Guide d'Implémentation de CoreMark-PRO sur STM32 avec IAR Workbench

Introduction

Ce guide détaillé a pour objectif de vous accompagner pas à pas dans l'intégration et l'exécution du benchmark CoreMark-PRO sur un microcontrôleur STM32, en utilisant l'environnement de développement intégré (IDE) IAR Embedded Workbench. CoreMark-PRO est un benchmark de performance processeur complet, conçu par l'EEMBC (Embedded Microprocessor Benchmark Consortium), qui évalue les capacités des systèmes embarqués, des microcontrôleurs aux processeurs haute performance. Il inclut une variété de charges de travail (integer et floating-point) et prend en charge les architectures multicœurs, ce qui en fait un outil précieux pour évaluer les performances réelles de votre système STM32.

L'exécution de CoreMark-PRO sur une plateforme embarquée comme le STM32 nécessite une adaptation de sa couche d'abstraction matérielle (HAL) pour interagir correctement avec les périphériques spécifiques du microcontrôleur (timers, UART pour la sortie, gestion de la mémoire). Ce guide se concentrera sur les étapes clés de cette adaptation, la configuration du projet IAR, et le processus d'exécution et de collecte des résultats.

1. Prérequis

Avant de commencer, assurez-vous de disposer des éléments suivants :

1.1 Matériel

- **Carte de développement STM32** : Une carte de développement basée sur un microcontrôleur STM32 (par exemple, une carte Nucleo ou Discovery) avec un port de débogage (SWD/JTAG) accessible. Le choix du microcontrôleur STM32 (par exemple, STM32F4, STM32H7) dépendra de vos besoins en performance et en ressources.
- **Câble USB** : Pour l'alimentation et la communication avec la carte de développement (via ST-Link intégré ou un débogueur externe).

1.2 Logiciel

- **IAR Embedded Workbench for ARM (EWARM)** : Une version installée et licenciée d'IAR EWARM. Assurez-vous que votre version prend en charge le microcontrôleur STM32 spécifique que vous utilisez [1].
- **STM32CubeMX (recommandé)** : Un outil graphique de STMicroelectronics pour la configuration des microcontrôleurs STM32 et la génération de code d'initialisation. Il peut simplifier la configuration des horloges, des GPIO et d'autres périphériques, et générer des fichiers de projet compatibles IAR [2].
- **Fichiers source de CoreMark-PRO** : Le fichier `coremark-pro-main.zip` que vous avez fourni, décompressé dans un répertoire de travail.

2. Création et Configuration du Projet IAR Workbench

La première étape consiste à créer un nouveau projet dans IAR Embedded Workbench et à le configurer pour votre microcontrôleur STM32.

2.1 Création d'un Nouveau Projet

1. Ouvrez IAR Embedded Workbench.
2. Allez dans `Project` -> `Create New Project...` .
3. Sélectionnez `C` ou `C++` comme type de projet, puis choisissez `Empty project` (ou un modèle de projet STM32 si disponible et adapté à vos besoins).
4. Spécifiez un nom pour votre projet (par exemple, `CoreMark_PRO_STM32`) et un emplacement pour le sauvegarder.

2.2 Sélection du Microcontrôleur et des Options du Projet

1. Dans la fenêtre `Project` -> `Options...` (ou en cliquant droit sur le projet dans l'explorateur de projet et en sélectionnant `Options...`):
    - **General Options** -> **Target** : Sélectionnez votre microcontrôleur STM32 spécifique (par exemple, `STMicroelectronics STM32F407VG`). IAR détectera automatiquement le

cœur ARM Cortex-M approprié.

