目录

OpenVINO™ 2022.1 中 AUTO 插件和自动批处理的最佳实践

作者：YASUNORI SHIMURA　Intel Technical Solution Specialist for Computer Vision Technology

翻译：张晶

1.1　概述

OpenVINO™ 2022.1 是自 OpenVINO™ 2018 年首次发布以来最大的更新之一，参见《OpenVINO™ 迎来迄今为止最重大更新，2022.1 新特性抢先看！》。在众多新特性中，AUTO 插件和自动批处理(Automatic-Batching)是最重要的新特性之一，它帮助开发者无需复杂的编程即可提高推理计算的性能和效率。

1.1.1 什么是 AUTO 插件？

AUTO 插件[1]，全称叫**自动设备选择**(Automatic device selection)，它是一个构建在 CPU/GPU 插件之上的虚拟插件，如图 1-1 所示。在 OpenVINO™ 文档中，“**设备(device)**”是指用于推理计算的 Intel® 处理器，它可以是受支持的 CPU、GPU、VPU（视觉处理单元）或 GNA（高斯神经加速器协处理器）或这些设备的组合[3]。

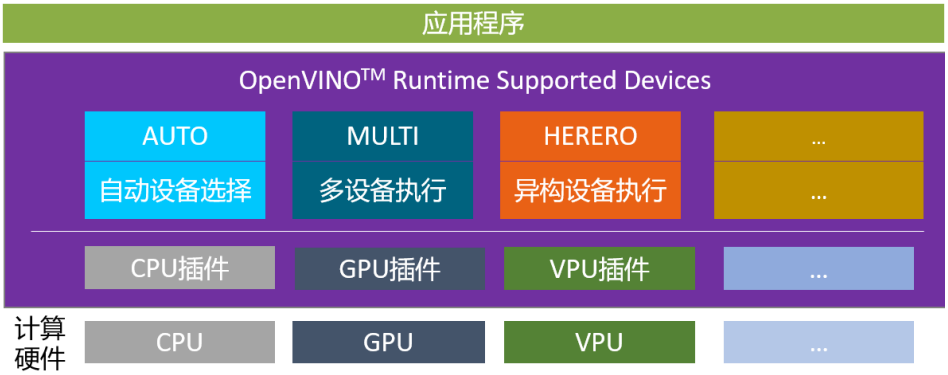

图 1-1　OpenVINO™ Runtime 支持的设备插件[3]

AUTO 插件**好处**有：

- 首先检测运行时平台上所有可用的计算设备，然后选择最佳的一个计算设备进行推理计算，并根据深度学习模型和所选设备的特性**以最佳配置使用**它。
- **使 GPU 实现更快的首次推理延迟**：GPU 插件需要在开始推理之前在运行时进行在线模型编译——可能需要 10 秒左右才能完成，具体取决于平台性能和模型的复杂性。当选择独立或集成 GPU 时，“AUTO”插件开始会首先利用 CPU 进行推理，以隐藏此 GPU 模型编译时间。
- **使用简单**，开发者只需将 compile_model()方法的 device_name 参数指定为“AUTO”即可，如图 1-2 所示。

```python
# Step2: Compile the Model, and specify the AUTO plugin
net = core.compile_model(model="yolov5s.onnx",device_name="AUTO")
```

Python

图 1-2　指定 AUTO 插件

1.1.2 什么是自动批处理？

自动批处理(Automatic Batching)[2]，又叫自动批处理执行(Automatic Batching Execution)，是 OpenVINO™ Runtime 支持的设备之一，如图 1-1 所示。

一般来说，批尺寸(batch size) 越大的推理计算，推理效率和吞吐量就越好。自动批处理执行将用户程序发出的多个异步推理请求组合起来，将它们视为多批次推理请求，并将批推理结果拆解后，返回给各推理请求。

自动批处理无需开发者手动指定。当 compile_model()方法的 config 参数设置为{“PERFORMANCE_HINT”: ”THROUGHPUT”}时，OpenVINO™ Runtime 会**自动启动**自动批处理执行，如图 1-3 所示，让开发人员以最少的编码工作即可享受计算设备利用率和吞吐量的提高。

```python
# Step2:Compile the Model,specify the AUTO plugin,start Automatic Batching
net = core.compile_model(model="yolov5s.onnx",device_name="AUTO", \
                         config={"PERFORMANCE_HINT": "THROUGHPUT"})
```

