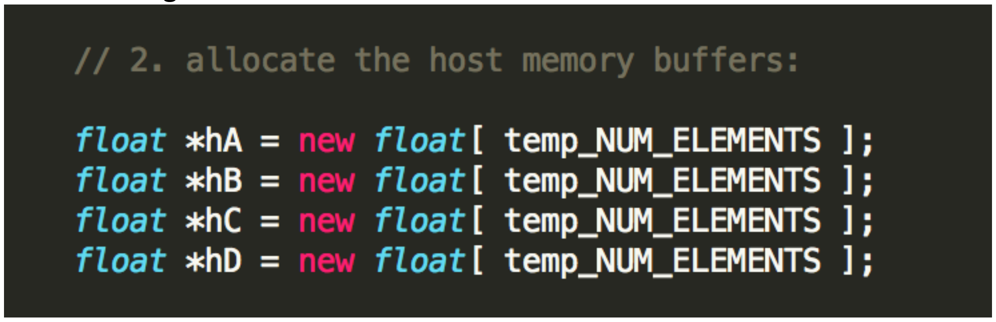
Parallel Programming

CS575

Chao Zhang

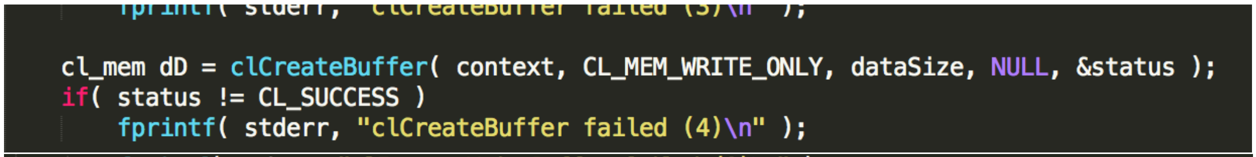
Project #6

1. Source listing



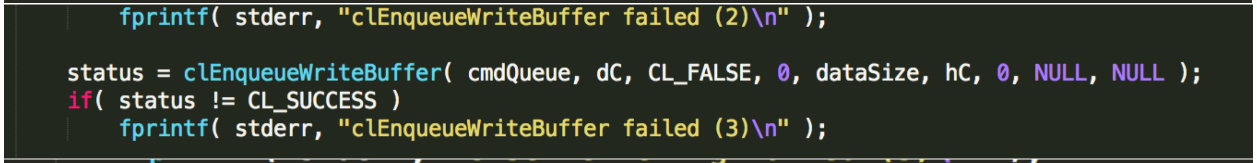
```
// 2. allocate the host memory buffers:

float *hA = new float[ temp_NUM_ELEMENTS ];
float *hB = new float[ temp_NUM_ELEMENTS ];
float *hC = new float[ temp_NUM_ELEMENTS ];
float *hD = new float[ temp_NUM_ELEMENTS ];
```



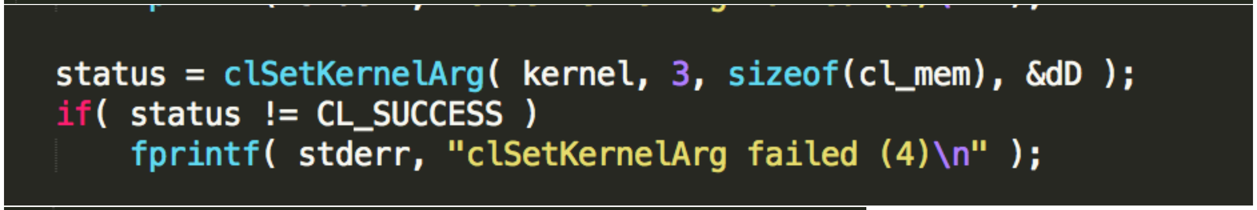
```
    fprintf( stderr, "clCreateBuffer failed (3)\n" );

cl_mem dD = clCreateBuffer( context, CL_MEM_WRITE_ONLY, dataSize, NULL, &status );
if( status != CL_SUCCESS )
    fprintf( stderr, "clCreateBuffer failed (4)\n" );
```