- **General Options** -> **Library Configuration** : Choisissez la bibliothèque C appropriée. Pour les systèmes embarqués, `Full` ou `Normal` avec support de la virgule flottante est généralement requis.
- **C/C++ Compiler** -> **Optimizations** : Définissez le niveau d'optimisation. Pour les benchmarks, des optimisations élevées (`High` ou `High, balanced`) sont souvent utilisées pour obtenir les meilleures performances. Notez que les règles de CoreMark-PRO autorisent les optimisations guidées par profil si elles sont appliquées à toutes les charges de travail [3].
- **Linker** -> **Config** : Assurez-vous que le fichier de script du linker (`.icf`) est correct pour votre microcontrôleur. Si vous utilisez STM32CubeMX, il générera un fichier `.icf` adapté. Vous devrez peut-être ajuster la taille du tas (heap) et de la pile (stack) ici. CoreMark-PRO peut nécessiter une grande quantité de mémoire : **au moins 300 KB de tas et 100 KB de pile** sont recommandés pour certaines charges de travail [4].
- **Debugger** : Sélectionnez votre pilote de débogage (par exemple, `ST-Link` ou `J-Link`). Configurez les options de connexion (SWD/JTAG).

2.3 Configuration de l'Horloge et des Périphériques (via STM32CubeMX)

Si vous utilisez STM32CubeMX, suivez ces étapes :

1. Ouvrez STM32CubeMX et créez un nouveau projet pour votre microcontrôleur STM32.
2. Configurez les horloges du système pour qu'elles fonctionnent à la fréquence maximale supportée par votre MCU. Une horloge système stable est cruciale pour des mesures de performance précises.
3. Activez un périphérique UART (par exemple, USART1) pour la sortie de débogage et des résultats du benchmark. Configurez-le en mode asynchrone avec un débit en bauds standard (par exemple, 115200).
4. Activez un timer (par exemple, TIM2 ou TIM3) pour générer des interruptions régulières (par exemple, toutes les 1 ms). Ce timer sera utilisé pour la fonction de temporisation de CoreMark-PRO.

5. Générez le code pour IAR Embedded Workbench (`Project Manager` -> `IDE` -> `EWARM`). Cela créera un projet IAR avec les fichiers d'initialisation (`main.c` , `stm32fxxx_hal_msp.c` , etc.) et le script du linker.

6. Copiez les fichiers générés par CubeMX dans votre répertoire de projet IAR ou importez-les dans votre projet IAR existant.

3. Intégration du Code Source CoreMark-PRO

Maintenant, nous allons ajouter les fichiers source de CoreMark-PRO à votre projet IAR.

3.1 Ajout des Fichiers Source

1. Dans IAR, cliquez droit sur votre projet dans l'explorateur de projet et sélectionnez `Add` -> `Add Files...` .

2. Naviguez vers le répertoire `coremark-pro-main` (où vous avez décompressé le zip).

3. Ajoutez les répertoires suivants à votre projet IAR en tant que groupes de fichiers (ou ajoutez les fichiers individuellement si vous préférez un contrôle plus granulaire) :

   - `mith/src/` : Contient le cœur du harnais de test MITH (`mith_main.c` , `mith_api.c` , etc.).
   - `mith/al/src/` : Contient la couche d'abstraction matérielle (`th_al.c` , `al_single.c` , `al_smp.c`).
   - `mith/inc/` : Contient les fichiers d'en-tête du MITH (`mith.h` , `th_types.h` , `th_cfg.h`). Ajoutez ce chemin aux chemins d'inclusion de votre compilateur (Options du projet -> C/C++ Compiler -> Preprocessor -> Additional include directories).
   - `benchmarks/` : Contient les noyaux de benchmark. Vous devrez ajouter les fichiers `.c` des noyaux spécifiques que vous souhaitez exécuter. Par exemple, pour le benchmark `core` , ajoutez `benchmarks/core/core_list_join.c` , `benchmarks/core/core_matrix.c` , etc.
   - `workloads/` : Contient les fichiers `main` pour chaque charge de travail. Vous devrez ajouter le fichier `.c` de la charge de travail spécifique que vous souhaitez exécuter

(par exemple, `workloads/core/bmark_lite.c` pour le benchmark `core`).

cible dans IAR) et inclure uniquement les fichiers source pertinents pour cette charge de travail. Les fichiers non pertinents doivent être exclus de la compilation pour cette cible.

