Final Project Report

110020007 施淙綸

Note: Currently, this project involves numerous intricate details in backend setup and overall parameter configuration, and it has not yet been deployed to a stable Cloud Computing Platform. Therefore, if you are interested in running or trying it out, please feel free to contact me for assistance with setup issues.

Complete Application Workflow

As described in the Proposal, and even exceeding the original schedule, the complete workflow has been integrated into the project. All processes are now seamlessly combined within a comprehensive application, allowing users to build their own envisioned storybook worlds, generating desired images and object sound effects.

Text-to-Prompt

This initial stage allows users to provide a high-level narrative or scene description. An intelligent model then processes this input to generate precise, detailed prompts optimized for Text-to-Image (T2I) models, ensuring that the generated image accurately reflects the user's vision.

Text-to-Image

Leveraging the refined prompts, a state-of-the-art T2I model generates the core visual scenes. This step transforms textual descriptions into rich, vivid images, forming the foundation of the user's interactive storybook world.

Image Mask to Text Description

Once the image is generated, users can interact with it by selecting specific regions or objects using an intuitive masking tool. A sophisticated model then analyzes these masked areas and provides detailed textual descriptions of the visual content within them.

Text Description to Sound Effect

The textual descriptions obtained from the masked image regions are further processed. Another AI model converts these visual descriptions into corresponding sound effect prompts. These prompts are then fed into a Text-to-Audio model, which generates realistic sound effects for the selected objects or areas, enhancing the immersion.

User Interface (Frontend)

The entire workflow is unified through a user-friendly frontend. This interface allows users to input their story ideas, view generated images, interact with them to select masks, and trigger the generation of accompanying sound effects, providing a complete and interactive experience.

Implementation

Step 1: Text-to-Prompt

User input for describing a scene may not always translate directly into the precise prompts required by Text-to-Image (T2I) models. To address this, an intermediary model is needed to assist users in generating optimal prompts, integrating seamlessly into the application's workflow. This part was relatively straightforward, primarily involving direct API integration with Google's services.

Step 2: Text-to-Image

Upon Google's announcement of Imagen 4, I identified it as the latest and most powerful T2I model, though currently inaccessible. Imagen 3's functionalities, however, require payment. To circumvent this, I leveraged the free $300 Google Cloud Platform trial to access Google's API, which proved to be a relatively straightforward integration.

Step 3: Image Mask to Text Description

Initial attempts involved direct local installation of Nvidia's GitHub source code for the image masking model. However, it quickly became apparent that local RAM was insufficient to run the model. This led me to explore various cloud platforms:

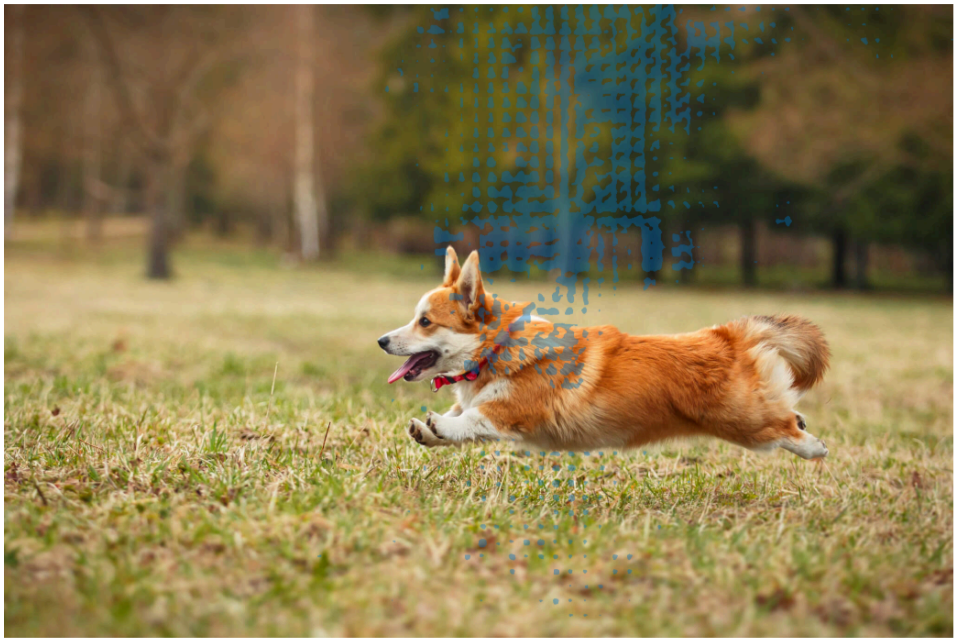
- **Hugging Face Space:** The free tier lacked GPU access, and CPU performance was insufficient for inference.
- **Google Colab:** While offering GPUs, the free tier's GPU performance was inadequate, and RAM was even less than local resources.
- **Google Cloud Platform:** GPU cloud computing proved prohibitively expensive.
- **Local:** The RAM limitation remained the primary challenge. I discovered that among `sam-vit-[huge, large, base]`, only the `huge` model provided accurate image embedding.
    - **Problem:** Even with the `huge` model, a `RuntimeError: Input type (float) and bias type (struct c10::Half) should be the same SAM` error persisted.
    - **Solution:** Convert input tensors to match the model's dtype (half precision): `python # Convert input tensors to the same dtype as the model (half precision) if hasattr(sam_model, "dtype"): inputs["pixel_values"] = inputs["pixel_values"].to(sam_model.dtype) else: # If dtype attribute not available, use half precision for quantized models inputs["pixel_values"] = inputs["pixel_values"].half()`
    - **Researching Image Embedding Model Sizes:**