Python

图 1-3　自动启动自动批处理执行

1.2　动手学 AUTO 插件的特性

读书是学习，实践也是学习，而且是更有效的学习。本文提供了完整的实验代码，供读者一边动手实践，一边学习总结。

Github 地址: https://github.com/yas-sim/openvino-auto-feature-visualization

1.2.1 搭建实验环境

第一步，克隆代码仓到本地。

```
git clone https://github.com/yas-sim/openvino-auto-feature-visualization.git
```

第二步，在 openvino-auto-feature-visualization 路径执行：

```
python -m pip install --upgrade pip
pip install -r requirements.txt
```

第三步，下载模型并完成转换

```
omz_downloader --list models.txt
omz_converter --list models.txt
```

到此，实验环境搭建完毕。实验程序的所有配置和设置参数都硬编码在源代码中，您需要**手动修改**源代码以更改测试配置，如图 1-4 所示。

auto-test.py

auto-test.py > main

```
84      cfg['PERFORMANCE_HINT'] = ['THROUGHPUT', 'LATENCY'][1]
85      cfg['ALLOW_AUTO_BATCHING'] = 'YES'      # default=YES
86      #cfg['AUTO_BATCH_TIMEOUT']= '100'        # (milli-second) default=1000
87      #cfg["CPU_THREADS_NUM"] = "2"
88      cfg['PERFORMANCE_HINT_NUM_REQUESTS'] = '4'
89      device = ['GPU', 'AUTO', 'AUTO:GPU', 'CPU', 'BATCH:GPU(8)'][0]
90      compiled_model = core.compile_model(model, device, cfg)
91      print('Device:', device)
92      print('Config:', cfg)
93      print('niter:', niter, ', interval:', interval, 'ms')
94      opt_nireq = compiled_model.get_property('OPTIMAL_NUMBER_OF_INFER_REQUESTS')
95      print('OPTIMAL_NUMBER_OF_INFER_REQUESTS', opt_nireq)
```

图 1-4　手动修改源代码中的配置

1.2.2 AUTO 插件自动切换计算设备

GPU 插件需要在 GPU 上开始推理之前将 IR 模型编译为 OpenCL 模型。这个模型编译过程可能需要很长时间，例如 10 秒，会延迟应用程序开始推理，使得应用程序启动时的用户体验不好。

为了隐藏这种 GPU 模型编译延迟，AUTO 插件将在 GPU 模型编译进行时使用 CPU 执行推理任务；当 GPU 模型编译完成后，AUTO 插件会自动将推理计算设备从 CPU 切换到 GPU，如图 1-5 所示。

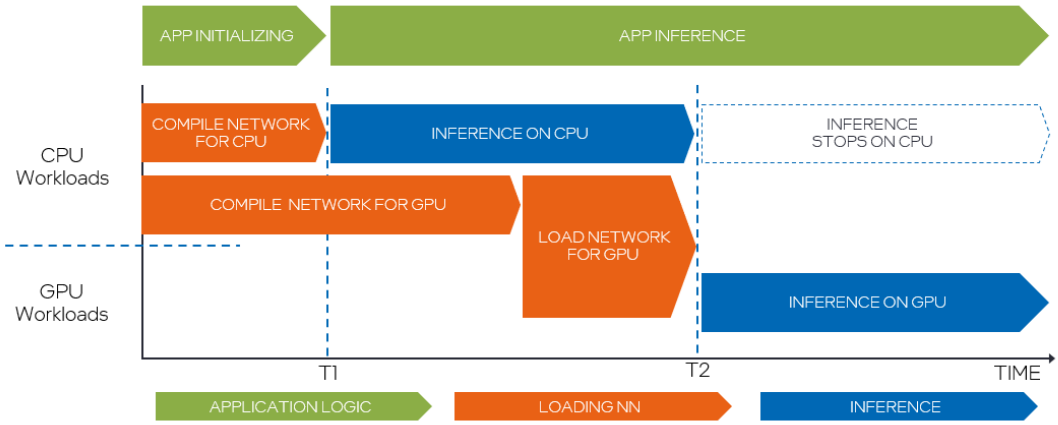

图 1-5　AUTO 插件自动切换计算设备

1.2.3 动手观察自动切换计算设备的行为

AUTO 插件会依据设备优先级[1]: dGPU > iGPU > VPU > CPU, 来选择最佳计算设备。当自动插件选择 GPU 作为最佳设备时，会发生推理设备切换，以隐藏首次推理延迟。

