IWOCL 2016

OpenCL-Based Mobile GPGPU Benchmarking: Methods and Challenges

Rotem Aviv
Guohui Wang

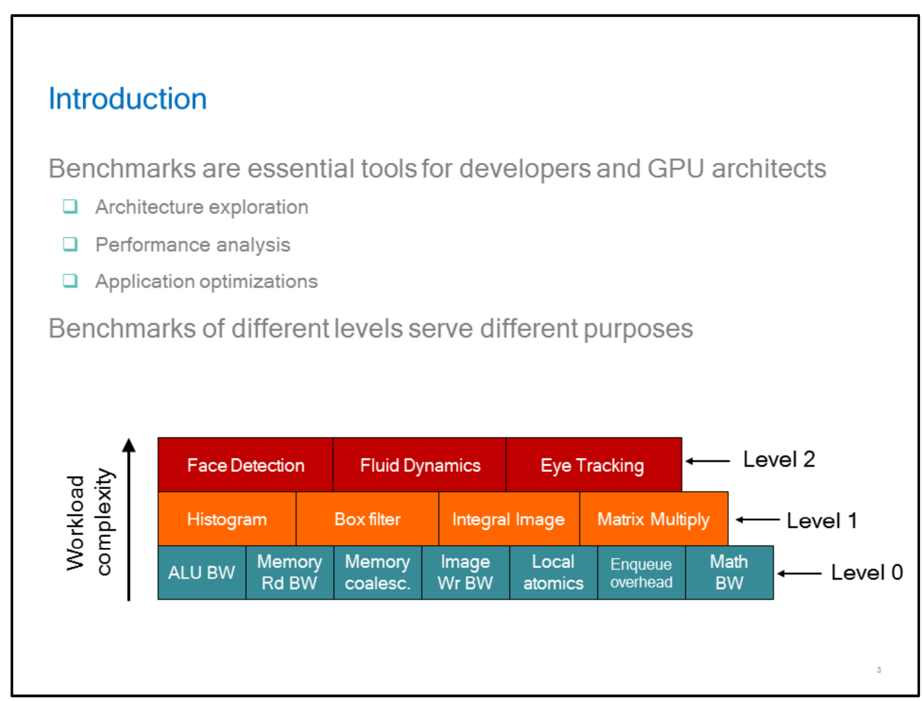
Qualcomm Technologies, Inc.

April 21st 2016

Agenda

- Introduction and Motivation
- Methods and Challenges in Benchmark Design
- Conclusions

2



Benchmarks levels are a suggestion for categorizing different tests based on complexity. The SHOC benchmark applies similar categorization (https://github.com/vetter/shoc).
Level 1&2 (high-level) benchmarks measure the performance of a device in running certain functionalities and use-cases, such as histogram, box filter, integral image, or more complex algorithms such as face detection, and fluid dynamics simulation.
Level 0 (low-level) benchmarks measure performance of specific logic modules or low-level capabilities of a device such as ALU bandwidth, memory read bandwidth, kernel enqueue overhead, etc.
Mobile GPGPU benchmarking is sensitive to multiple system factors such as power management, driver overhead, context switch, compiler, HW differences across platforms, and more.

Performance Variance

Why do we observe performance variance?

- Dynamic system memory load
- Power management and clock throttling
- Thermal limitation
- Timer accuracy

How can we mitigate performance variance?

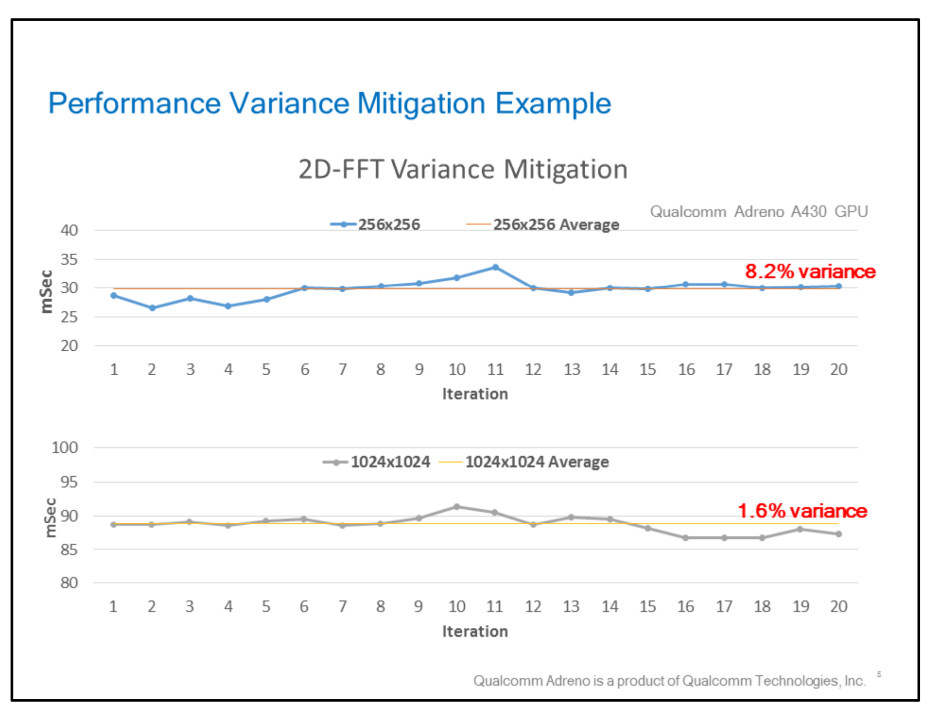
- Longer workloads
- Multiple test iterations
- Minimize host-device sync (e.g. clFinish)
- In-test variance monitoring
- Disable non-benchmark tasks
- Correlate multiple timer data
- Use timer that runs at constant rate

