BMF DiffusionFlow: 将 ComfyUI 工作流转换为 BMF 图

摘要

本文档是 **DiffusionFlow** 的实现方案，旨在将 ComfyUI 的前端交互式工作流与 Babit Multimedia Framework (BMF) 的高性能后端调度引擎深度整合。该方案的核心思想是实现 ComfyUI Node 到 BMF Module 的直接、通用化转换，并将 ComfyUI 的 JSON 工作流动态编译为 BMF 的 GraphConfig，并借助 BMF 的回调机制将执行进度实时反馈回 ComfyUI 前端。这将充分利用 ComfyUI 灵活的生态系统和 BMF 高性能的 C++ 调度核心，为复杂的多媒体 AI 处理流程提供一个兼具易用性与执行效率的解决方案。

---

0. 写在前面

本方案的设计与构思，是在与胡老师进行多次深入讨论后形成的最终成果。方案中所涉及的技术栈 (WebSocket/Vue.js/JavaScript)，以及与 Diffusion 模型相关的领域知识，与我本科阶段的学习内容和知识储备高度契合。

1. ComfyUI 与 BMF 运行流程分析

1.1 ComfyUI 工作流执行

ComfyUI 的执行流程始于用户在前端构建的图，该图以 JSON 格式描述。

1. **接收任务**: `ComfyUI/server.py` 中的 `/prompt` API 端点接收到包含工作流 JSON 的请求。
2. **验证与调度**: `ComfyUI/execution.py` 中的 `PromptExecutor` 接收此任务。它首先通过 `validate_prompt` 验证图的有效性（如必需输入是否连接）。
3. **拓扑排序**: `PromptExecutor` 创建一个 `ExecutionList` 实例，这是一个基于拓扑排序的调度器。它会分析节点依赖关系，确定一个可以执行的节点列表。
4. **节点执行**: `ExecutionList` 逐个执行节点。对于每个节点，它会：
    - 从 `nodes.NODE_CLASS_MAPPINGS` (在 `ComfyUI/nodes.py` 中定义，并由 `comfy_extras` 扩展) 中找到对应的节点类并实例化。
    - 调用该实例的执行函数 (由 `FUNCTION` 属性指定)。
    - 将输出结果缓存起来，供下游节点使用。
5. **进度反馈**: 在节点执行期间，通过 `ComfyUI/server.py` 中的 WebSocket 连接向前端发送 `executing`, `progress` 等消息。

这个流程完全在 Python 中进行，调度逻辑相对简单，但在处理大规模并发和复杂的IO密集型或计算密集型混合任务时，可能会遇到性能瓶颈。

1.2 BMF 图执行

BMF 设计用于高效的音视频和通用数据处理，其核心是 C++ 实现的调度引擎。

1. **图定义**: BMF 的图可以通过两种方式定义：
    - **Python API**: 使用 `bmf.graph()` 和 `.module()` 等链式调用构建图。
    - **JSON Config**: 定义一个 `GraphConfig` JSON 文件，描述图的拓扑结构、节点信息和流连接。
2. **模块 (Module)**: BMF 的基本处理单元，可以是 C++、Python 或 Go 实现的。每个模块都有一个 `process` 方法来处理输入的 `Packet` 并产生输出 `Packet`。
3. **包 (Packet)**: BMF 中数据流动的基本单位。关键特性是 `bmf.Packet` **可以封装任意 Python 对象**。
4. **执行**: 通过 `bmf.graph().run_by_config(config)`，BMF 的 C++ 调度器会解析 JSON 配置，实例化所有模块，并根据流的连接关系高效地调度数据包（Packets）在模块间传递。

