6.3 [MobiCom'20]Heimdall: mobile GPU coordination platform for augmented reality applications

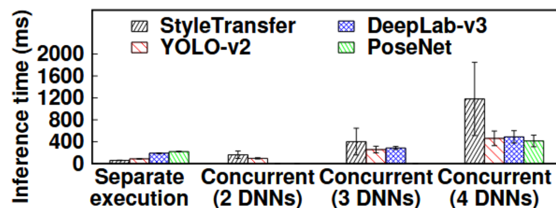
[9] 这篇文章主要工作:

面向增强现实（AR）应用的移动 GPU 协调平台，解决了多 DNN 并发执行和渲染之间的资源争用问题。通过设计伪抢占机制，灵活协调渲染和 DNN 任务，既能保持 30 fps 的渲染稳定性，又能解决多个 DNN 任务的资源争用和延迟问题。Heimdall 有效优化了 GPU 任务的调度，显著提高了帧率和推理速度。

6.3.1 Background / Problem Statement

AR 应用需要持续运行多个 DNN 来分析物理世界和用户行为，同时还需要实时渲染虚拟内容，并协调这些任务在移动 GPU 上的资源争用问题

Multi-DNN GPU Contention. 新兴的 AR 应用需要并发运行多个 DNN。并发运行多个 DNN 还会在有限的移动 GPU 资源上产生严重争用，导致整体性能下降。



(a) MACE over LG V50 (immersive online shopping scenario).

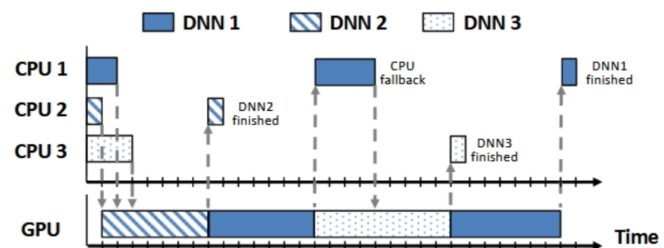


Figure 3: Multi-DNN GPU contention example.

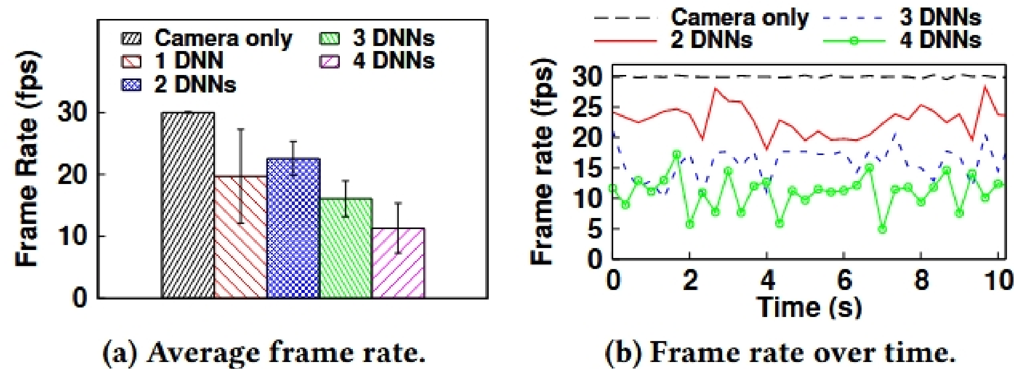
6.3.1.1 Challenges

图 2(a) 显示，当更多 DNN 在移动 GPU 上争用时，推理时间相较于只运行单个 DNN 时显著增加（标记为单独执行）。4 个 DNN 同时运行时，StyleTransfer 的延迟从 59.93±3.68 毫秒增加到 1181±668 毫秒），使得难以满足延迟要求。

图 3 展示了在上述推理过程中可能出现的 3-DNN GPU 争用场景。

这里 dnn2 首先派对到 GPU 中，当 DNN1 和 dnn3 完成预处理并且尝试访问 gpu 时，发生争用，因为 dnn2 已经在执行了，dnn1 就需要等 dnn2 才能执行然后 dnn1 可能某些算子需要在 cpu 上运行，此时会释放 gpu，释放之后呢 dnn3 就可以运行；当 dnn1 再次请求 gpu 执行时需要等到 dnn3 完成所以，dnn1 的推理时长会非常大

Rendering-DNN GPU Contention 当渲染与 DNN 并行运行时，GPU 争用会降低并波动帧率，损害用户体验



(a) Average frame rate. (b) Frame rate over time.

