manager

```
void
init_optimization_passes (void)
{
 struct opt_pass **p;

#define NEXT_PASS(PASS) (p = next_pass_1 (p, &((PASS).pass)))

 /* All passes needed to lower the function into shape optimizers can
 operate on. These passes are always run first on the function, but
 backend might produce already lowered functions that are not processed
 by these passes. */
 p = &all_lowering_passes;
 NEXT_PASS (pass_warn_unused_result);
 NEXT_PASS (pass_diagnose_omp_blocks);
 NEXT_PASS (pass_diagnose_tm_blocks);
 NEXT_PASS (pass_mudflap_1);
 NEXT_PASS (pass_lower_omp);
 NEXT_PASS (pass_lower_cf);
 NEXT_PASS (pass_lower_tm);
 NEXT_PASS (pass_refactor_eh);
 NEXT_PASS (pass_lower_eh);
 NEXT_PASS (pass_build_cfg);
 NEXT_PASS (pass_warn_function_return);
 NEXT_PASS (pass_build_cgraph_edges);
 *p = NULL;

 /* Interprocedural optimization passes. */
 p = &all_small_ipa_passes;
 NEXT_PASS (pass_ipa_free_lang_data);
 NEXT_PASS (pass_ipa_function_and_variable_visibility);
 NEXT_PASS (pass_early_local_passes);
```

Notre plugin  
a inséré une  
passe à cet  
endroit



# Nom des passes

- Une fois la position trouvée, il manque une information
  - Le nom de la passe de référence
- Comment trouver ce nom ?
  - Pas de solution immédiate simple
  - Besoin de regarder la structure qui définit cette passe
  - Si vous trouvez une meilleure solution...
- Exemple : fichier `omp-low.c`

```
struct gimple_opt_pass pass_lower_omp =
{
 GIMPLE_PASS,
 "lompower", /* name */
 NULL, /* gate */
 execute_lower_omp, /* execute */
 NULL, /* sub */
 NULL, /* next */
 0, /* static_pass_number */
 TV_NONE, /* tv_id */
 PROP_gimple_any, /* properties_required */
 PROP_gimple_lomp, /* properties_provided */
 0, /* properties_destroyed */
 0, /* todo_flags_start */
 0 /* todo_flags_finish */
}
};
```



# Plan du cours

---

- Modification du compilateur
  - Plugin
  - Événement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG
- Gestion des directives OpenMP dans GCC
  - Introduction à OpenMP
  - Transformation manuelle
  - Implémentation dans le compilateur GCC



# Manipulation du code

---

- Focalisation sur les passes en GIMPLE
  - Code source du fichier à compiler représenté en GIMPLE
  - GIMPLE est notre représentation intermédiaire
- Gestion du code en GIMPLE
  - Deux étapes : avant et après la création du graphe de flot de contrôle (CFG)
- Avant la création du CFG :
  - Tout peut être fait grâce à un traitement itératif en GIMPLE (récursif)
  - Documentation : `gimple.def` `gimple.h` `gimple.c`
- Après la création du CFG :
  - Accès au code à travers le graphe de flot de contrôle

