OpenCL Benchmark Progress Report (2016.11.24)

1. Float-point operation VS. Integer operation:

   The performance should be heavily relied on the instruction set of the processor and efficiency of instruction execution, which means it varies between different hardware. Considering the binary representation of float-point value is much more complex than the integer one, which intuitively may be more instructions involved in one operation for FP, see also:
   http://nicolas.limare.net/pro/notes/2014/12/12_arit_speed/

   It shows that on different platforms the FP operation are all slower than Integer operation.

   Even if the FP operation had been optimized on the target, making it faster than integer operation, the data transfer should still contribute the major part of the total delay,

   "A very, very slow floating point operation that already has the data in cache will be many times faster than an integer operation where an integer needs to be copied from system memory."

   "A rule of thumb would be that the memory footprint of your data has greater impact on speed than the arithmetic does nowadays."
2. MP (multiprocessor) split thread blocks (work-group) into wraps (32 threads) SIMT (instruction unit) will schedule the wraps to execute.
3. Size: private < local < constant <= global; Speed: private(registers) > local > constant >= global
4. Instruction process: Read operands->Execute->Write results(all for one wrap 32 threads)
5. GPU performs better for arithmetic intensity rather than memory transaction.
6. For NVDIA comparing the executing speed of Integer and Float are almost the same expect for division:

   Single-Precision Floating-Point Basic Arithmetic

   Throughput of single-precision floating-point add, multiply, and multiply-add is 8 operations per clock cycle.

   Throughput of reciprocal is 2 operations per clock cycle.

   Throughput of single-precision floating-point division is 0.88 operations per clock cycle, but `native_divide(x, y)` provides a faster version with a throughput of 1.6 operations per clock cycle.

Integer Arithmetic

Throughput of integer add is 8 operations per clock cycle.

Throughput of 32-bit integer multiplication is 2 operations per clock cycle, but **`mul24`** provide 24-bit integer multiplication with a troughput of 8 operations per clock cycle. On future architectures however, **`mul24`** will be slower than 32-bit integer multiplication, so we recommend to provide two kernels, one using **`mul24`** and the other using generic 32-bit integer multiplication, to be called appropriately by the application.

Integer division and modulo operation are particularly costly and should be avoided if possible or replaced with bitwise operations whenever possible: If **`n`** is a power of 2, **`(i/n)`** is equivalent to **`(i>>log2(n))`** and **`(i%n)`** is equivalent to **`(i&(n-1))`**; the compiler will perform these conversions if **`n`** is literal.

Bitwise Operations

Throughput of any bitwise operation is 8 operations per clock cycle.

Type Conversion

Throughput of type conversion operations is 8 operations per clock cycle.

Sometimes, the compiler must insert conversion instructions, introducing additional execution cycles. This is the case for:

- Functions operating on **`char`** or **`short`** whose operands generally need to be converted to **`int`**,
- Double-precision floating-point constants (defined without any type suffix) used as input to single-precision floating-point computations.

This last case can be avoided by using single-precision floating-point constants, defined with an **`f`** suffix such as **`3.141592653589793f`**, **`1.0f`**, **`0.5f`**.

7. For Memory Instruction and access:

   As an example, the throughput for the assignment operator in the following sample code:

```
__local float shared[32];
__global float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

   is 8 operations per clock cycle for the read from global memory, 8 operations per clock cycle for the write to shared memory, but above all, there is a latency of 400 to 600 clock cycles to read data from global memory.

   Much of this global memory latency can be hidden by the thread scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete.

8. The global memory access by all threads of a half-warp is coalesced into one single memory transaction. For instance:
   For buffer&int method, each thread access a 1-byte(8bit) data, then one transaction size should be 32-byte segment
   For buffer&float method, each thread access a 4-byte data, then one transaction size should be 128-byte segment
   And it is coalesced access under BUFFER method.