Optimisation de GCC

Ce guide est une introduction à l'optimisation de code compilé en recourant à des variables *CFLAGS* et *CXXFLAGS* saines. Il présente aussi la théorie sous-jacente à l'optimisation en général.

Contents [hide]

Introduction

Que sont les variables CFLAGS et CXXFLAGS ?

CFLAGS et *CXXFLAGS* sont des variables d'environnement normalement utilisées pour dire aux compilateurs de la collection GNU, (gcc), quelles options utiliser lorsque l'on compile un programme en C ou en C++. Bien que ces variables ne soient pas standardisées, tout compilateur bien élaboré devrait comprendre ces variables pour passer des options supplémentaire lorsque le compilateur est appelé. Allez voir la page GNU make pour plus d'information à propos des variables les plus utilisées dans cette catégorie.

Parce que la majorité des paquets constituant un système Gentoo sont écrits en C ou C++, ce sont deux variables qu'un administrateur voudra généralement paramétrer correctement, car elles influencent beaucoup la manière dont le système est construit.

Elles peuvent être utilisées pour diminuer le nombre de messages de débogage pour un programme, augmenter le niveau d'alerte, et bien-sûr, optimiser le code produit. Le manuel de gcc (en anglais) tient à jour une liste exhaustive des options disponibles et de leurs objectifs.

Comment sont-elles utilisées ?

Normalement les variables *CFLAGS* et *CXXFLAGS* sont définies dans l'environnement lorsque le script de configuration ou avec les Makefiles par le programme **automake**. Dans les systèmes basés sur gentoo, il faut définir ces variables dans le fichier `/etc/portage/make.conf`. Les variables définies dans ce fichier seront exportées par portage dans l'environnement de compilation afin que toutes les compilations utilisent ces paramètres comme base.

CODE Activer CFLAGS dans /etc/portage/make.conf

```
CFLAGS="-march=athlon64 -O2 -pipe"
CXXFLAGS="${CFLAGS}"
```

Important
Bien qu'il soit possible d'avoir plusieurs ligne pour les options de la variable USE, faire de même avec *CFLAGS* conduira à des problèmes avec des programmes tels que **cmake**. Assurez-vous que la déclaration des *CFLAGS* tient sur une seule ligne avec le moins d'espaces possible pour éviter ces problèmes. Reportez-vous au bug #500034 comme exemple.

Comme vu dans l'exemple ci-dessus, la variable *CXXFLAGS* est définie pour utiliser toutes les options présentes dans *CFLAGS*. La plupart des systèmes doivent être configurés de cette manière. Les options additionnelles pour *CXXFLAGS* sont moins courantes et ne s'appliquent pas assez généralement pour qu'il soit utile de les paramétrer globalement.

Tip
L'article Safe CFLAGS peut être utile aux débutants pour optimiser leur système.

Erreurs de conception

Bien que *CFLAGS* et *CXXFLAGS* puissent être un moyen efficace de produire des binaires plus compacts et/ou plus rapides, elles peuvent aussi empêcher votre code de fonctionner, augmenter sa taille, ralentir son temps d'exécution.Leurs attribuer des valeurs incorrecte peut causer des erreurs de compilation !

Ne pas oublier que les variables *CFLAGS* globales configurées dans `/etc/portage/make.conf` s'appliqueront à tous les paquets du système, ainsi l'administrateur définit généralement seulement des options vastes et universelles. Chaque paquet modifie ensuite ces options dans ebuild ou directement dans le système de 'build' pour générer un ensemble de paramètres utilisés pendant la compilation.

Prêt ?

Ayant pris conscience des risques potentiels, on peut s'attarder sur des optimisations sûres et sans danger. Celles-ci permettent de maintenir une bonne entente avec les développeurs la prochaine fois qu'un problème sera rapporté sur Bugzilla. (En effet, les développeurs demandent généralement de recompiler un paquet avec des options *CFLAGS* minimales, pour voir si le problème subsiste. Ne pas oublier que des options agressives peuvent gâcher le code !)

