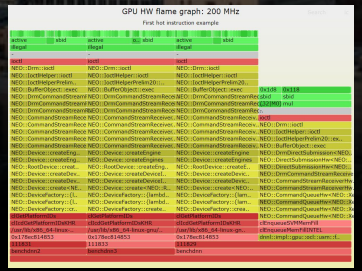
LINUX PLUMBERS CONFERENCE

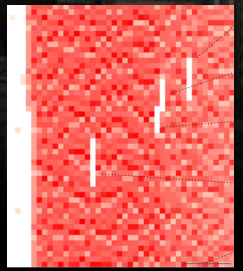
TOKYO, JAPAN / DECEMBER 11-13, 2025

AI Flame Graphs With eBPF

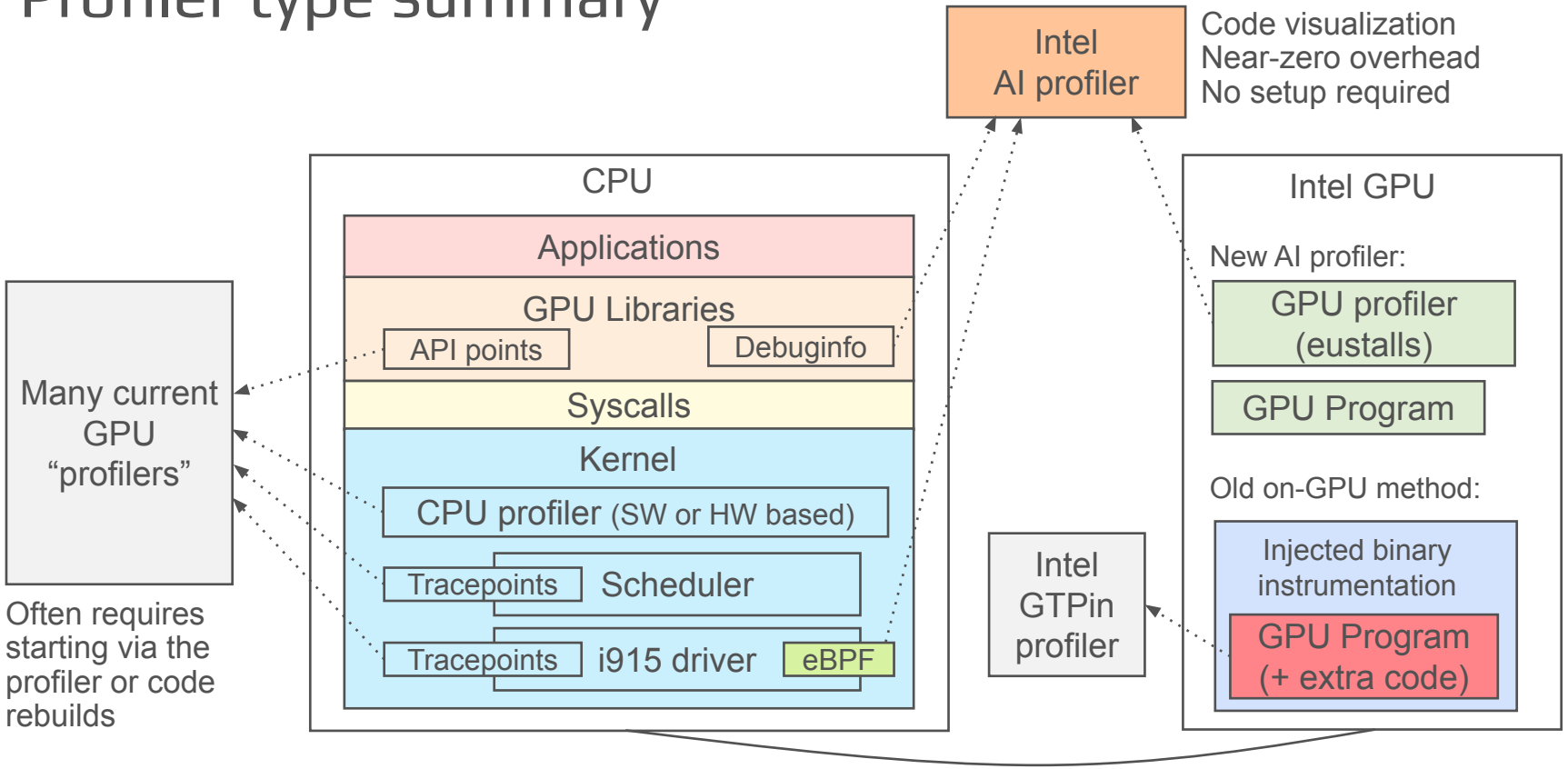
Brendan Gregg, Ben Olson

Dec 2025

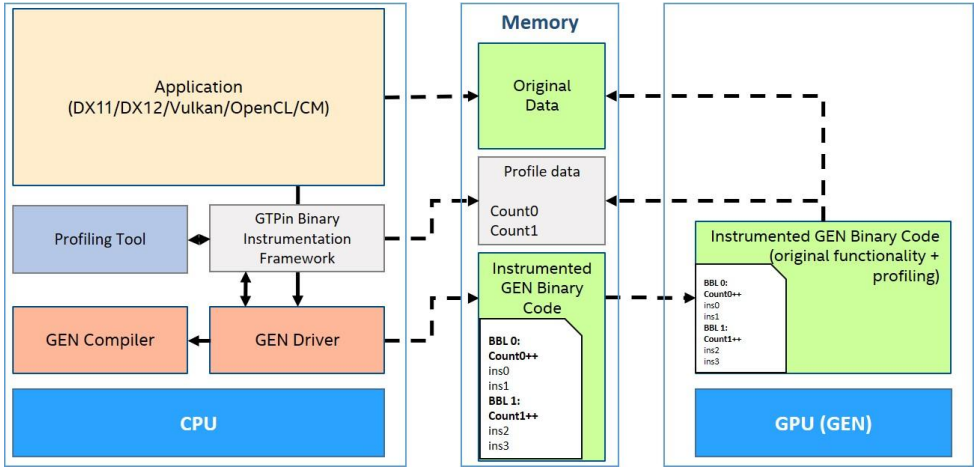




