令和 6 年度卒業研究報告書

AI 専用 PC の Live インストールイメージの作成

情報技術科　千葉　桂大

指導教員　　飯坂　ちひろ

目次

第 1 章　はじめに 4

第 2 章　研究概要 5

2.1 Live インストールイメージとは 5

2.2 ディストリビューションの選定理由 5

2.2.1　他のディストリビューションとの比較 6

2.2.2　Debian の主な特徴 7

2.3 公式の Debian との差異 8

第 3 章　Live インストールイメージの開発環境 9

第 4 章　開発の流れ 10

4.1 環境構築 10

4.2 ライブイメージの動作確認 11

4.2.1　デスクトップのイメージ 12

4.3 パッケージの追加 13

4.3.1　Docker のインストール 14

4.3.2　NVIDIA ドライバのインストール 14

4.4 アプリケーション毎の設定 15

4.4.1　Google Chrome の設定 15

4.4.2　シェルの設定 16

4.4.3　テーマの設定 16

4.4.4　CUI テキストエディタの設定 17

4.4.5　日本語表示の設定 18

4.4.6　dconf の設定 18

4.4.7 日本語入力の設定 19
4.4.8 Docker の設定 19
第 5 章 アプリケーションの作成 21
5.1 作成理由とその目的 21
5.2 アプリケーションの開発環境 21
5.3 アプリケーションの構造 23
5.4 作成したプログラムの説明 24
5.5 Live イメージへのインストール 27
第 6 章 動作確認 28
6.1 不具合の発見 28
6.2 不具合の一時的な修正 28
6.3 不具合の根本的な原因 29
6.4 不具合の原因の修正 32
6.5 Docker 内での nvidia-smi を用いた動作確認 33
6.6 cuda-sample:nbody を用いた動作確認 33
6.7 Ollama+Llama 2 を用いた動作確認 34
6.8 Ollama+DeepSeek-R1 を用いた動作確認 36
第 7 章 成果物 38
7.1 live-build の設定ファイル 38
7.2 Docker を用いた live-build 38
7.3 第 5 章で作成したアプリケーション 39
7.4 Live インストールイメージの本体 39
第 8 章 おわりに 40
第 9 章 参考文献 41

第1章　はじめに

現在、情報技術科にはAI用に特化したパソコンが設置されている。

今年、Ubuntu 18.04のサポート終了に伴ってOSの入れ替え作業を行ったが、そのままではメンテナンスをする人がいない、リカバリメディアが存在しないなどの問題が存在する。

そこで、これらの問題を解決する手段として、Debianを用いたLiveインストールイメージの作成に取り組んだ。

第 2 章　研究概要

2.1 Live インストールイメージとは

Live インストールイメージとは、OS をインストールする前に、USB メモリや DVD からシステムをインストールせずに試すことが出来るものである。

今回作成する Live インストールイメージは、付属するインストーラーを用いてそのままインストールできるようにする。

2.2 ディストリビューションの選定理由

一概に Linux と言っても様々な種類が存在し、それらはディストリビューションと呼ばれている。そして、Linux のディストリビューションには、いくつかの系統がある。

例えば、デスクトップ用途からサーバー用途まで、幅広く使用されている Ubuntu は、Debian をベースにしているため Debian 系、主にサーバー用途で利用されていた CentOS は、RedHat Enterprise Linux をベースにしているため、RedHat 系と呼ばれる。

他にも数多くのディストリビューションやその系統が存在する中、Debian は安定性において非常に高い評価を受けている。

また、本校の授業で扱うディストリビューションも、１年次には Ubuntu、２年次には Debian と、どちらも Debian をベースとしたディストリビューションであるため、他のディストリビューションに比べて利用者が扱いやすいと考え、Debian を採用することにした。

2.2.1　他のディストリビューションとの比較

他のディストリビューションと、Debian との比較を行う。

エラー! 参照元が見つかりません。エラー! 参照元が見つかりません。は、二つのディストリビューションと Debian との比較を行ったものである。この表の左から、Debian、Redhat Enterprise Linux、Arch Linux である。

他のディストリビューションと比較すると、Debian の主な強みとして、情報量の多さと価格、そして安定性が挙げられる。

まず、情報量の多さについて説明する。

Debian とそれをベースにしている Ubuntu は、共に Linux のディストリビューションの中でトップクラスのシェアを誇る。

そのため、ネット上にある情報量が非常に多く、必要な情報を入手しやすいという利点がある。

次に、価格での利点が大きい。

Redhat Enterprise Linux はサーバー用に開発されている有償のディストリビューションであり、高い安定性と信頼性の高いサポート体制で、サーバー用途では非常に高いシェアを誇っている。しかし、年額のサブスクリプション金額が非常に高く、手が出しづらい。

それと比較して Debian は Redhat Enterprise Linux ほどのサポートを期待することはできないが、無償であり、高い安定性を有している。

表 2.1 他のディストリビューションとの比較

	Debian	RedHat Enterprise Linux	Arch Linux
安定性	高い	高い	低い
パッケージの新しさ	古い	古い	新しい
リリースモデル	固定リリース	固定リリース	ローリングリリース
パッケージ管理	DEB (apt)	RPM (yum, dnf)	Pacman
主なターゲット	サーバー・デスクトップ	サーバー	デスクトップ
情報の多さ	とても多い	まあまあ多い	まあまあ多い

価格	無料	有料（最安で１年/¥125,700 ~）	無料

2.2.2 Debian の主な特徴

Debian は、安定性に重きを置いている分、パッケージのバージョンが他のディストリビューションよりも古い傾向がある。

表 2.1 で比較した Arch Linux というディストリビューションは、他の二つとは違い、ローリングリリースという方式を採用している。ローリングリリースとは、Debian のようなバージョン番号を持たず、常にアップデートによって断続的に更新されているリリース方式のことだ。ローリングリリースは常に最新のパッケージを利用することが可能だが、その反面、安定性に関しては全く持って期待することができないといっても過言ではない。

今回は、不特定多数の利用者がいること、また、継続的なメンテナンスが期待できないことから、新しいパッケージを利用できるよりも、ある程度メンテナンスをせずとも動作する安定性が重要だと考えた。

もし、利用者が新しいパッケージを利用したい場合にも、パッケージ側が Debian をサポートしている場合が多く、手動でインストールすることが可能だ。

また、Docker を用いることで間接的に新しいバージョンのパッケージを利用することが出来る。

なお、Docker とは、コンテナ型仮想化技術を利用したプラットフォームである。アプリケーションの実行に必要な環境を「コンテナ」と呼ばれる独立したパッケージにまとめ、異なる環境でも同じように動作させることを可能にする。

2.3　公式の Debian との差異

AI 専用 PC に搭載されている、NVIDIA 製の GPU を Linux 上で動作させるためには、煩雑な設定が必要となる。しかし、このカスタムした Debian の Live インストールイメージを使用すれば、そうした設定を省略して簡単に環境を構築できる。

また、システムに問題が発生した場合でも、このイメージを使えば再インストールが容易かつ、初期設定の手間も減らすことが出来る。

第 3 章　Live インストールイメージの開発環境

本研究に使用したライブイメージの開発用 PC の環境及び AI 専用 PC の構成は、表 3.1 及び表 3.2 の通りである。

表 3.1 ライブイメージ開発用 PC の環境

機種名	PC-VK540
CPU	AMD Ryzen 5 5500U
OS	Debian 12 (bookworm)
使用ツール	live-build

表 3.2 AI 専用 PC の構成

OS	Ubuntu 18.04
CPU	Intel Xeon W-2223
GPU	NVIDIA GeForce RTX3090
メモリ	Kingston 8GBx4 (32GB)
ストレージ	SSD 1TB x 2