Optimiser

Les bases

L'objectif derrière les options des variables *CFLAGS* et *CXXFLAGS* est de créer un code parfaitement adapté au système ; il devrait fonctionner parfaitement tout en étant aussi compact et rapide que possible. Parfois, ces conditions sont mutuellement incompatibles, c'est pourquoi ce guide se limitera à des combinaisons réputées pour bien fonctionner. Idéalement, ce sont les meilleurs possibles pour toute architecture de processeur. À titre d'information, les options agressives seront traitées plus tard. Toutes les options listées dans le manuel GCC (il y en a des centaines) ne seront pas traitées, mais seulement les plus basiques et courantes seront étudiées.

Note
Si une option est inconnue, se reporter au chapitre correspondant dans le manuel GCC. Si ce dernier n'est pas assez limpide, utiliser un moteur de recherche ou regarder la liste de diffusion de GCC.

-march

La première, et la plus importante des options est `-march`. Elle indique au compilateur quel code il devrait produire pour votre architecture de processeur (ou *arch*) ; elle indique à GCC qu'il devrait produire du code pour un certain type de processeur. Des processeurs différents ont des aptitudes différentes, prennent en charge différents jeux d'instructions et ont des manières différentes d'exécuter le code. L'option `-march` renseigne le compilateur pour qu'il produise le code spécifique au processeur, en tenant compte de toutes les aptitudes, fonctionnalités, jeux d'instructions, comportements, etc. de ce processeur, à condition que le code source soit disposé à les utiliser. Par exemple, pour bénéficier des instructions AVX, le code source doit être adapté pour les supporter.

`-march=` is an ISA selection option; it tells the compiler that it may use the instructions from the ISA. On an Intel/AMD64 platform with `-march=native -O2` or lower OPT level, the code will likely end up with AVX instructions used but using shorter SSE XMM registers. To take full advantage of AVX YMM registers, the `-ftree-vectorize`, `-O3` or `-Ofast` options should be used as well[1].

`-ftree-vectorize` is an optimization option (default at `-O3` and `-Ofast`), which attempts to vectorize loops using the selected ISA if possible. The reason it isn't enabled at `-O2` is that it doesn't always improve code, it can make code slower as well, and usually makes the code larger; it really depends on the loop etc.

Même si la variable *CHOST* dans le fichier `/etc/portage/make.conf` spécifie l'architecture générale utilisée, `-march` devrait quand même être utilisée pour que les programmes soient optimisés pour le processeur spécifique du système. Les processeur x86 et x86-64 (parmi d'autres) devrait utiliser l'option `-march`.

De quel type de processeur dispose le système ? Pour le savoir, exécutez la commande suivante :

```
user $ cat /proc/cpuinfo
```

or even install app-portage/cpuid2cpuflags and add the available CPU-specific options to the `/etc/portage/package.use/00cpuflags` file, which the tool does through e.g. the CPU_FLAGS_X86 variable:

```
user $ cpuid2cpuflags
CPU_FLAGS_X86: aes avx avx2 f16c fma3 mmx mmxext popcnt sse sse2 sse3 sse4_1 sse4_2 ssse3
```

```
root # echo "*/* $(cpuid2cpuflags)" >> /etc/portage/package.use/00cpuflags
```

Pour avoir plus de détails, y compris sur les valeurs `march` et `mtune`, deux commande peuvent être utilisées:

```
user $ gcc -c -Q -march=native --help=target
```

- The second command will show the compiler directives for building the header file, but without actually performing the steps and instead showing them on the screen (`-###`). The final output line is the command that holds all the optimization options and architecture selection:

```
user $ gcc -### -march=native /usr/include/stdlib.h
```

Maintenant, regardons l'option `-march` en action. Ceci est un exemple pour un ancien Pentium III :

FILE `/etc/portage/make.conf` Pentium III example

```
CFLAGS="-march=pentium3"
CXXFLAGS="${CFLAGS}"
```

En voici un autre pour un processeur AMD 64-bit :

FILE `/etc/portage/make.conf` AMD64 example

```
CFLAGS="-march=athlon64"
CXXFLAGS="${CFLAGS}"
```

S'il vous reste un doute quand au type de votre processeur, vous pouvez utiliser l'option `-march=native`. Lorsque cette option est utilisée, GCC tentera de détecte automatiquement le processeur et attribuer lui-même les options appropriées pour celui-ci. **Néanmoins, `-march=native` ne doit utilisée si vous voulez ou envisagez de compiler des paquets pour un autre processeur !**

Warning
N'utilisez **PAS** `-march=native` ou `-mtune=native` dans les variables *CFLAGS* et/ou *CXXFLAGS* de `make.conf` lors de compilation avec **distcc**.

