Московский Авиационный Институт
(Государственный Технический Университет)

Факультет прикладной математики и физики.
Кафедра вычислительной математики и программирования.

Лабораторная работа №2
по курсу «Программирование графических процессоров»

Операции над матрицами.

VII семестр.

Студент Баскаков О.А.
Группа 08-406
Преподаватель Семенов С.А.

Москва, 2011.

Постановка задачи

В рамках данной лабораторной работы требуется ознакомиться со средствами написания программ на языке OpenCL, написать и отладить примитивную программу-пример, содержащую базовые принципы программирования графических процессоров с использованием параллельных вычислений.
К выполнению поставленной задачи предъявляются следующие требования:

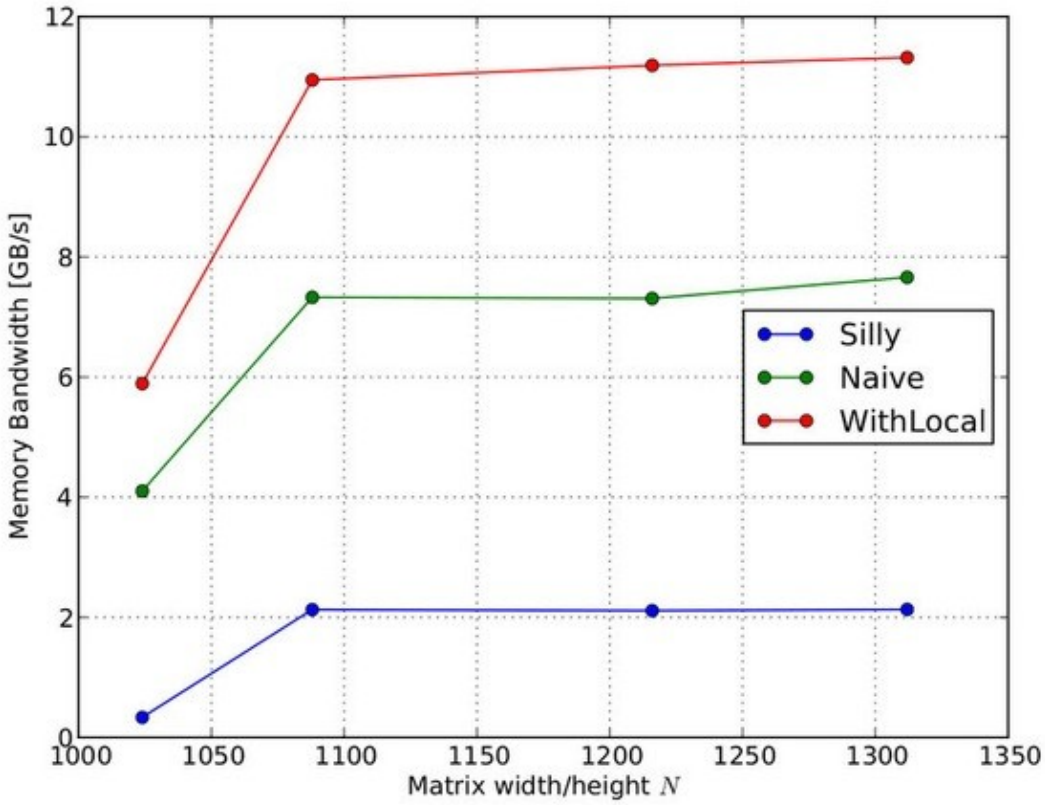
- Программа должна выполняться параллельно
- Программа должна распределять память
- Программа должна использовать векторные операции

Вариант задания:

1. Транспонирование матрицы;
2. Перемножение матриц;

Описание

Для решения поставленной задачи программа выполняет различные операции над данными и сравнивает результаты работы. Используются 2d _kernel.



Код программы на языке Python

1. Native transpose __kernel:

```
class NaiveTranspose:
	def __init__(self, ctx):
		self.kernel = cl.Program(ctx, """
__kernel
void transpose(
  __global float *a_t, __global float *a,
  unsigned a_width, unsigned a_height)
{
  int read_idx = get_global_id(0) + get_global_id(1) * a_width;
  int write_idx = get_global_id(1) + get_global_id(0) * a_height;

  a_t[write_idx] = a[read_idx];
}
		"""% {"block_size": block_size}).build().transpose
	def __call__(self, queue, tgt, src, shape):
		w, h = shape
		assert w % block_size == 0
		assert h % block_size == 0

		return self.kernel(queue, (w, h), (block_size, block_size),
			tgt, src, np.uint32(w), np.uint32(h))
```

