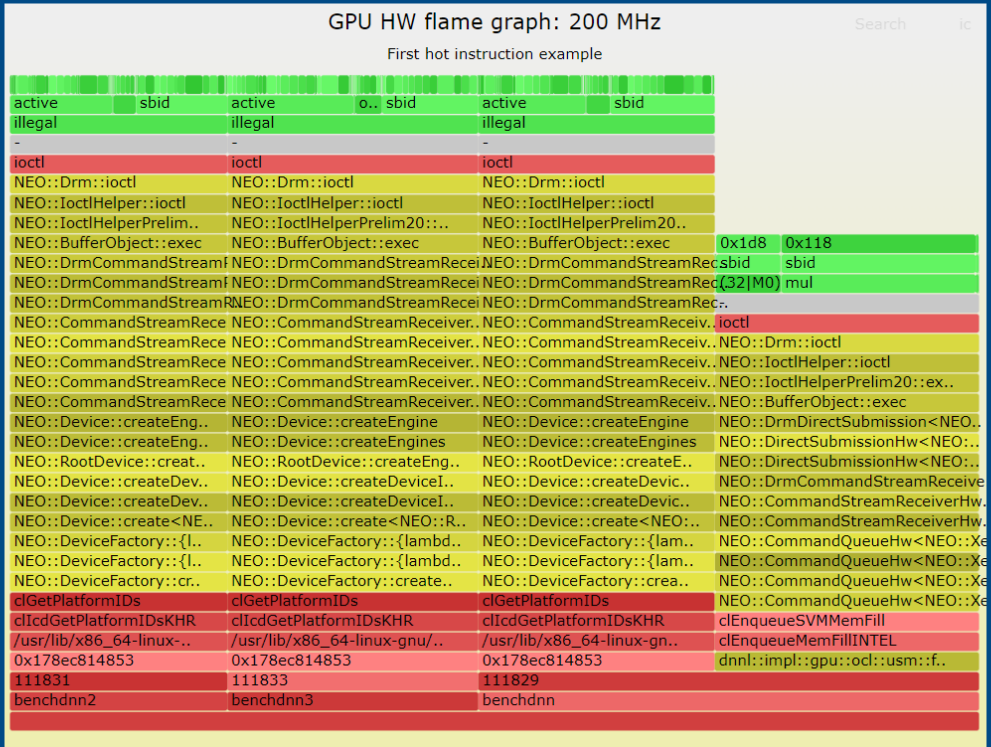
AI Profiler Summary

SATG (draft)

Brendan Gregg, Ben Olson, Brandon Kammerdiener, Gabriel Munoz

Nov 2024

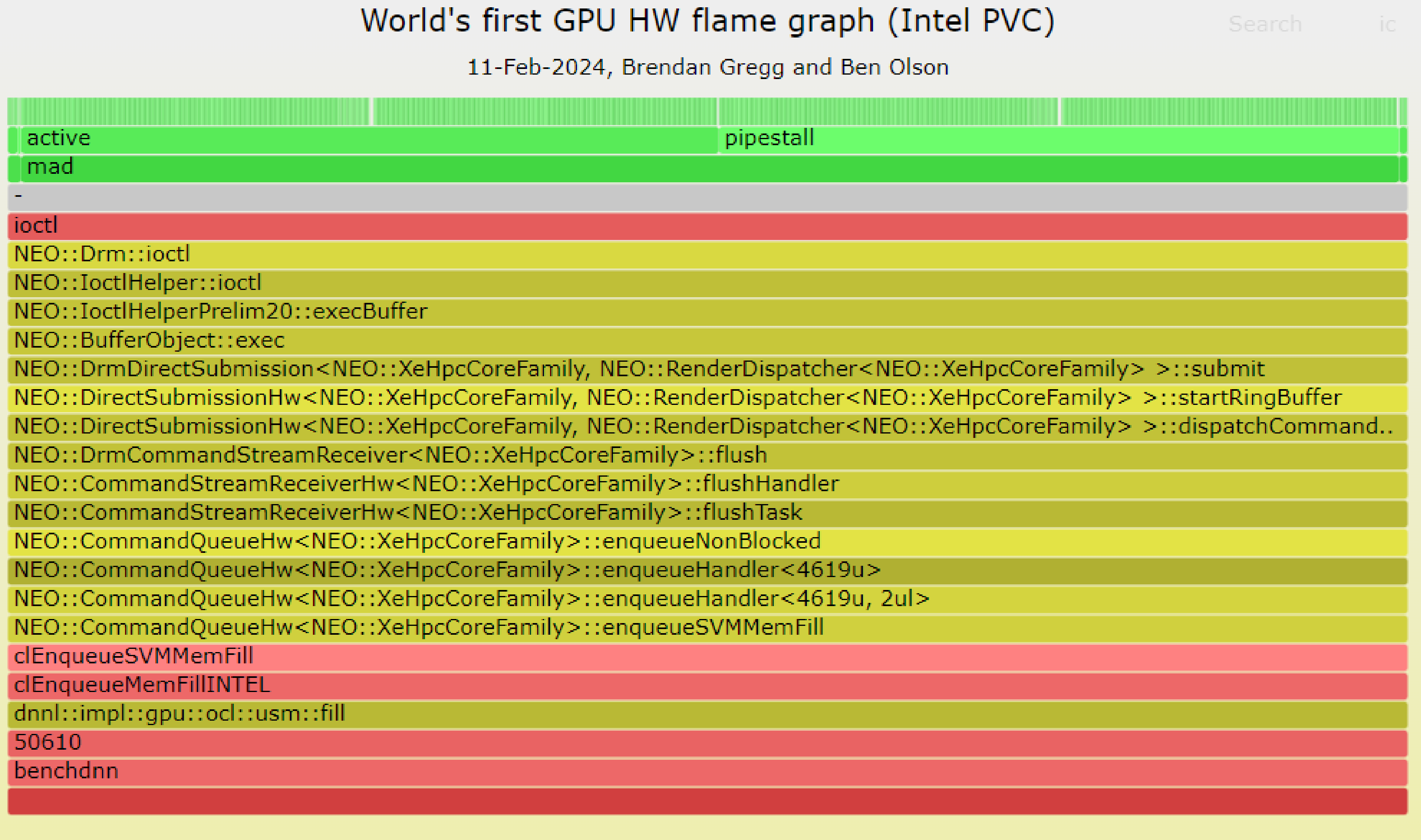


GPU HW flame graph: 200 MHz

First hot instruction example

intel®

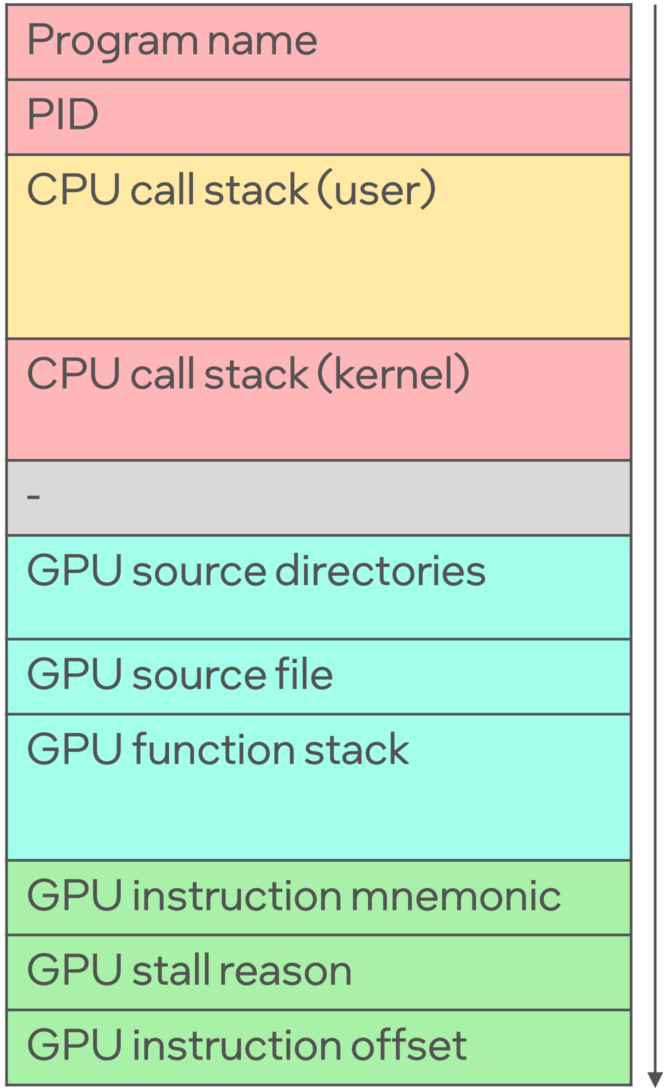
Public version of these slides


World's first GPU HW flame graph (Intel PVC)

11-Feb-2024, Brendan Gregg and Ben Olson

(Most basic example)

Reading a full (inverted) GPU Flame Graph



Program name

PID

CPU call stack (user)

CPU call stack (kernel)

-

GPU source directories

GPU source file

GPU function stack

GPU instruction mnemonic

GPU stall reason

GPU instruction offset

"why"

"how"

"This application

...uses the GPU for *this* reason

...via this driver

...and runs *this* GPU application

...from *this* source file

...and runs *this* function

...which runs *this* instruction

...and is slow for *this* reason

...for these *exact* instructions."