请注意，设备切换前后的推理延迟不同；此外，推理延迟故障可能发生在设备切换的那一刻，如图 1-6 所示。

请如图 1-6 所示，设置 auto-test-latency-graph.py 配置参数为：

```
cfg['PERFORMANCE_HINT'] = ['THROUGHPUT', 'LATENCY'][0]
```

并运行命令：

```
python auto-test-latency-graph.py
```

同时打开 Windows 任务管理器，观察 CPU 和 iGPU 的利用率。

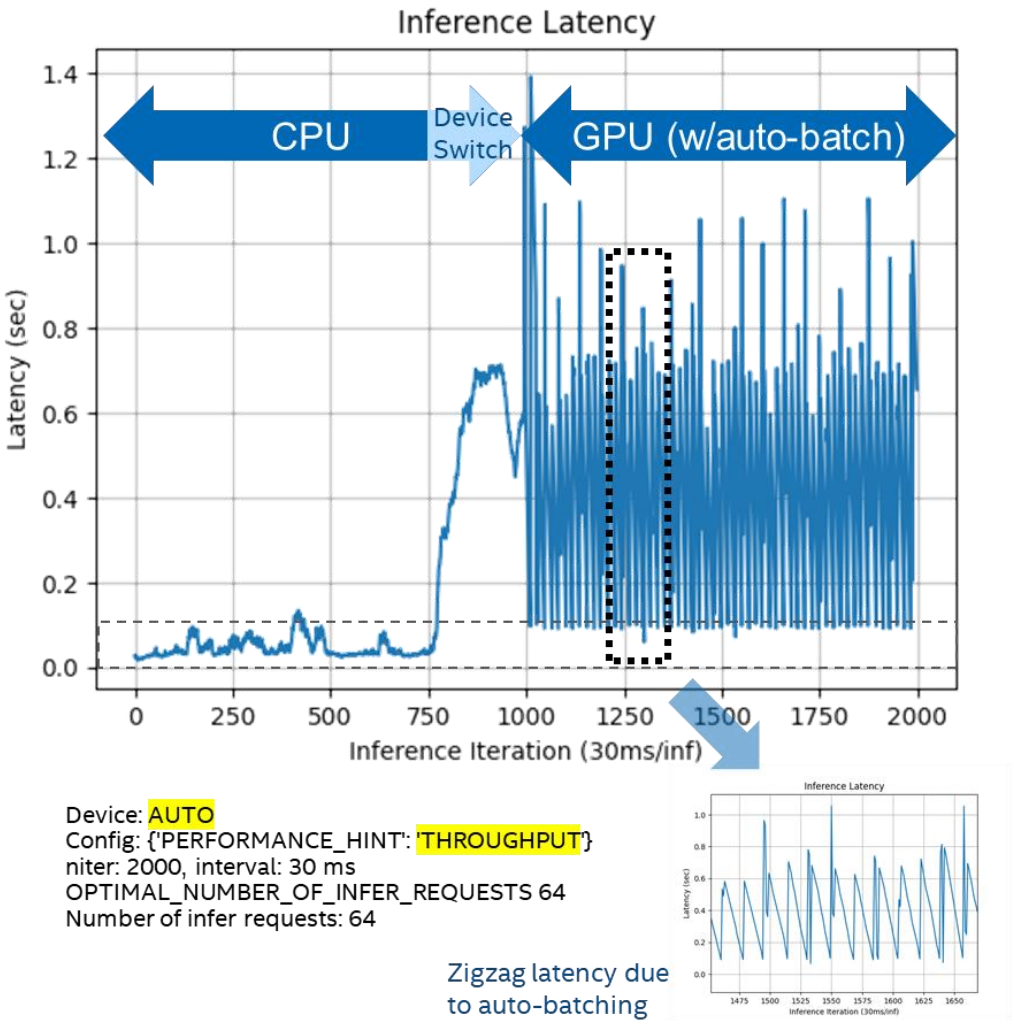

图 1-6　config={“PERFORMANE_HINT”:”THROUGPUT”}的执行行为

1.2.4 PERFORMANCE_HINT 设置

如 1.1.2 节所述，AUTO 插件的执行行为取决于 compile_model()方法的 config 参数的 PERFORMANCE_HINT 设置，如表 1-1 所示：

表 1-1　PERFORMANCE_HINT 设置

PERFORMANCE_HINT	应用场景	是否启动 Auto Batching?
'THROUGHPUT'	非实时的大批量推理计算任务	是
'LATENCY'	实时或近实时应用任务	否

设置 auto-test-latency-graph.py 配置参数为：

```
cfg['PERFORMANCE_HINT'] = ['THROUGHPUT', 'LATENCY'][1]
```

