ggml(Georgi Gerganov's machine learning)

- ggml代表的是Georgi Gerganov's machine learning，是一个tensor library for machine learning

Roadmap / Manifesto

Tensor library for machine learning

Note that this project is under active development.
Some of the development is currently happening in the llama.cpp and whisper.cpp repos

Georgi Gerganov's machine learning（ggml）是一个专注于机器学习的张量库。张量库通常提供了进行数据表示和操作的基础设施，这些是实现各种机器学习算法的核心组件。在ggml的情况下，它可能包含了一系列用于数据处理、数学运算和网络构建的工具和函数。

从提供的链接来看，这个项目还在积极的开发中。这意味着其功能、接口和性能还在持续优化和扩展。项目的一部分开发正在其他相关的仓库中进行，如`llama.cpp`和`whisper.cpp`。这些仓库可能包含了实现或优化特定算法或模型的代码，或是提供了支持ggml发展的基础技术或实验。

- **llama.cpp**：虽然具体详情不详，但从名称推测，这可能是一个针对低级别机器学习运算优化的库。`cpp`后缀表明它使用C++编写，可能关注性能和资源效率。
- **whisper.cpp**：参考当前流行的Whisper模型，这个库可能与语音处理或自然语言处理有关，但具体细节需要查看其仓库了解。

Roadmap（路线图）和Manifesto（宣言）是了解项目未来发展方向和背后哲学的重要资源。路线图将展示预计实现的功能和优化，而宣言可能解释了创建ggml的原因、愿景以及它如何与现有的机器学习库区分开来。

了解这些信息有助于你把握该项目的发展动态，以及如何可能在你的机器学习或数据科学项目中利用它。

Features

- Written in C
- 16-bit float support
- Integer quantization support (4-bit, 5-bit, 8-bit, etc.)
- Automatic differentiation
- ADAM and L-BFGS optimizers
- Optimized for Apple Silicon
- On x86 architectures utilizes AVX / AVX2 intrinsics
- On ppc64 architectures utilizes VSX intrinsics
- No third-party dependencies
- Zero memory allocations during runtime

上述内容描述了ggml（Georgi Gerganov's machine learning）库的一些关键特性。每个特性都与该库的性能和应用场景密切相关。下面我将逐一解释每个特性及其相关概念：

1. **Written in C**：ggml 是用C语言编写的，这意味着它可能具有高效的执行速度和较低的运行时开销。C语言提供了接近硬件的操作能力，常用于需要高性能计算的应用。

2. **16-bit float support**：支持16位浮点数，这是一种精度较32位浮点数低的数据格式。16位浮点（通常称为半精度）在训练深度学习模型时可以减少内存使用和加速计算，尽管精度较低，但对于许多任务来说已足够。

3. **Integer quantization support (4-bit, 5-bit, 8-bit, etc.)**：整数量化支持，指的是将浮点数模型参数转换为整数（如4位、5位、8位等）。这种技术可以进一步减少模型大小和提高运算速度，常用于将模型部署到资源受限的设备上。

4. **Automatic differentiation**：自动微分是一种计算函数导数的技术，它是训练神经网络的核心。通过自动微分，可以有效地计算损失函数相对于模型参数的梯度，这对于使用梯度下降等优化算法至关重要。

5. **ADAM and L-BFGS optimizers**：ADAM是一种自适应学习率的优化算法，广泛用于深度学习中。L-BFGS是一种基于拟牛顿方法的优化器，常用于较小的数据集或需要非常精确的优化的场景。

6. **Optimized for Apple Silicon**：为苹果硅片（Apple Silicon，基于ARM架构的苹果自研处理器）优化，意味着ggml可以在这类设备上提供更高的性能。

7. **On x86 architectures utilizes AVX / AVX2 intrinsics**：在x86架构上使用AVX（高级向量扩展）和AVX2内在函数。这些是由现代Intel和AMD处理器支持的技术，可以提高浮点运算的速度，特别是在并行处理大量数据时。

8. **On ppc64 architectures utilizes VSX intrinsics**：在ppc64架构上使用VSX（向量标量扩展）内在函数，这是PowerPC架构处理器的向量化指令集，用于加速数据密集型应用。

9. **No third-party dependencies**：没有第三方依赖，意味着ggml是一个独立的库，不需要额外安装其他库或框架即可使用。这使得部署和维护变得更简单。

10. **Zero memory allocations during runtime**：在运行时零内存分配，表示所有必需的内存分配都在程序启动之前完成，运行时不进行额外分配。这有助于减少延迟和避免运行时内存碎片化，特别是在实时应用中非常有用。

