将每流统计信息跟踪集成到 Accel-Sim

Shichen (Justin) Qiao*、Xin Su*、Matthew D. Sinclair+威斯康星大学麦迪逊分校 * 电气与计算机工程系 + 计算机科学系 {sqiao6, xsu57}@wisc.edu；sinclair@cs.wisc.edu

1. 摘要

Accel-Sim 是一款广泛使用的计算机体系结构模拟器，能够详细模拟现代 NVIDIA GPU 的行为 [1]。然而，尽管 Accel-Sim 及其底层的 GPGPU-Sim 模拟了真实 GPU 的许多特性，但迄今为止，它仍无法按流分别跟踪统计信息。相反，Accel-Sim 会汇总所有同时运行的流的统计信息（例如周期数和缓存命中/未命中次数）。这可能会妨碍用户准确识别特定内核和流的行为，并可能导致错误的结论。因此，在这项工作中，我们扩展了 Accel-Sim 和 GPGPU-Sim 的统计信息跟踪功能，以跟踪每流统计信息 [2]。为了验证这一功能，我们设计了一系列多流微基准测试，并检查了它们报告的每内核、每流计数。

2. 背景

目前，Accel-Sim 主要使用 src/gpgpu-sim/gpu-cache.cc 来跟踪缓存统计信息。这些统计信息存储在 m_stats、m_stats_pw 和 m_fail_stats 中的无符号长长整型向量的向量里，其中外层向量表示访问类型，内层向量表示访问结果。例如，当访问 L1 或 L2 缓存时，会使用访问类型和访问结果调用 inc stats、inc stats pw 或 inc fail stats_ _ _ _ _，相应的向量条目会加一。

当多个流并发执行时，当前方法可能会导致缓存统计信息不准确。例如，在图1中，当多个流重叠执行时，如果两个流在同一周期都访问缓存，相应的缓存统计数据总共只会递增一次，而不是每个流递增一次。此外，通过当前的跟踪输出，也无法判断重叠期间是哪个内核在更新某项统计数据，这使得优化底层内核的行为变得困难。尽管 Accel-Sim 允许用户串行执行各个流，使它们彼此隔离（从而能够获取每个流上各个内核的精确统计数据），但由于这些流并非并发运行，其行为会有所不同。

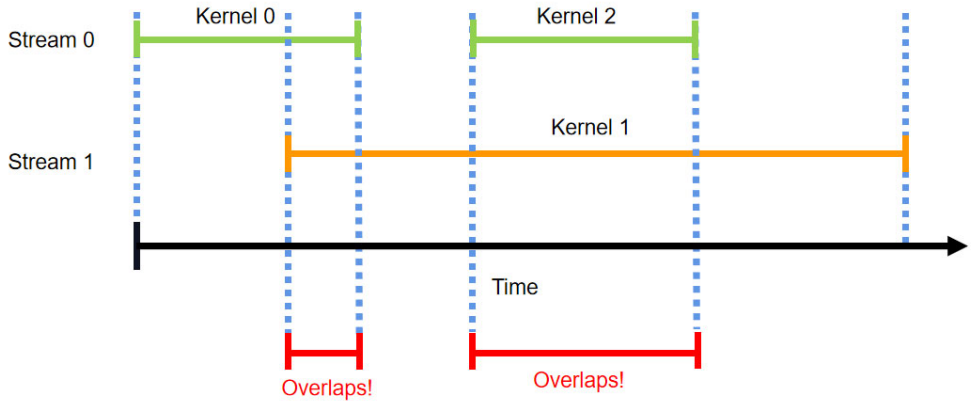

图1： 重叠的内核执行导致缓存统计数据不准确的时序图示例。

3. 实现

Accel-Sim：
基础提交 ID：2260456ea5e6a1420f5734f145a4b7d8ab1d4737 我们的功能：https://github.com/accel-sim/accel-sim-framework/pull/166

GPGPU-Sim:
基础提交 ID：13c67115070dc2f0876254a790d0238073ca364a 我们的功能：https://github.com/accel-sim/gpgpu-sim_distribution/pull/47