并运行命令：

```
python auto-test-latency-graph.py
```

同时打开 Windows 任务管理器，观察 CPU 和 iGPU 的利用率，运行结果如图 1-7 所示。

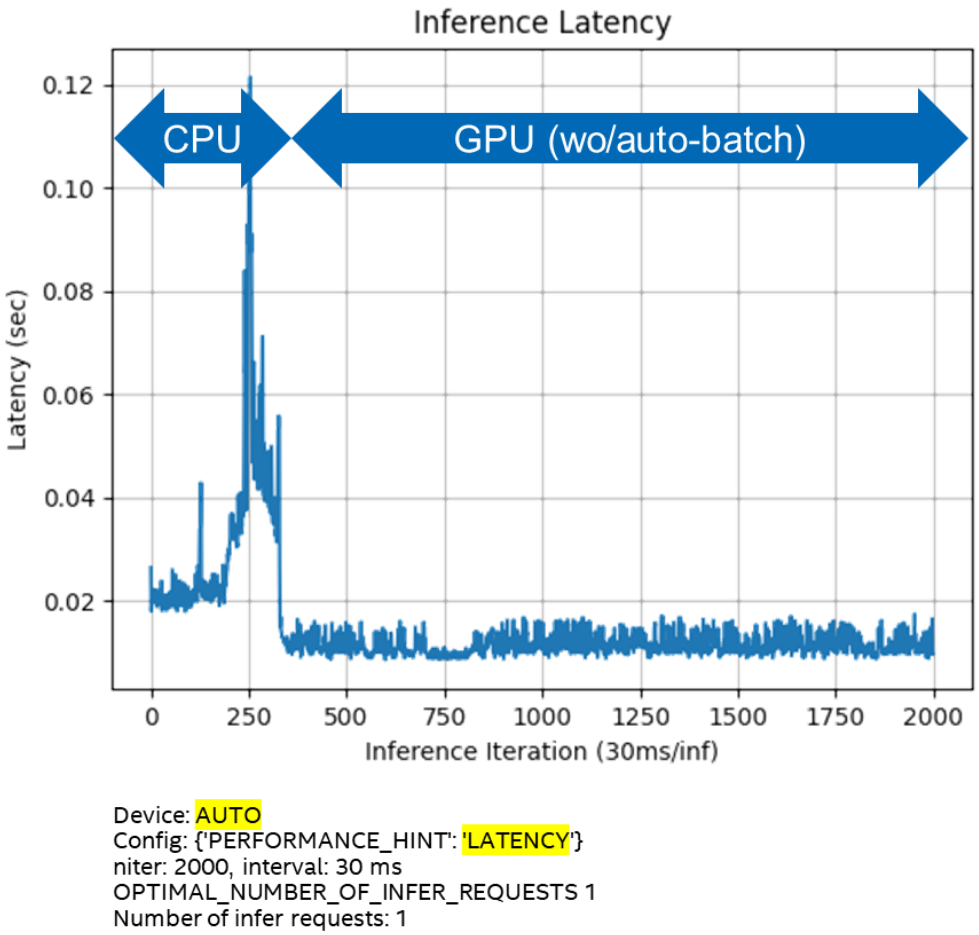

图 1-7　config={“PERFORMANE_HINT”:”LATENCY”}的执行行为

通过实验，我们可以发现，根据不同的 config 参数设置，使得 AUTO 插件可以工作在不同的模式下：

- 在 Latency 模式，不会自动启动 Auto Batching，执行设备切换后，GPU 上的推理延迟很小，且不会抖动。
- 在 THROUGHPUT 模式，自动启动 Auto Batching，执行设备切换后，GPU 上的推理延迟较大，而且会抖动。

接下来，本文将讨论 Auto Batching 对推理计算行为的影响。

1.3　动手学 Auto Batching 的特性

如 1.1.2 节所述，自动批处理执行将用户程序发出的多个异步推理请求组合起来，将它们视为多批次推理请求，并将批推理结果拆解后，返回给各推理请求，如图 1-8 所示。

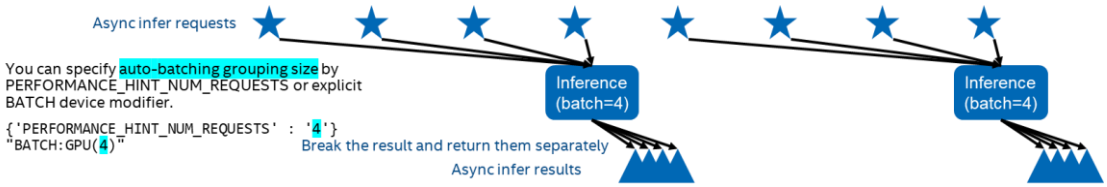

图 1-8　Auto Batching 的执行过程