1.3 核心思路：可行性与优势

将两者结合的核心思路是：**用 BMF 的调度器替代 ComfyUI 的 `ExecutionList` 调度器**。

- **可行性**:
    - **节点映射**: 我们可以创建一个通用的 BMF Python 模块 `ComfyNodeRunner`，它能够加载并执行任何 ComfyUI 节点。
    - **图转换**: ComfyUI 的工作流 JSON 可以被程序化地转换为 BMF 的 GraphConfig JSON。
    - **数据流**: ComfyUI 节点间传递的 `MODEL`, `LATENT`, `VAE` 等复杂 Python 对象，可以被直接放入 `bmf.Packet` 中，在 `ComfyNodeRunner` 实例之间传递。由于在同一进程内，这只是对象引用的传递，**完全避免了耗时的序列化和反序列化**。
- **优势**:
    - **高性能**: 享受 BMF C++ 调度器带来的低延迟和高吞吐。
    - **生态复用**: 完美复用 ComfyUI 庞大且活跃的自定义节点生态，无需对任何现有节点代码进行修改。
    - **前端体验**: 保持 ComfyUI 前端优秀、直观的交互体验。
    - **异构流水线**: 未来可将 ComfyUI 节点与 BMF 原生的音视频处理模块（如 FFmpeg 模块）无缝混编在同一张图中，构建更强大的 AI 多媒体应用。

2. 统一工作流转换方案架构设计

2.1 整体架构

1. **用户交互**: 用户在 ComfyUI 前端建图，点击执行。
2. **后端分发**: ComfyUI 后端 (`ComfyUI/execution.py`) 增加一个开关，例如在 `prompt` 的 `extra_data` 中包含 `{"enable_bmf": true}`。
3. **工作流转换**: 若开关为 `true`，则调用新增的 `BmfWorkflowConverter`，将 ComfyUI 工作流转换为 BMF GraphConfig。
4. **BMF 执行**: BMF 引擎根据生成的 GraphConfig 执行任务。
5. **进度反馈**: BMF 引擎通过回调机制，将节点执行状态通知给一个回调函数，该函数再通过 ComfyUI 的 `PromptServer` WebSocket 发送给前端。

2.2 ComfyUI 后端改造

在 `ComfyUI/execution.py` 的 `PromptExecutor.execute` 方法中进行修改：

```
            bmf_graph = converter.convert()
            bmf_graph.run() # This will be a blocking call# In execution.py ->
PromptExecutor.execute

def execute(self, prompt, prompt_id, extra_data={}, execute_outputs=[]):
    # ... (existing code) ...

    # DiffusionFlow: Check if BMF execution is enabled
    if extra_data.get("enable_bmf", False):
        try:
            from bmf_converter import BmfWorkflowConverter
            converter = BmfWorkflowConverter(prompt, self.server)
            bmf_graph = converter.convert()
            bmf_graph.run() # This will be a blocking call
            # After run, BMF callbacks will have handled progress
            self.success = True # Assume success if no exception
            # ... handle history etc. ...
            return
        except Exception as e:
            # ... handle BMF execution error ...
            print(f"BMF execution failed: {e}")
            self.success = False
            return

    # Fallback to original ComfyUI execution
    # ... (existing code for ExecutionList) ...
```