1. Transpose with local memory __kernel:

```
class TransposeWithLocal:
	def __init__(self, ctx):
		self.kernel = cl.Program(ctx, """
#define BLOCK_SIZE %(block_size)d
#define A_BLOCK_STRIDE (BLOCK_SIZE * a_width)
#define A_T_BLOCK_STRIDE (BLOCK_SIZE * a_height)

__kernel __attribute__((reqd_work_group_size(BLOCK_SIZE, BLOCK_SIZE, 1)))
void transpose(
  __global float *a_t,
  __global float *a,
  unsigned a_width,
  unsigned a_height,
  __local float *a_local)
{
  int base_idx_a   =
       get_group_id(0) * BLOCK_SIZE +
       get_group_id(1) * A_BLOCK_STRIDE;
  int base_idx_a_t =
       get_group_id(1) * BLOCK_SIZE +
       get_group_id(0) * A_T_BLOCK_STRIDE;

  int glob_idx_a   = base_idx_a + get_local_id(0) + a_width * get_local_id(1);
  int glob_idx_a_t = base_idx_a_t + get_local_id(0) + a_height * get_local_id(1);

  a_local[get_local_id(1)*BLOCK_SIZE+get_local_id(0)] = a[glob_idx_a];

  barrier(CLK_LOCAL_MEM_FENCE);

  a_t[glob_idx_a_t] = a_local[get_local_id(0)*BLOCK_SIZE+get_local_id(1)];
}
		"""% {"block_size": block_size}).build().transpose

	def __call__(self, queue, tgt, src, shape):
		w, h = shape
		assert w % block_size == 0
		assert h % block_size == 0

		return self.kernel(queue, (w, h), (block_size, block_size),
			tgt, src, np.uint32(w), np.uint32(h),
			cl.LocalMemory(4*block_size*(block_size+1)))
```

2. Matrix multiplication:

```
#define BLOCK_SIZE %(block_size)d
#define AS(i, j) As[j + i * BLOCK_SIZE]
#define BS(i, j) Bs[j + i * BLOCK_SIZE]
////////////////////////////////////////////////////////////////////////////////
//! Matrix multiplication on the device: C = A * B
//! uiWA is A's width and uiWB is B's width
////////////////////////////////////////////////////////////////////////////////
__kernel void
matrixMul(__global float* C,
          __global float* A,
          __global float* B,
             int uiWA,
             int uiWB)
{
       __local float As[BLOCK_SIZE*BLOCK_SIZE];
       __local float Bs[BLOCK_SIZE*BLOCK_SIZE];

    // Block index
    int bx = get_group_id(0);
    int by = get_group_id(1);

   // Thread index
    int tx = get_local_id(0);
    int ty = get_local_id(1);

    // Index of the first sub-matrix of A processed by the block
    int aBegin = uiWA * BLOCK_SIZE * by;

   // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + uiWA - 1;

   // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

   // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

   // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * uiWB;

   // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0.0f;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {
        // Load the matrices from device memory
        // to shared memory; each thread loads
        // one element of each matrix
        AS(ty, tx) = A[a + uiWA * ty + tx];
        BS(ty, tx) = B[b + uiWB * ty + tx];
       // Synchronize to make sure the matrices are loaded
        barrier(CLK_LOCAL_MEM_FENCE);

//        #pragma unroll //
/*    This extension extends the OpenCL C language with a hint that
 *   allows loops to be unrolled. This pragma must be used for a loop
 *   and can be used to specify full unrolling or partial unrolling by
 *   a certain amount. This is a hint and the compiler may ignore this
 *   pragma for any reason.
 */
        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
       for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);
```

```
        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
         barrier(CLK_LOCAL_MEM_FENCE);
    }
   // Write the block sub-matrix to device memory;
    // each thread writes one element
    C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = Csub;
}
```

4. Matrix to numpy:

```
def matrix_to_array(A, n, m):
      h = ((n-1) // block_size + 1) * block_size
      w = ((m-1) // block_size + 1) * block_size
      # not fills array by zero
      result = np.empty((h, w), dtype=np.float32)
      for i in range(n):
            for j in range(m):
                  result[i][j] = A[i][j]

      return result

def array_to_matrix(Arr, n, m):
      A = [[0.0] * m for i in range(n)]
      for i in range(n):
            for j in range(m):
                  A[i][j] = Arr[i][j]
      return A
```