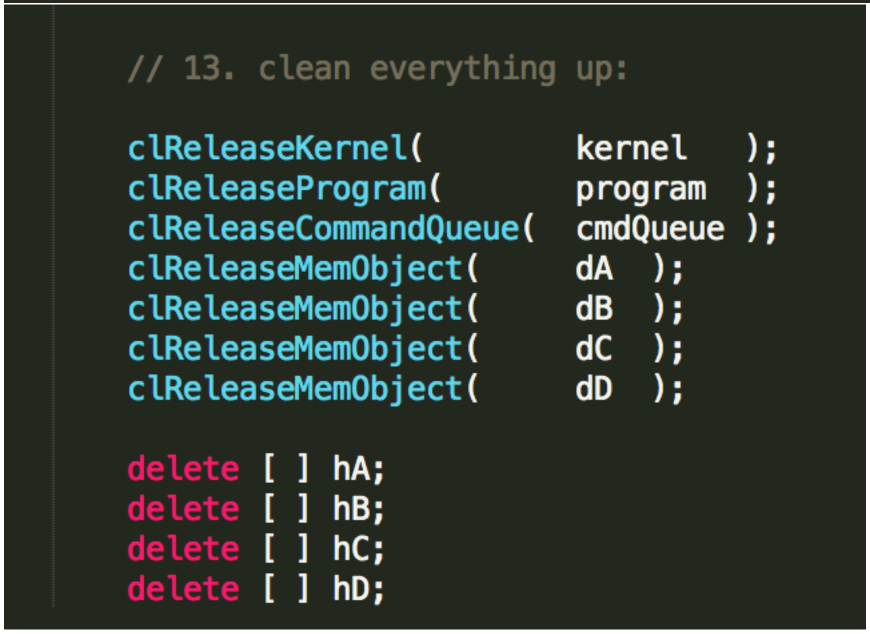


```
    fprintf( stderr, "clEnqueueWriteBuffer failed (2)\n" );

status = clEnqueueWriteBuffer( cmdQueue, dC, CL_FALSE, 0, dataSize, hC, 0, NULL, NULL );
if( status != CL_SUCCESS )
    fprintf( stderr, "clEnqueueWriteBuffer failed (3)\n" );
```



```
status = clSetKernelArg( kernel, 3, sizeof(cl_mem), &dD );
if( status != CL_SUCCESS )
    fprintf( stderr, "clSetKernelArg failed (4)\n" );
```

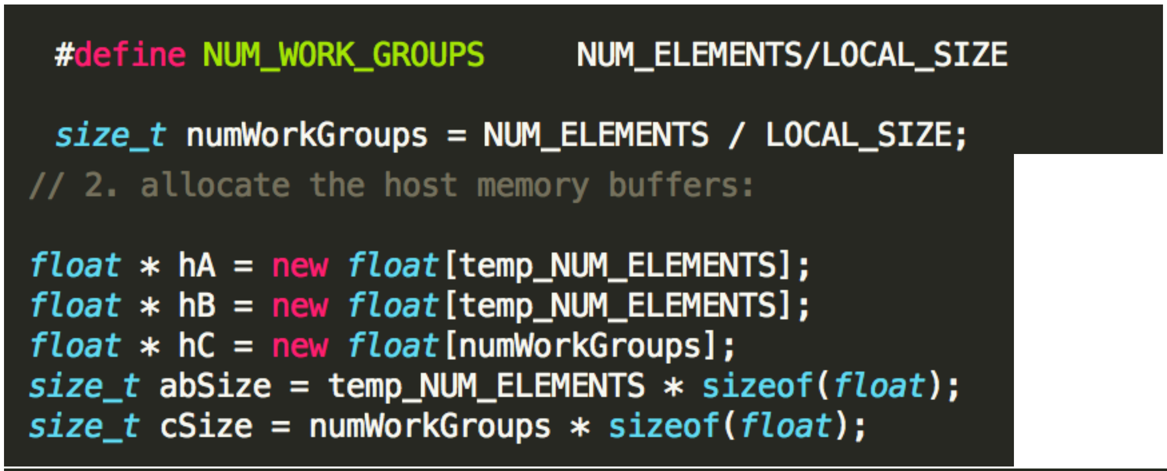


```
// 13. clean everything up:

clReleaseKernel(        kernel   );
clReleaseProgram(       program  );
clReleaseCommandQueue(  cmdQueue );
clReleaseMemObject(     dA   );
clReleaseMemObject(     dB   );
clReleaseMemObject(     dC   );
clReleaseMemObject(     dD   );

delete [ ] hA;
delete [ ] hB;
delete [ ] hC;
delete [ ] hD;
```