这些特性表明ggml是一个为高性能和资源优化设计的机器学习库，适合在各种硬件上运行，尤其是在资源受限的环境中。

Updates

- ☑ Example of GPT-2 inference examples/gpt-2
- ☑ Example of GPT-J inference examples/gpt-j
- ☑ Example of Whisper inference ggerganov/whisper.cpp
- ☑ Example of LLaMA inference ggerganov/llama.cpp
- ☑ Example of LLaMA training ggerganov/llama.cpp/examples/baby-llama
- ☑ Example of Falcon inference cmp-nct/ggllm.cpp
- ☑ Example of BLOOM inference NouamaneTazi/bloomz.cpp
- ☑ Example of RWKV inference saharNooby/rwkv.cpp
- ☑ Example of SAM inference examples/sam
- ☑ Example of BERT inference skeskinen/bert.cpp
- ☑ Example of BioGPT inference PABannier/biogpt.cpp
- ☑ Example of Encodec inference PABannier/encodec.cpp
- ☑ Example of CLIP inference monatis/clip.cpp
- ☑ Example of MiniGPT4 inference Maknee/minigpt4.cpp

- ☑ Example of ChatGLM inference li-plus/chatglm.cpp
- ☑ Example of Stable Diffusion inference leejet/stable-diffusion.cpp
- ☑ Example of Qwen inference QwenLM/qwen.cpp
- ☑ Example of YOLO inference examples/yolo
- ☑ Example of ViT inference staghado/vit.cpp
- ☑ Example of multiple LLMs inference foldl/chatllm.cpp
- ☑ SeamlessM4T inference *(in development)*
  https://github.com/facebookresearch/seamless_communication/tree/main/ggml

上述内容列出了ggml库的更新日志，展示了该库支持的一些主要模型及其推理（inference）示例。这些示例展示了ggml在不同模型上的应用，涵盖了广泛的机器学习和深度学习任务。下面我将详细解释每个更新项及相关概念。

1. **Example of GPT-2 inference examples/gpt-2**：
   - **GPT-2**：生成预训练变压器2（Generative Pre-trained Transformer 2），是OpenAI开发的一种大型语言模型，擅长生成连贯的文本。
   - **Inference**：推理，即使用训练好的模型对新数据进行预测或生成结果。
   - **示例**：这个链接提供了如何在ggml中进行GPT-2推理的示例代码。
2. **Example of GPT-J inference examples/gpt-j**：
   - **GPT-J**：由EleutherAI开发的类似GPT-3的模型，具有6B参数，提供了强大的文本生成能力。
   - **Inference**：同上，使用训练好的GPT-J模型进行推理的示例代码。
3. **Example of Whisper inference ggerganov/whisper.cpp**：
   - **Whisper**：一个针对语音识别的模型，通常用于将语音转录为文本。
   - **示例**：在ggml中进行Whisper模型推理的示例代码。
4. **Example of LLaMA inference ggerganov/llama.cpp**：
   - **LLaMA**：一种用于语言建模的深度学习模型。
   - **示例**：LLaMA模型在ggml中的推理示例代码。
5. **Example of LLaMA training ggerganov/llama.cpp/examples/baby-llama**：
   - **Training**：训练，即通过调整模型参数使其能够更好地理解数据。
   - **示例**：LLaMA模型在ggml中的训练示例代码。
6. **Example of Falcon inference cmp-nct/ggllm.cpp**：
   - **Falcon**：可能是另一种语言模型或特定任务的深度学习模型。
   - **示例**：Falcon模型在ggml中的推理示例代码。
7. **Example of BLOOM inference NouamaneTazi/bloomz.cpp**：
   - **BLOOM**：一个大型的多语言模型。
   - **示例**：BLOOM模型在ggml中的推理示例代码。
8. **Example of RWKV inference saharNooby/rwkv.cpp**：

- **RWKV**：可能是一种特定的神经网络模型架构。
- **示例**：RWKV模型在ggml中的推理示例代码。

9. **Example of SAM inference examples/sam**：

- **SAM**：具体模型未详细说明，可能是用于特定任务的模型。
- **示例**：SAM模型在ggml中的推理示例代码。

10. **Example of BERT inference skeskinen/bert.cpp**：

- **BERT**：双向编码器表示的变换器（Bidirectional Encoder Representations from Transformers），用于自然语言理解任务。
- **示例**：BERT模型在ggml中的推理示例代码。