Si vous compilez des paquets sur un ordinateur, mais avez l'intention les exécuter sur un autre (comme c'est parfois le cas lorsqu'on compile sur un ordinateur récent et rapide pour un ordinateur plus ancien et plus lent), alors *n'utilisez pas* l'option `-march=native`. *Native* signifie que ce code s'exécutera *seulement* sur ce type de processeur. Les applications compilées avec l'option `-march=native` sur un processeur AMD Athlon 64 **ne pourront pas** tourner sur un ancien processeur VIA C3.

Sont aussi disponibles, les options `-mtune` et `-mcpu`. Ces options sont normalement utilisées quand il n'y a pas d'option `-march` disponible ; certaines architecture de processeur peuvent demander les options `-mtune` ou même `-mcpu`. Malheureusement, le comportement de GCC n'est pas très cohérent sur la manière dont va ce comporter une option d'une architecture à une autre.

Sur les processeurs x86 et x86-64, `-march` produira un code spécifique pour ce type de processeur en utilisant tout le jeu d'instructions disponibles et l'ABI (Application Binary Interface) correcte ; il n'y aura pas de rétrocompatibilité pour des processeurs plus anciens ou différents. Si vous n'avez pas besoin d'exécuter le code sur autre chose que le système sur lequel vous faites tourner Gentoo, continuez à utiliser `-march`. Vous devriez seulement considérer l'utilisation de `-mtune` pour le cas où vous avez besoin de générer du code pour un processeur plus ancien comme les i386 et I486. `-mtune` produit un code plus générique que march ; bien qu'il adapte le code pour un certain processeur, il ne prend pas en compte l'ensemble du jeu d'instructions et de l'ABI. N'utilisez pas `-mcpu` sur des systèmes x86 ou x86-64, car cette option est maintenant déconseillée pour ces architectures.

Seuls les processeurs non x86/x-86-64 (comme Sparc, Alpha et PowerPC) peuvent nécessiter `-mtune` ou `-mcpu` plutôt que `-march`. Sur ces architectures, `-mtune` / `-mcpu` donneront parfois des résultats identiques à ceux fournis par `-march` (sur x86/x86-64)... mais avec un nom d'option différent. Là encore, le comportement de `gcc` et le nommage des options n'est pas cohérent à travers les différentes architectures, c'est pourquoi, vous devez consulter le des options de `gcc` pour déterminer laquelle utiliser pour votre système.

Note
Pour plus de suggestions sur les réglages `-march` / `-mtune` / `-mcpu`, lisez le chapitre 5 du 5 manuel d'installation de Gentoo adapté à votre architecture. Lisez aussi, la liste des options spécifiques à l'architecture du manuel de `gcc`, et les explications plus détaillées sur les différences entre `-march`, `-mcpu` et `-mtune`.

-O

Warning
Using `-O3` or `-Ofast` may cause some packages to break during the compilation.

Note
To print all packages that were built with specified *CFLAGS/CXXFLAGS* it's possible to use the following command: `grep Ofast /var/db/pkg/*/*/CFLAGS`

Vient ensuite l'option `-O`. Elle contrôle le niveau global d'optimisation. Ceci rend le temps de compilation quelque peu plus long, et peut nécessiter plus de mémoire, en particulier si vous augmentez le niveau d'optimisation.

Il y a 5 réglages de `-O` : `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Og`, and `-Ofast`. Vous ne devriez en utiliser qu'un dans `/etc/portage/make.conf`.

À l'exception de `-O0`, les réglages de `-O` activent chacun une série d'options additionnelles, c'est pourquoi vous devriez lire le chapitre sur les options d'optimisation dans le manuel de gcc, pour connaître les options qui sont activées par chacun des niveaux de `-O`, et des explications sur ce qu'elles font.