The Dream

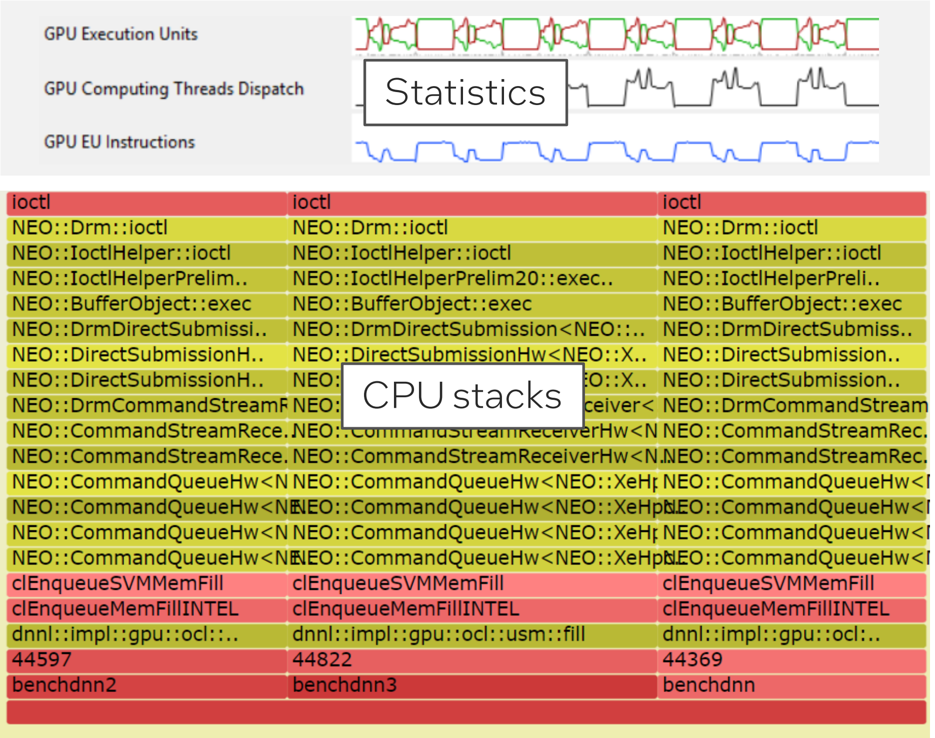
World’s first easy AI profiler

A. **Code visualization**
B. **Near-zero overhead**
C. **No setup required**

...same as CPU profiling

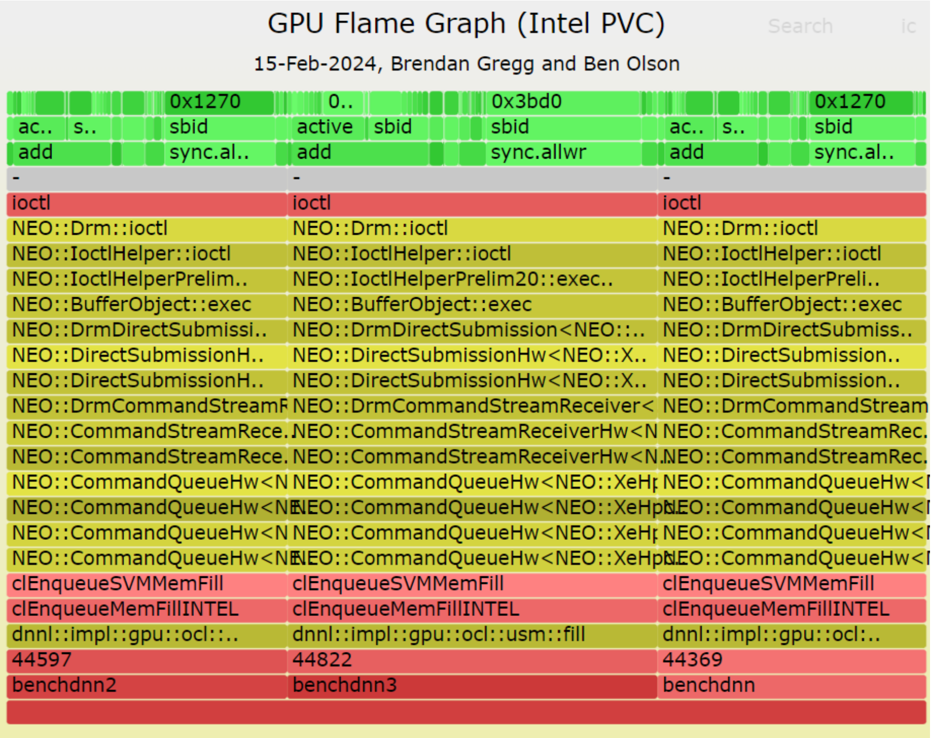
A) Code visualization

Most prior profilers


GPU Execution Units
GPU Computing Threads Dispatch
GPU EU Instructions
Statistics
CPU stacks
ioctl | ioctl | ioctl
NEO::Drm::ioctl | NEO::Drm::ioctl | NEO::Drm::ioctl
NEO::IoctlHelper::ioctl | NEO::IoctlHelper::ioctl | NEO::IoctlHelper::ioctl
NEO::IoctlHelperPrelim.. | NEO::IoctlHelperPrelim20::exec.. | NEO::IoctlHelperPreli..
NEO::BufferObject::exec | NEO::BufferObject::exec | NEO::BufferObject::exec
NEO::DrmDirectSubmissi.. | NEO::DrmDirectSubmission<NEO::.. | NEO::DrmDirectSubmiss..
NEO::DirectSubmissionH.. | NEO::DirectSubmissionHw<NEO::X.. | NEO::DirectSubmission..
NEO::DirectSubmissionH.. | NEO::DirectSubmission..
NEO::DrmCommandStreamR | NEO::DrmCommandStream
NEO::CommandStreamRece. | NEO::CommandStreamRec.
NEO::CommandStreamRece. | NEO::CommandStreamReceiverHw<N | NEO::CommandStreamRec.
NEO::CommandQueueHw<N | NEO::CommandQueueHw<NEO::XeHp | NEO::CommandQueueHw<
clEnqueueSVMMemFill | clEnqueueSVMMemFill | clEnqueueSVMMemFill
clEnqueueMemFillINTEL | clEnqueueMemFillINTEL | clEnqueueMemFillINTEL
dnnl::impl::gpu::ocl::.. | dnnl::impl::gpu::ocl::usm::fill | dnnl::impl::gpu::ocl::..
44597 | 44822 | 44369
benchdnn2 | benchdnn3 | benchdnn

No detailed visibility of code on the GPU.
Numerical statistics as counters or graphs, and/or CPU stacks.