3.2 Configuration des Macros du Préprocesseur

Les macros du préprocesseur sont cruciales pour adapter CoreMark-PRO à votre environnement bare-metal. Elles doivent être définies dans les options du compilateur IAR (`Project` -> `Options...` -> `C/C++ Compiler` -> `Preprocessor` -> `Defined symbols`).

Voici les macros essentielles à définir, basées sur le `README.md` de CoreMark-PRO et le guide de portage bare-metal [4] :

- `FAKE_FILEIO=0`
- `HAVE_FILEIO=0`
- `HAVE_SYS_STAT_H=0`
- `STUB_STAT=1`
- `NO_ALIGNED_ALLOC=1`
- `HAVE_PTHREAD=0`
- `USE_NATIVE_PTHREAD=0`
- `USE_SINGLE_CONTEXT=1` (pour une exécution mono-thread, ce qui est souvent le cas en bare-metal sans RTOS complexe)
- `HAVE_STRDUP=0`
- `HOST_EXAMPLE_CODE=1` (peut être utile pour utiliser les fonctions de temps du SDK si disponibles)

Pour les benchmarks à virgule flottante, vous devrez également définir :

- `USE_FP32=1` (pour les calculs en simple précision)
- `USE_FP64=1` (pour les calculs en double précision)

Ces macros doivent être définies en fonction de la charge de travail spécifique que vous compilez. Par exemple, si vous compilez `linear_alg-mid-100x100-sp`, vous devrez définir `USE_FP32=1`.

3.3 Adaptation de `th_al.c` (La Couche d'Abstraction)

Ceci est l'étape la plus critique du portage. Vous devrez modifier le fichier `mith/al/src/th_al.c` pour implémenter les fonctions de la couche d'abstraction (AL) en utilisant les API de votre STM32 (HAL/LL) et/ou un RTOS si vous en utilisez un. Voici les fonctions clés à adapter :

3.3.1 `al_main(int argc, char *argv[])`

Cette fonction est le point d'entrée de la couche d'abstraction. Elle sera appelée par la fonction `main` de votre charge de travail. C'est ici que vous initialiserez les périphériques nécessaires à l'exécution du benchmark.

Plain Text

```
void al_main(int argc, char *argv[]) {
    // Initialisation du système STM32 (horloges, GPIO, etc.)
    // Si vous utilisez CubeMX, le code d'initialisation sera dans main.c
    // Appelez ici les fonctions d'initialisation nécessaires avant de lancer
le benchmark

    // Exemple: Initialisation de l'UART pour al_printf
    // MX_USARTx_UART_Init(); // Fonction générée par CubeMX

    // Initialisation du timer pour la temporisation
    // MX_TIMx_Init(); // Fonction générée par CubeMX

    // Appeler la fonction principale du harnais de test MITH
    mith_main(argc, argv);
}
```

3.3.2 Fonctions de Temporisation (`al_signal_*`)

Ces fonctions sont utilisées par CoreMark-PRO pour mesurer le temps d'exécution. Vous devrez les implémenter en utilisant un timer matériel de votre STM32. L'objectif est de fournir une résolution de 1 ms.

Plain Text

```
// Variables globales pour le timer
volatile uint32_t ms_ticks = 0;

// Fonction d'interruption du timer (à appeler dans l'ISR de votre timer)
void HAL_SYSTICK_Callback(void) // Ou l'ISR de votre TIMx
{
    ms_ticks++;
}

// Implémentation des fonctions al_signal_*
void al_signal_init(void) {
    // Initialisation du timer (déjà fait dans al_main si via CubeMX)
    // Assurez-vous que le SysTick ou un autre timer est configuré pour 1ms
    ms_ticks = 0;
}

void al_signal_start(void) {
    ms_ticks = 0; // Réinitialiser le compteur au début du benchmark
}

long long al_signal_now(void) {
    return ms_ticks; // Retourne le nombre de millisecondes écoulées
}

void al_signal_stop(void) {
    // Pas d'action spécifique requise ici si le timer continue de
fonctionner
}

void al_signal_destroy(void) {
    // Nettoyage si nécessaire
}

// Définir CLOCKS_PER_SEC dans th_al.c ou via les options du compilateur
#define CLOCKS_PER_SEC 1000 // Si votre timer est configuré pour 1ms
```