Examinons les différents niveaux d'optimisation :

- `-O0` : ce niveau (la lettre O suivi du chiffre 0) supprime complètement toute optimisation et est la valeur par défaut si un aucune option `-O` n'est précisée dans CFLAGS ou CXXFLAGS. Ceci diminue le temps de compilation et peut améliorer les informations de débogage, mais quelques applications ne fonctionneront pas correctement sans que l'optimisation ne soit activée. Cette option n'est pas recommandée sauf dans un but de débogage.
- `-O1` : C'est le niveau d'optimisation le plus basique. Le compilateur va essayer de produire un code plus rapide et plus compact sans prendre trop de temps de compilation. C'est très basique mais ça fait toujours le travail.
- `-O2` : Un échelon au-dessus de `-O1`. C'est le niveau *recommandé* d'optimisation si vous n'avez de besoin spécifique. `-O2` active quelques options de plus que `-O1`. Avec `-O2`, le compilateur va essayer d'augmenter la performance sans compromettre la taille et sans prendre trop de temps en compilation.
- `-O3` : C'est le plus haut niveau d'optimisation possible. Il active des optimisations qui sont coûteuses en terme de temps de compilation et d'usage de la mémoire. Compiler tous vos paquets avec `-O3` ne garantit pas une amélioration de la performance. En réalité, dans de nombreuses situation, cela ralentit le système à cause des binaires plus volumineux qui réclament plus de mémoire. De plus cette option est réputé casser de nombreux paquets. C'est pourquoi utiliser `-O3` n'est pas recommandé.

```
4.x.
```

- `-Os` : Cette option optimise la taille de votre code. Elle active toutes les options activée par `-O2` qui n'augmentent pas la taille du code. Elle peut être utile pour des machines qui ont un espace disque très limité et/ou ont des processeurs avec un cache de petite taille.
- `-Og` : In gcc 4.8, un nouveau niveau d'optimisation général , -Og a été introduit.Il répond au besoin d'une compilation rapide et une amélioration du débogage tout en procurant un niveau de performance en exécution raisonnable. Le ressenti en développement devrait être meilleur qu'avec le niveau d'optimisation `-O0`. Notez que `-Og` n'implique pas `-g`, il se contente de désactiver les optimisations qui pourrait interférer avec le débogage.
- `-Ofast` : nouveau dans GCC 4.7, consiste en `-O3` plus `-ffast math`, `-fno-protect-parens<c/ode>, et -fstack-arrays`. Cette option brise la conformité stricte avec les normes, et n'est pas recommandée en utilisation.

Comme mentionné précédemment, `-O2` est le niveau d'optimisation recommandé. Si des erreurs de compilation se produisent, vérifiez que vous n'utilisez pas `-O3`. Comme option de repli, essayez de définir un niveau d'optimisation plus faible dans CFLAGS et CXXFLAGS, comme `-O1` ou même `-O0 -g2 -ggdb` (pour le rapport des erreurs et la vérification de problèmes possibles) et recompilez le paquet.

-pipe

Une option commune est `-pipe`. Celle-ci n'a aucun effet sur le code produit, mais réduit le temps de compilation. Elle indique au compilateur d'utiliser des *pipelines* pendant la compilation à la place de fichiers temporaires qui requièrent plus de mémoire. Sur les systèmes avec peu de mémoire, gcc peut se retrouver tué. Dans un tel cas, n'utilisez pas cette option.

-fomit-frame-pointer

C'est une option très commune conçue pour réduire la taille du code généré. Elle est activée pour tous les niveaux de l'option `-O` (excepté `-O0`) sur les architectures pour lesquelles procéder de cette manière n'interfère pas avec le débogage (comme x86-64), mais vous pouvez avoir besoin de l'activer vous-même en l'ajoutant à vos options. Bien que le manuel de `gcc` ne précise pas toutes les architectures sur lesquelles cette option est activée par l'utilisation de l'option GNU `gcc`, vous pourrez avoir besoin de l'activer sur x86. Néanmoins, l'utilisation de cette option rendra le débogage difficile voire impossible.

En particulier, cela rend le dépannage des applications écrites en Java beaucoup plus difficile, même si Java n'est pas le seul code affecté par l'utilisation de cette option. C'est pourquoi même si l'option apporte des bénéfices, elle rend le débogage plus difficile ; les *backtraces* en particulier seront inutiles. Cependant, si vous n'envisagez pas de faire beaucoup de débogage, et n'avez pas ajouté d'autres options en rapport avec le débogage à CFLAGS comme `-ggdb`, alors vous pouvez essayer d'utiliser `-fomit-frame-pointer`.

Important
Ne combinez pas `-fomit-frame-pointer` avec l'option similaire `-momit-leaf-frame-pointer`. Utiliser cette dernière option est déconseillé car `-fomit-frame-pointer` fait déjà le travail proprement. De plus, `-momit-leaf-frame-pointer` a démontré un impact négatif sur la performance du code.

