Rapports d'optimisation : Améliorer la performance avec les compilateurs Intel®

Par Martyn Corden, ***Ingénieur et Conseiller technique, Département des Produits destinés aux développeurs, Intel***

Même lorsque vous compilez une application pour l'optimisation, vous pouvez obtenir une amélioration de la performance augmentée en utilisant les rapports d'optimisation. Heureusement, ceci est à présent beaucoup plus facile avec les derniers compilateurs d'Intel.

Les compilateurs d'optimisation récents peuvent transformer le code de manière à largement améliorer la performance, mais les résultats peuvent dépendre de comment le code d'origine était écrit à la base et également de la quantité d'informations disponibles pour le compilateur. Le rapport d'optimisation du compilateur Intel® indique au programmeur quelles optimisations ont été effectuées et précise également pour quelle raison d'autres ne l'ont pas été. Ces informations peuvent être utilisées pour ajuster le code, permettre les optimisations d'autres compilateurs et améliorer d'avantage la performance de l'application.

Les versions antérieures de compilateurs Intel fournissaient des informations potentiellement précieuses à travers une série de différents rapports. Toutefois ces messages n'étaient pas organisés logiquement et étaient quelquefois incompréhensibles ou déconcertants, surtout avec des versions de compilateur dotées d'extension inline ou multiple qui génèrent des boucles. Certaines des informations n'étaient pas pratiques ou tout de suite utiles. La seule source de rapport pouvait être difficile à naviguer, difficile d'accès pour les autres outils, et était inadapté aux versions parallèles qui sont de plus en plus fréquemment utilisées pour réduire la durée de développement sur des multiprocesseurs modernes.

Avec la nouvelle version 15.0 du compilateur Intel® Parallel Studio XE2015, le **rapport d'optimisation a été complètement reconçu pour intégrer tous les rapports individuels en un seul et même rapport, facile d'emploi** et pour répondre aux difficultés énoncées ci-dessus. Ici, nous aborderons les nouvelles caractéristiques du rapport d'optimisation, ainsi que leurs utilisations qui nous permettront de comprendre quelles optimisations le compilateur a exécutées ou pas, et nous guider lors des ajustements supplémentaires de l'application.

Activer et contrôler le rapport

Les options de ligne de commande pour activer le rapport d'optimisation et le contrôle de haut niveau sont répertoriées dans Figure 1 pour les compilateurs Intel pour Windows*, Linux* et OS X*. Dans la plupart des cas, la version d'une option pour Linux ou OS X débute avec -q et en ce qui concerne Windows avec /Q. Les options sont identiques pour les compilateurs C/C + et Fortran.

Linux* et OS X*	Windows*	Fonctionnalité
-qopt-report[=N]	*/Qopt-report[:N]*	Cette option active le rapport ; N=1-5 spécifie un niveau croissant
-qopt-report-file=stdout | stderr | filename	*/Qopt-report-file:stdout | stderr | filename*	Cette option contrôle l'emplacement où le rapport sera écrit (par défaut, c'est
	/Qopt-report-format:vs	Le rapport est mis en forme pour activer l'affichage dans Microsoft
-qopt-report-routine= fn1[,fn2,…]	*/Qopt-report-routine: fn1[,fn2,…]*	Cela émet le rapport uniquement pour les fonctions dont le nom contient fn1
-qopt-report-filter= "filename,ln1-ln2"	*/Qopt-report-filter= "filename,ln1-ln2"*	Cela émet uniquement le rapport pour les lignes ln1 - ln2 du fichier
-qopt-report-phase=phase1[,phase2,…]	*/Qopt-report-phase:phase1[,phase2,…]*	Les informations d'optimisation sont uniquement fournies pour les phases

1a

Phase d'optimisation	Description
vec	La vectorisation automatique et explicite en utilisant les instructions SIMD
par	La parallélisation automatique par le compilateur
boucle	La mémoire, l'utilisation de la mémoire cache, et autres optimisations de boucle
openmp	Les processus légers (threads) explicites en utilisant les directives OpenMP
ipo	L'optimisation inter-procédurale, y compris inlining
pgo	L'optimisation guidée par le profil (en utilisant le commentaire d'exécution)
cg	Les optimisations pendant la génération du code natif
déchargement	Le déchargement et le transfert des données vers les coprocesseurs Intel® Xeon
tout	Les rapports sur toutes les phases d'optimisation (par défaut)

1b

Résultat du rapport

Le rapport est désactivé par défaut et peut être activé en sélectionnant l'option -le qopt-rapport. Par défaut, pour une compatibilité avec les versions parallèles, un rapport séparé correspondant à chaque fichier objet est créé avec l'extension du fichier. optrpt dans le même répertoire que le fichier objet. Le résultat du rapport peut être redirigé vers un fichier nommé différemment, ou bien vers stderr ou stdout, en utilisant l'option -qopt-report-file.