Profiler type summary



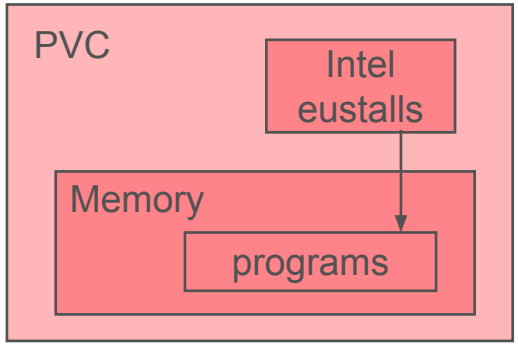
Other profilers (e.g., GTPin)



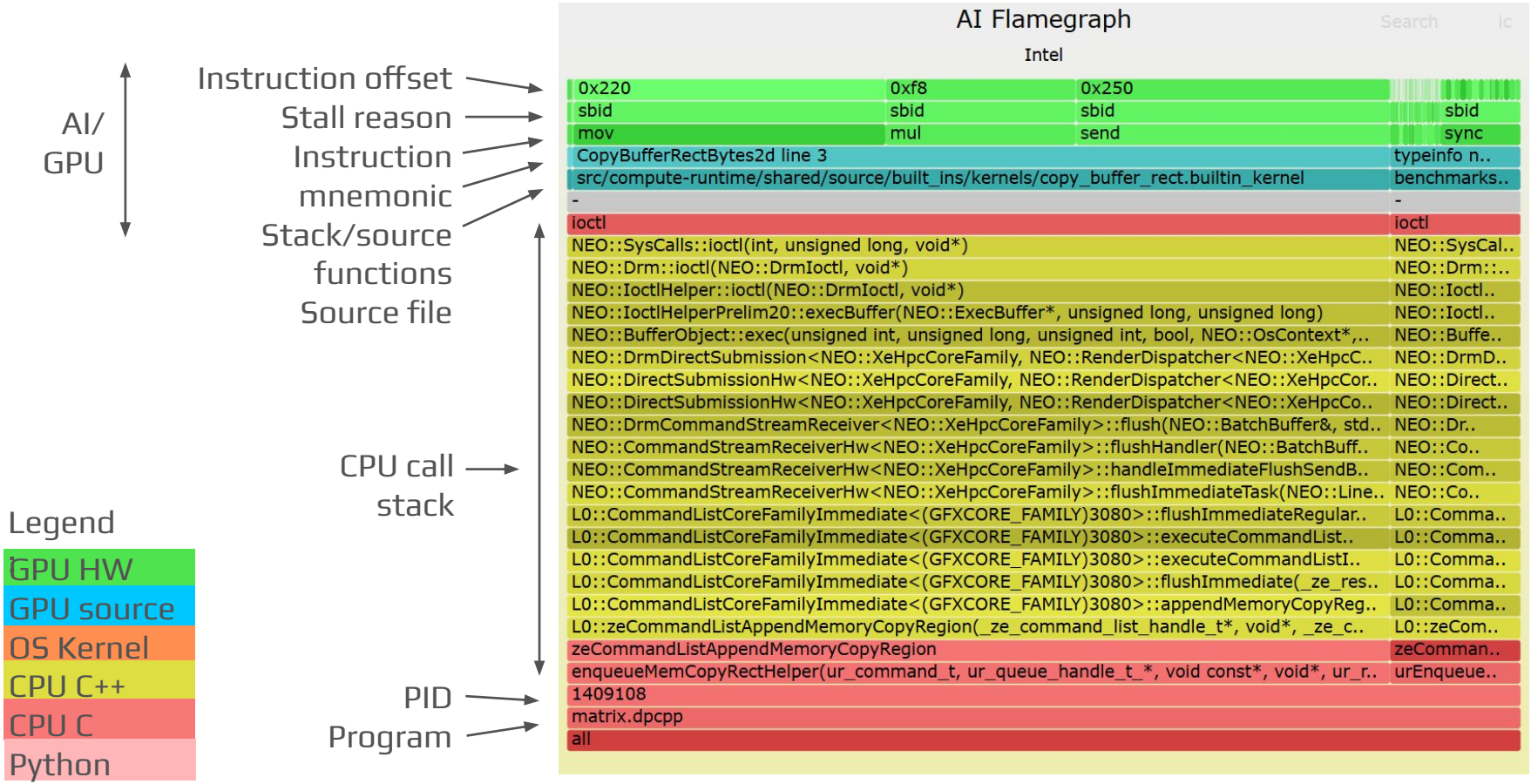
Source: https://software.intel.com/sites/landingpage/gtpin/index.html

Does show GPU software internals,
but costs high overhead.

