OpenCL BenchMarking

Huang, Ye

Supervisor: Esparza, Jose

Objective Kernel

- The benchmark based on a binomial filter dealing with 1920*1080 one channel picture.

1.f/16	2.f/16	1.f/16
2.f/16	4.f/16	2.f/16
1.f/16	2.f/16	1.f/16

Parallel Mechanism

- Divide 1920*1080 work-items into a certain number of work groups.
- The target has 4 Compute Units, each of them could actually execute one work group once, and also activate certain number of work groups to wait for current processing group to be hanged up.
- Usually, CU scheduler will combine 32 successive work-items as a wrap, assigning single instruction to parallel execute within the wrap.
- If the current processing wrap is waiting or suspended, the scheduler will invoke alternative wrap with independent code to process.

Original Performance

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

- There are still many factors without optimization have impact on the performance, but the data fetch process should be the major concern.
- Because GPU is not designed for memory access, it usually take about 400 times of cycle clock longer to finish data retrieve, comparing with other arithmetic operations.
- The major difference between Buffer and Image is the different data access pattern, will be explained in detail on later slides.
- The difference between Integer and Float is mainly about 2 factors:
  - The Data Size
  - The Division Operation Throughput
- PS: One transaction with larger amount of data is still better than many times of smaller data transaction.

Optimization(1)

- Since in our implementation, the picture only has one channel, we can reduce the repeated divisions in Image method:

```
"   write_imagef(image_out, (int2)(x, y), (float4)(l_g.x/16, l_g.y/16, l_g.z/16, l_g.w/16));\n"
```

```
"   float output = native_divide(l_g.x, 16);\n"
"   write_imagef(image_out, (int2)(x, y), (float4)(output, output, output, output));\n"
```

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.

Optimization(2)

- Intrinsic 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 Intrinsic function 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: Optimization is largely depended on the underlying hardware. Here, division optimization only shows significant improvement on Integer.

