为什么一次事件记录在 AMD 和海光上要 5～11 μs，在 NVIDIA 上几乎为零

SYCL/HIP 事件开销的成因调研：MI210、海光 BW1000（DCU）与 H100 的逐项拆分和对照

2026-11-18 · Aqua-Xpiler b9 失败分析的延伸调研（启动开销专题）
原始数据、测量程序和本报告的生成脚本：`research/data/h100_dcu_20261110/analysis_v2/claude/launch_overhead/why_amd/`

摘要

把 CUDA 程序翻译成 SYCL 后，在 AMD MI210 和海光 DCU 上，小算子每次调用会多出 5～11 μs，在 NVIDIA 上几乎没有。上一阶段已经查明：多出来的钱来自 SYCL 运行时在每次提交后都记录一个后端事件（HIP 的 `hipEventRecord`）。本次调研回答下一个问题：**同样是记录事件，为什么 AMD 和海光贵、NVIDIA 便宜，其中有没有不必要的部分**。

做法分三步：读 ROCm 7.0 的 HIP 运行时源码并抓取真实提交的 AQL 包，弄清 `hipEventRecord` 往 GPU 队列里放了什么；绕过 HIP、直接往 HSA 队列里手写同样的包，每次只加一样东西，逐项计时；在 H100 上做功能对应的操作作对照。三种卡都用 GPU 自己的时钟核对，海光 DCU 上把前两步完整重做了一遍。

结论。HIP 的一次事件记录，在队列里是一个会让队列停下来的屏障包，带系统范围的 acquire 和 release 两道栅栏，外加一个放在主机内存里的完成信号。MI210 上：屏障包 0.65 μs，主机内存里的信号 2.4～2.6 μs，系统 acquire 在空 kernel 上 1.5～1.9 μs，下一个 kernel 要访存时合计 4.9～6.8 μs；系统 release 要把 L2 里的脏数据写回显存，每 MiB 约 0.7 μs，但 L2 只留得住约 4～5 MiB 脏数据，所以最多约 3.1 μs。读数据的小 kernel 每次因此多 9～11.6 μs。海光 DCU 上 GPU 端每次多 5.4 μs（屏障包更慢，没有 acquire 后续罚时），主机端 DTK 的 `hipEventRecord` 另要 5.7 μs CPU 时间，因为它给每个事件都用能触发中断的信号，并挂一个主机回调。NVIDIA 的普通事件不让流停下来，也不需要写回 L2，GPU 端只多 0.04 μs。

不必要的部分主要有六处：记录事件时做 acquire；文档里写着的 `hipEventReleaseToDevice` 没有实现；每个事件都按"主机要直接读数据"做系统 release；只给 GPU 用的信号也放主机内存；DTK 给每个事件挂中断和回调；HIP 已有的"把事件挂在 kernel 包上"的路径，SYCL 运行时没有用。按"最少必要"算，AMD 上一个事件约 1～3 μs（取决于信号放在哪、要不要单独的屏障包）。

A100 上的另一笔开销。A100 上 SYCL 不管返不返回事件都比原生慢 0.5～0.9 μs。这不是事件的问题，而是 SYCL 运行时自己每次提交要跑约 0.7 μs 的 CPU 代码（返回事件约 1.1 μs）；A100 上提交本来就卡在 CPU 上，所以一比一显现。MI210 上同样有这笔钱，只是被 GPU 时间盖住了（6.2 节）。

一、背景：问题从哪里来
二、预备知识：队列、屏障、信号和栅栏
三、调研方法
四、HIP 往队列里放了什么
五、MI210 逐项拆分
六、NVIDIA 对照
七、海光 DCU
八、真实访存 kernel 上的总开销
九、综合：为什么 AMD 和海光贵，NVIDIA 便宜
十、哪些地方不必要或不合理
十一、可以怎么改，对 SYCL 和转译器的意义
十二、和已有工作的关系
十三、局限和待验证的问题
十四、建议的下一步
附录 A　文件与复现
附录 B　术语表
附录 C　调研过程记录与更正

一、背景：问题从哪里来

1.1 现象

b9 实验里，Sol 在 MI210 上交付的 router、qknorm 两个算子保持率低于 1（0.82、0.94）。只把 SYCL 的提交写法换成不返回事件的 `nd_launch`，其余代码不动，保持率就变成 1.47、1.54。CUDA 和 HIP 程序本身没有这笔开销，于是要问：是不是 SYCL 多做了本不需要做的事？

1.2 上一阶段已经查清的部分

上一阶段的调研（`launch_overhead/README.md`）查清了这笔开销从哪来。SYCL 规范要求 `queue::submit` 和 `queue::parallel_for` 每次返回一个 `sycl::event`。运行时不知道调用方会不会用这个事件，只好每次都在后端真的记录一个：

- DPC++ 的 HIP 适配层（Unified Runtime）每次"新建事件对象（`hipEventCreateWithFlags(..., hipEventDisableTiming)`）、启动 kernel、`hipEventRecord`"，调用方丢掉事件时再销毁；
- AdaptiveCpp 每个操作从事件池取一个事件，再 `hipEventRecord`。

原生 CUDA/HIP 程序只在需要时才记录事件，翻译过来的程序却每个 kernel 都记一次。表 1 是空 kernel 上的实测。

表 1　每次调用的时间（μs，空 kernel，连续提交，7 轮取最好；上一阶段的测量）

写法	MI210（DPC++ HIP）	DCU（AdaptiveCpp）	A100（DPC++ CUDA）
原生启动	1.32～1.44	1.92～1.98	2.52
原生启动 + 每次记录一个事件	6.08～6.49	7.24～7.63	2.90
SYCL 普通提交（`q.parallel_for`）	6.14～6.48	7.86～8.10	3.81
SYCL 不返回事件（DPC++ `nd_launch`）	1.43～1.58	—	3.03
AdaptiveCpp 即时提交 + 粗粒度事件	—	3.03～3.17	—

第 2 行是诊断用的对照：原生程序平时不会每次启动都记录事件。翻译出的程序实际付出的代价，要看第 3 行比第 1 行多多少。

这张表要分两件事看：

- **带事件的两行（第 2、3 行），MI210 和 DCU 上几乎一样慢。**可见普通提交慢，主要慢在它强加的那次事件记录，不是 SYCL 运行时的代码。原生 HIP 程序平时不记录事件（第 1 行），所以没有这笔钱。问题分两层：SYCL 的接口要求每次提交都返回事件，在 SYCL 层；HIP 的事件本身做得太重，在 HIP 层。本报告主要查后一层。
- **去掉事件之后，SYCL 仍比原生多一点。**A100 上多 0.5 μs（第 4 行），而且带事件时也多 0.9 μs（第 3 行比第 2 行）；DCU 上 AdaptiveCpp 即时提交加粗粒度事件比原生多约 1.1 μs（第 5 行）；MI210 上只多 0.1～0.25 μs。这是 SYCL 运行时自己每次提交的 CPU 开销。A100 上提交本来就卡在 CPU 上，这笔钱一比一显现；MI210 上同样有约 0.4 μs，只是大部分被 GPU 时间盖住了。见 6.2 节。

影响面不小。b9 里 Sol 的 221 个配置中，MI210 和 DCU 各有约 38% 每次调用不到 20 μs，在这些配置上 5～8 μs 的额外开销占四分之一以上。在 DCU 上，整个程序打开 AdaptiveCpp 的即时提交并加粗粒度事件，其余代码不动，官方计分器给出 router 0.82 → 1.07、dsv3 0.79 → 0.98、topk_moe 0.77 → 1.37、qknorm 1.28 → 1.58。

1.3 这次要回答的问题

1. `hipEventRecord` 在 GPU 上具体做了什么？每一部分各花多少时间？
2. 为什么同样的操作在 NVIDIA 上便宜？

3. 哪些部分是必要的，哪些不必要或不合理？在工具链层面能怎么改，各能省多少？
4. 海光 DCU（DTK）和 AMD 原版 ROCm 是否一样？
5. A100 上事件本身便宜，SYCL 为什么仍比原生慢？

注意，我们关心的不是"靠哪个参数能关掉这个事件"，而是"为什么在 AMD 和海光上贵"。关掉事件的办法上一阶段已经有了（第十一节会再总结）；这次要找的是事件本身贵的原因，以及其中不合理的设计。

二、预备知识：队列、屏障、信号和栅栏

这一节为不熟悉 GPU 运行时内部的读者准备。已经熟悉 AQL 和内存栅栏的读者可以跳到第三节。

2.1 GPU 怎么接收命令：队列、包、门铃

在 AMD 的 GPU 软件栈（HSA 架构）里，主机程序和 GPU 之间通过"用户态队列"交流。队列是一块环形缓冲区，每格放一个 64 字节的 **AQL 包**（Architected Queuing Language）。主机写好一个包，再往**门铃**（doorbell）写一个数，通知 GPU 上的**命令处理器**（Command Processor，下文简称 CP）来取。CP 按顺序处理包，主要有两类：

- **kernel 派发包**（dispatch）：让 GPU 启动一个 kernel。
- **屏障包**（barrier）：本身不做计算，只用来等待和同步。标准的 BARRIER_AND 包"等若干个信号都变成 0"；AMD 扩展的 BarrierValue 包"等一个信号满足某个条件"。

HIP 的 `hipLaunchKernel`、`hipEventRecord` 等调用，最终都翻译成往这个队列里写包。NVIDIA 也有类似的"命令缓冲区 + 门铃"，但格式不公开。

2.2 屏障位

每个包的 header 里有一个**屏障位**（barrier bit）。设了这一位的包，要等前面所有的包都执行完才开始。HIP 的 kernel 包都设了这一位，所以同一个队列里的 kernel 严格先后执行；屏障包也一定设这一位。因此，在两个 kernel 之间插一个屏障包，第二个 kernel 要等"第一个 kernel 结束 → CP 处理屏障包 → 屏障包完成"之后才能开始。**屏障包上的每一点工作，都直接加在相邻两个 kernel 的间隔上。**

2.3 完成信号

包可以带一个**完成信号**（completion signal）：放在内存里的一个 64 位整数。包执行完，CP 把它减 1。主机（例如 `hipEventSynchronize`）或者别的包（屏障包）可以等它变化。信号放在哪里很重要：

- 放在**主机内存**里：主机读起来快，但 GPU 要经 PCIe 去改它。实测每次要多 2.4～2.6 μs。
- 放在**显存**里：GPU 改起来快（约 1 μs），但主机要经 PCIe 去读，主机轮询会变慢。

信号还可以"带中断"：值变化时 GPU 给 CPU 发一个中断，叫醒睡眠中的等待线程。中断本身也有开销。

2.4 内存栅栏：acquire 和 release，设备范围和系统范围

GPU 有多级缓存。一个 kernel 写的数据可能还停在缓存里，别人不一定马上看得到。**栅栏**（fence）规定的是"什么时候、对谁可见"：

- **release**（发布）：把我之前的写入推到某个范围内大家都能看到的地方；
- **acquire**（获取）：作废我本地可能过期的缓存，保证之后读到别人已经发布的最新数据。

范围（scope）有两种：**设备范围**（agent scope）只对同一块 GPU 上的其他工作有效；**系统范围**（system scope）对主机 CPU 和其他 GPU 也有效。AQL 包的 header 里，acquire 和 release 各有一个字段，可以分别选"无 / 设备 / 系统"。acquire 在包开始前做，release 在包完成后、改完成信号之前做。

2.5 AMD 和 NVIDIA 的 L2 有什么不同

按 LLVM 文档对 GFX90A（MI210 的架构）内存模型的描述，本卡显存的数据可以以"脏"的形式留在 L2 里（MTYPE RW）。同一块卡上的所有计算单元共用一个 L2，所以**设备范围的 release 不需要写回**；但要让别的设备（包括经 PCIe 访问的主机）看到，必须先用 `buffer_wbl2` 把 L2 里的脏行写回显存。所以系统范围 release 的代价，和"kernel 写了多少、还有多少留在 L2 里"成正比。系统范围的 acquire 在着色器层面用 `buffer_invl2`，按文档只需要作废 MTYPE NC 的行（非本地内存），本地显存的行"由于探测机制，永远不会过期"。

NVIDIA 的 L2 是显存的一致性点：主机和其他设备经 PCIe 访问显存也经过 L2，所以系统栅栏不需要写回 L2。这一点是根据第六节的实测（栅栏代价不随脏数据变化）加上公开资料推断的，没有找到 NVIDIA 官方文档的明确表述。

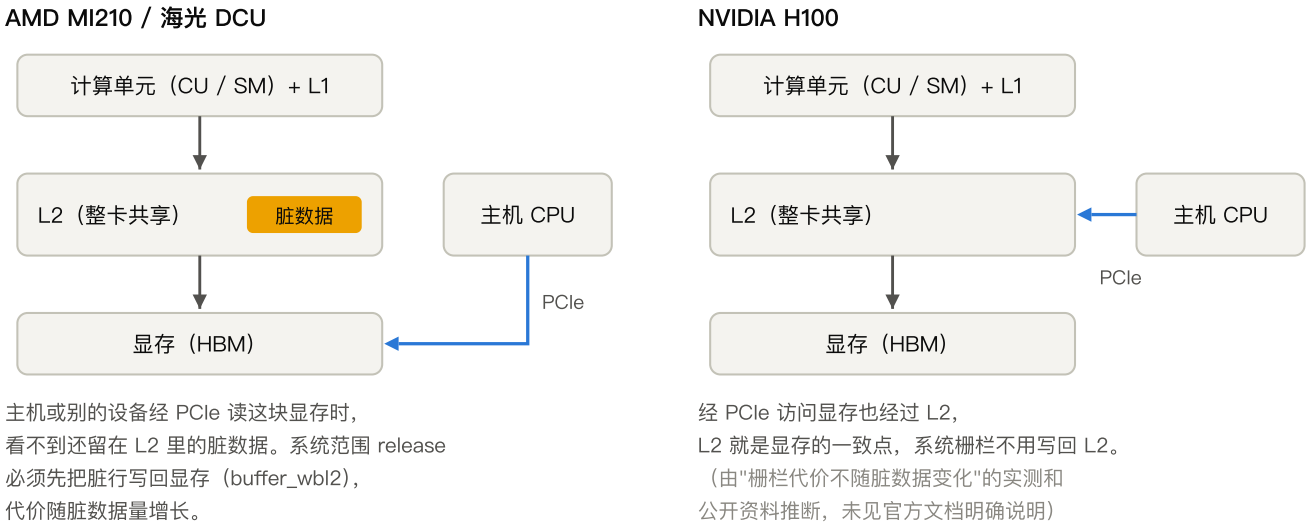


图 1　AMD 和 NVIDIA 的 L2 与主机访问路径（示意）。AMD 一侧依据 LLVM 的 GFX90A 内存模型；NVIDIA 一侧是推断。海光 DCU（gfx936）和 MI210 是同一类架构，第七节的实测也表现出同样的"系统 release 随脏数据增长"。