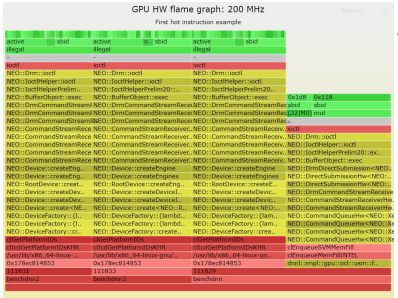
Intel AI Profiler



Programs run as-is.
HW-based profiling: Intel eustalls
(execution unit stalls).


AI Flamegraph
Search
ic
Intel
AI/
GPU
Instruction offset
Stall reason
Instruction
mnemonic
Stack/source
functions
Source file
CPU call
stack
PID
Program
0x220
0xf8
0x250
sbid
sbid
sbid
sbid
mov
mul
send
sync
CopyBufferRectBytes2d line 3
typeinfo n..
src/compute-runtime/shared/source/built_ins/kernels/copy_buffer_rect.builtin_kernel
benchmarks..
-
-
ioctl
ioctl
NEO::SysCalls::ioctl(int, unsigned long, void*)
NEO::SysCal..
NEO::Drm::ioctl(NEO::DrmIoctl, void*)
NEO::Drm::..
NEO::IoctlHelper::ioctl(NEO::DrmIoctl, void*)
NEO::Ioctl..
NEO::IoctlHelperPrelim20::execBuffer(NEO::ExecBuffer*, unsigned long, unsigned long)
NEO::Ioctl..
NEO::BufferObject::exec(unsigned int, unsigned long, unsigned int, bool, NEO::OsContext*,..
NEO::Buffe..
NEO::DrmDirectSubmission<NEO::XeHpcCoreFamily, NEO::RenderDispatcher<NEO::XeHpcC..
NEO::DrmD..
NEO::DirectSubmissionHw<NEO::XeHpcCoreFamily, NEO::RenderDispatcher<NEO::XeHpcCor..
NEO::Direct..
NEO::DirectSubmissionHw<NEO::XeHpcCoreFamily, NEO::RenderDispatcher<NEO::XeHpcCo..
NEO::Direct..
NEO::DrmCommandStreamReceiver<NEO::XeHpcCoreFamily>::flush(NEO::BatchBuffer&, std..
NEO::Dr..
NEO::CommandStreamReceiverHw<NEO::XeHpcCoreFamily>::flushHandler(NEO::BatchBuff..
NEO::Co..
NEO::CommandStreamReceiverHw<NEO::XeHpcCoreFamily>::handleImmediateFlushSendB..
NEO::Com..
NEO::CommandStreamReceiverHw<NEO::XeHpcCoreFamily>::flushImmediateTask(NEO::Line..
NEO::Co..
L0::CommandListCoreFamilyImmediate<(GFXCORE_FAMILY)3080>::flushImmediateRegular..
L0::Comma..
L0::CommandListCoreFamilyImmediate<(GFXCORE_FAMILY)3080>::executeCommandList..
L0::Comma..
L0::CommandListCoreFamilyImmediate<(GFXCORE_FAMILY)3080>::executeCommandListI..
L0::Comma..
L0::CommandListCoreFamilyImmediate<(GFXCORE_FAMILY)3080>::flushImmediate(_ze_res..
L0::Comma..
L0::CommandListCoreFamilyImmediate<(GFXCORE_FAMILY)3080>::appendMemoryCopyReg..
L0::Comma..
L0::zeCommandListAppendMemoryCopyRegion(_ze_command_list_handle_t*, void*, _ze_c..
L0::zeCom..
zeCommandListAppendMemoryCopyRegion
zeComman..
enqueueMemCopyRectHelper(ur_command_t, ur_queue_handle_t_*, void const*, void*, ur_r..
urEnqueue..
1409108
matrix.dpcpp
all
Legend
GPU HW
GPU source
OS Kernel
CPU C++
CPU C
Python

Reading a full (inverted) GPU Flame Graph



“why”

“how”

Program name	“This application
PID	
CPU call stack (user)	…uses the GPU for *this* reason
CPU call stack (kernel)	…via this driver
-	
GPU source directories	…and runs *this* GPU application
GPU source file	…from *this* source file
GPU function stack	…and runs *this* function
GPU instruction mnemonic	…which runs *this* instruction
GPU stall reason	…and is slow for *this* reason
GPU instruction offset	…for these *exact* instructions.”

AI Profiler: Requirements

Near-zero overhead

FAANGs target <0.1% overhead for profilers. A mere 5% overhead can trigger “bad instance” detection and auto termination. Methods involving baked-in instruction instrumentation are non-starters. We keep overhead low using eBPF and eustalls.

Easy to deploy

The profiler should not require special deployments such as requiring the developer SSH to their servers and modify their application start scripts. Most developers don’t SSH anymore and some sites have completely removed SSH, plus many sites use continuous delivery UIs leaving the developer with no idea how to manually modify the application start process to include some Intel library/layer. We used advanced eBPF to automatically instrument the kernel, no restarts of anything required. This method is becoming known nowadays as “**zero instrumentation**.”

Easy to understand

The tool must speak the language of the developer: show them their own code function/method names.