2.3 `BmfWorkflowConverter` 实现

这是一个新的 Python 类，负责转换逻辑。

```
# In a new file, e.g., bmf_converter.py
import bmf
import nodes # To access NODE_CLASS_MAPPINGS

class BmfWorkflowConverter:
    def __init__(self, comfy_workflow, server_instance):
        self.comfy_workflow = comfy_workflow
        self.server = server_instance
        self.bmf_graph_config = {
            "mode": "normal",
            "option": {"dump_graph": 1},
            "nodes": []
        }

    def convert(self):
        # 1. Iterate through ComfyUI nodes to create BMF node configs
        for node_id, node_info in self.comfy_workflow.items():
```

```
            bmf_node_config = self._create_bmf_node_config(node_id, node_info)
            self.bmf_graph_config["nodes"].append(bmf_node_config)

        # 2. Iterate through ComfyUI links to set up BMF streams
        self._setup_bmf_streams()

        # 3. Create and return a BMF graph object
        graph = bmf.graph()
        graph_config_obj = bmf.GraphConfig(self.bmf_graph_config)

        # Register callback for progress
        graph.add_user_callback(bmf.BmfCallBackType.LATEST_TIMESTAMP,
self._progress_callback)

        # This part needs to be implemented in bmf.graph to pass the config
        graph.load_config(graph_config_obj) # Hypothetical API
        return graph

    def _create_bmf_node_config(self, node_id, node_info):
        class_type = node_info["class_type"]

        # Pass ComfyUI node details as options to the generic BMF module
        option = {
            "class_type": class_type,
            "inputs": node_info["inputs"], # Pass original inputs
            "comfy_node_id": node_id # For progress reporting
        }

        return {
            "id": int(node_id), # BMF expects integer IDs
            "module_info": {
                "name": "ComfyNodeRunner",
                "type": "python",
                "path": "path/to/comfy_node_runner.py",
                "entry": "comfy_node_runner.ComfyNodeRunner"
            },
            "option": option,
            "input_streams": [],
            "output_streams": []
        }

    def _setup_bmf_streams(self):
        # ... logic to parse comfy_workflow links and populate
        # "input_streams" and "output_streams" in self.bmf_graph_config["nodes"]
        # A link from comfy_node_A (output slot 0) to comfy_node_B (input slot 0)
        # becomes a BMF stream from bmf_node_A to bmf_node_B.
        pass

    def _progress_callback(self, bmf_status_para):
        # bmf_status_para would contain info like node_id and status
        # Example: bmf_status_para = {"node_id": "3", "status": "executing"}
        node_id = bmf_status_para.get("node_id")
        status = bmf_status_para.get("status")
```

```
        if self.server and node_id and status:
            message = {"node": node_id}
            self.server.send_sync(status, message, self.server.client_id)
```

2.4 通用 ComfyUI 节点运行器 (`ComfyNodeRunner`) BMF 模块

这是整个方案的粘合剂。

```
# In comfy_node_runner.py
import bmf
from bmf import Module, Task, Packet, Timestamp
import nodes # ComfyUI's nodes
import torch

class ComfyNodeRunner(Module):
    def __init__(self, node_id, option):
        self.node_id = node_id
        self.option = option

        # Instantiate the actual ComfyUI node
        self.class_type = option['class_type']
        self.comfy_node_class = nodes.NODE_CLASS_MAPPINGS[self.class_type]
        self.comfy_node_instance = self.comfy_node_class()

        # Store constant widget inputs
        self.widget_inputs = {}
        for name, value in option.get('inputs', {}).items():
            if not isinstance(value, list): # It's a widget value, not a link
                self.widget_inputs[name] = value

    def process(self, task):
        # Collect inputs for the ComfyUI node's execution function
        kwargs = self.widget_inputs.copy()

        # Unpack inputs from BMF packets
        for i, (key, stream) in enumerate(task.get_inputs().items()):
            input_queue = stream
            if not input_queue.empty():
                pkt = input_queue.get()
                if pkt.timestamp != Timestamp.EOF:
                    # Get the ComfyUI input name that corresponds to this BMF input stream
                    input_name = self._get_input_name_by_stream_key(key)

                    # Unwrap the Python object from the packet
                    unwrapped_data = pkt.get(object)
                    kwargs[input_name] = unwrapped_data

        # All inputs ready, execute the node function
        function_name = self.comfy_node_instance.FUNCTION
        execute_func = getattr(self.comfy_node_instance, function_name)
```

```
        # Special handling for nodes that want list inputs
        if getattr(self.comfy_node_class, "INPUT_IS_LIST", False):
            # ... logic to wrap all inputs in lists ...
            pass

        with torch.inference_mode(): # Mimic ComfyUI execution context
             results = execute_func(**kwargs)

        # Wrap outputs in BMF packets and send
        if results:
            # results is a tuple of outputs
            for i, res_item in enumerate(results):
                if i in task.get_outputs():
                    # Wrap the Python object (Tensor, Model, etc.) directly
                    out_pkt = Packet(res_item)
                    out_pkt.timestamp = 0 # Manage timestamps if needed
                    task.get_outputs()[i].put(out_pkt)

        return bmf.ProcessResult.OK
```