# Fichier source en GIMPLE

```
#include <stdio.h>
```

```
int f(int a, int * m) {
 int i ;
 int b = a ;
 if (a) {
 b++ ;
 }
 else {
 for (i = 0 ; i < a ; i++) {
 b += m[i] ;
 }
 }
 return b ;
}
```

```
f (int a, int * m)
gimple_bind <
 unsigned int i.0;
 unsigned int D.1823;
 int * D.1824;
 int D.1825;
 int D.1826;
 int i;
 int b;

 gimple_assign <parm_decl, b, a, NULL>
 gimple_cond <ne_expr, a, 0, <D.1819>, <D.1820>>
 gimple_label <<D.1819>>
 gimple_assign <plus_expr, b, b, 1>
 gimple_goto <<D.1821>>
 gimple_label <<D.1820>>
 gimple_assign <integer_cst, i, 0, NULL>
 gimple_goto <<D.1816>>
 gimple_label <<D.1815>>
 gimple_assign <nop_expr, i.0, i, NULL>
 gimple_assign <mult_expr, D.1823, i.0, 4>
 gimple_assign <pointer_plus_expr, D.1824, m, D.1823>
 gimple_assign <mem_ref, D.1825, *D.1824, NULL>
 gimple_assign <plus_expr, b, D.1825, b>
 gimple_assign <plus_expr, i, i, 1>
 gimple_label <<D.1816>>
 gimple_cond <lt_expr, i, a, <D.1815>, <D.1817>>
 gimple_label <<D.1817>>
 gimple_label <<D.1821>>
 gimple_assign <var_decl, D.1826, b, NULL>
 gimple_return <D.1826>
>
```





# Fichier source avec le CFG

```
;; Function f (f, funcdef_no=0, decl_uid=1811, cgraph_uid=0)
```

```
f (int a, int * m)
```

```
{
 int b;
 int i;
 int D.1826;
 int D.1825;
 int * D.1824;
 unsigned int D.1823;
 unsigned int i.0;
```

```
<bb 2>:
 gimple_assign <parm_decl, b, a, NULL>
 gimple_cond <ne_expr, a, 0, NULL, NULL>
 goto <bb 3>;
 else
 goto <bb 4>;
```

```
<bb 3>:
 gimple_assign <plus_expr, b, b, 1>
 goto <bb 7>;
```

```
<bb 4>:
 gimple_assign <integer_cst, i, 0, NULL>
 goto <bb 6>;
```

```
<bb 5>:
```

```
 gimple_assign <nop_expr, i.0, i, NULL>
 gimple_assign <mult_expr, D.1823, i.0, 4>
 gimple_assign <pointer_plus_expr, D.1824, m, D.1823>
 gimple_assign <mem_ref, D.1825, *D.1824, NULL>
 gimple_assign <plus_expr, b, D.1825, b>
 gimple_assign <plus_expr, i, i, 1>
```

```
<bb 6>:
```

```
 gimple_cond <lt_expr, i, a, NULL, NULL>
 goto <bb 5>;
 else
 goto <bb 7>;
```

```
<bb 7>:
```

```
 gimple_assign <var_decl, D.1826, b, NULL>
```

```
gimple_label <<L6>>
 gimple_return <D.1826>
```

```
}
```



# Exemple de passe GIMPLE

- Première passe de *lowering* :  
warn\_unused\_result
  - Affiche un warning lorsque le retour d'un appel de fonction n'est pas capturé et que cette fonction a un attribut warn\_unused\_result
- Fonction
  - do\_warn\_unused\_result(gimple\_body(current\_function\_decl));
- current\_function\_decl
  - Pointeur vers la racine de la déclaration de la fonction courante
  - Tout le corps de la fonction (ainsi que les arguments, le retour et les variables locales) est accessible à partir de ce pointeur
- Accès à la représentation gimple de la fonction
  - gimple\_seq gimple\_body(tree t)

```
static unsigned int
run_warn_unused_result (void)
{
 do_warn_unused_result (gimple_body (current_function_decl));
 return 0;
}

static bool
gate_warn_unused_result (void)
{
 return flag_warn_unused_result;
}

struct gimple_opt_pass pass_warn_unused_result =
{
 {
 GIMPLE_PASS,
 "warn_unused_result", /* name */
 gate_warn_unused_result, /* gate */
 run_warn_unused_result, /* execute */
 NULL, /* sub */
 }
```