Pour déboguer les versions avec -g sur Linux ou OS X, /Zi sur Windows, certaines informations d'optimisation de boucle sont intégrées au code d'assemblée et au fichier objet. Cela rend la structure de la boucle dans le code d'assemblée plus intelligible, et les informations d'optimisation du compilateur sont alors disponibles et peuvent être utilisées par d'autres outils logiciels.

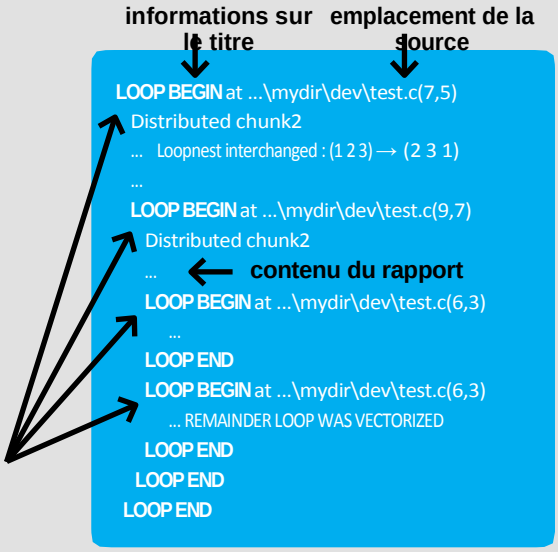
Les rapports d'optimisation peuvent parfois être très volumineux. Ils peuvent être réservés aux fonctions concernées en utilisant l'option -qopt-report-routine, ou à une gamme particulière de nombres de ligne dans un fichier source en utilisant l'option -qopt-report-filter.

Structure des rapports liés aux boucles

Les messages concernant l'optimisation de boucles intégrées sont affichés de manière hiérarchique, comme illustré dans la Figure 2. Le compilateur génère un message de "LOOP BEGIN" (début de la boucle) pour chaque boucle dans le code généré par le compilateur, de même que la ligne source initiale et le nombre de la colonne, et un message correspondant de "LOOP END" (fin de la fin). L'indentation est utilisée pour apporter plus de clarté à la structure imbriquée. Il peut y avoir de multiples boucles générées par un compilateur pour une boucle source unique et la structure imbriquée peut différer de celle du code source. Une boucle peut être "distribuée" (fractionnée) en deux ou plusieurs sous-boucles. Le rapport partiel affiché dans la Figure 2 illustre que la boucle extérieure à la ligne 6 du code source a été transformée en deux boucles intérieures dans le code optimisé qui a été généré.

```
1 double a[1000][1000],b[1000][1000],c
[1000][1000];
2
3 void foo() {
4 int i,j,k;
5
7  for( j=0; j< 1000; j++) {
8    c[j][i] = 0.0;
9    for( k=0; k<1000; k++) {
10     c[j][i] = c[j][i] + a[k][i]* b[j][k];
11    }
12   }
13  }
14 }
```

boucle imbriquée



Cet affichage hiérarchique permet aux optimisations du compilateur d'être directement associées à la boucle

concernée dans le code généré auquel elles s'appliquent.