3.1 缓存统计数据

为了按流方式跟踪缓存统计数据等统计信息，我们在以下数据结构中增加了一个额外的外层维度。例如，在 gpu-cache.h 中，我们进行了如下修改：

之前：

```
1. std::vector<std::vector<unsigned long long> >m_stats；2. std::vector<std::vector<
unsigned long long> >m_stats_pw；3. std::vector<std::vector<unsigned long long> >
>
>
long long>>>m_stats_pw；3. std::map<unsigned long long, std::vector<std::vector<
unsigned long long>>>m_fail_stats;
```

接下来，我们修改了构造函数、修改器、运算符和访问器，通过添加一个新的必需参数 unsigned long long streamID，以反映上述数据结构的变更：

修改前：1. void inc_stats(int access_type, int access_outcome); 2. void inc_stats_pw(int access_type, int

```
a
a
a
l
i
a
(
```

```

```

streamID 已在 Accel-Sim 的 trace_kernel_info_t 类中进行跟踪。因此，可以通过现有的访问器函数 get cuda stream id() 在 Accel-Sim 中获取它。不过，为了也能在 GPGPU-Sim 中访问它，我们修改了 trace_ _ _kernel_ info_t 的构造函数，将 cuda_stream_id 也传递给 kernel_info_t（该信息可在 GPGPU-Sim 中访问）。这样一来，我们便能在使用内核对象的任何地方确定内核运行在哪个流上，并将 streamID 传播到其他数据结构中，例如 mem_fetch 和 warp_inst_t。这样做使我们能够在整个 GPGPU-Sim 中确定给定统计数据应更新哪个流。

此外，当前 Accel-Sim 的实现会在任意内核退出后打印所有缓存统计数据。因此，当多个流并发执行且某个内核结束时，其他流的缓存统计数据也会被重复打印。我们修改了这一行为，现在只打印退出内核所属流的缓存统计数据：将该内核的 CUDA 流 ID 从 Accel-Sim 端传递到_ _ _ GPGPU-Sim，以便打印统计数据。

修改前：

```
    1. void print_stats(FILE *fout, const char*cache_name  = "Cache_stats") const; 2. void print_fail_stats(FILE
*fout, const char *cache_name ="Cache_fail_stats") const; 修改后：1. void print_stats(FILE *fout, unsigned l
(
unsigned long long streamID, const char *cache_name  ="Cache_fail_stats") const;
```

```

```

3.2 性能统计数据

接下来，为了跟踪每个流上每个内核的启动和退出时间，我们在 gpu-sim.h 中添加了以下数据结构：

```
1. 结构体定义 {
```

```
2. 无符长长整型 结束周期;4. }内核时间类型;
5. stdumen<无符号 内核时间类型>> GPU 内核时间;6. 无符
```

这些新字段会在 gpgpu_sim::launch 和 gpgpu_sim::set_kernel_done 执行期间进行跟踪和更新，并在每个内核统计信息的末尾打印出来。

4. 用法

1. 如果需要按流统计信息，应在相应的 gpgpusim.config 中将“-gpgpu_concurrent_kernel_sm”标志设为 1。 2. 在 accel-sim-framework/ 中，运行以下命令设置环境：source ./gpu-simulator/setup_environment.sh 3. 在

```
simulator/bin/release/accel-sim.out -trace ./util/tracer
```

s/SM7_TITANV/trace.config 4. 在输出跟踪中，查找“Total_core_cache_stats_breakdown”或“L2_cache_stats_breakdown”，以查看详细的按流缓存统计信息。

5. 验证

5.1 验证 L2 缓存统计信息

我们首先使用一个经过修改的简单基准测试版本验证了我们的设计。该基准测试的缓存访问次数已知且确定，`l2 lat.cu` [1]：https://github.com/accel-sim/gpu-app-collection/blob/release/src/cuda/GPU_Microbenchmark/l2_lat/l2lat.cu