Figure 4: Rendering-DNN GPU contention on MACE over LG V50 (immersive online shopping scenario).

这里是渲染任务的场景, 在没有任务 DNN 的情况下单独运行相机时，可以稳定 30FPS，当渲染与 DNN 并行运行时，右边的图是速率随时间变化，GPU 争用会降低并波动帧率，从而降低用户体验（例如，当后台运行 4 个 DNN 时，帧率从 30 fps 降至 11.99 fps

总结：现有的移动深度学习框架缺乏对多 DNN 和渲染并发执行的支持，严重的 GPU 争用导致 DNN 和渲染任务的性能显著下降。

Why Not Apply Desktop GPU Scheduling?

Limited Architecture Support. 桌面和服务器 GPU 通常通过硬件支持或抢占机制解决这一问题，但移动 GPU 由于缺乏架构支持;

Limited Memory Bandwidth. 通过细粒度的上下文切换实现 GPU 的时间共享, 允许高优先级任务在其他任务运行时抢占 GPU. 频繁的上下文切换由于状态大小较大，导致较高的内存开销，这对内存带宽有限的移动 GPU 来说是个负担。

例如，三星 Galaxy S10 中的 ARM Mali-G76 GPU（Exynos 9820）与 CPU 共享的内存带宽为 26.82 GB/s，比 NVIDIA RTX 2080Ti（即 616 GB/s）小 23 倍。每次上下文切换需要 120 MB 的内存传输（=20 核心 ×24 执行通道/核心 ×64 寄存器/通道 ×32 位），即使假设 GPU 完全利用共享的内存带宽，这也会导致至少 4.36 毫秒的延迟。

design: 我们设计了一种伪抢占机制，通过将 DNN 和渲染任务分解为小块，实现时间共享，以解决移动 GPU 不支持并行化的问题。尽管任务分块可能导致一些延迟开销，但可以通过灵活调整调度单元大小来平衡粒度和延迟，最大限度地减少开销。

6.3.2 Architecture

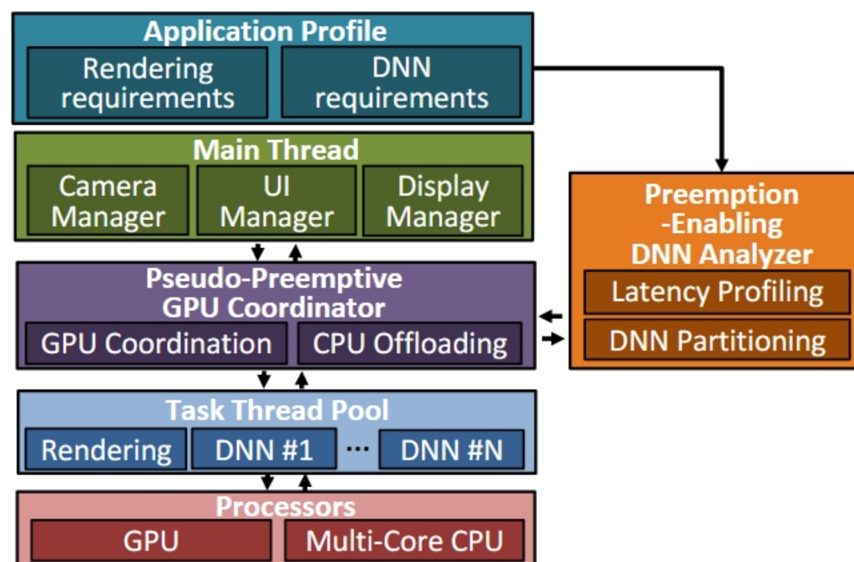


Figure 6: System Architecture of Heimdall.