Les instructions de chargement SIMD dans une boucle vectorisée sont les plus efficaces quand les données à charger sont alignées à une adresse de mémoire qui est un multiple de la largeur du répertoire SIMD. Pour y parvenir, le compilateur peut "retirer" certaines itérations initiales, afin que le noyau vectorisé puisse agir sur les données qui sont les mieux alignées. Le petit nombre d'itérations qui n'ont pas été traitées par le noyau vectorisé peuvent être optimisées séparément en tant boucle du "reste". Figure 3 illustre comment de telles itérations retirées et les boucles contenant le reste sont identifiées dans le rapport d'optimisation.

```
LOOP BEGIN at ggF.cc(124,5) inlined into ggF.cc(56,7)
  remark #15018: loop was not vectorized: not inner loop
  LOOP BEGIN at ggF.cc(138,5) inlined into ggF.cc(60,15)
  Retiré
      remark #25460: Loop was not optimized
  LOOP END
  LOOP BEGIN at ggF.cc(138,5) inlined into ggF.cc(60,15)
      remark #15145: vectorization support: unroll factor set to 4
      remark #15002: LOOP WAS VECTORIZED
  LOOP END
  LOOP BEGIN at ggF.cc(138,5) inlined into ggF.cc(60,15)
  Reste
       remark #15003: REMAINDER LOOP WAS VECTORIZED
  LOOP END
LOOP END
```

Vectorisé en retirant et avec le reste

3

Utiliser la boucle et les rapports de Vectorisation

L'objectif du nouveau rapport d'optimisation n'est pas uniquement de vous aider à comprendre de quelle manière le compilateur à traiter les codes source, il vous aide également à comprendre les obstacles qu'il a rencontré, vous pouvez donc l'aider à améliorer sa performance. Nous illustrerons ceci avec le simple exemple C dans la Figure 4 (le rapport et son interprétation sont très similaires pour C + + et pour Fortran). La fonction foo() forme une boucle la matrice d'entrée theta, réalise un calcul qui implique une fonction mathématique, et fournira un résultat dans la matrice sth.

```
#include <math.h>
void foo (float * theta, float * sth){
  int i;
  for (i = 0; i < 128; i++)
       sth[i] = sin(theta[i]+3.1415927);
}

$ icc -c -qopt-report=2 -qopt-report-phase=loop,vec -qopt-report-file=stderr

foo.c Begin optimization report for: foo(float *, float *)

    Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at foo.c(4,3)
<Multiversioned v1>
   remark #25228: Loop multiversioned for Data Dependence
   remark #15399: vectorization support: unroll factor set to 2
   remark #15300: LOOP WAS VECTORIZED
LOOP END

LOOP BEGIN at foo.c(4,3)
<Multiversioned v2>
   remark #15304: loop was not vectorized: non-vectorizable loop instance
   from multiversioning
LOOP END
```


Le rapport indique que le compilateur a généré deux versions de boucle qui correspondent à une boucle unique dans le code source (également appelé le "multiversioning") et explique que ceci est la conséquence de la dépendance de données. Le compilateur ne sait pas au terme de la compilation si les arguments theta et sth du pointeur sont des alias, c.-à-d., les données qu'il désigne se chevauchent de telle sorte que leur vectorisation comporte des risques. Ainsi, le compilateur crée deux versions de la boucle, une vectorisée
et l'autre pas. Le compilateur insère un test d'exécution pour la superposition des données afin que la vectorisation de la boucle soit exécutée si cela ne comporte pas de risques ; sinon, la version non-vectorisée de la boucle sera exécutée.

Si le programmeur sait que les deux arguments du pointeur ne sont pas des alias, il/elle peut fournir cette information au compilateur, soit en utilisant l'option de ligne de commande - fargument-noalias (Linux ou OS X) ou /Qalias-args- (Windows), ou le mot-clé restrict si vous utilisez -restrict (Linux ou OS X) ou/Qrestrict (Windows). Autrement, le compilateur peut être directement informé que la vectorisation de la boucle ne constitue pas de risque, en utilisant #pragma ivdep ou #pragma omp simd (ce dernier nécessite l'option -qopenmp ou l'option -qopenmp-simd). Dans chacun de ces cas, seule la version vectorisée de la boucle est générée,