第 4 章　開発の流れ

4.1 環境構築

当初は Windows の WSL 上に Debian をインストールし、開発を進める予定だった。

WSL（Windows Subsystem for Linux）とは、Windows 上で Linux の実行環境を提供する機能である。Windows を離れることなく、Linux ディストリビューションのコマンドラインツール、ユーティリティ、アプリケーションを直接実行できる。

しかし、途中からライブイメージ開発用 PC に Debian を直接インストールし、開発を進めた。

WSL から移行したメリットとして、WSL 上で開発を行った場合よりもファイルアクセスが高速化し、より快適に作業が行えるようになった。

また、ライブイメージ開発用 PC にインストールした Debian に、今回使用するパッケージである Live-build をインストールする。これは、先に説明した Debian のライブイメージを、自分好みにカスタマイズして作成するための Debian 公式のツールである。

Live-build を使うと、必要なソフトウェアを最初から含めることが出来る他、システム設定を調整したライブイメージを作成でき、特定の用途に特化した環境を作成することが出来る。

4.2 ライブイメージの動作確認

ここでは、作成したライブイメージの動作確認を行う。なお、ハードウェアに起因しない箇所については仮想環境上で動作確認を行った。

最低限の GUI 環境を動作させることが出来る状態での動作確認を行った時点で、インストールが止まってしまう不具合が発生したため、原因を特定した。

不具合の原因を特定するために、インストーラーのログを確認し、状況の把握を行った。インストーラーのログによると、インストーラーの終盤に行われるプロセスで、システム上重要なパッケージが既に不要なパッケージとして削除されようとしていたため、パッケージマネージャーがエラーを返していることが原因だった。

そのため、live-build でライブイメージをビルドする際に、明示的にそのパッケージを指定してインストール対象にすることで、自動的に削除される対象から外すことに成功した。

その後、再度インストールをテストし、エラーが発生せずにインストールが完了することを確認した。

また、AI 専用 PC 上での動作確認として、インストールを除いた簡易的な動作確認を行った。そこで、グラフィックの乱れが発生したため、こちらも修正を試みた。この不具合は、NVIDIA 製の GPU と、この動作確認の時点で採用していたデスクトップ環境との相性による問題であると考え、デスクトップ環境を KDE Plasma から Cinnamon に変更した。

この変更により、この不具合を回避することに成功したため、今後も Cinnamon に変更して開発を続けることにした。

4.2.1　デスクトップのイメージ

図 4.1 が、元々採用する予定だったデスクトップ環境である KDE Plasma 5 であり、図 4.2 が、最終的に採用した Cinnamon である。

どちらも普段使い慣れているであろう Windows に UI の配置が似ており、他のデスクトップ環境に比べて移行しやすい。

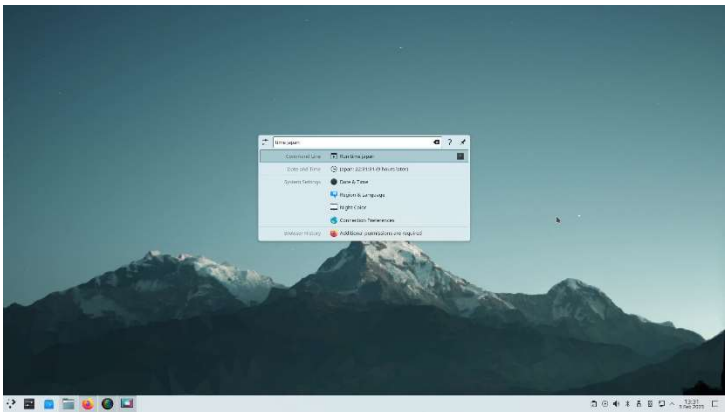

図 4.1 KDE Plasma 5 のデスクトップ画面

（出典：https://kde.org/announcements/plasma/5/5.27.0/）

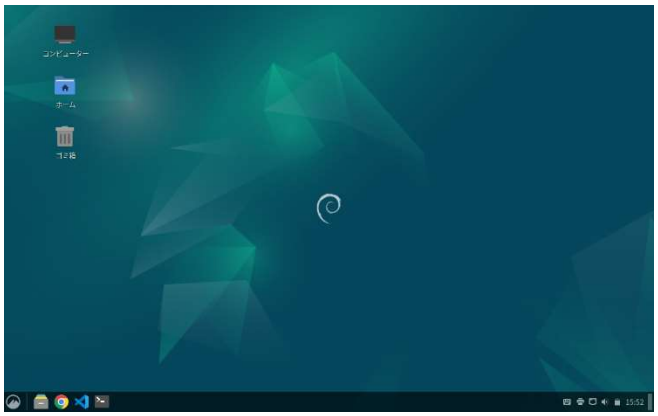

図 4.2 Cinnamon のデスクトップ画面

4.3 パッケージの追加

動作確認で使用した環境に加えて、元々の環境に入っていたパッケージを参考に、そのほかのパッケージを追加する。Debian の安定性を最大限に生かすため、パッケージを取得するリポジトリは出来るだけ Debian 公式のものを利用する。

ここでいうリポジトリとは、アプリケーション、パッケージを管理しておく倉庫のようなものであり、ユーザーはこのリポジトリから必要なソフトウェアをダウンロードし、インストールすることが出来る。

その中でも Debian 公式のリポジトリは Debian のプロジェクトによって管理されており、セキュリティや安定性が確保されている。

反対に、非公式リポジトリは、Debian のプロジェクトとは別の個人や団体によって運営されており、Debian 公式のリポジトリにはないソフトウェアや、より新しいバージョンのソフトウェアが提供されている反面、Debian のプロジェクトによって管理されていないため、Debian 公式のリポジトリに存在するソフトウェアに比べて安定動作しない可能性がある。

しかし、元々の環境にインストールされている Visual Studio Code（VSCode）や Google Chrome といった Debian 公式のリポジトリに存在しないソフトウェアについては、各ソフトウェア公式のリポジトリを登録し、インストールした。

4.3.1 Docker のインストール

例外として、Docker は Debian 公式のリポジトリに存在するが、後に動作確認を行う際に、バージョンによる互換性が原因の不都合が発生した。

そのため、VSCode や Google Chrome と同じようにソフトウェア公式のリポジトリを追加してインストールすることで、より新しいバージョンをインストールした。

4.3.2 NVIDIA ドライバのインストール

NVIDIA 製の GPU を動作させるために必要なドライバも、NVIDIA 公式のリポジトリを追加することによって、より新しいバージョンのドライバをインストールすることが可能だ。

しかし、安定性に難があり、致命的な不具合の原因になりかねないため、比較的安定している Debian 公式のリポジトリからインストールした。

4.4 アプリケーション毎の設定

4.4.1 Google Chrome の設定

元々の環境を使用した際に、過去の利用者のログイン情報がブラウザに残っていたことが気になったため、ブラウザのゲストモードをデフォルトで有効化することで、ログイン情報の保存を無効化している。

