14th International Workshop on OpenCL and SYCL

IWOCL 2026

Optimizing AI Workloads on Intel GPUs With OpenCL

Michał Mrozek, Intel

May 6-8, 2026 | Heilbronn, Germany | iwocl.org

Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more on the Performance Index site.

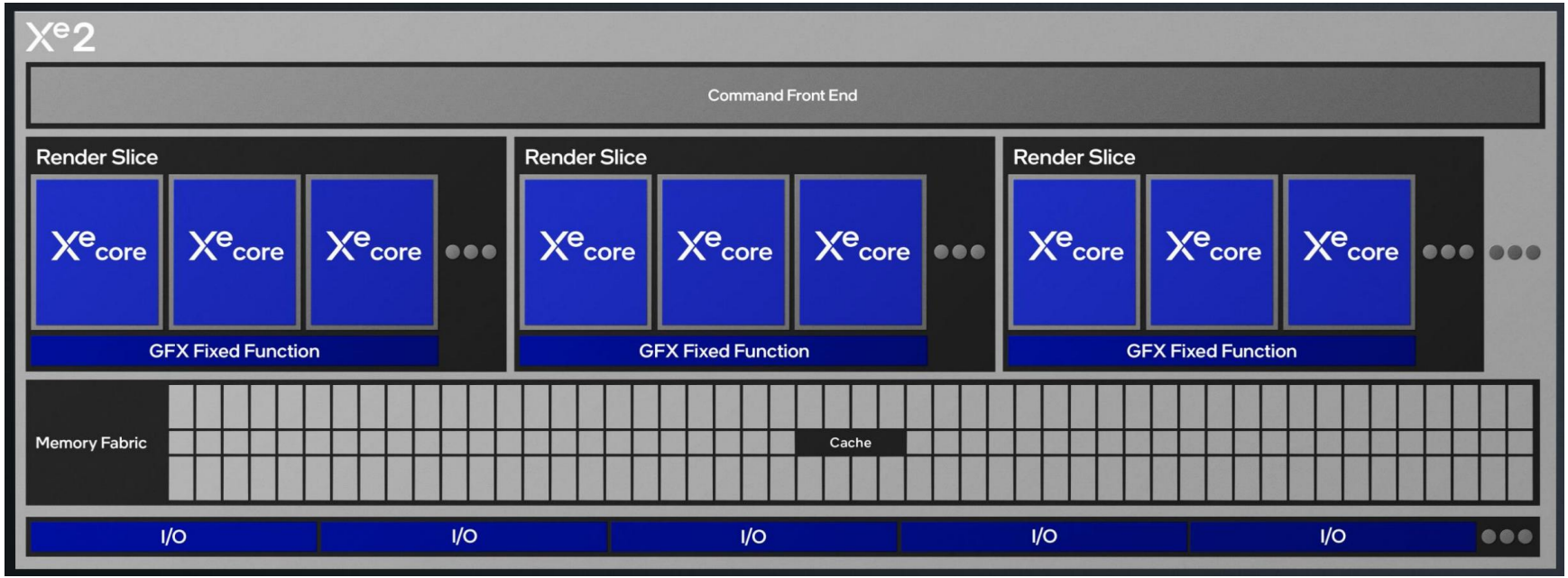
Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

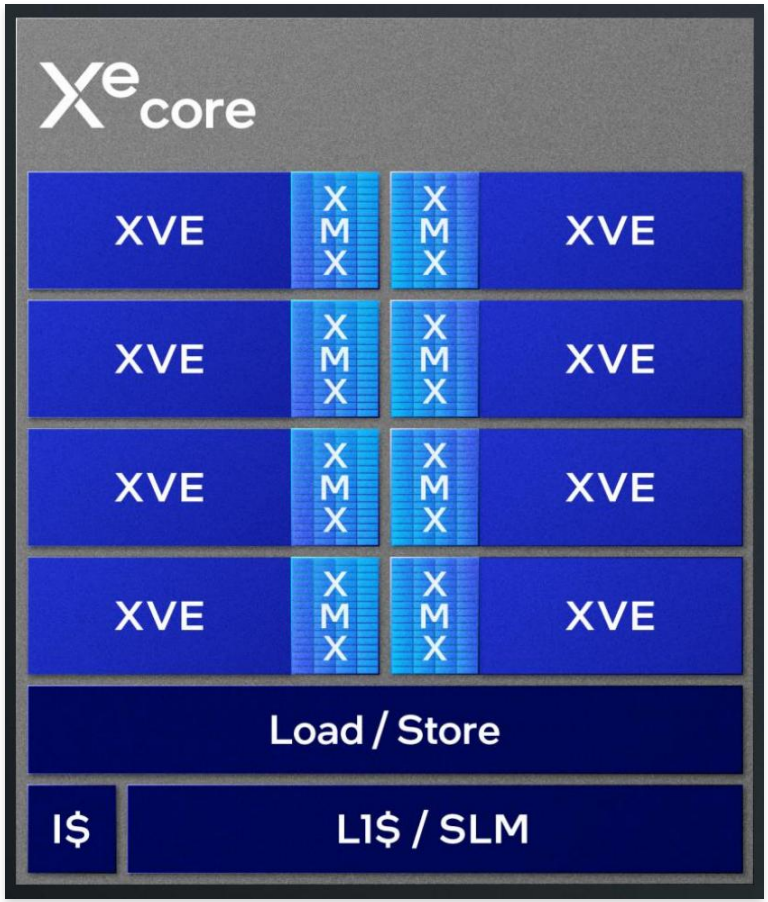
Intel technologies may require enabled hardware, software or service activation.

Quick look at Xe2+ devices construction

Xe2 architecture bird's eye view



XeCore basic building block



General purpose vector units:

- 8 x 512-bit Vector Engines
- 192 KB Shared L1$ / SLM
- 64b atomics support
- FP64 support
- Transcendentals SIN, COS, LOG, EXP

Xe Matrix Extension Engines

Resources shaped to execute AI:

- 8 x 2048 bit XMX Engines
  - INT2, INT4, INT8, FP16, BF16
- FP16 2048 OPS/clock
- INT8 4096 OPS/clock
- 3-way co-issue
  - FP + INT + XMX