# Exemple de passe GIMPLE

Boucle  
d'itération  
sur les  
statements

Accès au  
type  
d'instruction

Type de nœud  
possédant un  
ensemble de  
statements

```
static void
do_warn_unused_result (gimple_seq seq)
{
 tree fdecl, ftype;
 gimple_stmt_iterator i;

 for (i = gsi_start (seq); !gsi_end_p (i); gsi_next (&i))
 {
 gimple g = gsi_stmt (i);

 switch (gimple_code (g))
 {
 case GIMPLE_BIND:
 do_warn_unused_result (gimple_bind_body (g));
 break;
 case GIMPLE_TRY:
 do_warn_unused_result (gimple_try_eval (g));
 do_warn_unused_result (gimple_try_cleanup (g));
 break;
 case GIMPLE_CATCH:
 do_warn_unused_result (gimple_catch_handler (g));
 break;
 case GIMPLE_EH_FILTER:
 do_warn_unused_result (gimple_eh_filter_failure (g));
 break;

 case GIMPLE_CALL:
 if (gimple_call_lhs (g))
 break;
 if (gimple_call_internal_p (g))
 break;
 }
 }
}
```

Itérateur de  
*statement*

Accès à la  
représentation  
GIMPLE du  
statement

Appel récursif



# Construction du CFG

---

- Une fois le graphe de flot de contrôle (CFG) construit
  - Passage par la structure du CFG pour la manipulation du code
  - Ensemble de nœuds et d'arcs
- Passe qui construit le CFG
  - Nom dans le *pass manager* : `pass_build_cfg`
  - Nom de la passe : `cfg`
- Structures utilisées pour le CFG
  - `struct control_flow_graph`
  - CFG de la fonction courante :  
`cfun->cfg`



# Structure principale du CFG

```
struct GTY(()) control_flow_graph {
 /* Block pointers for the exit and entry of a function.
 These are always the head and tail of the basic block list. */
 basic_block x_entry_block_ptr;
 basic_block x_exit_block_ptr;

 /* Index by basic block number, get basic block struct info. */
 VEC(basic_block,gc) *x_basic_block_info;

 /* Number of basic blocks in this flow graph. */
 int x_n_basic_blocks;

 /* Number of edges in this flow graph. */
 int x_n_edges;

 /* The first free basic block number. */
 int x_last_basic_block;

 /* UUIDs for LABEL_DECLS. */
 int last_label_uid;

 /* Mapping of labels to their associated blocks. At present
 only used for the gimple CFG. */
 VEC(basic_block,gc) *x_label_to_block_map;

 enum profile_status_d x_profile_status;

 /* Whether the dominators and the postdominators are available. */
 enum dom_state x_dom_computed[2];

 /* Number of basic blocks in the dominance tree. */
 unsigned x_n_bbs_in_dom_tree[2];

 /* Maximal number of entities in the single jumtable. Used to estimate
 final flowgraph size. */
 int max_jumtable_ents;
};
```



# Noeuds du CFG

---

- Basic block

```
typedef struct basic_block_def *basic_block;
```
- Fichiers concernés :
  - `coretypes.h`, `basic-block.h`
- Champs
  - Vecteurs d'arc (edge) entrant et sortant : `preds`, `succs`
  - Double liste chaînée : `prev_bb`, `next_bb`
  - Indice dans le vecteur des BBs : `index`
  - Ensemble de flags...
- Notion de vecteurs : cf. `vec.h` pour plus d'infos...
- Deux BB spéciaux (source et puits)
  - `ENTRY_BLOCK_PTR`
  - `EXIT_BLOCK_PTR`



# Arcs du CFG

---

- Edge

```
typedef struct edge_def *edge;
```

- Fichiers concernées :

- `coretypes.h, basic-block.h`

- Champs

- Source de l'arc : `src`
- Destination de l'arc : `dest`
- Indice dans le vecteur de destination : `dest_idx`
- Ensemble de flags...