Intel AI Profiler

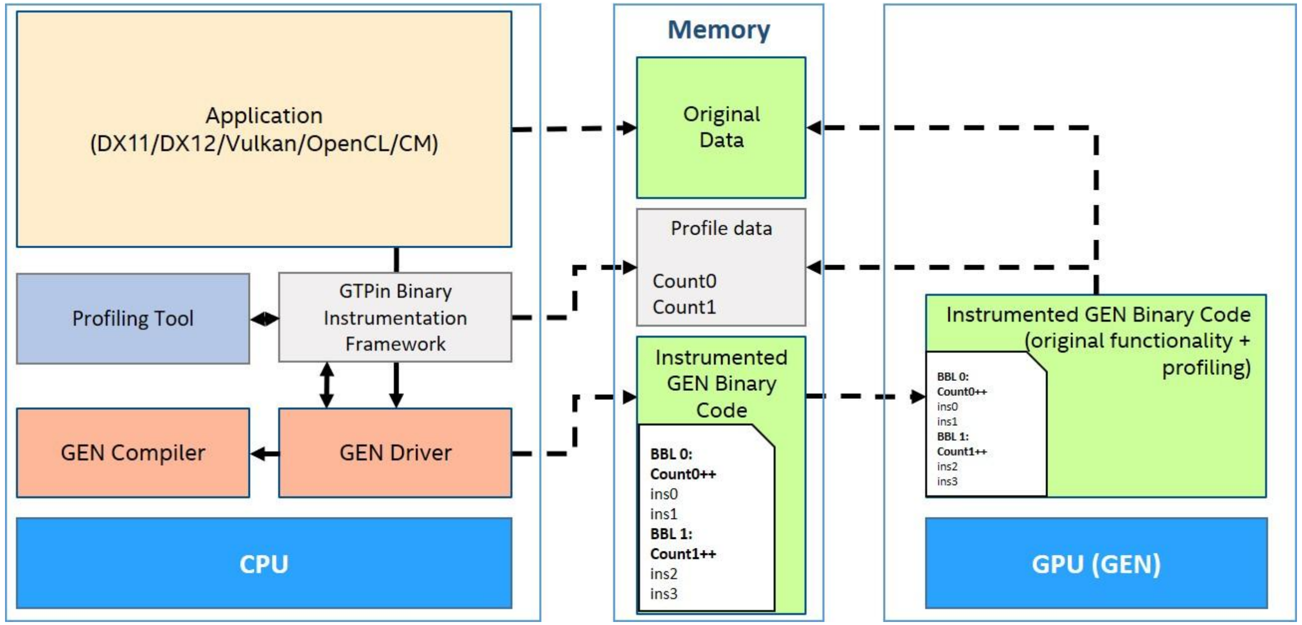

GPU Flame Graph (Intel PVC)
15-Feb-2024, Brendan Gregg and Ben Olson
Search
0x1270 | 0.. | 0x3bd0 | 0x1270
ac.. | s.. | sbid | active | sbid | sbid | ac.. | s.. | sbid
add | sync.al.. | add | sync.allwr | add | sync.al..
ioctl | ioctl | ioctl
NEO::Drm::ioctl | NEO::Drm::ioctl | NEO::Drm::ioctl
NEO::IoctlHelper::ioctl | NEO::IoctlHelper::ioctl | NEO::IoctlHelper::ioctl
NEO::IoctlHelperPrelim.. | NEO::IoctlHelperPrelim20::exec.. | NEO::IoctlHelperPreli..
NEO::BufferObject::exec | NEO::BufferObject::exec | NEO::BufferObject::exec
NEO::DrmDirectSubmissi.. | NEO::DrmDirectSubmission<NEO::.. | NEO::DrmDirectSubmiss..
NEO::DirectSubmissionH.. | NEO::DirectSubmissionHw<NEO::X.. | NEO::DirectSubmission..
NEO::DrmCommandStreamR | NEO::DrmCommandStreamReceiver< | NEO::DrmCommandStream
NEO::CommandStreamRece. | NEO::CommandStreamReceiverHw<N | NEO::CommandStreamRec.
NEO::CommandQueueHw<N | NEO::CommandQueueHw<NEO::XeHp | NEO::CommandQueueHw<
clEnqueueSVMMemFill | clEnqueueSVMMemFill | clEnqueueSVMMemFill
clEnqueueMemFillINTEL | clEnqueueMemFillINTEL | clEnqueueMemFillINTEL
dnnl::impl::gpu::ocl::.. | dnnl::impl::gpu::ocl::usm::fill | dnnl::impl::gpu::ocl::..
44597 | 44822 | 44369
benchdnn2 | benchdnn3 | benchdnn

All your code, CPU and GPU, in one visual.
Software and hardware complete.

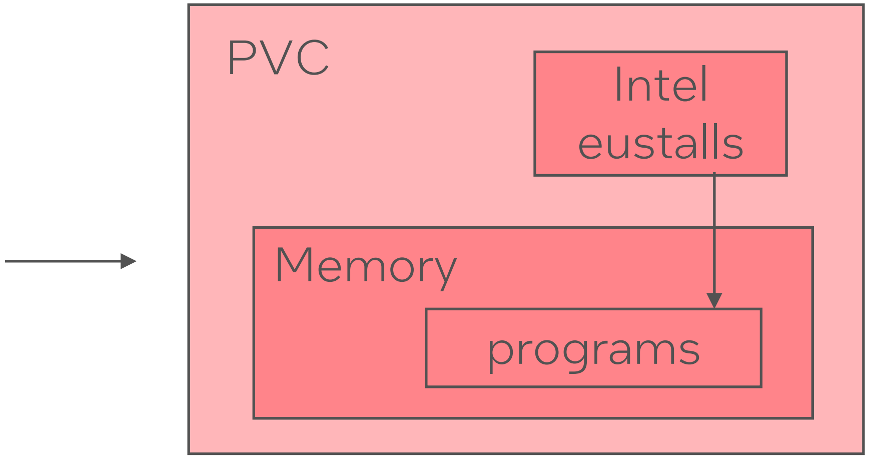
B) Near-zero overhead

Other profilers (e.g., GTPin)

Intel AI Profiler



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



Does show GPU software internals, but costs high overhead.

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

C) No setup required

Other profilers (incl. eustalls)

1. **Possibly rebuild the Linux kernel**
2. **Build program with debuginfo**
3. **Stop running program**
4. **Start via profiler**

Source includes: https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/set-up-system-for-gpu-analysis.html

(4) In today's cloud environment, this is often NOT easy. Developers may not have SSH access nor know where their programs are installed or how they are launched: it's all automated.

Intel AI Profiler

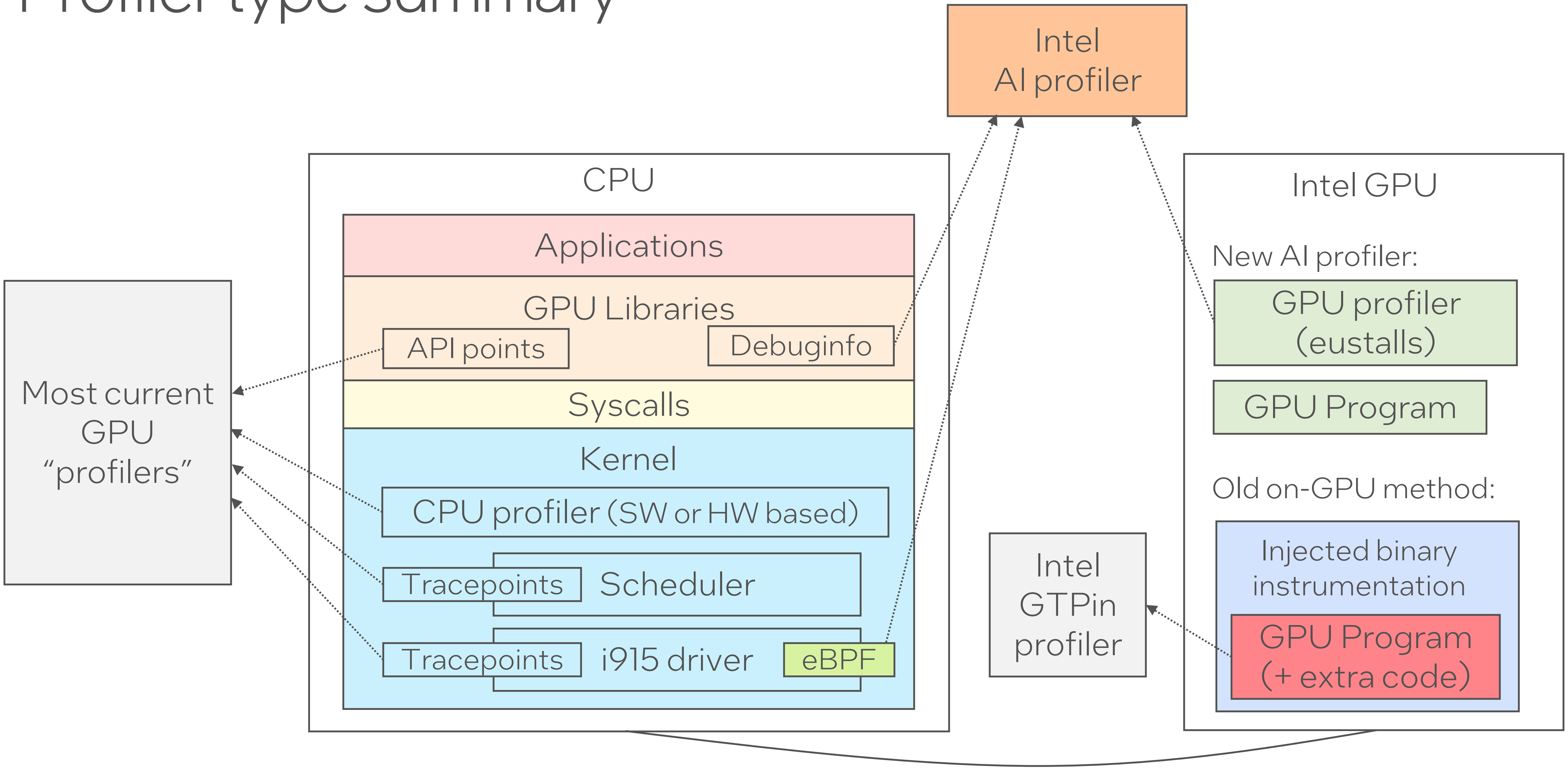
<NOTHING>

Full example:

1. Launch IDC instance
2. Click on gProfiler URL

That's the dream. Ease of use. Will need IGC changes to support symbols by default.

Profiler type summary



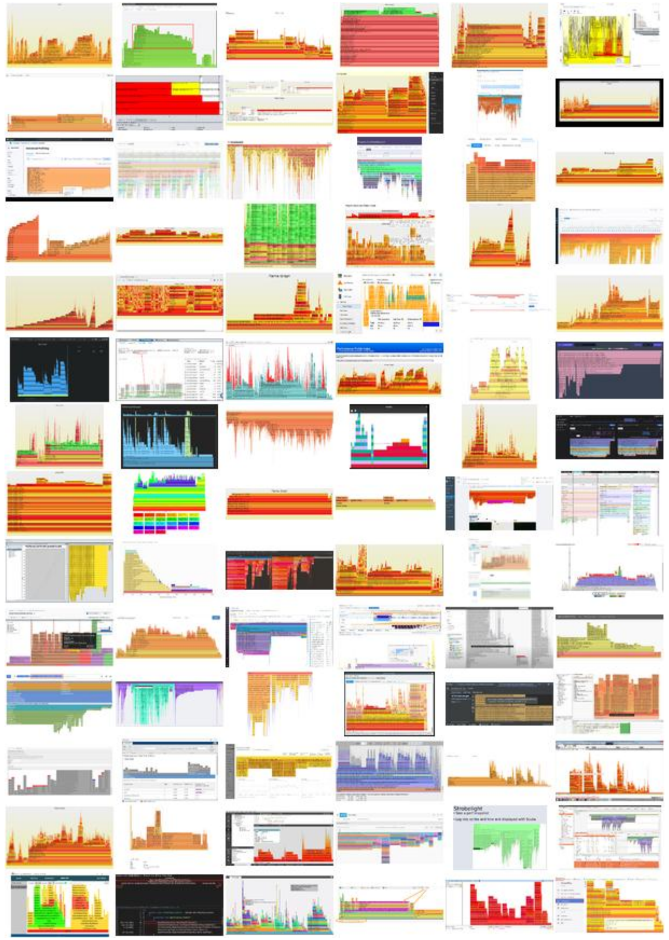
CPU Flame Graphs

(for background)