Optimization(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 image method kernel, in order to be easily readable, it takes some unnecessary operation into account such as assignment and reading previous value. It could be integrated into one sentence.

```
"   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"
```

```
"   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

Optimization(4)

- 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:

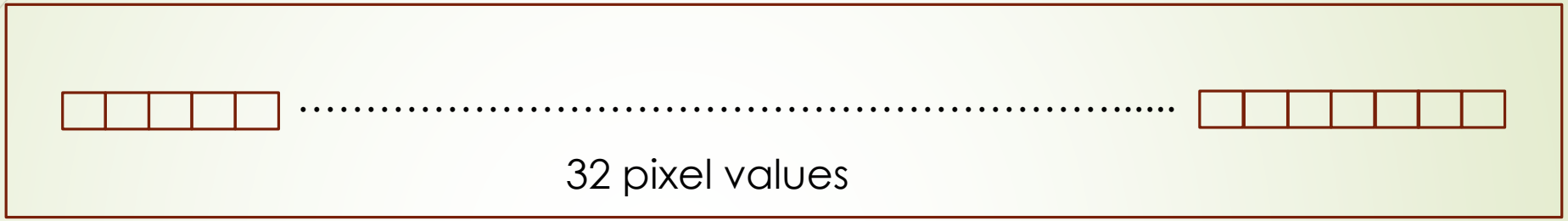
Type/Memory	Buffer	Image
Integer	85.172ms(90.7ms)	142.94ms(143.9ms)
Float	118.32ms(108.25ms)	127.517ms(125.238ms)

Buffer vs. Image

- Buffer data access pattern(for one 32-wrap):

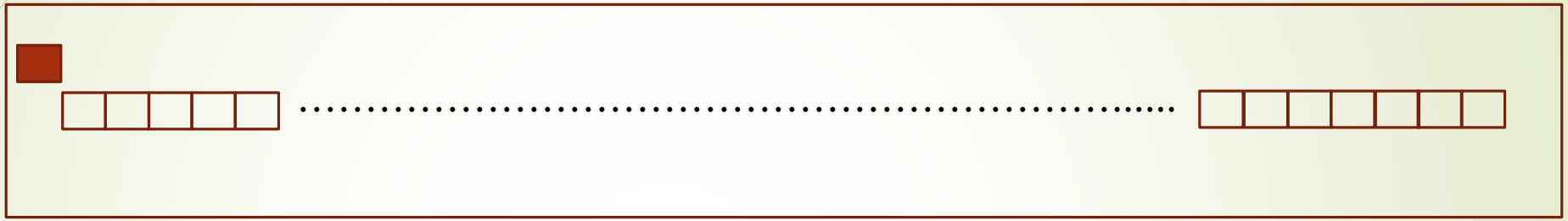
Buffer vs. Image

- Buffer data access pattern(for one 32-wrap):



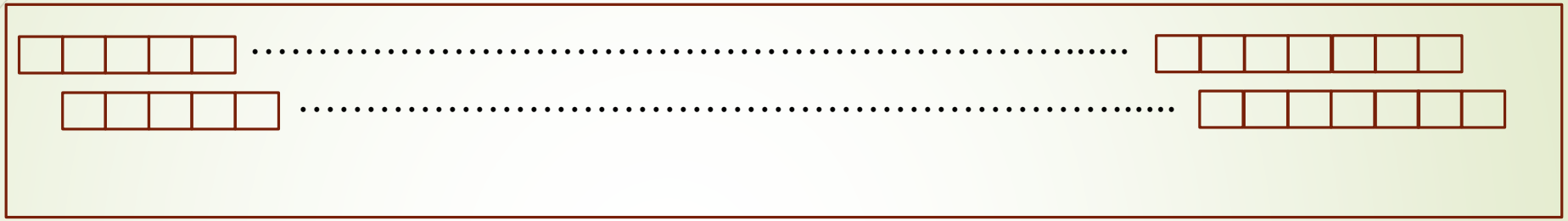
Buffer vs. Image

- Buffer data access pattern(for one 32-wrap):



Buffer vs. Image

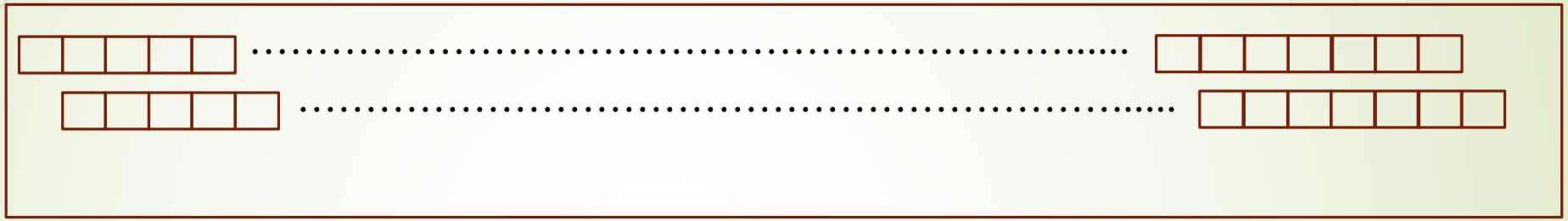
- Buffer data access pattern(for one 32-wrap):



- It only needs about 7 data transaction to finish one wrap

Buffer vs. Image

- Buffer data access pattern(for one 32-wrap):

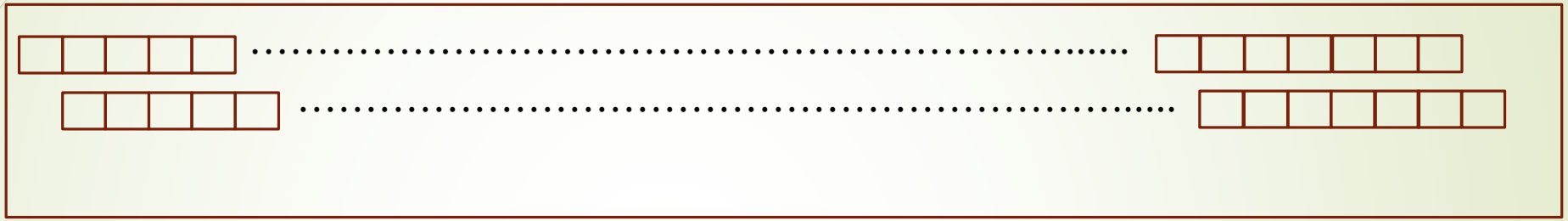


- It only needs about 7 data transaction to finish one wrap

- Image method access data via texture unit with cache, using differenet pattern(contiguous):

Buffer vs. Image

- Buffer data access pattern(for one 32-wrap):

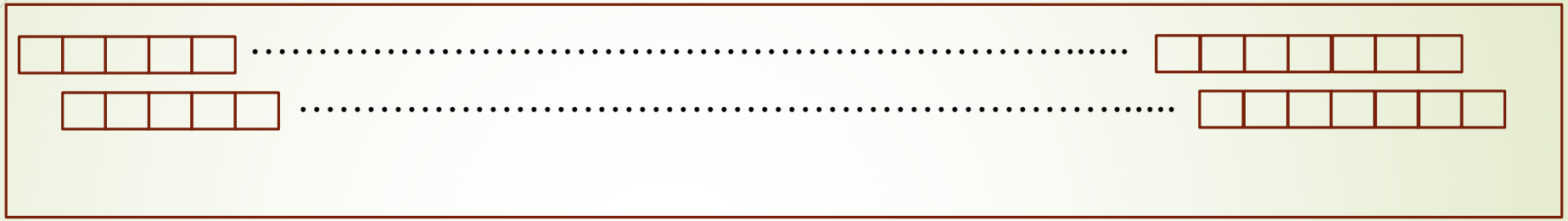


- It only needs about 7 data transaction to finish one wrap

- Image method access data via texture unit with cache, using differenet pattern(contiguous):