et le compilateur n'a pas besoin d'effectuer des tests d'exécution pour la superposition des données. Dans notre exemple, nous utilisons l'option de la ligne de commande et augmentons le degré de détails dans le rapport comme dans la Figure 5.

```
$ icc -c -fargument-noalias  -qopt-report=4 -qopt-report-phase=loop,vec
-qopt-report-file=stderr foo.c

Begin optimization report for: foo(float *, float *)

    Report from: Loop nest & Vector optimizations [loop, vec]

LOOP BEGIN at foo.c(4,3)
   remark #15389: vectorization support: reference theta has unaligned access
[ foo.c(5,14) ]
   remark #15389: vectorization support: reference sth has unaligned access   [
foo.c(5,5) ]
   remark #15381: vectorization support: unaligned access used inside loop body
   [ foo.c(5,5) ]
   remark #15399: vectorization support: unroll factor set to 2
   remark #15417: vectorization support: number of FP up converts: single
   precision to double precision 1
   [ foo.c(5,14) ]
   remark #15418: vectorization support: number of FP down converts: double
   precision to single precision 1
   [ foo.c(5,5) ]
   remark #15300: LOOP WAS VECTORIZED
   remark #15450: unmasked unaligned unit stride loads: 1
   remark #15451: unmasked unaligned unit stride stores: 1
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 114
   remark #15477: vector loop cost: 40.750
   remark #15478: estimated potential speedup: 2.790
   remark #15479: lightweight vector operations: 9
   remark #15480: medium-overhead vector operations: 1
   remark #15481: heavy-overhead vector operations: 1
   remark #15482: vectorized math library calls: 1
   remark #15487: type converts: 2
   remark #15488: --- end vector loop cost summary ---
   remark #25015: Estimate of max trip count of loop=64
LOOP END
```

5

Le rapport indique que seule une version de boucle unique a été générée. Le résumé du coût indique que l'accélération estimée de la vectorisation est d'environ 2,79. Pas un mauvais résultat, mais nous pouvons faire mieux. Nous notons les remarques 15417 et 15418 à propos des conversions concernant la simple précision et la double précision des colonnes 14 et 5 de la ligne 5, et la présence de type 2 est converti dans le résumé. En vérifiant le code source, nous observons que la matrice theta est la simple précision, mais la constante littérale 3,1415927 est la valeur par défaut de la double précision. Le résultat de l'addition est une double précision. Ainsi, on fait appel à la version de la double précision de la fonction sinus, uniquement pour que le résultat soit converti de nouveau en simple précision pour être stocké dans sth.

Ceci a un impact sur la performance de deux manières : il est plus long de calculer plus précisément une fonction sinus ; et puisqu'une double précision prend deux fois d'espace qu'un point flottant dans le registre SIMD, les instructions du vecteur peuvent uniquement fonctionner sur la moitié moins d'éléments simultanément. Si l'on modifie le code source en transformant la constante littérale et/ou la fonction sinus explicitement en simple précision,

```
sth[i] = sinf(theta[i]+3.1415927f);
```

alors les avertissements concernant les conversions de précision disparaissent, et l'accélération estimée double pratiquement pour atteindre 5,4. Ceci est dû au fait que la majorité du temps est utilisé à l'appel de la bibliothèque mathématique vectorisée (la remarque
#15482), et que peu de temps est consacré aux opérations de vectorisation plus légères (la remarque #15479).

Ensuite, nous remarquons que l'estimation du nombre de trajets maximum de la boucle vectorisée est de 64, (la remarque
#25015), à comparer du nombre d'itérations de la boucle originale qui lui est de 256. Ainsi chaque opération de vecteur agissent sur 4 points flottants, c'est-à-dire, 16 octets. Ceci s'explique par le fait que par défaut nous compilons pour Intel® Streaming SIMD Extensions (Intel® SSE), pour lesquelles la largeur du vecteur correspond à 16 octets. Si l'on dispose d'un processeur Intel® compatible avec Intel® A Avanced Vector Instructions (Intel® AVX), qui a une largeur de vecteur de 32 octets, nous pouvons cibler ceux-ci avec l'option -xavx du compilateur. Cela entraîne les changements suivants dans le rapport :

```
remark #15477: vector loop cost: 11,620
remark #15478: estimated potential speedup: 9,440
...
remark #25015: Estimate of max trip count of loop=32
```

Si nous avions ciblé un coprocesseur Intel® Xeon Phi™, le nombre maximum de trajet aurait été de 16 et la largeur du vecteur aurait été 16 points flottants soit 64 octets.

Nous aborderons à présent les messages concernant l'alignement. Les accès à la mémoire qui sont alignés à une limite de 32 octets pour Intel AVX (16 octets pour Intel SSE, 64 octets pour les coprocesseurs de Intel Xeon Phi) sont typiquement plus efficace que les accès à la mémoire qui ne sont pas si alignés. La remarque #15381 est un avertissement général pour indiquer qu'un accès à la mémoire non-alignée a été détecté quelque part dans la boucle.