Source: https://www.brendangregg.com/Slides/YOW2022_flame_graphs/

CPU Flame Graph Adoption (by 2022)

- Implementations: **>80**
- Related open source projects: **>400**
- Commercial product adoptions: **>30**
- New startups: **4** (so far)
- Startups exits: **1** (so far)
- Industry investment: **>AUD$1B**
- End users: (a lot)

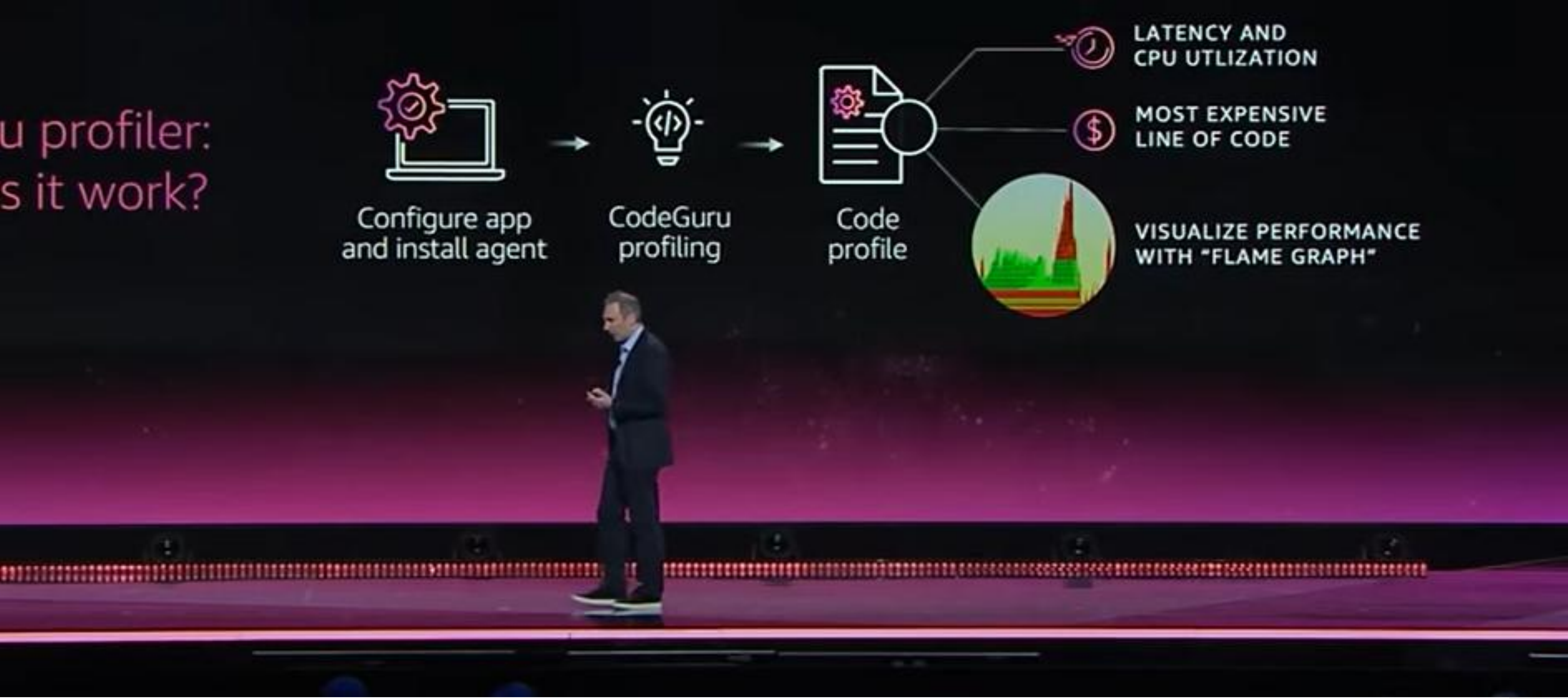


Source: https://www.brendangregg.com/Slides/YOW2022_flame_graphs/

Flame Graph Startups

- Superluminal (2018)
- Granulate (2019; sold to Intel)
- Pyroscope (2020; sold to Grafana Labs)
- Polar Signals (2021)
- INFERNOde (2022)

Incomplete list. Also not including >30 companies that adopted CPU flame graphs.


u profiler:
s it work?
Configure app and install agent
CodeGuru profiling
Code profile
LATENCY AND CPU UTLIZATION
MOST EXPENSIVE LINE OF CODE
VISUALIZE PERFORMANCE WITH "FLAME GRAPH"

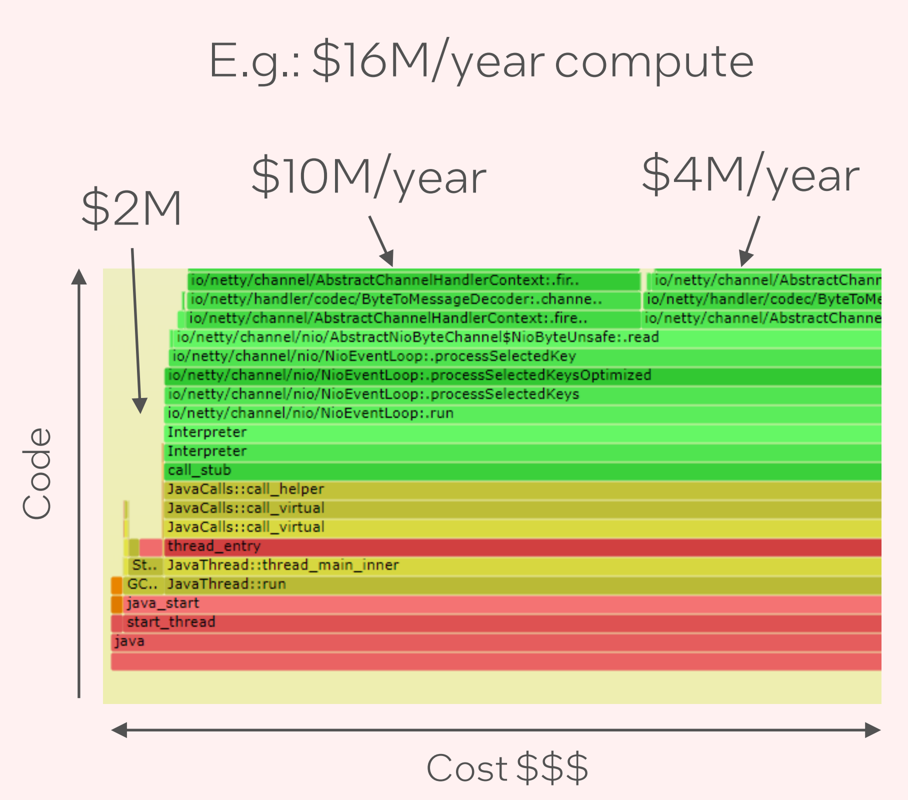
Andy Jassy, Amazon CEO, re:Invent 2019 keynote

https://youtu.be/7-31KgImGgU?si=IqLLXILsxgz3PEMH&t=7257

(just one example)

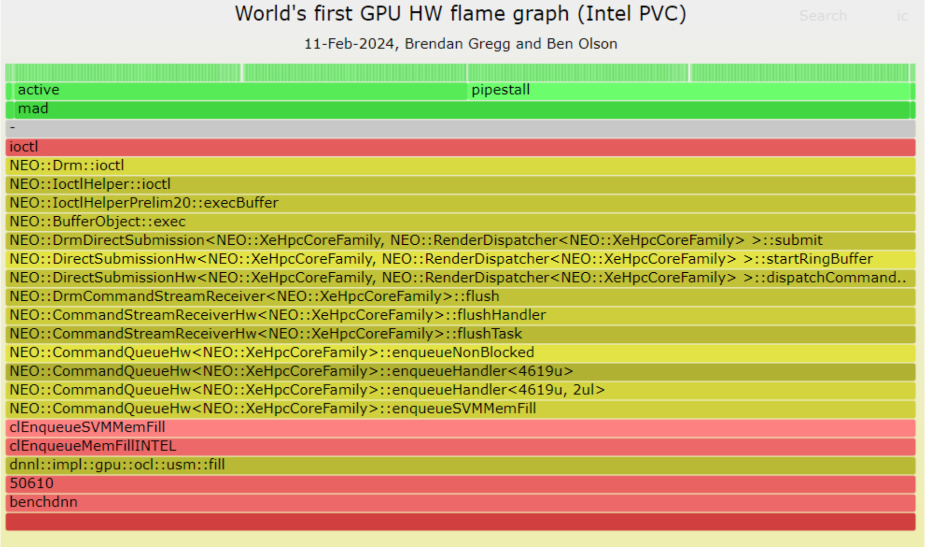
CPU Flame Graph Popularity

- Shows code proportional to cost
  - Can estimate cost reductions and performance gains through code and system tuning
  - Has found well over US$1B in savings
- Used by everyone:
  - Developers: Lets them "see" their own code executing and to develop faster code. It's in the **language of the developers** (code).
  - Sales support: Reveals the best places to tune (params/config/libs).
  - End-users: Provides "X marks the spot" for possibly millions in savings. If they don't understand how, they can share it with their perf team or a vendor (Intel). E.g., Cloud customers regularly share flame graphs with Intel, allowing Intel to find perf wins.


