Master Electronique Energie Electrique Automatique

Parcours : ROB

Année 2023-2024

De l'Université de Montpellier

Rapport sur le projet n°13

Acquisition d'un flux d'images RGBD depuis Microsoft Kinect 2 et détection des informations 3D des points clé du squelette humain

Par : Curtis MARTELET

Tuteur : Wanchen LI

Table des matières

I. Introduction

Ce projet vise à l'acquisition d'un flux d'images RGBD depuis une caméra Kinect2, puis à détecter des points clé du squelette humain.

Dans le cadre du développement de l'industrie 4.0, ce projet vise à améliorer la collaboration homme-robot. En intégrant un système de vision 3D dans la plateforme robotique pilotée par ROS2, les positions des personnes seront utilisées pour prédire leurs mouvements et actions dans un environnement industriel. Cette prédiction permettra de prévenir les accidents et d'assister plus efficacement les travailleurs. La.

Le flux d'images sera capturé à l'aide d'une caméra Kinect 2.
Le choix de cette caméra s'explique par sa capacité à enregistrer des images couleurs (RGB) ainsi que profondeur (D), son faible coût, sa facilité de mise en place (pas besoin d'installer des marqueurs sur les personnes), et sa relative disponibilité.

La reconnaissance des actions humaines se fera via un outil d'extraction de squelette. Ces données seront ensuite analysées pour déterminer les caractéristiques des mouvements humains.
Le programme que l'on utilisera sera Openpose.
Encore aujourd'hui, il s'agit de l'outil en temps réel d'extraction de squelette et d'estimation des points d'articulations le plus performant disponible gratuitement.

La communication entre les différents acteurs de ce projet devra être réalisée avec ROS2.
Sa modularité et sa capacité à gérer des systèmes complexes en font un choix idéal pour notre application.

Dans ce rapport, nous vous expliquerons et détaillerons les différentes étapes et outils nécessaire pour reproduire ce projet, ainsi que nos solutions aux problèmes rencontrés et nos résultats.

II. Présentation des axes du PFE.

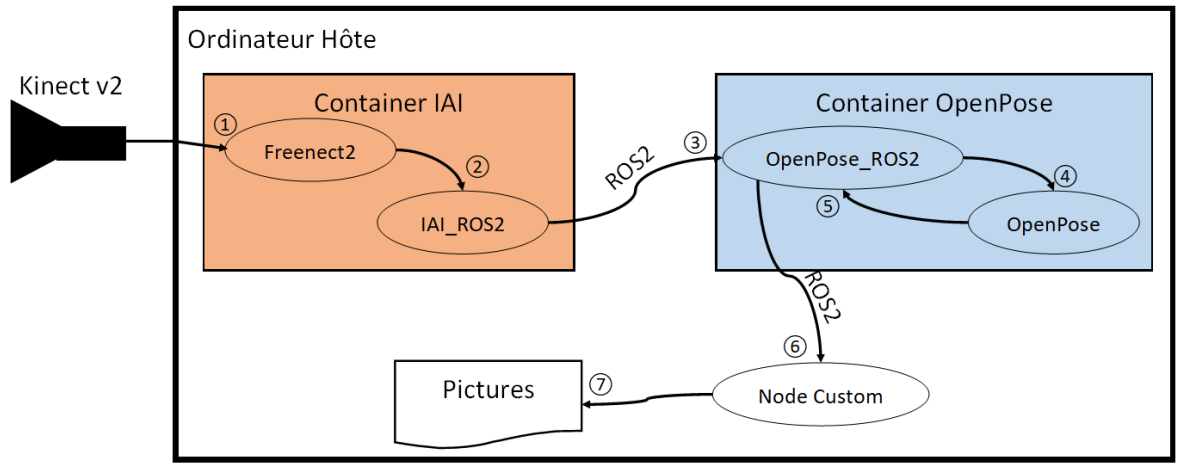

Figure 1 : Etapes de l'acquisition et traitements des images jusqu'à l'enregistrement

Pour structurer et améliorer l'efficacité du travail, ce PFE a été divisé en deux parties : l'acquisition et le traitement des images.

Cependant, cette structure a été modifiée en cours de route pour résoudre des problèmes de compatibilité entre les différents logiciels. Une troisième partie, simulation, a alors été ajoutée.

1. L'acquisition.

L'acquisition est la première étape du processus de détection et d'extraction des positions des articulations.

Nous avons utilisé le travail du premier semestre, qui consistait à créer une node « d'enregistrement » au paquet IAI_Kinect2.
IAI_Kinect2 est un paquet développé sous ROS1 utilisé pour calibrer et visualiser les données récupérées par une caméra Kinect v2. La node d'enregistrement est capable d'enregistrer les images couleurs et profondeurs calibrés à une fréquence, durée et dans un répertoire prédéfini.

Notre tâche consistait à corriger les quelques bugs encore présents et puis de l'adapter dans l'optique de migrer cette node vers IAI_ROS2, une mise à jour de IAI_Kinect2 pour **ROS2 Humble**.

2. Le Traitement.

La deuxième partie se concentre sur le traitement des images capturées par la Kinect.

Une fois les images et les données de profondeur collectées, elles doivent être fournies à Openpose pour extraire les données nécessaires.