2.5 前端进度反馈

如 `BmfWorkflowConverter._progress_callback` 所示，BMF 的回调机制是关键。BMF 引擎需要在执行每个节点的 `process` 函数前后触发回调，并将节点 ID 和状态（如 `executing`, `executed`）传递给回调函数。回调函数随后可以调用 `self.server.send_sync` 将信息发往前端。这需要对 BMF Engine 的 `Graph` 类进行少量修改或扩展，以支持更精细的节点级状态回调。

3. 实例：`nodes_sd3.py` 的转换

SD3 的核心采样器是 `KSamplerSD3`，但它内部调用了 `nodes.common_ksampler`，这与标准的 `KSampler` 行为类似。一个典型的 T2I (Text-to-Image) 流程是：

`CheckpointLoader` -> (`CLIPTextEncode` -> `KSamplerSD3`), (`EmptyLatentImage` -> `KSamplerSD3`) -> `VAEDecode` -> `SaveImage`

3.1 ComfyUI 工作流（简化表示）

```
{
    "nodes": [
        {"id": "4", "type": "CheckpointLoaderSimple", ...},
        {"id": "6", "type": "CLIPTextEncode", "inputs": {"clip": ["4", 1], ...}},
        {"id": "7", "type": "CLIPTextEncode", "inputs": {"clip": ["4", 1], ...}},
        {"id": "5", "type": "EmptyLatentImage", ...},
        {"id": "3", "type": "KSamplerSD3", "inputs": {
            "model": ["4", 0], "positive": ["6", 0], "negative": ["7", 0], "latent_image":
["5", 0], ...}
        },
        {"id": "8", "type": "VAEDecode", "inputs": {"samples": ["3", 0], "vae": ["4", 2]}},
        {"id": "9", "type": "SaveImage", "inputs": {"images": ["8", 0]}}
    ],
    "links": [...]
```

```
}
```

3.2 转换后的 BMF GraphConfig (片段)

```
{
    "nodes": [
        {
            "id": 4,
            "module_info": {"name": "ComfyNodeRunner", ...},
            "option": {"class_type": "CheckpointLoaderSimple", "inputs": {"ckpt_name":
"sd3_medium.safetensors"}, ...},
            "output_streams": [
                {"identifier": "stream_4_0"}, // MODEL
                {"identifier": "stream_4_1"}, // CLIP
                {"identifier": "stream_4_2"}  // VAE
            ]
        },
        {
            "id": 6,
            "module_info": {"name": "ComfyNodeRunner", ...},
            "option": {"class_type": "CLIPTextEncode", "inputs": {"text": "a cat"}, ...},
            "input_streams": [{"identifier": "stream_4_1"}],
            "output_streams": [{"identifier": "stream_6_0"}]
        },
        // ... other nodes ...
        {
            "id": 3,
            "module_info": {"name": "ComfyNodeRunner", ...},
            "option": {"class_type": "KSamplerSD3", "inputs": {"seed": 123, ...}, ...},
            "input_streams": [
                {"identifier": "stream_4_0"}, // from model
                {"identifier": "stream_6_0"}, // from positive
                {"identifier": "stream_7_0"}, // from negative
                {"identifier": "stream_5_0"}  // from latent
            ],
            "output_streams": [{"identifier": "stream_3_0"}]
        },
        // ... etc. ...
    ]
}
```