E.g.: $16M/year compute

GPU/AI Flame Graphs (2024)

- Work began in Nov 2023 (2 engineers: Brendan and Ben Olson). Focus was making it easy. It was not known if it was even possible.
- After 3 months developing eBPF instrumentation with an Intel-only PVC feature (eustalls), we created the first GPU flame graph:
- It is brittle and needs *far more* work than the more mature CPU profiling field to reach equivalent robustness. But this is also a major **turning point**: it is possible.


World's first GPU HW flame graph (Intel PVC)

11-Feb-2024, Brendan Gregg and Ben Olson

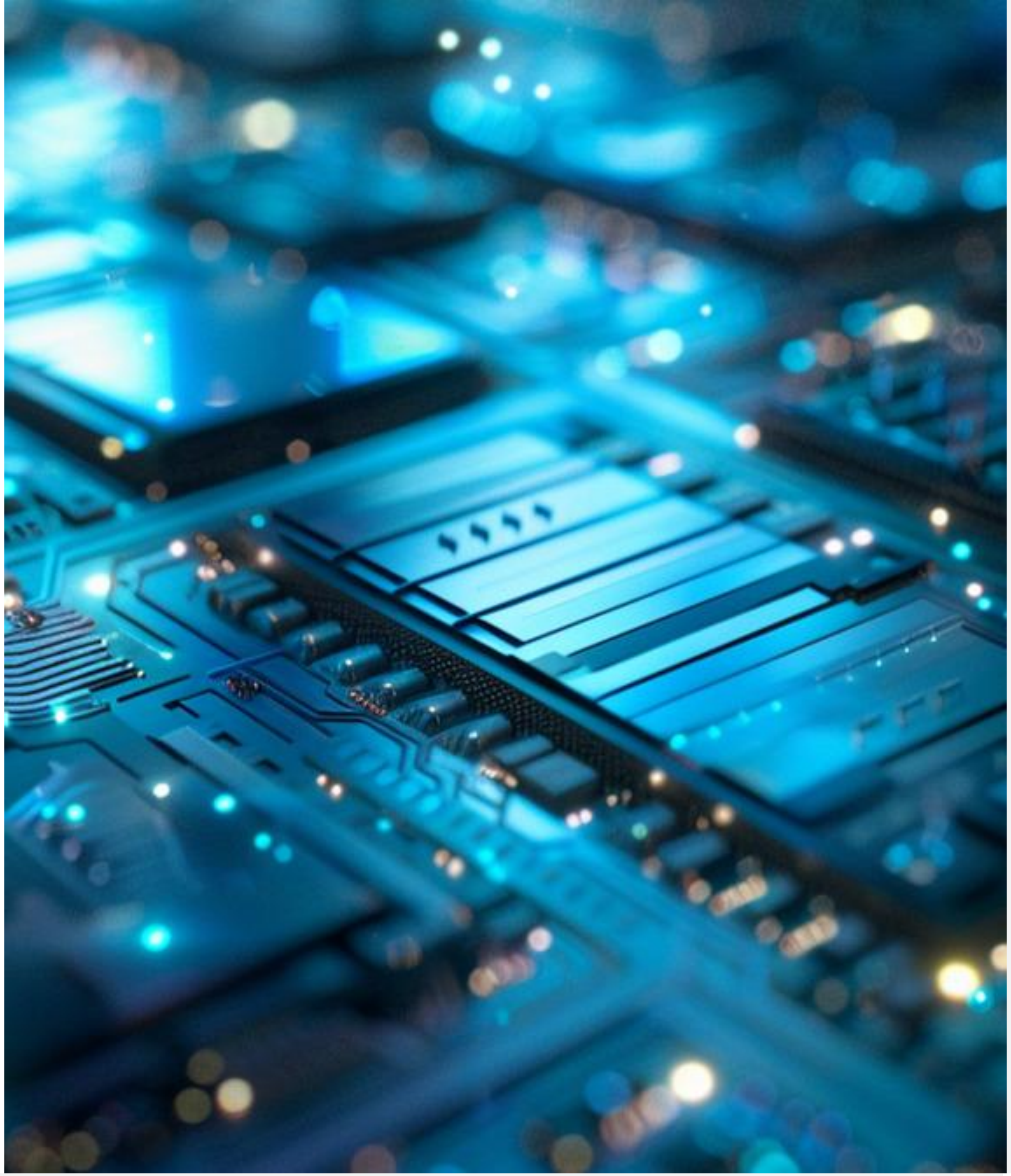
First GPU internals flame graph (2024)

Intel Innovation

World's first easy AI profiler

A. Code visualization
B. Low overhead
C. No setup required

…same as CPU profiling



AI/
GPU

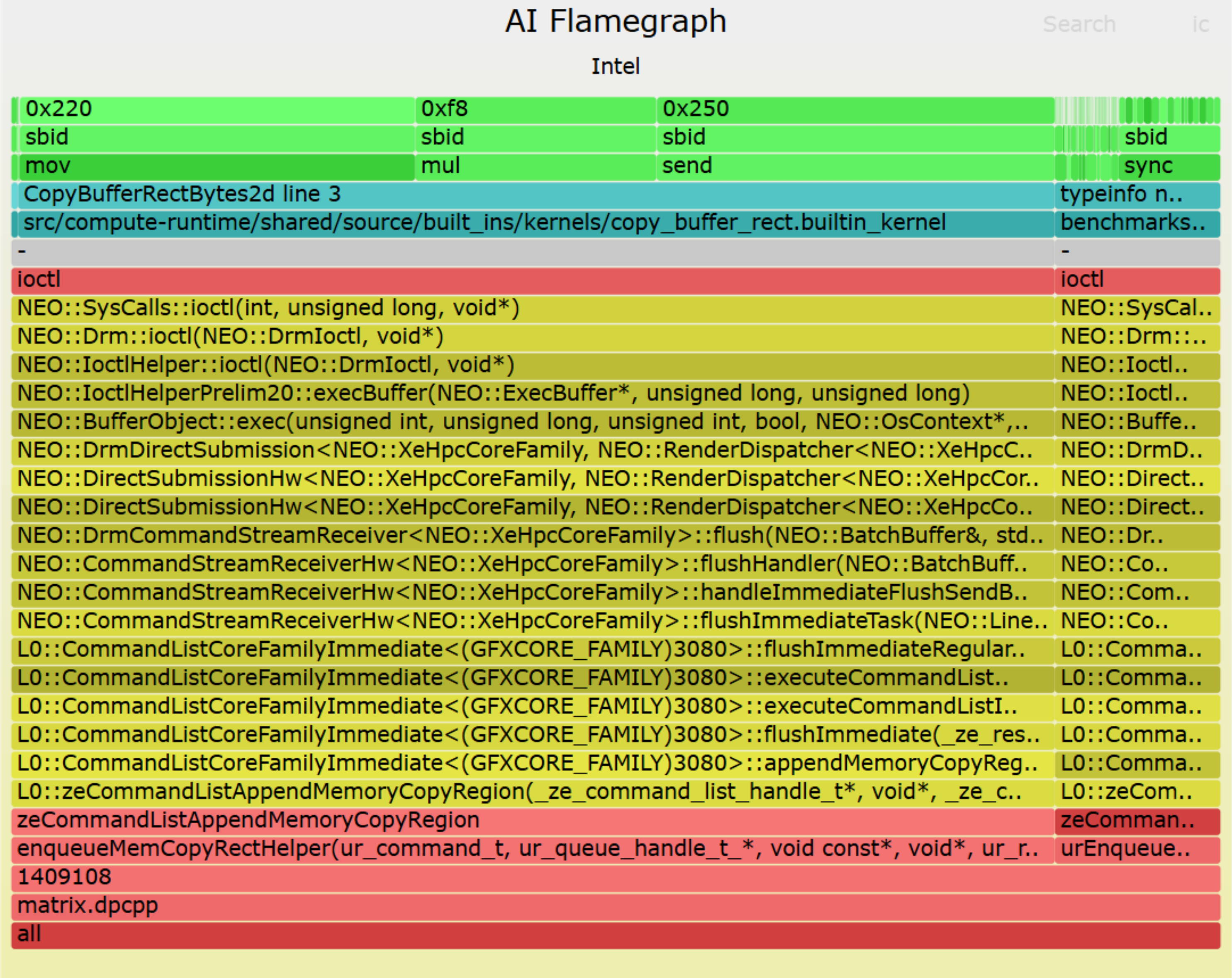
Instruction offset
Stall reason
Instruction mnemonic
Stack/source functions
Source file

CPU call stack

PID
Program

Legend:

🟩 GPU HW
🟦 GPU source
🟧 OS Kernel
🟨 CPU C++
🟥 CPU C
Python

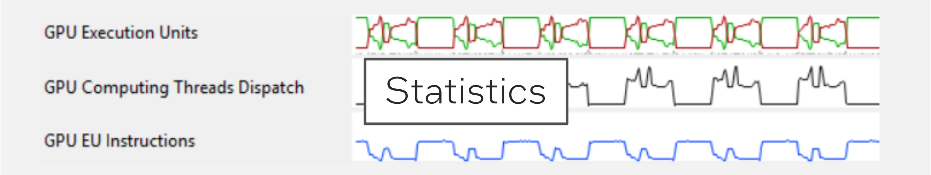


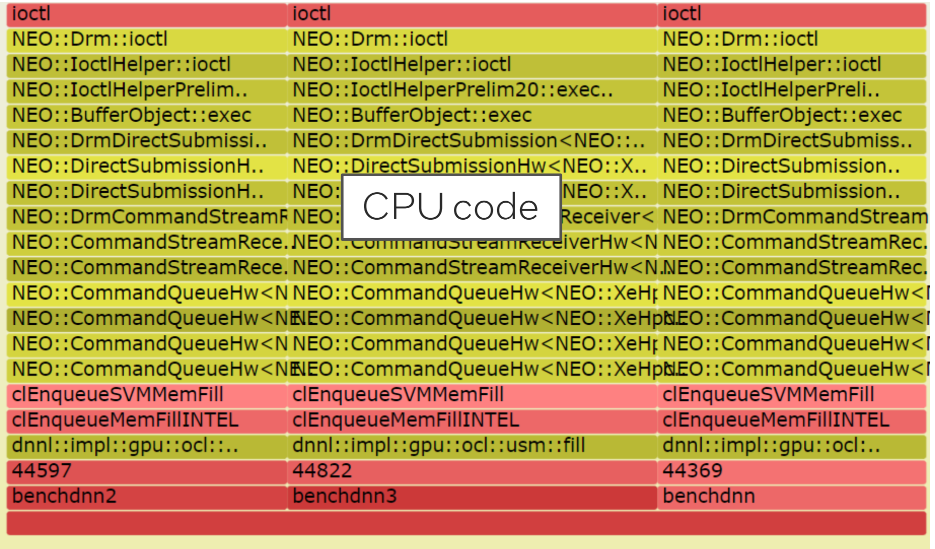
```
                                   AI Flamegraph                                   Search        ic

                                       Intel

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
```

Prior AI Profilers





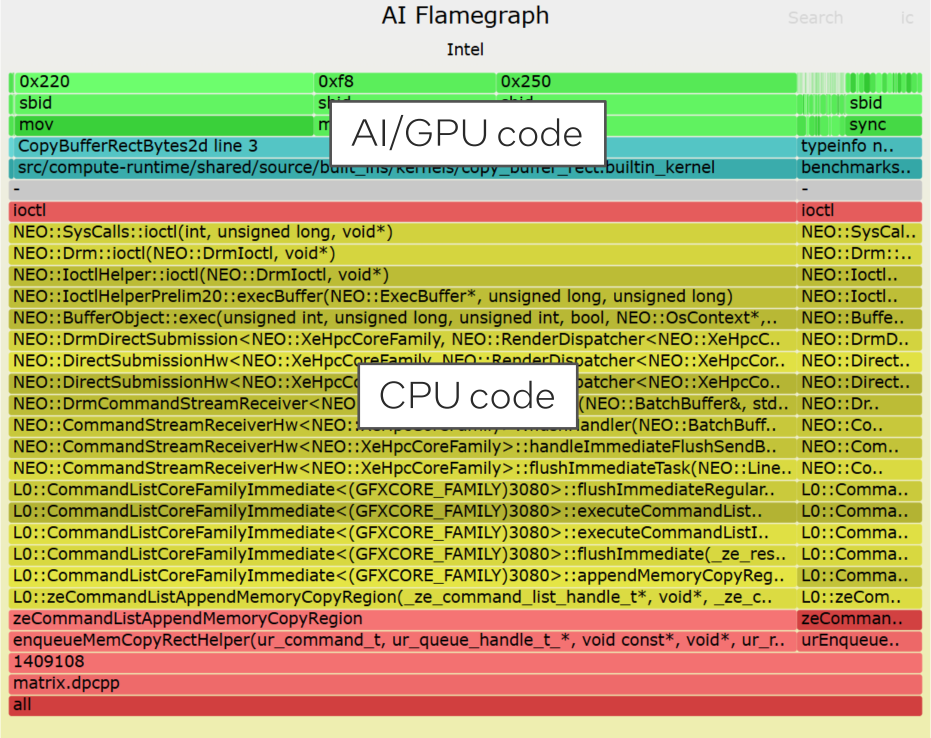
```
ioctl                       | ioctl                              | ioctl
NEO::Drm::ioctl             | NEO::Drm::ioctl                    | NEO::Drm::ioctl
NEO::IoctlHelper::ioctl     | NEO::IoctlHelper::ioctl            | NEO::IoctlHelper::ioctl
NEO::IoctlHelperPrelim..    | NEO::IoctlHelperPrelim20::exec..   | NEO::IoctlHelperPreli..
NEO::BufferObject::exec     | NEO::BufferObject::exec            | NEO::BufferObject::exec
NEO::DrmDirectSubmissi..    | NEO::DrmDirectSubmission<NEO::..   | NEO::DrmDirectSubmiss..
NEO::DirectSubmissionH..    | NEO::DirectSubmissionHw<NEO::X..   | NEO::DirectSubmission..
NEO::DirectSubmissionH..    | NEO:            NEO::X..           | NEO::DirectSubmission..
NEO::DrmCommandStreamR      | NEO:            Receiver<          | NEO::DrmCommandStream
NEO::CommandStreamRece..    | NEO::CommandStreamReceiverHw<N     | NEO::CommandStreamRec..
NEO::CommandStreamRece..    | NEO::CommandStreamReceiverHw<N..   | NEO::CommandStreamRec..
NEO::CommandQueueHw<N       | NEO::CommandQueueHw<NEO::XeHp      | NEO::CommandQueueHw<I
NEO::CommandQueueHw<N..     | NEO::CommandQueueHw<NEO::XeHp..    | NEO::CommandQueueHw<I
NEO::CommandQueueHw<N       | NEO::CommandQueueHw<NEO::XeHp      | NEO::CommandQueueHw<I
NEO::CommandQueueHw<N..     | NEO::CommandQueueHw<NEO::XeHp..    | NEO::CommandQueueHw<I
clEnqueueSVMMemFill         | clEnqueueSVMMemFill                | clEnqueueSVMMemFill
clEnqueueMemFillINTEL       | clEnqueueMemFillINTEL              | clEnqueueMemFillINTEL
dnnl::impl::gpu::ocl::..    | dnnl::impl::gpu::ocl::usm::fill    | dnnl::impl::gpu::ocl::..
44597                       | 44822                              | 44369
benchdnn2                   | benchdnn3                          | benchdnn
```

No code profiling on the AI accelerator or GPU

(Or partial code visibility via static instrumentation, or full accelerator code profiling but via binary instrumentation and prohibitively high overhead)

→

Intel AI Profiler



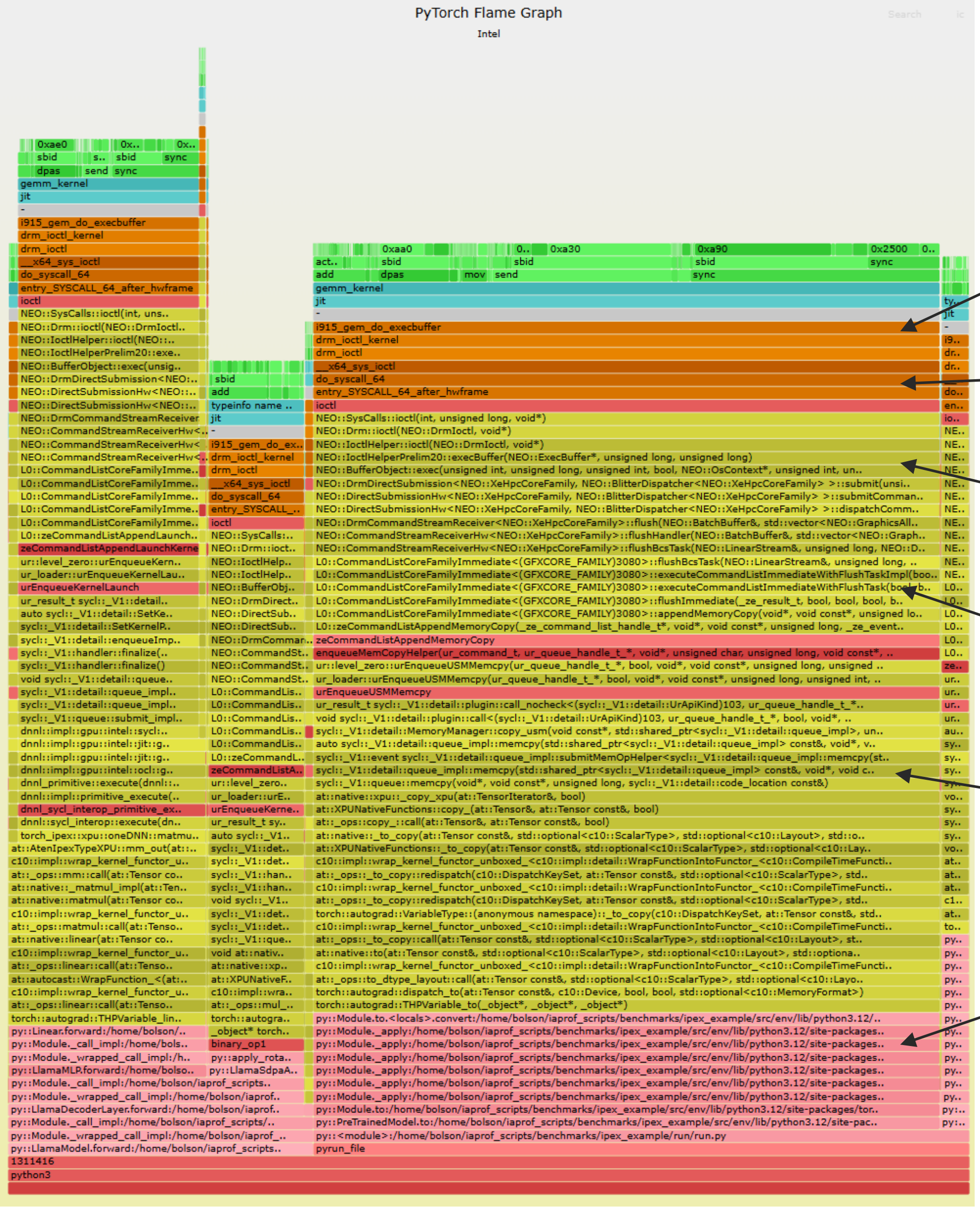
AI Flamegraph

Intel

```
0x220 | 0xf8 | 0x250
sbid | sb | sbid
mov | m | 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::XeHpcC          patcher<NEO::XeHpcCo.. | NEO::Direct..
NEO::DrmCommandStreamReceiver<NEO          (NEO::BatchBuffer&, std.. | NEO::Dr..
NEO::CommandStreamReceiverHw<NEO:          andler(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
```