Openpose est une bibliothèque de vision par ordinateur développée par le Carnegie Mellon Perceptual Computing Lab, permettant la détection en temps réel des poses humaines à partir d'images ou de vidéos. Elle détecte non seulement les articulations du corps, mais également les visages et les mains. C'est un outil construit pour être utilisé aussi bien sur un flux vidéo qu'un flux d'image en temps réel.

Openpose n'a pas été développé pour ROS2, nous utilisons un wrapper nommé Openpose_ROS2. Ce wrapper permet de recevoir des images depuis des topics ROS2, de les traiter avec Openpose et de publier les résultats sous forme de messages ROS2.

3. La Simulation.

Comme détaillé dans la partie suivante, nous avons dû ajouter une troisième partie, prioritaire : la simulation.
En effet, IAI_ROS2 fonctionne avec **ROS2 Humble** tandis que le wrapper Openpose_ROS2 nécessite **ROS2 Dashing**. **ROS2 Humble** est compatible avec Ubuntu 22, alors que **ROS2 Dashing** ne fonctionne qu'avec Ubuntu 18.

Nous avons donc dû employer une machine virtuelle pour simuler l'environnement Ubuntu 18 sur une session Ubuntu 22.

III. Problèmes et travail réalisé.

Dans ce chapitre, nous allons expliquer étape par étape notre cheminement dans chaque partie du projet.
Nous y exposerons nos difficultés rencontrées, nos solutions et résultats au fur et à mesure que nous progressions.

1. L'Acquisition

Comme mentionné dans la présentation du PFE, cette première partie se concentre sur la correction de bugs et l'adaptation de la node d'enregistrement pour IAI_ROS2.

Contrairement au premier projet, cette node ne doit pas seulement enregistrer les images reçues, mais aussi les publier vers Openpose : il s'agit donc de "contrôler" le flux d'images.

Si l'on se reporte à la Figure 1, il s'agit de l'étape 1 et 2.

a. Correction de bugs

Initialement, nous avons dû résoudre un bug majeur : la saturation de la mémoire. Plus qu'un bug, le problème relevait d'une erreur de conception.

Lors du projet de premier semestre, nous avions considéré que les séquences d'enregistrements ne dureraient pas plus d'une minute, ou que nous ne ferions que publier les images vers Openpose. Nous avions donc opté pour stocker les images dans la mémoire vive de l'ordinateur jusqu'à la fin de l'enregistrement.
Ce défaut de conception provoquait une accumulation excessive des images en mémoire, provoquant rapidement une saturation et des plantages système.

Pour résoudre ce problème, nous avons décidé d'enregistrer les images au fur et à mesure de leur réception. Cette méthode a nécessité l'ajout d'un mécanisme de pile et de threads car enregistrer une image sur un disque peut prendre beaucoup de temps.

Pour rappel :

- Une pile est une structure de données qui suit le principe LIFO (Last In, First Out), où le dernier élément ajouté est le premier à être retiré.
- Les threads sont des unités d'exécution indépendantes dans un programme, permettant d'effectuer plusieurs opérations en parallèle.

Voir le code 1 dans l'annexe pour comment nous avons déclaré un mutex, une pile et les threads.

L'utilisation de threads avec une pile soulève un nouveau problème : comment garantir la synchronicité, c'est-à-dire que les informations soient traitées dans le bon ordre.

Pour gérer correctement les accès concurrents à la pile par plusieurs threads, c'est-à-dire la synchronicité, nous avons utilisé des méthodes de synchronisation avec des mutex.
Un mutex (abréviation de "mutual exclusion") est un verrou empêchant plusieurs threads d'accéder simultanément à une ressource partagée, comme une pile dans notre cas. Il coordonne l'exécution des threads pour éviter les conflits et s'assure que certaines opérations se déroulent dans un ordre précis.
En verrouillant l'accès à la pile lorsqu'un thread ajoute ou retire une donnée, le mutex garantit que les informations sont traitées sans l'interférence d'autres threads.

Dans notre programme, le mutex empêche plusieurs threads d'accéder à la pile en même temps. Sans cette précaution, plusieurs threads risqueraient d'accéder à la même paire d'images, ce qui entraînerait des erreurs lorsque le premier thread supprimera les images de la pile, laissant les autres threads avec des données incomplètes ou en doubles.

Voir le code 2 dans l'annexe pour regarder comment nous avons implémenté cette solution.

Une fois le mutex implémenté, nous avons voulu évaluer l'impact de verrouiller la pile lors de l'ajout d'une image à cette dernière *(voir code 3 dans l'annexe)*.

Après plusieurs tests, nous avons conclu qu'il n'y avait aucune différence notable en termes de performance. Ainsi, nous avons décidé de continuer à verrouiller la pile pour renforcer la sécurité.

Pour optimiser les performances de notre node, nous avons expérimenté avec le nombre de threads nécessaires. Notre programme peut fonctionner avec un nombre de threads compris entre 1 et 8 (limite fixée par l'ordinateur).
Nous avons observé qu'utiliser plus de 3 threads causait des erreurs lors de la sauvegarde des images. Avoir 3 threads était excessif car l'enregistrement n'était pas assez lent pour nécessiter autant de puissance de calcul.

Finalement, nous avons opté pour n'utiliser qu'un seul thread.
Bien que la fréquence de réception des images soit supérieure à la fréquence d'enregistrement, la différence n'est pas suffisante pour provoquer une surcharge de la mémoire.
En n'utilisant qu'un seul thread, nous réduisons également la charge sur l'ordinateur. Comme nous devons exécuter au moins deux programmes simultanément dans deux containers, la gestion des ressources est un facteur à prendre en compte.