2.6 记录一个事件，最少需要做什么

事件的语义是：标记队列里的一个位置，以后可以查询或等待"这个位置之前的工作是否都完成了"。要实现它，最少需要：

1. 前面的工作完成后，改一个等待方能看到的值（完成信号）。
2. 如果等待方要直接读前面 kernel 写的数据（例如主机等完事件后直接读显存），这些数据要对等待方可见，需要一次范围合适的 release。**只有主机或其他设备要直接读时，才需要系统范围**；同一块 GPU 上的依赖只要设备范围，而 kernel 包结束时本来就带设备范围 release。

记录事件的一方**不需要 acquire**：acquire 是"读别人的数据之前"做的，应该由等待方在等到之后做。另外，同一个 in-order 队列里的后续 kernel 已经被屏障位排好了顺序，连事件都不需要。

三、调研方法

3.1 思路

1. **看清 HIP 放了什么。**读 ROCm 7.0 CLR（HIP 运行时）的源码，并用 `AMD_LOG_LEVEL=4` 抓下真实提交的每个 AQL 包的 header，两者对照。
2. **逐项拆分。**绕过 HIP，用 HSA 接口直接往队列里写同样的包。从"只有 kernel"开始，每次只加一样东西：屏障包、完成信号、信号放在哪、acquire、release、栅栏的范围，测每一项让每次迭代多出多少时间。再回到 HIP 层面，用真实的 HIP 调用核对。
3. **对照。**在 NVIDIA H100 上做功能对应的操作（不计时事件、计时事件、带系统栅栏往主机内存写一个值），看它便宜在哪里。在海光 DCU 上把 1、2 两步重做一遍。

3.2 测量程序

表 2　本次写的测量程序（都在 `why_amd/` 下）

程序	作用
`kernels.cl`	HSA 探针用的 kernel：空 kernel、写 n 字节（`wr`）、读 n 字节（`rd`）、按页跨步读（`rds`）、按 GPU 时钟空转（`spin`）、在 GPU 上记时间戳（`stamp`）。OpenCL C，不链接设备库，同一份源码可编 gfx90a（MI210）和 gfx936（DCU）。
`hsa_probe.cpp`	手工构造 AQL 包：kernel 包、BARRIER_AND、BarrierValue，acquire/release 范围任意组合。完成信号可以用 ROCr 建的，也可以自己在指定内存池（主机细粒度、显存粗粒度、显存细粒度）里构造。kernel 参数放显存。模式：主表与时间线、信号位置、写入量、读 kernel、冷热数据、插空 kernel、GPU 忙时提交。
`hip_probe.cpp`	HIP 层：启动加各种标志的 `hipEventRecord`、每次新建/销毁事件、`hipExtLaunchKernelGGL` 挂停止事件；输出每次调用时间、GPU 时间线、GPU 忙时的单次调用耗时、AQL 日志。
`cuda_probe.cu`	NVIDIA 对照：不计时/计时事件、新建–记录–销毁、`cuStreamWriteValue32` 带系统栅栏写显存或主机内存、不同写入量、GPU 时间线。
`../a100_why/` 下的 `native_prof.cu`、`sycl_prof.cpp`、`prof.h`、`gen_count_shim.py`	A100 上的提交循环：原生 CUDA 的几种启动方式和 SYCL 的两种提交；进程内采样（GPU50 上用不了 perf）；LD_PRELOAD 计数库，统计 UR 的 CUDA 适配层每次提交调用了哪些驱动函数（6.2 节）。
`counters_probe.cpp`	rocprof 硬件计数器（L2 命中、地址翻译）。计数器采集本身会在每个 kernel 前后清缓存，测不出差别，结果没有用上。

3.3 几种测法

- **每次迭代时间。**连续放 1000～4000 次迭代（kernel 越长次数越少），最后同步一次，总时间除以次数；7 轮取最好。和项目 harness 的计时方式相同。
- **只敲一次门铃**（`BATCH_DOORBELL=1`）。先把一轮的所有包写进队列，最后只敲一次门铃，GPU 连续处理，主机完全不参与，测到的是纯 GPU 时间。MI210 上和逐包敲门铃相差不超过 0.3 μs；DCU 上逐包敲门铃时主机提交接近 GPU 速度，所以 DCU 的主拆分以这种测法为准。
- **GPU 时间线。**每次迭代放一个只记时间戳的 kernel（MI210、DCU 用 `s_memrealtime`，25 MHz；H100 用 `%globaltimer`），相邻时间戳之差就是 GPU 上相邻 kernel 的间隔，取中位数和 p10/p90。
- **GPU 忙时的单次调用。**先启动一个在 GPU 上空转 2 ms 的 kernel，再计时单个 API 调用。GPU 还在忙，调用不可能在等 GPU 做完，测到的就是这个调用本身花的主机 CPU 时间。
- **冷热数据。**读 kernel 反复读同一块（数据在 L2 里），或者轮流读 32 MiB 里的不同块（数据不在 L2 里）。

- **写入量和读取量的加密扫描。**1～24 MiB 每 1 MiB 一个点，再加 28、32、48、64 MiB（MI210 在 2～10 MiB 另有 0.5 MiB 一个点的扫描），找 release 代价和 L2 命中的拐点。除了每个工作项读写一个 float4 的 kernel，还用了固定 256 个工作组、循环读写整块数据的 kernel，让工作组数不随数据量增长；冷数据在 512 MiB 里轮流取至少 64 块。
- **按页跨步读。**每 4 KiB 只读 16 字节，数据量只有 1/256，但每一页都要做地址翻译，用来区分"缓存的数据丢了"和"地址翻译丢了"。
- **插空 kernel。**在 acquire 和读 kernel 之间插 0～3 个空 kernel，看代价是后台慢慢做完的，还是下一次访存才付的。
- **交替测，并记录频率。**数据 kernel 的几组配置按轮次交替（每组的第 r 轮都跑完，再跑第 r+1 轮），避免频率或温度漂移只影响某一组。每行都记录 GPU 核心频率：MI210 全程 1700 MHz，DCU 全程 1500 MHz。
- **进程内采样（只用在 A100 上）。**提交循环连续跑 4 秒，定时器每 50 μs 打断一次提交线程，记下它当时停在哪个库、哪个函数里。样本比例乘以每次调用的时间，就是每次调用在各库里花的 CPU 时间。A100 上所有写法都是主机瓶颈，提交线程一直在忙，所以可以这样分摊。

3.4 平台

表 3　平台和软件版本

	MI210	海光 DCU	H100
型号	AMD Instinct MI210（gfx90a）	BW1000 64G（gfx936）	H100 80GB HBM3
L2	8 MiB	8 MiB	50 MiB
软件	ROCm 7.0.0（CLR `rocm-7.0.0` 标签）	DTK 26.04；HIP 是 `libgalaxyhip`（AMD HIP 6.3 的分支），HSA 运行时用计算节点上的 `/opt/hyhal`	驱动 580.178.04，CUDA 12.8 编译
机器	GPU62 card3，宿主机直接运行	齐物集群 bw1000 分区，Slurm 作业	齐物集群 gxn74；GPU70 card4（驱动 570）复测
频率	1700 MHz（全程）	1500 MHz（全程）	—

另外：A100 在 GPU50 card1（主机 CPU 为 AMD EPYC 7763，CUDA 12.0），表 1 是上一阶段的数字，6.2 节是这次在同一张卡、同一个容器里重测的。SYCL 实现：MI210 上是 DPC++（2026-10-14 的 intel/llvm），A100 上是 b9 A100 容器里的 DPC++（clang 21 开发版，libsycl.so.8），DCU 上是 AdaptiveCpp 24.10。MI210 本该在 b9 的容器里跑，但 GPU62 的 `/data1` 已满，Docker 起不了容器，所以用宿主机上同一版本的 ROCm 7.0.0 运行时，并用 `ROCR_VISIBLE_DEVICES` 固定在空闲的 card3 上。

四、HIP 往队列里放了什么

4.1 源码

看的是 `clr` 仓库的 `rocm-7.0.0` 标签：

- `hipamd/src/hip_event.cpp` 的 `Event::recordCommand` 只看 `hipEventDisableSystemFence` 一个标志：有它就用 `kCacheStateIgnore`（不加栅栏），没有就用 `kCacheStateInvalid`（默认）。`hipEventReleaseToDevice` 和 `hipEventReleaseToSystem` 读出来后再没有用到。
- `rocclr/device/rocm/rocvirtual.cpp` 的 `VirtualGPU::submitMarker`：`kCacheStateIgnore` 用 `kBarrierVendorPacketNopScopeHeader`，其他一律用 `kBarrierVendorPacketHeader`（MI200/MI300 上是厂商 BarrierValue 包，其他卡是 BARRIER_AND 包）。这些 header 的 acquire 和 release 都是 `HSA_FENCE_SCOPE_SYSTEM`。文件开头的注释写着，系统范围 release 会使 L1、L2 失效并写回 L2。

- kernel 包：`AMD_OPT_FLUSH=1`（默认）时用设备范围的 acquire/release，带屏障位，不带完成信号（header `0xb02`）。
- 信号：直接派发、主动等待的默认模式下，标记用的是 `hsa_amd_signal_create(..., HSA_AMD_SIGNAL_AMD_GPU_ONLY)` 建的不带中断的信号；要挂回调的标记才用带中断的信号。ROCr 的信号池用系统分配器给所有信号分配内存，也就是主机上的细粒度内存。
- 每个 HIP 队列建好后都调用 `hsa_amd_profiling_set_profiler_enabled(queue, 1)`，即所有队列都开着时间戳采样。

4.2 AQL 包日志

表 4　各种操作放进队列的包（MI210，`AMD_LOG_LEVEL=4`；`out/mi210_hip_aqllog.txt`、`out/mi210_sycl_aqllog.txt`）

操作	放进队列的包
`kernel<<<>>>` / `hipLaunchKernel`	Dispatch `0xb02`：屏障位，acquire=设备，release=设备，无信号
`hipEventRecord`（DisableTiming、计时、ReleaseToDevice、ReleaseToSystem、每次新建事件，全部一样）	Dispatch 之后加 BarrierValue `0x1500`：屏障位，**acquire=系统，release=系统**，带信号
`hipEventRecord`（DisableSystemFence）	BarrierValue `0x100`：无栅栏，带信号
`hipDeviceSynchronize`	BARRIER_AND `0x1503`：系统/系统，带信号
DPC++ `q.parallel_for`	每个 Dispatch 后一个 BarrierValue `0x1500`
DPC++ `nd_launch`	只有 Dispatch
`hipExtLaunchKernelGGL(..., stopEvent)`	没有单独的标记包，事件挂在 kernel 包的完成信号上；第一个 kernel 包是 `0x1502`（系统/系统），后面连续的都被降成 `0xb02`（设备/设备）

图 2 把这几种情况画在一起。和 2.6 节的"最少需要"相比，HIP 的事件多了三样东西：一个让队列停下来的独立屏障包、一道 acquire、以及不分场合的系统范围 release；而且信号放在主机内存里。



图 2　几种提交方式放进队列的东西（示意，长度不按比例）。颜色和图 3 一致：灰色是 kernel，蓝色是 CP 处理屏障包本身，绿色是系统 acquire，黄色是系统 release，橙色是经 PCIe 改主机内存里的完成信号。NVIDIA 一行是根据实测推断的行为，不是公开文档。

4.3 HIP 内部已有的"挂在 kernel 包上"的路径和懒栅栏

表 4 最后一行是 HIP 自己已有的路径：`hip_module.cpp` 里的 `BindCommand` 把停止事件直接挂在 kernel 包的完成信号上，不另插屏障包。连续的系统范围包被降级，是 `dispatchGenericAqlPacket` 的规则：前一个包已经是系统

release 时，下一个降成设备范围，同时记一笔 `fence_dirty_`，等 `hipStreamSynchronize`、`hipStreamQuery` 或队列 finish 时再补一个系统范围的屏障包。也就是说，**HIP 内部已经有"需要时再付系统栅栏"的懒做法，只是没有用在普通的 `hipEventRecord` 上**。DPC++ 和 AdaptiveCpp 都没有用这条路径，用的是"启动，再单独记录事件"。

这里有一个没有验证的疑点：`hipEventSynchronize` 只等事件的信号，不看 `fence_dirty_`。连续用 `hipExtLaunchKernelGGL` 绑定停止事件时，后面的 kernel 只有设备范围 release；主机等到这个事件后直接读显存（不经过拷贝）时，按内存模型可能看不到 L2 里的数据。见第十三节。

4.4 海光 DTK 的 HIP

DTK 的 HIP 库是 `libgalaxyhip`，从 AMD HIP 6.3 分出，运行时 `dlopen` 计算节点上的 `/opt/hyhal/lib/libhsa-runtime64.so`。它用自己的日志开关（`HIP_LOG_LEVEL`、`HIP_LOG_OUTPUT_PATH`），日志是 NanoLog 二进制格式，要用 DTK 自带的 `.hyhal/NanoLog/bin/hiplogdump sort` 解码；默认级别下不打印逐包 header。解码后的日志（`out/dtk_hiplog/`）显示：