Actual GPU HW profiling

Many current GPU profilers do not see inside the HW, and merely time requests. They are not “profilers” in the common use of the term. Our solution actually sees inside the HW thanks to Intel eustalls.

Complete: CPU & GPU

Full stack visibility, to answer why (GPU) and how (GPU).

How

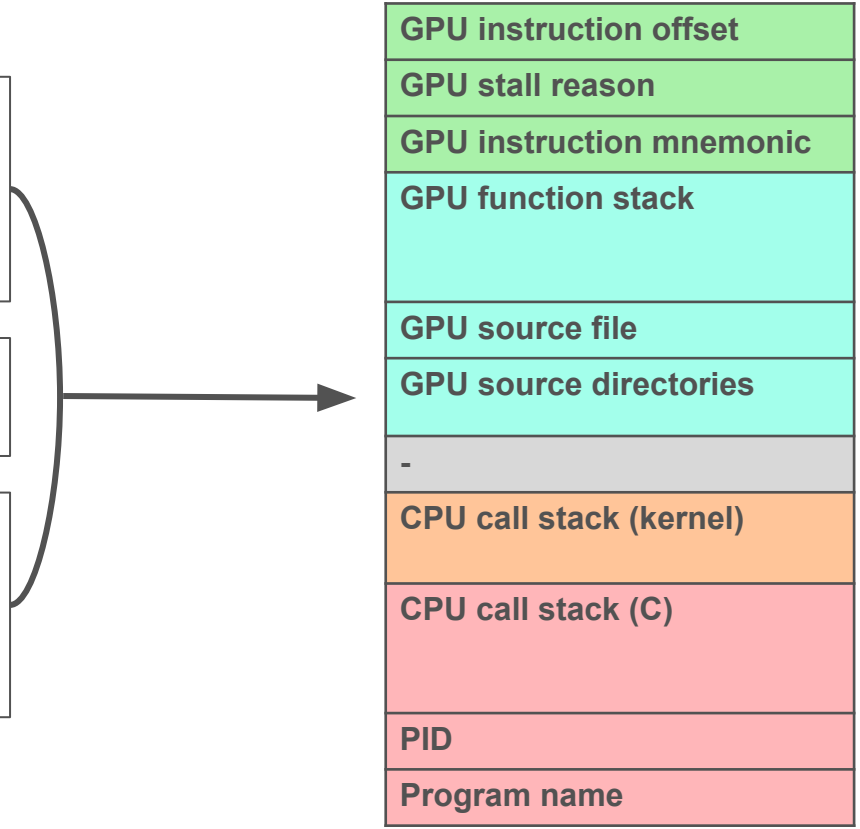
Stall Sample

```
GPU Address (minus top bits)
Stall Reason
Count
```

Userspace Stacks of Kernel Driver Calls

ELF of GPU Program

```
Full GPU Address
Native GPU Binary
GPU Symbols (maybe!)
```



A Tale of Two Methods

uprobes

Probe runtime writing commands

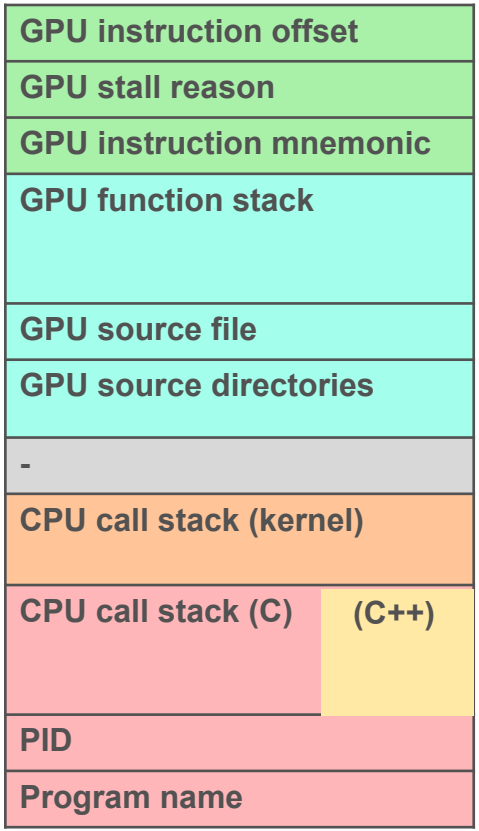
Extract program addresses and names

kprobes

- Probe where all buffer objects get allocated by the kernel driver
- Probe where all buffer objects get bound to a VM
- Probe where commands get sent to the GPU
- Implement a simulator of a GPU Command Streamer, including:
  - Registers
  - Feature flags
  - Base addresses
  - Per-context instruction pointers
  - 3D commands
  - Program execution
  - Semaphores
  - Jumps
- Ensure the simulator doesn't run forever
- Oops! Can't page fault.
- Oops! Runtime hasn't written commands there yet.
- Ensure the simulator sees every single buffer object allocation, binding, and execution
- Ensure overhead of simulator doesn't grind workload to a halt
- Maintain BPF maps of GPU address ranges containing programs
- Send GPU address ranges of programs to userspace
- Parse ELFs from a special driver feature to associate GPU addresses with program names

Caveats

Need to collect ELFs of all GPU programs, requires expensive cooperation with the kernel driver (Xe).