3.3.3 Fonctions d'Impression (`al_printf`)

`al_printf` est utilisée pour afficher les résultats et les messages de débogage. Vous devrez la rediriger vers une interface de communication de votre STM32 (UART, SWO, semihosting).

Option 1 : UART (recommandé pour la simplicité)

Plain Text

```
#include <stdio.h>
#include "stm32f4xx_hal.h" // Ou l'en-tête HAL de votre MCU

extern UART_HandleTypeDef huartx; // Déclarez votre handle UART (généré par
CubeMX)

int _write(int file, char *ptr, int len) {
    HAL_UART_Transmit(&huartx, (uint8_t*)ptr, len, HAL_MAX_DELAY);
    return len;
}

void al_printf(const char *fmt, ...) {
    char buffer[256]; // Taille du buffer à ajuster selon vos besoins
    va_list args;
    va_start(args, fmt);
    vsnprintf(buffer, sizeof(buffer), fmt, args);
    va_end(args);
    _write(0, buffer, strlen(buffer));
}
```

Assurez-vous que `_write` est correctement liée et que votre UART est initialisée. Vous devrez peut-être désactiver la redirection `printf` par défaut d'IAR si elle entre en conflit.

Option 2 : Semihosting (si supporté par votre débogueur et votre configuration IAR)

Le semihosting permet de rediriger les appels `printf` vers la console du débogueur sur votre PC. Activez le semihosting dans les options du débogueur IAR (`Project` -> `Options...` -> `Debugger` -> `Semihosting`). Ensuite, vous pouvez simplement mapper `al_printf` à `printf` .

Plain Text

```
#include <stdio.h>

void al_printf(const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    vprintf(fmt, args);
    va_end(args);
}
```

3.3.4 Fonctions de Gestion de la Mémoire (`al_malloc` , `al_free`)

Ces fonctions gèrent l'allocation dynamique de mémoire. Vous pouvez utiliser les fonctions standard de la bibliothèque C.

Plain Text

```
#include <stdlib.h>

void *al_malloc(size_t size) {
    return malloc(size);
}

void al_free(void *ptr) {
    free(ptr);
}
```

Assurez-vous que la taille du tas (heap) est suffisante dans le script du linker de votre projet IAR.

3.3.5 Gestion des Threads (`al_thread_*`)

Si vous exécutez CoreMark-PRO en mode mono-thread (`USE_SINGLE_CONTEXT=1`), ces fonctions peuvent être des stubs ou utiliser l'implémentation de `al_single.c` . Si vous utilisez un RTOS (comme FreeRTOS) et que vous souhaitez exécuter en mode multi-thread, vous devrez mapper ces fonctions aux API de votre RTOS.

Exemple (mono-thread ou stubs) :

Plain Text

```
// Définitions de stubs pour le mode mono-thread
typedef void *th_thread_t;
typedef void *th_mutex_t;

void al_thread_create(th_thread_t *thread, void *(*start_routine)(void *),
void *arg) {
    // En mode mono-thread, le thread est exécuté directement ou cette
fonction est un stub
    // Pour CoreMark-PRO avec USE_SINGLE_CONTEXT, cette fonction n'est pas
utilisée pour la création de threads réels.
}

void al_thread_join(th_thread_t thread, void **retval) {
    // Stub
```

```
}

void al_thread_mutex_init(th_mutex_t *mutex) {
    // Stub
}

void al_thread_mutex_lock(th_mutex_t mutex) {
    // Stub
}

void al_thread_mutex_unlock(th_mutex_t mutex) {
    // Stub
}

void al_thread_mutex_destroy(th_mutex_t mutex) {
    // Stub
}
```