-msse, -msse2, -msse3, -mmmx et -m3dnow

Ces options activent les jeux d'instructions SSE, SSE2, SSE3, MMX et 3DNow! pour les architectures x86 and x86-64. Ils sont utiles avant tout dans le multimedia, les jeux et autres applications utilisant les calculs en virgule flottante de manière intensive, bien qu'ils incluent aussi plusieurs autres améliorations mathématiques. Ces jeux d'instructions se rencontrent dans les processeurs les plus modernes.

Important
Vérifiez que votre processeur les prend en charge en exécutant la commande `cat /proc/cpuinfo`. La sortie présentera tous les jeux d'instructions additionnels pris en charge. Notez que **pni** n'est qu'un nom différent pour SSE3.

Vous n'avez normalement pas besoin d'ajouter ces options à /etc/portage/make.conf tant que vous utilisez l'option `-march` (par exemple, `-march=nocona` implique `-msse3`). Quelques exceptions notables sont les processeurs plus récents VIA et AMD64 qui prennent en charge des instructions qui ne découlent pas de l'utilisation de `-march` (telles que SSE3). Pour de tels processeurs, vous devrez activer des options additionnelles là ou c'est approprié après avoir vérifié la sortie de `cat /proc/cpuinfo`.

Note
Vous devriez vérifier la liste des options spécifiques aux x86 et x86-64 pour voir lesquels de ces jeux d'instructions sont activés par l'option propre au type de processeur. Si un jeu d'instruction est listé alors vous n'avez pas besoin de le spécifier ; il sera activé automatiquement par l'utilisation de l'option `-march` propre au processeur.

FAQs sur l'optimisation

Mais j'obtiens de meilleures performance avec -funroll-loops -fomg-optimize !

Non, vous le *pensez* uniquement parce que quelqu'un vous a convaincu qu'utiliser plus d'options agressives est mieux. Les options agressives ne feront qu'endommager vos applications quand elles sont utilisées à l'échelle du système entier. Même le manuel de `gcc` dit qu'utiliser `-funroll-loops` et `-funroll-all-loops` rend le code plus volumineux et plus lent. Néanmoins, pour quelques obscures raisons, ces deux options, ainsi que `-ffast-math`, `-fforce-mem`, `-fforce-addr` et d'autres options similaires, continuent à être très populaires parmi ceux qui désirent avoir les droits les plus grands à la vantardise.

La vérité sur ce sujet, c'est qu'il y a des options dangereusement agressives. Jetez donc un coup d'œil aux forums Gentoo et à Bugzilla pour savoir ce que ces options font réellement : rien de bon !

Vous n'avez pas besoin d'utiliser ces options globalement dans CFLAGS ou CXXFLAGS. Cela ne fera que dégrader la performance. Elles peuvent vous faire penser que vous avez une haute performance en fonctionnant à la limite, mais elles ne font que faire grossir votre code et vous apporter des bogues marquées INVALID ou WONTFIX.

Vous n'avez pas besoin de telles options dangereuses. **Ne les utilisez pas** !. Contentez-vous de vous en tenir aux basiques : `-march`, `-O` et `-pipe`.

Que dire des niveaux -O supérieurs à 3 ?

Quelques utilisateurs se vantent même d'obtenir une meilleure performance en utilisant `-O4`, `-O9` et plus, mais en réalité, une option `-O` d'un niveau supérieur à 3 n'a aucun effet. Le compilateur peut accepter des options telles que `-O4` pour CFLAGS, mais il n'en fait rien. Il ne cherche à optimiser que jusqu'à `-O3`, rien de plus.

Vous avez besoin de preuves ? Jetez un coup d'œil au code source de `gcc` :

CODE -O source code

```
case OPT_LEVELS_3_PLUS:
      enabled = (level >= 3);
      break;

 case OPT_LEVELS_3_PLUS_AND_SIZE:
      enabled = (level >= 3 || size);
      break;
```

Comme vous pouvez le constater, aucune valeur supérieure à `-O3` n'est prise en compte.

What about compiling outside the target machine?

Some readers might wonder if compiling outside the target machine with a strictly inferior CPU or GCC sub-architecture will result in inferior optimization results (compared to a native compilation). The answer is simple: **No.** Regardless of the actual hardware on which the compilation takes place and the CHOST for which GCC was built, as long as the same arguments are used (except for `-march=native`) and the same version of GCC is used (although minor version might be different), the resulting optimizations are strictly the same.