Factors contributing to performance variance:

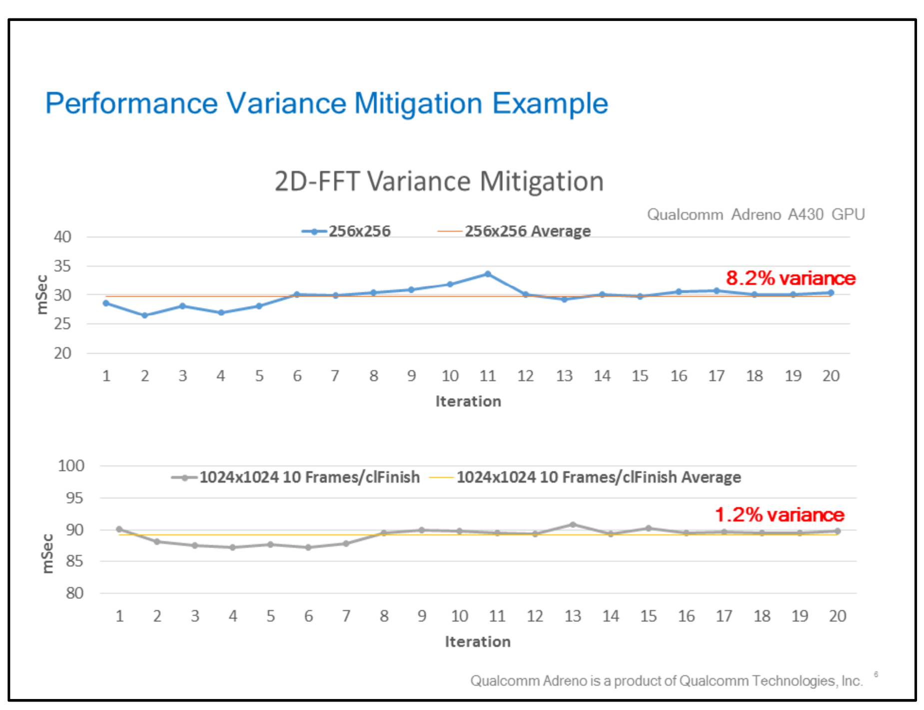
- Dynamic system memory load, as the memory serves multiple clients in the system
- Power management mechanism will throttle clocks through different parts of the system, which may change performance across various benchmark runs
- Thermal limitation will protect HW from over-heating, may limit performance at certain physical conditions
- Timer accuracy may introduce variance in measured performance

