How to Optimize Compute Driver? Let's Start with Writing Good Benchmarks!

Michal Mrozek

IWOCL 2022

Notices & Disclaimers

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

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.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

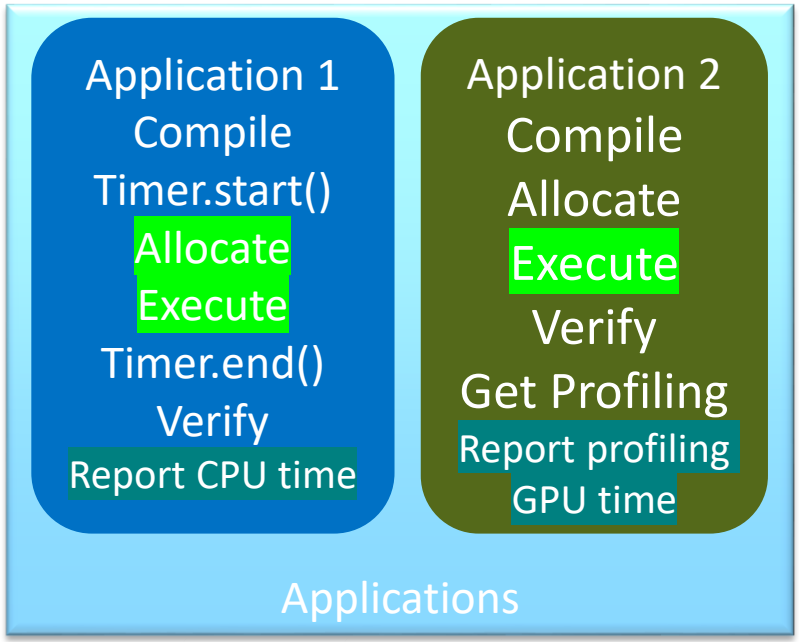
** Other names and brands may be claimed as the property of others.*

Agenda

- Why have we created Compute Benchmarks?
- Framework capabilities
- Current test sets and success story
- Plans & contribution guidelines

Why have we created Compute Benchmark?

Optimizing drivers is not straightforward!



Finding what to optimize in the driver is challenging:

- What applications measures?
- What are the bottlenecks?
- How it uses the driver?
- What needs to be optimized?
- What may be redundant?

Even if you know all of this:

- Application execution may be very long
- Application may behave differently when you start changing things

Applications composed of many elements

Create Device
Create Context
Create Programs
Create Kernels
Create Command Queues
Create Buffers/Images
Populate Buffers/Images with data
Set Arguments to the kernel
Enqueue Kernels
Flush Kernels
Read data back to the host
Wait for Kernels Completion

Applications – API usages

Optimizing a step may not be even visible in final application score

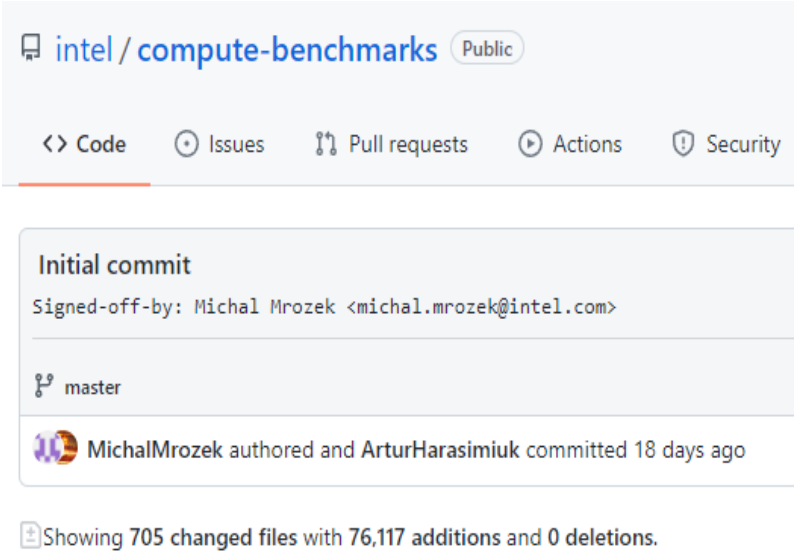
- If a step only takes 1% of whole application, even when it is removed totally it will only boost workload by ~1%.
- Applications often measure multiple steps executed together
- It is challenging to focus and optimize one step in the whole pipeline using current workloads

