Setting Up a Whisper ASR Worker Node with 8GB GPUs

Introduction

This guide outlines how to set up a dedicated Whisper Automatic Speech Recognition (ASR) worker node for the RENTAHAL architecture. Leveraging the power of affordable 8GB GPUs, you can create a highly efficient speech transcription service that integrates seamlessly with your existing infrastructure.

Why 8GB GPUs Are Perfect for Whisper

Modern 8GB GPUs offer an excellent balance for running Whisper models:

1. **Sufficient VRAM**: 8GB is enough for most Whisper models, including the medium model that delivers excellent accuracy
2. **Cost-effective**: Current generation 8GB GPUs (RTX 4060, RTX 3060 Ti) are affordable ($250-350)
3. **Energy efficient**: Lower power consumption than higher-end cards
4. **Widely available**: Easy to source in the market

Model Options and Hardware Requirements

Model	Parameters	VRAM Required	Speed (relative)	Ideal GPU
tiny	39M	~1GB	~32x	Any 8GB GPU
base	74M	~1GB	~16x	Any 8GB GPU
small	244M	~2GB	~6x	Any 8GB GPU
medium	769M	~5GB	~2x	Any 8GB GPU
large	1550M	~10GB	1x	12GB+ GPU required

For 8GB GPUs, the **medium** model offers the best balance of accuracy and speed, with the capability to transcribe audio significantly faster than real-time.

Hardware Setup

Recommended Specifications

- **GPU**: NVIDIA RTX 3060/4060 (8GB) or better
- **CPU**: 6+ cores recommended
- **RAM**: 16GB system RAM
- **Storage**: 20GB+ for model storage
- **OS**: Windows 10/11 or Linux (Ubuntu 22.04+ recommended)

- **CUDA**: CUDA 11.8 or newer

Software Implementation

Option 1: Faster-Whisper Implementation (Recommended)

We'll use the Faster-Whisper library, which is up to 4x faster than the original OpenAI implementation while using less memory.

Step 1: Set Up the Environment

```bash
# Create a virtual environment
python -m venv whisper-worker-env
source whisper-worker-env/bin/activate  # On Windows: whisper-worker-env\Scripts\activate

# Install dependencies
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
pip install fastapi uvicorn pydantic scipy numpy
pip install faster-whisper
```