Voir le code 4 pour comment nous créons les threads, et le code 5 pour voir l'intégralité du code exécuté par chaque thread.

Avec ce code remanié, notre node a réalisé une séquence d'enregistrement de plus de 20 minutes sans problème, et sans aucun problème de sauvegarde des images.

b. Migration vers ROS2.

Dans un second temps, il a fallu migrer ce programme de **ROS1 Melodic** vers **ROS2 Dashing**.

Cette étape n'a pas été particulièrement difficile car les principales différences entre ROS1 et ROS2 résident dans l'appel des fonctions et l'API de ROS.

Cependant, un problème majeur est survenu lors de la construction de IAI_ROS2 (voir le *log 1* dans l'annexe).

Après plusieurs jours de recherche, nous n'avons que des hypothèses sur la source de cette erreur, la plus probable étant un problème lié à CUDA.
Pour rappel, CUDA (Compute Unified Device Architecture) est une plateforme de calcul parallèle et une API développée par Nvidia qui permet d'utiliser le GPU pour des calculs généraux.

Nous avons réalisé que l'installation de CUDA s'était mal passée lorsque nous avons tenté d'utiliser libfreenect2, le pilote nécessaire pour récupérer les informations de la caméra Kinect2. Ce dernier n'utilisait que le CPU alors que nous comptions sur CUDA pour améliorer les performances. Après avoir correctement réinstallé CUDA, nous avons essayé de reconstruire libfreenect2 pour qu'il prenne en compte CUDA. Cependant, la compilation a échoué en raison de l'absence d'une bibliothèque.
Cette bibliothèque fait partie du répertoire **cuda_samples**, qui était installé par défaut avec CUDA Toolkit dans les versions antérieures à la 11.2. Comme nous utilisons Ubuntu 22.04, la version la plus ancienne de CUDA Toolkit disponible est la 11.5, ce qui nous oblige à installer manuellement ce répertoire.

Nous n'avons pas réussi à l'installer car les fichiers *libnvscibuf.so*, *libnvscisync.so*, *nvscibuf.h* et *nvscisync.h* nécessaires à la construction étaient absents de notre Ubuntu.
Les forums de Nvidia n'ont pas été d'une grande aide, l'erreur ayant été signalée en 2019 sans solution, et ajouter le flag -k (keep going) au make n'a rien résolu.

En dernier recours, nous nous sommes tournés vers les images CUDA fournies par Nvidia. Notre raisonnement était que, puisque ces images sont construites par Nvidia, elles ne devraient pas poser de problèmes. Nous avons choisi la version devel de l'image pour ce test.

Sur ces images, **cuda_samples** n'est pas installés, nous avons donc répété les actions précédentes et cela a fonctionné : le répertoire de bibliothèques a été installé.
Cependant, nous n'avons pas trouvé où ces bibliothèques étaient situées, et les variables d'environnement définies pour ce répertoire lors de l'installation n'existaient pas.
Nous avons essayé de créer puis de déplacer les bibliothèques à cet emplacement, sans succès.

2. Le Traitement

Cette seconde partie sera consacrée à Openpose et à son wrapper ROS2, Openpose_ROS2. Utiliser Openpose_ROS2 nécessite d'avoir installé au préalable Openpose.

Openpose et Openpose_ROS2 nous obligent tous deux à utiliser Ubuntu 18. Pour Openpose, nous sommes limités par CuDNN. Mme Li a testé différentes versions de cette bibliothèque pour déterminer la plus récente compatible : nous pouvons utiliser CuDNN 7.6.5 avec CUDA Toolkit 10.2.
Quant à Openpose_ROS2, c'est la version de ROS2 qui nous contraint : comme mentionné dans l'introduction, le wrapper nécessite **ROS2 Dashing**, une version de ROS2 qui fonctionne uniquement sur Ubuntu 18.

Si l'on se reporte à la Figure 1, ce chapitre concerne les étapes 3 à 7 du processus en temps réel.

Dans un premier temps, nous avons vérifié le fonctionnement d'Openpose et d'Openpose_ROS2 sur une session Ubuntu 18.

a. Openpose

Openpose est une bibliothèque puissante et flexible permettant de détecter et de représenter les poses humaines à partir d'images et de vidéos.

Elle utilise des réseaux neuronaux convolutifs (CNN) pour extraire des informations sur les articulations et les squelettes humains, permettant ainsi de suivre les mouvements et les postures. L'installation d'Openpose nécessite des prérequis spécifiques en termes de logiciels et de matériel, notamment CUDA et CuDNN pour l'accélération matérielle sur GPU.

b. Openpose_ROS2

Dans le contexte de ROS2, un wrapper agit comme une interface entre le Framework ROS2 et un programme. Il facilite la communication entre les deux systèmes en masquant les détails d'implémentation spécifiques du programme et en fournissant des méthodes et des classes faciles à utiliser pour les développeurs ROS2.