This created a need to develop Compute Benchmarks

- Focused tests checking only one thing per scenario
- Each test as simple as possible
- Each test produces a reliable and repeatable result
- Plain API usage to make sure there is no additional overhead
- Each test can be easily showcased as sample of API usage
- Easy download & use approach, no dependencies required
- Easy addition of new scenarios

Compute Benchmarks

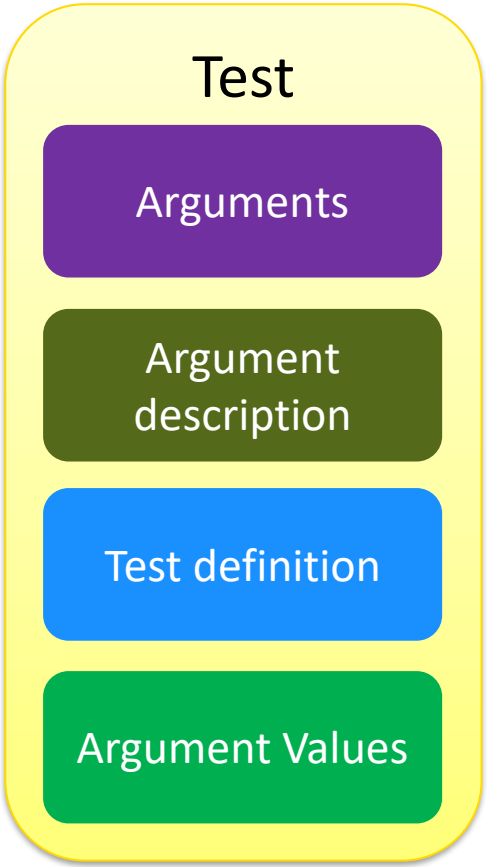
- Supports Linux and Windows
- Permissive license (MIT)
- Runs on any GPU Hardware supporting OpenCL
- Allows Vendor extensions
- Allows Vendor specific tests
- Full code in Open Source