11. **Example of BioGPT inference PABannier/biogpt.cpp**：

- **BioGPT**：一种专注于生物医学文本的GPT模型。
- **示例**：BioGPT模型在ggml中的推理示例代码。

12. **Example of Encodec inference PABannier/encodec.cpp**：

- **Encodec**：可能是一种用于编码任务的模型。
- **示例**：Encodec模型在ggml中的推理示例代码。

13. **Example of CLIP inference monatis/clip.cpp**：

- **CLIP**：一种由OpenAI开发的模型，将图像和文本连接在一起进行多模态理解。
- **示例**：CLIP模型在ggml中的推理示例代码。

14. **Example of MiniGPT4 inference Maknee/minigpt4.cpp**：

- **MiniGPT4**：可能是GPT-4的一个小型版本，具有类似的文本生成能力。
- **示例**：MiniGPT4模型在ggml中的推理示例代码。

15. **Example of ChatGLM inference li-plus/chatglm.cpp**：

- **ChatGLM**：可能是一种用于对话生成的模型。
- **示例**：ChatGLM模型在ggml中的推理示例代码。

16. **Example of Stable Diffusion inference leejet/stable-diffusion.cpp**：

- **Stable Diffusion**：一种生成模型，常用于生成图像。
- **示例**：Stable Diffusion模型在ggml中的推理示例代码。

17. **Example of Qwen inference QwenLM/qwen.cpp**：

- **Qwen**：具体模型未详细说明，可能是用于特定任务的模型。
- **示例**：Qwen模型在ggml中的推理示例代码。

18. **Example of YOLO inference examples/yolo**：

- **YOLO**：You Only Look Once，一种实时目标检测模型。
- **示例**：YOLO模型在ggml中的推理示例代码。

19. **Example of ViT inference staghado/vit.cpp**：
    - **ViT**：视觉变压器（Vision Transformer），用于图像分类任务。
    - **示例**：ViT模型在ggml中的推理示例代码。
20. **Example of multiple LLMs inference foldl/chatllm.cpp**：
    - **Multiple LLMs**：多个大型语言模型的推理示例。
    - **示例**：多个LLM模型在ggml中的推理示例代码。
21. **SeamlessM4T inference *(in development)***
    https://github.com/facebookresearch/seamless_communication/tree/main/ggml：
    - **SeamlessM4T**：可能是Facebook Research开发的某个项目，目前仍在开发中。
    - **示例**：该模型的推理示例代码将在开发完成后提供。

这些示例展示了ggml库的广泛应用，涵盖了文本生成、自然语言理解、语音识别、图像生成和目标检测等多种任务。通过这些示例，用户可以了解如何在ggml中实现和应用这些模型。

Python environment setup and building the examples

```
git clone https://github.com/ggerganov/ggml
cd ggml
# Install python dependencies in a virtual environment
python3.10 -m venv ggml_env
source ./ggml_env/bin/activate
pip install -r requirements.txt
# Build the examples
mkdir build && cd build
cmake ..
cmake --build . --config Release -j 8
```

上述内容描述了如何在一个Python环境中设置并构建ggml库的示例代码。这一过程涉及从GitHub克隆仓库、设置Python虚拟环境、安装依赖和编译示例。我将一步一步解释每个命令及其作用：

1. **克隆仓库**：

   ```
   git clone https://github.com/ggerganov/ggml
   ```

   这个命令使用Git从GitHub上克隆（即复制）ggml的代码库到本地计算机。这样可以获取到最新的代码。

2. **进入项目目录**：

   ```
   cd ggml
   ```

   这个命令改变当前目录，进入克隆下来的ggml目录中。

3. **创建并激活Python虚拟环境：**

```
python3.10 -m venv ggml_env
source ./ggml_env/bin/activate
```

   - `python3.10 -m venv ggml_env`：使用Python 3.10的虚拟环境模块创建一个名为`ggml_env`的虚拟环境。虚拟环境是一个独立的Python运行环境，可以在其中安装与主系统隔离的Python包，避免版本冲突。
   - `source ./ggml_env/bin/activate`：激活虚拟环境，之后在该终端中运行的Python命令和安装的库都将限定在这个虚拟环境中。

4. **安装Python依赖：**

```
pip install -r requirements.txt
```

   这个命令使用pip（Python包管理器）安装`requirements.txt`文件中列出的所有依赖包。这个文件通常包含了项目运行所需的第三方库及其版本信息。

5. **创建并进入构建目录：**

```
mkdir build && cd build
```

   - `mkdir build`：创建一个新的目录叫做`build`，用于存放构建过程中生成的所有文件。
   - `cd build`：进入这个新创建的`build`目录。