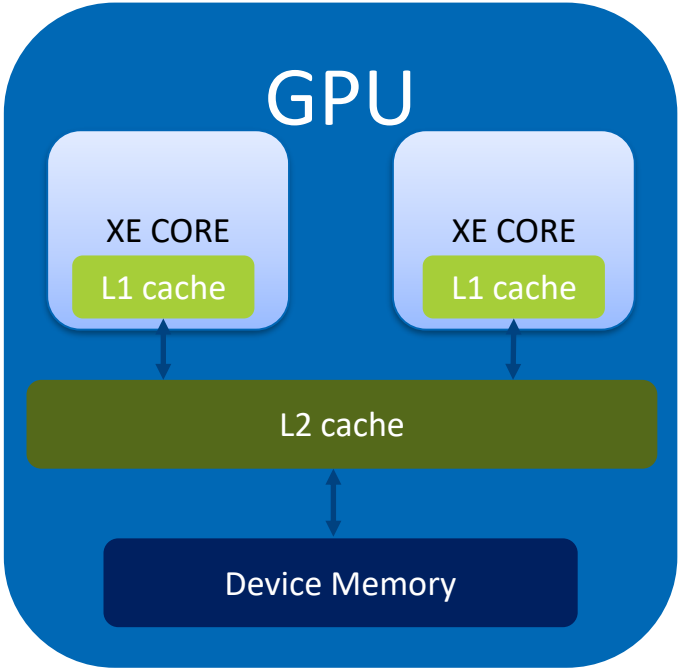
Peak metrics for Xe2 single XeCore

	Number of XVE	SIMD width	MAC/lane	Depth	Ops/MAC	Ops/clock
FP32	8	16	1	1	2	256
FP16	8	16	2	1	2	512
DP4a INT8	8	16	4	1	2	1024
XMX FP16 / BF16	8	16	2	4	2	2048
XMX INT8	8	16	4	4	2	4096
XMX INT4 / INT2	8	16	8	4	2	8192

Handling memory
Unified Shared Memory

Device Allocations: Performance

- Cached in L1/L2
- Bandwidth Compression
- No Host access
- Best Performance possible




```
void* clDeviceMemAllocINTEL(
    cl_context context,
    cl_device_id device,
    const cl_mem_properties_intel* properties,
    size_t size,
    cl_uint alignment,
    cl_int* errcode_ret);
```

Host Allocations: Zero Copy Sharing (no Migration)

Integrated parts:

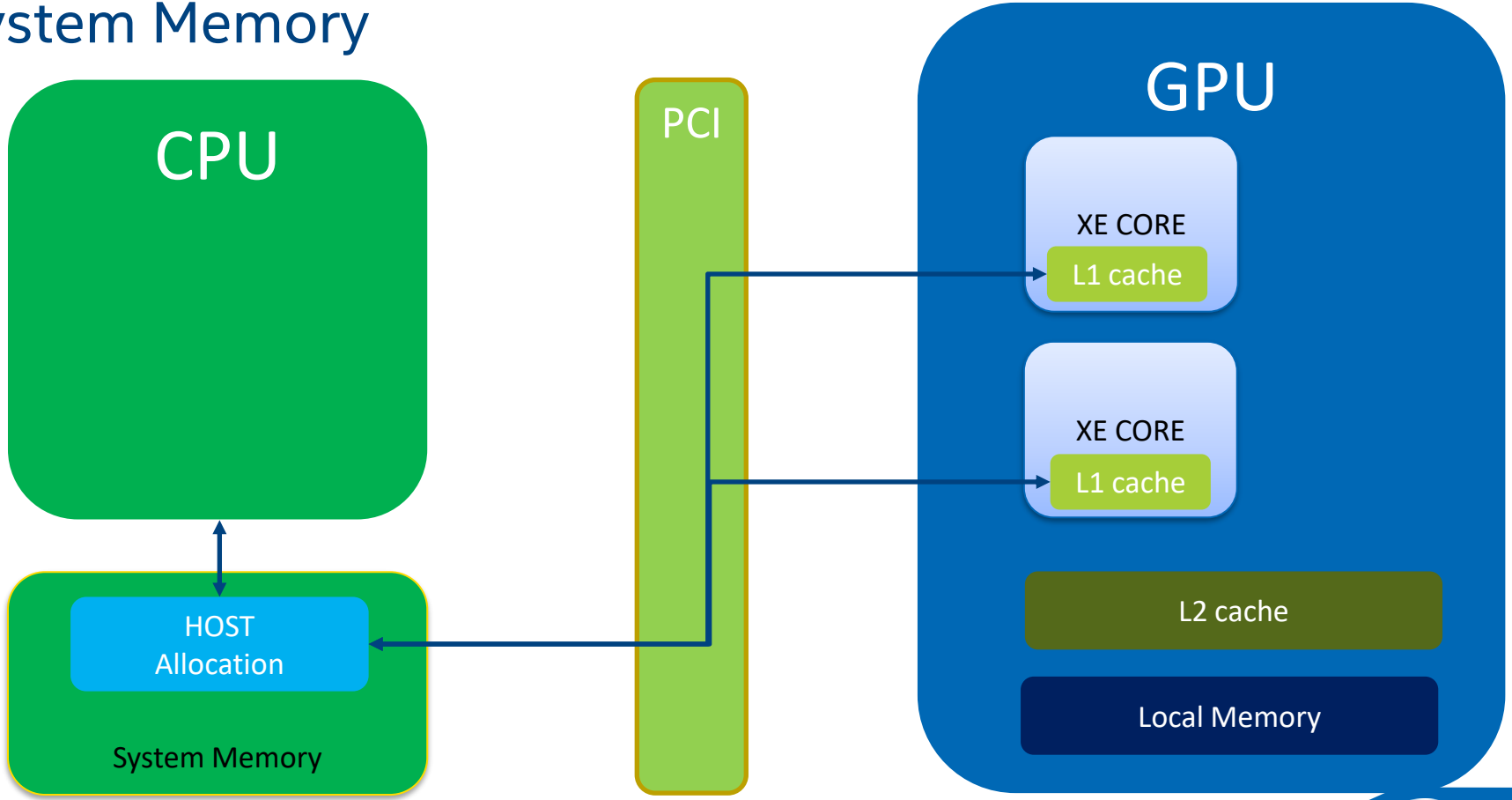
- L1/L2 cached
- CPU->GPU coherent
- GPU -> CPU requires L1+ L2 flush

Discrete parts:

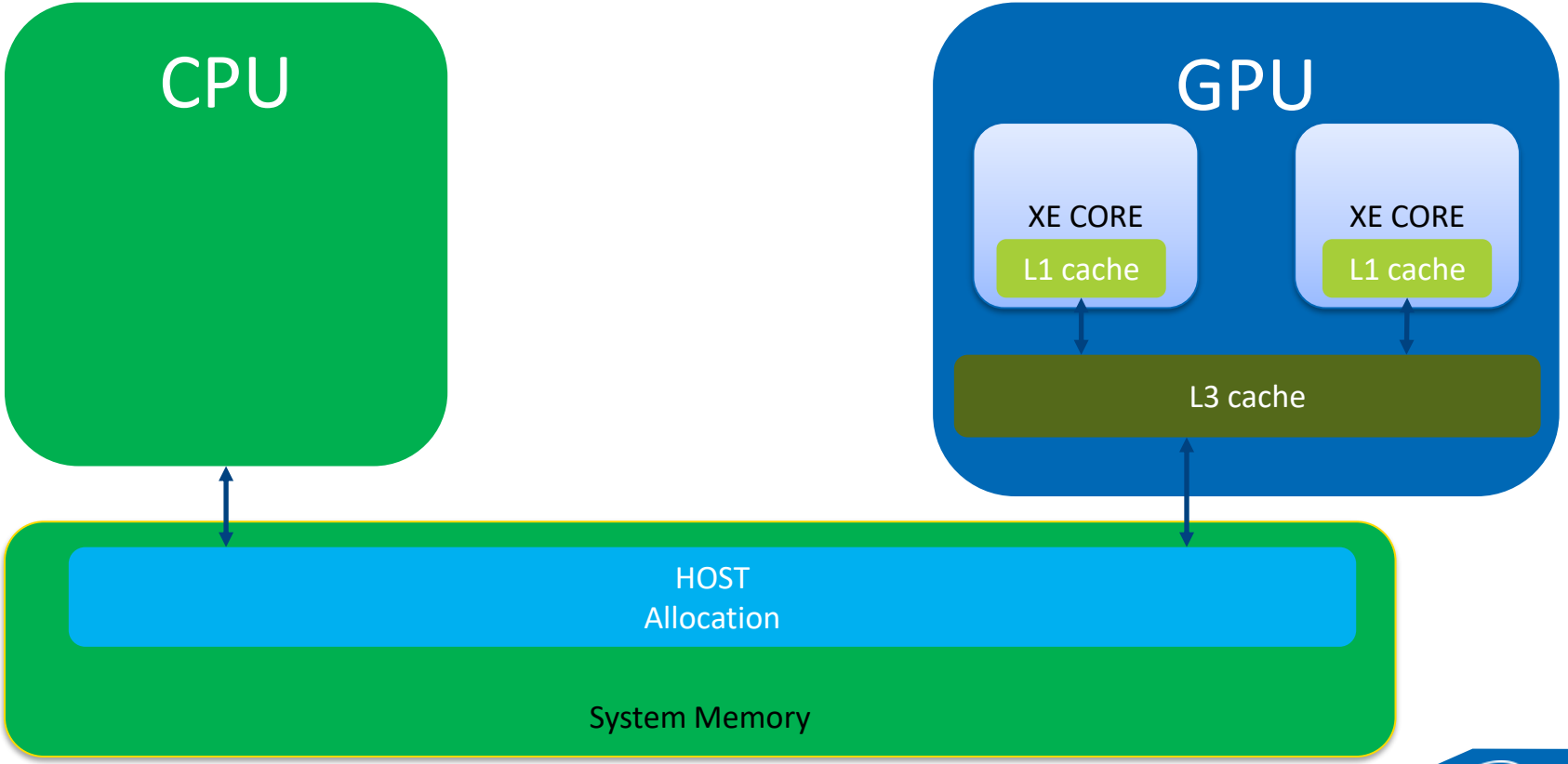
- L1 cached
- CPU->GPU coherent
- GPU -> CPU requires L1 flush

```
void* clHostMemAllocINTEL(
    cl_context context,
    const cl_mem_properties_intel* properties,
    size_t size,
    cl_uint alignment,
    cl_int* errcode_ret);
```