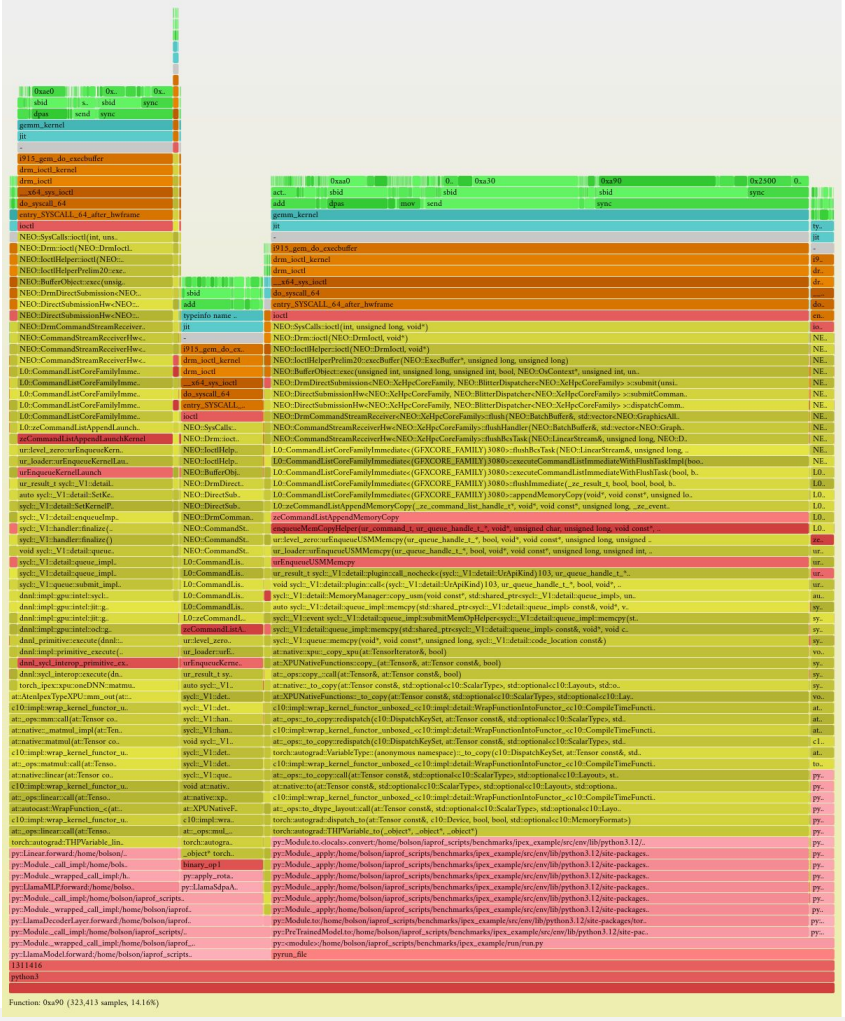
Underlying hardware sampling on Intel hardware, eustalls, does not provide a context ID to differentiate between simultaneous workloads. Workarounds in SW, but need HW changes.

Currently using GPU kernel name, with very little support for actual callstacks in GPU code.

A few choices on which CPU callstack to use: buffer object allocation, execbuffer, or maybe something else?

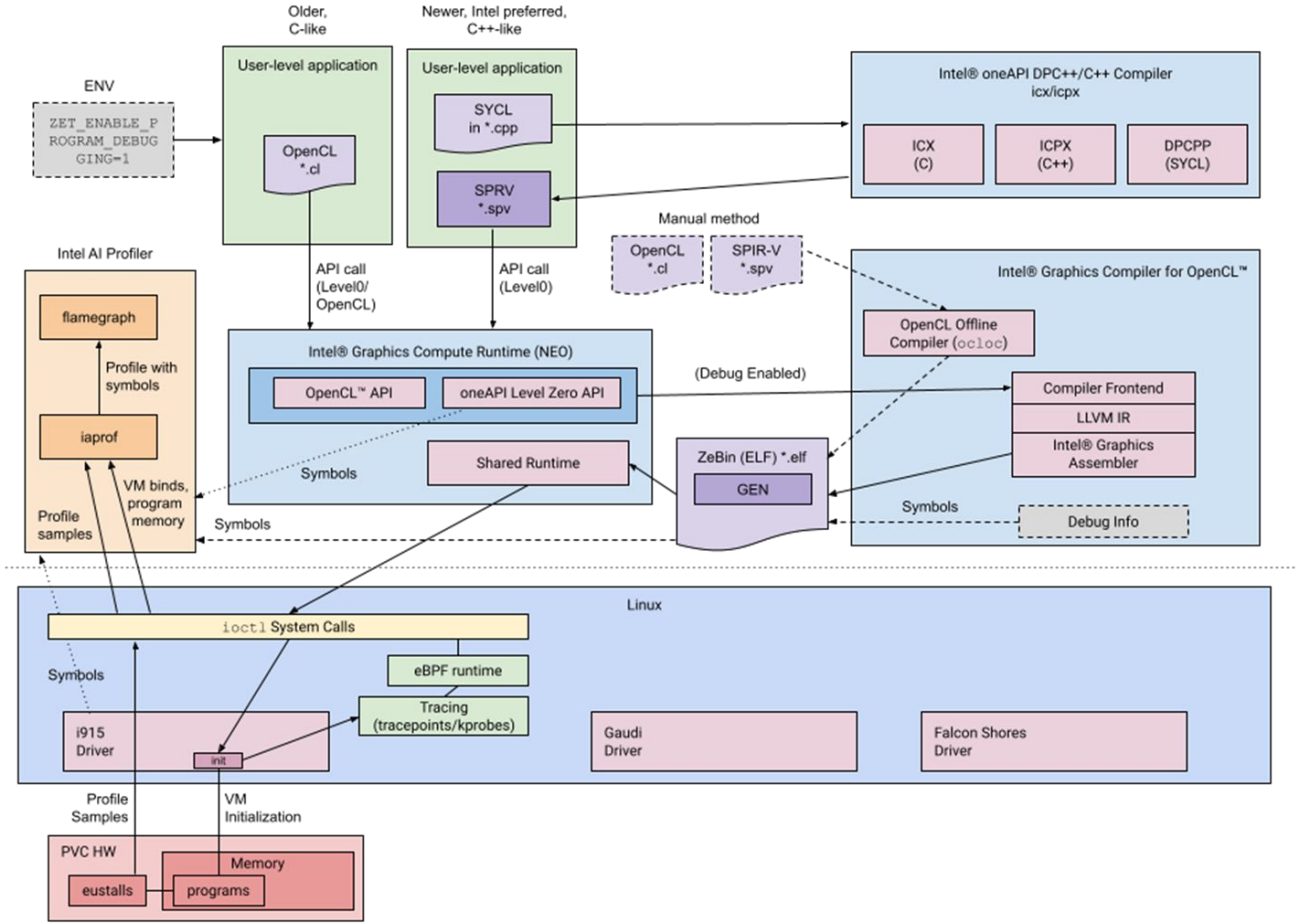
CPU-side callstacks currently require frame pointers built into *all* layers of the stack.