6. **配置项目：**

```
cmake ..
```

   使用CMake工具对项目进行配置。这个命令告诉CMake读取父目录（`..`）中的CMake配置文件（`CMakeLists.txt`），并设置好编译环境和参数。

7. **编译项目：**

```
cmake --build . --config Release -j 8
```

   - `cmake --build .`：告诉CMake在当前目录（`.`）下开始构建过程。
   - `--config Release`：设置构建类型为`Release`，这意味着编译器将优化代码，生成用于生产环境的执行效率更高的二进制文件。
   - `-j 8`：这是一个并行构建的参数，`-j 8`意味着使用8个并行作业来编译代码，可以显著加快构建速度，特别是在多核处理器上。

这一系列命令提供了一个从代码获取到构建并运行ggml示例的完整流程，使用户能够在本地机器上测试和使用这些示例。

GPT inference (example)

With ggml you can efficiently run GPT-2 and GPT-J inference on the CPU.

Here is how to run the example programs:

```
# Run the GPT-2 small 117M model
../examples/gpt-2/download-ggml-model.sh 117M
./bin/gpt-2-backend -m models/gpt-2-117M/ggml-model.bin -p "This is an example"

# Run the GPT-J 6B model (requires 12GB disk space and 16GB CPU RAM)
../examples/gpt-j/download-ggml-model.sh 6B
./bin/gpt-j -m models/gpt-j-6B/ggml-model.bin -p "This is an example"

# Run the Cerebras-GPT 111M model
# Download from: https://huggingface.co/cerebras
python3 ../examples/gpt-2/convert-cerebras-to-ggml.py /path/to/Cerebras-GPT-111M/
./bin/gpt-2 -m /path/to/Cerebras-GPT-111M/ggml-model-f16.bin -p "This is an
example"
```

The inference speeds that I get for the different models on my 32GB MacBook M1 Pro are as follows:

Model	Size	Time / Token
GPT-2	117M	5 ms
GPT-2	345M	12 ms
GPT-2	774M	23 ms
GPT-2	1558M	42 ms
GPT-J	6B	125 ms

For more information, checkout the corresponding programs in the examples folder.

上述内容展示了如何使用ggml库在CPU上运行GPT-2和GPT-J等不同规模的生成预训练变换器（GPT）模型的推理（inference）。这些步骤允许用户在本地计算机上生成文本。以下是详细的解释和步骤：

运行GPT-2和GPT-J推理示例

1. **下载并运行GPT-2小模型（117M）**：

```
../examples/gpt-2/download-ggml-model.sh 117M
./bin/gpt-2-backend -m models/gpt-2-117M/ggml-model.bin -p "This is an
example"
```

- 首先通过脚本`download-ggml-model.sh`下载117M版本的GPT-2模型。
- 使用`gpt-2-backend`执行文件来加载模型并传入一个文本前缀（"This is an example"），程序将基于此前缀生成文本。

2. **下载并运行GPT-J大模型（6B）**：

```
../examples/gpt-j/download-ggml-model.sh 6B
./bin/gpt-j -m models/gpt-j-6B/ggml-model.bin -p "This is an example"
```

- 类似地，下载GPT-J的6B模型，需要12GB的磁盘空间和至少16GB的CPU内存来处理。
- 运行`gpt-j`执行文件，加载模型，并以同样的方式传入文本前缀来生成文本。

3. **运行Cerebras-GPT 111M模型**:

```
python3 ../examples/gpt-2/convert-cerebras-to-ggml.py /path/to/Cerebras-GPT-111M/
./bin/gpt-2 -m /path/to/Cerebras-GPT-111M/ggml-model-f16.bin -p "This is an example"
```

- 从Hugging Face下载Cerebras-GPT 111M模型。
- 使用Python脚本`convert-cerebras-to-ggml.py`将下载的模型转换为ggml能使用的格式。
- 使用`gpt-2`执行文件加载转换后的模型，传入文本前缀生成文本。

推理速度

表格展示了不同模型在MacBook M1 Pro（32GB内存）上每生成一个token（即文本单位，通常是一个词或字符）所需的时间：

- GPT-2的不同版本（117M, 345M, 774M, 1558M）的时间从5毫秒到42毫秒不等。
- GPT-J 6B模型每个token需要125毫秒。

这个推理时间可以帮助用户评估模型的性能，以及在不同硬件配置下模型的实用性。

更多信息

更多关于如何运行这些程序的细节可以在项目的examples文件夹中找到。这为用户提供了一个关于如何使用这些模型的具体实践指南，以及如何自定义和扩展这些示例以满足特定需求。

Using Metal (only with GPT-2)

For GPT-2 models, offloading to GPU is possible. Note that it will not improve inference performances but will reduce power consumption and free up the CPU for other tasks.