我们选择 l2_lat.cu，因为它的 L2 访问次数是确定的，因此易于验证。我们对其进行了修改，使四个流并行运行相同的 l2_lat 工作负载：

```
1. ...2. #define THREADS_NUM 1 // 一个线程用于初始化指针追逐数组3. #define
```

```
全局函数限定符 void L2缓存延迟(无符号32位整数 *startClk, 无符号32位整数 *stopClk, uint64_t *posArray, uint64_t *dsink){
    // 线程索引
    无符号32位整数 tid = t
    ;

    // 初始化指针追逐数组
    如果 (tid == 0){
        for(无符号32位整数 i=0; i<(ARRAY_SIZE-1); i++)
posArray[i] =(uint64_t)(posArray + i + 1);
        posArray[ARRAY_SIZE-1] =(uint64_t)posArray;
    }

if(tid < THREADS_NUM){
        uint64_t *ptr = posArray + tid ;
        uint64_t ptr1 , ptr0;
            …
        // 指针追逐 ITERS 次
        // 使用 cg 修饰符将加载数据缓存到 L2，并绕过 L1
        for(无符号32位整数 i=0;i<ITERS; ++i) {
            <escape>asmvolatile ("{\t\n"
            "全局缓存加载指令 %0, [%1];\n\t"
            "}" : "=l"(ptr0): "l"((uint64_t*)ptr1) : "内存"
            );
            ptr1 =ptr0 ;  //为下一个加载替换寄存器
        }
        ...
    }
}

int main(){
    ...
    CUDA 流类型 stream_1 , stream_2 , stream_3 , 流_4 ;
    ...
    L2缓存延迟<<<1,THREADS_NUM,0,流_1>>>(startClk_g, stopClk_g, posArray_g, dsink_g);
    L2缓存延迟<<<1,THREADS_NUM,0,流_2>>>(startClk_g, stopClk_g, posArray_g, dsink_g);
    L2缓存延迟<<<1,THREADS_NUM,0,流_3>>>(startClk_g, stopClk_g, posArray_g, dsink_g);
    L2缓存延迟<<<1,THREADS_NUM,0,流_4>>>(startClk_g, stopClk_g, posArray_g, dsink_g);
    cudaStreamSynchronize(NULL);
    ...
}
```

为验证我们的更改，我们运行三种配置：a) 串行化内核启动

（*tip_serialized*，表示未经我们修改且流被串行化的 Accel-Sim）、b）未经我们修改（*clean*，表示未经我们修改的基线 Accel-Sim），以及 c）在并发启动内核时应用了我们的修改（*tip*）。接下来，我们比较了这些配置的所有缓存统计信息。

图2展示了这些配置下的性能和缓存结果。时间线显示，我们现在以并发方式运行4个内核；由于它们是具有相同访问模式的同一个内核的副本，因此所需时间大致相同。这一点也反映在缓存统计数据中：对于相同的访问类型和相同的访问结果，*clean* 统计数据始终等于我们的*tip* 统计数据在全部四个流上的总和。请注意，当内核被串行化时（蓝色条），与并发*tip* 计数（绿色条）相比，我们可能会有更多的“命中”计数。这是因为并发执行时缺失的“命中”计数，由于不同流之间存在加载依赖，被计为“MSHR_命中”。四个流中每个流的总读写访问计数彼此一致，并且完全符合我们的预期计数。