Model Name	File Size	In-memory Size
`sam-vit-base`	375 MB	3.3 GB
`sam-vit-large`	1.25 GB	4.9 GB
`sam-vit-huge`	2.56 GB	6.3 GB

- **Reducing Model Size:** I explored quantization methods (specifically Bitsandbytes with `bnb_4bit_quant_type="nf4"`) available through Hugging Face's Transformers library, which significantly simplified the process.

Model Name	In-memory Size	Inference Time (sec)
`sam-vit-huge`	6.3 GB	17.743
`sam-vit-huge-quantization`	3.3 GB	2.925

- **Problem:** A challenge arose where using quantization halved the image size received by the frontend.
- **Solution:** Given the lower modification cost, the backend simply needed to cast the result to the correct type.

Model Name	Embedding Effect
sam-vit-base	
sam-vit-large	

Model Name	Embedding Effect
`sam-vit-huge`	
`sam-vit-huge-quantization`	

- **Kaggle:** Compared to Google Colab, Kaggle offered more powerful GPUs and larger RAM. Initially, I was concerned about Gradio's compatibility, but I later found that Gradio could run successfully, providing a 72-hour HTTPS backend connection. This made Kaggle a more viable solution than local execution.
- **Mid-way Decisions on Backend Frameworks:** I also considered using FastAPI instead of Gradio. However, Nvidia's provided frontend examples were designed to work with Gradio's Server-Sent Events (SSE) streaming. For convenience, and considering FastAPI's lack of direct full support for all SSE protocol specifics, I decided to stick with Gradio. Another benefit was Gradio's compatibility with Kaggle, including HTTPS connectivity, which eliminated the need to address IP connection issues for a FastAPI server on Kaggle.

Researching Local Execution (Cont.)

Embedding Inference Model (DAM)

After successfully reducing the SAM model's size and improving inference performance without significant quality degradation, the next step was to apply the quantization technique to the DAM embedding inference model.

- **Problem:** To use Hugging Face's quantization methods, the model must be a `transformers` model. However, DAM has custom inference methods that are not accessible via `AutoModel`. Not using `AutoModel` would, in turn, prevent the use of quantization.
- **Solution:** I discovered that custom `nn.Module`s in the source code passed `**kwargs` to `LlavaLlamaModel`, which *is* a `transformers` model. This allowed quantization via `dam = DescribeAnythingModel(model_path="./checkpoints/dam", quantization_config=bnb_config)`.

Model Name	In-memory Size	File Size
`DAM-3B`	7.0 GB	4.63 GB
`DAM-3B-quantization`	3.4 GB	4.63 GB

- **Problem:** When testing SAM for image embedding alone, inference time was 2.925 sec, but when simultaneously using DAM for image embedding, the time drastically increased to 81.731 sec.
- **Solution:** Due to time constraints and the plan to deploy the backend on Kaggle, I did not delve deeply into the exact cause. However, I suspect this is due to interactions between dedicated and shared GPU memory, leading to a significant drop in inference performance.