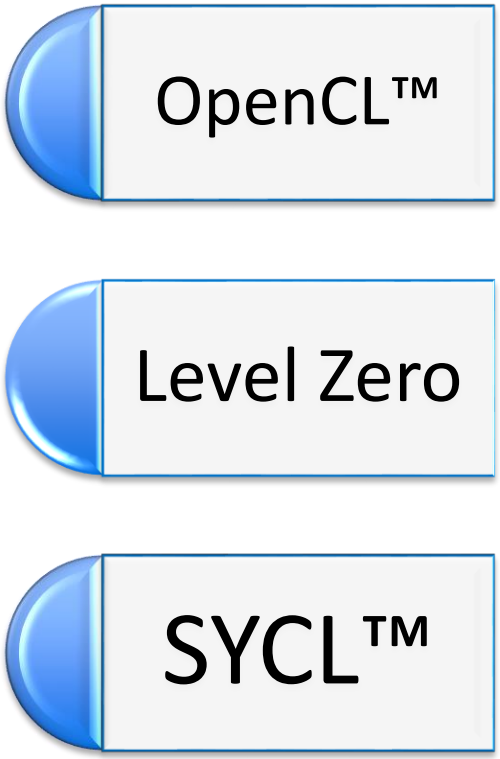


https://github.com/intel/compute-benchmarks/

Framework capabilities

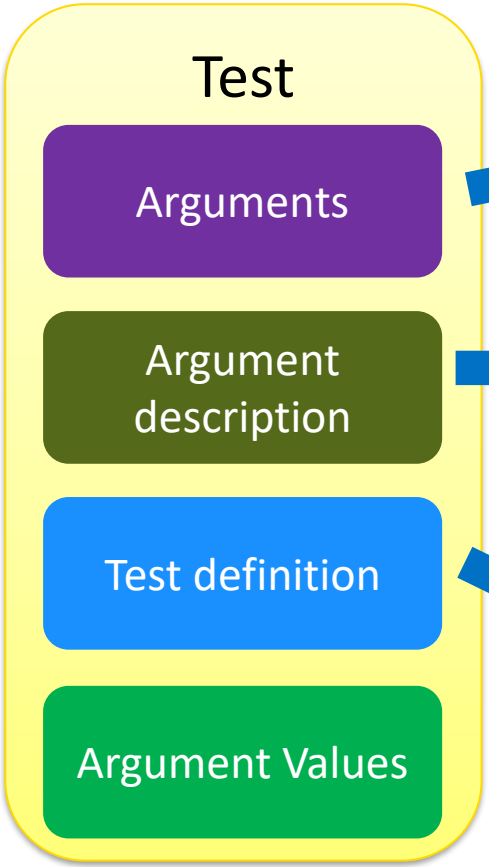
Multiple Backend Architecture





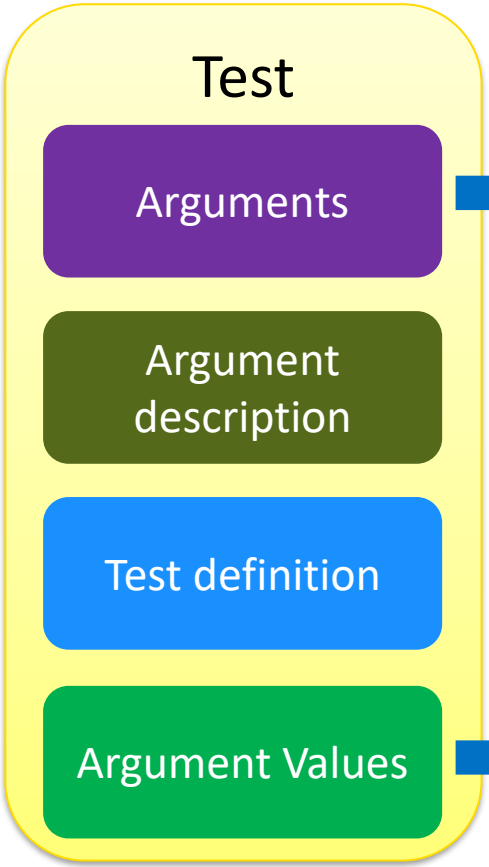
- Backend is optional for given test
- There is no upper limit on backends, currently we have OpenCL™, Level Zero and SYCL™
- Each test reuses common resource gathering logic, all results look the same
- Each Backend receives the same arguments, which allows easy comparison

Adding test – common definition




```
struct EnqueueNdrTimeArguments : TestCaseArgumentContainer {
    PositiveIntegerArgument workgroupCount;
    PositiveIntegerArgument workgroupSize;
    BooleanArgument useOoq;
    BooleanArgument useProfiling;
    BooleanArgument useEvent;

    EnqueueNdrTimeArguments()
        : workgroupCount(*this, "wgc", "Workgroup count"),
          workgroupSize(*this, "wgs", "Workgroup size"),
          useOoq(*this, "ooq", "Use out of order queue"),
          useProfiling(*this, "profiling", "Creating a profiling queue"),
          useEvent(*this, "event", "Pass output event to the enqueue call") {}
};

struct EnqueueNdrTime : TestCase<EnqueueNdrTimeArguments> {
    using TestCase<EnqueueNdrTimeArguments>::TestCase;

    std::string getTestCaseName() const override {
        return "EnqueueNdrTime";
    }

    std::string getHelp() const override {
        return "measures time spent in clEnqueueNDRangeKernel on CPU.";
    }
};
```

Adding test – common argument values




```
TEST_P(EnqueueNdrTimeTest, Test) {
    EnqueueNdrTimeArguments args{};
    args.api = Api::OpenCL;
    args.workgroupCount = std::get<0>(GetParam());
    args.workgroupSize = std::get<1>(GetParam());
    args.useOoq = std::get<2>(GetParam());
    args.useProfiling = std::get<3>(GetParam());
    args.useEvent = std::get<4>(GetParam());

    EnqueueNdrTime test;
    test.run(args);
}

INSTANTIATE_TEST_SUITE_P(
    EnqueueNdrTimeTest,
    EnqueueNdrTimeTest,
    ::testing::Combine(
        ::testing::Values(1, 100, 1000),
        ::testing::Values(1, 32, 256),
        ::testing::Values(false, true),
        ::testing::Values(false, true),
        ::testing::Values(false, true)));
```

