# User perspectives on GPU performance

Scalable Tools Workshop – Discussion Group Out-brief

# Perspective of “naïve user” trying to tune GPU code

- Data provided by vendor tools is overwhelming and complex – what does it all mean?
- LLNL application team process (approx.)
    - Example: new platform - AMD - porting from code that runs on NVIDIA
    - Compare theoreticals, if comparable performance to expectation, all good.
    - If drastically different, start a manual process to find out how it is limited
        - Memory access, compute, etc.
    - Didn’t have sufficient metrics (to date) on AMD side.
    - Want to do roofline (currently can’t).
    - Ask for the data, then ask for an explanation of the data.
    - Some people are focused on their piece of the code, some people are looking holistically.
- Other obvious process – porting from CPU to GPU

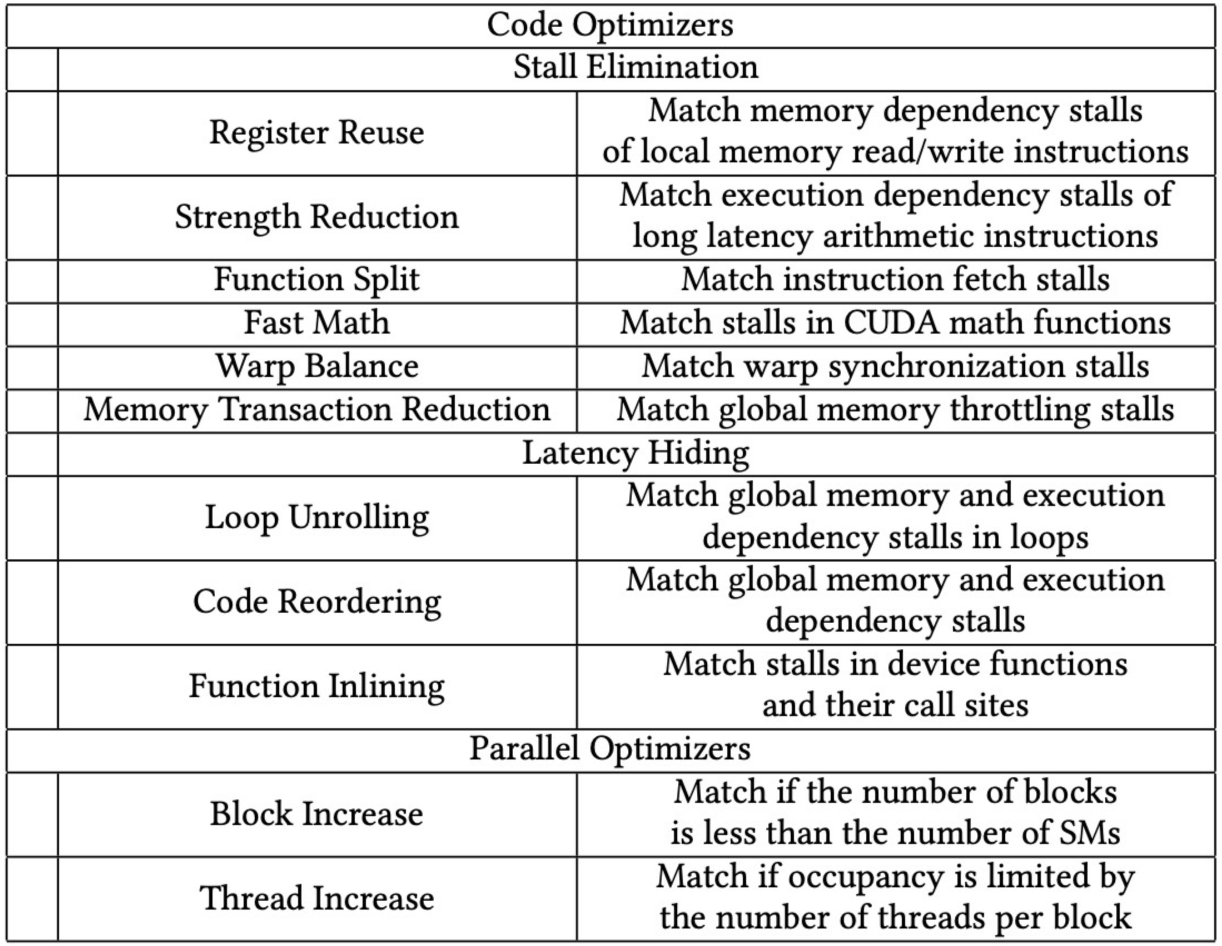
# Problem types

- Two types of kernels - those that fill the GPU and don’t overlap, and many small kernels that can
- LLNL Kernel types - C++ is using Raja. Remainder is Fortran+OpenMP offload. (Kokkos examples similar)
- What about tuning one kernel? Is roofline enough? Would loop nest information help?
  - Understanding register pressure is key
  - Differential analysis is important (GPU-GPU or CPU-GPU)

# Proposal – something like top-down or CPI stack analysis process

- GPA - gpu performance advisor
- Distinguishes stalls and leads back to potential causes and potential improvements
- https://github.com/Jokeren/GPA
- K. Zhou, X. Meng, R. Sai and J. Mellor-Crummey, "GPA: A GPU Performance Advisor Based on Instruction Sampling," *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Seoul, Korea (South)*, 2021, pp. 115-125, doi: 10.1109/CGO51591.2021.9370339.