To enable GPU offloading on MacOS:

```
cmake -DGGML_METAL=ON -DBUILD_SHARED_LIBS=Off ..

# add -ngl 1
./bin/gpt-2 -t 4 -ngl 100 -m models/gpt-2-117M/ggml-model.bin -p "This is an
example"
```

上述内容介绍了如何在MacOS系统上使用Metal API将GPT-2模型的运算负载从CPU转移到GPU。这主要目的不是为了提高推理性能，而是为了降低电力消耗，并释放CPU资源以便执行其他任务。下面详细解释这些步骤和命令：

启用GPU Offloading（卸载至GPU）

1. **配置CMake以启用Metal**:

   ```
   cmake -DGGML_METAL=ON -DBUILD_SHARED_LIBS=Off ..
   ```

   - `cmake`：这是一个构建系统生成器，用于控制软件编译过程。
   - `-DGGML_METAL=ON`：这个选项告诉CMake在构建ggml时启用Metal支持。Metal是Apple开发的一个应用程序编程接口（API），用于直接与iOS和MacOS设备上的图形处理硬件（GPU）交互。
   - `-DBUILD_SHARED_LIBS=Off`：这个选项指定CMake构建静态库而非动态库。静态库在执行文件中直接包含，有助于减少依赖和简化部署。
   - `..`：指向包含`CMakeLists.txt`文件的目录，通常是项目的根目录。

2. **运行GPT-2模型并指定GPU参数**:

   ```
   ./bin/gpt-2 -t 4 -ngl 100 -m models/gpt-2-117M/ggml-model.bin -p "This is an
   example"
   ```

   - `./bin/gpt-2`：运行编译后的GPT-2执行文件。
   - `-t 4`：可能指定使用的线程数为4。这有助于控制并发执行的线程数，以优化性能。
   - `-ngl 100`：这个参数指定了GPU相关的某种设置（虽然没有明确解释，可能表示每次加载到GPU的数据量或批处理大小）。
   - `-m models/gpt-2-117M/ggml-model.bin`：指定模型文件的路径。
   - `-p "This is an example"`：提供给模型的输入文本，用作生成文本的起点或提示。

总结

通过使用Metal，可以有效地利用MacOS设备上的GPU资源来运行GPT-2模型，这样做可以降低CPU的负载和电力消耗，尤其适合那些需要在后台长时间运行或并行处理多任务的应用场景。这种配置对于开发高效能和资源优化的应用尤其重要。

Using cuBLAS

```
```

```
# fix the path to point to your CUDA compiler
cmake -DGGML_CUDA=ON -DCMAKE_CUDA_COMPILER=/usr/local/cuda-12.1/bin/nvcc ..
```

上述内容说明了如何配置和启用cuBLAS（一种基于CUDA的库，用于进行基础线性代数计算）以利用NVIDIA GPU进行计算加速，这是通过CMake在构建过程中设置CUDA相关选项来实现的。这里的步骤特别是为了使用NVIDIA的CUDA技术优化计算过程，具体操作如下：

配置CMake以启用CUDA和cuBLAS

1. **配置CMake命令**:

   ```
   cmake -DGGML_CUDA=ON -DCMAKE_CUDA_COMPILER=/usr/local/cuda-12.1/bin/nvcc ..
   ```

   - `cmake`：CMake是一个构建系统生成器，用于控制软件编译过程。
   - `-DGGML_CUDA=ON`：这个选项告诉CMake在构建ggml时启用CUDA支持。CUDA（Compute Unified Device Architecture）是NVIDIA推出的一个并行计算平台和应用编程接口模型，它允许软件利用GPU进行复杂的计算任务。
   - `-DCMAKE_CUDA_COMPILER`：这个参数指定CUDA编译器（nvcc）的路径。`nvcc`是NVIDIA CUDA编译器，用于编译CUDA代码。这里指定的路径是`/usr/local/cuda-12.1/bin/nvcc`，这意味着CUDA的版本是12.1，位于系统的`/usr/local/cuda-12.1`目录下。
   - `..`：指向包含`CMakeLists.txt`文件的目录，通常是项目的根目录。