3.4 Gestion de `argc` et `argv`

Les charges de travail de CoreMark-PRO s'attendent à recevoir des arguments via `argc` et `argv`. Dans un environnement embarqué, cela peut être problématique car le `main` n'est pas toujours appelé avec ces arguments. Voici les stratégies possibles :

1. **Semihosting** : Si votre débogueur et IAR supportent le semihosting, vous pouvez passer les arguments via les options du débogueur IAR. C'est la méthode la plus simple.
2. **Modification du `main` de la charge de travail** : Si le semihosting n'est pas une option, vous devrez modifier le fichier `main.c` de chaque charge de travail (c'est l'exception où la modification du code de la charge de travail est autorisée [4]). Vous pouvez définir `argc` et `argv` manuellement avant d'appeler `mith_main`.

4. Compilation, Débogage et Exécution

Une fois que vous avez configuré votre projet IAR et adapté `th_al.c`, vous êtes prêt à compiler et exécuter le benchmark.

4.1 Compilation

1. Assurez-vous que la cible (target) correcte est sélectionnée dans IAR pour la charge de travail que vous souhaitez compiler.
2. Allez dans `Project` -> `Make` ou cliquez sur l'icône `Make` dans la barre d'outils. IAR compilera votre projet et générera le fichier `.out` (ou `.elf`).

4.2 Débogage et Téléchargement

1. Connectez votre carte STM32 à votre PC via le débogueur (ST-Link, J-Link).
2. Dans IAR, allez dans `Project` -> `Download and Debug` ou cliquez sur l'icône `Download and Debug`. IAR téléchargera le firmware sur votre STM32 et démarrera la session de débogage.

4.3 Exécution et Collecte des Résultats

1. Une fois la session de débogage démarrée, exécutez le code (`Debug` -> `Go` ou F5).
2. Ouvrez la fenêtre `Terminal I/O` dans IAR (`View` -> `Terminal I/O`). Vous devriez voir les messages de `al_printf` s'afficher ici, y compris les résultats du benchmark (itérations par seconde).
3. **Mode Vérification (`-v0`)** : Exécutez chaque charge de travail avec l'argument `-v0` pour vous assurer qu'il n'y a pas d'erreurs de validation. Si des erreurs apparaissent, cela indique un problème dans votre portage ou dans la configuration des macros.
4. **Mode Performance (`-v1`)** : Exécutez chaque charge de travail avec l'argument `-v1` . Notez le score

d'itérations par seconde pour chaque charge de travail. Selon les règles de CoreMark-PRO, chaque charge de travail doit s'exécuter pendant au moins 10 secondes. Si ce n'est pas le cas, ajustez le nombre d'itérations (`-i<nombre_itérations>`) jusqu'à ce que cette condition soit remplie.

4.4 Collecte des Résultats

Pour obtenir le score final CoreMark-PRO, vous devrez exécuter chacune des neuf charges de travail individuellement et collecter leur score d'itérations par seconde. Notez ces valeurs

avec précision.

5. Calcul du Score Global CoreMark-PRO

Le score final CoreMark-PRO est une moyenne géométrique pondérée des scores de chaque charge de travail, normalisés par rapport à des valeurs de référence. Si vous n'utilisez pas le script PERL fourni avec la version hôte de CoreMark-PRO, vous devrez calculer ce score manuellement.