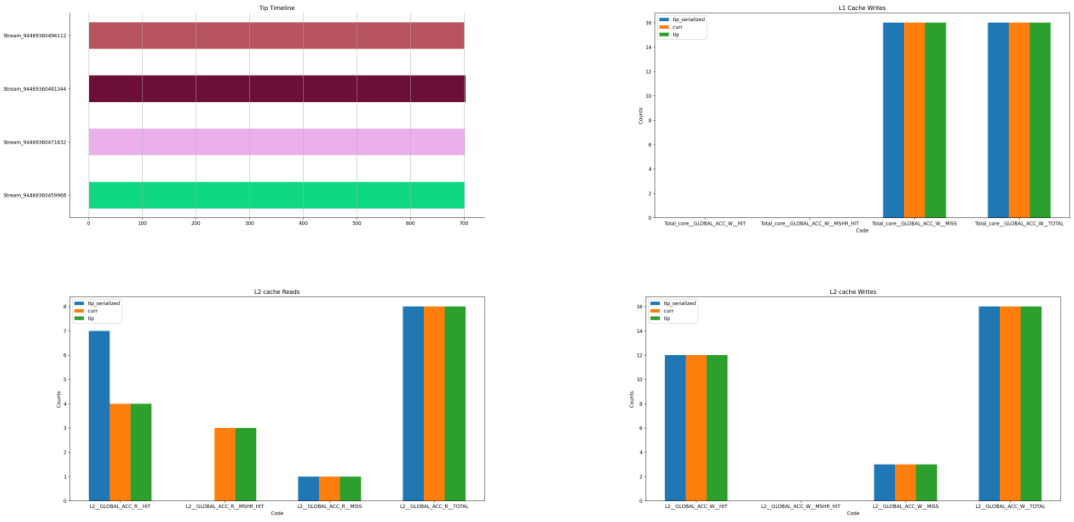

图2： l2_lat_4stream.cu 验证结果。启用新功能（绿色）后的缓存访问计数与未应用补丁（橙色）时的对应计数完全一致。

请注意，上文提到的序列化内核执行是通过将此补丁应用于 Accel-Sim 实现的：

```
diff--git a/gpu-simulator/main.cc b/gpu-simulator/main.cc
索引 cbd42bd..fa6f1db 100644
```

```
6. std::cc                                                
```

4. if (i <n) y[i] = a*x[i] + y[i]; 5. } 6. 7. 全局函数限定符 void 缩放(int n, float
*x, float *a) 8. { 9. int i =blockIdx.x*blockDim.x + threadIdx.x; 10. if (i <n)