# Parcours du CFG

---

- Parcours du corps de la fonction à travers le CFG
  - Possibilité d'itérer sur les nœuds
  - Ensuite, sur les arcs
- Plusieurs solutions pour le parcours des nœuds
  - Tous les bloc de base sauf source et puits (peu importe leur ordre)

```
basic_block bb;
FOR_EACH_BB (bb) { /* ... */ }
```
  - Tous les bloc de base (peu importe leur ordre)

```
basic_block bb;
FOR_ALL_BB (bb) { /* ... */ }
```
  - Commencer au premier BB

```
basic_block bb = ENTRY_BLOCK_PTR ;
```





# Parcours du CFG

---

- Exemple simple du parcours du CFG
  - Itération sur tous les BBs (sauf source et puits)
  - Ordre non défini

```
gimple_stmt_iterator gsi;
gimple stmt;
```

```
FOR_EACH_BB (b)
{
 for (gsi = gsi_start_bb (b); !gsi_end_p (gsi); gsi_next
 (&gsi))
 {
 stmt = gsi_stmt (gsi);
 /* ... */
 }
}
```



# Example

---

```
unsigned int execute_my_pass (void) {
 basic_block bb ;
 gimple_stmt_iterator gsi;
 gimple stmt;

 printf("Function %s w/ %d BB(s)\n",
 get_name (current_function_decl),
 n_basic_blocks) ;

 FOR_EACH_BB (bb)
 {
 printf("BB #%d\n", bb->index) ;
 for (gsi = gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi)) {
 printf("\tStatement\n") ;
 stmt = gsi_stmt (gsi);
 debug_gimple_stmt(stmt) ;
 }
 }
 return 0 ;
}
```



# Example

```
carribaultp$ cat test.c
#include <stdio.h>

int f(int a) {
 int b = a ;
 if (a) {
 printf("A is not 0\n") ;
 {
 b = a + 1 ;
 }
 }
 return b ;
}
```

```
carribaultp$ gcc -fplugin=./plugin.so -c test.c
Function f w/ 6 BB(s)
BB #2
 Statement
b = a;

 Statement
if (a != 0)

BB #3
 Statement
__builtin_puts (&"A is not 0"[0]);

 Statement
b = a + 1;

BB #4
 Statement
D.1816 = b;

BB #5
 Statement
<L2>:

 Statement
return D.1816;
```



# Plan du cours

---

- Modification du compilateur
  - Plugin
  - Événement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG
- Gestion des directives OpenMP dans GCC
  - Introduction à OpenMP
  - Transformation manuelle
  - Implémentation dans le compilateur GCC



# Introduction à OpenMP

---

- Modèle de programmation parallèle
  - Exploitation du parallélisme de données
  - Exploitation du parallélisme de tâches (depuis OpenMP 3.0)
- Mémoire partagée
  - Exécution sur un nœud de calcul
  - Possibilité d'extension à de la mémoire distribuée (DSM – Distributed Shared Memory)
    - Par exemple Intel Cluster OpenMP
- Basé sur des threads
- Exemple de coopération entre un compilateur et une bibliothèque



# Historique d'OpenMP

---


- Gestion par l'OpenMP ARB (*Architecture Review Board*)
- OpenMP 1.0 pour Fortran
  - Octobre 1997
- OpenMP 1.0 pour C/C++
  - Octobre 1998
- OpenMP 2.0 pour Fortran → 2000
- OpenMP 2.0 pour C/C++ → 2002
- OpenMP 2.5
  - Unification de la norme pour Fortran et C/C++
  - Sortie en Mai 2005
- OpenMP 3.0
  - Ajout du support du parallélisme de tâches
  - Sortie en Mai 2008 (remplacée par OpenMP 3.1 depuis 2011)
- OpenMP 4.0
  - Gestion des accélérateurs



# Exemple de programme OpenMP

- Exemple de code séquentiel
- Addition de 2 vecteurs
  - Fonction `vecAdd`
- Parallélisme disponible dans cette fonction ?
- OUI, sur la boucle, mais
  - Indépendance des zones d'allocation de `a`, `b` et `c`