これにより、不特定多数の利用者が使う AI 専用 PC において、プライバシーを保護することが可能である。

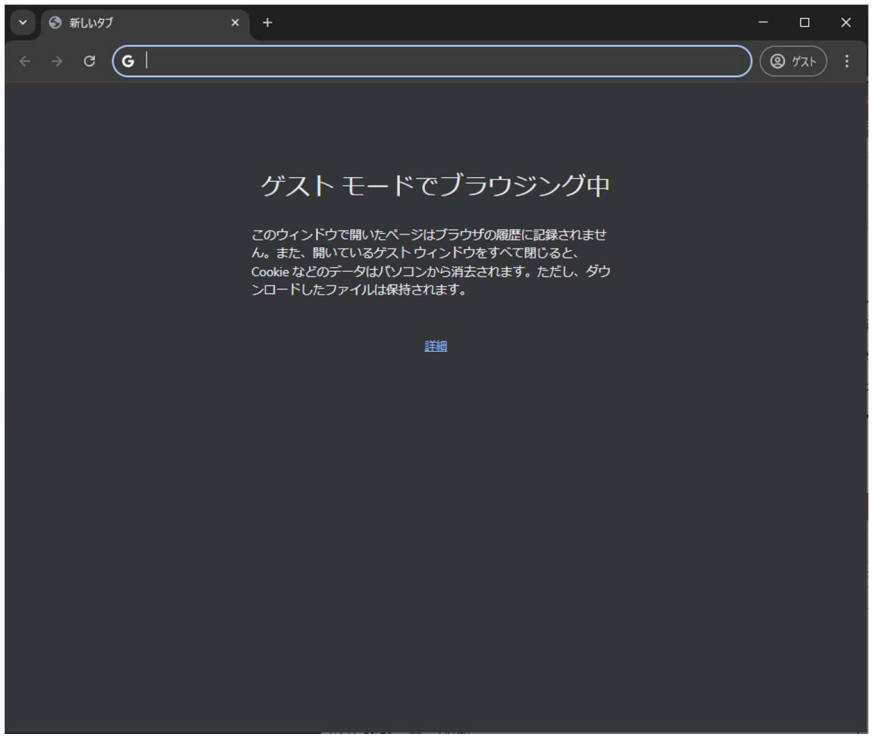

図 4.3 Google Chrome のゲストモード

4.4.2　シェルの設定

CLI 上での操作が楽になるよう、シェルの自動補完などの設定を一部変更している。

Debian のデフォルトで使用されるシェルは Bash（Bourne Again shell）だが、これを zsh（Z Shell）に置き換えている。これにより、bash に比べて高機能な入力補完を利用することが可能となっている。これは、コマンドだけでなく、Docker のコンテナ ID も補完することが可能なため、この AI 専用 PC で使われることが多いであろう Docker の操作も楽になっている。

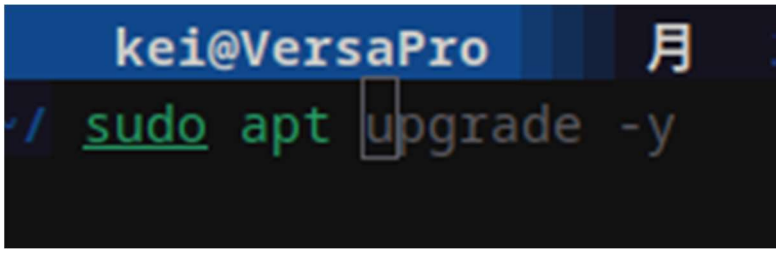

図 4.4 zsh の自動補完

4.4.3　テーマの設定

デスクトップ上でテーマがバラバラにならないよう、Linux で主に使われている GUI ツールキットである Qt と GTK の両方でテーマを明示的に指定している。また、デフォルトで設定されているデスクトップ環境自体のテーマも、一昔前の古さを感じるテーマだったため変更し、同様の理由でアイコンのテーマも変更している。

4.4.4　CUI テキストエディタの設定

本校の授業にて Linux を扱う際には、Vi エディタの使い方を学ぶ。

そのため、Vi を拡張した Vim をメインの CUI テキストエディタとして使用することを前提として、一部設定を変更した。具体的には、行番号の表示やシンタックスハイライトの有効化、タブ文字の可視化など、デフォルトの設定を拡張するような方向で設定をしている。

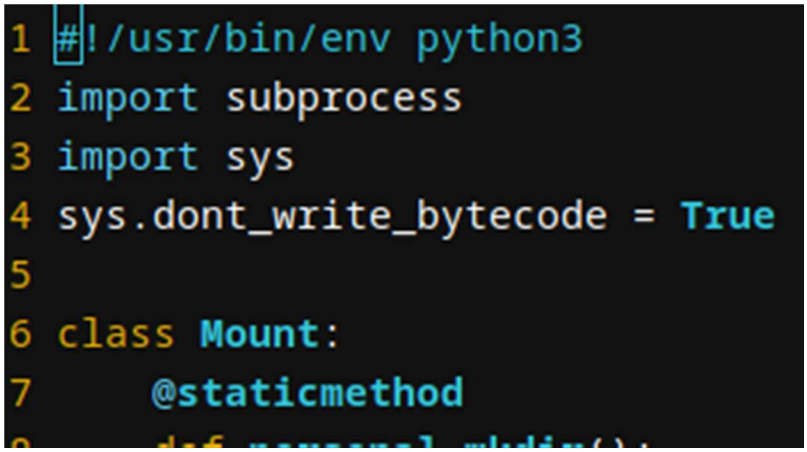

図 4.5 vim の画面

加えて、Python を含む多くのプログラミング言語では、タブ文字によるインデントではなく、スペース文字でのインデントが推奨されているため、タブ文字を入力する代わりに二つスペースが入力されるように設定している。

なお、Makefile などのスペース文字でのインデントが使用できない場合には、vim の機能で自動的にオフになる。

4.4.5　日本語表示の設定

Linux のディストリビューションによっては、最初から日本語を表示するために必要なフォントがインストールされているものもあるが、今回はインストールするパッケージのほとんどをこちらで指定するため、自動的にインストールされることはない。

日本語を表示するためのフォントがないと、文字の代わりに白い四角に文字化けしてしまう。そのため、手動でインストールする必要がある。

今回インストールしたフォントは、Noto フォントだ。このフォントは多くの言語に対応し、白い四角（豆腐）をなくすことを目的に作られているため、

Noto (**No tofu**)

という名称が付けられている。(出典：https://fonts.google.com/noto)

このフォントはオープンソースであり、また、ほとんどの Unicode をカバーしているため、世界中の様々な言語や記号を表示することが可能だ。

また、Debian のリポジトリに存在しているため、インストールが簡単に出来る。

4.4.6　dconf の設定

dconf とは、Gnome デスクトップ環境や、Gnome をベースにしたデスクトップ環境で使われる、設定を保存するシステムで、Windows のレジストリのような機能を担っている。

今回採用したデスクトップ環境である Cinnamon は、Gnome をフォークして開発されていたため、Cinnamon と関連するアプリケーションの設定では、dconf を用いる。

ここで先ほどインストールした Noto フォントをシステム全体のデフォルトフォントとして設定することによって、日本語表示が正しく行われるようにしている。

4.4.7　日本語入力の設定

Linux は、Windows とは違い、日本語入力するためにはパッケージの追加及び設定が必要である。そのため、日本語入力をするために、必要なパッケージをインストールする。今回は日本語入力環境として Fcitx5+Mozc を使用する。Fcitx5 はインプットメソッドフレームワークであり、Mozc は変換エンジンである。