根据应用配置文件（渲染帧率和分辨率、需要运行的 DNN 以及延迟约束），DNN 分析器首先对所需信息进行分析，对渲染和 DNN 推理延迟进行分析，以确定在渲染事件之间 DNN 可以占用 GPU 的时间

将 DNN 分割成可以在渲染事件之间运行的小块（调度单元），以尽量减少推理延迟开销。

在运行时，伪抢占 GPU 协调器从主线程（控制摄像头、用户界面和显示）接收多 DNN 和渲染任务，并协调它们的执行；协调器会定义一个效用函数，基于推理延迟和场景内容比较哪个 DNN 在特定时间点更重要

6.3.3 Methods

为实现伪抢占机制，Heimdall 包含以下组件：

- Preemption-Enabling DNN Analyzer: 分析器通过将庞大的 DNN 分解为小的调度单元，使多任务能够在细粒度级别上共享 GPU，而不会产生额外的调度开销。
- Pseudo-Preemptive GPU Coordinator: 根据不同的任务需求灵活调度渲染和 DNN 任务，优先保证渲染任务的帧率，并在争用严重时考虑将 DNN 任务卸载至 CPU，确保应用在资源受限情况下的性能表现。

6.3.3.1 Preemption-Enabling DNN Analyzer

分析器的任务是分析目标设备上的渲染和 DNN 推理延迟，并将 DNN 划分为适合在渲染事件之间执行的小块。在 GPU 上，离线分析是可行的，因为执行时间是稳定的，而 CPU 上的执行时间可能会因为资源争用而波动，需要动态跟踪。DNN 分割到操作符级别通常已经足够，但过细的分割可能会带来额外的延迟开销。

Latency Profiling **Rendering Latency:** 给定目标渲染帧率（f）和分辨率，分析器首先测量渲染延迟，T_{render}。DNN 在渲染事件之间能够占用 GPU 的时间（即 $\frac{1}{f} - T_{\text{render}}$）。

DNN Latency: 分析器测量目标 GPU 和 CPU 上的 DNN 推理延迟。在运行时测量 DNN 在 CPU 上的推理延迟，以跟踪由于 CPU 资源争用引起的延迟变化。

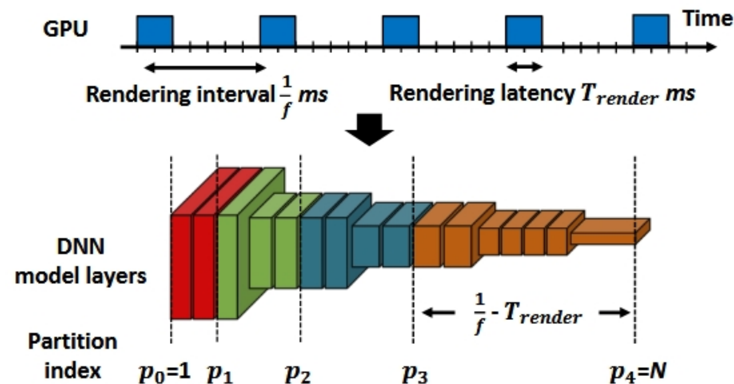


Figure 12: Operation of DNN partitioning.