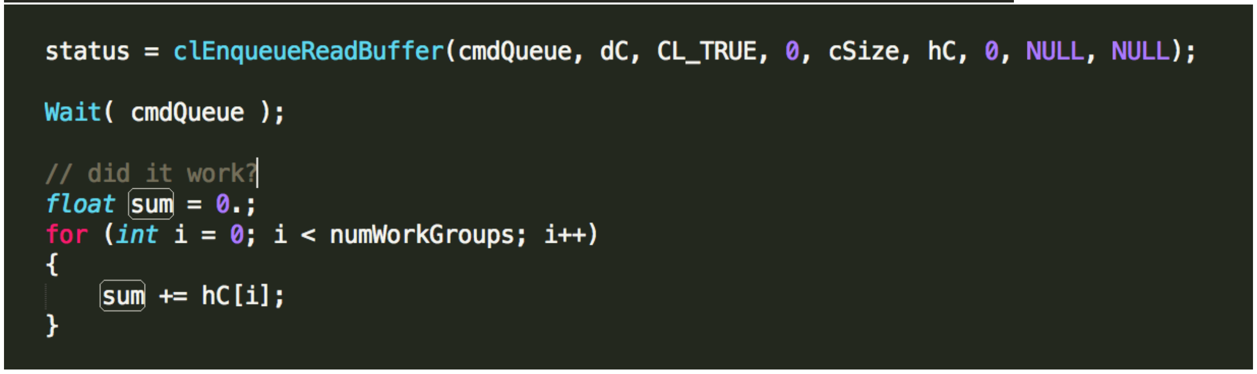
Those are the code added to the first.cpp to make the Multipy-add work. All of those code are used to add the third array.



```
#define NUM_WORK_GROUPS      NUM_ELEMENTS/LOCAL_SIZE

 size_t numWorkGroups = NUM_ELEMENTS / LOCAL_SIZE;
// 2. allocate the host memory buffers:

float * hA = new float[temp_NUM_ELEMENTS];
float * hB = new float[temp_NUM_ELEMENTS];
float * hC = new float[numWorkGroups];
size_t abSize = temp_NUM_ELEMENTS * sizeof(float);
size_t cSize = numWorkGroups * sizeof(float);
```



```
status = clEnqueueReadBuffer(cmdQueue, dC, CL_TRUE, 0, cSize, hC, 0, NULL, NULL);

Wait( cmdQueue );

// did it work?
float sum = 0.;
for (int i = 0; i < numWorkGroups; i++)
{
    sum += hC[i];
}
```

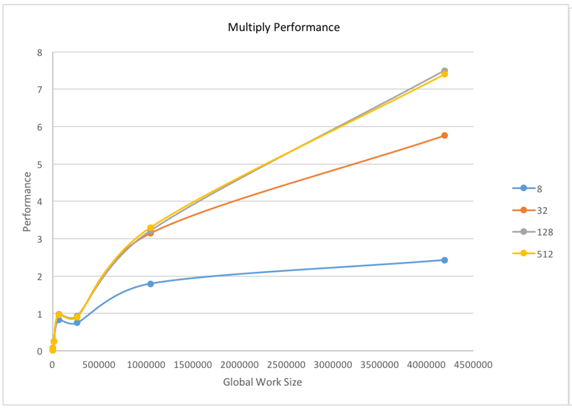
Those are for the reduction, it make the size_t numWorkGroups = NUM_ELEMENTS / LOCAL_SIZE; and use the sum to do the reduction.

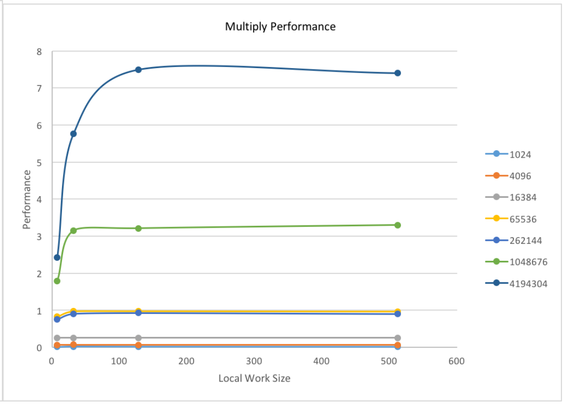
2. Result and analysis

I run this project on the rabbit.

	Multiply						
	1024	4096	16384	65536	262144	1048676	4194304
8	0.017	0.066	0.258	0.827	0.752	1.797	2.426
32	0.021	0.075	0.26	0.971	0.906	3.146	5.759
128	0.02	0.064	0.258	0.976	0.928	3.21	7.491
512	0.019	0.066	0.256	0.968	0.902	3.297	7.398