Host Allocation on discrete: Direct GPU access to System Memory

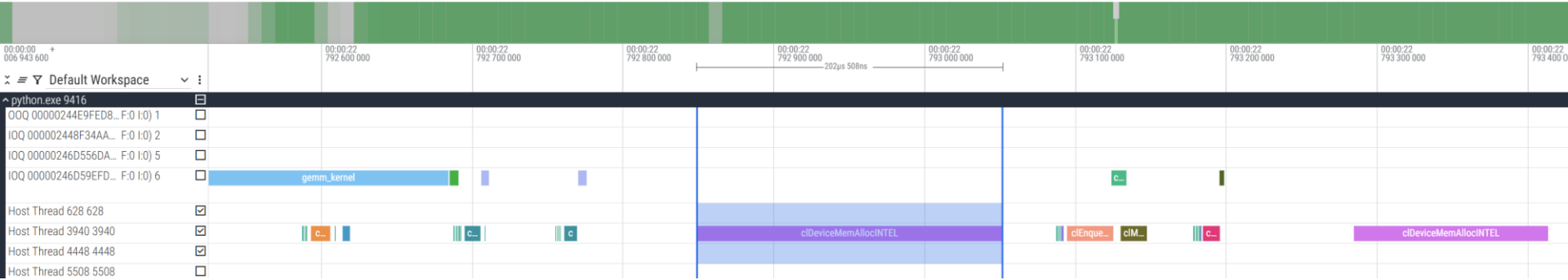


Host Allocation on integrated parts



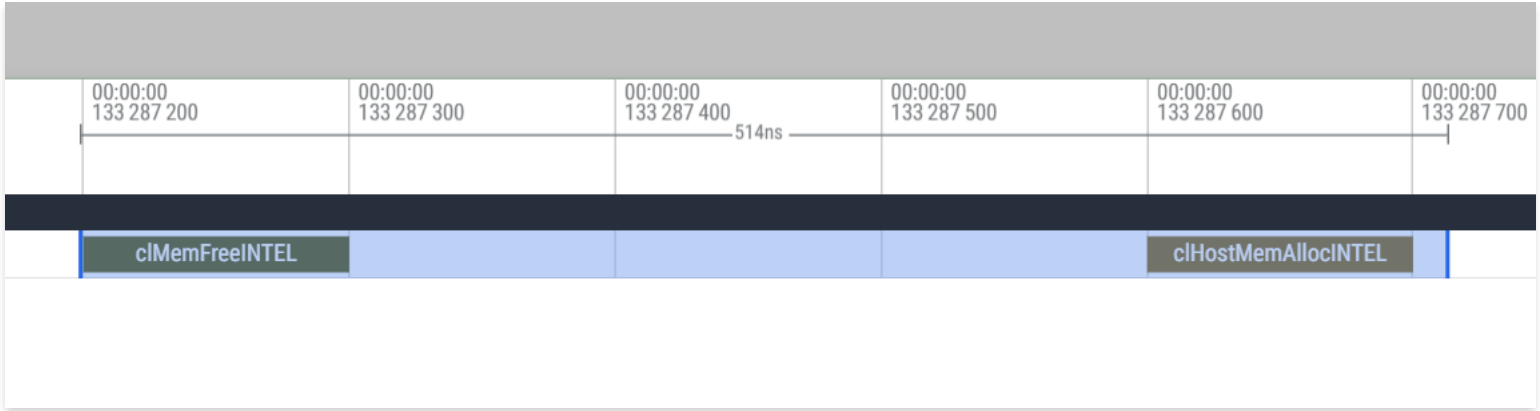
Resource allocation

LLMs started to allocate in the hot path



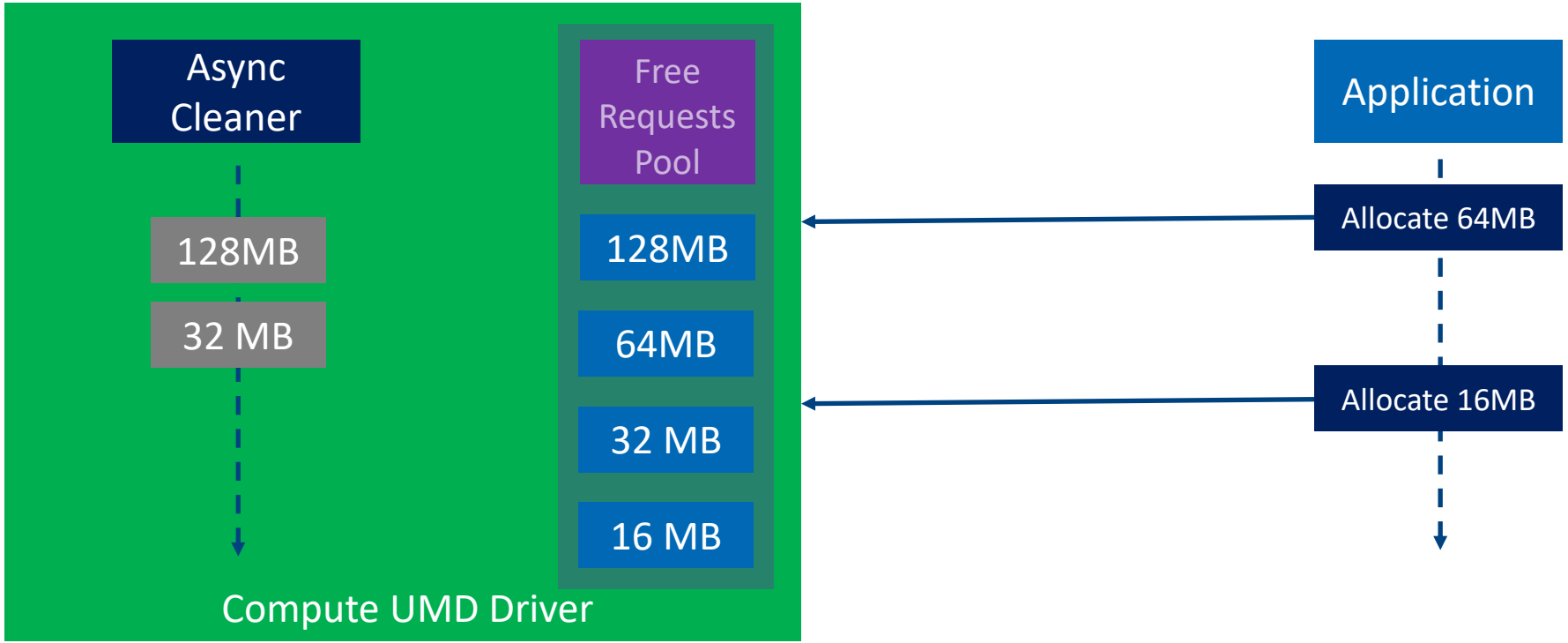
- Massive memory pressure due to larger model size
- Cannot pre-allocate upfront
- Memory allocations very time consuming (page table creation, pinning, TLB invalidations,etc.)

Resource recycling and pooling to the rescue



- Expensive freeing cost eliminated
- Expensive allocation cost eliminated
- KMD/OS calls eliminated from allocation path
- Small allocation from pre-allocated memory pools with 2MB alignment
- But only for allocations <=256MB

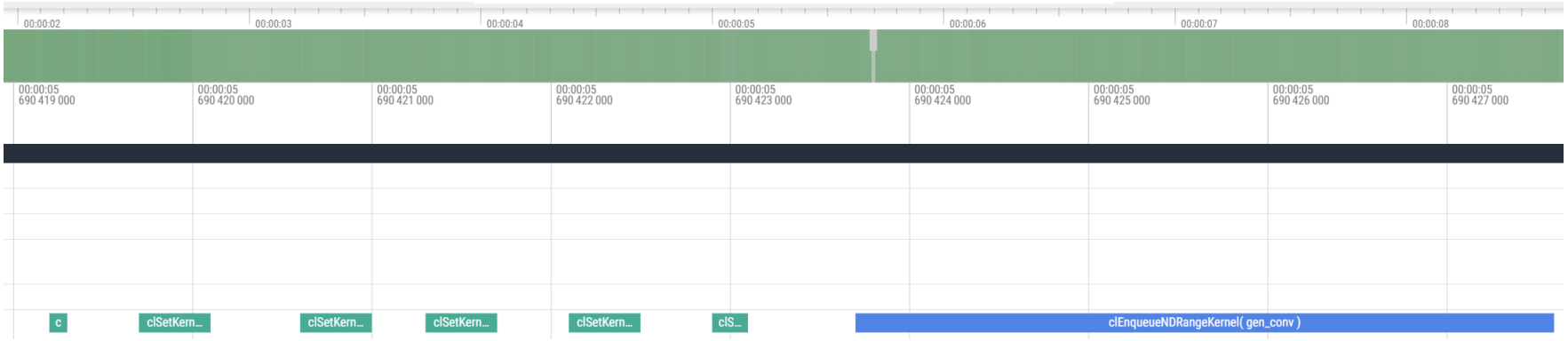
Adaptive cleanup



No resources wasted

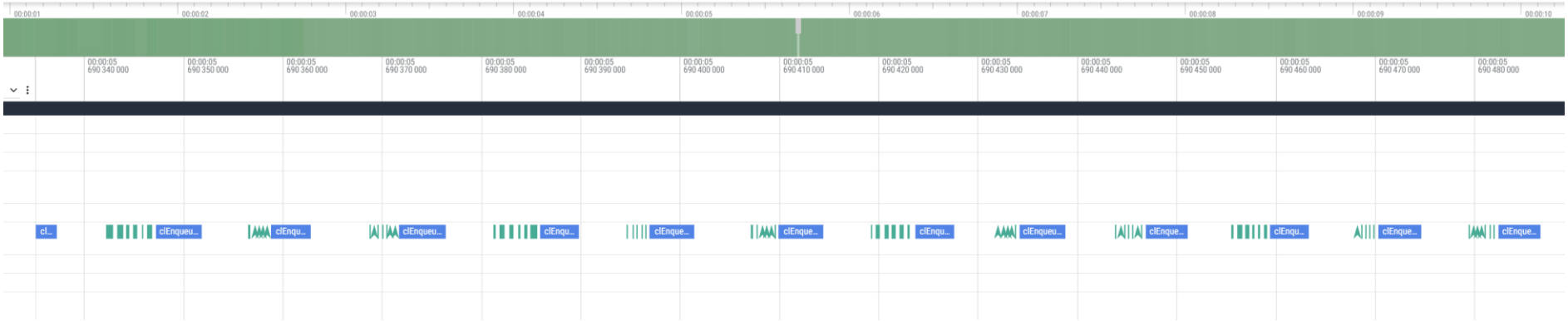
Deep dive into scheduling of AI workloads