Adding test – backend implementation

1. Common argument set passed to test
2. Arguments contain values that changes test behavior
3. Prior to every benchmark, warmup phase is done which basically does the same things that test will do
4. Each benchmark runs configurable number of iterations to collect multiple samples and provide aggregated results
5. Timer class used whenever we want to measure something from CPU perspective, measureStart & measureEnd clearly shows what is being measured in the test
6. Each test has statistic class available that is responsible for aggregating results from multiple iterations, user can specify what kind of measurement is used which will further influence results processing phase.

```
static TestResult run(const EnqueueNdrTimeArguments &arguments, Statistics &statistics) {
    // Setup
    QueueProperties queueProperties = QueueProperties::create().setProfiling(arguments.useProfiling).setOoq(arguments.useOoq);
    Opencl opencl(queueProperties);
    cl_int retVal{};
    Timer timer;

    // Get parameters for the enqueue call
    cl_event event{};
    cl_event *eventForNdr = arguments.useEvent ? &event : nullptr;
    size_t gws = arguments.workgroupCount * arguments.workgroupSize;
    const size_t lws = arguments.workgroupSize;

    if (gws == 0) {
        gws = 1;
    }

    // Create kernel
    const char *source = "__kernel void empty() {}";
    const auto sourceLength = strlen(source);
    cl_program program = clCreateProgramWithSource(opencl.context, 1, &source, &sourceLength, &retVal);
    ASSERT_CL_SUCCESS(retVal);
    ASSERT_CL_SUCCESS(clBuildProgram(program, 1, &opencl.device, nullptr, nullptr, nullptr));
    cl_kernel kernel = clCreateKernel(program, "empty", &retVal);
    ASSERT_CL_SUCCESS(retVal);

    // Warmup, kernel
    ASSERT_CL_SUCCESS(clEnqueueNDRangeKernel(opencl.commandQueue, kernel, 1, nullptr, &gws, &lws, 0, nullptr, eventForNdr));
    ASSERT_CL_SUCCESS(clFinish(opencl.commandQueue));
    if (eventForNdr) {
        ASSERT_CL_SUCCESS(clReleaseEvent(event));
    }

    // Benchmark
    for (auto i = 0u; i < arguments.iterations; i++) {
        timer.measureStart();
        ASSERT_CL_SUCCESS(clEnqueueNDRangeKernel(opencl.commandQueue, kernel, 1, nullptr, &gws, &lws, 0, nullptr, eventForNdr));
        timer.measureEnd();
        statistics.pushValue(timer.get(), MeasurementUnit::Microseconds, MeasurementType::Cpu);
        if (eventForNdr) {
            ASSERT_CL_SUCCESS(clReleaseEvent(event));
        }
    }

    // Cleanup
    ASSERT_CL_SUCCESS(clReleaseKernel(kernel));
    ASSERT_CL_SUCCESS(clReleaseProgram(program));
    return TestResult::Success;
```