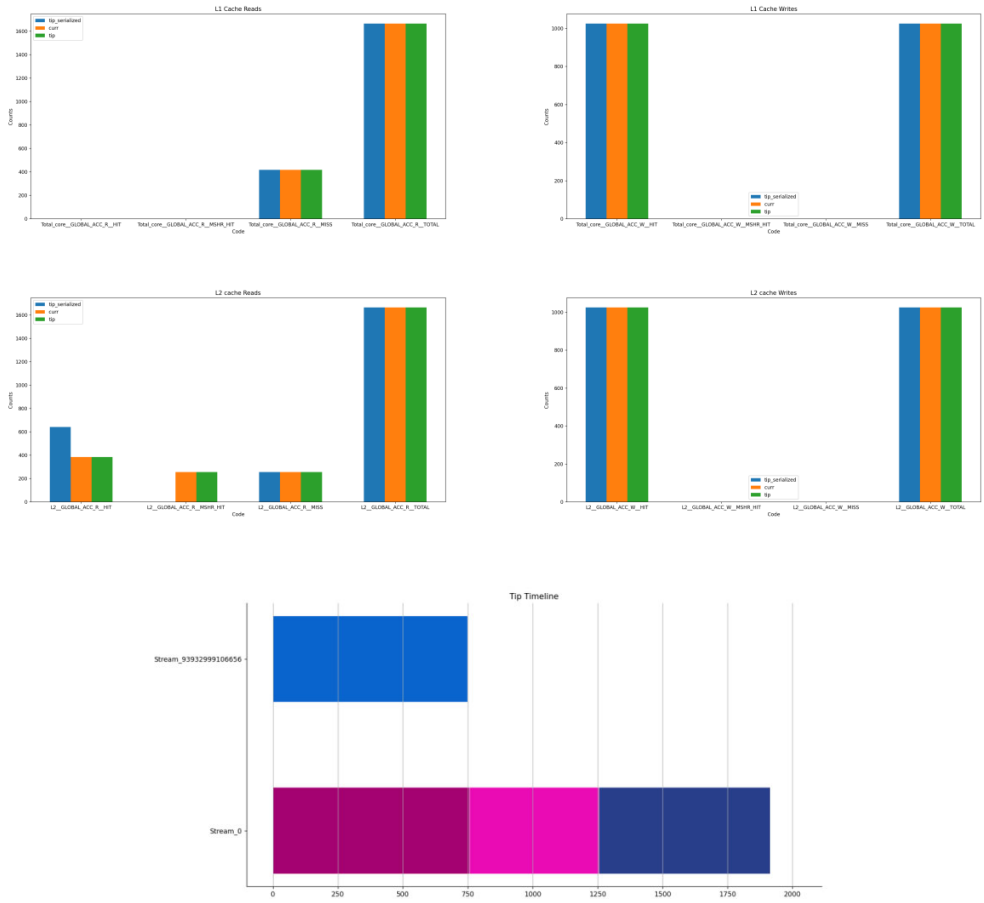

图 3： benchmark_1_stream.cu 验证结果。启用新功能（绿色）时的缓存访问计数与未应用我们的补丁（橙色）时完全一致。

基准测试_3_流.cu [5]:

```
全局函数限定符 void SAXPY(int n, float a,float *x, float *y)
{
 <escape>int i =blockIdx.x*blockDim.x + threadIdx.x;</escape>
 <escape>if (i <n) y[i] = a*x[i] + y[i];
</escape>
}

全局函数限定符 void 缩放(整数n, 浮点数 s, 浮点数 *a)
{
 int i = blockIdx.x*blockDim.x + threadIdx.x;
 如果 (i <n) a[i] = s*a[i];
}
```

