OpenCL Benchmark Progress Report (2016.12.08)

1. Performance (for one kernel) on target before optimization:

Type/Memory	Buffer	Image
Integer	92.324ms	216.713ms
Float	118.247ms	128.237ms

2. Buffer vs Image Major Factors:
   1) How many times of Data transaction? (usually, the time to access global memory is 400 to 600 times longer than other operations)
      Buffer: considering 32 work-items as a wrap, they are parallel executed, memory access request in first work-item will result in a coalesced data retrieve, which has a size of 32*sizeof(data), then the data could be shared among the 32 work-items. Thus, when applied to Gaussian filter, only **9 times** of data transaction needed for a wrap.
      Image: image method only takes advantage of adjacent pixels cache scheme, however, when implement Gaussian filter, considering it only cache the neighbor pixels with reach index of 1, then it demands about **32 times** of data transaction to finish one wrap.

      PS: One transaction with larger amount of data is still better than many times of smaller data transaction.

   2) The Division operation is time-consuming for integer.
      Before optimization, Image method has to compute (l_g.x/16, l_g.y/16, l_g.z/16, l_g.w/16), even if x,y,z,w are the same value, it still operate 4 times of division, while in Buffer method, it only takes once.

3. Integer vs Float Major factor:
   1) In integer implementation, we actually use unsigned char type, which means 8 bits for each data, while for float type, it takes 32 bits for one data, which means a four-time larger data size needed to be transmitted. Considering the same bandwidth, the later one will take more time to finish.

4. Optimization
   1) Since in our implementation, the picture only has one channel, we can reduce the repeated divisions in Image method:
      (l_g.x/16, l_g.y/16, l_g.z/16, l_g.w/16) → output=l_g.x/16; (output, output, output, output)
      Performance(the result after optimization shows in brace):

Type/Memory	Buffer	Image
Integer	92.324ms	216.713ms(179.58ms)
Float	118.247ms	128.237ms(128.587ms)

      Conclusion: It is obvious that integer division is pretty time-consuming on the target, while for float point, the time could be ignored.

2) It is recommended to replace integer division as bit shift operation:
   1_g/16 → l_g >> 4;
   And float point division could be replaced by native_divide().

Type/Memory	Buffer	Image
Integer	92.324ms(85.172ms)	179.58ms(165.95ms)
Float	118.247ms(118.32ms)	128.237ms(127.517ms)

   Conclusion: division optimization only shows significant improvement on Integer.

3) Each kernel code is implemented by a set of instructions, thus, the less operation or instruction required, the better performance will get. Since in integer method kernel, in order to be easily readable, the function has been separated as:

```
"   l_g = l_g + 4*read_imageui(image_in, sampler, (int2)(x,       y));\n"
"   l_g = l_g + 2*read_imageui(image_in, sampler, (int2)(x-reach, y));\n"
"   l_g = l_g + 2*read_imageui(image_in, sampler, (int2)(x+reach, y));\n"
"\n"
"   l_g = l_g + 1*read_imageui(image_in, sampler, (int2)(x-reach, y-reach));\n"
"   l_g = l_g + 2*read_imageui(image_in, sampler, (int2)(x,       y-reach));\n"
"   l_g = l_g + 1*read_imageui(image_in, sampler, (int2)(x+reach, y-reach));\n"
"\n"
"   l_g = l_g + 1*read_imageui(image_in, sampler, (int2)(x-reach, y+reach));\n"
"   l_g = l_g + 2*read_imageui(image_in, sampler, (int2)(x,       y+reach));\n"
"   l_g = l_g + 1*read_imageui(image_in, sampler, (int2)(x+reach, y+reach));\n"
```

   However, it takes some unnecessary operation into account such as assignment and reading previous value. It could be integrated into one sentence.

```
"   l_g = 1*read_imageui(image_in, sampler, (int2)(x-reach, y-reach)) \n "
"           + 2*read_imageui(image_in, sampler, (int2)(x,       y-reach)) \n "
"           + 1*read_imageui(image_in, sampler, (int2)(x+reach, y-reach)) \n "
"           + 2*read_imageui(image_in, sampler, (int2)(x-reach, y)) \n "
"           + 4*read_imageui(image_in, sampler, (int2)(x,       y)) \n "
"           + 2*read_imageui(image_in, sampler, (int2)(x+reach, y)) \n "
"           + 1*read_imageui(image_in, sampler, (int2)(x-reach, y+reach)) \n "
"           + 2*read_imageui(image_in, sampler, (int2)(x,       y+reach)) \n "
"           + 1*read_imageui(image_in, sampler, (int2)(x+reach, y+reach));\n"
```

Type/Memory	Buffer	Image
Integer	85.172ms	165.95ms(142.94ms)
Float	118.32ms	127.517ms

4) For Image method, the original code starts to retrieve upper-left pixel first, however, considering the cache mechanism, it is better to fetch the central pixel first, which will reduce the amount of data transactions.

Type/Memory	Buffer	Image
Integer	85.172ms	142.94ms(143.567ms)
Float	118.32ms	127.517ms(128.01ms)

Conclusion: there are some improvement on Laptop platform, but on target, there is not any difference, it largely depends on the implementation of texture cache mechanism on target. Or the range of adjacent pixels it will cached in one data transaction.

5) Originally, we let OpenCL implementation to determine the size of work group, by setting local_work_size to NULL in clEnqueueNDRangeKernel(), and it could be a potential way to improve the performance by manually set up the group size value.

   After testing, the target could afford a maximum of 128 work-items in one group.

Since, it is better to set group size as a multiple of 64. And since the wrap size is likely to be 32, thus, it is better to set the x-dimension as a multiple of 32. Also the larger group size could perform a better parallel execution. I test the work group size of 32*4 under 4 different situations:

Type/Memory	Buffer	Image
Integer	85.172ms(90.7ms)	142.94ms(143.9ms)
Float	118.32ms(108.25ms)	127.517ms(125.238ms)

Conclusion: There is only some improvement under float condition. The reason is that when launch a workgroup of larger size, it demands a larger resource of local and private memory (or registers), which will reduce the amount of block in active state, which means processing block is waiting or hanged up, less substitutional block there will be. And on the other hand, it is not sure that the target follow the 32 thread wrap scheme as NIVDIA does.