3.3 节点配置详解

对于 `KSamplerSD3` 节点（ID: 3）：

- `module_info`: 指向我们通用的 `ComfyNodeRunner` 模块。
- `option`:
  - `"class_type": "KSamplerSD3"`: 告诉 `ComfyNodeRunner` 需要实例化哪个 ComfyUI 节点。
  - `"inputs": {"seed": 123, ...}`: 包含了所有非链接的、由 Widget 提供的输入值。

- `input_streams`: 定义了该 BMF 节点的输入。`identifier` 的值 (`stream_4_0`, `stream_6_0` 等) 会与上游节点的 `output_streams` 中的 `identifier` 相匹配，从而建立连接。
- `output_streams`: 定义了该节点的输出流。下游节点（如 `VAEDecode`）的 `input_streams` 会引用这里的 `identifier` (`stream_3_0`)。

4. 实现计划

采用分阶段的实施策略，从核心功能的验证逐步扩展到与 ComfyUI 的深度集成和高级功能支持。

阶段一：核心转换逻辑与离线验证 (6.29-7.12)

此阶段的目标是在不涉及 ComfyUI 服务器的情况下，验证工作流转换和节点执行的核心逻辑是否正确。

1. **实现 `BmfWorkflowConverter` (基础版)**
   - **任务**: 编写 `BmfWorkflowConverter` 类，重点关注图的拓扑结构转换。
   - **细节**:
     - 解析 ComfyUI 工作流 JSON 中的 `nodes` 和 `links` 列表。
     - 将每个 ComfyUI 节点映射为一个 BMF `node` 配置。`module_info` 固定指向 `ComfyNodeRunner`。
     - 根据 ComfyUI `links`，为每个 BMF `node` 配置正确填充 `input_streams` 和 `output_streams`。流的 `identifier` 必须能够唯一地连接上下游节点。
   - **产出**: 一个能够将 ComfyUI JSON 转换为功能上等价的 BMF GraphConfig Python 字典的转换器。
2. **实现 `ComfyNodeRunner` BMF 模块 (基础版)**
   - **任务**: 创建核心的通用节点运行器模块。
   - **细节**:
     - 在模块的 `__init__` 方法中，根据传入的 `option`（包含 `class_type`），从 `nodes.NODE_CLASS_MAPPINGS` 实例化对应的 ComfyUI 节点。
     - 在 `process` 方法中，实现核心执行逻辑：
       - 从输入的 BMF `Task` 中获取 `Packet`。
       - 调用 `pkt.get(object)` 从 `Packet` 中拆包出上游节点传递过来的 Python 对象。
       - 收齐所有输入后，动态调用 ComfyUI 节点实例的 `FUNCTION` 指定的方法。
       - 将执行结果（一个元组）中的每个元素分别封装进新的 `bmf.Packet` 中，并放入输出 `Task` 的对应队列。
   - **产出**: 一个可独立运行的 BMF Python 模块，能够执行任何遵循 ComfyUI 节点约定的类。
3. **离线测试与验证**
   - **任务**: 编写一个独立的 Python 脚本，将阶段一的成果串联起来。
   - **细节**:
     - 手动准备一个简单的 ComfyUI 工作流 JSON 文件（例如：`LoadImage` -> `ImageInvert` -> `SaveImage`）。
     - 在脚本中，调用 `BmfWorkflowConverter` 将该 JSON 转换为 BMF GraphConfig。
     - 使用 `bmf.graph().run_by_config(config)` 来执行生成的图。
     - 检查 `SaveImage` 节点对应的 `ComfyNodeRunner` 是否成功保存了预期的、经过反色处理的图片。