- 每个命令都"建时间戳"（`profilingInfo enabled create the timestamp`），即时间戳采样开着；
- 每次 `hipEventRecord` 都插一个带完成信号的屏障包（`releaseGpuMemoryFence add barrier`、`barrier_packet_.completion_signal`），kernel 包的 header 是 `0xb02`，和 AMD 一样；
- 每个屏障包对应一行 `Batch handler done`（24 个屏障包，24 行），其中 21 行在另一个线程上，是信号触发后在主机上执行的回调；
- 设备信息里有 `Device merge barrier status 1`、`HIP_CUSTOM_AQL_BEHAVIOR value is 0x0` 这类 AMD 原版没有的项。

在 DCU 上用手工构造的同样的包计时，结果和 HIP 实测一致（第七节）。

五、MI210 逐项拆分

5.1 空 kernel：每一项各花多少

表 5 从"只有 kernel"开始，每次加一样东西。两列分别是逐包敲门铃和只敲一次门铃（纯 GPU 时间），增量按后一列算。

表 5　MI210，每次迭代的时间（μs，空 kernel；`out/mi210_hsa_.txt`、`out/mi210_hsa__batch.txt`）

每次迭代放进队列的东西	逐包敲门铃	只敲一次门铃	增量
kernel	1.32	1.33	
+ 屏障包（不带栅栏、不带信号）	1.97	1.98	+0.65
+ 完成信号（主机内存）	4.41	4.56	+2.58
同上，信号带中断	5.12	5.16	+0.60
屏障包 + 信号 + 设备范围 acquire	4.66	4.78	+0.22
屏障包 + 信号 + 设备范围 release	4.41	4.56	0.00
屏障包 + 信号 + 系统范围 release	4.75	4.96	+0.40
屏障包 + 信号 + 系统范围 acquire	6.29	6.09	+1.53
屏障包 + 信号 + 两道系统栅栏（= HIP 的事件，BarrierValue 包）	6.83	6.31	+4.98
同上，用 BARRIER_AND 包	6.61	6.29	+4.96
BarrierValue + 信号，不带栅栏（= DisableSystemFence）	4.35	4.48	+3.15
信号直接挂在 kernel 包上（不加屏障包）	3.83	3.94	+2.61
kernel 包自己带系统 release	1.53	1.54	+0.21
kernel 包自己带系统 acquire	2.19	2.23	+0.90
两个 kernel	2.64	2.66	+1.33

增量的对照：第 2 行相对第 1 行，第 3、4 行相对上一行，第 5～8 行相对第 3 行（屏障包 + 信号、不带栅栏），其余相对第 1 行。逐包敲门铃时，主机每次迭代写包加敲门铃只花 0.3～0.8 μs，所以两列都是 GPU 处理命令的时间。

逐项看：

- **屏障包本身 0.65 μs。**不带栅栏、不带信号，只是让 CP 多处理一个包，并让下一个 kernel 等它。作为对比，多跑一个空 kernel 是 1.32 μs。
- **主机内存里的完成信号 2.4～2.6 μs，是最大的单项。**CP 要经 PCIe 去改主机内存里的值，并等这次写完成，才算屏障包完成。信号带中断再多 0.6～0.7 μs。
- **设备范围的栅栏几乎不花钱**（acquire +0.2～0.25，release 0）。
- **系统范围 release 在空 kernel 上只多 0.3～0.4 μs**：没有脏数据要写回。它的代价随 kernel 写的数据量增长，见 5.3 节。
- **系统范围 acquire 在空 kernel 上多 1.5～1.9 μs**；下一个 kernel 要访存时合计多 4.9～6.8 μs，比空 kernel 时多出约 3～5 μs，见 5.4 节。
- 两道系统栅栏都加上，就是 HIP 事件放进队列的东西：6.3～6.8 μs，比只有 kernel 多 5.0～5.5 μs，和 HIP 层面的实测（6.1 μs，5.5 节）一致。

开不开队列的时间戳采样，各行相差不超过 0.15 μs。HIP 的每个队列都开着采样，所以 `hipEventDisableTiming` 设不设，放进队列的包和花的时间都一样。

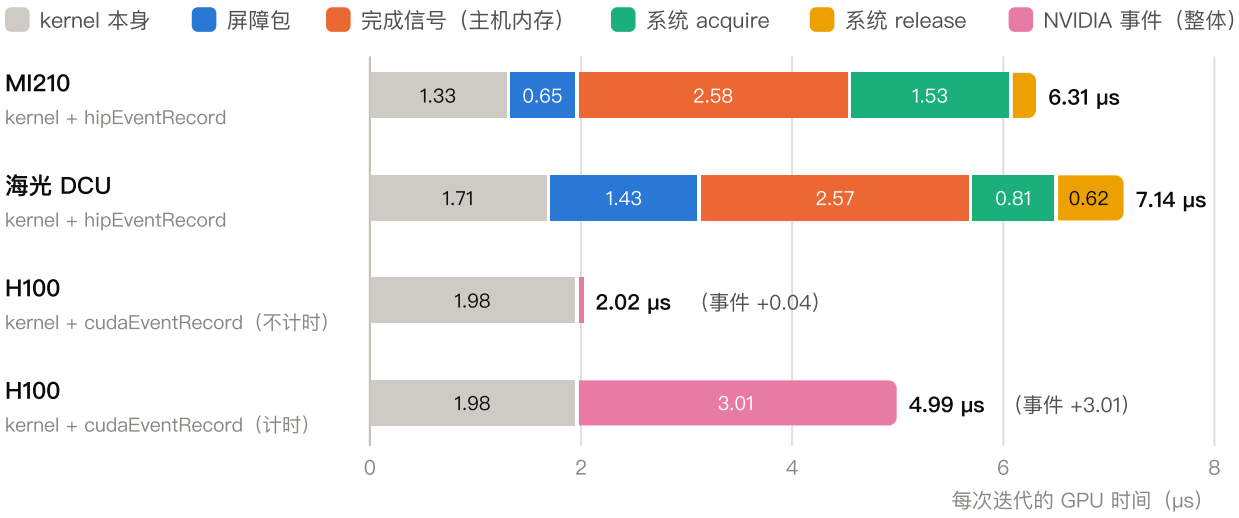


图 3　一次事件记录在 GPU 上的时间构成（空 kernel）。MI210 和 DCU 是只敲一次门铃时每次迭代的 GPU 时间，按"kernel → 加屏障包 → 加主机内存信号 → 加系统 acquire → 再加系统 release"逐项累加，最后一段的右端就是 HIP 事件的实测总时间；H100 是 GPU 时间线上相邻 kernel 的间隔。数值见表 5、表 12、表 15。

5.2 完成信号放在哪里

ROCr 建的信号，包括声明了"只有 GPU 用"的（`HSA_AMD_SIGNAL_AMD_GPU_ONLY`），都放在主机内存里，和自己建在主机内存里的信号一样慢。把同样的信号块放到显存（粗粒度），挂在 kernel 包上只多 0.9 µs（主机内存 2.5 µs）；屏障包 + 信号多 1.6 µs（主机内存 3.1 µs）。

表 6　信号位置的影响（µs；`out/mi210_hsa_sig.txt`、`out/dcu_hsa_sig.txt`，逐包敲门铃）

每次迭代（µs）	MI210	海光 DCU
只有 kernel	1.33	1.69
信号挂在 kernel 包上		
ROCr 建的"只给 GPU 用"的信号（实际在主机内存）	3.95	4.17
自建信号，主机细粒度内存	3.84	4.13
自建信号，显存粗粒度	2.25	2.74
自建信号，显存细粒度	—	3.12
屏障包（不带栅栏）+ 信号		
ROCr 的信号（主机内存）	4.55	5.62
自建信号，显存粗粒度	2.92	4.16
屏障包 + 两道系统栅栏 + 信号（= HIP 的事件）		
ROCr 的信号（主机内存）	6.71	6.94
自建信号，显存粗粒度	5.60	5.64

"自建信号"是测量手段：在指定内存池里按 `amd_signal_t` 的布局自己构造信号块，不是 ROCr 支持的用法。放在粗粒度显存里的信号，主机经 PCIe 直接读可能看不到最新值，只适合只由 GPU 等待的事件。

5.3 系统 release 和 L2 里的脏数据

让 kernel 先写一段数据，再看系统 release 比不加栅栏多多少（设备范围 release 和不加栅栏实测完全一样）。写入量从 1 MiB 到 64 MiB，每 1 MiB 一个点。同时用读 kernel 比较"反复读同一块"和"轮流读不同的块"，看 L2 能留住多少数据。

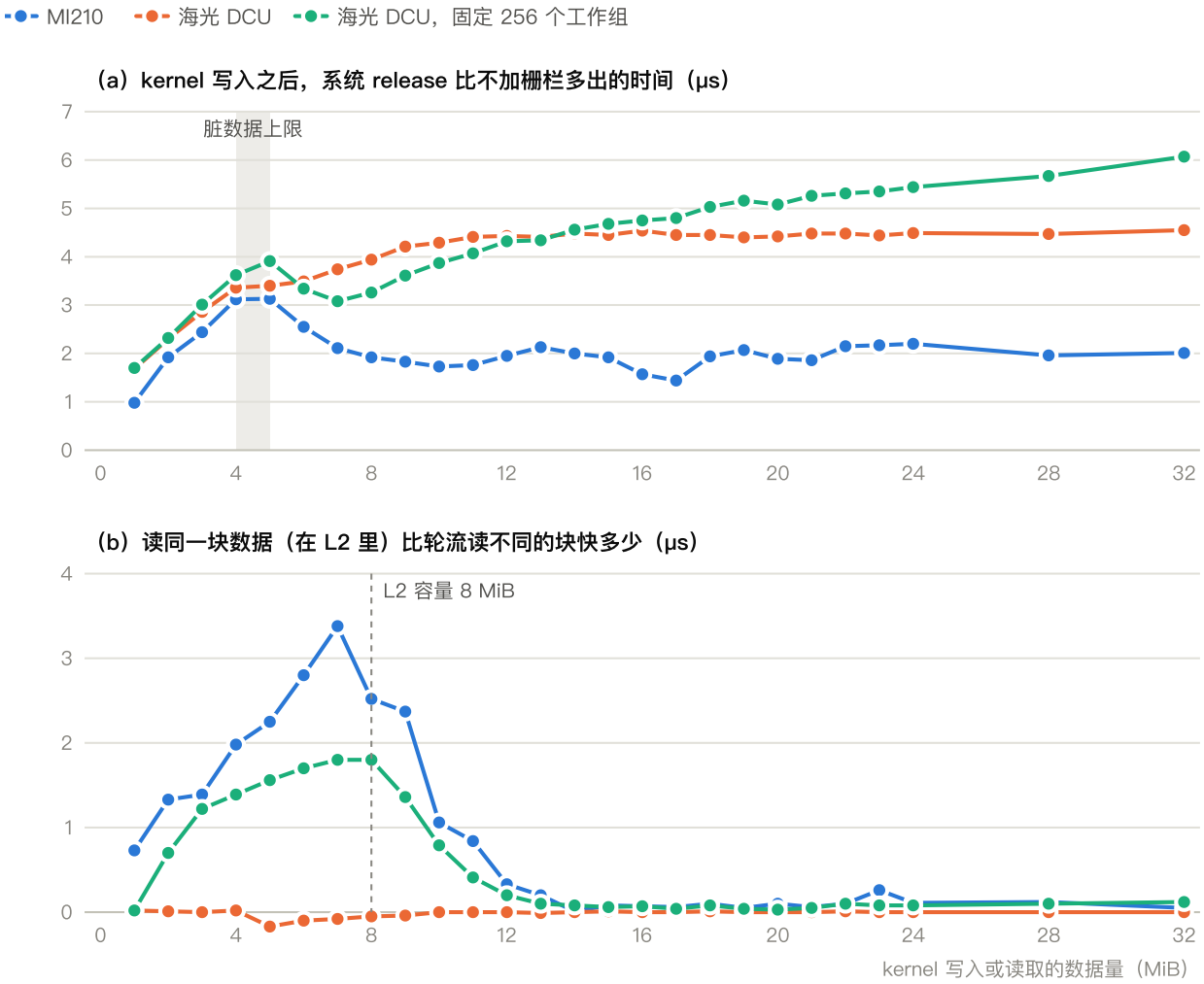


图 4　系统 release 的代价随写入量的变化，以及 L2 能留住多少数据。（a）纵轴是"屏障包 + 信号 + 系统 release"比"屏障包 + 信号"多出的时间（逐包敲门铃）。两块卡都在 4～5 MiB 处到顶：MI210 到 4 MiB 为止每 MiB 约 0.7 μs，接近显存写带宽，4～5 MiB 时 3.1 μs，之后回落到 1.7～2.2 μs；DCU 在 4～6 MiB 停在 3.4～3.5 μs，然后又慢慢涨到 4.4 μs。（b）两块卡在 7～8 MiB 附近收益最大，12～13 MiB 以后基本没有，和 8 MiB 的 L2 容量一致。DCU 上每个工作项一个 float4 的读 kernel 卡在工作组派发上，数据在不在 L2 里都一样快（橙线贴着 0），所以另画了固定 256 个工作组的曲线。48、64 MiB 见表 7。

表 7　系统 release 比不加栅栏多出的时间，以及数据在 L2 里时读快多少（μs；`out/*_hsa_wfine*.txt`、`out/*_hsa_rfine*.txt`）

数据量	MI210 release 多	DCU release 多	DCU（固定工作组）release 多	MI210 在 L2 里快	DCU（固定工作组）在 L2 里快
1 MiB	+0.98	+1.68	+1.70	0.73	0.02
2 MiB	+1.92	+2.30	+2.32	1.33	0.70
3 MiB	+2.44	+2.86	+3.01	1.39	1.22
4 MiB	+3.12	+3.36	+3.62	1.98	1.39
5 MiB	+3.13	+3.40	+3.91	2.25	1.56
6 MiB	+2.55	+3.49	+3.34	2.80	1.70
7 MiB	+2.11	+3.74	+3.08	3.38	1.80
8 MiB	+1.92	+3.94	+3.26	2.52	1.80
10 MiB	+1.73	+4.29	+3.87	1.06	0.79
12 MiB	+1.95	+4.43	+4.32	0.33	0.20
16 MiB	+1.57	+4.54	+4.75	0.07	0.07
24 MiB	+2.20	+4.49	+5.44	0.11	0.08
32 MiB	+2.01	+4.55	+6.07	0.05	0.12
64 MiB	+1.69	+4.46	+6.51	0.05	0.12