Step 4: Text Description to Sound Effect

Similar to Step 1, using LLM-generated image mask text descriptions directly as prompts for sound effects proved insufficient to accurately convey sound information. Therefore, an additional LLM layer was introduced to transform visual descriptions into auditory textual descriptions, integrated into the application workflow, and finally connected to a text-to-audio sound effect model.

I explored various models:

- **Google AudioLM:** Not yet available for public access in Taiwan.
- **Google MusicLM:** Not yet available for public access in Taiwan, and its use case is more geared towards music generation than sound effects.
- **Meta AudioGen:** An open-source model on Hugging Face, but after testing, it also leaned more towards music generation than sound effects.

- **AudioLDM 2:** An open-source model on Hugging Face with text-to-audio capabilities, including text-to-sound effect functionality according to its official description. After testing, it seemed to be the most suitable model. The experience with SAM and DAM models taught me to use quantization to effectively reduce model size and improve inference performance, a technique also applied here.
- **Problem:** `diffusers` ' current PyPI version did not include `PipelineQuantizationConfig`, requiring a direct download from the main branch.
- **Problem:** Errors like `Expected types for language_model: (<class 'transformers.models.gpt2.modeling_gpt2.GPT2LMHeadModel'>,), got <class 'transformers.models.gpt2.modeling_gpt2.GPT2Model'>.` and `'GPT2Model' object has no attribute '_get_initial_cache_position'` occurred.
- **Solution:** The `transformers` version used during AudioLDM 2's training differed from the latest version, causing a type mismatch. I manually modified the `__init__` method of `AudioLDM2Pipeline` to change the `language_model` parameter to `GPT2LMHeadModel`, resolving the issue.
- **Problem:** `'GPT2Config' object has no attribute 'max_new_tokens'` error.
- **Solution:** After changing `language_model` to `GPT2LMHeadModel`, the latest version of `GPT2LMHeadModel` no longer had a `max_new_tokens` parameter. I had to explicitly pass `max_new_tokens=8` during `pipe` inference to specify the length of the `language_model` 's generation.

After resolving these issues, I found that the model could not generate meaningful sound effects, producing only sharp noise. I suspected this was due to incompatibility between the modified `language_model` and the model originally used for training. Consequently, I began searching for newer open-source Text-to-Audio academic models.

Ultimately, I opted for TangoFlux, a Text-to-Audio model recently released and open-sourced this year. It offers better and faster performance than AudioLDM 2, with lower hardware requirements. Crucially, it did not encounter the unknown errors caused by API and implementation changes in the `transformers` package that plagued AudioLDM 2.

Step 5: Frontend

Initially, I intended to use Streamlit for the frontend. However, it proved unsuitable for integrating the entire workflow, as Streamlit lacks control over iframe object behavior and state. Therefore, I decided to implement the frontend using React.js, with Gradio serving as the backend API connector, leveraging Nvidia DAM model's frontend example.

Since I am not proficient in React.js, and mastering the framework would take more than a couple of months, I developed the frontend using GitHub Copilot in VSCode's agent mode. While "Vibe Coding" has recently gained popularity, I never thought I would need to use it in

practice. This project provided me with an opportunity to experience it firsthand, and I discovered both its advantages and drawbacks:

- **Advantages:**
  - Rapid generation of frontend code with instant visual feedback.
  - Enabling the completion of a surprisingly detailed frontend project in a short time, even without prior expertise.
  - Delegating tedious HTML configurations and object interactions to the LLM.
  - Allowing focus on application logic rather than technical minutiae.
- **Disadvantages:**
  - Generated code sometimes contains errors, and the model often struggles to identify the root cause, leading to prolonged debugging in unrelated areas when the solution might simply involve commenting out a few lines.
  - For a smooth development experience, a certain level of software design and development experience is crucial. Otherwise, it's easy to overlook potential issues in LLM-generated code, leading to project failure or flawed architectural design.
  - This approach offers no deep understanding of frontend technical details and principles for a professional learner. However, this drawback might become irrelevant once LLMs become truly intelligent.