Une fois Openpose installé et fonctionnel, il suffit de télécharger le wrapper, de modifier un fichier param.yaml qui contient ses paramètres et de compiler le wrapper pour pouvoir utiliser Openpose avec ROS2. Ce fichier contient 4 paramètres :

- is_debug_mode : Ce paramètre active ou désactive le mode débogage. Lorsque le mode débogage est activé, le wrapper publiera également l'image avec les squelettes superposés.
- openpose_root : Ce paramètre spécifie le chemin vers le répertoire racine d'Openpose.
- is_image_compressed : Ce paramètre indique si les images ont été compressées avant d'être envoyées à Openpose.
- image_node : Ce paramètre spécifie le nom du topic ROS2 à partir duquel les images sont reçues.

Un exemple du fichier param.yaml est donné dans l'annexe sous l'appellation de code 6.

Ce wrapper retourne les datas sous trois topics :

- /openpose/pose_key_points, du type *`openpose_ros2_msgs/msg/PoseKeyPointsList`*. Ce topic retourne la position x, y et le "score" de précision de chaque joint.
- /openpose/preview, du type *`sensor_msgs/msg/Image`* et qui est disponible en mode debug.
- /openpose/preview/compressed du type *`sensor_msgs/msg/CompressedImage`* et qui est disponible en mode debug.

c. Test de Openpose_ROS2

Pour tester ce wrapper, nous avons programmé une node ROS2.

Cette node publie des messages d'images au format `sensor_msgs/msg/Image` en parcourant un dossier spécifié pour lire les images et les publie une par une sur un topic ROS2 appelé "image_out". Le processus de publication est initié lors de la création de l'instance de la node, et chaque image est convertie en un message ROS grâce à la bibliothèque cv_bridge avant d'être publiée. Par la suite, une image est publiée à chaque fois qu'un message est reçu par la node.

La node Test_Openpose s'abonne à deux types de messages :

1. **Images traitées (`sensor_msgs/msg/Image`) :**
   - Les images traitées sont reçues via un topic auquel la node est abonnée.
   - Ces images sont converties de messages ROS en objets `cv::Mat` de OpenCV.
   - Chaque image est sauvegardée dans un dossier de destination spécifié avec un nom unique.
2. **Coordonnées des points clés (`openpose_ros2_msgs/msg/PoseKeyPointsList`) :**
   - Les données de points clés générées par Openpose sont reçues via un autre topic.

- Chaque message contient une liste de poses, et chaque pose contient des points clés avec leurs coordonnées (x, y) et un score de confiance (score).
- Les données reçues sont sauvegardées dans un fichier texte, où chaque point clé est listé avec ses coordonnées et son score.

Le message *PoseKeyPointsList* est un message custom : il a été créé spécifiquement pour Openpose_ROS2.
Pour pouvoir l'utiliser dans notre node, le CMakefile doit inclure le package openpose_ros2_msgs.

Nous n'avons pas eu plus de temps à accorder à Openpose_ROS2 : réaliser les Dockerfiles est très vite devenu la priorité.

Nous aurions voulu savoir, par exemple, l'impact sur les performances qu'a ce wrapper puisqu'il est écrit exclusivement en Python. Dans le cas de notre projet, Python est un non catégorique car il est souvent trop lent comparé aux langages compilés comme C++.
Enfin, nous aurions aussi voulu savoir s'il fallait envoyer à Openpose les images les unes après les autres, ou bien faire des paquets d'images pour les envoyer ensemble.

3. La Simulation

En raison des problèmes de versions causé par ROS2 Humble et Dashing, nous avons été contraints de recourir à l'utilisation de machines virtuelles.

Parmi les diverses plateformes de gestion de machines virtuelles disponibles sur Ubuntu, nous avons opté pour la plus populaire : Docker.

a. Docker.

Docker est une plateforme de virtualisation légère qui utilise des conteneurs pour encapsuler des applications et leurs dépendances.

Contrairement aux machines virtuelles traditionnelles, les conteneurs Docker partagent le même noyau du système d'exploitation hôte, ce qui les rend plus légers et plus rapides à démarrer.

Les containers incluent tout ce dont une application a besoin pour s'exécuter, y compris le code, les bibliothèques, les outils système et les configurations. Cette méthode nous permet de créer des environnements de développement et de test cohérents et reproductibles, indépendamment du système d'exploitation hôte.

Ces environnements sont construits à l'aide d'une image Docker. Bien qu'il en existe des déjà faites, il est également possible de créer sa propre image à l'aide d'un Dockerfile.

Un Dockerfile est un fichier texte qui contient toutes les instructions nécessaires pour créer une image Docker. A partir d'une image de base, on y spécifie les instructions pour installer et configurer l'environnement.

En utilisant un Dockerfile, nous pouvons automatiser le processus de création d'un environnement pour exécuter IAI_ROS2 et Openpose_ROS2, et garantir le fonctionnement qu'importe la machine hôte.

Dans le cadre de notre projet, nous avons besoin de deux containers :

- Un premier pour Openpose, se basant sur l'image de ROS2 Dashing (Ubuntu 18).
- Un second pour IAI, se basant sur l'image de CUDA 12.5 (la dernière version du pilote CUDA, pour Ubuntu 22).

Nous pensions n'avoir besoin que d'un container pour Openpose. Cependant, nous n'avons pas réussi à faire communiquer par l'intermédiaire de ROS2 le container et la machine hôte, alors qu'il était possible pour deux containers de communiquer ensemble.

b. Docker_Openpose.

Dans ce second container, nous voulons y préparer un environnement pour Openpose et Openpose_ROS2.