Queue model – schedule a kernel



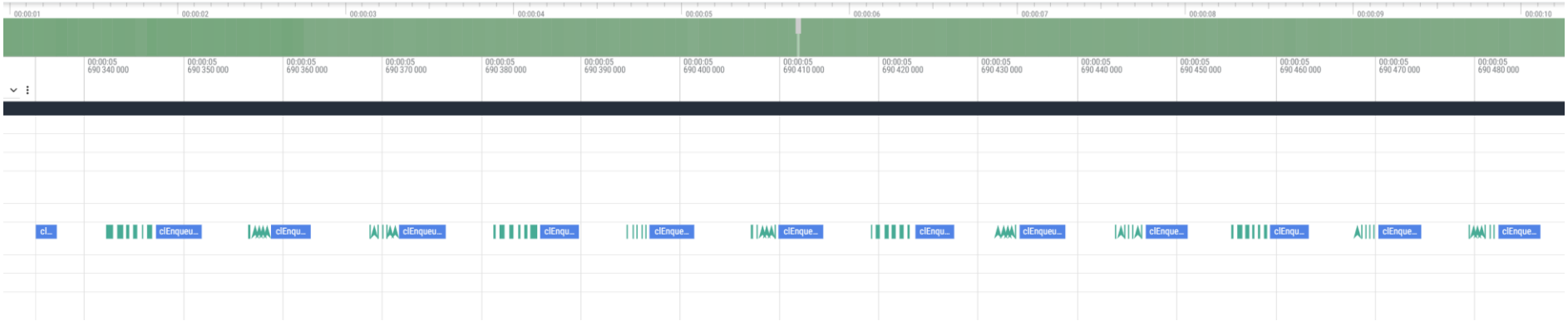
- For each kernel set all arguments
- And enqueue it to the queue

Queue model – schedule multiple kernels



- Repeat as long as there are kernels required

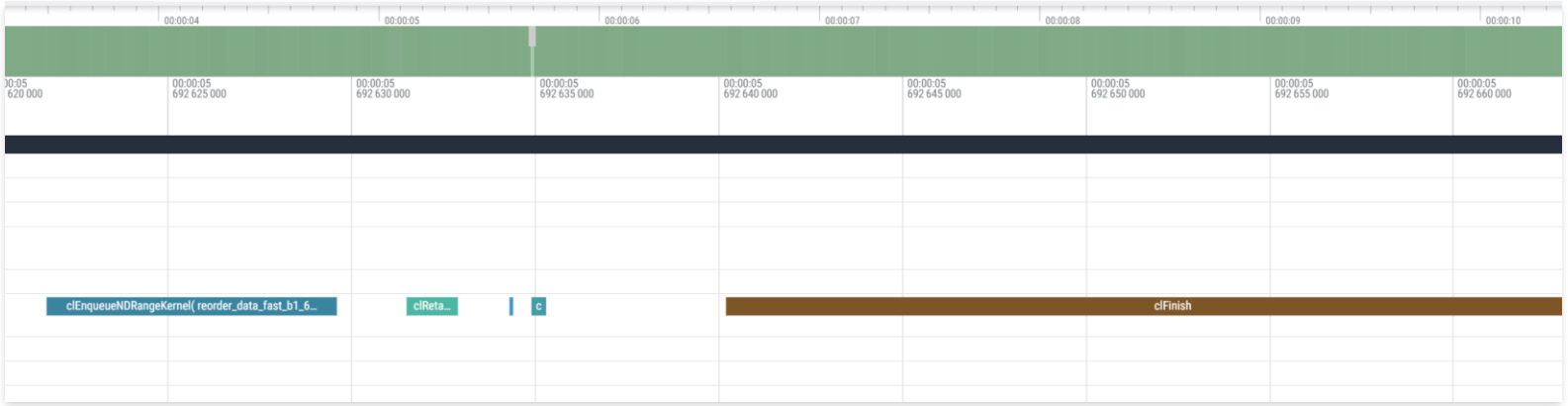
Queue model – clSetKernelArg is persistent



- No need to call clSetKernelArg with the same value on the same kernel
- But many workloads do this, so driver optimizes this path and no-ops calls if same parameter passed

- *arg_value* is a pointer to data that should be used as the argument value for argument specified by *arg_index*. The argument data pointed to by *arg_value* is copied and the *arg_value* pointer can therefore be reused by the application after **clSetKernelArg** returns. The argument value specified is the value used by all API calls that enqueue *kernel* (**clEnqueueNDRangeKernel** and **clEnqueueTask**) until the argument value is changed by a call to **clSetKernelArg** for *kernel*.

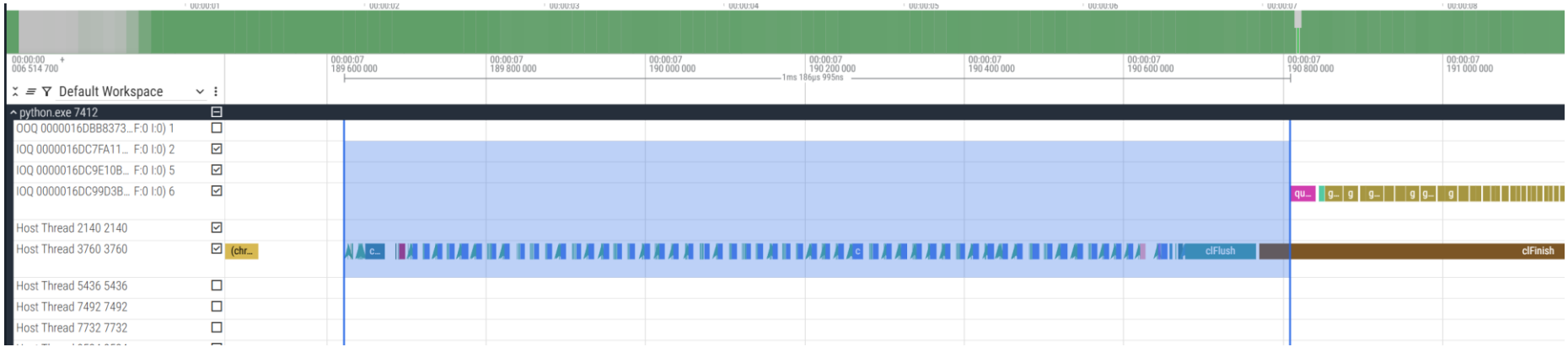
Queue model – wait for completion



- Finally wait on the host for everything that was scheduled before
- With Host USM, no need to do any transfers to host memory, it is already available once GPU completes execution