Les remarques #15389, 15450, et 15451 nous informent que lorsque le compilateur génère des chargements de theta et enregistre vers sth, il suppose que les données ne sont pas alignées. Puisque theta et sth sont perçus comme des arguments, le compilateur ne connait pas leur alignement. Les données peuvent être alignées à

l'emplacement où elles sont déclarées en utilisant declspec(align(32)) (Windows) ou attribute ((align(32))) (Linux ou OS X), ou où ils sont alloués, par exemple, en utilisant _mm_malloc () ou Posix memalign (). S'il est avéré que les arguments de la fonction foo () sont alignés, le mot-clé
_ assume_aligned() peut être utilisé pour informer le compilateur :

```
_ assume_aligned(theta,32);
_ assume_aligned(sth,32);
```

Ces mots-clés devraient uniquement être utilisés si vous êtes sûr que les arguments du pointeur de la fonction désigneront toujours des données alignées. Il n'y a pas de contrôle d'exécution. Après avoir recompilé avec le
_ mot-clé assume_aligned, seuls les accès à la mémoire alignée sont signalés, par exemple :

```
remark #15388: vectorization support: reference theta has aligned access
```

On estime que l'accélération entraînée par la vectorisation est d'environ 20% :

```
remark #15477: vector loop cost: 9,870
remark #15478: estimated potential speedup: 11,130
```

Maintenant que sth est aligné, le compilateur a la possibilité de générer des systèmes de stockage en continu (également appelés stockages non temporels) directement sur la mémoire. Ceci est particulièrement utile si l'accès à ces données stockées est peu probable dans un avenir proche, (c'est-à-dire., avant qu'elles ne soient expulsées de cache). Ceci évite une "lecture-pour-propriétaire " (RFO) de la ligne de cache, ce qui peut être utile pour les applications qui lisent et écrivent beaucoup de données et dont l'exécution est limitée par la largeur de bande disponible de la mémoire. Cela libère également de l'espace sur la mémoire cache pour des usages plus productifs. Pour le compilateur, générer automatiquement des systèmes de stockage en continu est uniquement utile pour les quantités de données beaucoup plus importante que dans cet exemple, typiquement plusieurs méga-octets. Si le nombre d'itérations augmente jusqu'à 2000000, ou si #pragma vector nontemporal est placé devant la boucle, le compilateur génère des instructions de systèmes de stockage en continu et les messages supplémentaires suivants apparaissent dans le rapport d'optimisation :

```
remark #15467: unmasked aligned streaming stores: 1
remark #15412: vectorization support: streaming store was generated for sth
```

Même pour une si petite fonction, le rapport d'optimisation peut être une source riche en informations.

Exemple de Rapport d'optimisation inter-procédural (IPO) sur Inlining

Le rapport IPO offre des informations sur les optimisations au-delà des limites de la fonction. Ici, nous nous concentrerons sur inlining.

```
 3    static void bar (float a[N], float b[N])
      {
11   }
12
13   static void foo(float a[N], float b[N]){
       …    //   small body
21    bar(a, b);
22   }
23
24   extern int main() {
26    float a[N]; float
27    b[N];
        …
35    foo(a,
36    b);
37    foo(a, b);
38    printf("result %d %d\n",b[0], b[N-1]]);
```

```
icc -qopt-report=3 -qopt-report-phase=ipo sm.c
 INLINING OPTION VALUES:
  -inline-factor: 100
 ...
 INLINE REPORT: (main) [1]  sm.c(24,19)
   -> INLINE: [35] foo()
     -> [21] bar()
  - > INLINE: [36] foo()
     -> [21] bar()
   - >EXTERN: [37] printf

 INLINE REPORT: (bar) [2] sm.c(3,42)

 DEAD STATIC FUNCTION: (foo) sm.c(13,42)
```

Figure 6 illustre schématiquement un programme principal qui fait appel deux fois une petite fonction statique foo(), et puis fait appel à printf pour imprimer un résultat final. foo() fait appel une grande fonction statique bar(). Chaque fonction
live obtient son propre rapport inlining. Ainsi, le texte principal main(), dont le corps débute à la ligne 24, la colonne 19, remplace un appel de fonction foo () à la ligne 35 et à la ligne 36. foo() à son tour, remplace un appel de fonction bar() à la ligne 21. main() fait aussi appel à printf () à la ligne 37 ; printf est marqué comme étant externe, parce que son contenu n'est pas visible pour le compilateur. bar(), dont le corps débute à la ligne 3, la colonne 42, ne contient pas d'appels de fonction. La fonction statique foo(), dont le corps débute à la ligne 13, la colonne 42, est marquée comme inactive parce que tous les appels à elle dans le fichier source remplacent un appel de fonction ; puisqu'il ne peut pas être appelé de l'extérieur, le compilateur n'a pas besoin de généré une version autonome de la fonction.