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i;
 for (i=0 ; i<N ; i++) {
 c[i] = a[i] + b[i];
 }
}
```





# Exemple de programme OpenMP

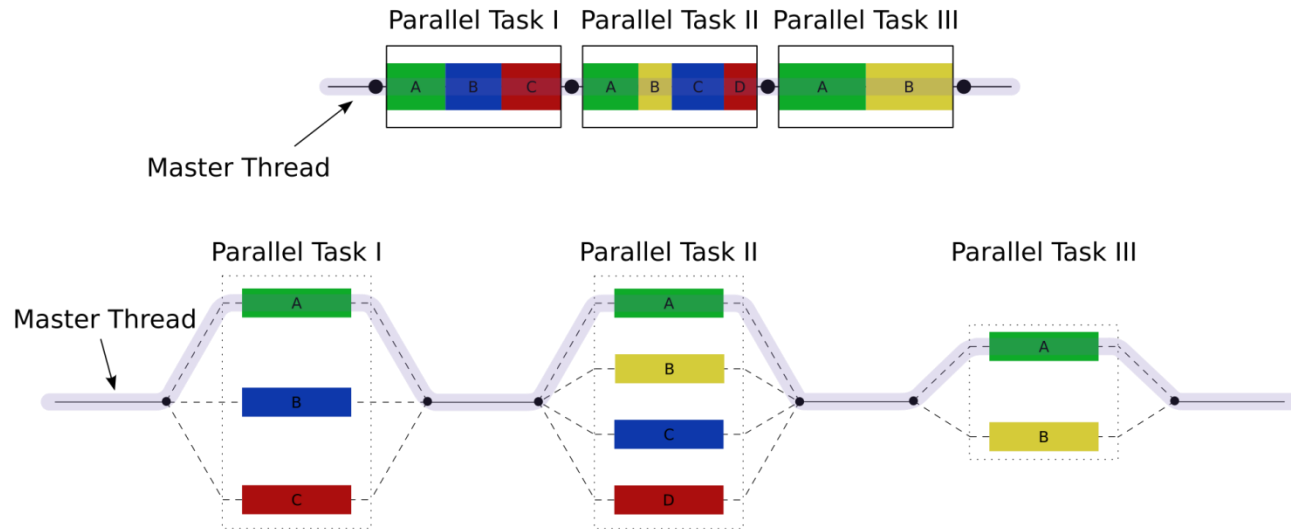
- Exploitation du parallélisme de données dans la boucle
- Ajout d'une directive pour la parallélisation
- Effets de cette directive
  - Création d'une région parallèle
  - Partage du domaine d'itérations de la boucle entre les threads participant à la région parallèle
  - Chaque thread a une copie privée de la variable  $i$
  - La fin de la région parallèle implique une barrière
- Concrètement
  - Avec  $T$  threads
  - Le thread 0 va exécuter les  $N/T$  premières itérations
  - Le thread 1 va exécuter les  $N/T$  itérations suivantes
  - ...

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 #pragma omp parallel for\
 private(i)
 for (i=0; i<N; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```



# Modèle d'exécution d'OpenMP

- Modèle d'exécution fork/join
  - Entrée dans une région parallèle → fork
  - Sortie de région parallèle → join (barrière)
  - A l'intérieur : coopération entre les threads grâce à des constructions dédiées (comme `#pragma omp for`)





# Plan du cours

---

- Modification du compilateur
  - Plugin
  - Événement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG
- Gestion des directives OpenMP dans GCC
  - Introduction à OpenMP
  - Transformation manuelle
  - Implémentation dans le compilateur GCC



# Transformation manuelle

---

- Comment transformer le code source avec directives OpenMP en code qui s'exécute en parallèle ?
- Participation du compilateur et d'une bibliothèque externe
- Dans GCC :
  - Le compilateur transforme le code et génère les appels à la bibliothèque GOMP (GCC OpenMP)
  - GOMP est en charge de la création/maintenance des threads et des synchronisations/communications entre eux



# Transformation manuelle - Etapes

---

- Etude du flot de données
  - Quelles variables sont en vie dans la région parallèle ?
- Création d'une structure intermédiaire
  - On copiera les variables nécessaire au flot de données de la région parallèle
- Extraction de la région parallèle dans une nouvelle fonction (*outlining*)
- Restauration du flot de données
  - Copie des données de la structure intermédiaire
- Appel au *runtime* pour le démarrage de la région parallèle
  - Appel à une fonction de la bibliothèque OpenMP avec un pointeur sur la nouvelle fonction extraite
- Tout ceci est fait en pratique par le compilateur GCC
- Application manuelle sur notre code d'exemple !



# Etude du flôt de données

---

- Etude du flot de données dans notre fonction `vecAdd`
- Les variables `a`, `b`, `c`, `N` et `i` sont utilisées de la région parallèle
  - La variable `i` est déclarée privée dans la région parallèle
  - Chaque thread va avoir sa propre copie
- Elles sont *en vie* avant et après la région parallèle

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i ;

 #pragma omp parallel for\
 private(i)
 for (i=0; i<N; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```



# Création d'une structure

- Création d'une structure
  - Un champ par variable en entrée de la région parallèle
  - Les variables *private* ne sont pas concernées
- Cette structure est ensuite remplie avec les bonnes données
- Dans notre exemple
  - Transfert de *a*
  - Transfert de *b*
  - Transfert de *c*
  - Transfert de *N*