这就是 L2 写回：LLVM 的 GFX90A 内存模型说，系统范围 release 要用 `buffer_wbl2` 写回 MTYPE RW（本地粗粒度显存）和 NC 的脏 L2 行；设备范围不需要，因为同一块卡的所有工作组共用一个 L2。这笔钱只有在主机或别的设备要直接读这块显存时才有必要付。SYCL in-order 队列里 kernel 之间的依赖都在同一块卡上，不需要它。

图 4 还说明了三件事：

- **两块卡的 L2 都只留得住约 4～5 MiB 脏数据，约为 8 MiB 容量的一半多。**同样大小的地址范围，读能留住约 8 MiB（图 4b），写只留得住一半，所以不是容量或地址映射的问题，而是 L2 的写回策略：脏数据超过上限后，在 kernel 运行期间就提前写回显存。MI210 上写 kernel 自己也从 5 MiB 起变慢，每 MiB 从约 0.4 μs 变成约 1.3 μs；按 0.5 MiB 的网格，release 在 4.5 MiB 时最高（3.32 μs）。这条策略没有公开文档：Linux amdgpu 驱动在 MI200 上不设置 L2 的控制寄存器 `TCC_CTRL`（其中有一个含义没有公开的 `WRITEBACK_MARGIN` 字段），ROCr 和 LLVM 也不涉及；AMD 的性能计数器文档只说写回有一部分出于"其他内部硬件原因"。
- **超过上限之后，两块卡不一样。**MI210 在 kernel 运行期间写回得更多，结束时剩下的脏数据更少，release 反而便宜（回落到 1.7～2.2 μs）。DCU 不回落，之后还会慢慢涨：每个工作项一个 float4 的 kernel 从 7 MiB 涨到 11 MiB，然后持平在 4.4 μs；固定 256 个工作组的 kernel 一直涨到 48 MiB 的 6.6 μs。这一段不会是 L2 里的脏数据（已经封顶），推测是已经离开 L2、还没写进显存的写请求，系统 release 要等它们完成；没有验证。
- **DCU 上，每个工作项一个 float4 的 kernel 测不出 L2。**这时读和写、数据在不在 L2 里，1～64 MiB 每一点耗时都一样，每 MiB（256 个工作组）0.85 μs，说明卡在工作组派发上。固定 256 个工作组以后，数据在 L2 里的读快 0.7～1.8 μs（图 4b）。表 9 里 DCU"热数据和冷数据一样快"也是这个原因（7.2 节）。

5.4 系统 acquire 的后续罚时

空 kernel 上系统 acquire 自身多 1.9 μs。换成读数据的 kernel，代价大得多：

表 8　MI210，读数据的 kernel（μs；`out/mi210_hsa_l2.txt`、`out/mi210_hsa_cold.txt`）

MI210，读 kernel 每次迭代（μs）	64 KiB	1 MiB	4 MiB	16 MiB
只有 kernel（数据在 L2 里）	1.31	1.44	2.40	13.09
只有 kernel（数据不在 L2 里，轮流读 32 MiB 里的不同块）	1.49	2.17	4.34	—
+ 屏障包 + 信号（不带栅栏）	4.68	5.00	6.04	17.16
+ 屏障包 + 信号 + 设备范围 acquire	4.94	5.24	6.30	17.39
+ 屏障包 + 信号 + 系统范围 release	5.05	5.35	6.41	17.49
+ 屏障包 + 信号 + 系统范围 acquire	9.58	10.47	12.78	20.80
+ HIP 的事件（两道系统栅栏）	10.42	11.71	13.48	22.24
只有 kernel，kernel 包自己带系统 acquire	5.81	6.36	8.75	17.35

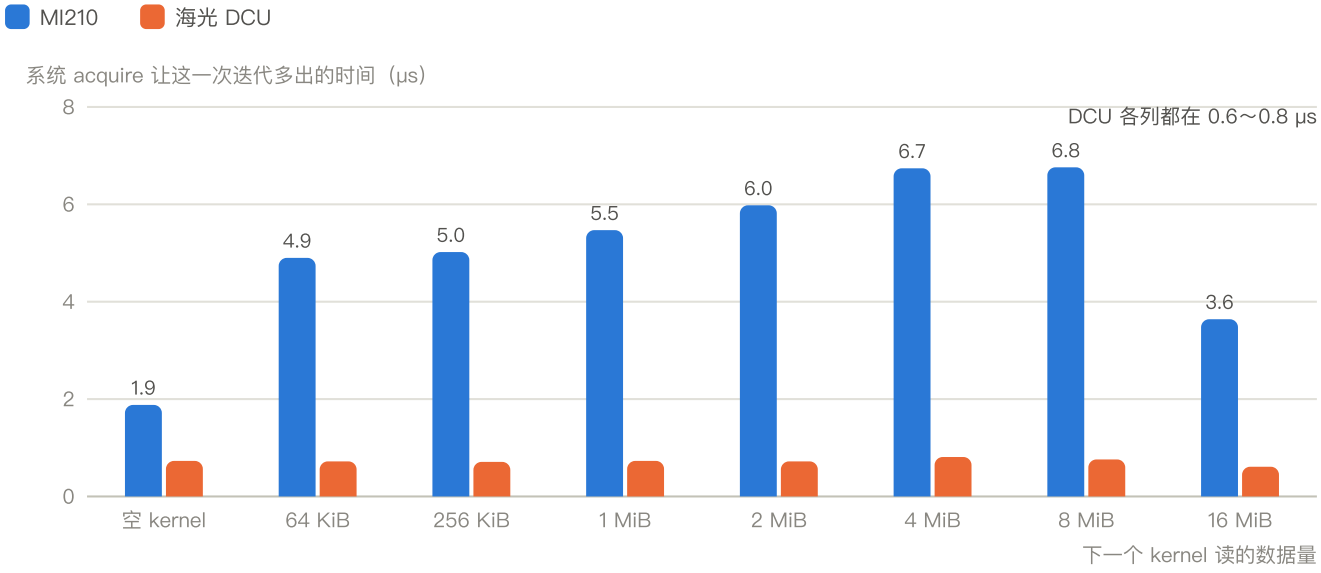


图 5　系统 acquire 让每次迭代多出的时间。"屏障包 + 信号 + 系统 acquire"减去"屏障包 + 信号"（逐包敲门铃）。MI210 上，读数据的 kernel 多 4.9～6.8 μs（16 MiB 时 3.6 μs），比空 kernel 的 1.9 μs 多出约 3～5 μs；DCU 上无论读多少都只多 0.6～0.8 μs。

访存 kernel 上多出来的这 3～5 μs"罚时"是什么？下面几组对照逐个排除：

- **不是数据被清出 L2 的代价。**数据本来就不在 L2 时（轮流读不同的块），读 kernel 只比数据在 L2 里慢 0.2～1.9 μs。而 acquire 之后，数据原本在不在 L2，时间都一样；在"数据不在 L2"的基础上，acquire 还要再多 4.5～4.8 μs（表 9）。
- **和读多少数据基本无关。**按页跨步、每 4 KiB 只读一个缓存行（数据量只有 1/256），也多 5.3～6.3 μs（表 10）。
- **不是后台还在做的操作。**在 acquire 和读 kernel 之间插 0～3 个空 kernel，额外开销仍是 5.7～6.2 μs（表 10）。如果是 acquire 触发了一个后台操作，插的空 kernel 会把它掩盖掉；实际没有，说明它是下一次访存时才付的固定代价。

表 9　热数据（在 L2 里）和冷数据（不在 L2 里）（μs；`out/mi210_hsa_cold.txt`、`out/dcu_hsa_cold.txt`）

数据量	kernel（热）	kernel（冷）	加屏障包与信号（热）	（冷）	再加系统 acquire（热）	（冷）	acquire 多（冷）
MI210							
64 KiB	1.33	1.49	4.79	5.59	10.14	10.09	+4.50
1 MiB	1.45	2.17	5.07	5.95	10.77	10.78	+4.83
4 MiB	2.43	4.34	6.18	8.38	13.12	13.11	+4.73
海光 DCU							
64 KiB	1.73	1.73	5.61	5.60	6.34	6.34	+0.74
1 MiB	2.22	2.23	5.60	5.60	6.34	6.34	+0.74
4 MiB	4.78	4.79	6.43	6.55	7.31	7.31	+0.76

表 10　按页跨步读、在 acquire 后插空 kernel（μs；`out/*_hsa_l2.txt`、`out/*_hsa_acq.txt`）

系统 acquire 让每次迭代多出的时间（μs）	MI210	海光 DCU
按页跨步读，范围 4 MiB（只读 1/256 的数据）	+5.27	+0.70
按页跨步读，范围 16 MiB（只读 1/256 的数据）	+5.57	+0.70
按页跨步读，范围 64 MiB（只读 1/256 的数据）	+6.29	+1.45
读 1 MiB；acquire 和读 kernel 之间插 0 个空 kernel	+5.70	+0.73
读 1 MiB；acquire 和读 kernel 之间插 1 个空 kernel	+5.83	+1.04
读 1 MiB；acquire 和读 kernel 之间插 2 个空 kernel	+6.16	+1.06
读 1 MiB；acquire 和读 kernel 之间插 3 个空 kernel	+5.85	+1.08

推断（没法直接验证）：CP 执行的系统范围 acquire 把整个 L2 作废了，下一个 kernel 的代码、参数、页表项都要从显存一级一级重新取，所以第一次访存要多付几 μs，和读多少数据无关。LLVM 的内存模型说，着色器层面的系统 acquire（`buffer_invl2`）只需要作废 MTYPE NC 的行，本地粗粒度显存的行不会过期；CP 实际做的比这多得多。用 rocprof 采计数器想验证，但计数器采集本身会在每个 kernel 前后清缓存，测不出差别。

AMD 自己的头文件对系统栅栏的描述是：会带来"缓存写回和失效的开销，以及这些操作对后续工作执行的影响"。实测正是如此，而且后一项（对后续工作的影响）在 MI210 上是最大的一块。

5.5 HIP 层面的核对

表 11　HIP 层面（每次调用和 GPU 时间线上的相邻 kernel 间隔，μs；`out/mi210_hip.txt`、`out/dcu_hip.txt`）

HIP 层面的写法	MI210 每次调用	MI210 GPU 间隔	DCU 每次调用	DCU GPU 间隔
只启动 kernel	1.46	2.40	1.93	2.40
+ hipEventRecord（DisableTiming）	6.11	8.00	7.63	7.68
+ hipEventRecord（计时）	6.10	8.00	7.63	7.68
+ hipEventRecord（DisableSystemFence）	4.41	5.92	6.18	6.24
+ hipEventRecord（ReleaseToDevice）	6.10	8.00	7.62	7.68
+ hipEventRecord（ReleaseToSystem）	6.11	8.00	7.62	7.68
每次新建、记录、销毁事件	6.11	8.00	7.62	7.68
hipExtLaunchKernelGGL 挂停止事件	3.87	3.84	4.78	4.80
同上，停止事件带 DisableSystemFence	3.86	3.84	4.99	4.80

- MI210 上启动加 `hipEventRecord` 每次 6.1 µs，GPU 时间线上相邻 kernel 的间隔从 2.40 变成 8.00 µs，和手工构造的 BarrierValue 包一致（HSA 层时间线 8.3～8.5 µs）。
- `ReleaseToDevice`、`ReleaseToSystem`、计时与否、每次新建事件，和默认完全一样，与 4.1 节的源码一致。
- `DisableSystemFence` 去掉两道栅栏，每次 4.41 µs，GPU 间隔 5.92 µs。
- `hipExtLaunchKernelGGL` 挂停止事件：每次 3.87 µs，GPU 间隔 3.84 µs，比只启动多 1.4 µs，比"启动再记录"少 4.2 µs。
- 主机端：GPU 正忙时，`hipEventRecord` 1.6 µs 就返回（表 16），直接写包加敲门铃只要 0.16 µs。MI210 上贵的是 GPU 端，不是主机端。

> **小结（MI210）。**空 kernel 上，一次事件让 GPU 多花 5.0～5.5 µs：主机内存信号 2.4～2.6、系统 acquire 1.5～1.9、屏障包 0.65、系统 release 0.2～0.4。真实的读 kernel 上，acquire 合计多 4.9～6.8 µs，整个事件每次多 9～11.6 µs；写 kernel 上，release 的写回最多再多约 3 µs。

六、NVIDIA 对照

这一节分两部分：6.1 节看 NVIDIA 的事件为什么便宜；6.2 节回答表 1 里的另一个现象，即 A100 上事件便宜，SYCL 为什么仍比原生慢。

6.1 NVIDIA 的普通事件做了什么

在集群的 H100 上做功能对应的操作（`cuda_probe.cu`，`out/h100_cuda.txt`）。

表 12　H100，每次迭代（µs）

H100（驱动 580）每次迭代	主机计时（µs）	GPU 上相邻 kernel 间隔（µs）
只启动 kernel	1.63	1.98
+ cudaEventRecord（DisableTiming）	1.69	2.02
+ cudaEventRecord（计时）	4.76	4.99
+ cuStreamWriteValue32，带系统栅栏，写显存	4.62	4.67
+ cuStreamWriteValue32，带系统栅栏，写主机内存	5.45	5.60
每次新建、记录、销毁事件（DisableTiming）	2.03	—
两个 kernel	3.24	—

表 13　H100，写数据的 kernel 后面加事件或带栅栏的写值，多出的时间（µs）

kernel 写入量	kernel 本身（µs）	+ 不计时事件，多	+ 带系统栅栏写主机内存，多
64 KiB	1.83	+0.01	+3.55
1 MiB	2.07	+0.01	+3.59
4 MiB	2.72	0.00	+3.70
16 MiB	5.93	0.00	+3.65
64 MiB	22.81	−0.01	+3.93

GPU70 上的 H100（驱动 570）：只启动 2.05 µs，加不计时事件 2.44 µs；上一阶段的 A100：2.52 → 2.90 µs。不计时事件只在主机端多 0.06～0.4 µs。CUDA 12 拒绝 `CU_STREAM_WRITE_VALUE_NO_MEMORY_BARRIER`，所以没有"不带栅栏写值"这一项。