Mitigation

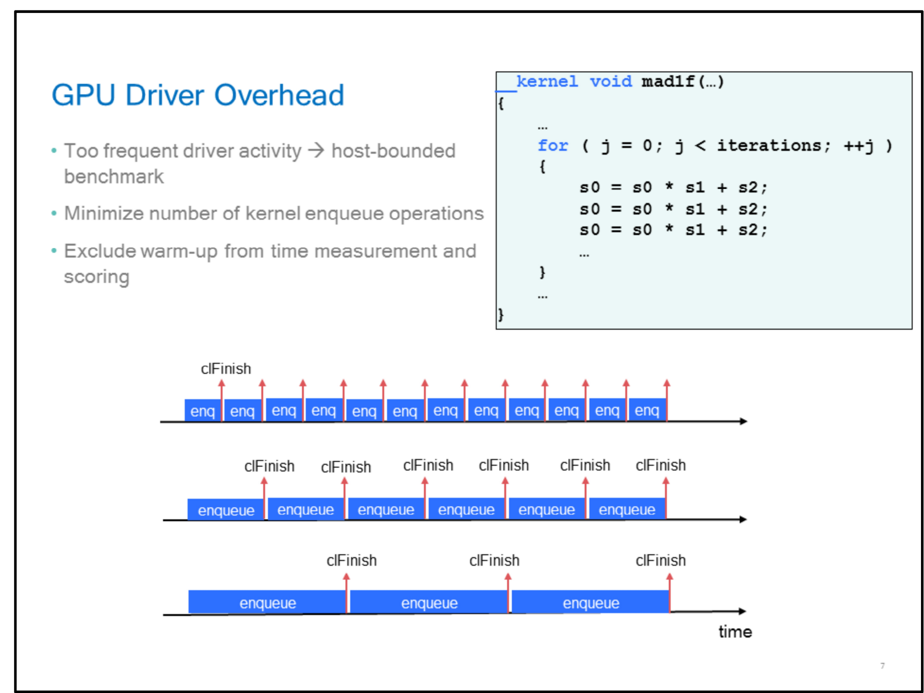
- Use longer workloads, run multiple test iterations while monitoring variance
- Minimize host-device sync operations, such as clFinish
- Minimize non-benchmark tasks
  - Ease system memory load and CPU workload
  - Make sure CPU and memory load is low before and during benchmark run
  - Benchmark can detect high CPU load automatically, disable test run at extreme conditions
  - Avoid any unnecessary application activity during benchmark execution
- Use more than one timer if possible, correlate results. OpenCL 2.1 clGetDeviceAndHostTimer allows synchronizing host and device timers, and could assist in correlation process
- Use system timer that runs at constant clock rate, if possible, and is not affected by power management



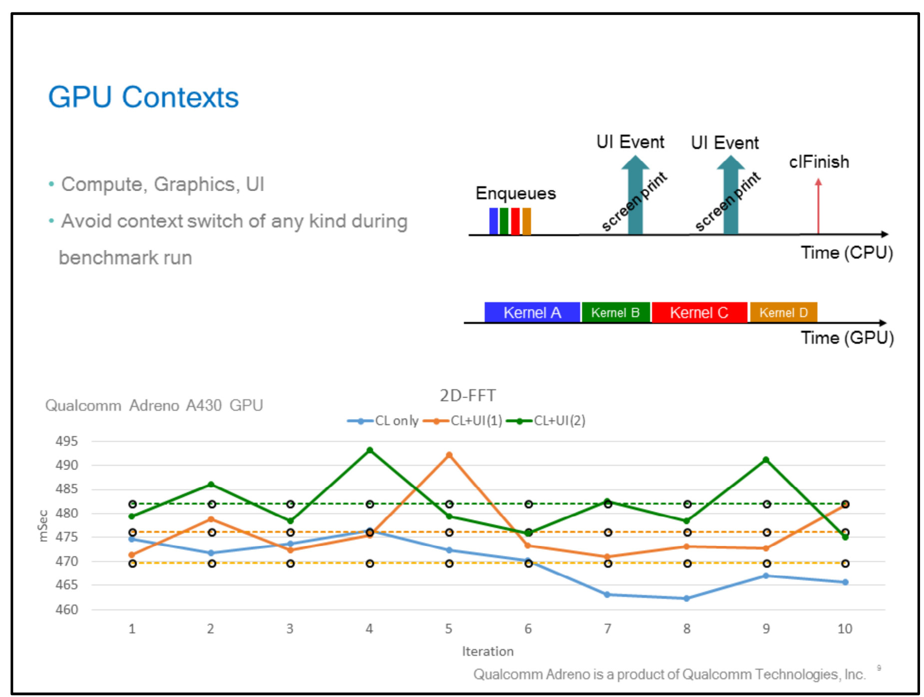
In this case we look at performance of 2D-FFT running on Qualcomm Adreno A430 GPU using OpenCL. Increasing the workload from input of 256x256 to 1024x1024 reduces variance from 8.2% to 1.6%.