为什么使用cuBLAS？

cuBLAS是NVIDIA提供的一个基于CUDA的库，专门用于进行高性能的线性代数计算。在机器学习和深度学习中，很多算法（尤其是大规模的矩阵运算和线性运算）可以通过这种库显著提高执行速度。启用cuBLAS可以让ggml利用NVIDIA GPU执行这些密集型的计算任务，从而提高性能，减少计算时间。

总结

通过配置CMake以使用CUDA和cuBLAS，可以使得ggml在支持CUDA的NVIDIA GPU上运行时得到性能优化。这种优化特别适用于需要处理大量数据和复杂模型的机器学习任务，能够有效地缩短训练和推理时间。这一配置步骤对于希望在支持CUDA的硬件上最大化其计算资源的用户来说非常重要。

Using hipBLAS

```
cmake -DCMAKE_C_COMPILER="$(hipconfig -l)/clang" -DCMAKE_CXX_COMPILER="$(hipconfig -l)/clang++" -DGGML_HIPBLAS=ON
```

上述内容展示了如何通过CMake配置使用hipBLAS，这是一种基于AMD HIP的库，用于在支持HIP的硬件（主要是AMD GPU）上进行高效的线性代数计算。HIP（Heterogeneous-compute Interface for Portability）是由AMD开发的一个库，旨在提供一个类似于CUDA的接口，使开发者能够编写可在NVIDIA和AMD GPU上运行的代码。以下是具体配置说明：

配置CMake以使用hipBLAS

1. **配置CMake命令**:

```
cmake -DCMAKE_C_COMPILER="$(hipconfig -l)/clang" -
DCMAKE_CXX_COMPILER="$(hipconfig -l)/clang++" -DGGML_HIPBLAS=ON
```

   - `cmake`：CMake是一个构建系统生成器，用于控制软件编译过程。
   - `-DCMAKE_C_COMPILER="$(hipconfig -l)/clang"` 和 `-DCMAKE_CXX_COMPILER="$(hipconfig -l)/clang++"`：这两个选项分别设置C语言和C++编译器。这里使用了`hipconfig -l`命令来动态获取HIP安装路径下的`clang`和`clang++`编译器的位置。`hipconfig`是HIP配置工具，用于查询HIP相关的配置信息，包括编译器路径。
   - `-DGGML_HIPBLAS=ON`：这个选项告诉CMake在构建ggml时启用hipBLAS支持。hipBLAS是AMD提供的针对其GPU优化的线性代数库，类似于NVIDIA的cuBLAS，用于加速线性代数计算。

为什么使用hipBLAS？

hipBLAS的目标是提供高性能的BLAS（基础线性代数子程序）实现，使得基于AMD GPU的系统能够在执行矩阵和向量运算时达到最优性能。在机器学习和深度学习中，许多算法涉及大量的矩阵运算，使用hipBLAS可以显著提升这些运算的效率，减少计算时间。

总结

通过配置CMake以使用hipBLAS，可以使得ggml在支持HIP的AMD GPU上运行时得到性能优化。这一步骤特别重要对于希望在AMD硬件上充分利用其GPU计算资源的用户，以及需要处理大规模数据和模型的机器学习项目。这种配置帮助实现了代码的跨平台兼容性和性能优化。

Using SYCL

```
# linux
source /opt/intel/oneapi/setvars.sh
cmake -G "Ninja" -DCMAKE_C_COMPILER=icx -DCMAKE_CXX_COMPILER=icpx -DGGML_SYCL=ON
..

# windows
"C:\Program Files (x86)\Intel\oneAPI\setvars.bat"
cmake -G "Ninja" -DCMAKE_C_COMPILER=cl -DCMAKE_CXX_COMPILER=icx -DGGML_SYCL=ON ..
```

上述内容描述了如何在Linux和Windows操作系统上配置和使用SYCL，这是一个为异构计算设计的高层次、跨平台的编程模型。SYCL允许代码运行在不同类型的处理器上，包括CPU、GPU和其他形式的加速器，是基于C++的开放标准。这里通过配置CMake来启用SYCL支持，以便利用这一技术优化并行计算性能。下面详细解释这些步骤：

Linux上的配置

1. **设置环境变量**:

```
source /opt/intel/oneapi/setvars.sh
```

- 这个命令通过源（source）操作执行Intel oneAPI工具集的环境配置脚本，`setvars.sh`，来初始化Intel oneAPI开发环境。此脚本设置必要的环境变量，以便能正确使用编译器和库。