**Table 2.** A brief description of GPU optimizers in GPA.

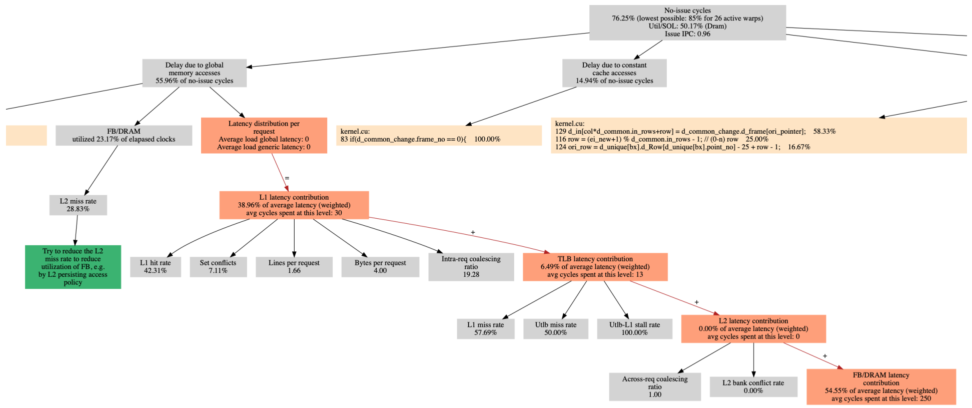


| Code Optimizers | | |
|---|---|---|
| | Stall Elimination | |
| | Register Reuse | Match memory dependency stalls of local memory read/write instructions |
| | Strength Reduction | Match execution dependency stalls of long latency arithmetic instructions |
| | Function Split | Match instruction fetch stalls |
| | Fast Math | Match stalls in CUDA math functions |
| | Warp Balance | Match warp synchronization stalls |
| | Memory Transaction Reduction | Match global memory throttling stalls |
| | Latency Hiding | |
| | Loop Unrolling | Match global memory and execution dependency stalls in loops |
| | Code Reordering | Match global memory and execution dependency stalls |
| | Function Inlining | Match stalls in device functions and their call sites |
| Parallel Optimizers | | |
| | Block Increase | Match if the number of blocks is less than the number of SMs |
| | Thread Increase | Match if occupancy is limited by the number of threads per block |

# Another example: Yueming Hao’s work

- Potential organization - assume GPU only for compute, only care about issue stalls

- https://github.com/drgpu/drgpu-artifact
- Top-down organization of metrics
- Latency bound, compute bound, bandwidth bound, contention bound…?


No-issue cycles
76.25% (lowest possible: 85% for 26 active warps)
Util/SOL: 50.17% (Dram)
Issue IPC: 0.96
Delay due to global memory accesses
55.96% of no-issue cycles
Delay due to constant cache accesses
14.94% of no-issue cycles
FB/DRAM
utilized 23.17% of elapased clocks
Latency distribution per request
Average load global latency: 0
Average load generic latency: 0
=
L2 miss rate
28.83%
Try to reduce the L2 miss rate to reduce utilization of FB, e.g. by L2 persisting access policy
L1 latency contribution
38.96% of average latency (weighted)
avg cycles spent at this level: 30
L1 hit rate
42.31%
Set conflicts
7.11%
Lines per request
1.66
Bytes per request
4.00
Intra-req coalescing ratio
19.28
+
TLB latency contribution
6.49% of average latency (weighted)
avg cycles spent at this level: 13
L1 miss rate
57.69%
Utlb miss rate
50.00%
Utlb-L1 stall rate
100.00%
+
L2 latency contribution
0.00% of average latency (weighted)
avg cycles spent at this level: 0
Across-req coalescing ratio
1.00
L2 bank conflict rate
0.00%
+
FB/DRAM latency contribution
54.55% of average latency (weighted)
avg cycles spent at this level: 250

```
kernel.cu:
83 if(d_common_change.frame_no == 0){    100.00%
```

```
kernel.cu:
129 d_in[col*d_common.in_rows+row] = d_common_change.d_frame[ori_pointer];    58.33%
116 row = (ei_new+1) % d_common.in_rows - 1; // (0-n) row    25.00%
124 ori_row = d_unique[bx].d_Row[d_unique[bx].point_no] - 25 + row - 1;    16.67%
```

- Non issue cycles
  - Delay due to dependent instructions/issue rate
    - Integer
    - Fp32
    - Uniform
    - Memory
  - Waiting at barrier
  - Delay due to global memory access
    - Latency per request
    - Occupancy
  - etc.

# Final suggestion

- Need something like profiler guided optimization for GPU