Integrating the Backend

- **Problem:** SAM, DAM, and TangoFlux are all large model packages, each with distinct environmental requirements. In this project, we needed them to run within the same environment and script, necessitating a solution for environmental conflicts.
- **Solution:** The primary issue was TangoFlux's overly restrictive `requirements.txt`, which clashed with other models' dependencies. I found that some package versions could be upgraded to accommodate all models.
- **Problem:** The next challenge was how to modify TangoFlux's package versions while enabling arbitrary remote installation.
- **Solution:** I forked the TangoFlux GitHub repository here and modified its installation requirements. This allowed successful installation of all necessary model environments on Kaggle and any local machine.

Migrating Backend to Kaggle

- **Problem:** All three models require GPU for inference, but my local GPU's performance was insufficient to run them concurrently. Thus, the backend needed to be migrated to Kaggle. However, even after extensive model quantization, the P100 GPUs on Kaggle still presented RAM limitations.
- **Solution:** I set up a Kaggle notebook and installed the required environment. The RAM issue was not fully resolved by higher quantization methods (which would require too much source code modification within limited time). The eventual solution involved utilizing two

15GB T4 GPUs, dedicating one GPU to TangoFlux and the other to SAM and DAM models. This allowed all three models to run without affecting performance.

- **Problem:** Since all three models were initially configured with `device_map="auto"`, they automatically allocated to available GPUs. On Kaggle, with only two GPUs, many layers were consequently placed on different devices, leading to errors.
- **Solution:** I manually specified the GPU for each model during initialization (requiring modifications to some of the inference source code within each model).
- **Problem:** `ImportError: Using bitsandbytes 4-bit quantization requires the latest version of bitsandbytes: pip install -U bitsandbytes`
- **Solution:** Kaggle's Docker image had an incompatible built-in `transformers` version with `bitsandbytes`. Updating `transformers` with `!pip install -U transformers` resolved the issue.

Given more time, I would consider deploying the frontend and backend on Google Cloud Platform's Cloud Run, which could provide more stable and flexible connectivity.

Conclusion & What I Learned

This project provided me with invaluable experience in model integration. From initial model selection and environment configuration to subsequent model quantization, frontend development, and backend integration, I gained a deeper understanding of how to combine multiple large models into a single application. Although I already possess a solid foundation in software development, this project still taught me many new techniques and knowledge, particularly in model quantization and frontend development.

I also had the opportunity to experience "Vibe Coding" development. While it allowed for rapid completion of a conceptual product demonstration, it also confirmed my concerns: this approach is not optimal for learners, as it often fails to provide sufficient technical detail and underlying principles. It also functions best when one already has a certain level of software development experience.

Furthermore, this experience solidified my belief that software development often involves unexpected problems and challenges, frequently necessitating significant compromises. These issues demand flexible adaptation rather than sole reliance on theoretical knowledge or predefined plans.

During the recent Google I/O 2025 conference, Google showcased many cutting-edge AI models. While much of the content wasn't groundbreaking innovation, there was a palpable sense that models applied to vision and speech had reached new heights. This includes significant improvements in image generation quality and speed, as well as the realism of sound generation (though Google has not yet publicly accessed related products).

These technological advancements offer better and more stable support for integration in artistic creation, game development, and many other fields, allowing creators to focus more on

creativity and content. As exemplified by my project, these technologies enable individuals without painting skills, artistic understanding, or programming knowledge to easily create their own storybook worlds, providing enhanced immersion and interactivity. As the barrier to entry for applying these technologies continues to lower, it is conceivable that more people will participate in creative endeavors, producing more diverse and rich content, thereby, to some extent, achieving the democratization of creativity and technology.

I am truly excited to be participating in the next great era of global transformation and eagerly anticipate the unforeseen advancements these technologies will bring.

Appendix

Hardware & Software Specifications

Device Name	Specification
CPU	11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz
GPU	NVIDIA GeForce RTX 3050 Ti Laptop GPU
RAM	16 GB
SSD	512 GB + 1 TB
OS	Windows 11 Home Version 24H2 for x64-based Systems (26100.4061)
Python	3.12.9 (Local), 3.11.11 (Kaggle)