- **产出**: 验证核心方案可行性的成功案例，为后续集成打下基础。

阶段二：与 ComfyUI 服务器集成（7.13-7.26）

此阶段的目标是将 BMF 执行流程无缝嵌入到 ComfyUI 的后端，并实现进度反馈。

4. **注入 BMF 调度逻辑**
   - **任务**: 修改 ComfyUI 后端，使其能够选择性地使用 BMF 调度器。
   - **细节**:
     - 按照设计文档，在 `ComfyUI/execution.py` 的 `PromptExecutor.execute` 方法中添加逻辑分支。
     - 通过检查 `extra_data` 中的标志位（如 `enable_bmf`）来决定是进入新的 BMF 执行路径，还是沿用原有的 `ExecutionList` 路径。
   - **产出**: 一个修改版的 ComfyUI 后端，能够根据前端请求触发 BMF 工作流。
5. **实现前端进度反馈**
   - **任务**: 利用 BMF 的回调机制将执行状态实时同步到 ComfyUI 前端。
   - **细节**:
     - **BMF 端**: 探索并利用 `bmf.graph().add_user_callback()`。注册一个回调函数，该函数应在 BMF 引擎执行每个节点的 `process` 函数前后被触发。这可能需要 BMF 提供相应的回调类型（如 `NODE_EXEC_START`, `NODE_EXEC_END`）或通过现有回调（如 `LATEST_TIMESTAMP`）变通实现。
     - **Converter 端**: 在 `BmfWorkflowConverter` 中实现该回调函数 (`_progress_callback`)。
     - **ComfyUI 端**: 回调函数内部，获取当前执行的 ComfyUI 节点 ID 和状态，然后调用 `self.server.send_sync(status, message, ...)`，以 ComfyUI 前端兼容的格式发送 WebSocket 消息。
   - **产出**: 用户在前端点击执行后，能够像原生执行一样看到节点变绿、进度条更新等实时反馈。

阶段三：高级功能与 CI/CD （7.27-8.9）

此阶段专注于处理复杂的节点行为和系统的稳定性。

6. **支持高级节点特性**
   - **任务**: 增强 `ComfyNodeRunner` 和 `BmfWorkflowConverter` 以处理特殊节点。
   - **细节**:
     - `INPUT_IS_LIST`: `ComfyNodeRunner` 在调用节点执行函数前，检查该标志。若为 `True`，确保所有传入的参数都以列表形式提供。
     - `OUTPUT_IS_LIST`: `BmfWorkflowConverter` 在生成图时需识别此标志。一种可能的实现是，在该 BMF 节点后自动插入一个"拆包" (Scatter) BMF 模块，该模块接收包含列表的单个 Packet，并输出多个 Packet，每个包含列表中的一个元素。
   - **产出**: 对 ComfyUI 列表操作的兼容支持，使得更多复杂工作流可以被正确转换。
7. **完善前端交互**
   - **任务**: 在 ComfyUI 前端提供一个用户友好的触发方式。
   - **细节**: 开发一个简单的 ComfyUI 前端扩展（JavaScript），在界面上添加一个"使用 BMF 执行"的复选框或按钮。当启用时，它会自动在提交给后端的 prompt JSON 的 `extra_data` 中添加 `"enable_bmf": true`。
   - **产出**: 一个完整的、从前端到后端的 End-to-End 解决方案。

8. **测试与 CI/CD**
    - **任务**: 建立 CI/CD 流水线以保障系统稳定性。
    - **细节**:
        - **自动化测试**: 借鉴 ComfyUI 和 BMF 现有的 CI 实践 (参考)，搭建自动化测试流水线。利用 Github Actions 等工具，在代码提交时自动执行一系列基准 ComfyUI 工作流（如 SD3 T2I），并验证 BMF 执行后的输出结果，确保核心转换和执行功能不被破坏。
        - **错误处理**: 在 `ComfyNodeRunner` 和 `BmfWorkflowConverter` 中增加完善的`try...except`机制，确保节点执行或图转换失败时，能捕获并上报清晰的错误信息。
        - **手动验证**: 补充性地，使用社区中更复杂的自定义节点和工作流进行手动测试，以覆盖自动化测试未包含的边缘情况。
    - **产出**: 一个稳定、可靠，并具备良好错误报告机制和自动化测试保障的系统。