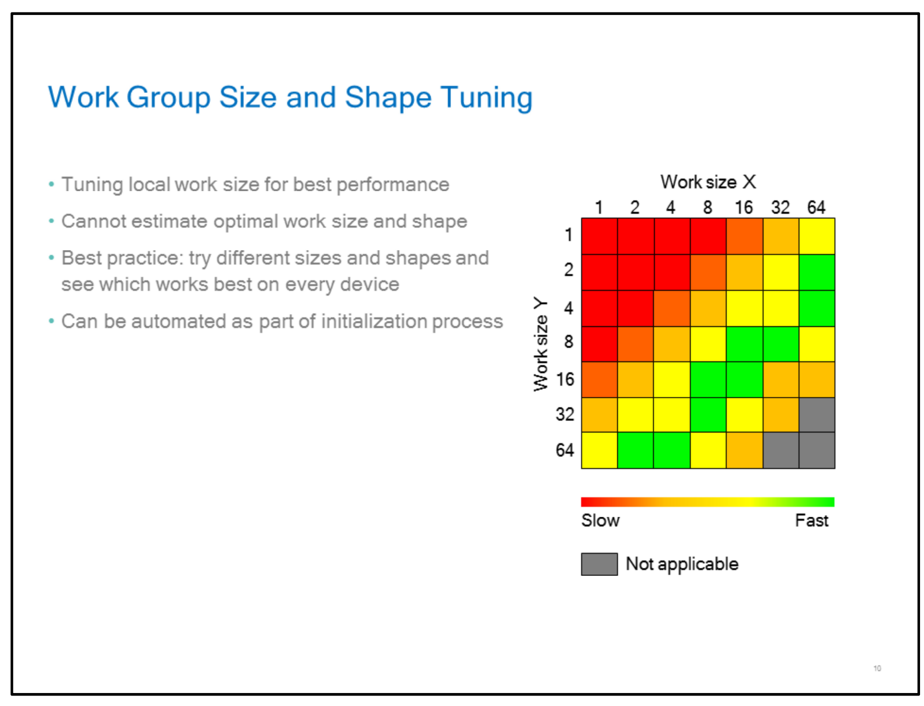
Further variance reduction is achieved through reducing host-device sync (clFinish) from every frame to once every 10 frames. As a result, variance is reduced from 1.6% to 1.2%.



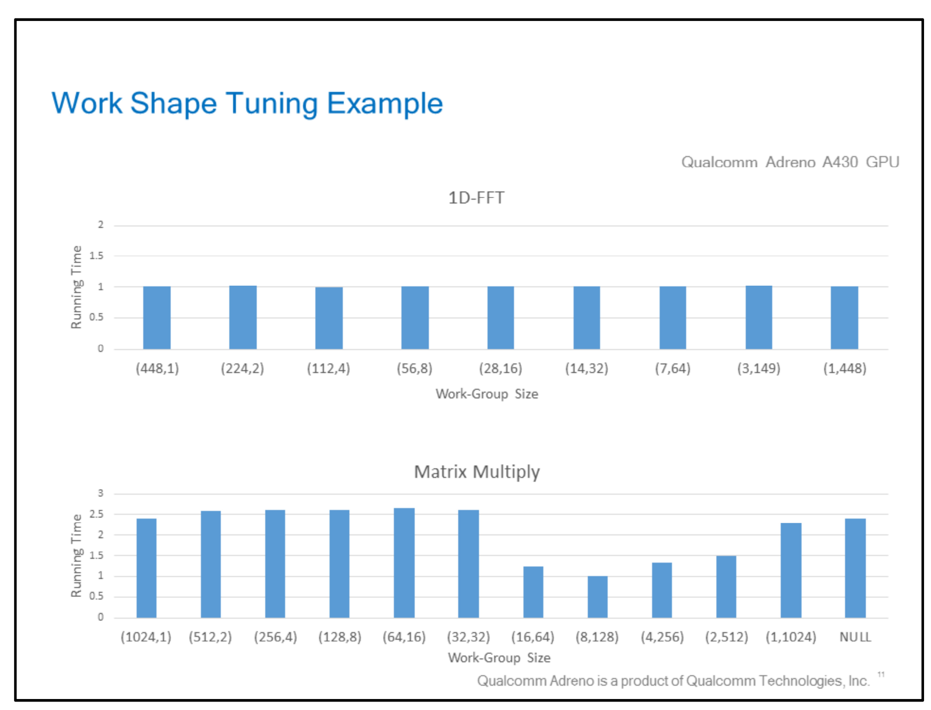
Too frequent driver operations (triggered by OpenCL API calls) may shift the workload bottleneck to the host, making the benchmark to be bounded by the host and not by the GPU. To reduce driver activity in the benchmark, minimize number of kernel enqueue operations. Increase workload per kernel enqueue and reduce number of enquques. For small kernels, consider merging kernel functionality to create larger kernels.
Warm-up enqueue may improve overall performance in some cases. Exclude warm-up activity from time measurement.



GPU contexts can be of different types and triggered by different applications. In a GPU compute benchmark, the main workload typically runs through the benchmark's compute context. Best performance can be achieved when the workload context is not switched while running the workload and measuring time. However, if a context switch occurs, the benchmark workload execution may be interrupted in some cases, and the measured performance may be lower.
In this example, we show performance of 2D-FFT application, where 4 kernels are enqueued sequentially. Shortly after the kernels are enqueued and before clFinish returns, the application launches a UI event in the form of a screen print. Since printing to the screen is handled by the UI context, a context switch occurs. As a result, measured performance is slower. As can be seen in the chart, performance slowdown is proportional to the number of UI events.