2. **配置和生成项目**:

```
cmake -G "Ninja" -DCMAKE_C_COMPILER=icx -DCMAKE_CXX_COMPILER=icpx -
DGGML_SYCL=ON ..
```

- `cmake`：调用CMake来配置项目。
- `-G "Ninja"`：指定使用Ninja作为构建系统生成器。Ninja是一种小巧且高效的构建系统，比make更快。
- `-DCMAKE_C_COMPILER=icx` 和 `-DCMAKE_CXX_COMPILER=icpx`：分别指定C和C++的编译器为Intel的新一代编译器（Intel next-generation compiler）。
- `-DGGML_SYCL=ON`：启用SYCL支持。这允许ggml利用SYCL进行异构计算，以优化跨不同硬件的性能。
- `..`：指定CMake配置文件（CMakeLists.txt）所在的目录。

Windows上的配置

1. **设置环境变量**:

```
"C:\Program Files (x86)\Intel\oneAPI\setvars.bat"
```

- 在Windows上执行Intel oneAPI的环境配置脚本，`setvars.bat`，初始化开发环境。

2. **配置和生成项目**:

```
cmake -G "Ninja" -DCMAKE_C_COMPILER=cl -DCMAKE_CXX_COMPILER=icpx -
DGGML_SYCL=ON ..
```

- `-DCMAKE_C_COMPILER=cl`：指定C的编译器为Microsoft的CL编译器。
- 其他选项同Linux配置，都是为了使项目能够使用SYCL。

为什么使用SYCL？

SYCL提供了一个标准化的方式来编写代码，该代码可以无缝地在多种硬件平台上执行，包括CPU、GPU以及其他形式的加速器。使用SYCL能够帮助开发者编写出更为通用、可移植的高性能计算应用。通过这种方式，可以显著提高应用程序在不同设备上的运行效率，优化资源使用，增加应用的灵活性和可扩展性。

总结

这些步骤为开发者提供了如何在两种主要操作系统（Linux和Windows）上通过Intel oneAPI和CMake配置SYCL的方法，使得开发者能够开发出跨平台、高性能的应用程序。

Compiling for Android

Download and unzip the NDK from this download page. Set the NDK_ROOT_PATH environment variable or provide the absolute path to the CMAKE_ANDROID_NDK in the command below.

```
cmake .. \
    -DCMAKE_SYSTEM_NAME=Android \
    -DCMAKE_SYSTEM_VERSION=33 \
    -DCMAKE_ANDROID_ARCH_ABI=arm64-v8a \
    -DCMAKE_ANDROID_NDK=$NDK_ROOT_PATH
    -DCMAKE_ANDROID_STL_TYPE=c++_shared
```

```
# Create directories
adb shell 'mkdir /data/local/tmp/bin'
adb shell 'mkdir /data/local/tmp/models'

# Push the compiled binaries to the folder
adb push bin/* /data/local/tmp/bin/

# Push the ggml library
adb push src/libggml.so /data/local/tmp/

# Push model files
adb push models/gpt-2-117M/ggml-model.bin /data/local/tmp/models/


# Now lets do some inference ...
adb shell

# Now we are in shell
cd /data/local/tmp
export LD_LIBRARY_PATH=/data/local/tmp
./bin/gpt-2-backend -m models/ggml-model.bin -p "this is an example"
```

上述内容涉及了如何为Android平台编译和部署一个使用ggml库的应用程序。这包括设置编译环境、使用CMake进行交叉编译、将编译后的二进制文件和库推送到Android设备上，并在设备上执行推理。下面是每个步骤的详细解释：

1. 设置Android NDK（Native Development Kit）

1. **下载并解压NDK**：
    - 访问Android NDK下载页面下载最新版本的NDK。NDK是一套工具，允许你使用C和C++代码为Android应用进行本地编译。

2. **设置NDK环境变量：**

   - 设置`NDK_ROOT_PATH`环境变量，以便在构建过程中引用。环境变量应指向你解压NDK的目录。

2. 使用CMake配置Android项目

使用以下CMake命令配置项目，为Android平台编译：

```
cmake .. \
    -DCMAKE_SYSTEM_NAME=Android \
    -DCMAKE_SYSTEM_VERSION=33 \
    -DCMAKE_ANDROID_ARCH_ABI=arm64-v8a \
    -DCMAKE_ANDROID_NDK=$NDK_ROOT_PATH \
    -DCMAKE_ANDROID_STL_TYPE=c++_shared
```