Dans un premier temps, nous voulions utiliser une image avec CUDA 10.2 et CuDNN 7.6.5 préinstallé, mais quelques mois auparavant, Nvidia supprima ces images et les forks que nous avons trouvé ne fonctionnent pas.
Dans un second temps, nous avions trouvé une image d'Openpose, mais cette dernière était sur Ubuntu 22.
Finalement, notre choix s'est finalement arrêté sur osrf/ros:dashing-desktop, l'image officielle de ROS2 Dashing car l'initialisation et l'installation de ROS2 y est déjà faite.

Avant de commencer avec les instructions du Dockerfile, nous avons défini certaines variables d'environnement nécessaires et exposé les ports requis pour les communications réseau :

- `ENV DEBIAN_FRONTEND=noninteractive` : Définit l'interface frontale de Debian en mode non-interactif pour éviter les prompts interactifs pendant l'installation des paquets. Cela permet une installation automatique sans intervention humaine.
- `ENV NVIDIA_VISIBLE_DEVICES=all` : Spécifie que tous les périphériques GPU disponibles doivent être visibles à l'intérieur du conteneur Docker, permettant aux applications de les utiliser pour le calcul GPU.
- `ENV NVIDIA_DRIVER_CAPABILITIES=video, compute, utility` : Définit les capacités du pilote NVIDIA à activer dans le conteneur. Ici, les capacités vidéo, de calcul et d'utilitaire sont activées, nécessaires pour le traitement vidéo et les calculs GPU intensifs.
- `EXPOSE 9090 11311 8080` : La directive EXPOSE indiquent les ports que le conteneur va écouter, soit le port 9090 souvent utilisé pour les interfaces web, le port 11311, utilisé par défaut pour le maître ROS et le port 8080, généralement utilisé pour les services web ou les applications HTTP.

Puisque nous allions utiliser une image sans les outils d'accélération matériel, il était de notre ressort de les installer manuellement.

En reprenant les commandes utilisées pour installer CUDA sur la session Ubuntu 18.04 qui ont été donnée plutôt, l'installation fut aisée.
A savoir, il est impossible de récupérer les fichiers de CuDNN avec la commande `wget` : ces fichiers nécessitent de se connecter au site de Nvidia pour les télécharger. Pour résoudre ce contretemps, il a suffi d'utiliser la commande `COPY` du Dockerfile pour transférer des fichiers dans un container.

Une fois CUDA et CuDNN installé, il est temps d'installer Openpose et Openpose_ROS2.

Encore une fois, nous avons suivi les étapes données dans le chapitre sur le Traitement. Si l'installation de Openpose_ROS2 n'a pas causé de problème, ce ne fut pas complètement le cas de Openpose.
Précédemment, nous avions utilisé l'interface graphique de cmake (cmake-gui) pour construire le projet, mais il n'est pas possible d'utiliser un gui dans un Dockerfile. Après quelques recherches, nous avons trouvé la commande pour construire le projet. Cette dernière se trouve dans l'annexe (code 8). Voici une explication détaillée de chaque argument, avec la raison de sa présence :

- `-DBUILD_PYTHON=ON` : Active la compilation des bindings Python pour Openpose. Openpose_ROS2 nécessite d'activer ce paramètre.
- `-DCMAKE_CUDA_COMPILER=/usr/local/cuda-10.2/bin/nvcc` : Spécifie le compilateur CUDA à utiliser.
- `-DCUDA_ARCH_BIN="6.0 6.1 7.0"` : Définit les architectures CUDA supportées (GPU utilise l'architecture 6.1). Sans cette ligne, la compilation dure beaucoup plus longtemps.
- `-DCUDA_ARCH_PTX=""` : Désactive la génération de code PTX (Parallel Thread Execution) (comme précédent).
- `-DDOWNLOAD_BODY_25_MODEL:Bool=OFF` : Désactive le téléchargement du modèle BODY_25.
- `-DDOWNLOAD_BODY_MPI_MODEL:Bool=OFF` : Désactive le téléchargement du modèle BODY_MPI.
- `-DDOWNLOAD_BODY_COCO_MODEL:Bool=OFF` : Désactive le téléchargement du modèle BODY_COCO.
- `-DDOWNLOAD_FACE_MODEL:Bool=OFF` : Désactive le téléchargement du modèle de détection de visage.
- `-DDOWNLOAD_HAND_MODEL:Bool=OFF` : Désactive le téléchargement du modèle de détection de mains.
- `-DUSE_MKL:Bool=OFF` : Désactive l'utilisation de la bibliothèque MKL (Math Kernel Library) (recommandé sur un forum).

Nous avons désactivé le téléchargement des modèles car, comme expliqué dans le chapitre précédent, ces derniers étaient tous corrompus. De plus, le temps de téléchargement de ces fichiers causait un 'Timeout' lors de la construction de l'image Docker.

Enfin, nous avons configuré les variables d'environnement de ROS2.

Pour ce faire, nous avons écrit dans le fichier `~/.bashrc` :

- `export ROS_LOCALHOST_ONLY=0` : Permet à ROS2 de communiquer non seulement sur le localhost mais aussi sur les interfaces réseau externes.
- `export ROS_DOMAIN_ID=5` : Définit l'identifiant du domaine ROS2 pour la communication DDS (Data Distribution Service), permettant d'isoler les systèmes ROS2 en fonction de leurs domaines. Nous avons choisi arbitrairement le numéro 5.
- `source /opt/ros/dashing/setup.bash` : Charge les configurations et les chemins d'environnement spécifiques à la distribution ROS2 Dashing.
- `export PATH="/root/openpose/bin:${PATH}"` : Ajoute le chemin du binaire d'Openpose à la variable d'environnement `PATH`, permettant un accès facile aux exécutables d'Openpose.