Fcitx5 を使用するために、以下の環境変数を設定する。

GTK_IM_MODULE=fcitx

QT_IM_MODULE=fcitx

XMODIFIERS=@im=fcitx

また、/etc/xdg/autostart 下にデスクトップエントリを作成し、起動時に自動的に起動するように設定する。この作業により、日本語入力が可能となった。

また、上記の環境変数を/etc/environment に書き込むことで、設定を永続化する。

4.4.8　Docker の設定

設定を済ませていない状態での Docker では、GPU を使うことが出来ないため、設定する。

Docker 内で NVIDIA 製の GPU を使用するために、NVIDIA から NVIDIA Container Toolkit というツールが配布されている。NVIDIA Container Toolkit とは、GPU 処理を必要とするアプリケーションを Docker 内で実行するために必須となるツールである。

これをインストールするために、NVIDIA Container Toolkit の公式ドキュメントを参考にして、

live-build のフックスプリクトを作成する。

また、Docker が NVIDIA 製の GPU を使用することが出来るように、デフォルトランタイムを nvidia に設定する。

加えて、元々の環境では設定されていなかった、Docker を非ルートユーザーでも使用できるようにする設定をした。

公式のドキュメントによると、ユーザーを docker グループに所属させることで非ルートユーザーでも Docker を使用することが可能である。

そのため、インストーラーの設定で、インストール時に作成するユーザーを自動的に docker グループに所属させることでこれを設定した。

第 5 章　アプリケーションの作成

5.1 作成理由とその目的

学内ストレージである P ドライブや S ドライブへの接続を GUI で行えるアプリを作成した。

これを用いることで、手軽に P ドライブ・S ドライブへの接続が可能になる。

Linux からこれらのドライブにアクセスするためには手間がかかるため、その手間を削減することを目的としている。

図 5.1 作成したアプリケーションのレイアウト

5.2 アプリケーションの開発環境

このアプリケーションの作成を始めた当初は、GUI 部分を Rust、処理部分をシェルスクリプトで開発していた。しかし、Debian のリポジトリに存在する Rust のバージョンが古く、依存関係の問題が生じたため、途中から使用する言語を Python に切り替えている。

Rust と Python には処理速度に大きな差があると言われているが、起動速度などを比較した際に、特に差がなかったため、Python を採用した。

また、当初は Rust とシェルスクリプトを合わせて使用することで、Rust の特徴である高い学習難

易度を埋めようとしていたが、Python に移行にするにあたってその必要は無くなった。

そのため、Python に移行した当初は処理部分をシェルスクリプトのままで開発を進めていたが、最終的には Python のみに変更した。

最終的なアプリケーション開発環境は表 5.1 のようになっている。なお、ハードウェア構成は表 3.1 と同様であるため、省略する。

表 5.1 アプリケーション開発用 PC の環境

言語	Python
エディタ	Vim
ライブラリ	python3-gi gir1.2-gtk3.0
GUI ビルダ	Glade Interface Designer

GUI 部分には GTK を使用する。GTK は Linux で幅広く使用されている GUI ツールキットであり、Linux 向けの多くのソフトウェアで使用されている。

GUI ビルダの Glade Interface Designer は、GTK を用いたアプリケーションの GUI を視覚的に定義することが出来るアプリケーションである。また、定義した配置を XML ファイルとして生成する。

そのファイルをプログラムから読み込むことで、Glade Interface Designer で作成したレイアウトをアプリケーションに反映させることが出来る。

このファイルはサポートされているプログラミング言語であれば構文などの変更は無いため、Rust から Python に移行した際にもスムーズに移行することが出来た。

図 5.2 は、今回使用した Glade Interface Designer の編集画面である。

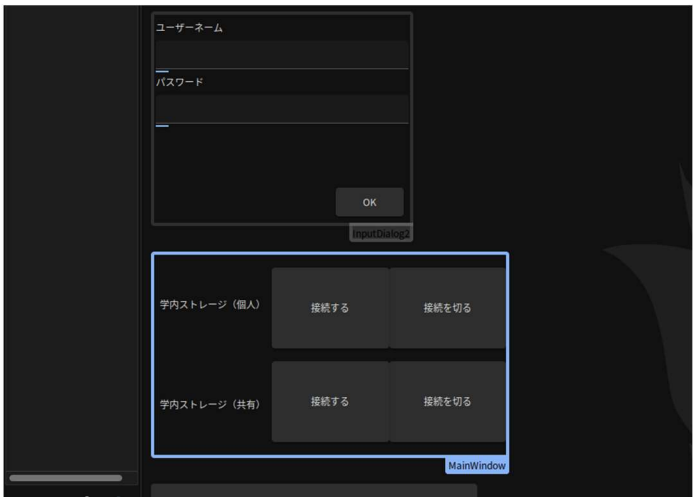

図 5.2 Glade Interface Designer の編集画面

5.3 アプリケーションの構造

図 5.3 が、今回作成したアプリケーションの構造だ。

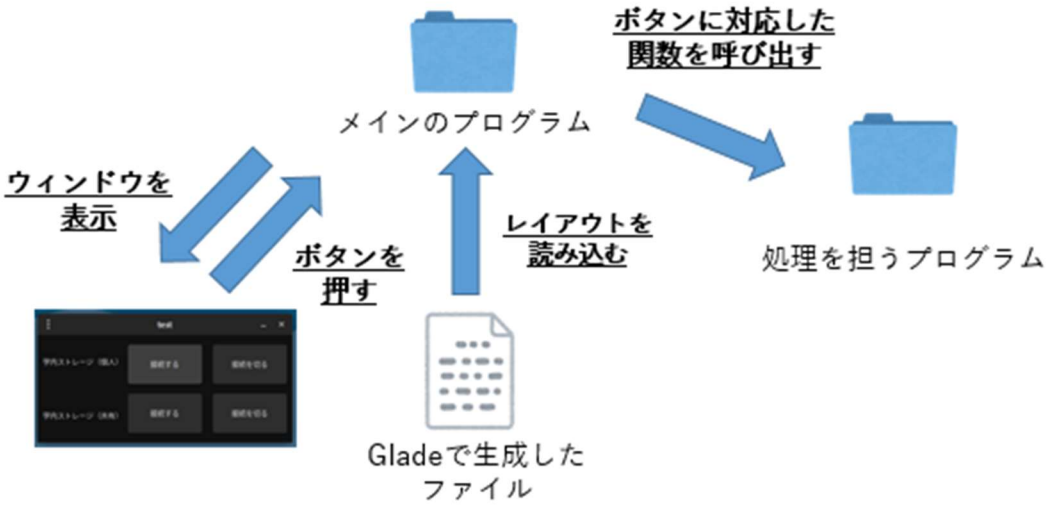

図 5.3 アプリケーションの構造

メインのプログラムから、Glade Interface Designer で作成したレイアウトを定義したファイルを読み込み、そこで定義されたレイアウトでウィンドウを表示する。

そして、ウィンドウの中にあるボタンが押されると、メインのプログラムは処理部分のプログラムを呼び出し、実行する。その際に使用されるユーザーネームとパスワードは、メインのプログラムで入力フォームを表示させ、入力された情報を使用している。

このアプリケーションの開発を始めた当初は、学内ストレージの接続の他にも機能を追加しようと考えていたが、レイアウトが複雑になってしまうこと、また、それに見合うメリットがあまり感じられなかったため、途中から学内ストレージの接続のみに機能を絞って開発をしている。