12. 13. 全局函数限定符 void add(int n, float *a, float *b) 14. { 15. int i =

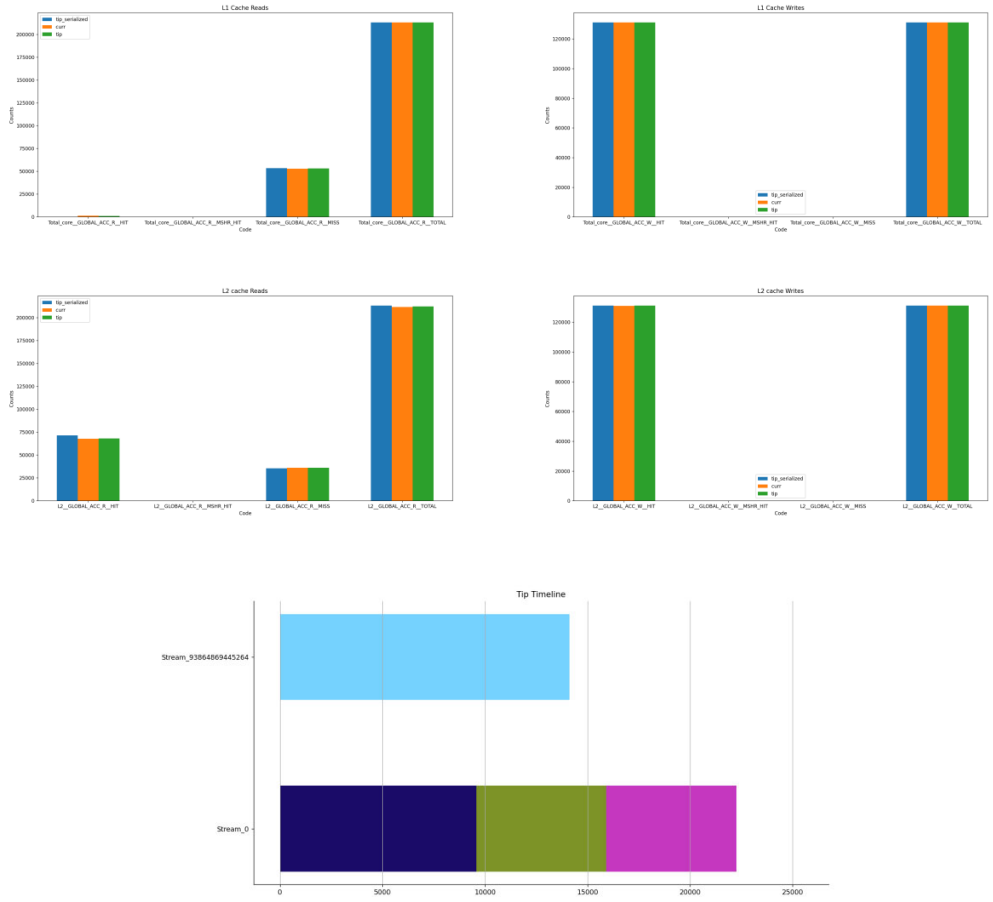

图4： 基准测试_3_stream.cu 验证结果。启用新功能（绿色）后的缓存访问计数与未应用补丁（橙色）时的对应计数完全一致。

我们对 benchmark-1-stream.cu 和 benchmark-3-stream.cu 也进行了类似的比较。在图3和图4中可以观察到，借助我们的每流统计跟踪功能，某些访问类型/结果组合的聚合计数有所增加：部分绿色柱状条略高于其对应的橙色柱状条。这些差异符合我们的预期，正如第1节所述——有时，当多个流并发运行，且来自不同流的统计数据在同一周期内被访问时，就会发生少计数现象。我们还多次重新运行了这两个基准测试的仿真，并发现由于基准测试源代码中的确定性问题，每次运行得到的数值略有不同。尽管如此，在所有情况下，我们统计得到的计数始终大于或等于来自干净代码库的统计数据，这进一步证实了，如果不进行我们的修改，Accel-Sim 往往会少计数缓存统计数据。

5.3 DeepBench 验证

最后，我们利用来自 DeepBench [3][4], 推理_半精度_35_1500_2560_0_0的跟踪，在包含多个流的更复杂工作负载下进一步验证我们的实现（图 5）。与先前的微基准测试不同，DeepBench 包含许多大型内核。因此，很难验证 DeepBench 的精确计数。不过，我们仍然对其进行研究，因为它可以作为合理性检查，确认我们的改动不会显著影响大型基准测试中的结果。