Why does the work-group size and shape affects performance? The work-group size and shape determine the memory access pattern, which may affect performance in memory-bound cases. The Memory access pattern can determine effective memory bandwidth, bank conflicts, cache utilization and other factors.
How to determine the work-group shape? Try various shapes (1-D, 2-D, 3-D), select the best performing one. Naturally, the local shape is limited by the global shape.
For example, work size tuning in 2-D case: initial size = N, measure performance with work-group size = (N,1), (N/2,2), (N/4, 4), …, (1,N), change N and try various work-group shapes again.
The figure on the right is a theoretic illustration of performance with different 1D/2D work size and shapes (diagram may change for different use-cases). The grey area represents non-applicable work-group size (larger than maximum work-group size). Small work-group size may yield slow performance due underutilization of HW, whereas larger work-group size will better utilize the HW and have better performance. Good performance is achieved by utilizing a large enough work-group and by tuning the shape. In some cases, a work-group size that is too large may slow-down performance.



As can be seen in the charts above, the work group shape tuning has a significant effect on matrix multiply performance in Qualcomm Adreno A430 GPU, but almost no effect in case of 1D-FFT.

Compiler Optimization in Low-Level Benchmarks

- Compiler optimization can change the test functionality, making the result invalid

```
float data_a = mem_in[get_global_id(0)];
float data_b = mem_in[get_global_id(0)+1];
for(int i=0; i < LOOP_CNT; ++i)  {
        data_a += data_b;
        data_a += data_b;
        …
}
```

Compiler optimization → `data_a = data_b * C`

```
float data_a = mem_in[get_global_id(0)];
float data_b = mem_in[get_global_id(0)+1];

float data_c = mem_in[get_global_id(0)+2];
float data_d = mem_in[get_global_id(0)+3];
while(data_a < LOOP_CNT)  {
        data_a += data_b;
        data_c += data_d;
        …
}
mem_out[get_global_id(0)] = data_a + data_c;
```

Unlikely to be optimized by compiler (`data_a += data_b;`)

12

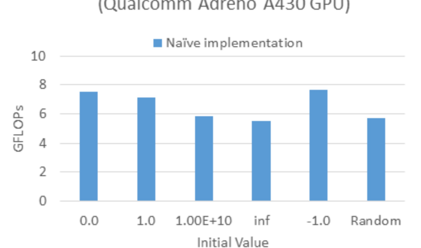
QUALCOMM®

Math Benchmarking

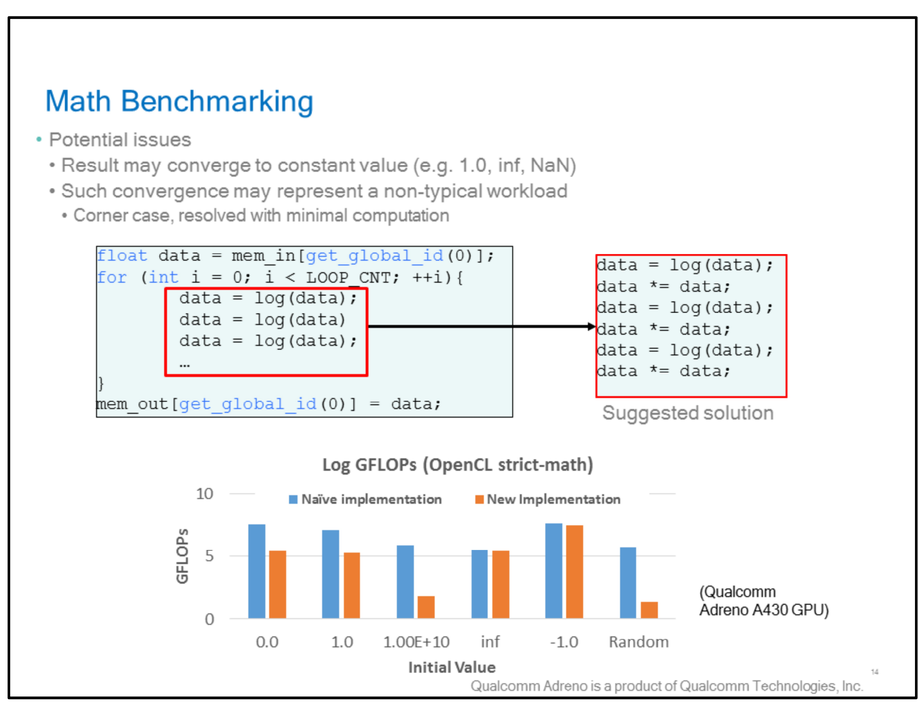
- Potential issues
  - Result may converge to constant value (e.g. 1.0, Inf, NaN)
  - Such convergence may represent a non-typical workload
    - Corner case, resolved with minimal computation

```
float data = mem_in[get_global_id(0)];
for (int i = 0; i < LOOP_CNT; ++i){
        data = log(data);
        data = log(data)
        data = log(data);
        …
}
mem_out[get_global_id(0)] = data;
```