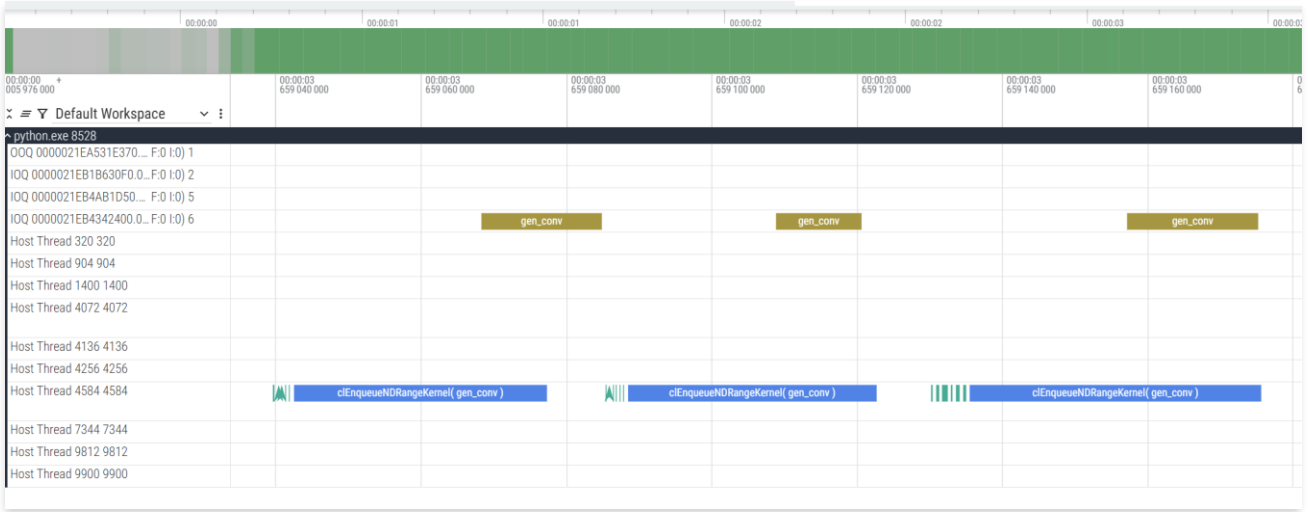
Queue Model:
When to Submit to GPU?

Queue model – submit on clFlush



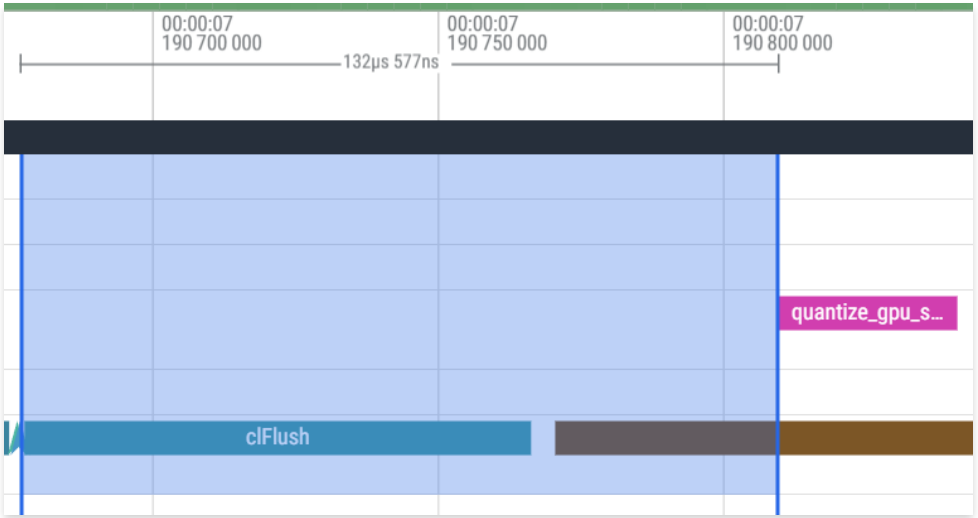
- GPU idle during enqueue operations
- No concurrency between CPU and GPU
- Good performance of actual execution as kernels are batched together

Queue model – submit on each enqueue call



- Massive overhead on the host, we need to go to KMD which is costly
- Cost on Host greater than GPU execution time
- GPU still idle
- GPU execution very slow due to context switches

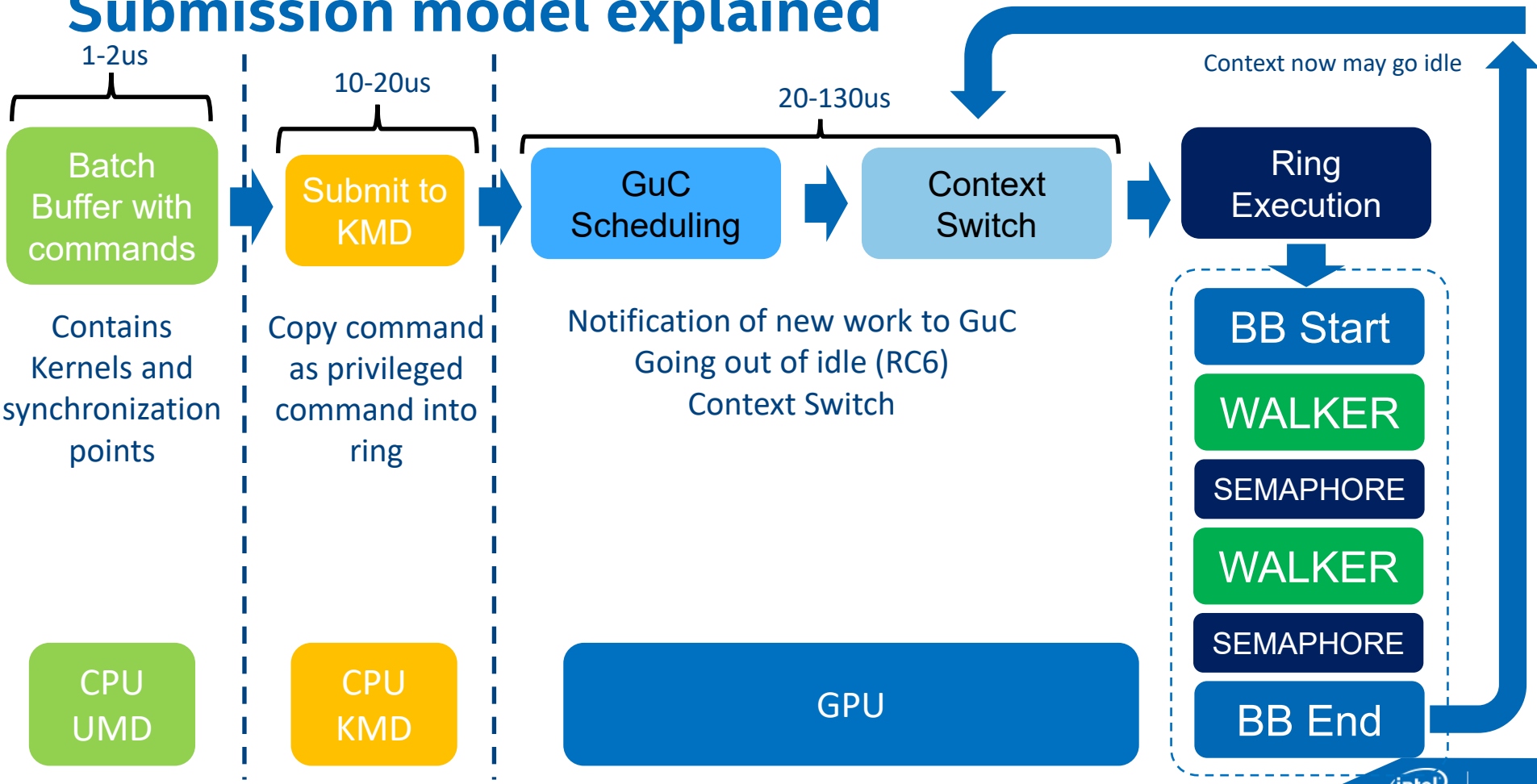
Queue model – KMD submissions very costly