解读：

- **NVIDIA 的普通事件不让流停下来。**GPU 时间线上，加不计时事件后相邻 kernel 的间隔只从 1.98 变成 2.02 μs；kernel 写 64 MiB 时也一样。它更像一个"做完了就记一笔"的完成报告，后面的 kernel 不用等它。
- **NVIDIA 做同样"重"的事也不便宜。**需要"等前面做完才能动"的操作，NVIDIA 也要 3～4 μs：计时事件（完成时要取时间戳）多 3.0 μs；带系统栅栏往主机内存写一个值多 3.6 μs（写显存 2.7 μs）。这和 AMD 的"屏障包 + 主机内存信号"3.1 μs 是一个量级。
- **NVIDIA 的系统栅栏不随脏数据增长。**kernel 写 64 KiB 到 64 MiB，带栅栏写值始终多 3.5～3.9 μs。说明 NVIDIA 的 L2 不需要写回（2.5 节）。
- **NVIDIA 的事件记录不做 acquire。**

所以 NVIDIA 便宜，不是因为它的硬件做同样的事更快，而是因为它的普通事件根本不做那些事：不阻塞流、不做 acquire、不写回缓存。

6.2 A100 上 SYCL 为什么仍比原生慢：运行时自己的 CPU 开销

表 1 里，A100 上原生启动加一次事件记录只多 0.4 μs，SYCL 却不管返不返回事件都比原生慢：不返回事件多 0.5 μs，返回事件比"原生 + 记录"多 0.9 μs。为了查清这笔钱花在哪里，在 GPU50 的空闲 A100（card1）上，用和表 1 相同的容器和 DPC++，把几种原生启动方式和两种 SYCL 提交各连续跑 4 秒，用进程内采样（3.3 节）统计 CPU 时间落在哪个库里；再用一个 LD_PRELOAD 计数库，统计 UR 的 CUDA 适配层每次提交调用了哪些驱动函数（`launch_overhead/a100_why/`）。

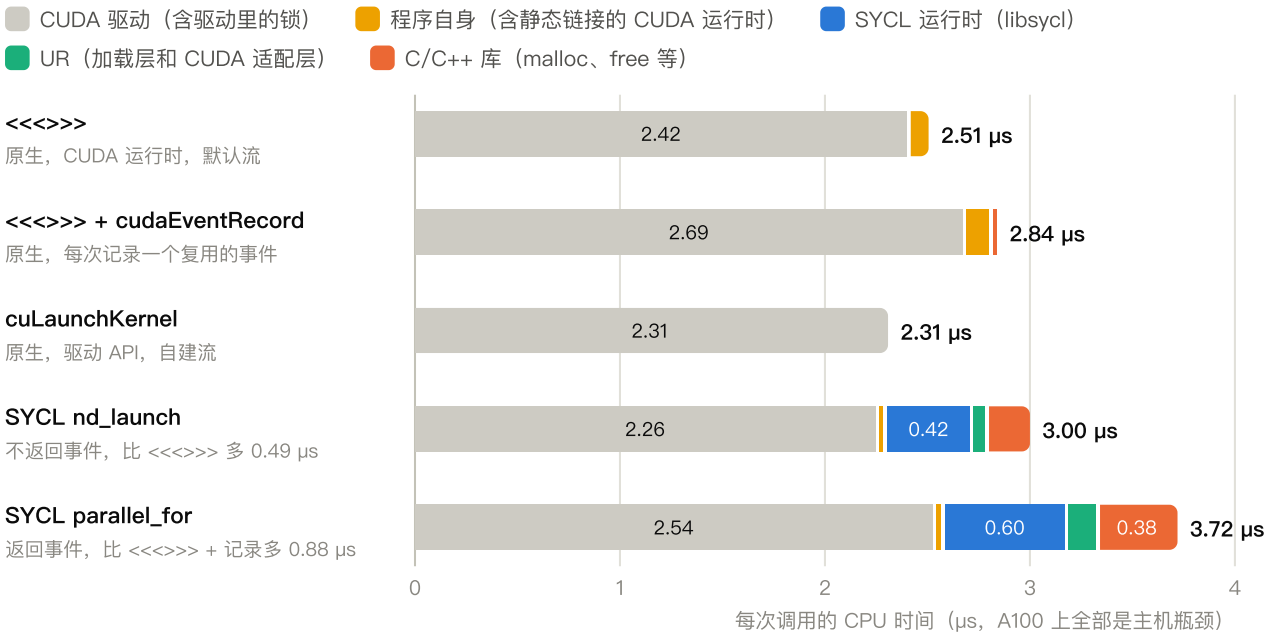


图 6　A100 上每次提交的 CPU 时间花在哪里。每条的总长是每次调用的时间（7 轮取最好；所有写法都是主机瓶颈，每次调用的时间等于主机提交一次的时间），各段按采样落在各库里的比例分摊。"CUDA 驱动"包括驱动调用的 `pthread_mutex_lock`。数值见表 14。

表 14　A100，每次调用的 CPU 时间按库分摊（μs；`a100_why/out/`）

写法	每次调用	CUDA 驱动	程序自身	libsycl	UR	C/C++ 库
<<<>>>，CUDA 运行时，默认流	2.51	2.42	0.09	0.00	0.00	0.00
<<<>>>，自建非阻塞流	2.39	2.30	0.09	0.00	0.00	0.00
cuLaunchKernel，驱动 API，自建流	2.31	2.31	0.00	0.00	0.00	0.00
<<<>>> + cudaEventRecord（复用一个事件）	2.84	2.69	0.13	0.00	0.00	0.02
cuLaunchKernel + cuEventRecord（复用一个事件）	2.43	2.42	0.00	0.00	0.00	0.01
<<<>>> + 每次建、记录、销毁事件	3.13	2.84	0.17	0.00	0.00	0.11
cuLaunchKernel + 每次建、记录、销毁事件	2.71	2.62	0.00	0.00	0.00	0.09
SYCL nd_launch（不返回事件）	3.00	2.26	0.04	0.42	0.07	0.20
SYCL q.parallel_for（返回事件）	3.72	2.54	0.04	0.60	0.15	0.38

"程序自身"对原生写法主要是静态链接进程序的 CUDA 运行时（`<<<>>>` 走的那一层），对 SYCL 写法是头文件里内联进来的提交代码。"C/C++ 库"是 libc、libstdc++ 和动态链接器，主要是 `malloc` / `free` 和线程局部变量的访问。

- **A100 上提交是 CPU 瓶颈。**单是一次启动，在 CUDA 驱动里就要 2.3～2.4 μs CPU（原生也一样），而 GPU 跑一个空 kernel 不到 1 μs（上一阶段测的 command graph 每个 kernel 0.98 μs）。GPU 一直在等提交，主机每多花 0.1 μs，每次调用就多 0.1 μs。
- **DPC++ 调驱动的次数已经最少。**每次 `nd_launch`：`cuLaunchKernel` 1 次、`cuCtxGetCurrent` 2 次；每次 `parallel_for` 再加 `cuEventCreate`、`cuEventRecord`、`cuEventDestroy` 各 1 次。它直接调驱动 API、提交到自建的非阻塞流，在驱动里花的时间（2.27 μs）比 `<<<>>>` 经 CUDA 运行时提交到默认流（2.42 μs）还少。
- **多出来的是 SYCL 运行时自己的代码。**不返回事件时每次约 0.70 μs：libsycl 0.42（构造和析构 handler、建 command group 对象、在程序缓存里查 kernel，以及即使不返回事件也会建的内部 `event_impl`），UR 0.08，C/C++ 库 0.20。返回事件时约 1.13 μs，多出的 0.44 μs 是 SYCL 和 UR 自己的事件对象（新建、引用计数、调用方丢掉时释放）。没有单个热点，最热的函数各只占 1% 左右。
- **事件在驱动里的代价，两边几乎一样。**UR 每次"建、记录、销毁"一个 CUDA 事件，驱动里多 0.28 μs；原生每次在默认流上记录一个复用的事件，驱动里多 0.27 μs。所以返回事件多出来的那部分，几乎全在 SYCL 和 UR 自己的事件对象上，不在 CUDA 驱动里。
- **净差值。**`nd_launch` 比 `<<<>>>` 多 0.49 μs：SYCL 侧代码多 0.69，驱动路径和程序自身合计省 0.20。`parallel_for` 比"`<<<>>>` + 记录"多 0.88 μs：SYCL 侧多 1.12，省 0.24。
- **MI210 上也有这笔钱，只是被盖住了。**上一阶段的原始输出（`launch_overhead/results/mi210_card3.out`）里，MI210 原生启动的主机端每次 1.15 μs，`nd_launch` 1.57 μs，同样多约 0.4 μs CPU。但 MI210 跑一个空 kernel 在 GPU 上要 1.33 μs（表 5），主机多花的时间大部分藏在 GPU 时间后面，每次调用只多 0.25 μs（1.32 → 1.57）；加了事件后 GPU 端每次 6 μs，主机端的差别完全看不见。

所以表 1 里"MI210 上 SYCL 和原生差不多，A100 上差得多"，是两件事叠在一起：SYCL 运行时每次提交的 CPU 代价（不返回事件约 0.7 μs，返回事件约 1.1 μs）在各平台都有，只在提交卡在 CPU 上的平台看得见；AMD 和海光上真正贵的是 SYCL 强加的那次事件记录，花在 GPU 端。两件事的修法也不同，见 11.2 节。

七、海光 DCU

同样的探针在 DCU 上跑（`run_dcu_hsa.sbatch`、`run_dcu_batch.sbatch`、`run_dcu_busy.sbatch`、`run_dcu_acpp_sig.sbatch`）。DTK 的 HIP 运行时 `dlopen` 计算节点上的 `/opt/hyhal/lib/libhsa-runtime64.so`；

DTK 目录里 `.hyhal/lib` 那一份在作业里看不到 GPU，`lib/libhsa-runtime64.so` 是断链（`diag_dcu.sbatch`）。HSA 探针要链接 `/opt/hyhal/lib`，而且 `LD_LIBRARY_PATH` 里不能有 `$DTK/.hyhal/lib`。

7.1 GPU 端：每次 5.4 μs

表 15　海光 DCU，每次迭代的时间（μs，空 kernel；`out/dcu_hsa_.txt`、`out/dcu_hsa_batch.txt`）

每次迭代放进队列的东西	逐包敲门铃	只敲一次门铃	增量
kernel	1.69	1.71	
+ 屏障包（不带栅栏、不带信号）	3.12	3.14	+1.43
+ 完成信号（主机内存）	5.62	5.71	+2.57
同上，信号带中断	6.21	6.30	+0.59
屏障包 + 信号 + 设备范围 acquire	5.63	5.71	0.00
屏障包 + 信号 + 设备范围 release	5.62	5.71	0.00
屏障包 + 信号 + 系统范围 release	6.20	6.30	+0.59
屏障包 + 信号 + 系统范围 acquire	6.35	6.52	+0.81
屏障包 + 信号 + 两道系统栅栏（= HIP 的事件，BarrierValue 包）	6.85	7.14	+5.43
同上，用 BARRIER_AND 包	6.94	7.14	+5.43
BarrierValue + 信号，不带栅栏（= DisableSystemFence）	5.42	5.53	+3.82
信号直接挂在 kernel 包上（不加屏障包）	4.17	4.27	+2.56
kernel 包自己带系统 release	2.28	2.30	+0.59
kernel 包自己带系统 acquire	2.44	2.44	+0.73
两个 kernel	3.38	3.42	+1.71

增量的对照同表 5。

- **屏障包本身比 MI210 贵一倍多**（1.43 对 0.65 μs）；两个空 kernel 连着跑也要 3.42 μs（MI210 2.66 μs）。DCU 的命令处理器处理每个包都更慢。
- **主机内存信号 2.57 μs**，和 MI210 相当；放显存（粗粒度）挂在 kernel 包上只多 1.0 μs（表 6）。
- **两道系统栅栏合计 1.4 μs**：release 0.59，acquire 0.81（单独加时）。
- 合计 7.14 μs，比只有 kernel 多 5.43 μs。

7.2 栅栏：release 同样随脏数据增长，acquire 没有后续罚时

- **系统 release 随脏数据增长**（图 4、表 7）：写 1 MiB 多 1.7 μs，4 MiB 多 3.4 μs，4～6 MiB 停在 3.4～3.5 μs，这是脏数据上限，和 MI210 一样在 4～5 MiB；之后又慢慢涨到 4.4 μs 持平，固定 256 个工作组写大数据时涨到 6.6 μs。没有 MI210 那样超过上限后的回落。
- **系统 acquire 没有 MI210 那种后续罚时**（图 5）：读 kernel 在 acquire 之后只多 0.6～0.8 μs，和读多少无关；按页跨步读、插空 kernel 也只多 0.7～1.5 μs（表 10）。
- **表 9 里 DCU 上同一块数据连着读和每次读新数据一样快，是因为探针的读 kernel 在 DCU 上卡在工作组派发上**（5.3 节），不是 L2 没有复用：固定 256 个工作组以后，数据在 L2 里能快 0.7～1.8 μs。所以上面"acquire 没有后续罚时"是用对 L2 不敏感的 kernel 测的。按页跨步读只有很少几个工作组，也只多 0.7～1.5 μs，说明没有 MI210 那种几 μs 的固定罚时；但对受访存限制的 kernel，acquire 之后要多花多少，没有测。