- `export LD_LIBRARY_PATH="/usr/local/cuda-10.2/lib64:${LD_LIBRARY_PATH}"` : Ajoute le chemin des bibliothèques CUDA à la variable `LD_LIBRARY_PATH` pour permettre la liaison dynamique des bibliothèques CUDA.
- `export CUDA_HOME="/usr/local/cuda"` : Définit la variable `CUDA_HOME` pour spécifier le répertoire d'installation de CUDA.
- `export ROS_MASTER_URI=http://$(hostname --ip-address):11311` : Configure l'URI du maître ROS pour utiliser l'adresse IP de la machine courante sur le port 11311, nécessaire pour le networking ROS.
- `export ROS_HOSTNAME=$(hostname --ip-address)` : Définit le nom d'hôte ROS à l'adresse IP de la machine courante, essentiel pour la résolution des noms dans le réseau ROS.

Il est fondamental que les variables réseaux de ROS2 soient identiques sur toutes les machines qui communiquent ensemble.

Une fois ce Dockerfile rédigé, l'image de Docker_Openpose fut construit avec succès.

c. Docker_IAI.

Une fois le Dockerfile de Openpose et Openpose_ROS2 écrit, nous sommes passé à IAI_ROS2.

Nous avons choisi d'utiliser l'image de base nvidia/cuda:12.5.0-devel-ubuntu22.04, qui inclut CUDA 12.5 et une distribution Ubuntu 22.04.
Contrairement au précédent Dockerfile, nous avons accès aux images de Nvidia : nous allons donc utiliser l'image avec la dernière image de CUDA supporté par le GPU de l'ordinateur hôte, CUDA 12.5. Utiliser cette image nous obligera à installer **ROS2 Humble**.

Tout comme le précédent Dockerfile, nous avons défini les mêmes variables pour le container (port, noninteractive etc.).

Pour installer ROS2, il suffit de suivre le tutoriel sur le site de la distribution de ROS2 Humble :

1. Définir la langue et le fuseau horaire (sinon, la construction de l'image restera bloquée, code).
2. Ajout des sources ROS 2.
3. Puis pour installer ROS2 Humble et ses outils de développement :

Nous n'oublions pas d'ajouter au bashrc les même variables d'environnement que le container d'Openpose.

Une fois ROS2 Humble installé, il faut installer libfreenect2 et notre version de IAI_ROS2 (avec la node d'enregistrement).

Cependant, comme expliqué à la fin du chapitre III.1.a, nous n'avons pas réussi à installer libfreenect2 et IAI_ROS2 du fait de la librairie manquante.
Pour rappel, nous avons besoin de librairies présente dans le répertoire de librairie **cuda_samples**, mais ce dernier n'est pas installé par défaut dans le container. Nous avons donc suivi le tutoriel d'installation sur le GitHub avec succès.
Cependant, nous n'avons pas trouvé où ces bibliothèques étaient situées, et les variables d'environnement définies lors de l'installation pointaient vers des dossiers qui n'existaient pas.
Nous avons essayé de créer puis de déplacer les bibliothèques à cet emplacement, sans succès.

d. Nvidia Container Toolkit.

Maintenant que les deux Dockerfiles ont été rédigé, il est temps de passer à l'ultime étape de ce projet : installer NVIDIA Container Toolkit.

NVIDIA Container Toolkit est essentiel pour permettre aux conteneurs Docker d'accéder aux capacités GPU de l'ordinateur en intégrant les GPU NVIDIA dans les environnements des conteneurs.
Ce toolkit doit être installé sur la machine hôte, à la différence de CUDA toolkit et CuDNN.

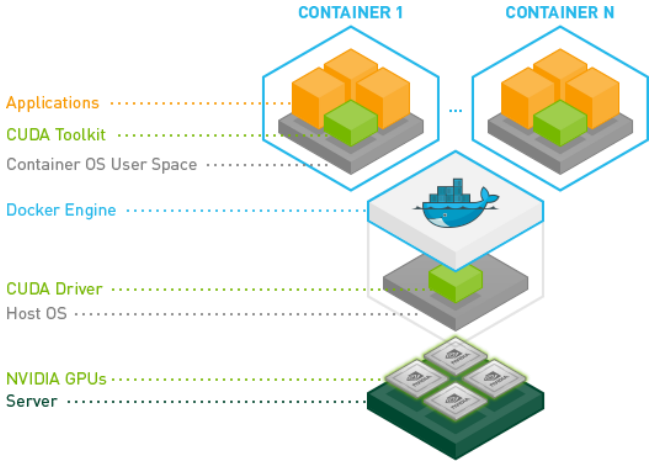

Figure 2 : Schéma de fonctionnement de CUDA en tandem avec Docker.

Le seul prérequis est de s'assurer que les pilotes Nvidia et que CUDA soient installés sur la machine hôte. Une fois ceci fait, il suffit de suivre le tutoriel sur la documentation de Nvidia Container Toolkit.

e. Lancer les containers.