5.4 作成したプログラムの説明

ここからは、作成したプログラムの説明をする。

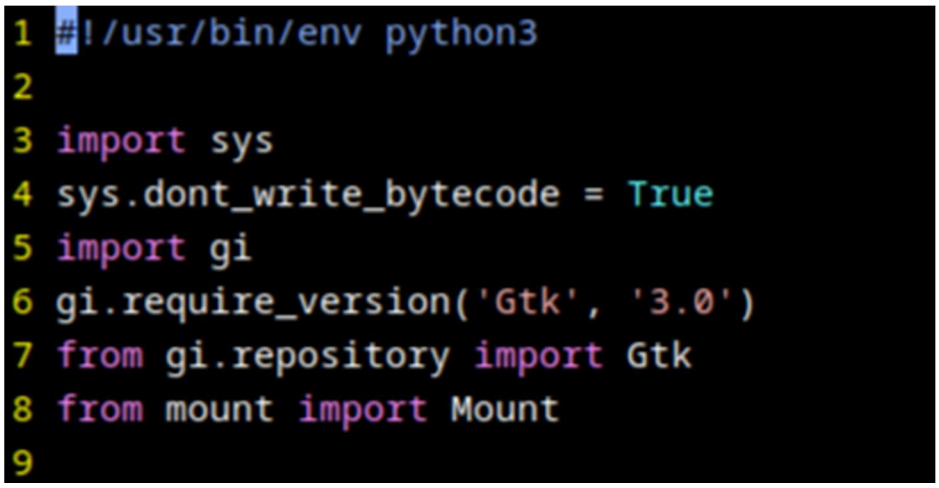

図 5.4 ライブラリのインポート部 (main.py)

図 5.4 は、ライブラリのインポートを行う部分だ。

図の序盤に記述している sys.dont_write_bytecode により、この Python ファイルが置かれているディレクトリに__pycache__というディレクトリを作成しないようにしている。

その下部でインポートしている gi は、プログラム中で GTK を使用するためのライブラリである。

また、最後にインポートしている mount は、mount.py から Mount クラスをインポートしている。

この mount.py は、先ほどの図 5.3 にある、マウント・アンマウントの処理を担うプログラムである。

```
43
44      def dokbtn_1_clicked(button):
45          name = dname_1.get_text()
46          passw = dpass_1.get_text()
47          mkdir_return_code, mkdir_message = Mount.personal_mkdir()
48
49          if mkdir_return_code == 0:
50              return_code, message = Mount.personal_mount(name, passw)
51              label = msgdialog1.get_message_area().get_children()[0]
52
53              if return_code == 0:
54                  label.set_text("個人用の学内ストレージを/mnt/personal_driveにマウントしました")
55              elif return_code == 1:
56                  label.set_text("既にストレージがマウントされています")
57              elif return_code == 2:
58                  label.set_text("ユーザーネームもしくはパスワードが間違っています")
59          else:
60              label = msgdialog1.get_message_area().get_children()[0]
61              label.set_text(mkdir_message)
62
```

図 5.5 ログイン画面で OK を押したときの処理（main.py）

そして、図 6.5 が学内ストレージ（個人用）のユーザーネームとパスワードを取得する部分である。

図 5.6 が実際のログイン画面であり、その OK ボタンを押したときにこの処理が開始される。

OK を押すことで、入力されたユーザーネームとパスワードを使い、学内ストレージをマウントするための処理を mount.py から呼び出す。

そして、返り値によって、その後に表示されるメッセージを分岐させている。返り値が 0 だった場合は正常な、その他の場合は返り値に応じたメッセージが表示される。

また、OK を押した際にマウントする先のディレクトリが存在しない場合には、自動でディレクトリを作成する処理が動くようになっている。

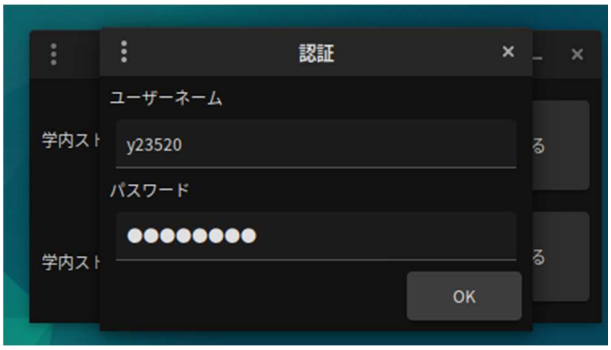

図 5.6 実際のログイン画面

```
33      @staticmethod
34      def personal_mount(name, password):
35          if Mount.is_mounted("/mnt/personal_drive"):
36              return 1, "drive already mounted"
37
38          result = run_command([
39              "mount", "-t", "cifs",
40              f"//storage.iit.iwate-it.ac.jp/home$/student/{name}",
41              "/mnt/personal_drive",
42              "-o", f"username={name},password={password},file_mode=0777,dir_mode
    =0777,iocharset=utf8,x-gvfs-show"
43          ])
44          if result.returncode != 0:
45              return 2, "can't drive mount"
46          return 0, "drive mounted"
```

図 5.7 personal_mount のプログラム（mount.py）

図 5.7 が、先ほど説明したログイン画面で OK を押した際に実行されるプログラムである。

このプログラムでは、個人用の学内ストレージをマウントする処理を行う。

最初の if 文で個人用の学内ストレージがマウントされているかを確認し、すでにマウントされている場合は、返り値 1 を返す。

その場合には、「既にストレージがマウントされています」といったメッセージが表示されるようになっている。

次に、マウント処理を行う。

ここでは、ログイン画面で入力したログイン情報を使用して、学内ストレージの接続を行う。接続に使われるネットワーク上のパスには、storage.iit.iwate-it.ac.jp を使用することで、ストレージサーバ

ーの IP アドレスが変更されても接続が可能となっている。

次に行われるのは、マウントが正常に行われたかを確認する処理だ。

ここでは、正常に終了した場合には 0 を返し、なんらかの異常が発生した場合には 2 を返すようになっている。また、正常に終了した場合には「個人用の学内ストレージを/mnt/personal_drive にマウントしました」というメッセージを表示し、なんらかの異常が発生した場合には「ユーザーネームもしくはパスワードが間違っています」というメッセージを表示する。

もしユーザーネームやパスワードが間違っておらず、他の原因でマウントが失敗した場合にも「ユーザーネームもしくはパスワードが間違っています」というメッセージが表示される。

しかし、このプログラムの動作確認をした限りでは、ユーザーネームもしくはパスワードが間違っている場合のみエラー処理となったため、このようなメッセージとなっている。

説明した個人用の学内ストレージ以外の処理も同じような流れで記述しているため、省略する。

5.5 Live イメージへのインストール

先ほど作成したアプリケーションをライブイメージへインストールする。

具体的には、アプリケーションを deb パッケージ化し、live-build の機能を用いてライブイメージにインストールした。deb パッケージとは、Debian 系の Linux ディストリビューションで利用されるソフトウェアのインストールファイルである。