Step 2: Create the FastAPI Service

Create a file named `whisper_worker.py`:

```
python
```

```python
import os
import time
import tempfile
import base64
import logging
from typing import List, Optional, Dict
from fastapi import FastAPI, HTTPException, BackgroundTasks
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel, Field
import torch
from faster_whisper import WhisperModel

# Configure logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
logger = logging.getLogger("whisper-worker")

# Initialize FastAPI
app = FastAPI(title="Whisper ASR Worker")

# Add CORS middleware
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Configuration
MODEL_SIZE = os.environ.get("MODEL_SIZE", "medium")
COMPUTE_TYPE = os.environ.get("COMPUTE_TYPE", "float16")  # float16, int8
NUM_WORKERS = int(os.environ.get("NUM_WORKERS", "2"))
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
BEAM_SIZE = int(os.environ.get("BEAM_SIZE", "5"))
VAD_FILTER = os.environ.get("VAD_FILTER", "True").lower() == "true"

# Initialize model on startup
model = None

@app.on_event("startup")
async def startup_event():
    global model
```

```
    logger.info(f"Initializing Whisper Model (size={MODEL_SIZE}, device={DEVICE}, compute_type=
    try:
        model = WhisperModel(
            MODEL_SIZE,
            device=DEVICE,
            compute_type=COMPUTE_TYPE,
            num_workers=NUM_WORKERS
        )
        logger.info("Model loaded successfully")
    except Exception as e:
        logger.error(f"Failed to load model: {str(e)}")
        raise

class TranscriptionRequest(BaseModel):
    audio: str  # Base64 encoded audio data
    language: Optional[str] = None
    task: str = "transcribe"  # transcribe or translate
    beam_size: Optional[int] = None
    vad_filter: Optional[bool] = None
    word_timestamps: bool = False

class TranscriptionSegment(BaseModel):
    id: int
    start: float
    end: float
    text: str

class TranscriptionResponse(BaseModel):
    text: str
    segments: List[TranscriptionSegment]
    language: str
    processing_time: float

@app.get("/health")
async def health_check():
    """Health check endpoint."""
    if model is None:
        raise HTTPException(status_code=503, detail="Model not loaded")

    gpu_info = {
        "available": torch.cuda.is_available(),
        "device_count": torch.cuda.device_count(),
        "device_name": torch.cuda.get_device_name(0) if torch.cuda.is_available() else "N/A",
        "memory_allocated": f"{torch.cuda.memory_allocated(0) / 1024**2:.2f} MB" if torch.cuda.
```

```
    }

    return {
        "status": "healthy",
        "model_size": MODEL_SIZE,
        "device": DEVICE,
        "compute_type": COMPUTE_TYPE,
        "gpu_info": gpu_info,
    }

@app.post("/predict", response_model=TranscriptionResponse)
async def transcribe_audio(request: TranscriptionRequest, background_tasks: BackgroundTasks):
    """Transcribe audio using Whisper."""
    if model is None:
        raise HTTPException(status_code=503, detail="Model not loaded")

    try:
        start_time = time.time()

        # Decode base64 audio
        audio_data = base64.b64decode(request.audio)

        # Save to temporary file
        with tempfile.NamedTemporaryFile(suffix=".wav", delete=False) as temp_file:
            temp_path = temp_file.name
            temp_file.write(audio_data)

        background_tasks.add_task(os.unlink, temp_path)  # Schedule file cleanup

        # Set transcription parameters
        beam_size = request.beam_size if request.beam_size is not None else BEAM_SIZE
        vad_filter = request.vad_filter if request.vad_filter is not None else VAD_FILTER

        # Perform transcription
        logger.info(f"Transcribing audio file: {temp_path}")
        segments, info = model.transcribe(
            temp_path,
            language=request.language,
            task=request.task,
            beam_size=beam_size,
            vad_filter=vad_filter,
            word_timestamps=request.word_timestamps
        )
```

```
        # Process results
        segments_list = []
        full_text = ""

        for i, segment in enumerate(segments):
            segments_list.append(TranscriptionSegment(
                id=i,
                start=segment.start,
                end=segment.end,
                text=segment.text
            ))
            full_text += segment.text + " "

        processing_time = time.time() - start_time
        logger.info(f"Transcription completed in {processing_time:.2f} seconds")

        return TranscriptionResponse(
            text=full_text.strip(),
            segments=segments_list,
            language=info.language,
            processing_time=processing_time
        )

    except Exception as e:
        logger.error(f"Error in transcription: {str(e)}")
        raise HTTPException(status_code=500, detail=str(e))

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("whisper_worker:app", host="0.0.0.0", port=8002, log_level="info")
```

Step 3: Create Startup Scripts

Create `start-whisper-worker.cmd` for Windows:

```batch
@echo off
cd /d %~dp0
call whisper-worker-env\Scripts\activate
set MODEL_SIZE=medium
set COMPUTE_TYPE=float16
set NUM_WORKERS=2
set BEAM_SIZE=5
set VAD_FILTER=True
python whisper_worker.py
```

Or `start-whisper-worker.sh` for Linux:

```bash
#!/bin/bash
cd "$(dirname "$0")"
source whisper-worker-env/bin/activate
export MODEL_SIZE=medium
export COMPUTE_TYPE=float16
export NUM_WORKERS=2
export BEAM_SIZE=5
export VAD_FILTER=True
python whisper_worker.py
```

Option 2: Original Whisper Implementation

If you prefer to use the original OpenAI Whisper implementation:

```bash
# Install dependencies
pip install openai-whisper
pip install fastapi uvicorn
```

Then modify the `whisper_worker.py` file to use OpenAI's Whisper instead of Faster-Whisper. The API endpoints can remain the same.