All your code in one visual, proportional to cost

(Statistics still available; E.g., in VTune)



Visualising an Open AI Ecosystem

Intel i915 (GPU kernel driver)

Linux (OS Kernel)

Intel NEO (Compute Runtime)

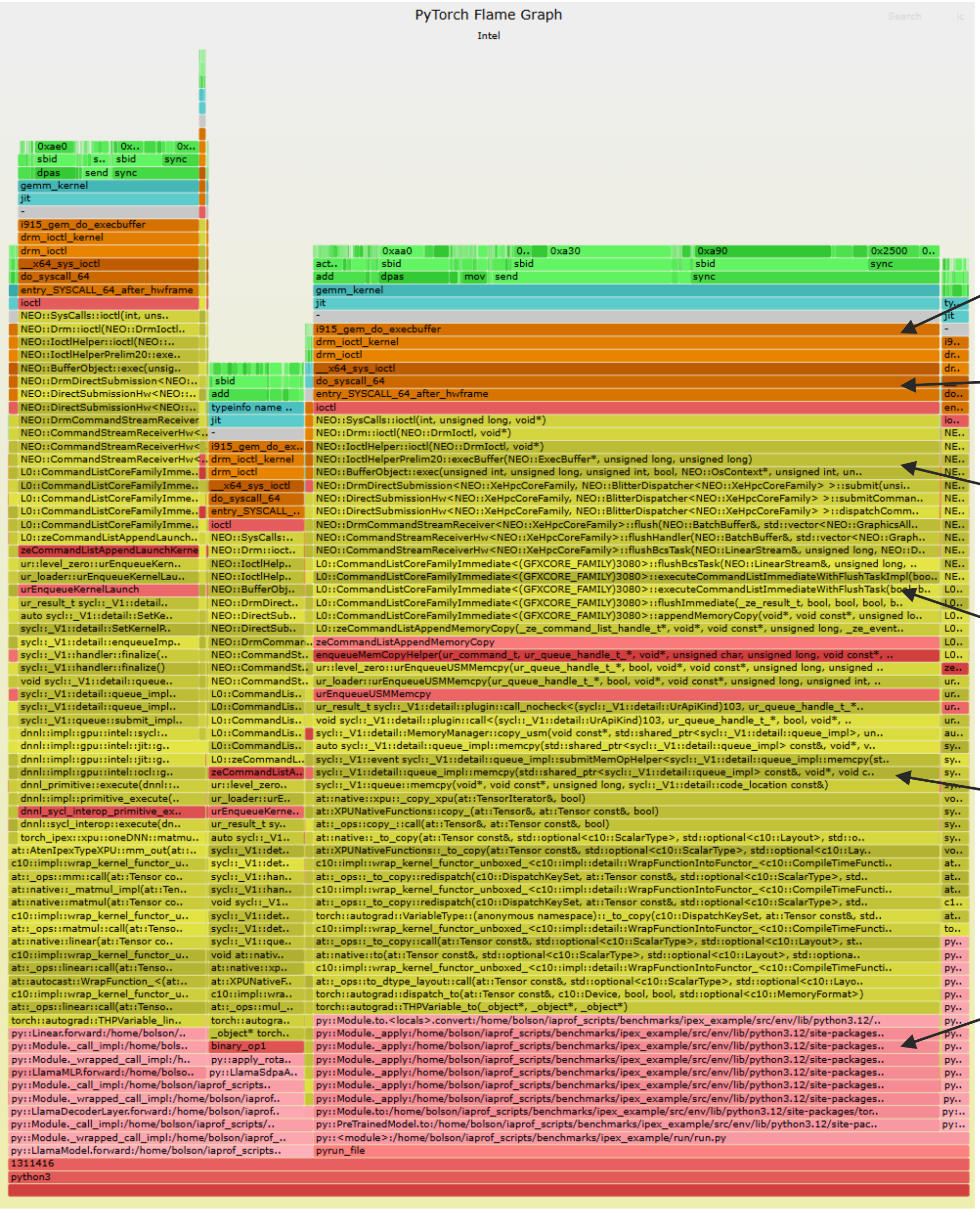
Intel Level Zero API (User-mode driver)

Intel SYCL (Runtime library)

Intel Extension for PyTorch (Framework)

Profiler is based on Intel EU Stalls, flamegraph.pl, and eBPF

Visualising an Open AI Ecosystem



Intel i915 (GPU kernel driver)

https://github.com/torvalds/linux/tree/master/drivers/gpu/drm/i915

Linux (OS Kernel)

https://github.com/torvalds/linux

Intel NEO (Compute Runtime)

https://github.com/intel/compute-runtime

Intel Level Zero API (User-mode driver)

https://github.com/oneapi-src/level-zero

Intel SYCL (Runtime library)

https://github.com/intel/llvm/tree/sycl/sycl

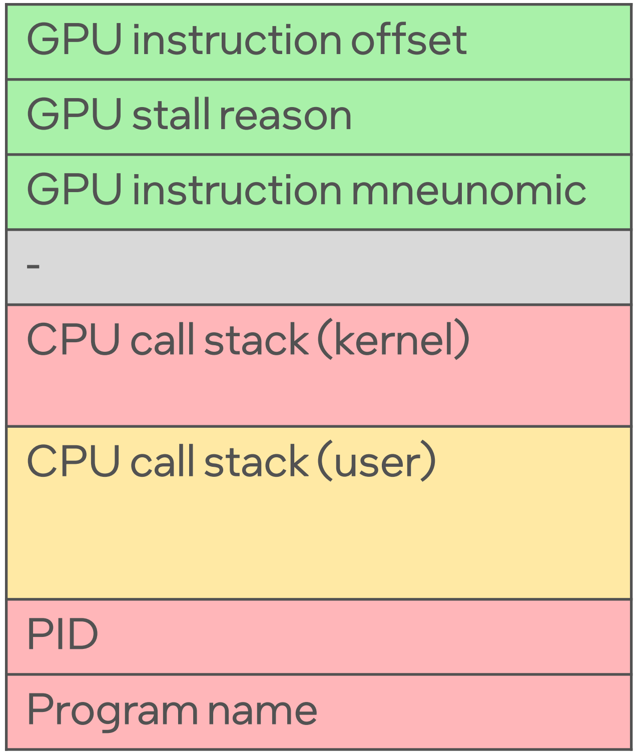
Intel Extension for PyTorch (Framework)

https://github.com/intel/intel-extension-for-pytorch

Profiler is based on Intel EU Stalls, flamegraph.pl, and eBPF

https://github.com/brendangregg/FlameGraph
https://github.com/torvalds/linux/tree/master/kernel/bpf

Current PoC (Feb '24)