5. Transponate:

```
def transponate(A):
      ctx = cl_init()
      queue = cl.CommandQueue(ctx)
      n = len(A)
      m = len(A[0])

      src_array = matrix_to_array(A, n, m)

      print("src_array size: " + str(src_array.shape))

      mf = cl.mem_flags
      a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=src_array)
      a_t_buf = cl.Buffer(ctx, mf.WRITE_ONLY, size=src_array.nbytes)

      method = NaiveTranspose

#     TransposeWithLocal(ctx)(queue, a_t_buf, a_buf, src_array.shape)
      method(ctx)(queue, a_t_buf, a_buf, src_array.shape)

      w, h = src_array.shape
      result = np.empty((h, w), dtype=src_array.dtype)
      cl.enqueue_read_buffer(queue, a_t_buf, result).wait()

      a_buf.release()
      a_t_buf.release()

      print("numpy result array: ")
      print(result)

      err = src_array.T - result
      print("err = ", la.norm(err))

      print("source = ")
      print(A)
      print("result = ")
      print(array_to_matrix(result, m, n))
```

5. Multiplication:

```
def multiply(A, B):
## check sizes here!!!
	ctx, queue = cl_init()
	h1 = len(A)
	h2 = len(B)
	w1 = len(A[0])
	w2 = len(B[0])

	if "NVIDIA" in queue.device.vendor:
		options = "-cl-mad-enable -cl-fast-relaxed-math"
	else:
		options = ""

	a_buf = matrix_to_array(A, h1, w1)
	b_buf = matrix_to_array(B, h2, w2)
	c_buf = np.empty((a_buf.shape[0], b_buf.shape[1])).astype(np.float32)

	print("a_buf")
	print(a_buf)
	print("b_buf")
	print(b_buf)

	kernel_params = {"block_size": block_size}

	prg = cl.Program(ctx, open("mul.cl").read() % kernel_params, )
	prg..build(options=options)

	mf = cl.mem_flags
	d_a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_buf)
	d_b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b_buf)
	d_c_buf = cl.Buffer(ctx, mf.WRITE_ONLY, size=c_buf.nbytes)

	# actual benchmark ----------------------------------------------------------
	t1 = time()

	event = prg.matrixMul(queue, c_buf.shape, (block_size, block_size),
	                      d_c_buf, d_a_buf, d_b_buf,
	                      np.uint32(c_buf.shape[0]), np.uint32(c_buf.shape[1]))

	event.wait()
	gpu_time = (time() - t1)

	cl.enqueue_copy(queue, c_buf, d_c_buf)

	print("c_buf")
	print(c_buf)

	res = array_to_matrix(c_buf, h1, w2, c_buf.shape[1])

	print("result:")
	for row in res:
		print(row)
	print("origin with numpy:")
	print( np.matrix(A) * np.matrix(B))
```

Протокол

```
oleg@spetz:~/CL/lab2$ python lab2_1.py
[[  0.00000000e+00   3.00000000e+00   7.00000000e+00   0.00000000e+00]
 [  1.00000000e+00   4.00000000e+00   8.00000000e+00   0.00000000e+00]
 [  3.36340904e+00   0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  4.56052585e-41   0.00000000e+00   0.00000000e+00   0.00000000e+00]]
('err = ', 0.0)
source = [[0, 1], [3, 4], [7, 8]] , result = [[0.0, 3.0, 7.0], [1.0, 4.0, 8.0]]

oleg@spetz:~/CL/lab2$ python lab2_2.py
a_buf
[[ 1.  2.  3.  0.]
 [ 3.  4.  3.  0.]
 [ 5.  6.  3.  0.]
 [ 0.  0.  0.  0.]]
b_buf
[[ 5.  6.  7.  0.]
 [ 7.  8.  9.  0.]
 [ 7.  8.  9.  0.]
 [ 0.  0.  0.  0.]]
c_buf
[[  40.   46.   52.    0.]
 [  64.   74.   84.    0.]
 [  88.  102.  116.    0.]
 [   0.    0.    0.    0.]]
result:
[[40.0, 46.0, 52.0] , [64.0, 74.0, 84.0], [88.0, 102.0, 116.0] ]
origin with numpy:
[[ 40  46  52] , [ 64  74  84], [ 88 102 116]]
```

Вывод

Снова была показана важность оптимизации программ на примере транспонирования матрицы. Правильный подбор block_size дал прирост производительности в 2 раза, а использование локальной памяти – в 6 раз!

Задача транспонирования матрицы демонстрирует ограничения пропускной способности шины данных видеокарты. Задача перемножения – классический алгоритм, поддающийся распараллеливанию. Используя его, например, можно решать СЛАУ методом простых итерраций.