Tout appel indirect de fonction apparaitraient également au niveau 3 du rapport, et serait qualifié "INDIRECT." À des niveaux plus élevés, la taille de toutes les fonctions appelées visibles pour le compilateur sont affichées, de même que l'augmentation de la taille des fonctions d'appel quand elles remplacent un appel de fonction .

Au début de la phase de inlining du rapport d'optimisation se trouve une liste des valeurs des paramètres de l'extension inline qui ont été utilisées, à côté des options du compilateur qui peuvent être utilisées pour les modifier. Celles-ci peuvent être utilisées pour contrôler la quantité de inlining, fondé sur les informations contenues dans le rapport. Par exemple, en changeant l'argument de -inline-factor (/Qinline-Factor sur Windows) de 100 à 200, on double toutes limites de tailles qui sont utilisées pour contrôler ce qui pourrait remplacer un appel de fonction . L'inlining de fonctions individuelles peut être requis ou peut être inhibé en utilisant des directives pragma telles que inline, noinline, et forceinline, ou
par les attributs de la fonction correspondante en utilisant ou les mots-clés declspec.
attribute_ plus Pour
en détails, veuillez consulter les Guides de référence et de l'utilisateur
du compilateur Intel®.

Autres phases du rapport

-qopt-report-phase=par: Les rapports sur parallélisation automatique (processus léger) par le compilateur, structurés de la même manière et intégrés avec les rapports de vectorisation et les rapports de la boucle.

-qopt-report-phase=openmp: Les rapports sur les constructions de processus légers provenant des pragmas ou des directives d'OpenMP*.

-qopt-report-phase=pgo: Les rapports sur l'optimisation guidé par le profil, y compris les fonctions qui sont dotées de profils utiles.

-qopt-report-phase=cg: Les rapports sur les optimisations pendant la génération du code, comme l'abaissement de la fonction intrinsèque (la conversion vers des niveaux inférieures de constructions).

-qopt-report-phase=loop: Les rapports sur la boucle supplémentaire et les optimisations de mémoire, comme le blocage de cache, la récupération, l'échange de boucle, la fusion de boucle, etc.

-qopt-report-phase=offload: Les données planifiées sont résumées pour leur transfert vers et d'un coprocesseur Intel Xeon Phi.

Résumé

Le nouveau rapport d'optimisation consolidé dans les compilateurs 15.0 C/C ++ et Fortran d'Intel® fournissent une mine d'informations dans un format facilement accessible. Ceci inclut des rapports indiquant quelles optimisations n'ont pu être exécutées, ainsi que celles qui ont été exécutées. Ces rapports peuvent fournir de précieux conseils sur les ajustements supplémentaires qui pourraient améliorer l'exécution de l'application.

Pour plus d'informations, veuillez consulter Intel® Parallel Studio XE 2015 Composer Edition et le Guide de l'utilisateur du compilateur ainsi que le Guide de référence du compilateur.

Notification d'optimisation

Le niveau d'optimisation des compilateurs Intel peut varier pour les microprocesseurs autres qu'Intel pour les optimisations qui ne sont pas uniques aux microprocesseurs Intel. Ces optimisations incluent les séries d'instruction SSE2, SSE3 et SSSE3 et autres optimisations. Intel ne garantit pas la disponibilité, le caractère fonctionnel, ou l'efficacité de toute optimisation sur des microprocesseurs qui ne sont pas fabriqués par Intel. Les optimisations dépendantes d'un microprocesseur fournies dans ce produit sont destinées à être utilisées sur des microprocesseurs Intel. Certaines optimisations non-spécifiques à la micro-architecture Intel sont réservées pour les microprocesseurs Intel. Pour plus d'informations, veuillez consulter les Guides de référence et de l'utilisateur pour le produit concerné quant aux séries d'instruction mentionnées dans cette notification.

Révision de la notification #20110804