DNN Partitioning 图 12 展示了 DNN 分区的操作。给定一个由 N 个操作符组成的 DNN D，设 $T(D_{i,j})$ 表示从第 i 个操作符到第 j 个操作符的子图的执行时间。我们的目标是确定一组 K 个索引 $\{p_1 = 1, p_2, p_3, ..., p_K = N\}$，以使每个分区的执行时间都在渲染间隔内，

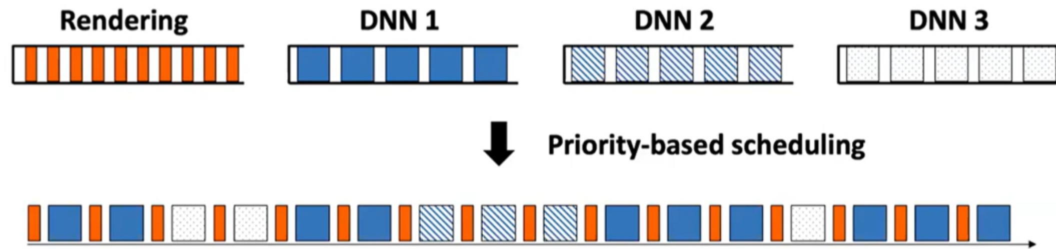
$$T(D_{p_i,p_{i+1}}) \leq \frac{1}{f} - T_{\text{render}} \quad 1 \leq i \leq K-1.$$

DNN 分区的目标是将 DNN 划分为能够在渲染间隔内执行的小块。过细的分区会增加调度开销，因此需要将尽可能多的操作符组合在一起以减少分区数量。同时，为避免 GPU 空闲时间，分区执行时间允许略微超出渲染间隔，从而提高 GPU 利用率。

6.3.3.2 Pseudo-Preemptive GPU Coordinator

协调器嵌入到应用级深度学习框架中，而不是操作系统或设备驱动层; 优先考虑渲染任务以确保 AR 应用的用户体验稳定。它通过准入控制管理 DNN 推理请求，并在每次渲染事件后进行调度，确保 DNN

和渲染任务之间的 GPU 时间共享。协调器还定义了一个效用函数来优先排序 DNN，并在 GPU 争用严重时考虑将任务卸载到 CPU。



Utility Function 比较在给定时间哪个 DNN 更重要,DNN D_i 在主线程将第 k 次推理排队的时间点 $t^i_{\text{start},k}$ 的效用被建模为两项的加权和:

$$U_{D_i}(t) = L_{D_i}(t, t^i_{\text{start},k}) + \alpha \cdot C_{D_i}(t^i_{\text{start},k}, t^i_{\text{start},k-1}),$$

其中，$L(t, t_{\text{start}})$ 是延迟效用，用于衡量推理的新鲜度，$C_{D_i}(t^i_{\text{start},k}, t^i_{\text{start},k-1})$ 是内容变化效用，用于捕捉场景内容自上次 DNN 推理以来的变化速度，且 α 是缩放因子（当前实现中经验值为 0.01）。

效用值低通常代表该任务的重要性或紧迫性较低

DNN D_i 的延迟效用计算为:

$$L_{D_i}(t, t^i_{\text{start},k}) = L^0_{D_i} - \left(\beta_i \cdot (t - t^i_{\text{start},k})^{\gamma_i}\right)^2.$$

延迟效用被建模为一个凹函数，以便效用随时间迅速降低，从而防止协调器将执行延迟过久。可以通过配置三个参数来设置 DNN 之间的优先级。β_i 控制每个 DNN 可以占用的 GPU 时间比例（例如，设置所有 DNN 的 $\beta_i = 1$ 将启用相等共享）。$L^0_{D_i}$ 和 γ_i 控制 DNN 之间的优先级；具有较高 $L^0_{D_i}$ 和 γ_i 的 DNN 将具有更高的初始效用，但效用下降更快，从而协调器可以在效用下降前更频繁地允许其抢占 GPU。

内容变化效用 D_i

计算为在 $t^i_{\text{start},k}$ 和 $t^i_{\text{start},k-1}$ 连续推理时输入帧的差异。采用【42】中的方法计算两帧的 Y 值（亮度）Y^k_h 之间的差异（它与 SSIM 高度相关，只需 $O(N)$ 计算）:

$$C_{D_i}(t^i_{\text{start},k}, t^i_{\text{start},k-1}) = \sum_{h=1}^{H} \sum_{w=1}^{W} \left| Y^k_{h,w} - Y^{k-1}_{h,w} \right|,$$