```
struct _s {
 double * _a ;
 double * _b ;
 double * _c ;
 int _N ;
};
```

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i;
 struct _s s ;
 s._a = a ;
 s._b = b ;
 s._c = c ;
 s._N = N ;
 #pragma omp parallel for\ private(i)
 for (i=0; i<N; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```

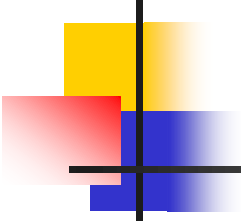


# Extraction de la région parallèle

---

- L'étape 3 consiste en l'extraction de la région parallèle en une nouvelle fonction
  - *Outlining* de fonction
  - Par opposition à l'*inlining*
- Déroulement
  - Sélection du bloc délimitant la région parallèle
  - Dans notre cas, il s'agit juste de la boucle `for`
  - Création d'une nouvelle fonction nommée `omp1` (importance de l'unicité du symbole)
  - Passage en argument d'une structure de donnée intermédiaire pour rétablir le flôt de données

# Extraction de la région parallèle



```
struct _s {
 double * _a ;
 double * _b ;
 double * _c ;
 int _N ;
} ;

void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i;
 struct _s s ;
 s._a = a ;
 s._b = b ;
 s._c = c ;
 s._N = N ;
 GOMP_start_parallel(omp1, &s);
}
```

```
void omp1(struct _s * s) {
 int i;
 double * a ;
 double * b ;
 double * c ;
 int N ;
 int min ;
 int max ;

 // Calcul des bornes
 // de l'espace d'itérations
 min = ... ;
 max = ... ;

 for (i=min; i<max; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```



# Restauration du flot de données



---

- Appel d'une fonction interne à la bibliothèque OpenMP pour démarrer une région parallèle
- En pratique, lance plusieurs threads ou réveille certains threads dormant
- Parmi les paramètres d'entrée : notre structure
- Ajout d'affectations au début de la nouvelle fonction pour mettre à jour les variables



# Restoration du flot de données

---

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i;
 struct {
 double * _a ;
 double * _b ;
 double * _c ;
 int _N ;
 } s ;
 s._a = a ;
 s._b = b ;
 s._c = c ;
 s._N = N ;
 GOMP_start_parallel(omp1, &s);
}
```

```
void omp1(struct _s * s) {
 int i;
 double * a ; double * b ;
 double * c ; int N ;
 int min ;
 int max ;