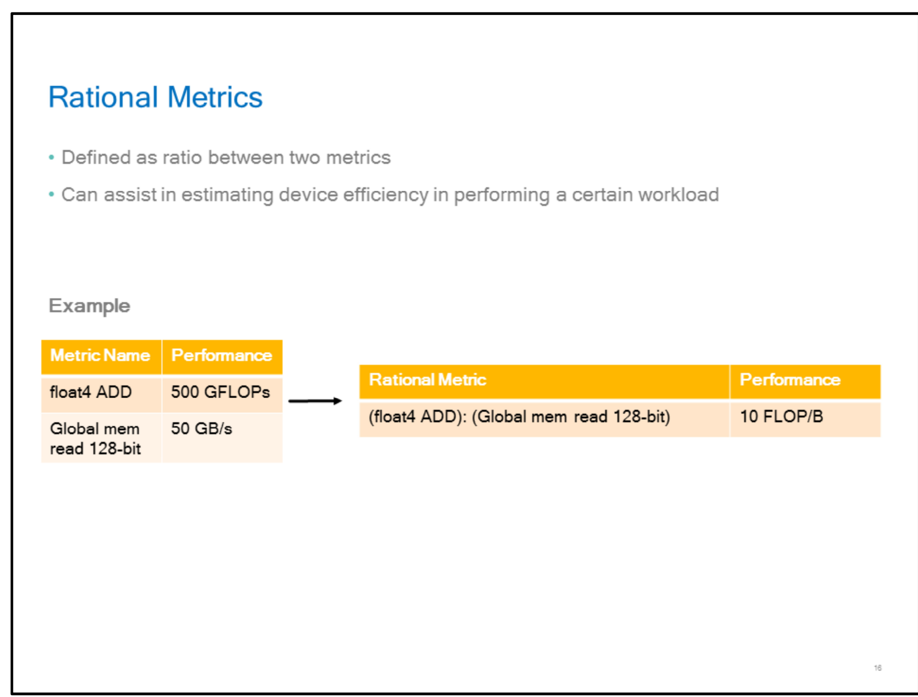
Log GFLOPs (strict-math)
(Qualcomm Adreno A430 GPU)



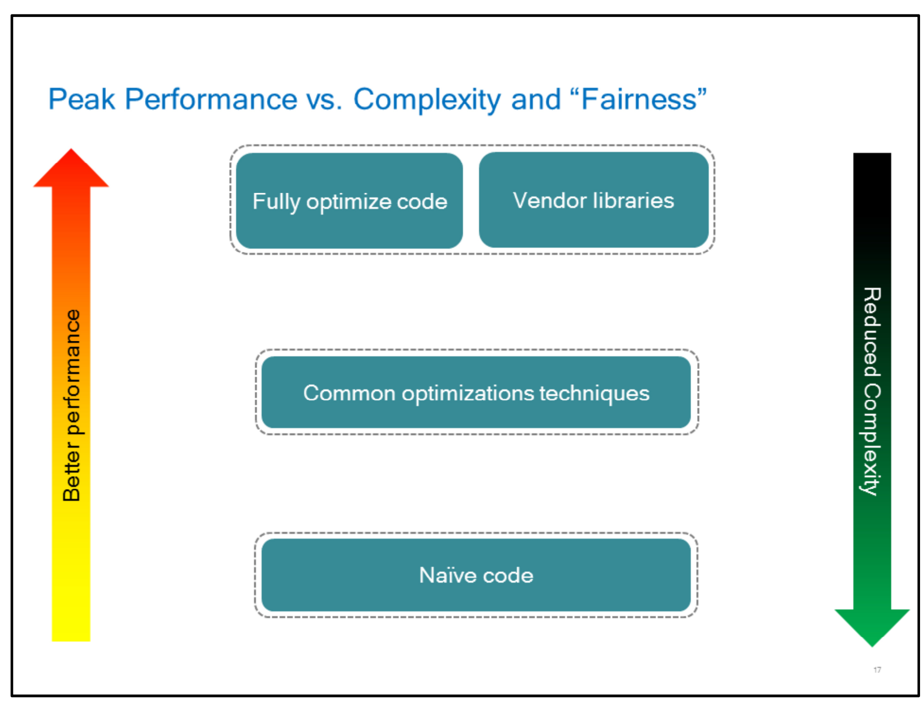
Qualcomm Adreno is a product of Qualcomm Technologies, Inc. 13