Maintenant que les Dockerfiles ont été écrit et fonctionnent, nous utilisons Docker Compose pour les exécuter ensemble. Cet utilitaire Docker permet de lancer simultanément plusieurs images Docker avec les mêmes arguments.
Pour utiliser Docker Compose, il faut créer un fichier nommé *`docker-compose.yml`* dans le même répertoire que nos Dockerfiles. Ce fichier définira les services pour les deux images Docker avec les arguments requis.

Notre répertoire devrait ressembler à cela :

```
Répertoire Docker
    ├── docker-compose.yml
    ├── openpose_docker/
    │   ├── Dockerfile
    │   └── Archives
    │       └── Toutes les ressources nécessaires à la construction de l'image
    └── iai_docker/
        ├── Dockerfile
        └── Archives
            └── Toutes les ressources nécessaires à la construction de l'image
```

Vous trouverez notre fichier compose en entier en annexe (voir code 10).

Ces containers nécessitent des arguments particuliers pour être exécuter :

- L'argument `pid: host` indique que le conteneur n'isolera pas son espace PID (Process ID), permettant ainsi aux processus du conteneur de voir et d'interagir avec tous les processus de l'hôte.
- De même, l'argument `network_mode: host` signifie que le conteneur utilisera le réseau de l'hôte, partageant ainsi les interfaces réseau et les configurations réseau avec l'hôte.
- L'argument `ipc: host` signifie que le conteneur partagera l'espace de communication inter-processus avec l'hôte, permettant le partage de mémoire et de files de messages entre le conteneur et l'hôte.
  Sans ces trois premiers arguments, nous risquons d'avoir des problèmes avec les communications ROS2.
- Enfin, l'argument `runtime: nvidia` indique que le container pourra exploiter les GPU de la machine hôte pour les applications tournant dans le conteneur.

Pour construire le docker compose, il suffit d'entrer la commande `docker-compose build`.
Une fois construit, la commande `docker-compose up` lancera les containers et utiliser `docker-compose down` les stoppera.

IV. Conclusion

Confrontés à des erreurs de conception initiales, nous avons réussi à les corriger en implémentant des mécanismes de pile et de threads pour un enregistrement efficace des images. La migration du programme d'enregistrement vers ROS2 a également été réalisée avec succès.

Nous avons réussi à résoudre les problèmes de versions causés par les programmes utilisés grâce à Docker, et identifié puis contourné une défaillance dans la communication ROS2 du container.

Ce projet nous a permis d'acquérir des compétences approfondies en C++, notamment en optimisation de la mémoire avec des threads. Nous avons également renforcé notre expertise dans l'utilisation de ROS2, en particulier sur le volet réseau.

En outre, ce projet nous a introduit à Docker, nous permettant de mieux comprendre ses utilités, ses atouts et ses contraintes. Enfin, nous avons acquis des connaissances sur l'utilisation et ce qu'est l'accélération matérielle.

Nous aurions aimé conclure en affirmant que tout fonctionne parfaitement, mais ce n'est pas le cas : les problèmes causés par les bibliothèques nous ont considérablement ralentis et ne sont toujours pas corrigés, si bien que la partie Openpose a été à peine développé. Maintenant que les environnements Docker sont prêts, nous pouvons enfin comprendre le fonctionnement de Openpose_ROS2. Enfin, comprendre et résoudre les problèmes causés par les bibliothèques CUDA nous permettra d'améliorer les performances du driver Kinect et, possiblement, de résoudre le problème de IAI_ROS2.

En conclusion, ce projet a permis de résoudre des problèmes majeurs liés à l'acquisition et au traitement des images en temps réel, tout en établissant plusieurs environnements de travail pérennes et faciles à déployer pour la suite de ce sujet.

V. Annexes

1. Code

```
1. struct ImagePair { cv::Mat color_image;  cv::Mat depth_image; };
2. queue<ImagePair> image_queue_;
3. mutex queue_mutex_;
4. condition_variable queue_cond_var_;
```

Code 1 : Ce code définit une pile pour stocker les images, un mutex pour gérer les accès concurrents et une condition variable pour synchroniser les threads.

```
1. ImagePair image_pair;
 2. {
 3.     unique_lock<mutex> lock(queue_mutex_);
 4.     queue_cond_var_.wait(lock, [this] { return !recording_ ||!image_queue_.empty(); });
 5.
 6.     if (!recording_ && image_queue_.empty())
 7.         return;
 8.     if (!image_queue_.empty()) {
 9.         image_pair = image_queue_.front();
10.         image_queue_.pop();
11.     }
12.     else
13.         return;
14. }
15. queue_cond_var_.notify_one();
16. saveImagesToDisk(image_pair.color_image, image_pair.depth_image);
```

Code 2 : Ce code extrait une paire d'images de la pile de manière sécurisée en utilisant un mutex et une condition variable, puis notifie un thread qu'il peut réaliser un enregistrement.

```
1. {
2.     lock_guard<mutex> lock(queue_mutex_);
3.     image_queue_.emplace(ImagePair{color_image, depth_image});
4. }
```

Code 3 : Ce code verrouille la pile avant d'ajouter une nouvelle paire d'images, garantissant ainsi que l'opération se déroule en toute sécurité.

```
1. for (size_t i = 0; i < num_thread_; ++i)
2.     consumer_threads_.emplace_back(&Kinect2Record::consumerThread, this);
```