 // Calcul des bornes
 // de l'espace d'itérations
 min = ... ;
 max = ... ;

 a = s->_a ;
 b = s->_b ;
 c = s->_c ;
 N = s->_N ;

 for (i=min; i<max; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```



# Plan du cours

---

- Modification du compilateur
  - Plugin
  - Événement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG
- Gestion des directives OpenMP dans GCC
  - Introduction à OpenMP
  - Transformation manuelle
  - Implémentation dans le compilateur GCC



# Implémentation dans GCC

---

- Toutes les étapes de cette transformation sont assurées par le compilateur
- Cahier des charges
  - Gestion des pragmas (syntaxe, sémantique, ...)
  - Création de nouvelles fonctions (gestion des symboles)
  - Analyse du flot de données
  - Création de nouvelles structures
  - Ajout d'appels de fonctions
  - Support d'une bibliothèque OpenMP
- Détails dans les différentes parties de GCC



# Gestion des pragmas

---

- Enregistrement dans le préprocesseur
- Concerne les langages C et C++
  - GCC supporte également OpenMP dans les applications FORTRAN
  - Le support des directives est alors un peu différent (commentaires spéciaux dans le code source)
- Permet de laisser passer les pragmas
- Gestion réelle par le front-end



# Gestion des pragmas

---

- Fichier `gcc/c-pragma.c`
- Structure contenant les informations sur les directives

```
struct const omp_pragma_def omp_pragmas[] = {
 { "parallel", PRAGMA_OMP_PARALLEL},
 ...
}
```
- Enregistrement des directives dans le préprocesseur

```
if (flag_omp)
 for (i = 0 ; i < n_omp_pragmas ; i++)
 cpp_register_deferred_pragma(parse_in, "omp", omp_pragmas[i].name,
 omp_pragmas[i].id, true, true) ;
```
- Note : dépend des options de compilation
  - La variable `flag_omp` est automatiquement générée pendant la compilation de GCC et est mise à 1 lorsque l'option `-fopenmp` est activée pendant la compilation d'un fichier



# Front-end

---

- Transforme les directives en nœuds de la RI
  - Représentation GENERIC
- Pragmas viennent du préprocesseur
- Certaines constructions OpenMP sont directement transformées en appels de fonction
  - Exemple : `#pragma omp barrier`
  - Devient lors du front-end : `GOMP_Barrier()` ;
- Création de noeud (*tree code*) spéciaux
  - OMP\_PARALLEL pour les région parallèles



# Front-end

---

- Fichier `c-parser.c`
- Fonction `c_parser_pragma`
  - Général pour tous les pragmas du langage C
- Selection sur les directives OpenMP à transformer directement
  - Certaines directives sont traités directement (exemple : `PRAGMA_OMP_BARRIER`)
  - Dans ce cas : appel à une fonction qui termine la transformation (ex : appel à `c_parser_omp_barrier`)
- Concernant les autres directives :
  - appel à fonction `c_parser_omp_construct`
  - Selection sur le nom des directives





# Front-end – Barrière OpenMP

---

- Fichier c-omp.c
- Fonction c\_finish\_omp\_barrier()

Void

```
c_finish_omp_barrier(location_t loc) {
 tree x;
 x = built_in_decls[BUILT_IN_GOMP_BARRIER];
 x = build_call_expr_loc(loc, x, 0)
 add_stmt (x);
}
```

- Effets :
  - Création d'un appel de fonction
  - Cette fonction est *built-in* (définie dans le fichier omp-builtins.def) → GOMP\_barrier()
  - Ajout de cet appel aux *statements* courants



# Front-end – Région parallèle

---

- Fichier c-typeck.c
- Fonction c\_finish\_omp\_parallel
  - Après avoir parsé les clauses

```
void
c_finish_omp_parallel(location_t loc, tree clauses, tree block) {
 stmt = make_node(OMP_PARALLEL);
 TREE_TYPE(stmt) = void_type_node
 OMP_PARALLEL_CLAUSES(stmt) = clauses;
 OMP_PARALLEL_BODY(stmt) = block
 SET_EXPR_LOCATION(stmt, loc)
}
```