第 6 章　動作確認

6.1 不具合の発見

ここまでの作業で大体の部分が完成したため、実際に AI 専用 PC にインストールをして動作を確認した。

その結果として、ライブ環境での動作及びインストールまでは上手く行ったが、インストール後の環境で GUI 環境が起動しないという不具合が発生した。

そのため、原因の特定と、不具合の修正を試みた。

6.2 不具合の一時的な修正

不具合を修正するため、まずは原因を特定する。

最初に、システムが NVIDIA ドライバを認識できているかを確認した。その際に使用したコマンドは、nvidia-smi である。

しかし、この出力は図 6.1 と同じように正常だったため、これは原因ではないと判断した。

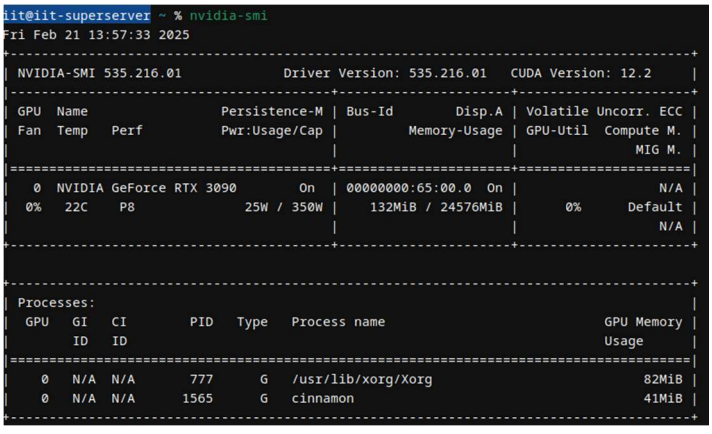

図 6.1 正常な nvidia-smi の出力

その後、実際にエラーが発生していたディスプレイマネージャーのログを確認した。

ディスプレイマネージャーとは、起動時にログイン画面を表示するソフトウェアのことである。

しかし、そのログからは起動に失敗したという情報しか得られず、手がかりとなるような情報を得ることは出来なかった。

そこで、GUI そのものを描画する X Window System (X11)のログを確認した。ログを読んだところ、X11 が起動する際に NVIDIA のドライバを正しく読み込めておらず、エラーが発生していた。

そのため、これが原因だと考え、X11 が起動する際にドライバを読み込む先に、NVIDIA ドライバのシンボリックリンクを作成した。

この処置によって、GUI が起動するようになった。

この時点では、ドライバがライブ環境で正常に動作し、インストール後の環境で動作しなかった理由が不明だったが、その後の動作確認によってインストール後の環境でも動作するようになった。

6.3 不具合の根本的な原因

まず、ドライバが入っていない状態の Debian に NVIDIA ドライバをインストールした場合は、正常に使用できる、という前提の元で原因を探った。

そのため、先ほど作成したシンボリックリンクを削除し、一度パッケージ名に nvidia が入っているパッケージをすべて削除することで、前述した Debian に NVIDIA ドライバがインストールされていない状態にした。

その後、NVIDIA ドライバをインストールし、再起動したところ、シンボリックリンクを作成せずとも GUI が動作した。

そのため、この状態を live-build 上で再現することで、インストール後の環境でも GUI が動作すると考えた。

live-build にパッケージをインストールする際には、通常、以下のディレクトリにインストールするパッケージ名を記述したファイルを置く必要がある。なお、ここで用いる workdir は作業ディレクトリのことである。

/workdir/config/package-lists/

このファイルに記述されたパッケージは、live-build の chroot 段階でインストールされる。

live-build には以下の段階が存在し、上から順に実行される。

lb bootstrap(1)

executes the first build stage, creating (bootstraping) a basic Debian root filesystem

lb chroot(1)

executes the second build stage, building the live OS filesystem

lb installer(1)

executes the third build stage, obtaining installer components (optional)

lb binary(1)

executes the fourth build stage, generating the binary (live) image

lb source(1)

executes the fifth build stage, generating a corresponding source image (optional)

(https://manpages.debian.org/testing/live-build/live-build.7.ja.html より引用)

つまり、今回の場合は、上から 2 番目の chroot 段階でインストールされるパッケージを指定している。

今後の説明のためにまず、chroot について説明する。IPA(独立行政法人 情報処理推進機構)の説明によると、

chroot は、Unix 系および GNU/Linux 系のシステムで利用できるシステムコール、および同名のコマンドである。これらを使うとプロセスから見えるファイルシステムのルートディレクトリの位置を特定のサブディレクトリに変更できる。

(出典：https://www.ipa.go.jp/archive/security/vuln/programming/cc/chapter5/cc5-3.html)

live-build は、この技術を使用して、実行したシステムに影響を与えずにライブイメージを構築することが出来る。

このように、AI 専用 PC にインストールした後の環境と比べると、live-build で使用される chroot の環境は通常の環境と比べて特殊なため、NVIDIA ドライバのインストールが正常に行われていなかったことが原因だと考えられる。

そのため、AI 専用 PC にインストールした後の環境で NVIDIA に関連するパッケージをすべて消した後に、再度 NVIDIA ドライバのインストールを行うと、正常に動作するようになった。

6.4 不具合の原因の修正

live-build には、先に示した段階の最終段階を終え、chroot 環境を抜けた後に、完成間近の環境でシェルスクリプトを実行する機能がある。

そこでパッケージをインストールすることで、chroot 環境よりもライブ環境及びインストール後の環境に近い状態でパッケージのインストールを行うことが出来る。

そのため、その環境で実行されるフックスプリクトを用意し、その中で NVIDIA 関係のパッケージの消去及び NVIDIA ドライバのインストールを行った。

また、先述の修正を反映した Live インストールイメージを、仮想環境上で動作確認した。

その際に、手動で NVIDIA ドライバのシンボリックリンクを作成せずとも、X11 がドライバを読み込む先に NVIDIA ドライバのシンボリックリンクが存在することを確認した。

そのため、その状態で AI 専用 PC へのインストールを行い、再度動作確認を行った。

結果、設定をせずとも NVIDIA ドライバが動作し、正常に GUI 環境が立ち上がった。

GUI 環境が動作したため、Docker 内での GPU 処理について、動作確認を行った。元々の環境では、Docker 内から GPU を利用することが可能な設定になっており、Docker 内での GPU を用いた AI 処理が出来るようになっていた。

6.5 Docker 内での nvidia-smi を用いた動作確認

まず、Docker が GPU を認識しているかを確認するために、**エラー! 参照元が見つかりません。**で使用した nvidia-smi を Docker 内で実行し、出力を確かめる。

結果は図 6.2 のようになっており、正常に認識していた。

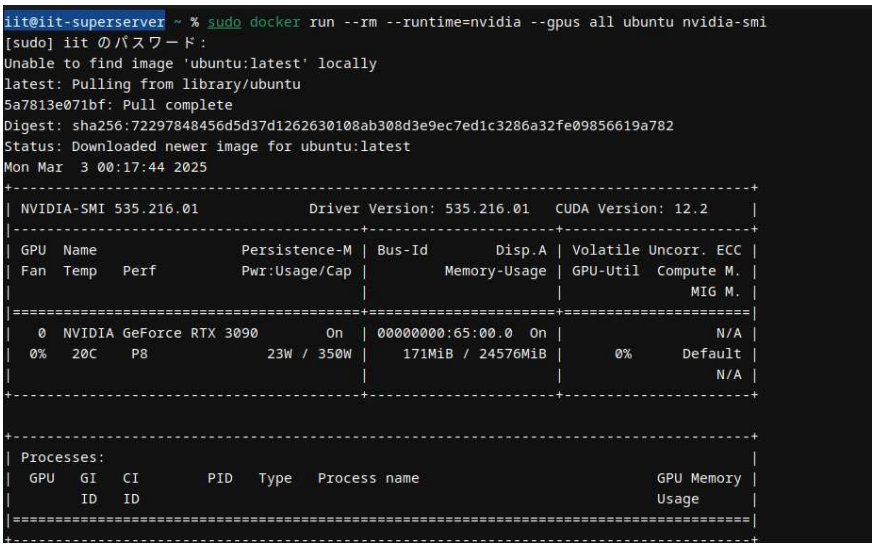

図 6.2 nvidia-smi の実行結果（Docker 内）

6.6 cuda-sample:nbody を用いた動作確認

Docker の cuda-samples:nbody は、NVIDIA が提供する CUDA のサンプルイメージの一つで、N 体シミュレーションを実行するプログラムを含んでいる。