Sample test output

TestCase	Mean	Median	StdDev	Min	Max	Type	Label [unit]
EnqueueNdrTime(api=ocl wgc=1 wgs=1 ooq=0 profiling=0 event=0)	4.961	4.111	49.75%	4.012	12.353	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=1 ooq=0 profiling=0 event=1)	4.813	4.431	23.87%	4.257	8.229	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=1 ooq=0 profiling=1 event=0)	4.434	4.112	21.07%	4.001	7.220	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=1 ooq=0 profiling=1 event=1)	9.370	8.948	11.13%	8.882	12.445	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=1 ooq=1 profiling=0 event=0)	2.842	2.569	27.79%	2.443	5.183	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=1 ooq=1 profiling=0 event=1)	3.804	2.912	55.41%	2.757	9.706	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=1 ooq=1 profiling=1 event=0)	2.835	2.587	24.83%	2.468	4.927	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=1 ooq=1 profiling=1 event=1)	7.878	7.339	15.13%	7.175	11.169	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=32 ooq=0 profiling=0 event=0)	3.258	3.002	23.43%	2.830	5.495	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=32 ooq=0 profiling=0 event=1)	3.477	3.200	22.91%	3.041	5.836	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=32 ooq=0 profiling=1 event=0)	3.283	2.981	24.19%	2.865	5.637	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=32 ooq=0 profiling=1 event=1)	8.557	7.926	16.30%	7.616	11.968	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=32 ooq=1 profiling=0 event=0)	2.829	2.550	27.74%	2.426	5.155	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=32 ooq=1 profiling=0 event=1)	3.952	2.782	84.99%	2.707	14.011	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=32 ooq=1 profiling=1 event=0)	2.764	2.491	28.32%	2.381	5.094	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=32 ooq=1 profiling=1 event=1)	7.888	7.511	11.69%	7.160	10.292	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=256 ooq=0 profiling=0 event=0)	3.181	2.865	26.13%	2.815	5.645	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=256 ooq=0 profiling=0 event=1)	3.396	3.151	21.19%	2.986	5.523	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=256 ooq=0 profiling=1 event=0)	3.201	2.915	23.59%	2.862	5.436	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=256 ooq=0 profiling=1 event=1)	8.041	7.733	10.87%	7.562	10.600	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=256 ooq=1 profiling=0 event=0)	2.822	2.572	25.33%	2.479	4.950	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=256 ooq=1 profiling=0 event=1)	3.059	2.795	23.99%	2.725	5.234	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=256 ooq=1 profiling=1 event=0)	2.797	2.534	24.97%	2.449	4.865	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=1 wgs=256 ooq=1 profiling=1 event=1)	8.016	7.500	14.46%	7.246	10.394	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=1 ooq=0 profiling=0 event=0)	3.198	2.927	22.80%	2.874	5.353	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=1 ooq=0 profiling=0 event=1)	3.410	3.157	20.13%	3.075	5.442	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=1 ooq=0 profiling=1 event=0)	3.240	2.898	24.74%	2.835	5.583	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=1 ooq=0 profiling=1 event=1)	8.174	7.652	14.34%	7.340	10.767	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=1 ooq=1 profiling=0 event=0)	2.795	2.558	24.04%	2.413	4.769	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=1 ooq=1 profiling=0 event=1)	3.040	2.748	26.74%	2.661	5.453	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=1 ooq=1 profiling=1 event=0)	2.780	2.521	26.32%	2.427	4.949	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=1 ooq=1 profiling=1 event=1)	8.231	7.561	14.74%	7.209	10.594	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=32 ooq=0 profiling=0 event=0)	3.179	2.944	21.02%	2.857	5.148	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=32 ooq=0 profiling=0 event=1)	3.444	3.173	22.09%	3.017	5.691	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=32 ooq=0 profiling=1 event=0)	3.201	2.925	21.47%	2.856	5.220	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=32 ooq=0 profiling=1 event=1)	8.052	7.625	11.69%	7.574	10.701	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=32 ooq=1 profiling=0 event=0)	2.809	2.576	25.41%	2.431	4.931	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=32 ooq=1 profiling=0 event=1)	3.101	2.817	25.09%	2.736	5.410	[CPU]	[us]
EnqueueNdrTime(api=ocl wgc=100 wgs=32 ooq=1 profiling=1 event=0)	2.824	2.576	24.79%	2.479	4.897	[CPU]	[us]

Framework automatically produces documentation

api_overhead_benchmark

Api Overhead Benchmark is a set of tests aimed at measuring CPU-side execution duration of compute API calls.

Test name	Description	Params	L0
AppendLaunchKernel	measures time spent in zeCommandListAppendLaunchKernel on CPU.	• --event Pass output event to the enqueue call (0 or 1) • --wgc Workgroup count • --wgs Workgroup size, pass 0 to make the driver calculate it during enqueue	✔
CreateCommandList	measures time spent in zeCommandListCreate on CPU.	• --CmdListCount Number of cmdlists to create • --CopyOnly Create copy only cmdlist (0 or 1)	✔
CreateCommandListImmediate	measures time spent in zeCommandListCreateImmediate on CPU.	• --CmdListCount Number of cmdlists to create	✔
DestroyCommandList	measures time spent in zeCommandListDestroy on CPU.	• --CmdListCount Number of cmdlists to destroy	✔

https://github.com/intel/compute-benchmarks/blob/master/TESTS.md

Each test can be run separately with any parameters

- --dumpCommandLines - will run the test suite and provide command line for each test

```
                                                                TestCase          Mean        Median         StdDev           Min           Max   Type   Label [unit]
--test=EnqueueNdrNullLws --api=ocl --gws=1 --ooq=0 --profiling=0 --event=0         5.035         4.115        52.22%         3.968        12.907  [CPU]           [us]
```