- Effets :
  - Création d'un noeud de *code* OMP\_PARALLEL
  - Le type est considéré comme *void*
  - L'emplacement dans le fichier source est lié grâce à la structure *location\_t*



# Front-end → Middle-end

---

- Une fois les fonctions parsées, la représentation intermédiaire évolue
  - Passage de GENERIC à GIMPLE
  - Nom de code : *gimplification*
- Fichier `gimplify.c`
  - Fonction `gimplify_expr()`
  - Sélection sur la valeur du *tree code*
  - Dans le cas de `OMP_PARALLEL`
    - Appel à `gimplify_omp_parallel()`
    - Création du noeud `GIMPLE_OMP_PARALLEL`



# Middle-end

---

- Bilan en entrée du middle-end
  - Les directives ont été parsées
  - Certaines ont déjà été transformées en appels de fonction à la bibliothèque GOMP
  - Les autres ont été intégrées à la représentation GIMPLE
- Le coeur principal de la gestion OpenMP se situe dans le middle-end
- Gestion en 2 transformations (fichier `omp-low.c`)
  - `omp-low`
  - `omp-exp`
- Place dans le pass manager
  - Fichier `passes.c`
  - Fonction `init_optimizations_passes()`
  - `NEXT_PASS(pass_lower_omp)`
  - `NEXT_PASS(pass_expand_omp)`



# Middle-end - Exemple

- Revenons à notre exemple
  - Addition de 2 vecteurs
  - Fonction `vecAdd`
  - Fichier `test.c.008t.omplower`

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i ;
 #pragma omp parallel for\
 private(i)
 for (i=0; i<N; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```

```
void vecAdd(/* ... */) {
 struct .omp_data_s.0 .omp_data_o.1
 ;
 int i;
 .omp_data_o.1.a = a;
 .omp_data_o.1.b = b;
 .omp_data_o.1.c = c;
 .omp_data_o.1.N = N;
 // ...
 a = .omp_data_o.1.a ;
 b = .omp_data_o.1.b ;
 c = .omp_data_o.1.c ;
 N = .omp_data_o.1.N ;
}

void
vectAdd.omp_fn.0(.omp_data_o.1
) {
 .omp_data_i = &.omp_data_o.1
 D.3455 = .omp_data_i->N ;
}
```



# Middle-end - Exemple

- Revenons à notre exemple
  - Addition de 2 vecteurs
  - Fonction `vecAdd`
  - Ficheir `test.c.022t.ompexp`

```
void vecAdd(double * a,
 double * b,
 double * c,
 int N) {
 int i ;
 #pragma omp parallel for\
 private(i)
 for (i=0; i<N; i++) {
 c[i] = a[i] + b[i] ;
 }
}
```

```
void vecAdd(/* ... */) {
 struct .omp_data_s.0
 .omp_data_o.1 ;
 int i;
 .omp_data_o.1.a = a;
 .omp_data_o.1.b = b;
 .omp_data_o.1.c = c;
 .omp_data_o.1.N = N;
 // ...
 __builtin_GOMP_parallel_start
 (vectAdd.omp_fn.0,
 &.omp_data_o.1, 0);
 vectAdd.omp_fn.0(
 &.omp_data_o.1);
 __builtin_GOMP_parallel_end()
 ;
 // ...
}
```



# Conclusion

---

- Modification du compilateur possible grâce à un plugin
  - Avantage : en dehors des sources du cœur du compilateur
  - Inconvénient : modifications limitées
- Documentation
  - Manuel *internals* et transparents disponibles
  - Documentation la plus efficace : code source
- Représentation intermédiaire
  - Attention au type de représentation utilisée (Arbre GIMPLE, CFG, les deux, ...)
  - Certaines données ne sont construites que plus tard (par exemple boucles)