其中，H, W 是帧的高度和宽度。

Scheduling Problem and Policy 协调器通过两步操作：i) 调度 DNN 以高效共享 GPU。ii) 决定是否将一些 DNN 卸载到 CPU，以解决争用问题。

假设 N 个 DNN $D_1, ..., D_N$ 正在 GPU 上运行，并具有延迟约束 $t_{1,\max}, ..., t_{M,\max}$（根据应用场景适当设置）。这两种策略的表述如下。

MaxMinUtility: 试图最大化当前效用最低的 DNN 的效用。

$$\min_i U_{D_i}(t),$$

subject to

$$t^i_{\text{end},k} - t^i_{\text{start},k} \leq t_{i,\max}.$$

在 MaxMinUtility 策略下，协调器尝试公平分配 GPU 资源，以平衡多个 DNN 的性能。

MaxTotalUtility: 最大化所有 DNN 效用的总和。

$$\max_i \sum_{t=1}^{N} U_{D_i}(t),$$

subject to

$$t^i_{\text{end},k} - t^i_{\text{start},k} \leq t_{i,\max}.$$

协调器倾向于效用较高的 DNN（即允许其更频繁地抢占 GPU），而剩余的 DNN 则在不违反其截止时间的情况下以最小资源运行。

总结： MaxMinUtility 策略平衡多个 DNN 的效用，适用于需要公平资源分配的连续 DNN 场景；MaxTotalUtility 策略优先高效用的 DNN，适用于需要低响应时间的事件驱动场景。

Opportunistic CPU Offloading 当 GPU 争用过高时，协调器会动态决定是否将一些 DNN 任务卸载到 CPU 上，减少 GPU 的压力。

设 $P_1, P_2, ..., P_N$ 表示 N 个 DNN 运行的处理器（GPU 或 CPU）。处理器的映射通过以下问题求解：

$$\max_{P_1,P_2,...,P_N} \sum_{t=1}^{N} U_{D_i,P_i}(t),$$

其中，$U_{D_i,P_i}(t)$ 表示 D_i 在处理器 P_i 上运行的效用（受 P_i 上的推理时间影响，推理时间由分析器分析）。

6.3.3.3 Greedy Scheduling Algorithm

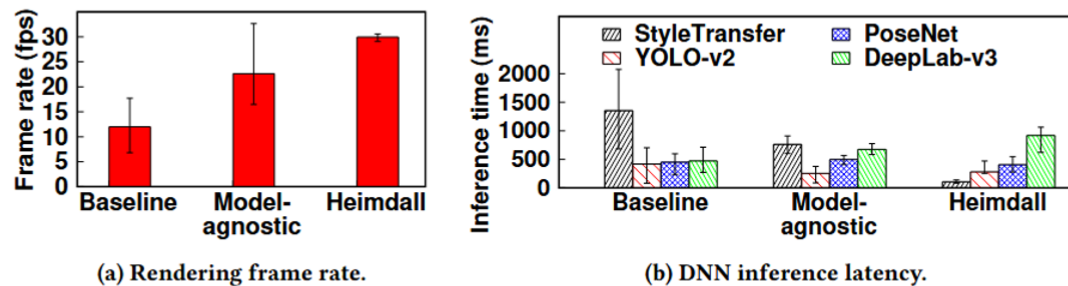
解决上述调度问题计算复杂且难以离线规划（因为解决方案会因场景内容而变化）。因此，我们采用贪心方法来获得近似解。

贪心调度算法通过实时检查 DNN 的效用值和分区剩余执行时间，动态决定在 GPU 上运行哪个 DNN 块。若某个 DNN 延迟过大，则优先执行它。此外，CPU 卸载机制用于减轻 GPU 负载，通过比较 GPU 和 CPU 的推理时间来选择卸载或重新加载 DNN，避免资源争用和延迟过大。

6.3.3.4 Additional Optimizations

在 DNN 推理过程中，一些步骤（如预处理、后处理和不受 GPU 支持的操作符）需要在 CPU 上执行。这可能会导致 GPU 空闲时间增加，特别是在处理高分辨率复杂场景时显著影响推理性能。为了解决这一问题，可以通过 GPU 和 CPU 并行化这些步骤来提高 GPU 的利用率并减少整体延迟。

6.3.4 Evaluation



(a) Rendering frame rate. (b) DNN inference latency.