7.3 主机端：DTK 的中断信号和回调

GPU 正忙着跑一个 2 ms 的 kernel 时，测各种调用在主机上多久返回：

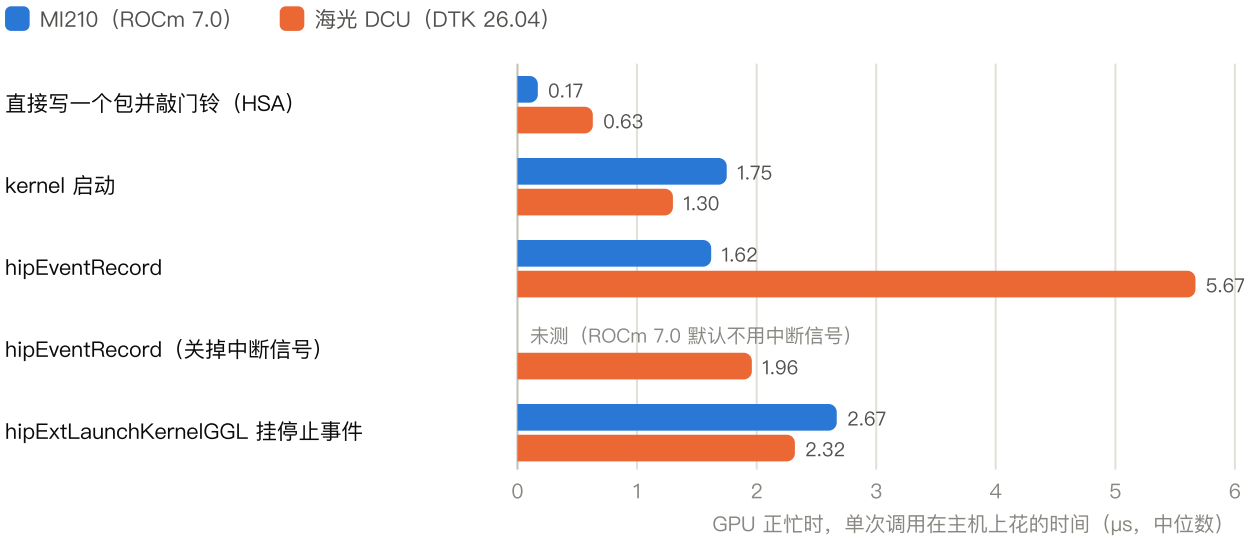


图 7　GPU 忙时，单次调用在主机上花的时间。所有调用都没有在等 GPU（2 ms 的 kernel 还在跑就返回了），所以这是调用本身的 CPU 代价。DTK 的 `hipEventRecord` 要 5.7 μs，是 MI210 的 3.5 倍；关掉中断信号（设 `ROC_SYSTEM_SCOPE_SIGNAL=0`）后降到 2.0 μs。

表 16　GPU 忙时的单次调用（μs，中位数；`out/*_hip_busy*.txt`、`out/*_hsa_busy.txt`）

调用（GPU 正忙，中位数，μs）	MI210	DCU	DCU，ROC_SYSTEM_SCOPE_SIGNAL=0
直接写一个 kernel 包并敲门铃（HSA）	0.17	0.63	—
直接写一个屏障包（两道系统栅栏 + 信号）并敲门铃	0.16	0.60	—
kernel 启动	1.75	1.30	1.47
hipEventRecord（DisableTiming）	1.62	5.67	1.96
hipEventRecord（DisableSystemFence）	1.85	5.86	3.04
hipEventRecord（计时）	2.21	5.71	5.27
hipEventQuery（记录之后）	0.22	0.18	0.29
hipExtLaunchKernelGGL 挂停止事件	2.67	2.32	3.56

- DTK 的 `hipEventRecord` 不会等 GPU 跑完才返回，但它本身要 5.7 μs CPU 时间。MI210 上是 1.6 μs。
- 多出的约 3.7 μs 来自中断信号加主机回调：DTK 给每个事件都用能触发中断的信号，并在主机上挂一个回调（4.4 节日志里每个屏障包都对应一行 `Batch handler done`）。ROCm 7.0 在 MI210 上默认不这样做。
- 设 `ROC_SYSTEM_SCOPE_SIGNAL=0` 让 DTK 建信号时不带中断，单次调用降到 2.0 μs，GPU 端也少 0.6 μs 的中断开销。DTK 不开源，这个解释来自日志和这个开关的效果，没有看到源码。

结果是，DCU 上连续"启动 + 记录"时，主机端（约 7 μs：启动 1.3 + 记录 5.7）和 GPU 端（约 7 μs）差不多一样慢。上一阶段说"DCU 瓶颈在主机端"只说对了一半。

7.4 只关中断信号，连续提交为什么快不了多少

表 17　DCU：默认和 `ROC_SYSTEM_SCOPE_SIGNAL=0` 的对比（μs；`out/dcu_acpp_sig.txt`、`out/dcu_hip*.txt`）

DCU，每次调用（μs）	默认	ROC_SYSTEM_SCOPE_SIGNAL=0
原生 HIP：只启动	1.76	2.00
原生 HIP：启动 + 记录事件	7.44	7.04
原生 HIP：启动 + 记录（DisableSystemFence）	5.73	5.68
原生 HIP：写 4 MiB 的 kernel + 记录	9.75	9.75
AdaptiveCpp 普通提交（默认批量提交）	7.73	7.25
AdaptiveCpp 即时提交（整个程序定义宏）	7.68	7.02
AdaptiveCpp 即时提交 + 粗粒度事件（command group 属性）	2.91	3.27
AdaptiveCpp 即时提交 + 粗粒度事件（队列属性）	2.82	3.20
GPU 间隔：启动 + hipEventRecord	7.68	7.04
GPU 间隔：hipExtLaunchKernelGGL 挂停止事件	4.80	4.16

关掉中断信号，主机端省了 3.7 μs，但连续提交时被 GPU 端挡住：GPU 本身每次还要约 7 μs（kernel 加标记包）。所以原生启动加记录只从 7.44 变成 7.04 μs，AdaptiveCpp 普通提交 7.73 → 7.25 μs，即时提交 7.68 → 7.02 μs。要真正省下来，还是要去掉事件本身：即时提交加粗粒度事件每次 2.8～3.3 μs。

HIP 层面（表 11）：DCU 上启动加 `hipEventRecord` 每次 7.6 μs，`DisableSystemFence` 6.2 μs，`ReleaseToDevice`、`ReleaseToSystem` 和默认一样；`hipExtLaunchKernelGGL` 挂停止事件，GPU 间隔 4.8 μs（关中断信号后 4.2 μs），比"启动再记录"的 7.7 μs 少约 3 μs。

> **小结（DCU）。**GPU 端每次 5.4 μs：屏障包 1.43、主机内存信号 2.6、两道栅栏 1.4，没有 MI210 那种 acquire 后续罚时；release 的写回再多 1.7～4.5 μs（固定 256 个工作组写大数据时到 6.6 μs）。主机端 DTK 的 `hipEventRecord` 另要 5.7 μs，其中约 3.7 μs 来自中断信号和主机回调。和 MI210 共有的问题：`ReleaseToDevice` 没实现，信号放主机内存，HIP 现成的扩展启动路径没被使用。

八、真实访存 kernel 上的总开销

前面的拆分大多用空 kernel。真实算子都要读写数据，事件的总开销会因此变化。图 8 把 HIP 事件让每次迭代多出的总时间，按 kernel 读或写的数据量画出来，H100 作对照。

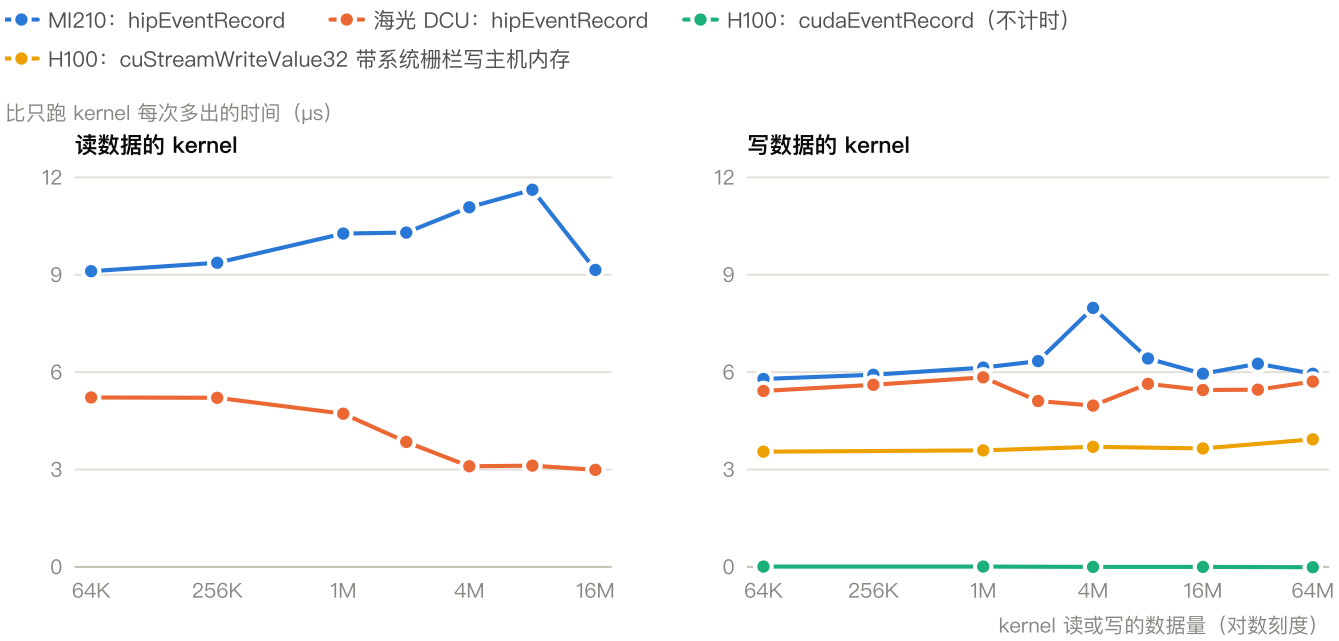


图 8　一次事件的总开销随 kernel 读写数据量的变化。AMD 两条线是"kernel + HIP 事件（手工构造的同样的包）"减去"只有 kernel"；H100 两条线是"写 kernel + 不计时事件"和"写 kernel + 带系统栅栏写主机内存"减去"只有写 kernel"（只测了写 kernel）。逐包敲门铃，数值见表 18。

表 18　一次事件让每次迭代多出的时间（μs）

数据量	MI210 读	MI210 写	DCU 读	DCU 写	H100 写 + 不计时事件	H100 写 + 带栅栏写主机内存
64 KiB	+9.11	+5.79	+5.22	+5.42	+0.01	+3.55
256 KiB	+9.37	+5.92	+5.21	+5.61	—	—
1 MiB	+10.27	+6.14	+4.72	+5.84	+0.01	+3.59
2 MiB	+10.30	+6.34	+3.85	+5.11	—	—
4 MiB	+11.08	+7.98	+3.10	+4.97	0.00	+3.70
8 MiB	+11.62	+6.42	+3.12	+5.64	—	—
16 MiB	+9.15	+5.95	+2.99	+5.45	0.00	+3.65
32 MiB	—	+6.26	—	+5.46	—	—
64 MiB	—	+5.95	—	+5.71	−0.01	+3.93

- **MI210 读数据的 kernel 每次多 9.1～11.6 μs**，主要是 acquire 的后续罚时；写数据的 kernel 多 5.8～8.0 μs，4 MiB 时最多（release 写回 3.2 μs）。上一阶段 MI210 router 换写法后每次省 8.6 μs，比空 kernel 的 5 μs 多，原因就在这里。
- **DCU 的总开销没有随数据量增长，反而在读 kernel 上下降**（读 16 MiB 时只多 3.0 μs）。原因是另一个现象：kernel 越长，"屏障包 + 信号"这部分在 DCU 上越能和 kernel 执行重叠（表 19）。
- **H100 的不计时事件始终是 0**；带系统栅栏的写值始终 3.5～3.9 μs，不随数据量变。

表 19　信号和屏障包的代价随 kernel 长度的变化（μs；写 kernel，逐包敲门铃，`out/*_hsa_sweep.txt`）

写入量	MI210 kernel	信号挂在 kernel 包上，多	屏障包 + 信号，多	DCU kernel	信号挂在 kernel 包上，多	屏障包 + 信号，多
64 KiB	1.32	+2.47	+3.41	1.73	+2.44	+3.89
1 MiB	1.45	+2.37	+3.67	2.22	+1.94	+3.38
4 MiB	2.82	+1.08	+3.70	4.78	0.00	+1.66
16 MiB	12.91	0.00	+4.20	15.22	0.00	+1.61
64 MiB	48.89	+0.06	+4.19	56.26	+0.01	+1.63

表 19 有两点值得注意：

- **信号挂在 kernel 包上时，kernel 一长，信号就不再增加时间。**两张卡都这样：带信号的 kernel 包，每次迭代至少要约 3.8～3.9 μs（MI210）或约 4.15 μs（DCU）；kernel 本身比这更长时，多出的时间变成 0。看起来这是"带信号的包"的吞吐上限，而不是固定加在每次迭代上的延迟。这对 `hipExtLaunchKernelGGL` 这条路径是好消息：真实的、几 μs 以上的 kernel 挂停止事件几乎不花钱。
- **单独的屏障包加信号则不同。**MI210 上 kernel 再长也固定多 3.4～4.2 μs；DCU 上从 3.9 μs 降到 1.6 μs，部分被 kernel 执行掩盖。两家 CP 的行为不同，原因没有查。

九、综合：为什么 AMD 和海光贵，NVIDIA 便宜

表 20　三个平台上"记录一个事件"的对比

	MI210（ROCm 7.0）	海光 DCU（DTK 26.04）	H100
事件在队列里是什么	一个屏障包（BarrierValue），让队列停下来等它	同左（DTK 的 HIP 是 AMD HIP 的分支）	不让流停下来的完成报告（由实测推断）
栅栏	系统 acquire + 系统 release	同左	普通事件不做
完成信号	主机内存，GPU 经 PCIe 去改	同左，而且带中断、挂主机回调	—
L2	系统 release 要写回脏行	同左	不需要写回
空 kernel，GPU 端多	5.0～5.5 μs	5.4 μs	0.04 μs
访存 kernel，GPU 端多	读 9.1～11.6 μs，写 5.8～8.0 μs	读 3.0～5.2 μs，写 5.0～5.8 μs	写 −0.01～0.01 μs（即 0）
主机端单次调用（GPU 忙）	1.6 μs	5.7 μs	未测（连续调用时主机端只多 0.06 μs）
同类"重"操作的代价	屏障包 + 主机内存信号 3.1 μs	屏障包 + 主机内存信号 4.0 μs	计时事件 3.0 μs，带栅栏写主机内存 3.6 μs