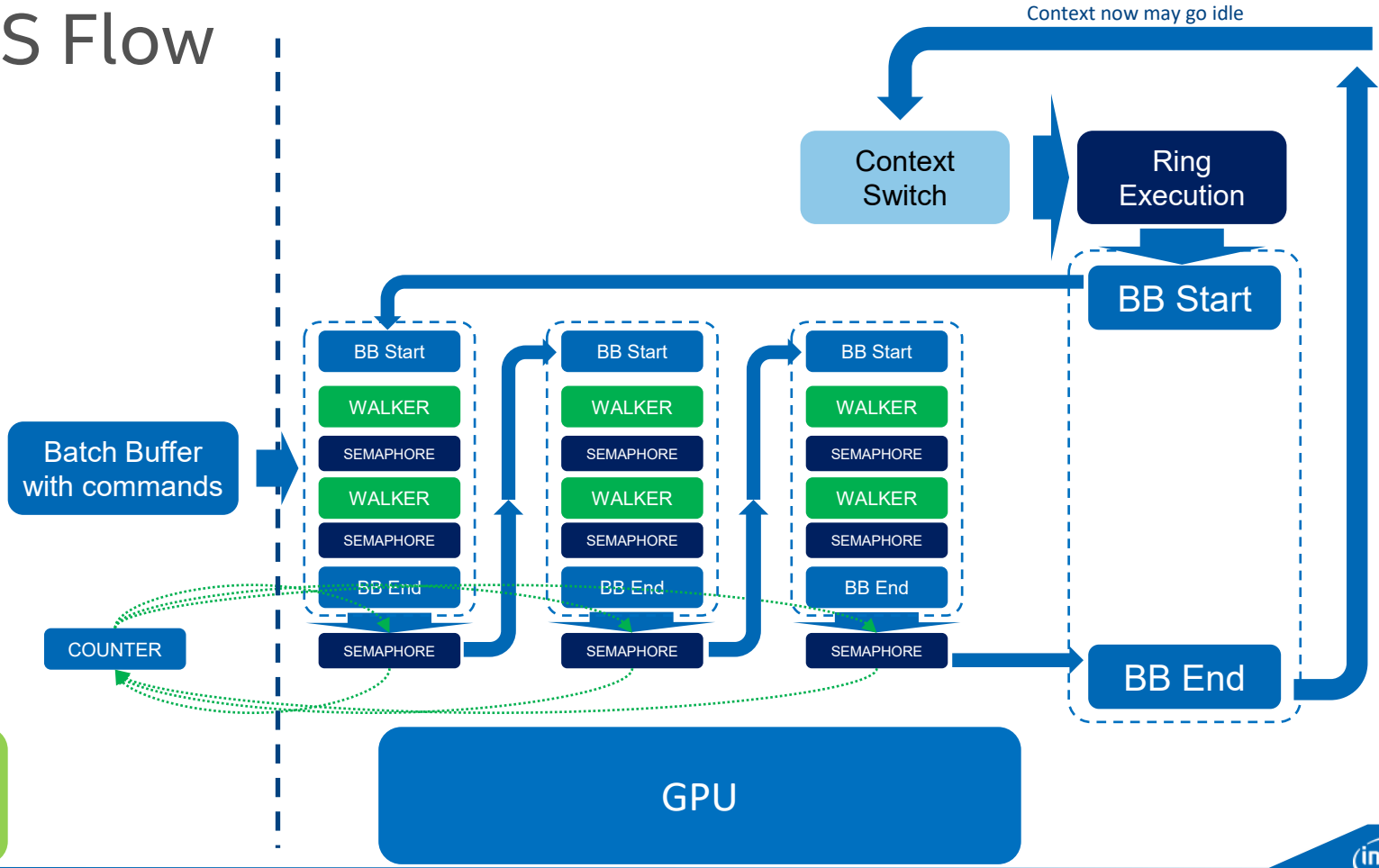
- GPU submissions are privileged operations: Ring3 → Ring0 transition
- GPU may be put into Render Standby, needs to be woken up
- Execution starts with caches cold

ULLS
(Ultra Low Latency Submission)