Auto Batching 在收集到指定数量的异步推理请求或计时器超时（默认超时=1,000 毫秒）时启动批推理计算(batch-inference)，如图 1-9 所示。

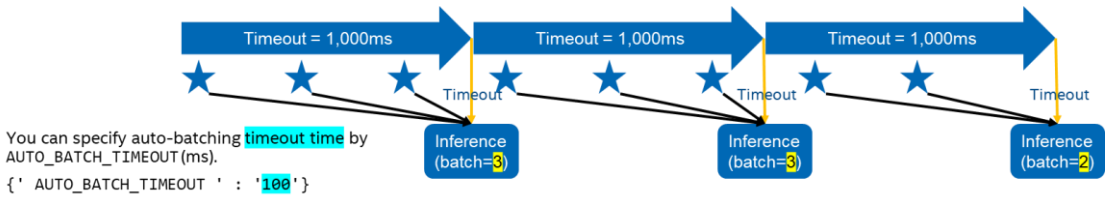

图 1-9　启动批推理计算

1.3.1 Auto Batching 被禁止时

Auto Batching 被禁止时，所有推理请求都是单独被处理的。

请配置并运行 auto-test.py。

```
Device: AUTO
Config: {'PERFORMANCE_HINT': 'LATENCY'}
niter: 20 , interval: 30 ms
OPTIMAL_NUMBER_OF_INFER_REQUESTS 1
Number of infer requests: 1
```

运行结果如图 1-10 所示，可见每一个推理请求是被单独处理的。

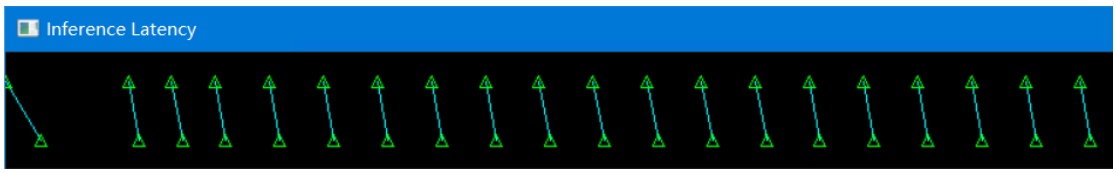

图 1-10　Auto Batching 被禁止时的运行结果

1.3.2 Auto Batching 被使能时

Auto Batching 被使能时，异步推理请求将作为多批次推理请求进行绑定和处理。推理完成后，结果将分发给各个异步推理请求并返回。需要注意的是：批推理计算不保证异步推理请求的推理顺序。