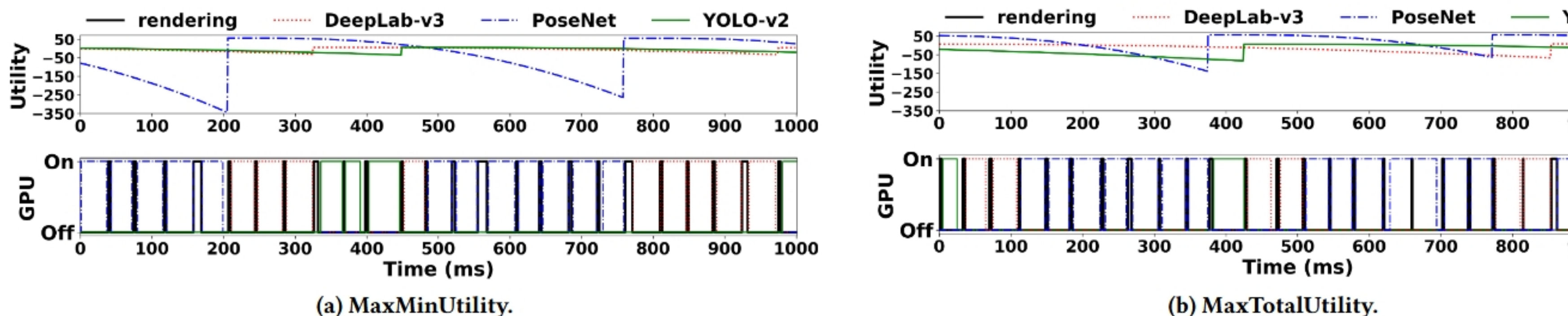
Figure 14: Performance overview of Heimdall on LG V50.

在渲染方面，baseline 的速率非常低并且波动大，中间的划分 dnn 方案比 baseline 高的原因是切分后的 dnn 避免了长期占用 gpu 运行，但是波动还是比较大的 Heimdall 能够稳定地支持 30 帧运行

Baseline 方案有显著的争用延迟，中间的方案能用减少一定的争用，但是对于 styletransfer 来说被阻塞的延迟非常大

Heimdall 能够将渲染从 12 帧到稳定 30 帧，并且能够减少 dnn 最坏执行情况的 15 倍

6.3.5 Evaluation



(a) MaxMinUtility. (b) MaxTotalUtility.

Figure 16: Performance comparison of GPU coordination policies.

图 16 展示了在沉浸式在线购物场景中，GPU 在两种策略下如何协调 3 个 DNN（即，随时间变化的效用值和 GPU 占用情况）

图 16(a) 显示了 MaxMinUtility 策略，在 t=200-800 时，随着时间的推理，utility 随之下降

下面的这里是 gpu 上的执行，可以看到，渲染是周期性占用 gpu，并且执行效用值当前最低的 DNN

显示了 MaxTotalUtility 策略，它优先处理 PoseNet，因为它的优先级高于其他 DNN, 所以蓝色这条线变化地更频繁

6.3.6 Conclusion

Heimdall：这是一个为增强现实（AR）应用设计的移动 GPU 协调平台。

伪抢占机制：该机制用于协调多个 GPU 任务。将帧率从 12 fps 提升至 30 fps，将最坏情况下的 DNN 推理延迟减少了多达 15 倍。

Heimdall 具有通用性：可以跨不同的框架进行推广应用