N 体問題とは、互いに重力などの相互作用をする複数の物体の運動を扱う問題である。例えば、太陽系の惑星の運動や、銀河内の星の動きなどが N 体問題として扱われる。

この cuda-samples:nbody イメージは、CUDA 環境が正しくセットアップされ、GPU が Docker コンテナ内で正常に認識・利用できるかを確認するための動作確認ツールとして利用できる。

各物体は他の全ての物体から力を受けるため、N 体シミュレーションは多数の粒子間の相互作用を計算するもので、GPU の並列処理能力を効果的に活用する。

cuda-samples:nbody を実行することで、CUDA Toolkit、NVIDIA ドライバ、Docker の NVIDIA コンテナランタイム（nvidia-container-toolkit）が連携して動作しているかを確認でき、GPU を用いた計算処理が期待通りに行える状態であるかを検証できる。

結果は図 6.3 のようになっており、正常に動作していた。

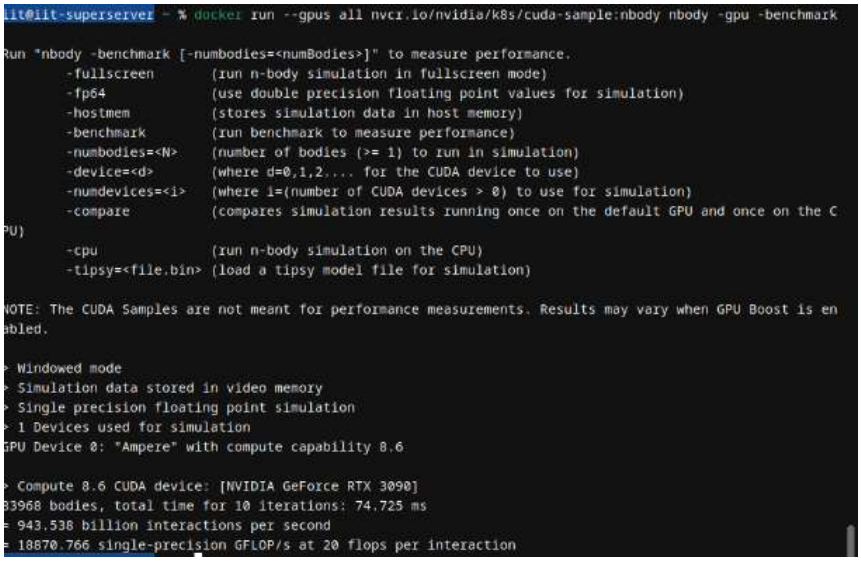

図 6.3 cuda-samples:nbody の実行結果

エラー! 参照元が見つかりません。で不具合を修正する前には、この実行結果が表示されず、GPU が見つからないといった趣旨のエラーメッセージが表示されていた。

6.7 Ollama+Llama 2 を用いた動作確認

Ollama は、大規模言語モデル（LLM）をローカル環境で容易に実行・運用するためのソフトウェアである。

Web UI や API 経由で、多様な言語モデルをダウンロードし、対話形式での利用や、文章生成などのタスクを実行させることが可能である。

今回は、ネット上の記事を参考に、Llama 2 を動作させた。

今回使用する Llama 2 とは、Meta 社が開発したオープンソース大規模言語モデル（LLM）である。

今回ダウンロードした 7B（70 億）パラメータモデルは、Llama 2 の中でも比較的小型であり、性能とリソース消費のバランスに優れている。

そのため、GPU メモリの少ない PC や、計算資源が限られた環境でも動作させやすいという特徴を持つ。

より高性能な 70B パラメータモデルも存在するが、7B モデルは、手軽に Llama 2 の機能を試したい場合や、リソースの制約がある場合に適している。

今回は動作確認として用いるため、7B モデルをダウンロードした。

以下の画像が、Docker+Ollama の環境で Llama 2 が動作している様子である。

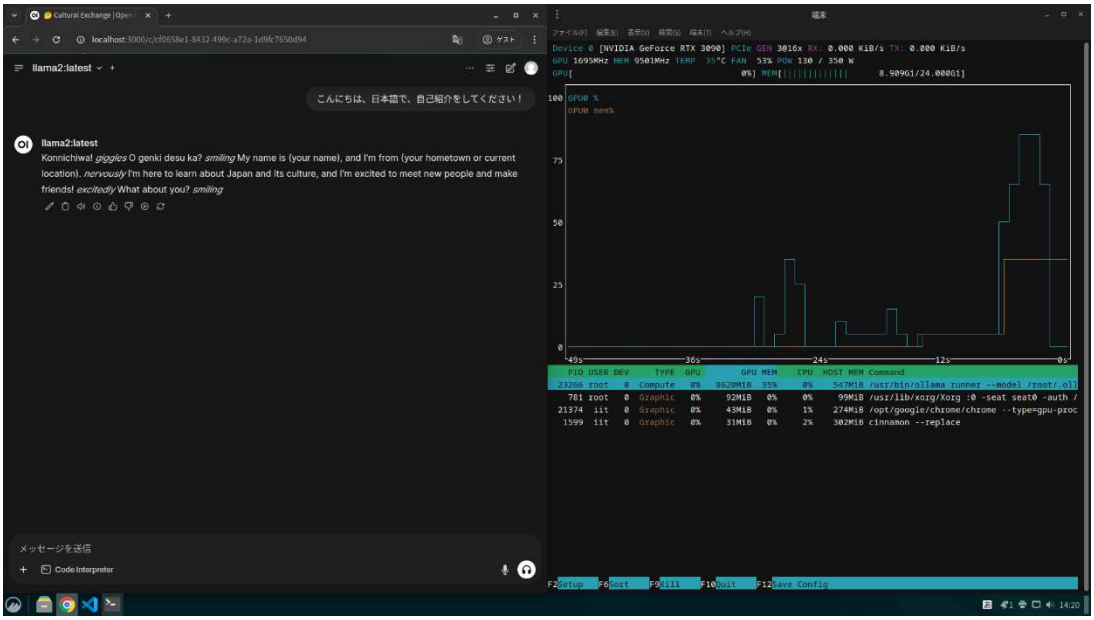

図 6.4 Docker + Ollama の環境で動作する Llama2

図 6.4 の右側に写っているのは、nvtop というソフトウェアである。

nvtop は、NVIDIA 製 GPU の使用状況をリアルタイムで監視するためのツールである。

図 8.3 に写っているように、プロセスごとの GPU 使用率、メモリ使用量、温度などを、ターミナル上にグラフィカルに表示する。

今回の場合は、Llama2 が回答を生成する際に GPU の使用率が上がっていることから、正常に

GPU を使用して AI の処理を行っていることが分かる。

エラー! 参照元が見つかりません。で不具合を修正する前は、GPU がうまく認識しておらず、CPU を使用して AI の処理を行ってしまっていた。

6.8 Ollama+DeepSeek-R1 を用いた動作確認

先ほど用いた Ollama を再度使用し、別のモデルを使用して動作確認を行う。今回は、DeepSeek-R1 を使用する。

DeepSeek-R1 とは、DeepSeek のフラッグシップ推論モデルだ。

数学、コーディング、推論の複数のベンチマークで、OpenAI の定評ある o1 モデルと同等か、それ以上の性能を発揮する。

R1 に関して特に興味深いのは、テクノロジー大手の他の上位モデルと異なり、オープンソースであるため、誰でもダウンロードして使用できることだ。

(https://japan.zdnet.com/article/35228851/より引用)