归结起来，AMD 和海光贵，是因为 HIP 把每一次 `hipEventRecord` 都按"最坏情况"来做：

1. **按"要让队列停下来"来做。**事件被做成一个设了屏障位的独立包，后面的 kernel 必须等它（以及它的信号写）完成。NVIDIA 的普通事件不阻塞流。
2. **按"主机要立刻直接读所有数据"来做。**每个事件都带系统范围 release，在 AMD 的缓存设计下意味着 L2 写回；还多带一道本不需要的系统 acquire，在 MI210 上让下一个访存的 kernel 多付约 3～5 μs。
3. **按"主机要轮询这个信号"来做。**信号放在主机内存里，GPU 每次都要经 PCIe 去改，并等改完；DTK 上还要触发中断、跑一个主机回调。

这三点在 HIP 的使用场景里不一定都错：原生 HIP 程序只在确实要和主机同步的地方记录事件，按最坏情况做也只付几次。问题在于 SYCL 运行时把它用在了"每个 kernel 一次"的场合，而这些事件绝大多数没有人等，或者只在同一块

GPU 上被等。同样的设计放在 NVIDIA 上不会暴露，因为 CUDA 的普通事件本来就便宜；所以在 NVIDIA 上开发和测试的 SYCL 代码，换到 AMD 和海光上才出现这笔开销。

十、哪些地方不必要或不合理

按代价从大到小排：

1. **每次记录都带系统范围 acquire。**记录事件只是"发布前面的工作做完了"，是 release；acquire 应该由等待的一方做（2.6 节）。在 MI210 上，它在空 kernel 上多 1.5～1.9 μs，下一个 kernel 要访存时合计多 4.9～6.8 μs，和读多少数据、数据原本在不在 L2 都基本无关，是最大的一块；DCU 上只多 0.6～0.8 μs。
2. **`hipEventReleaseToDevice` 写在文档里，却没有实现。**头文件说它"记录事件时使用设备范围的 release"，但 ROCm 7.0 的 `Event::recordCommand` 把它和 `ReleaseToSystem` 都归成默认值，在 MI210 和 DCU 上实测都和默认完全一样。结果用户只能二选一：要么两道系统栅栏全要，要么一道都不要（`DisableSystemFence`，主机可能看不到数据），没有"只要 release"或"只要设备范围"的选项。
3. **每个事件都按"主机要直接读"来做 release。**同一块 GPU 上的依赖（SYCL in-order 队列里每个 kernel 的事件、同卡两个流之间的等待）只要设备范围 release，实测代价为 0。系统范围 release 要把 L2 的脏数据写回显存，这是 AMD 的缓存设计决定的；但只有主机或别的设备真要直接读的时候，才需要付这笔钱（MI210 最多约 3 μs，DCU 最多约 4.5 μs）。
4. **完成信号放在主机内存。**ROCr 把所有信号都放在主机内存，连声明了"只有 GPU 用"的信号也一样。放显存每次省约 1.5 μs。代价是主机轮询会变慢，所以这是一个取舍；但对没人在主机上等的事件，这笔钱是白付的。
5. **DTK 默认给每个事件用中断信号和主机回调。**GPU 端多 0.6 μs，主机端多约 3.7 μs。ROCm 7.0 在 MI210 上默认不这么做。
6. **HIP 自己有更便宜的现成路径，SYCL 运行时没用。**`hipExtLaunchKernelGGL(..., stopEvent)` 把事件挂在 kernel 包上，不另插屏障包。GPU 上相邻 kernel 的间隔：MI210 3.84 μs（"启动再记录"8.00 μs），DCU 4.2～4.8 μs（7.0～7.7 μs）。而且 kernel 一长，挂在 kernel 包上的信号几乎不花钱（第八节）。
7. **`hipEventDisableTiming` 在 AMD 上不起作用。**带不带计时，放进队列的包完全一样，因为 HIP 的每个队列都开着时间戳采样。开不开采样实测没有差别，所以这一条不花钱，只是标志名不副实。
8. **SYCL 运行时每次提交的 CPU 代码偏重（不限于 AMD）。**每次都构造并拆除 handler、command group 对象，在缓存里查 kernel，即使不返回事件也建一个内部 `event_impl`，A100 上约 0.7 μs CPU；返回事件再多 0.44 μs。它在提交卡在 CPU 上的平台（A100）上直接显现，在 MI210 上被 GPU 时间盖住（6.2 节）。

最少必要的代价

按 2.6 节的"最少需要"，一个事件在 AMD 上最少应该花多少？表 21 用同一硬件、同样的包，只换信号位置和是否单独插屏障包：

表 21　不同做法下，空 kernel 上一次事件让 GPU 多花的时间（μs）

做法（空 kernel，GPU 端每次多出的时间，μs）	MI210	海光 DCU
信号挂在 kernel 包上，信号放显存（粗粒度）	0.92	1.05
单独一个屏障包，信号放显存（粗粒度）	1.59	2.47
信号挂在 kernel 包上，信号放主机内存	2.51	2.56
单独一个屏障包，信号放主机内存	3.09	4.00
HIP 现在的事件（屏障包 + 两道系统栅栏 + 主机内存信号）	4.98	5.43

前两行信号放在显存，只适合由 GPU 等待（或主机能接受较慢轮询）的事件；后两行信号在主机内存，主机可以直接轮询。都不带栅栏；需要主机直接读数据时，再加上随脏数据增长的系统 release（表 7）。

所以，按"最少必要"算，只由 GPU 等待的事件约 0.9～1.6 μs（DCU 1.0～2.5 μs），要让主机轮询的事件约 2.5～3.1 μs（DCU 2.6～4.0 μs）。HIP 现在实际要 5～11 μs，多出来的主要是 acquire 和不分场合的系统范围 release，DCU 上还有主机端的中断信号和回调。

十一、可以怎么改，对 SYCL 和转译器的意义

11.1 工具链层面的修法（都有对应的实测）

表 22　各种修法下，一次事件让每次迭代多出的时间（μs）

改法	改哪里	MI210，读 4 MiB 的 kernel	MI210，空 kernel	DCU，空 kernel
现状（两道系统栅栏 + 主机内存信号）	—	11.1	5.0～5.5	5.4（主机端另 5.7）
SYCL 运行时改用 HIP 现成的 `hipExtModuleLaunchKernel(..., stopEvent)`	DPC++ UR 的 HIP 适配层 / AdaptiveCpp 的 HIP 后端，不用改 HIP	未测	GPU 时间线上 1.4	GPU 时间线上 1.8～2.4
事件只做 release（去掉 acquire）	CLR `submitMarker` 的包头	4.0	3.4～3.6	4.6
实现 `ReleaseToDevice`，SYCL 运行时的内部事件用它	CLR `Event::recordCommand`；DPC++/AdaptiveCpp 建事件时传这个标志	3.6～3.9	3.1～3.2	4.0
再把只给 GPU 用的信号放显存	ROCr 的信号分配	约 2	1.6	2.5
DTK：事件信号不带中断、不挂回调	DTK 的 HIP 默认设置	—	—	GPU 端少 0.6，主机端少 3.7
SYCL 运行时不建事件（惰性事件、`nd_launch`、即时提交 + 粗粒度事件）	DPC++ / AdaptiveCpp	0	0	0

除"未测"和"约"之外都是实测：HSA 层用同一硬件、同样的包，只换包头和信号位置；HIP 层是真的调用。"约 2"是按空 kernel 上"信号放显存"省下的量推算的。范围的两端分别来自逐包敲门铃和只敲一次门铃两种测法。没有真的去改 CLR 重新编译；真正打补丁并在 SYCL 程序上验证，是下一步可以做的事（第十四节）。

11.2 SYCL 运行时可以怎么做

- **in-order 队列默认返回"惰性事件"。**先只记下流上的位置，有人 `wait()` 或拿它做依赖时再记录后端事件。这正是 AdaptiveCpp 粗粒度事件的语义，不改 SYCL 的 API。
- **必须建后端事件时，用 HIP 的扩展启动把事件挂在 kernel 包上**，而不是"启动再单独记录"。
- **运行时内部的事件（只用于同一块 GPU 上的依赖）用设备范围 release**，前提是 HIP 先实现 `ReleaseToDevice`。在此之前，DPC++ 的 HIP 适配层可以给内部事件加 `hipEventDisableSystemFence`，但主机等事件后要直接读数据时，得另外补一道系统栅栏。

AdaptiveCpp 的粗粒度事件加即时提交、DPC++ 的 enqueue functions（`nd_launch`），都是在 SYCL 层绕开这一切。本次调研说明，即使在 HIP 层，按事件的用途选对栅栏，也能省掉大部分。

NVIDIA 上情况不同：`nd_launch` 去掉事件之后还剩约 0.5 μs，这是 SYCL 运行时每次提交的固定 CPU 代价（6.2 节）。它只能靠 command graph 这类批量提交摊掉（上一阶段测的 A100 上每个 kernel 0.98 μs，比原生还快），或者在运行时里给最常见的提交（in-order 队列、不要事件、没有 accessor）做一条更轻的路径。

11.3 对我们转译器的规则

表 23　转译规则（可以直接写进转译器或已验证原语）

后端	规则	条件
DPC++（HIP、CUDA、Level Zero）	`kernel<<<g,b,s,stream>>>` 译成 `syclex::nd_launch(q, nd_range, k)`；需要 local memory 时用 `syclex::submit(q, cgf)`	队列是 in-order；源程序没有用这个 kernel 的完成事件
AdaptiveCpp	构建时对**整个程序**加 `-DACPP_ALLOW_INSTANT_SUBMISSION=1`（不能只写在一个源文件里，否则反而变慢一倍），每个 `submit` 带粗粒度事件属性	同上，且只用 USM；转译器要能控制整个程序的构建
源程序确实要等某个 kernel	只在那一处要事件	—
一次调用里有多个小 kernel	DPC++ command graph 录一次、反复执行	形状固定，参数不变

这件事和 paraTRANS 论文里发现的两个问题（相同 GPU 上多占 1 个寄存器导致占用率减半；DPC++ 在 MI100 上倾向高占用率导致 scratch 溢出）是同一类：移植后多出了源代码没有的代价，而且只在特定硬件上显现。不同的是，那两个出在编译器，这个出在运行时，而且一半在 SYCL 层（每次都建事件），一半在 HIP 层（事件本身做得太重）。

十二、和已有工作的关系

以下是 2026-11-17 的粗查，没有逐篇细读。

- **GROMACS 在 AMD GPU 上的 SYCL 移植**（Alekseenko 等，CUG 2024，arXiv 2405.01420）：已经指出多余的 `hipEventRecord` 是开销来源，用放宽事件同步保证的扩展（即粗粒度事件）解决，并测了即时提交。核心现象"AMD 上 SYCL 每次提交的事件记录很贵"已经被发表。
- **低比特矩阵向量乘的 SYCL 可移植性研究**（DOI 10.1145/3731599.3767574，2025）：AMD 上小问题受调度开销影响，用 `discard_events` 把启动时间降了约 40%。
- **SYCL-Bench 2020**（DOI 10.1145/3648115.3648120）：DPC++ 的 AMD 后端调度开销明显偏高。
- **AdaptiveCpp stdpar**（Alpay，DOI 10.1145/3648115.3648117）：引入即时提交，stdpar 默认用粗粒度事件以省去后端事件。
- **Intel 的设计说明**（Brodman，IWOCL）：去掉事件开销的最好办法是不创建；`discard_events` 并不能真正避免创建，于是有了 enqueue functions，随后 `discard_events` 被弃用（intel/llvm PR #18059）。

本次调研和这些工作的不同之处在于：

1. 把事件的代价拆到了包、信号位置、acquire、release 各自多少，并量化了 release 随脏数据增长、acquire 在 MI210 上的后续罚时；
2. 发现 `hipEventReleaseToDevice` 没有实现，以及 HIP 已有的扩展启动路径没有被 SYCL 运行时使用；
3. 海光 DCU（DTK）上的同类测量，包括主机端的中断信号和回调；
4. 用 NVIDIA 做"功能对应"的对照，说明 NVIDIA 做同样重的事也不便宜，便宜的原因是普通事件不做这些事；
5. 从 CUDA 自动转译（转译器或 LLM agent）的角度看这笔开销：翻译出的程序每个 kernel 都带一个源程序没有的同步对象。

十三、局限和待验证的问题

- **MI210 上"acquire 之后下一个访存 kernel 多出约 3～5 μs"的微观原因是推断。**CP 固件不开源；rocprof 采计数器时会在每个 kernel 前后清缓存，测不出差别。
- **自建信号块放在显存，是测量手段，不是 ROCr 支持的用法。**放在粗粒度显存里的信号，主机经 PCIe 直接读可能看不到最新值，只适合只由 GPU 等待的事件。
- **`hipEventSynchronize` 和 `fence_dirty_` 的疑点没有验证。**连续用 `hipExtLaunchKernelGGL` 绑定停止事件时，后面的 kernel 只有设备范围 release；主机等到事件后直接读显存（不经过拷贝）时，按内存模型可能看不到 L2 里的数据。需要写一个程序验证。
- **DTK 不开源。**DCU 上"每个事件用中断信号和回调"来自它的日志和 `ROC_SYSTEM_SCOPE_SIGNAL=0` 的效果，没有看到源码。
- **NVIDIA 的结论来自黑盒测量**（驱动不开源）。"普通事件不让流停下来""L2 不需要写回"是从实测加公开资料推出来的，没有找到官方文档的明确表述。
- **修法没有真的实现。**第十一节的数字是"用同样的包、换包头和信号位置"测出来的，不是打了补丁的 CLR 或 SYCL 运行时。`hipExtModuleLaunchKernel` 在真实算子（读 4 MiB）上没测。
- **DCU 的部分作业不在原计划的节点上。**dcu3 一直被别人整机占用，有四个短作业（每个 12～63 秒）跑在了 dcu1 的空卡上（那是 b10 Qwen 所在的节点），只用了自己分到的卡和 4 个 CPU 核。DCU 上主机端的数字可能受同节点负载影响，但前后几次测量一致。
- **L2 的写回策略是从测量推出来的。**两块卡都只留得住约 4～5 MiB 脏数据，DCU 超过上限后 release 还会慢慢变贵；AMD 和海光都没有公开这条策略。Linux 驱动里只有一个含义未公开的寄存器字段（`TCC_CTRL` 的 `WRITEBACK_MARGIN`），MI210 上它的值要 root 权限才能读。
- **没有在 MI300、更新的 ROCm 或其他海光型号上测。**
- **A100 的采样只记录被打断时的指令地址**，没有调用栈；CUDA 驱动去掉了内部符号，驱动内部只能看成一个整体。各库的分摊是样本比例乘以每次调用的时间，绝对值和主机 CPU（EPYC 7763）有关。DCU 上 AdaptiveCpp 去掉事件后仍比原生多约 1.1 μs（表 1），推测也是运行时自己的 CPU 开销，没有细查。