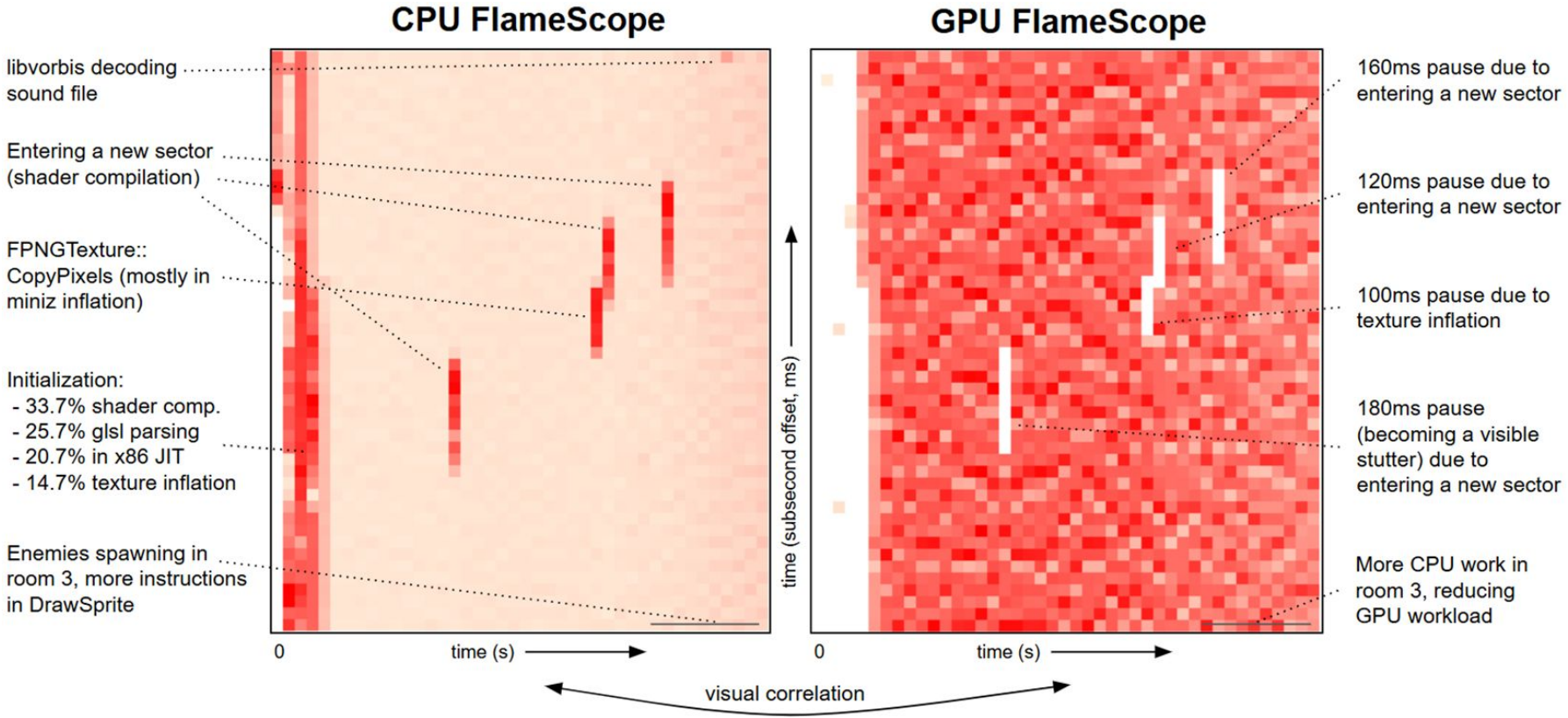
PyTorch



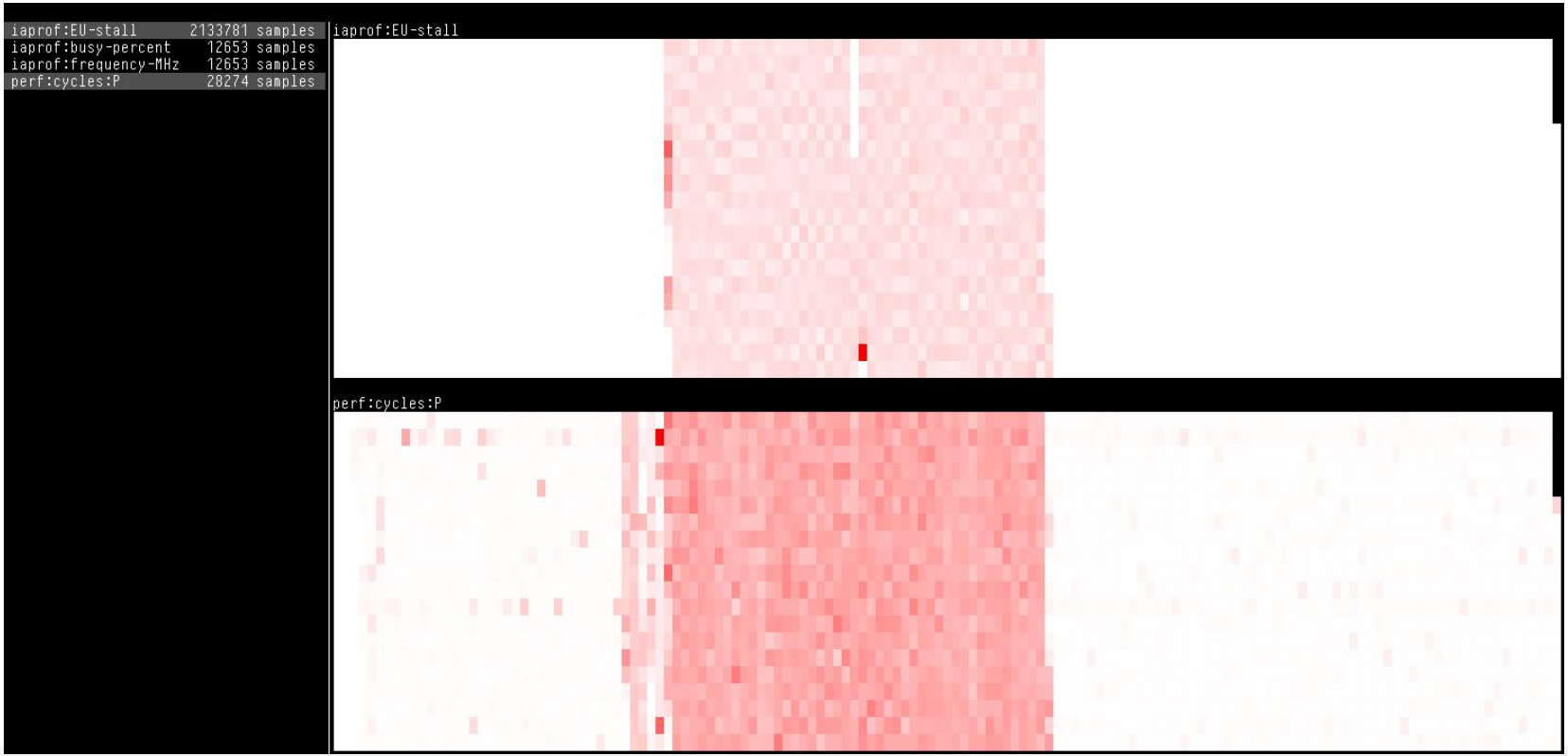
Function: 0xa90 (323,413 samples, 14.16%)

Intel AI Profiler Internals




CPU FlameScope
GPU FlameScope
libvorbis decoding sound file
Entering a new sector (shader compilation)
FPNGTexture:: CopyPixels (mostly in miniz inflation)
Initialization:
- 33.7% shader comp.
- 25.7% glsl parsing
- 20.7% in x86 JIT
- 14.7% texture inflation
Enemies spawning in room 3, more instructions in DrawSprite
160ms pause due to entering a new sector
120ms pause due to entering a new sector
100ms pause due to texture inflation
180ms pause (becoming a visible stutter) due to entering a new sector
More CPU work in room 3, reducing GPU workload
time (subsecond offset, ms)
0
time (s)
0
time (s)
visual correlation

Proviz (demo?)


iaprof:EU-stall 2133781 samples
iaprof:busy-percent 12653 samples
iaprof:frequency-MHz 12653 samples
perf:cycles:P 28274 samples
iaprof:EU-stall
perf:cycles:P