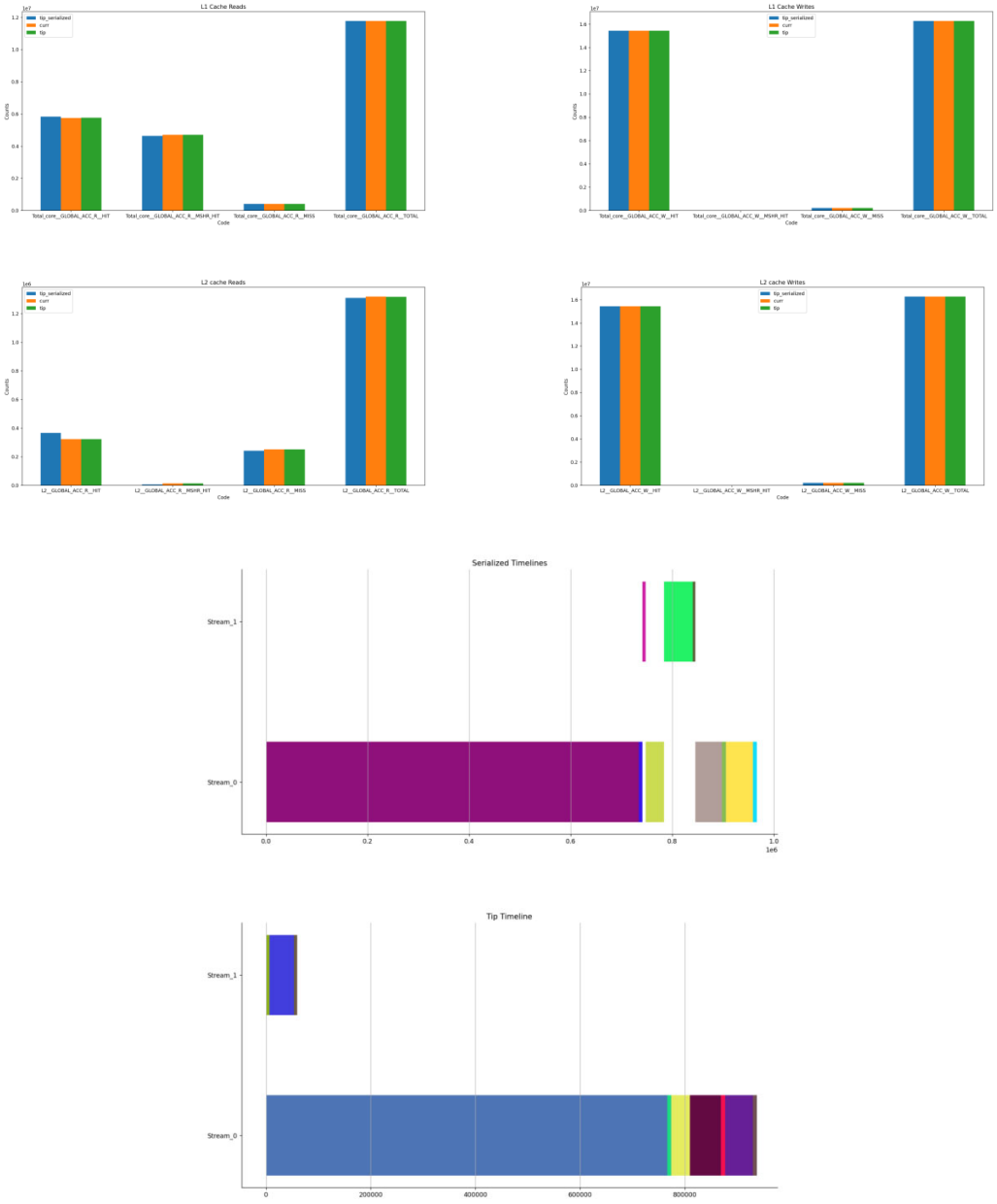

图 5： 推理_半精度_35_1500_2560_0_0 验证结果。请注意，时序图中每个内核的颜色都不同。

图 5 展示了 DeepBench 推理跟踪的结果。同样，我们功能的行为保持了与上述简单基准测试验证期间相同的趋势。此外，通过查看性能时间线，我们可以看到我们的更改正确地跟踪了重叠的内核——提供了不会按周期聚合的有用信息。

6. 局限性与后续工作

目前，我们仅支持跟踪缓存（第 3.1 节）和性能（第 3.2 节）的每流、每内核统计信息。尽管如此，由于我们前面提到的修改已将流 ID 传递至整个 GPGPU-Sim，因此也可以为其他组件（例如互连、主存）开发类似的功能扩展。

此外，由于 print_stats 函数现在需要一个流 ID 输入参数，power stats.cc 和 gpgpusim entrypoint.cc 可能会受到影响。这些_ _模块目前不了解流 ID，也尚未能够访问我们添加的get_流 ID 函数。添加更多支持后，应根据需要重新审视这些不完善之处。

7. 附录：绘图工具

使用绘图脚本生成了图 2 至图 5。如需进行更多验证，请在主函数中选择要绘制的统计数据，并使用类似以下命令执行脚本：

```
py .\graph.py .\l2_lat_4stream_tip_serialized.log .\l2_lat_4stream_clean.log .\l2_lat_4stream_tip.log
```

本报告中使用的绘图脚本和数据可在此处获取：
https://github.com/hal-uw/accel-sim-multiStream-stats

8. 参考文献

[1] M. Khairy、Z. Shen、T. M. Aamodt 和 T. G. Rogers，《Accel-Sim：用于经验证的 GPU 建模的可扩展模拟框架》，载于 2020 年 ACM/IEEE 第 47 届国际计算机体系结构年会研讨会，ISCA 系列，2020 年，第 473–486 页

[2] Jonathan Lew、Deval Shah、Suchita Pati、Shaylin Cattell、Mengchi Zhang、Amruth Sandhupatla、Christopher Ng、Negar Goli、Matthew D. Sinclair、Timothy G. Rogers、Tor M. Aamodt，使用详细的 GPU 模拟器分析机器学习工作负载，arXiv:1811.08933。

[3] Sharan Narang。DeepBench。https://github.com/baidu-research/DeepBench，2016。

[4] Sharan Narang 和 Greg Diamos。DeepBench 的更新，重点关注深度学习推理。https://svail.github.io/DeepBench-update/，2017 年。

[5] 通过更细粒度同步提高机器学习工作负载中的 GPU 利用率。Reese Kuper、Suchita Pati 和 Matthew D. Sinclair。发表于第三届青年架构师研讨会（YArch），2021年4月。