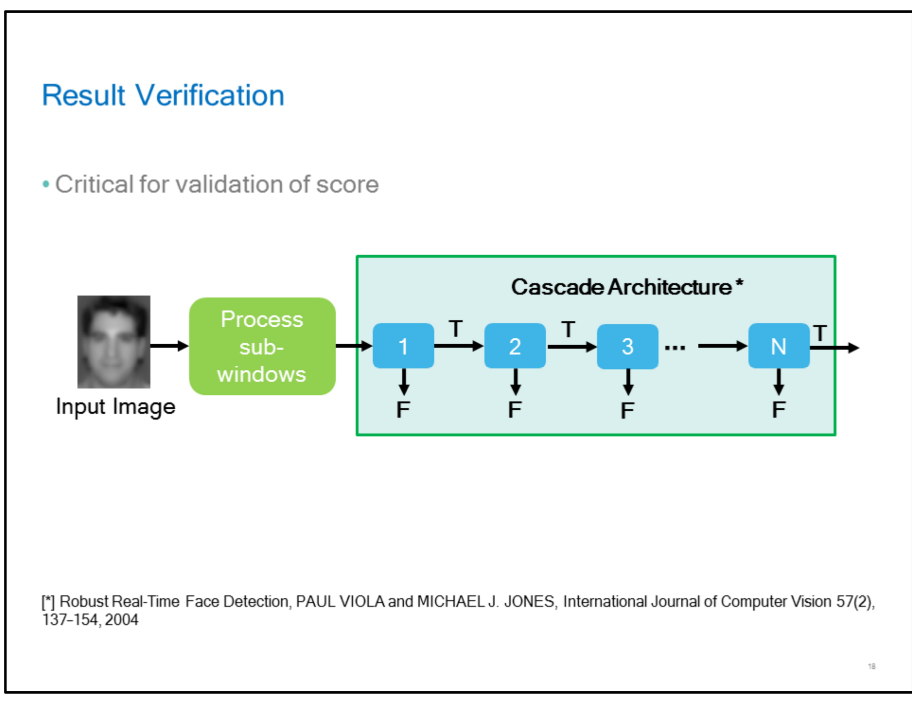
Math functions, in some cases, are implemented by complex set of instructions, which may check for corner cases and have multiple execution paths. As a result, math function performance may depend on the input value. In this case, we look at performance of function log in OpenCL strict-math mode running on Qualcomm Adreno A430 GPU. By modifying the input value to log() before every function call, we can avoid log result from convergence into a constant value for certain types of inputs, and the measured result reflects a more typical performance of log in a real use-case. Developing math benchmarks requires special attention to functions which may perform differently depending on the input value.



Relative performance metrics helps in understanding the strength and weakness points in a device architecture. With the example shown above, we can compute the relative performance of float4 ADD to global memory read of 128-bit data, and derive the rational performance of these two metrics. Given a certain workload with known requirements for operations such as data fetch bandwidth and ALU operation count, rational performance metrics can also assist in predicting whether a device can efficiently execute the workload.



Developing a high-level benchmark requires balancing between several considerations. On the one hand, a correct assessment of performance requires running a workload that can maximize device utilization and efficiency. On the other hand, the benchmark has to be designed in a cross-platform approach so that it can be executed on multiple devices. Optimizing performance of a benchmark often required careful optimization and even using vendor libraries, if those are available. In some cases, using dedicated HW features for better performance generates differences in results, mainly due to HW differences in features that are not explicitly defined by the API spec. These differences may expose differences in workload across devices, making the performance comparison less “fair”. In order to make sure all devices run the exact same workload and to make the benchmark cross-platform tool, a developer may compromise on performance. Such compromise can include using only common optimization techniques that are supported by multiple devices, which are less likely to generate differences in result. Using naïve code is beneficial in terms of development time and cross-platform capability, but will poorly utilize the device, making the benchmark result highly uncorrelated with the device true capability. Benchmark developers should find the right balance between these different considerations. Achieving “fairness” by maintaining workload consistency across devices is highly important, yet developers should also try to improve performance using common and device specific optimization, while utilizing the latest API features. The advancement of compilers and drivers, as well as the gradually increasing availability of optimized libraries can be leveraged by benchmark developers in creating more accurate and balanced benchmarks.



Result verification is a key component in benchmarking. It enables the benchmark to make sure the device executed the functionality as expected, validating the performance measurement.
Workload output should be compared to pre-calculated reference. Reference data can be computed offline or outside time measurement scope. Online reference computation may increase chances of thermal gating. It is recommended to avoid long and compute-heavy reference computations during benchmark run.
In some use-cases, performance may be data dependent. One example is the Viola-Jones Face Detection algorithm*, which is based on a cascade of logic modules. HW differences across devices may cause differences between intermediate results of the same stage across different devices, leading to workload and performance differences. To make sure workload is consistent across devices, it is recommended to compare workload output to reference data in intermediate stages as well as in the final stage of the computation.
When comparing workload output to reference, define error margins large enough to accommodate differences between various GPU implementations. Error margin definition may require experimenting with multiple devices.