Buffer vs. Image

- Buffer data access pattern(for one 32-wrap):

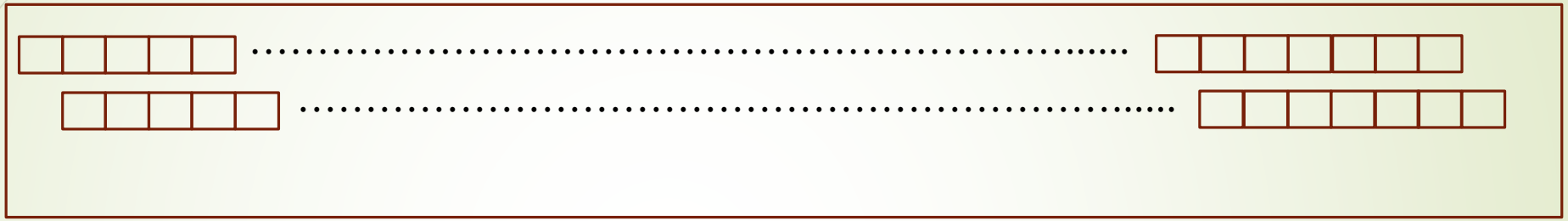


- It only needs about 7 data transaction to finish one wrap

- Image method access data via texture unit with cache, using differenet pattern(contiguous):

Buffer vs. Image

- Buffer data access pattern(for one 32-wrap):

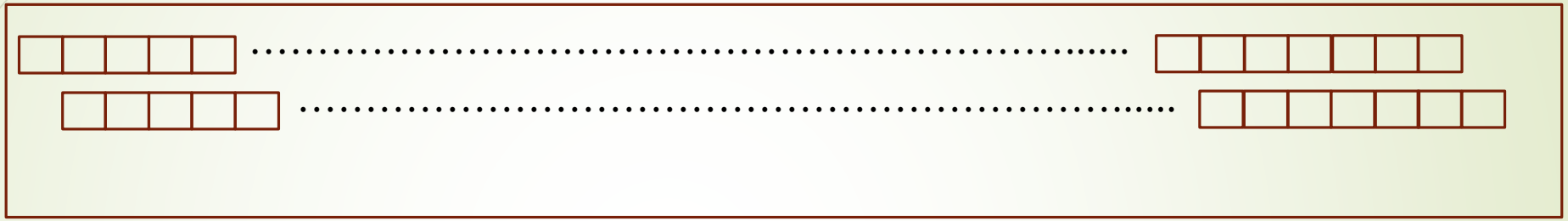


- It only needs about 7 data transaction to finish one wrap

- Image method access data via texture unit with cache, using differenet pattern(contiguous):

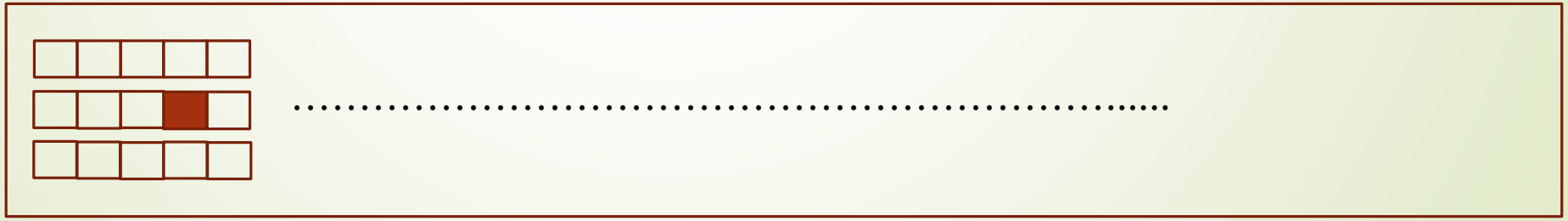
Buffer vs. Image

- Buffer data access pattern(for one 32-wrap):



- It only needs about 7 data transactions to finish one wrap

- Image method access data via texture unit with cache, using differenet pattern(contiguous):



- It will take about 16 data transactions to finish one wrap

Integer vs. Float

Type/Memory	Buffer	Image
Integer	85.172ms	142.94ms
Float	118.32ms	125.238ms

- 1) Buffer Condition:
  - Integer here actually means unsigned char (8 bit) while float takes 32 bits.
- 2) Image Condition:
  - No matter what type the original data is, once data is stored in cl_mem as image object, it will be converted to 32-bit floating-point normalized values in the range [0.0, 1.0] or [-1.0, 1.0].
  - Integer Division takes more time than Float.
  - The read_image() function used in image method is more complex than it looks like. Integer means additional conversion.

BOSCH
Invented for life

Thanks

Huang, Ye

Supervisor: Esparza, Jose