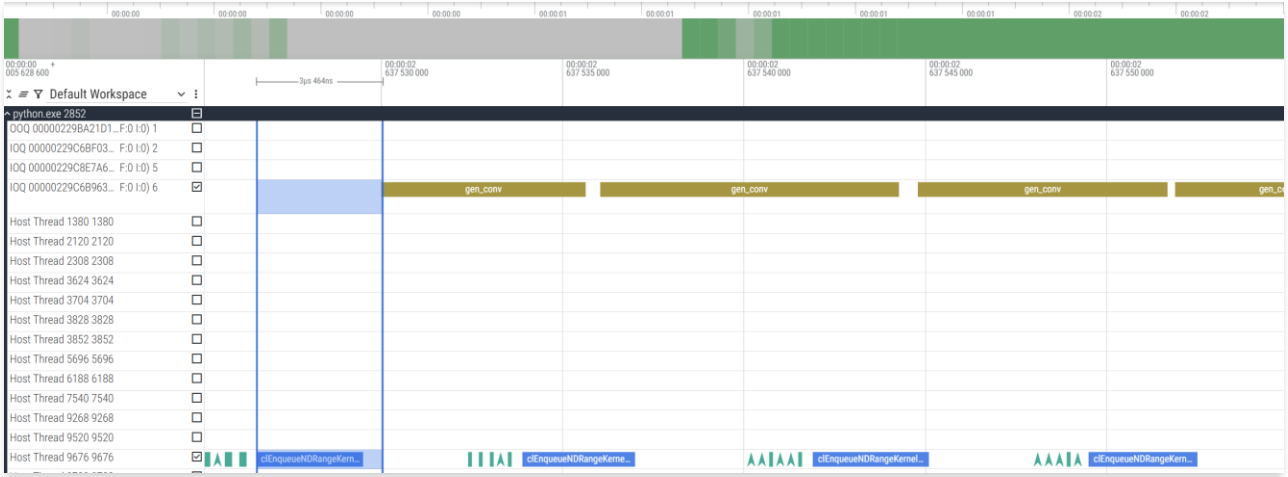
Submission model explained



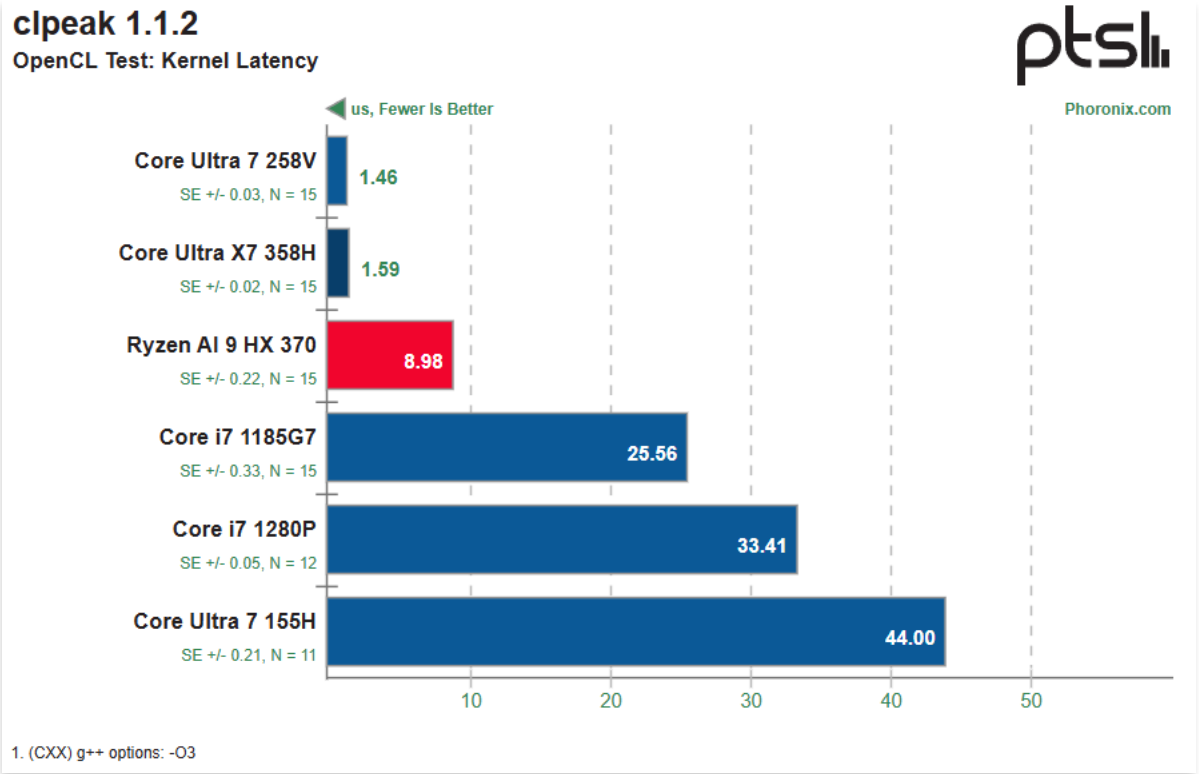
ULLS Flow


Context now may go idle
Context Switch
Ring Execution
BB Start
Batch Buffer with commands
BB Start
WALKER
SEMAPHORE
WALKER
SEMAPHORE
BB End
BB Start
WALKER
SEMAPHORE
WALKER
SEMAPHORE
BB End
BB Start
WALKER
SEMAPHORE
WALKER
SEMAPHORE
BB End
COUNTER
SEMAPHORE
SEMAPHORE
SEMAPHORE
BB End
CPU UMD
GPU

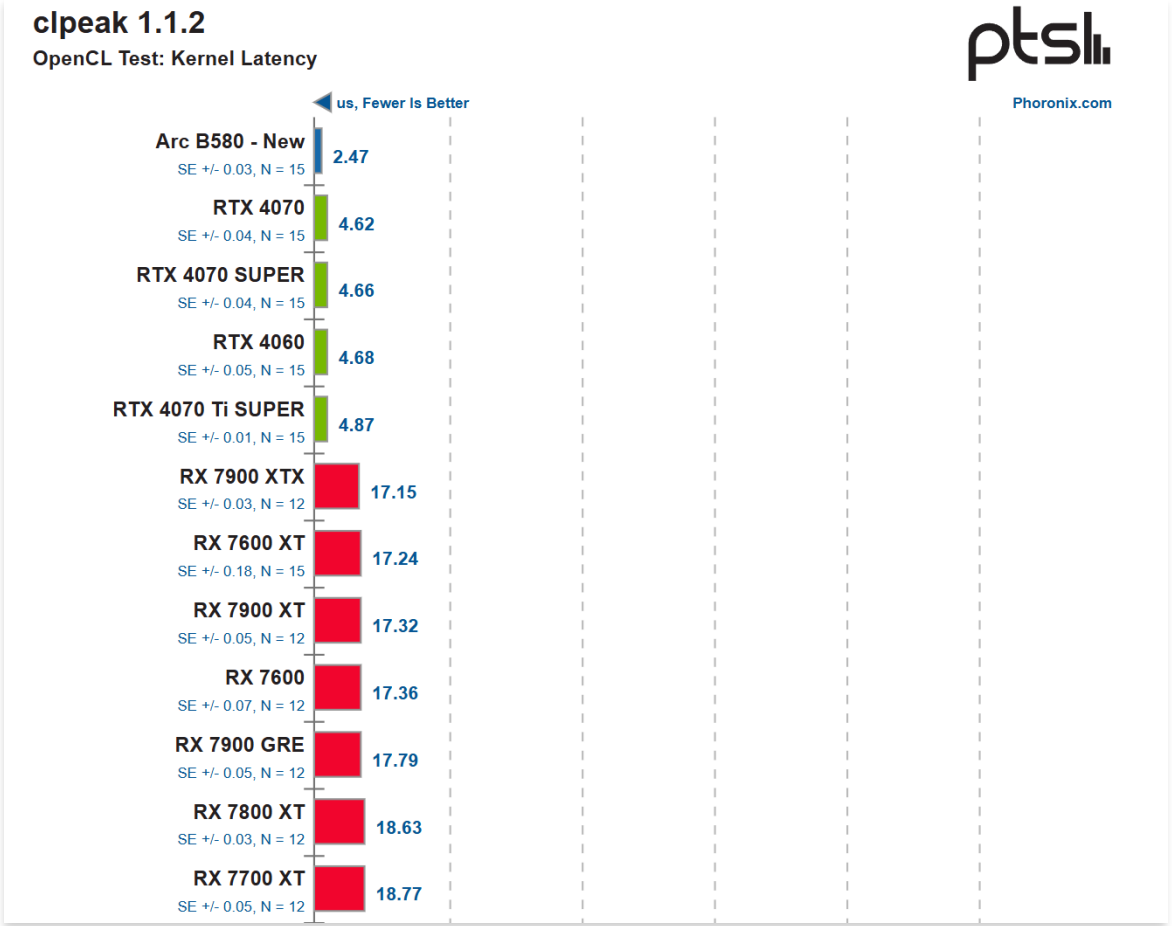
ULLS – execution starts right with first enqueue



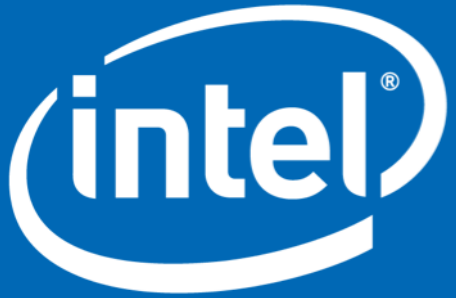
- Batching happens on the GPU itself
- Host produces more work which is immediately consumed by GPU
- No context switches result in fast GPU execution
- Implicit/Explicit cache flushes eliminated



https://www.phoronix.com/review/intel-arc-b390-compute/3



Source : https://www.phoronix.com/review/intel-b580-opencl-january/3


intel