请配置并运行 auto-test.py。

```
Device: GPU
Config: {'CACHE_DIR': './cache', 'PERFORMANCE_HINT': 'THROUGHPUT',
'ALLOW_AUTO_BATCHING': 'YES'}
niter: 200 , interval: 30 ms
OPTIMAL_NUMBER_OF_INFER_REQUESTS 64
Number of infer requests: 16
```

运行结果如图 1-11 所示，可见每 16 个推理请求被组合成一个批次进行批推理计算，推理计算顺序不被保证。

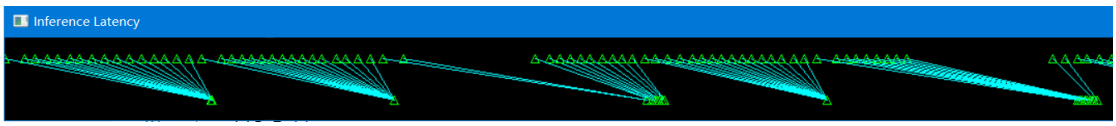

图 1-11　Auto Batching 被使能时的运行结果

1.3.3 Auto Batching 会导致推理延迟变长

由于较长的默认超时设置(默认 timeout = 1,000ms)，在低推理请求频率情况下可能会引入较长的推理延迟。

由于 Auto Batching 将等待指定数量的推理请求进入或超时计时器超时，在低推理频率的情况下，它无法在指定的超时时间内收集足够的推理请求来启动批推理计算，因此，提交的推理请求将被推迟，直到计时器超时，这将引入大于 timeout 设置的推理延迟。

为解决上述问题，用户可以通过 AUTO_BATCH_TIMEOUT 配置参数指定超时时间，以尽量减少此影响。

请使用 AutoBatching 的默认 timeout，运行 auto-test.py。

```
Device: GPU
Config: {'CACHE_DIR': './cache', 'PERFORMANCE_HINT': 'THROUGHPUT'}
niter: 20, interval: 300 ms
OPTIMAL_NUMBER_OF_INFER_REQUESTS 64
Number of infer requests: 64
```

运行结果如图 1-12 所示，由于每次都无法在 timeout 时间内收集到指定数量的推理请求，由此导致推理请求的延迟很高。

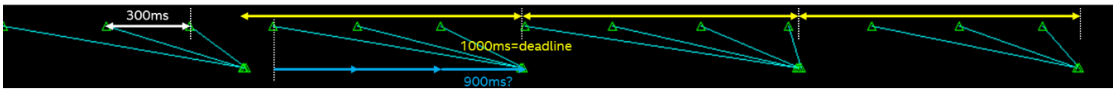

图 1-12　timeout=1000ms 运行结果

请配置 AutoBatching 的 timeout=100ms，然后运行 auto-test.py。

```
Device: GPU
Config: {'CACHE_DIR': './cache', 'PERFORMANCE_HINT': 'THROUGHPUT',
'AUTO_BATCH_TIMEOUT': '100'}
niter: 20 , interval: 300 ms
OPTIMAL_NUMBER_OF_INFER_REQUESTS 64
Number of infer requests: 16
```