- You can later run just a single test

```
./api_overhead_benchmark_ocl --test=EnqueueNdrNullLws --api=ocl --gws=1 --ooq=0 --profiling=0 --event=0
```

- You can add --markTimers to see exactly what is being measured (useful combined with Intercepting Layers)

```
 Timer START
>>>> clEnqueueNDRangeKernel( empty ): queue = 0x557203381c60, kernel = 0x557203b2fd80, global_work_size = < 1 >, local_work_size = < NULL >
<<<< clEnqueueNDRangeKernel -> CL_SUCCESS

 Timer END
```

- --verbose -> provides output from every test iteration

```
                                                                TestCase          Mean        Median         StdDev           Min           Max   Type   Label [unit]
                                    EnqueueNdrNullLws(api=ocl gws=1 ooq=0 profiling=0 event=0)         4.936         4.059        49.13%         3.946        12.176  [CPU]           [us]
individual results: [ 12.176 4.841 4.161 4.048 4.017 3.988 3.946 4.073 4.044 4.071 ]
```

Highly efficient focused performance optimizations are easy with Compute Benchmarks !

Test suites and success story

Currently implemented test suites

API Overhead

- Host Overhead for crucial API calls

Memory Benchmark

- All variants of memory transfers, including host calls and ND-range kernels

Submission Benchmark

- All aspects of submission & completion
- Resource allocations costs
- Thread scheduling costs

Currently implemented test suites (2)

Overlap Benchmark

- Concurrent execution from multiple processes

Multithreaded Benchmark

- Concurrent execution from multiple threads

Atomic Benchmark

- Checks performance of various atomic operations types / scopes / memory orders

Kernel Benchmark

- Measure various operations done in compute kernels

Optimization – caching kernel arguments

Performance opportunity:

- Application calls clSetKernelArgSVMPointer with the same values
- Add driver mechanism to detect this scenario and skip programming logic
- Becomes very tricky as app may free SVM allocation and get the same pointer for different one
- We added all scenarios to Compute Benchmark including corner cases with reallocation
- 6x reduction in time of clSetKernelArgSVMPointer
- +13.2% workload performance
- With highly precise Compute Benchmark we were able to optimize calls that take 30ns

Contribution guidelines and plans

Contribution guidelines

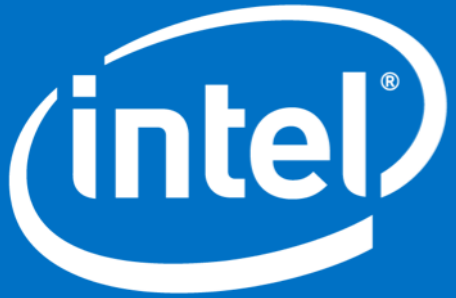
- Test needs to be quick, whole test suite should execute in seconds, not minutes
- Do not measure multiple things, focus on one thing in test
- Avoid wrappers, complicated logic, tests should be simple & comprehensive
- Make sure test is stable, do warmups, eliminate noise factor
- Do not add any external dependencies/libraries , compute-benchmarks are expected to be easy to build & run
- Do not require additional package installation in the system (apart from drivers)
- Do not add too many permutations, add new suites to keep them small
- Clearly describe parameters and test definition, it should be obvious what test measures

Plans & call to action

- Initial version of Compute Benchmarks already open sourced -> https://github.com/intel/compute-benchmarks/
- Try it out !
- New tests being added on daily basis, plan is to have all OpenCL™ / Level Zero APIs covered
- Contributions highly welcomed!

Acknowledgments

Thanks to Ben Ashbaugh , Lukasz Jobczyk and Dominik Dabek for help with the presentation!


intel

Useful additional capabilities

- --doNotPrintBandwidth – Bandwidth tests will provide time in us instead of GB/s, very useful to translate bandwidth test to latency test
- --noop – do not run tests, just print their names, very useful to see what are the tests in given test suite
- --csv - prints results in csv format
- We also have tools directory for storing useful tools that are handy during day-to-day work
- Tests are based on Google Test Framework, so traditional gtest command line parameters also works, like –gtest_filter etc.