Proviz

0
0x418
active
dist.. pip..
active
sbid
di.. ..
active
.. d
a..
a
[failed decode]
mov
add
mul
.. .. a s s m
a a a a
typeinfo name for dequantize_mul_mat_vec_q4_0_sycl(void const*, float const*, float*, int, int, sycl::_V1::queue*)::'lambda'(sycl::_V1::nd_item<3>)
m
mov mul .. ..
/data/projects/iaprof/llama/llama.cpp/ggml/src/ggml-sycl/dmmv.cpp
typ..
typeinfo name for ..
_
/da..
/data/projects/iap..
L0::CommandListCoreFamily<(GFXCORE_FAMILY)3081>::appendLaunchKernelWithParams(L0::Kernel*, _ze_group_count_t const&, L0::Event*, L0::CmdListKernelLaunchParams&)
L0::CommandListCoreFamily<(GFXCORE_FAMILY)3081>::appendLaunchKernel(_ze_kernel_handle_t*, _ze_group_count_t const&, _ze_event_handle_t*, unsigned int, _ze_eve..
L0:..
L0::CommandListCor..
L0::CommandListCoreFamilyImmediate<(GFXCORE_FAMILY)3081>::appendLaunchKernel(_ze_kernel_handle_t*, _ze_group_count_t const&, _ze_event_handle_t*, unsigned int..
L0::zeCommandListAppendLaunchKernel(_ze_command_list_handle_t*, _ze_kernel_handle_t*, _ze_group_count_t const*, _ze_event_handle_t*, unsigned int, _ze_event_h..
zeCommandListAppendLaunchKernel
L0::zeCommandListA..
ur_command_list_manager::appendKernelLaunchUnlocked(ur_kernel_handle_t_*, unsigned int, unsigned long const*, unsigned long const*, unsigned long const*, unsi..
zeC..
zeCommandListAppen..
ur_command_list_manager::appendKernelLaunch(ur_kernel_handle_t_*, unsigned int, unsigned long const*, unsigned long const*, unsigned long const*, unsigned int..
ur_..
ur_command_list_ma..
v2::ur_queue_immediate_in_order_t::enqueueKernelLaunch(ur_kernel_handle_t_*, unsigned int, unsigned long const*, unsigned long const*, unsigned long const*, u..
ur::level_zero::urEnqueueKernelLaunch(ur_queue_handle_t_*, ur_kernel_handle_t_*, unsigned int, unsigned long const*, unsigned long const*, unsigned long const..
v2::..
v2::ur_queue_immed..
urEnqueueKernelLaunch
ur::..
ur::level_zero::ur..
sycl::_V1::detail::enqueueImpKernel(sycl::_V1::detail::queue_impl&, sycl::_V1::detail::NDRDescT&, std::vector<sycl::_V1::detail::ArgDesc, std::allocator<sycl::..
urE..
urEnqueueKernelLau..
sycl::_V1::handler::finalize()::'lambda'()::operator()() const::'lambda'()::operator()() const
syc..
sycl::_V1::detail::..
sycl::_V1::handler::finalize()::'lambda'()::operator()() const
sycl::_V1::handler..
sycl::_V1::handler::finalize() (.localalias)
std::shared_ptr<sycl::_V1::detail::event_impl> sycl::_V1::detail::queue_impl::finalizeHandlerInOrderNoEventsUnlocked<sycl::_V1::handler>(sycl::_V1::handler&)
sycl::_V1::detail::queue_impl::submit_impl(sycl::_V1::detail::type_erased_cgfo_ty const&, bool, sycl::_V1::detail::code_location const&, bool, sycl::_V1::deta..
std..
std::shared_ptr<sy..
sycl::_V1::queue::submit_with_event_impl(sycl::_V1::detail::type_erased_cgfo_ty const&, sycl::_V1::detail::v1::SubmissionInfo const&, sycl::_V1::detail::code_..
sycl::_V1::event sycl::_V1::queue::submit_with_event<sycl::_V1::ext::oneapi::experimental::properties<sycl::_V1::ext::oneapi::experimental::detail::properties..
sycl::_V1::queue::..
ggml_sycl_op_dequantize_mul_mat_vec(ggml_backend_sycl_context&, ggml_tensor const*, ggml_tensor const*, ggml_tensor*, char const*, float const*, char const*, ..
sycl::_V1::event s..
void ggml_sycl_op_mul_mat<no_quantize_q8_1>(ggml_backend_sycl_context&, ggml_tensor const*, ggml_tensor const*, ggml_tensor*, void (*)(ggml_backend_sycl_conte..
ggml_sycl_cpy(ggml_..
ggml_sycl_mul_mat(ggml_backend_sycl_context&, ggml_tensor const*, ggml_tensor const*, ggml_tensor*)
ggml_sycl_dup(ggml_..
ggml_backend_sycl_graph_compute_impl(ggml_backend_sycl_context*, ggml_cgraph*)
ggml_backend_sycl_graph_compute(ggml_backend*, ggml_cgraph*)
ggml_backend_sched_graph_compute_async
llama_context::graph_compute(ggml_cgraph*, bool)
llama_context::process_ubatch(llama_ubatch const&, llm_graph_type, llama_memory_context_i*, ggml_status&)
llama_context::decode(llama_batch const&)
llama_decode
main
__libc_init_first
__libc_start_main
_start
1338469
llama-cli
all

Current Release

GPU profiler is called `iaprof` (Intel Accelerator Profiler)

- Needs frame pointers for full CPU stacktraces
- Overhead is high by default, needs to parse all batchbuffers
- https://github.com/Intel/iaprof

Proviz

- https://github.com/kammerdienerb/proviz
- Also supports `perf script` output

```
# sudo iaprof record -i 10 1> profile.txt 2> output.txt &
# [Run workload]
# sudo kill $(pidof iaprof)
# proviz report profile.txt
```

intel