5.1 Scores de Référence et Facteurs d'Échelle

Le tableau suivant, tiré de la documentation de CoreMark-PRO [4], fournit les facteurs d'échelle (xN) et les scores de référence (rN) pour chaque composant (charge de travail) :

Composant	Facteur d'échelle (xN)	Score de référence (rN)
`cjpeg-rose7-preset`	1	40.3438
`core`	10000	2855
`linear_alg-mid-100x100-sp`	1	38.5624
`loops-all-mid-10k-sp`	1	0.87959
`nnet_test`	1	1.45853
`parser-125k`	1	4.81116
`radix2-big-64k`	1	99.6587
`sha-test`	1	48.5201
`zip-test`	1	21.3618

5.2 Formule de Calcul

Le score global CoreMark-PRO est calculé à l'aide de la formule suivante :

`Score global = 1000 × (produit de (sN / rN) * xN pour chaque charge de travail N)^(1/nombre de charges de travail)`

Où :

- `sN` est le score d'itérations par seconde obtenu pour la charge de travail N sur votre plateforme STM32.
- `rN` est le score de référence pour la charge de travail N (tiré du tableau ci-dessus).
- `xN` est le facteur d'échelle pour la charge de travail N (tiré du tableau ci-dessus).
- Le produit est calculé sur les neuf charges de travail.
- `1/nombre de charges de travail` est l'exposant pour calculer la moyenne géométrique (soit 1/9).
- Le facteur 1000 est appliqué pour obtenir un nombre entier plus gérable.

Exemple de calcul (pour une seule charge de travail, à étendre aux 9) :

Supposons que vous ayez obtenu un score `s_core` de 3500 itérations/seconde pour la charge de travail `core`. En utilisant les valeurs du tableau :

`s_core / r_core * x_core = 3500 / 2855 * 10000 = 12259.19`

Répétez ce calcul pour les neuf charges de travail, multipliez tous les résultats entre eux, prenez la racine neuvième de ce produit, puis multipliez par 1000 pour obtenir le score final.

6. Optimisation

Une fois que vous avez un score de base, vous pouvez expérimenter avec différentes options pour optimiser les performances de votre STM32 et améliorer votre score CoreMark-PRO :

- **Options du Compilateur IAR** : Ajustez les niveaux d'optimisation (`-Oh` , `-Os` , etc.), activez les optimisations spécifiques au processeur (par exemple, FPU si votre STM32 en a une), et explorez les options d'optimisation guidée par profil (PGO) si votre chaîne d'outils le permet.
- **Configuration du Linker** : Optimisez la carte de liaison pour un placement optimal du code et des données en mémoire. Assurez-vous que le tas et la pile sont dimensionnés de manière appropriée pour chaque charge de travail.

- **Configuration du Microcontrôleur** : Maximisez la fréquence d'horloge du CPU, optimisez l'accès à la mémoire Flash (par exemple, en utilisant le cache d'instructions/données si disponible), et désactivez les périphériques inutilisés pour réduire la consommation d'énergie et les interférences.
- **Implémentation de `th_al.c`** : Assurez-vous que vos implémentations des fonctions de temporisation, d'impression et de gestion de la mémoire sont aussi efficaces que possible. Par exemple, une fonction `al_printf` trop lente peut fausser les résultats.

Conclusion

L'exécution de CoreMark-PRO sur un STM32 avec IAR Workbench est un processus qui implique une compréhension approfondie de l'architecture du benchmark et une adaptation minutieuse à l'environnement embarqué. En suivant ce guide, vous devriez être en mesure de configurer votre projet, de porter la couche d'abstraction matérielle, d'exécuter les benchmarks et de calculer le score final. Ce score vous fournira une mesure standardisée des performances de votre microcontrôleur STM32, utile pour la comparaison et l'optimisation de vos applications embarquées.

Références

[1] IAR Systems. *IAR Embedded Workbench for Arm*. Disponible sur : https://www.iar.com/ewarm
[2] STMicroelectronics. *STM32CubeMX*. Disponible sur : https://www.st.com/en/development-tools/stm32cubemx.html
[3] EEMBC. *CoreMark-PRO README.md*. Disponible sur : https://github.com/eembc/coremark-pro/blob/main/README.md
[4] EEMBC. *Application Note CMP01: Porting CoreMark-PRO to Bare-Metal*. Disponible sur : https://www.eembc.org/techlit/apnotes/cmp01.pdf