Setting Up a Bark AI Special Worker Node with 8GB GPUs

Introduction

Modern 8GB GPUs offer an excellent price-to-performance ratio for running specialized AI models like Bark. This guide will walk you through setting up a dedicated Bark AI worker node that integrates with your existing RENTAHAL architecture.

Why 8GB GPUs Are Perfect for Bark

Current generation 8GB GPUs (like the RTX 4060, RTX 3060 Ti, or RX 7600) provide:

1. Sufficient VRAM for Bark's transformer models (~2-3GB for smaller models)
2. CUDA cores/compute units needed for fast inference
3. Excellent value proposition (typically $250-350)
4. Low power consumption compared to higher-end cards
5. Wide availability in the market

Hardware Requirements

- **GPU**: NVIDIA RTX 3060/4060 (8GB) or better
- **CPU**: 4+ cores recommended
- **RAM**: 16GB minimum
- **Storage**: 20GB+ for model storage
- **OS**: Windows 10/11 or Linux (Ubuntu 20.04+ recommended)

Software Setup

Step 1: Install Basic Dependencies

```
bash

# Create a dedicated environment
python -m venv bark-worker-env
source bark-worker-env/bin/activate  # On Windows: bark-worker-env\Scripts\activate

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

Step 2: Install Bark and Dependencies

```bash
# Install Bark from the official repository
pip install git+https://github.com/suno-ai/bark.git
```

Step 3: Create the FastAPI Service

Create a file named `bark_worker.py`:

```
python
```

```python
import os
import torch
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import numpy as np
from bark import SAMPLE_RATE, generate_audio, preload_models
from scipy.io.wavfile import write as write_wav
import base64
import tempfile
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Initialize FastAPI
app = FastAPI(title="Bark AI Worker")

# Model configuration
USE_SMALL_MODELS = os.environ.get("USE_SMALL_MODELS", "False").lower() == "true"
PRELOAD_ON_START = os.environ.get("PRELOAD_ON_START", "True").lower() == "true"

# Preload models if configured
if PRELOAD_ON_START:
    logger.info("Preloading Bark models...")
    preload_models(
        text_use_small=USE_SMALL_MODELS,
        coarse_use_small=USE_SMALL_MODELS,
        fine_use_small=USE_SMALL_MODELS,
        use_gpu=torch.cuda.is_available()
    )
    logger.info("Models loaded successfully")

class BarkInput(BaseModel):
    prompt: str
    speaker: str = None
    text_temp: float = 0.7
    waveform_temp: float = 0.7

class BarkResponse(BaseModel):
    audio_base64: str
    sample_rate: int
```

```python
@app.get("/health")
async def health_check():
    """Health check endpoint."""
    gpu_info = "GPU available" if torch.cuda.is_available() else "CPU only"
    return {
        "status": "healthy",
        "model": "Bark AI",
        "hardware": gpu_info,
        "version": "1.0"
    }

@app.post("/predict", response_model=BarkResponse)
async def generate_speech(input_data: BarkInput):
    """Generate speech from text using Bark."""
    try:
        logger.info(f"Generating audio for prompt: {input_data.prompt[:50]}...")

        # Generate audio
        audio_array = generate_audio(
            input_data.prompt,
            history_prompt=input_data.speaker,
            text_temp=input_data.text_temp,
            waveform_temp=input_data.waveform_temp
        )

        # Save audio to temporary file
        with tempfile.NamedTemporaryFile(suffix=".wav", delete=False) as tmp_file:
            temp_path = tmp_file.name
            write_wav(temp_path, SAMPLE_RATE, audio_array)

        # Read binary data and encode as base64
        with open(temp_path, "rb") as audio_file:
            audio_data = audio_file.read()
            audio_base64 = base64.b64encode(audio_data).decode("utf-8")

        # Clean up
        os.remove(temp_path)

        logger.info("Audio generation complete")
        return BarkResponse(audio_base64=audio_base64, sample_rate=SAMPLE_RATE)

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

```
if __name__ == "__main__":
    import uvicorn
    uvicorn.run("bark_worker:app", host="0.0.0.0", port=8010, log_level="info")
```

Step 4: Create a Startup Script

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

```batch
@echo off
cd /d %~dp0
call bark-worker-env\Scripts\activate
set USE_SMALL_MODELS=False
set PRELOAD_ON_START=True
python bark_worker.py
```

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

```bash
#!/bin/bash
cd "$(dirname "$0")"
source bark-worker-env/bin/activate
export USE_SMALL_MODELS=False
export PRELOAD_ON_START=True
python bark_worker.py
```

Integrating with RENTAHAL

Step 1: Add the Worker to Your RENTAHAL Config

Add the Bark worker to your existing worker configuration:

```python
# Add via your webgui.py or API
workers = [
    # Existing workers
    {'name': 'llama_worker', 'address': 'localhost:8000', 'type': 'chat'},
    {'name': 'llava_worker', 'address': 'localhost:8001', 'type': 'vision'},
    # Add Bark worker
    {'name': 'bark_worker', 'address': 'localhost:8010', 'type': 'speech'}
]
```

Step 2: Update WebSocket Handler

Ensure your WebSocket Manager can handle the "speech" query type:

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

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

Performance Optimization

For optimal performance on an 8GB GPU:

1. **Memory Management**:
   - Use `USE_SMALL_MODELS=True` for lower VRAM usage
   - Clear PyTorch CUDA cache between requests: `torch.cuda.empty_cache()`
2. **Batch Processing**:
   - Implement a queue system for handling multiple requests
   - Set reasonable timeouts (30-60 seconds per generation)

3. **Temperature Settings**:
   - Lower temperatures (0.6-0.7) provide more consistent results
   - Higher temperatures (0.8-1.0) offer more creative variations

Advanced Configuration

Speaker Presets

Create a speaker presets directory and JSON file:

```python
SPEAKER_PRESETS = {
    "en_male_1": "v2/en_speaker_6",
    "en_female_1": "v2/en_speaker_9",
    "german_male": "v2/de_speaker_1",
    "german_female": "v2/de_speaker_3",
    "spanish_male": "v2/es_speaker_2",
    "spanish_female": "v2/es_speaker_1",
    # Add more presets as needed
}

# Then in your predict endpoint:
speaker_preset = SPEAKER_PRESETS.get(input_data.speaker, input_data.speaker)
```

Voice Cloning (Optional)

For voice cloning capabilities, you can integrate the community-developed Bark voice cloning tools:

```bash
# Install additional requirements
pip install encodec hubert-base fairseq
```

Note: Use voice cloning responsibly and ensure you have the necessary rights to clone voices.

Testing

1. Start your Bark worker: `start-bark-worker.cmd`
2. Test the health endpoint: `http://localhost:8010/health`
3. Test generation with curl:

bash

```
curl -X POST http://localhost:8010/predict \
  -H "Content-Type: application/json" \
  -d '{"prompt":"Hello, this is a test of the Bark AI speech generation system. How does it sou
```

Troubleshooting

- **CUDA Out of Memory**: Reduce batch size, use small models, or upgrade GPU
- **Slow Generation**: Check for CPU bottlenecks, ensure model is on GPU
- **Model Loading Errors**: Verify proper installation of dependencies

Conclusion

You now have a dedicated Bark AI worker node that integrates with your RENTAHAL architecture. This setup leverages affordable 8GB GPUs to provide high-quality text-to-speech capabilities while keeping the model loaded in memory for fast inference.

By using specialized workers like this, you can efficiently allocate resources across your system, with different nodes optimized for different tasks, all while maintaining a unified API interface.