Integrating with RENTAHAL

Step 1: Add the Worker Node to RENTAHAL Configuration

Add the Whisper worker to your database:

python

```python
# Add via your webgui.py or API
whisper_worker = {
    'name': 'whisper_worker',
    'address': 'localhost:8002',  # Update with your actual server address
    'type': 'transcribe',
    'health_score': 100.0,
    'is_blacklisted': False,
    'last_active': datetime.now().isoformat()
}
```

Step 2: Update WebSocket Handler

Add transcription handling to your WebSocket Manager:

python

```python
# Add to your query type handling logic
if query.query_type == 'transcribe':
    # Route to Whisper worker
    result = await process_query_worker_node(query)

    # Process result as text
    return {
        "type": "query_result",
        "result": result,
        "result_type": "text",
        "processing_time": processing_time,
        "cost": cost
    }
```

Step 3: Update UI to Include Speech Transcription Option

Add a new query type for audio transcription in your UI:

```javascript
// Add this option to your query type dropdown
const queryTypes = [
    { value: "chat", label: "Chat" },
    { value: "vision", label: "Vision" },
    { value: "imagine", label: "Imagine" },
    { value: "transcribe", label: "Transcribe Audio" }
];
```

Performance Optimization

Memory Optimization

1. **Use `float16` or `int8` compute type**:
   - `float16` offers good balance between accuracy and speed
   - `int8` is useful when memory is constrained
2. **VAD Filtering**:
   - Enable Voice Activity Detection to skip silent parts of audio
3. **Batch Processing**:
   - Implement a queue system for handling multiple requests

Speed Optimization

1. **Beam Size**:
   - Lower beam sizes (2-3) for faster processing
   - Higher beam sizes (5+) for better accuracy
2. **Model Selection**:
   - Use `medium` model for the best balance of speed and accuracy on 8GB GPUs
   - Use `small` model for greater speed at slight accuracy cost

Monitoring and Maintenance

Health Check Monitoring

Implement a health check routine in your main RENTAHAL system:

python

```python
async def check_worker_health(worker_name, address):
    try:
        async with aiohttp.ClientSession() as session:
            url = f"http://{address}/health"
            async with session.get(url, timeout=5) as response:
                if response.status == 200:
                    # Update worker health in database
                    update_worker_health(worker_name, 100)
                    return True
                else:
                    logger.warning(f"Worker {worker_name} health check failed: {response.status
                    update_worker_health(worker_name, 50)
                    return False
    except Exception as e:
        logger.error(f"Error checking worker health: {str(e)}")
        update_worker_health(worker_name, 0)
        return False
```

GPU Memory Monitoring

Add GPU memory monitoring to your health check:

python

```python
def get_gpu_memory_info():
    if torch.cuda.is_available():
        memory_allocated = torch.cuda.memory_allocated(0) / 1024**2  # MB
        memory_reserved = torch.cuda.memory_reserved(0) / 1024**2    # MB
        return {
            "allocated_mb": memory_allocated,
            "reserved_mb": memory_reserved,
            "utilization_percent": torch.cuda.utilization(0)
        }
    return {"error": "CUDA not available"}
```

Conclusion

By setting up a dedicated Whisper ASR worker node with an 8GB GPU, you've expanded your RENTAHAL architecture to include high-quality speech transcription capabilities. This setup enables:

1. **Fast transcription**: Process audio much faster than real-time

2. **Memory efficiency**: Optimized for 8GB GPUs
3. **Scalability**: Add more nodes as demand increases
4. **Seamless integration**: Works within your existing RENTAHAL infrastructure

This approach of keeping speech recognition as a distinct service allows you to optimize resources and scale each component independently, maintaining the design philosophy of your modular architecture.