GPU instruction offset

GPU stall reason

GPU instruction mneunomic

-

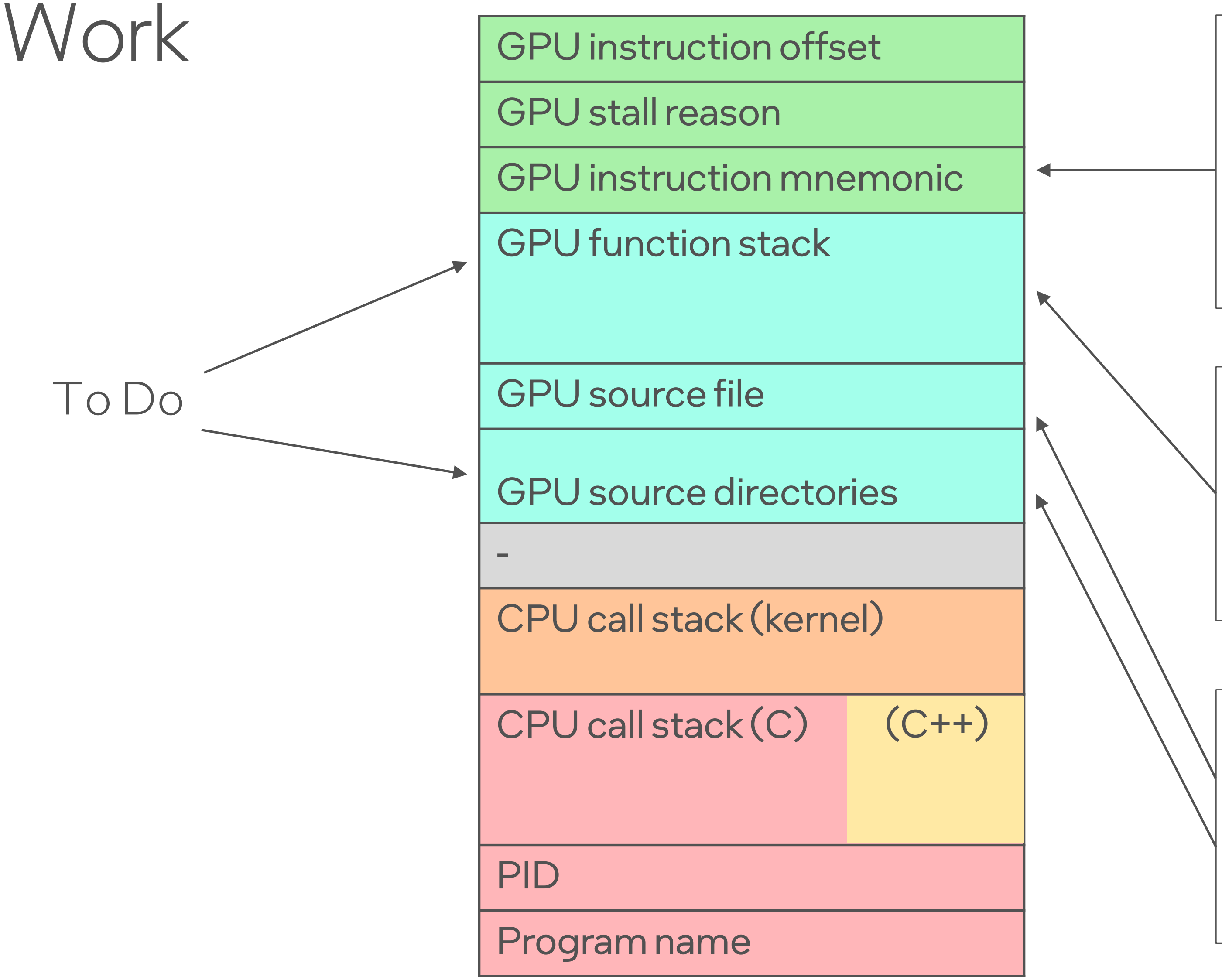
CPU call stack (kernel)

CPU call stack (user)

PID

Program name

Future Work



To Do

GPU instruction offset

GPU stall reason

GPU instruction mnemonic

GPU function stack

GPU source file

GPU source directories

-

CPU call stack (kernel)

CPU call stack (C) (C++)

PID

Program name

The eustall feature has a behavior where programs may share the same GPU virtual addresses but doesn’t provide a context ID to differentiate. Workaround in SW & eventually fix in GPU HW/firmware.

GPU programs typically lose their stacks after compiler inlining. It should be possible to recover most stack frames given compiler support for inlined symbol tables.

GPU compilers should produce symbol tables by default using a standard API, and costing near-zero overhead. This needs to work across different GPU runtime/compilers.

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).

Requirements Competition

(Still researching, may be incomplete and erroneous)

	Linux equiv.	Near-zero overhead	Easy to deploy	Easy to understand	On-GPU profiling	Complete visibility
HW counters	`vmstat`	Yes	Yes	No	No	Partial: CPU & GPU stats
GTPin (Intel)	`gcc -pg`	No	No	Yes	Yes	No: GPU only
unitrace (Intel)	`perf`	Yes	No	Partial	Yes	No: GPU only
[redacted]	`VTune`	Probably	No	Yes	No (*)	No
eBPF work in progress (at [redacted])**	`funcstack` `funclat`	Yes	Yes	CPU yes, GPU never	No	No: CPU only
[redacted]	`bpftrace`	No	No	No	No	No
AI profiler (this)	flamegraph	Yes	Yes	CPU yes, GPU not yet	Yes	Yes

<redacted> GPU profiler is an impressive product, however, based on dynamic instrumentation

* This was "No" when this slide was created, but was later added months after our working PoC

** Based on limited public information shared at conferences and github, and unrelated to our work at Intel.

Technologies Competition

(Still researching, may be incomplete and erroneous)

	Linux equiv.	CPU stats	CPU funcs	GPU stats	GPU funcs	Overhead	Easy to deploy	Output
HW counters	`vmstat`	Yes	No	Yes	No	None	Yes	Statistics
GTPin (Intel)	`gcc -pg`	No	No	? (probably)	Yes	High	No (wrapper)	Instrumentation listing
unitrace (Intel)	`perf`	No	No	Yes	Yes	None	No	Sample list
eBPF work in progress ([redacted])**	`funcstack` `funclat`	Yes	Yes	No	No	Lowest	Yes	Flame graph
AI profiler (this)	flamegraph	No	Yes	Some (eustall reason)	Not yet	Low	Yes	Flame graph

Our focus has been the unsolved problem of a low-overhead, easy to use, GPU stack profiler. CPU/GPU statistics are already available and can be added to our product later.

** Based on limited public information shared at conferences and github, and unrelated to our work at Intel.

GPU/AI Flame Graph Adoption (thought exercise)

	CPU	GPU/AI
Implementations	>100	
Related open source projects	>400	
Commercial product adoptions	>30	
New startups	6	
Startup exits	2	
Industry investment	>USD$700M	
End users	Widely used	

Challenges

HW access	We had no PVC access for most of 2023
Staffing	Only 2 (Brendan & Ben) and currently 1 (Ben out on leave). Needs advanced skills (AI, eBPF, kernel eng., perf) and ability to compete and win.
HW	eustalls: no unique ID across contexts. We employ workarounds. Flame Graphs will sometimes show unresolved symbols and warnings.
I915 driver	Turned out to be difficult to trace. Advanced eBPF required.
Xe driver	i915 driver will eventually be replaced by new Intel Xe driver. Needs support.
GPU symbols	These aren't usually available on the host. We need an easy way to get them that is ideally "zero instrumentation": E.g., always-on symbol generation from the IGC stack.
GPU workload support	SYCL, OpenCL, Level 0 API (SPIR-V IR and GEN binaries). Getting AI profiler to work on all workload permutations will take multiple engineers.

Current Status (May 24)

- eustall-based IP sampling: Working
- eBPF-based i915 instrumentation: Working
- Test workloads: 5/6 profile, 92-99% “matched”
- Default GPU Symbols: Work just begun
  - Once PoC working, will reach out more broadly for help
- Other HW: Work not started

25-May-2024

```
Percent_matched
bench       input     iaprof
l0_matmul   default   99.402
benchdnn    default   99.991
llama       default   96.039
oidn        default   97.827
openvkl     default   92.442
```

Current Release

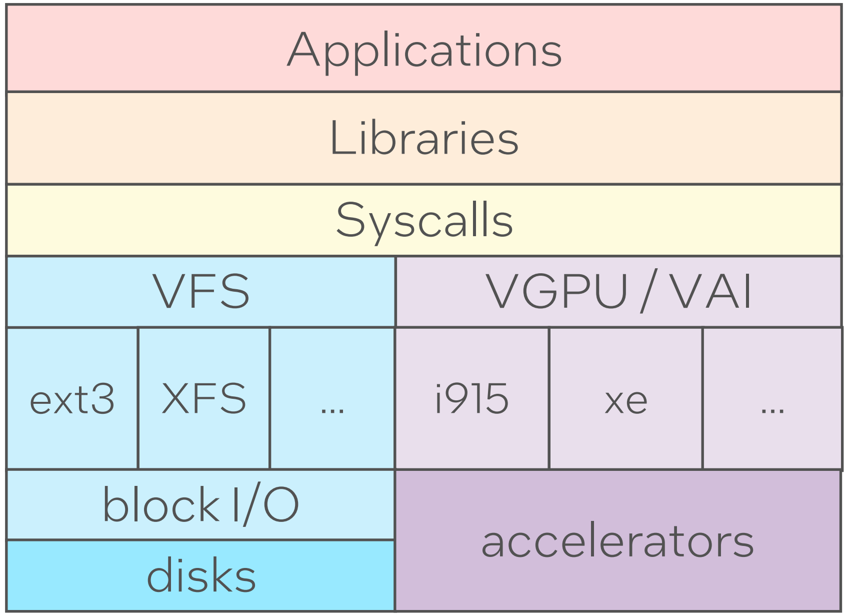
- Backend is called `iaprof` (Intel Accelerator Profiler)
- Early release is available
  - Note that we're currently renaming it from ai_flamegraph to iaprof.

```
# sudo iaprof sleep 30 > profile01.txt
# git clone https://github.com/brendangregg/FlameGraph
# cat profile01.txt | ./FlameGraph/flamegraph.pl > profile01.svg
```

Possible help by team

- IGC / DPCPP
  - Always-on symbol tables
- GPU HW
  - Add context ID to eustalls
  - Add a timer-based profiler
- i915/xe
  - Kernel tracepoints.
- NEO
  - I'm sure there'll be something

- Everybody
  - **Support a new kernel filesystem (/proc/sys/gpu) and kernel virtual interface (VGPU) for exposing metrics, program memory, and symbols. (s/gpu/ai/).**



Applications

Libraries

Syscalls

VFS | VGPU / VAI

ext3 | XFS | ... | i915 | xe | ...

block I/O | accelerators

disks

intel®