Multiply Results Table

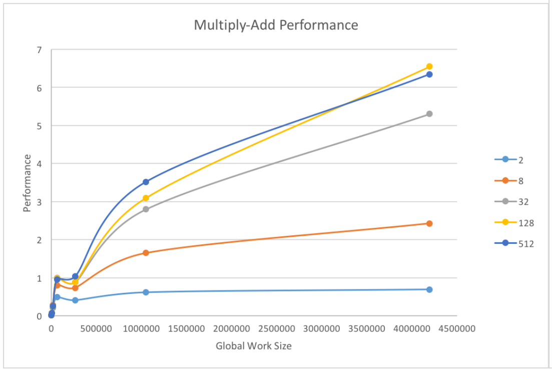


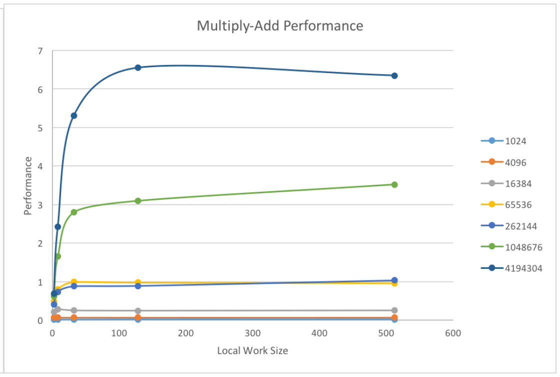


Multiply Results Graphs

Multiply-Add							
	1024	4096	16384	65536	262144	1048676	4194304
2	0.013	0.067	0.215	0.485	0.409	0.612	0.687
8	0.017	0.067	0.285	0.802	0.734	1.646	2.425
32	0.017	0.065	0.251	0.988	0.877	2.795	5.306
128	0.019	0.066	0.245	0.977	0.888	3.095	6.548
512	0.019	0.064	0.252	0.959	1.031	3.519	6.347

Multiply-Add Results Table



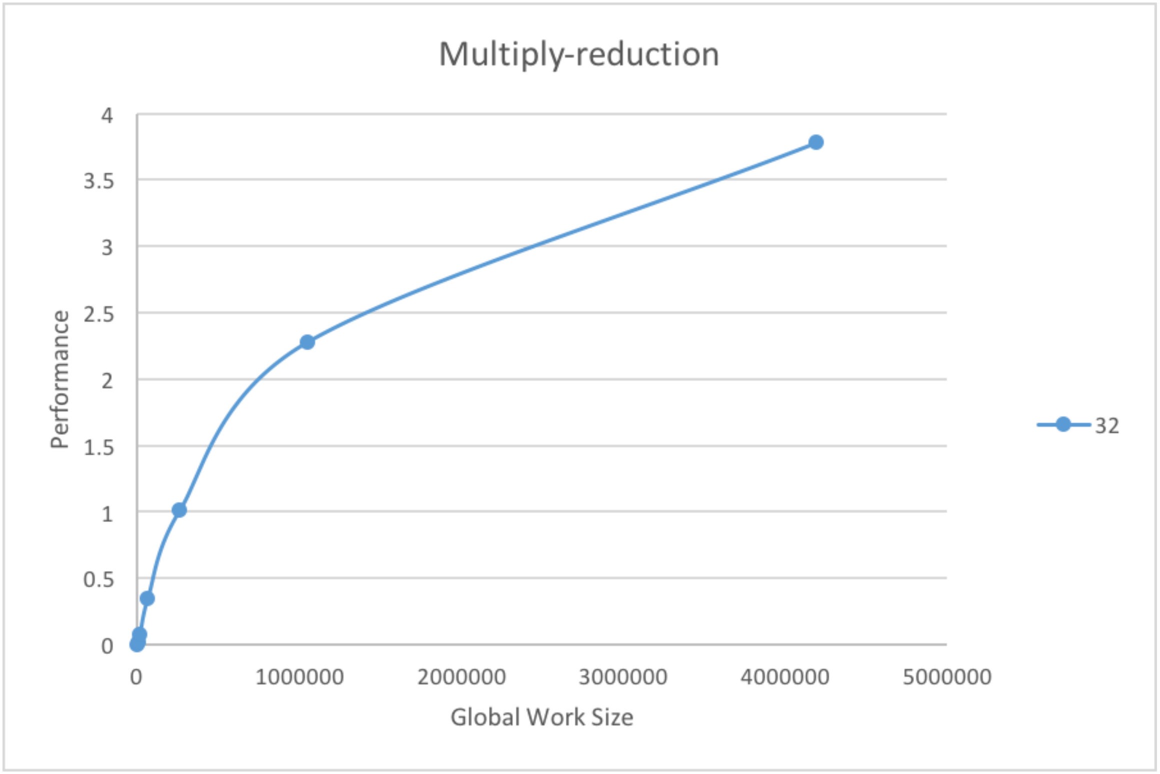


Multiply-Add Results Graphs

We can easily get the conclusion from those four graphs that the performance increased with the increase of the work size. The performance gets better when the work size gets larger. For the size of 128 and 512, I think the performance are almost get the limit so the performance looks the same. The reason why the Multiply and the Multiply-Add has the similar graph because the data is too small to shown the difference. But the Multiply should have a better performance. With the use of GPU, the computing performance can be increased by the control or it can make a good usage of the power of the GPU.

Multiply-reduction							
	1024	4096	16384	65536	262144	1048676	4194304
32	0.005	0.024	0.081	0.353	1.012	2.278	3.782

Multiply-Reduction Table



Multiply-Reduction Graph

Compare the Multiply-Reduction and the other two, I can say that the Multiply-Reduction has a better performance. As we know that the reduction will half the size each time, so the bigger the size is, the better performance we got. So I think the reduction is a good way to deal the multiply calculation in the GPU.