十四、建议的下一步

1. **给 CLR 打补丁并端到端验证。**改 `submitMarker`，让事件只做 release；在 `Event::recordCommand` 里实现 `ReleaseToDevice`。重新编译 HIP 运行时，在 MI210 上用 DPC++ 的 HIP 后端跑 Sol 的 router、qknorm 等小算子，看能省多少。
2. **改 DPC++ 的 UR HIP 适配层**，让 `urEnqueueKernelLaunch` 在要事件时改用 `hipExtModuleLaunchKernel(..., stopEvent)`；同样在真实算子上测，并写程序验证 `fence_dirty_` 的可见性疑点。
3. **DCU：**在真实算子上测 `ROC_SYSTEM_SCOPE_SIGNAL=0` 加扩展启动的组合；把"每个事件都挂中断和回调"作为问题反馈给 DTK。
4. **进一步定位 MI210 的 acquire 罚时：**在下一个 kernel 内部用 `s_memrealtime` 打点，看时间花在取代码、取参数还是第一次访存上；有条件时在 MI300 上复测。
5. **SYCL 运行时的提交路径：**在更新的 DPC++ 上复测 6.2 节的采样，看 handler、command group 和内部事件对象的开销有没有减少；DCU 上对 AdaptiveCpp 做同样的采样。
6. **如果要单独成文：**补 HeCBench 和 llama.cpp 这类真实程序的端到端数据，说明修法在完整应用上的收益。

附录 A　文件与复现

目录 `research/data/h100_dcu_20261110/analysis_v2/claude/launch_overhead/why_amd/`：

表 24　文件

文件	内容
`README.md`	本次调研的文字结论（本报告的底稿）
`kernels.cl`、`hsa_probe.cpp`、`hip_probe.cpp`、`cuda_probe.cu`、`counters_probe.cpp`	测量程序（表 2）
`run_mi210_host.sh`、`run_mi210_more.sh`	GPU62 宿主机上运行（`run_mi210_more.sh <card> <modes>`）
`run_h100.sbatch`、`run_h100.sh`	集群 H100、GPU70 的 H100
`diag_dcu.sbatch`、`run_dcu_hsa.sbatch`、`run_dcu_batch.sbatch`、`run_dcu_busy.sbatch`、`run_dcu_acpp_sig.sbatch`、`run_dcu_fine.sbatch`	DCU 作业（`run_dcu.sbatch` 是第一次、库路径错了的版本）
`out/`	全部原始输出：`mi210_*`、`dcu_*`、`h100_*`、DTK 日志解码（`dtk_hiplog/`、`dtk_sycllog/`）、rocprof 计数器（`pmc/`）
`../a100_why/`	A100 上 SYCL 提交开销的采样和驱动调用计数（6.2 节）：程序、运行脚本 `run_a100_why.sh`、`run_count.sh`，原始输出 `out/`，说明 `README.md`
`report/build_report.py`、`report/report_template.html`	本报告的生成脚本：图和表从 `out/` 直接读取计算，再用 Chrome 打印成 PDF

远端副本：GPU62 `~/claude-b9-analysis/launch_overhead/why_amd/`，集群 `~/aqua/claude-b9-analysis/launch_overhead/why_amd/`，GPU70 `~/claude-launch-overhead/why_amd/`；A100 的采样在 GPU50 的 `claude-b9-analysis/launch_overhead/a100_why/`。

构建命令（摘自运行脚本）：

```
# kernel（MI210 用 ROCm 的 clang，DCU 用 DTK 的 clang，-mcpu=gfx936）
clang -x cl -cl-std=CL2.0 -target amdgcn-amd-amdhsa -mcpu=gfx90a -nogpulib -O3 kernels.cl -o kernels_gfx9
0a.hsaco
# HSA 探针：MI210
clang++ -O2 -std=c++17 hsa_probe.cpp -I$R/include -L$R/lib -lhsa-runtime64 -Wl,-rpath,$R/lib -o hsa_probe
_mi210
# HSA 探针：DCU（必须链接计算节点上的 /opt/hyhal，LD_LIBRARY_PATH 里不能有 $DTK/.hyhal/lib）
clang++ -O2 -std=c++17 hsa_probe.cpp -I$HI -L/opt/hyhal/lib -lhsa-runtime64 -Wl,-rpath,/opt/hyhal/lib -o
hsa_probe_dcu
# HIP 探针（DCU 上要 -std=c++17）
hipcc -O3 --offload-arch=gfx90a hip_probe.cpp -o hip_probe_mi210
# NVIDIA
nvcc -O3 -arch=sm_90 cuda_probe.cu -lcuda -o cuda_probe_h100
```

运行：`./hsa_probe_mi210 kernels_gfx90a.hsaco [模式]`，模式为空（主表与时间线）、`sig`、`sweep`、`wfine`、`rfine`、`l2`、`cold`、`acq`、`busy`；加 `BATCH_DOORBELL=1` 只敲一次门铃。`wfine` / `rfine` 用 `FINE_LIST`（KiB，逗号分隔）指定数据量，`FINE_GS=1` 换成固定 256 个工作组的循环 kernel。`./hip_probe_mi210 [log|busy]`。AQL 日志：`AMD_LOG_LEVEL=4 AMD_LOG_MASK=0x8`。DTK 日志：`HIP_LOG_LEVEL`、`HIP_LOG_OUTPUT_PATH`，再用 `$DTK/.hyhal/NanoLog/bin/hiplogdump sort <文件>` 解码。

重新生成本报告：`python3 -I report/build_report.py <输出.pdf>`（需要 macOS 上的 Google Chrome）；`--check` 只打印推导出的数字。

附录 B　术语表

术语	含义
HSA、ROCr	AMD GPU 的底层运行时接口（Heterogeneous System Architecture）和它的实现 ROCr（`libhsa-runtime64`）
CLR、ROCclr	HIP 运行时的实现（`clr` 仓库），HIP 调用在这里翻译成 AQL 包
DTK、hyhal、libgalaxyhip	海光的软件栈；hyhal 是它的 HSA 层，libgalaxyhip 是它的 HIP 库（AMD HIP 6.3 的分支）
AQL 包	写进 GPU 队列的 64 字节命令，kernel 派发包或屏障包
CP	命令处理器，GPU 上按顺序处理队列里的包的部件
门铃（doorbell）	主机写好包后通知 CP 来取的寄存器
屏障位	包 header 里的一位：设了它，这个包要等前面的包都完成才开始
BARRIER_AND、BarrierValue	两种屏障包：前者等若干信号都为 0，后者（AMD 扩展）等一个信号满足条件
完成信号	包完成时 CP 修改的内存中的整数，供主机或其他包等待
acquire / release	获取 / 发布：读别人数据前作废过期缓存；让自己的写入对别人可见
设备范围 / 系统范围	栅栏的作用范围：同一块 GPU（agent）/ 包括主机和其他 GPU（system）
`buffer_wbl2`、`buffer_invl2`	GFX90A 上写回 L2 脏行、作废 L2 行的指令
MTYPE RW / NC	GFX90A 的内存类型：RW 用于本卡显存（L2 中可脏），NC 用于非本地内存
细粒度 / 粗粒度内存	主机和 GPU 能否在 kernel 运行期间细粒度地看到彼此的写入；粗粒度只在同步点保证一致
UR	Unified Runtime，DPC++ 的后端适配层
in-order 队列	按提交顺序执行的 SYCL 队列
粗粒度事件、即时提交	AdaptiveCpp 的两个扩展：事件只和整个流同步（不建后端事件）；提交时直接调用后端，不经后台线程
enqueue functions、`nd_launch`	DPC++ 的扩展，提交 kernel 而不返回事件
`s_memrealtime`、`%globaltimer`	AMD 和 NVIDIA GPU 上的恒定频率时钟，用来在 GPU 上打时间戳

附录 C　调研过程记录与更正

C.1 过程

1. 读 CLR 源码、抓 AQL 日志，确定 `hipEventRecord` = BarrierValue 包（系统/系统）+ 主机内存信号；发现 `ReleaseToDevice` 被忽略、DisableTiming 不改变包。
2. 写 `hsa_probe`，在 MI210 上逐项拆分；发现空 kernel 上栅栏的钱主要在 acquire，release 只在写数据时明显。
3. 为解释 acquire 在读 kernel 上的大代价，加冷热数据、按页跨步、插空 kernel 三组对照，排除"数据被清出 L2"和"后台操作"两种解释。
4. H100 对照：普通事件几乎为 0；计时事件和带栅栏写值 3～4 μs，但不随脏数据增长。
5. DCU：先修好库路径；HSA 层重做拆分；为区分主机端和 GPU 端，加"只敲一次门铃"和"GPU 忙时单次调用"两种测法；解码 DTK 日志，找到中断信号和回调；测 `ROC_SYSTEM_SCOPE_SIGNAL=0` 的效果。

6. 写本报告时，从原始输出重新计算了所有图表数字，补上了"kernel 越长，挂在 kernel 包上的信号越不花钱"这一观察（第八节）。
7. 读表 1 时发现 A100 上 SYCL 不管返不返回事件都比原生慢，补做了 A100 上的进程内采样和驱动调用计数（6.2 节）。
8. 看图 4 时发现：MI210 的拐点只是 2 的幂采样点，DCU 的曲线看起来没有拐点。按 1 MiB（MI210 另有 0.5 MiB）加密重测，并加了固定工作组数的 kernel 和冷热读对照；又查了 Linux 驱动和 AMD 文档里有没有写回策略的说明（5.3 节）。

C.2 走过的弯路

- 最初的 HSA 探针把 kernel 参数放在主机细粒度内存里，每个工作组都要经 PCIe 去读参数，数据 kernel 慢了一倍。改成放显存后重测；旧结果保留为 `out/*_hostkernarg.txt`，没有使用。
- rocprofv3 的计数器（L2 命中、地址翻译）在各种情况下都一样，因为采集本身会在每个 kernel 前后清缓存，放弃。
- GPU62 的 Docker 因 `/data1` 满了起不来，MI210 改在宿主机上跑同一版本的 ROCm 运行时。
- GPU50 的 A100 都被占用，改用集群 H100；GPU70 的卡在两次运行之间被别人占用，只留下一组复测。
- CUDA 12 拒绝 `NO_MEMORY_BARRIER` 标志，cuda_probe 改为跳过被拒绝的标志；第一组计时受预热影响，加了预热。
- DCU 上 `LD_LIBRARY_PATH` 里有 `$DTK/.hyhal/lib` 时看不到设备；DTK 的 AQL 日志开关和 AMD 不同，要用 NanoLog 解码。

C.3 对之前说法的更正

下面第 3～5 处是写本报告时从原始输出重新计算发现的，第 6～8 处是报告写好后补充的；涉及的 README 都已同步改正。

- 之前说"AMD 上的钱主要是 release 把 L2 脏数据写回"，**不对**：空 kernel 上栅栏的钱主要在 acquire，release 只在 kernel 写了数据时才明显。
- 之前说"DCU 瓶颈在主机端"，**只对一半**：主机端和 GPU 端都在 7 μs 左右。
- `why_amd/README.md` 里 MI210 扩展启动的对比用了两种来源的数字（HSA 时间线的 8.32 μs 和 HIP 时间线的 3.84 μs）。本报告统一用 HIP 层面的 GPU 时间线：只启动 2.40 μs，启动再记录 8.00 μs，扩展启动挂停止事件 3.84 μs，比只启动多 1.4 μs（README 原写 1.3 μs）。
- `why_amd/README.md` 第 1 节把系统 acquire 写成"自身 1.5～1.9 μs，下一个访存的 kernel 再多约 4.5～5 μs"，容易误读成两者相加。本报告改为：空 kernel 上 1.5～1.9 μs；下一个 kernel 要访存时合计 4.9～6.8 μs（数据本来就不在 L2 时合计 4.5～4.8 μs），比空 kernel 时多出约 3～5 μs。
- "最少必要"的代价，README 写成"一个主机可见的事件约 1～3 μs"。更准确的说法是：只由 GPU 等待的事件 0.9～1.6 μs（DCU 1.0～2.5 μs），要让主机轮询的事件 2.5～3.1 μs（DCU 2.6～4.0 μs）。
- 上一阶段说"SYCL 运行时自己的软件开销很小（MI210 约 0.1～0.2 μs，A100 约 0.5 μs）"，这只是看得见的部分。SYCL 运行时每次提交约 0.7 μs CPU（返回事件约 1.1 μs），在 MI210 上大部分被 GPU 时间盖住，在 A100 上一比一显现（6.2 节）。
- 之前说 MI210"到 4 MiB 为止"每 MiB 0.7～0.8 μs、更大的写入"被挤出 L2"，DCU"一直增长到 16 MiB"，都来自只测 2 的幂的采样。加密以后：两块卡的脏数据上限都在 4～5 MiB（MI210 在 4.5 MiB 处最高），L2 容量 8 MiB；DCU 在 11～16 MiB 持平的是拐点之后另一段缓慢增长。7.2 节原来说"DCU 相邻 kernel 之间本来就没有从 L2 得到复用"也不对，是探针 kernel 卡在工作组派发上。

- 表 20 里 H100 写数据的 kernel 一格原写"0.00～0.01 μs"，按原始数据应为"−0.01～0.01 μs（即 0）"，生成网页版时核对发现。