To exemplify, if Gentoo is installed on a machine whose GCC's CHOST is *i686-pc-linux-gnu*, and a Distcc server is setup on another computer whose GCC's CHOST is *i486-linux-gnu*, then there is no need to be afraid that the results would be less optimal because of the strictly inferior sub-architecture of the remote compiler and/or hardware. The result would be as optimized as a native build, as long as the same options are passed to both compilers (and the `-march` parameter doesn't get a `native` argument). In this particular case the target architecture needs to be specified explicitly as explained in Distcc and -march=native.

The only difference in behavior between two GCC versions built targeting different sub-architectures is the implicit default argument for the `-march` parameter, which is derived from the GCC's CHOST when not explicitly provided in the command line.

Que dire des options redondantes ?

Très souvent des options CFLAGS et CXXFLAGS qui sont activées par des niveaux de `-O` sont spécifiées de manière redondante dans /etc/portage/make.conf. Quelques fois cela est fait par ignorance, mais c'est aussi fait pour éviter le filtrage d'options ou le remplacement d'options.

Le filtrage/remplacement d'options est fait dans de nombreux ebuilds de l'arbre de Portage. C'est généralement fait parce que la compilation de certains paquets échoue à certains niveaux de `-O`, ou quand le code source est trop sensible pour que des options supplémentaires soient ajoutées. L'ebuild soit filtrera quelques options de CFLAGS et CXXFLAGS, soit remplacera le niveau de `-O` par un autre.

Le Manuel du développeur de Gentoo indique quand et comment le filtrage/remplacement d'options fonctionne.

Il est possible de contrecarrer le filtrage de `-O` en listant de manière redondante les options d'un certain niveau, (tel que `-O3`) en faisant ceci :

CODE Specifying redundant CFLAGS

```
CFLAGS="-O3 -finline-functions -funswitch-loops"
```

Néanmoins, **ce n'est pas très élégant de le faire.** Les options de CFLAGS sont filtrées pour une raison ! Quand des options sont filtrées, cela signifie que ce n'est pas sûr de compiler un paquet avec de telles options. Clairement, ce n'est *pas* sûr de compiler tout votre système avec l'option `-O3` si quelques unes des options activées par ce niveau sont susceptibles de provoquer des problèmes à certains paquets. En conséquence, vous ne devriez pas essayer d'être *plus intelligent* que les développeurs qui maintiennent ces paquets. *Faites confiance* aux développeurs ! . **Le filtrage et le remplacement d'options est fait pour votre intérêt ! Si un ebuild spécifie des options alternatives, n'essayez pas de l'éviter.**

Vous continuerez probablement à rencontrer des problèmes si vous compilez un paquet avec des options inacceptables. Quand vous rapportez vos problèmes sur Bugzilla, les options que vous utilisez dans /etc/portage/make.conf seront pleinement visibles et on vous demandera de recompiler le paquet sans ces options. Évitez d'avoir à recompiler en n'utilisant pas ces options redondantes dès l'origine ! Ne supposez pas de manière automatique que vous en savez plus que les développeurs.

Que dire de LDFLAGS ?

Les développeurs de Gentoo ont déjà défini des options de base sûres de la variable LDFLAGS dans les profils de base. Vous n'avez donc pas besoin de les changer.

Puis-je utiliser des options par paquet ?

Attention !
L'utilisation d'options par paquet complique le débogage et l'assistance. Pensez à signaler dans vos rapport de bogues si vous utilisez cette fonctionnalité et quels changements vous avez faits.

Une information sur comment utiliser les variables d'environnement par paquet (y compris CFLAGS) est fournie dans le manuel de Gentoo , "Variables d'environnement par paquet" .

See also

- Configuring compile options (AMD64 Handbook)

Ressources

Les ressources suivantes vous seront utiles pour aller plus loin dans la compréhension de l'optimisation :

- La documentation en ligne sur gcc
- `man make.conf`
- Wikipedia
- Les forums de Gentoo

References

1. ↑ GNU GCC Bugzilla, AVX/AVX2 no ymm registers used in a trivial reduction. Retrieved on 2017/07/18.

This page is based on a document formerly found on our main website gentoo.org.