Other Key Topics in GPGPU Benchmarking

Not discussed due to lack of time

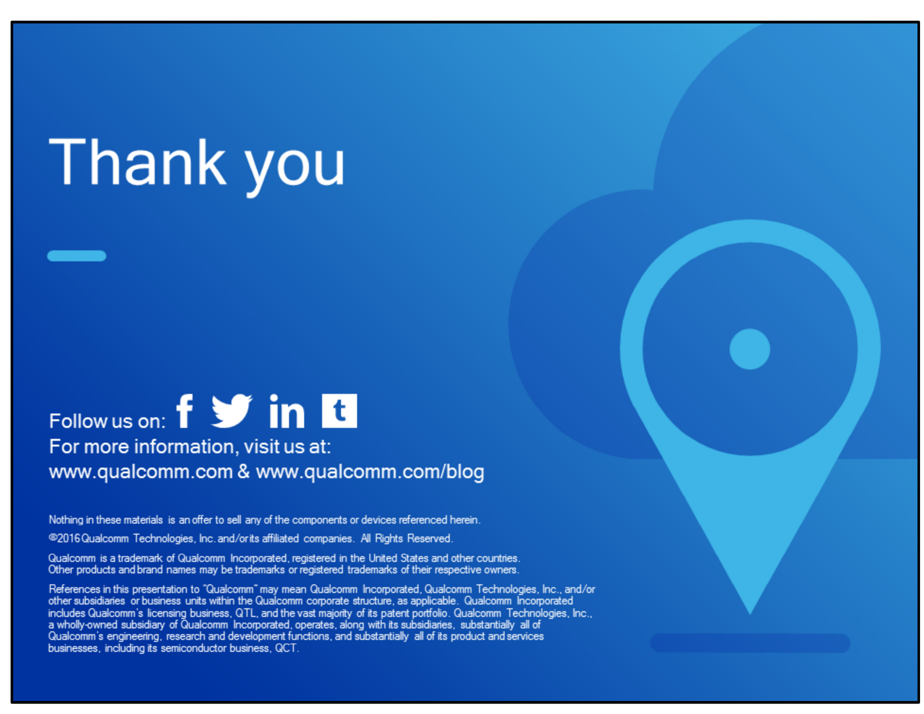
- Global work-size selection
  - Make sure GPU is fully utilized
  - Partially active work-groups should be relatively small in number
  - Tune global shape if possible
- GPU Workload
  - Balance workload length and thermal limit
- OpenCL compiler flags
- Optimization
  - Use common optimization techniques shared by multiple devices
    - Vectorization, native math, SIMD friendly (minimal branching), minimal data type, etc.

19

Conclusion

- **Utilize the desired benchmark level** to extract low-level HW performance data, estimate device performance of real use cases and compare performance across device
- **Mobile devices are sensitive to multiple system-level factors:** power management, driver overhead, context switch. Design your workloads to be long enough, avoid context switch and keep CPU-GPU sync events to a minimum
- **Local work-size tuning** is a simple low-cost approach for obtaining better performance in many use-cases
- While developing low-level benchmark, **be aware of compiler optimization**, try to detect it and work-around if possible
- **Balance performance with fairness and design consideration**. If comparing performance across platforms, make sure workload is identical. Comparing across APIs can be misleading due to differences in API features and precision
- **Verification is a key step** in validating measured performance. It is also a tool for ensuring workload consistency.

20


Thank you
Follow us on:
For more information, visit us at:
www.qualcomm.com & www.qualcomm.com/blog
Nothing in these materials is an offer to sell any of the components or devices referenced herein.
©2016 Qualcomm Technologies, Inc. and/or its affiliated companies. All Rights Reserved.
Qualcomm is a trademark of Qualcomm Incorporated, registered in the United States and other countries.
Other products and brand names may be trademarks or registered trademarks of their respective owners.
References in this presentation to "Qualcomm" may mean Qualcomm Incorporated, Qualcomm Technologies, Inc., and/or other subsidiaries or business units within the Qualcomm corporate structure, as applicable. Qualcomm Incorporated includes Qualcomm's licensing business, QTL, and the vast majority of its patent portfolio. Qualcomm Technologies, Inc., a wholly-owned subsidiary of Qualcomm Incorporated, operates, along with its subsidiaries, substantially all of Qualcomm's engineering, research and development functions, and substantially all of its product and services businesses, including its semiconductor business, QCT.