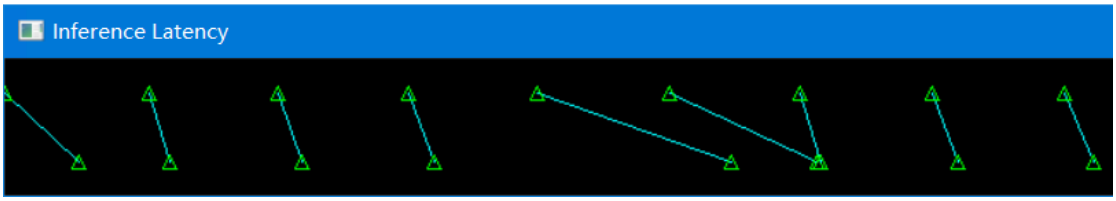

图 1-13　timeout=100ms 运行结果

运行结果如图 1-13 所示， timeout=100ms 时间内，仅能收集到一个推理请求。

1.3.4 Auto Batching 最佳实践

综上所述，Auto Batching 的最佳编程实践：

- 要记住，默认情况下 Auto Batching 不会启用。
- 只有在{'PERFORMANCE_HINT': 'THROUGHPUT', 'ALLOW_AUTO_BATCHING': 'YES'}时，Auto Batching 才启用。
- 如果您的应用程序能够**以高频率连续提交推理请求**，请使用自动批处理
- **警告**：如果您的应用间歇性地提交推理请求，则最后一个推理请求可能会出现意外的长延迟。
- 如果推理节奏或频率较低，即推理频率远低于 AUTO_BATCH_TIMEOUT（默认为 1,000 毫秒），请勿开启自动批处理。
- •您可以使用 AUTO_BATCH_TIMEOUT 参数更改自动批处理的超时设置，以最大限度地减少不需要的长延迟，参数值的单位是“ms”。
- 如果您知道工作负载的最佳批处理大小，请使用 PERFORMANCE_HINT_NUM_REQUESTS 指定适当的批处理数量，即 {'PERFORMANCE_HINT_NUM_REQUESTS'：'4'}。同时，以 GPU 为例，AUTO 插件会在后台根据可以使用的内存，模型精度等计算出最佳批处理大小

1.4 总结

本节给出 AUTO 插件和 Auto Batching 的快速小结，如表 1-2 所示。

表 1-2 AUTO 插件和自动批处理执行快速小结表

	Automatic device selection	**Automatic Batching**
描述	- 枚举系统上的可用设备，选择最佳设备并将其用于推理 - 通过使用 CPU 开始推理来隐藏 GPU 模型编译时间，编译好后再切换到 GPU	将用户程序发出的多个异步推理请求组合起来，将它们视为多批次推理请求，并将批推理结果拆解后，返回给各推理请求
优点	- 开发者不需要做详细的硬件配置 - 应用程序可以利用系统的最佳性能 - 更短的第一次推理延迟：AUTO 插件可以隐藏 GPU 模型编译时间	- 设备利用率和效率将提高 - 开发人员可以以最少的编程工作享受多批次吞吐量
缺点	* 不适合需要一致且可预测性能的人 * 设备切换前后的推理性能会有所不同（例如，“CPU” -> “GPU”） * 推理性能可能会在设备切换时刻下降（几秒钟的顺序）	* 仅在 GPU 上可用 * 默认超时=1,000 毫秒。 这可能会导致意外的长延迟性能问题。
默认启用？	默认不启动。 需要明确指定“AUTO”作为插件名称。	默认启用 仅限 GPU

如何启动	指定“AUTO”作为插件名称	ALLOW_AUTO_BATCHING=YES 是默认值。 1. ALLOW_AUTO_BATCHING=YES, device=GPU, PERFORMANCE_HINT=THROUGHPUT 2. 指定“BATCH:GPU”作为设备名称
额外注意	默认设备选择优先级：dGPU > iGPU > VPU > CPU 重要提示：如果 AUTO 插件能够选择“GPU”作为最终计算设备，并且 PERFORMANCE_HINT=THROUGHPUT 已设置，将启用自动批处理功能。	如何禁用 Auto Batching？ 使用 compile_model()或 set_property()： 1. 设置 ALLOW_AUTO_BATCHING = NO，或者 2 指定 PERFORMANCE_HINT = LATENCY

本文 GitHub 源代码链接：https://github.com/yas-sim/openvino-auto-feature-visualization

参考文献：

[1] https://docs.openvino.ai/latest/openvino_docs_OV_UG_supported_plugins_AUTO.html

[2] https://docs.openvino.ai/latest/openvino_docs_OV_UG_Automatic_Batching.html

[3] https://docs.openvino.ai/latest/openvino_docs_OV_UG_supported_plugins_Supported_Devices.html#doxid-openvino-docs-o-v-u-g-supported-plugins-supported-devices