Code 4 : Ce code montre la création de threads consommateurs, ici le nombre de threads est ajustable.

```
1. void consumerThread() {
 2.     while (true) {
 3.         ImagePair image_pair;
 4.         {
 5.             unique_lock<mutex> lock(queue_mutex_) ;
 6.              queue_cond_var_.wait(lock, [this] { return !recording_ || !image_queue_.empty();
});
 7.
 8.             if (!recording_ && image_queue_.empty())
 9.                 return;
10.             if (!image_queue_.empty()) {
11.                 image_pair = image_queue_.front() ;
12.                 image_queue_.pop();
13.             }
14.             else
15.                 return;
16.         }
17.         queue_cond_var_.notify_one();
18.         saveImagesToDisk(image_pair.color_image, image_pair.depth_image) ;
19.     }
20. }
```

Code 5 : Ce code montre le fonctionnement du thread consommateur, qui extrait et enregistre les images de la pile.

```
1. openpose_ros2:
2.   ros__parameters:
3.     is_debug_mode: false
4.     openpose_root: "/path/to/openpose"
5.     is_image_compressed: true
6.     image_node: "/camera/color/image_raw"
```

Code 6 : Fichier Params.yaml où l'on définit les paramètres ROS2 du wrapper. On y met le chemin racine d'Openpose, le mode de débogage, l'utilisation d'images compresses et la node qui publie les images.

```
1.    find_package(openpose_ros2_msgs REQUIRED)
2.    include_directories(
3.      ${openpose_ros2_msgs_INCLUDE_DIRS}
4.    )
5.
6.    ament_target_dependencies(test_openpose
7.      rclcpp
8.      sensor_msgs
9.      cv_bridge
10.     OpenCV
11.     openpose_ros2_msgs
12.   )
```

Code 7 : Morceau du CMakeList pour ajouter le custom message au projet.

```
1. cmake \
 2.     -DBUILD_PYTHON=ON \
 3.     -DCMAKE_CUDA_COMPILER=/usr/local/cuda-10.2/bin/nvcc \
 4.     -DCUDA_ARCH_BIN="6.0 6.1 7.0" \
 5.     -DCUDA_ARCH_PTX="" \
 6.     -DDOWNLOAD_BODY_25_MODEL:Bool=OFF \
 7.     -DDOWNLOAD_BODY_MPI_MODEL:Bool=OFF \
 8.     -DDOWNLOAD_BODY_COCO_MODEL:Bool=OFF \
 9.     -DDOWNLOAD_FACE_MODEL:Bool=OFF \
10.     -DDOWNLOAD_HAND_MODEL:Bool=OFF \
11.     -DUSE_MKL:Bool=OFF \
12.     ..
```

Code 8 : Commande à mettre dans le Dockerfile pour construire Openpose.

```
1. RUN apt-get update && apt-get install -y locales && \
2.     locale-gen en_US en_US.UTF-8 && \
3.     update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8 && \
4.     export LANG=en_US.UTF-8
5.
6. ENV TZ=Europe/Paris
7. RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone
8  RUN apt-get update && apt-get install -y software-properties-common curl && \
9.     add-apt-repository universe && \
10.    curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o
/usr/share/keyrings/ros-archive-keyring.gpg && \
11.    echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-
keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME)
main" | tee /etc/apt/sources.list.d/ros2.list > /dev/null
12.    RUN apt-get update && apt-get upgrade -y && apt-get install -y \
13.    ros-humble-desktop \
14.    build-essential git nano wget iputils-ping libpython3-dev python3-pip python3-colcon-
common-extensions python3-colcon-mixin python3-rosdep python3-setuptools python3-vcstool \
15.    && pip3 install -U argcomplete \
```

Code 9 : La partie du Dockerfile qui installe ROS2 Humble.

```
1. version: '3.8'
 2.
 3. services:
 4.   openpose:
 5.     build:
 6.       context: ./openpose_docker  # chemin vers le répertoire contenant le Dockerfile pour
l'image openpose
 7.     runtime: nvidia
 8.     deploy:
 9.       resources:
10.         reservations:
11.           devices:
12.             - capabilities: [gpu]
13.     network_mode: host
14.     ipc: host
15.     pid: host
16.
17.   iai:
18.     build:
19.       context: ./iai_docker  # chemin vers le répertoire contenant le Dockerfile  pour l'image
iai
20.     runtime: nvidia
21.     deploy:
22.       resources:
23.         reservations:
24.           devices:
25.             - capabilities: [gpu]
26.     network_mode: host
27.     ipc: host
28.     pid: host
```

Code 10 : Notre docker-compose.

2. Log

```
X Error of failed request:  BadAccess (attempt to access private resource denied)
Major opcode of failed request:      152 (GLX)
Minor opcode of failed request:      5 (X_GLXMakeCurrent)
```

Log 1 : Erreur de compilation de IAI_ROS2

3. Figures

Source de l'image :
https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/index.html

VI. Abstract

This project aims to extract skeletons and estimate the joint points of individuals in real-time using an RGBD camera. These data will subsequently be used to determine the actions of the individuals.

Initially, this project involved setting up several programs to perform a series of real-time operations. The versatile and modular nature of ROS2 allows us to use it to manage communication between the different programs.

However, after several setbacks, this project evolved into the development of an environment where these different programs could operate on the same computer.
To achieve this, we chose to use Docker to build two containers (virtual machines): one for the raw image collection and another for the retrieval and processing of the relevant data.
These data would then be recorded on the host machine.