- `-DCMAKE_SYSTEM_NAME=Android`：指定目标系统为Android。
- `-DCMAKE_SYSTEM_VERSION=33`：指定目标Android API等级（例如33代表Android 13）。
- `-DCMAKE_ANDROID_ARCH_ABI=arm64-v8a`：设置目标架构为ARM 64位。
- `-DCMAKE_ANDROID_NDK`：提供NDK的路径，用于交叉编译。
- `-DCMAKE_ANDROID_STL_TYPE=c++_shared`：使用共享版本的C++标准库。

3. 在Android设备上创建目录和推送文件

使用Android Debug Bridge (ADB) 工具推送文件并创建必要的目录：

```
# 创建目录
adb shell 'mkdir /data/local/tmp/bin'
adb shell 'mkdir /data/local/tmp/models'

# 推送编译后的二进制文件
adb push bin/* /data/local/tmp/bin/

# 推送ggml库文件
adb push src/libggml.so /data/local/tmp/

# 推送模型文件
adb push models/gpt-2-117M/ggml-model.bin /data/local/tmp/models/
```

4. 在Android设备上运行推理

1. **连接到设备并设置环境：**

   - 使用`adb shell`连接到Android设备的命令行。
   - 设置库路径环境变量，确保系统能找到动态链接库。

```
adb shell
cd /data/local/tmp
```

```
export LD_LIBRARY_PATH=/data/local/tmp
```

2. **运行推理程序：**
    - 使用`./bin/gpt-2-backend`执行推理，指定模型文件和输入文本。

```
./bin/gpt-2-backend -m models/ggml-model.bin -p "this is an example"
```

这些步骤为在Android设备上编译和运行使用ggml库的应用程序提供了完整指南，使得可以在移动设备上本地执行模型推理，有助于提高执行速度并减少网络依赖。这对于需要在移动端进行快速数据处理和决策的应用尤为重要。

Resources

- GGML - Large Language Models for Everyone: a description of the GGML format provided by the maintainers of the `llm` Rust crate, which provides Rust bindings for GGML
- marella/ctransformers: Python bindings for GGML models.
- go-skynet/go-ggml-transformers.cpp: Golang bindings for GGML models
- smspillaz/ggml-gobject: GObject-introspectable wrapper for use of GGML on the GNOME platform.

上述内容列举了几个与GGML（Georgi Gerganov's machine learning library）相关的资源，这些资源涵盖了GGML格式的描述以及为不同编程语言提供的绑定（bindings）。以下是对每个资源的详细解释：

1. GGML - Large Language Models for Everyone

- **链接：**GGML - Large Language Models for Everyone
- **描述**：这是由`llm` Rust crate的维护者提供的GGML格式的描述文档。`llm`是一个为Rust语言提供GGML绑定的库，使得Rust开发者能够方便地使用GGML库。这个文档可能详细介绍了GGML的设计理念、功能以及如何在Rust中使用GGML。

2. marella/ctransformers

- **链接：**marella/ctransformers
- **描述**：这是一个提供Python绑定的库，使得Python开发者可以直接在Python中使用GGML模型。这对于Python社区中的机器学习和数据科学从业者特别有用，因为他们可以利用Python的便捷性和丰富的数据科学生态系统，同时享受到GGML提供的高效计算能力。

3. go-skynet/go-ggml-transformers.cpp

- **链接：**go-skynet/go-ggml-transformers.cpp
- **描述**：这是为Go语言提供GGML模型的绑定库。通过这个库，Go开发者能够在Go应用程序中集成和运行GGML模型。这对于构建高性能服务器端应用或需要在Go环境中处理大量数据的应用程序尤其有用。

4. smspillaz/ggml-gobject

- **链接：**smspillaz/ggml-gobject
- **描述**：这是一个为GGML提供GObject-introspectable包装的库。GObject是GNOME平台的核心对象系统，通过这个包装，GGML模型可以在支持GObject的语言（如C, Python, JavaScript等）中使用，特别适

用于GNOME和其他基于GTK的应用程序。

总结

这些资源展示了GGML库的多样性和可扩展性，不同语言的绑定使得GGML可以被广泛应用在各种项目和平台上。对于机器学习和深度学习开发者来说，这些资源提供了将GGML集成到他们所选择的开发环境中的能力，无论是在桌面应用、服务器端还是其他任何支持这些语言的环境。