DeepSeep-R1 にも、Llama2 と同じように複数のパラメータモデルが存在する。

Ollama には 1.5B、7B、8B、14B、32B、70B、671B の蒸留モデルがラインナップされているが、今回はより負荷の高いテストをしたかったため、先ほど動作確認で用いた Llama2 の 7B モデルよりもパラメータ数の多い 32B のモデルをダウンロードした。

蒸留モデルとは、大きなモデルの知識を小さなモデルに転移させることで、モデルの性能を大きく損なわずに、要求するスペックを削減する手法だ。

図 6.5 が、Ollama+DeepSeek-R1(32B)を動作させている様子である。

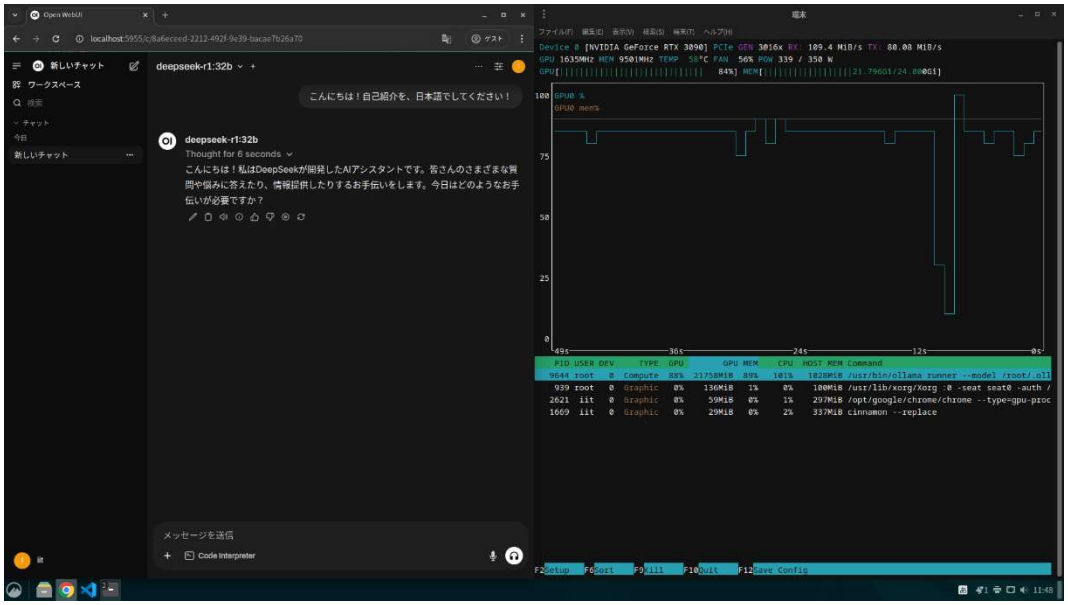

図 6.5 Docker+Ollama の環境で動作する DeepSeek-R1(32B)

図 6.5 から、先に検証した Llama2 に比べて、GPU 使用率と GPU メモリの使用率が高いことが分かる。また、回答も Llama2 に比べて精度の高いものになっている。

第 7 章　成果物

7.1 live-build の設定ファイル

今回の Live インストールイメージの作成で使用した live-build の設定ファイルを用いることで、新たにカスタマイズした Live インストールイメージの作成や、より新しいバージョンの Debian を用いた Live インストールイメージの作成が可能である。

しかし、今回作成した Debian 12 には存在するが、Debian 13 には存在しないパッケージや、バージョン間でパッケージ名が異なるパッケージも存在するため、おそらくそのままの設定では動作しない。

また、NVIDIA に関連するパッケージを消去することで、通常のパソコン向けにビルドすることも可能だ。

例として、今回使用したライブイメージ開発用 PC にインストールした Debian 12 は、Windows の WSL 上で作業していた際に作成した Live インストールイメージを用いて、インストールしたものである。

7.2 Docker を用いた live-build

Docker を用いることで、今回開発に使用した Debian 12 以外の環境でも Live インストールイメージのビルドが可能になっている。

今回、Docker について学ぶ過程で、Docker 内で live-build を実行し、iso ファイルを作成するものを作成した。

7.3 第 5 章で作成したアプリケーション

Live インストールイメージ用に作成したため、付属しているシェルスクリプトは Debian を前提としたものとなっているが、今回作成したプログラム自体は他のディストリビューションでも使用することが出来る。

7.4 Live インストールイメージの本体

今回作成した Live インストールイメージを、USB メモリに保存している。AI 専用 PC からブート順を指定し、Live インストールイメージが入った USB から起動することで、今回作成した Live インストールイメージを起動させることが出来る。

第 8 章　おわりに

本研究を通して、当初の予定であった

・設定をせずとも NVIDIA 製のグラフィックボードが動作する

・再インストールが容易

を満たすものを作成することが出来た。

途中で発生した不具合についても、把握している限りはほとんど解決することが出来た。

また、今回の教訓として、使用するツールやソフトウェアについて詳しく把握しておくことが大事だと改めて実感した。

不具合の修正で時間がかかった部分として、NVIDIA ドライバのインストール部分が一番に挙げられるが、その不具合も live-build の仕様や内部で行われる処理、chroot についてよく知っておけばすぐに解決できたかもしれない。

加えて、アプリケーションの作成でも、事前にライブラリの互換性などを調べておけば、使用言語の変更を回避できた可能性がある。

そのため、今後開発を行う際には、事前に使用するツールやソフトウェアについて詳しく調べていきたい。

また、今回の成果が、将来的に AI 専用 PC を用いた研究に貢献することを期待する。

第 9 章　参考文献

[1] Debian -- Live インストールイメージ

https://www.debian.org/CD/live/index.ja.html

[2] Debian Live Manual

https://live-team.pages.debian.net/live-manual/html/live-manual/about-manual.ja.html

[3] NvidiaGraphicsDrivers - Debian Wiki

https://wiki.debian.org/NvidiaGraphicsDrivers

[4] Docker Engine インストール（Debian 向け）

https://docs.docker.jp/desktop/install/debian.html

[5] シークレット ブラウジングを許可する - Chrome Enterprise and Education ヘルプ

https://support.google.com/chrome/a/answer/9302896?hl=JA

[6] Installing the NVIDIA Container Toolkit

https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html

[7] Post-installation steps | Docker Docs

https://docs.docker.com/engine/install/linux-postinstall/

[7] Glade インターフェース・デザイナー マニュアル

https://help.gnome.org/users/glade/stable/index.html.ja

[8] Python + gtk + glade で GUI アプリケーションの作成　その 1 - Symfoware

https://symfoware.blog.fc2.com/blog-entry-894.html

[8] Running a Sample Workload — NVIDIA Container Toolkit

https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/sample-workload.html

[9] Home | Open WebUI

https://docs.openwebui.com/

[10] Ollama is now available as an official Docker image · Ollama Blog

https://ollama.com/blog/ollama-is-now-available-as-an-official-docker-image

[11] WSL2 と Docker で Windows 上に Ollama を構築する

https://zenn.dev/toki_mwc/articles/d1ebbd634ff488

[12] Docker を使って Ollama と Open WebUI で llama3 を動かす

https://zenn.dev/misora/articles/1037a94c53a5f0