Московский Авиационный Институт
(Государственный Технический Университет)

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

Лабораторная работа №9

по курсу «Программирование графических процессоров»

Численные методы в алгоритмах линейной алгебры.

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. Построение LUP разложения матрицы;
2. Решение СЛАУ с использованием LUP;

Описание

Для решения поставленной задачи программа:

1) Определяет наличие необходимого устройства (видеокарты);
2) Выделяет необходимое количество памяти;
3) Определяет размерность блоков;
4) Определяет число необходимых блоков;
5) Запускает ядро, передавая ему число блоков и число нитей в каждом блоке;
6) В рамках ядра производит необходимые вычисления параллельно;
7) Копирует результаты работы программы из памяти видеокарты;
8) Высвобождает память видеокарты;
9) Выводит на экран результаты работы программы;

Код программы на языке Python

0. Host code:

```
def LUP(A):
	global ctx
	global queue
	global prg

	ctx, queue = cl_init()
	kernel_params = {"block_size": block_size}
	prg = cl.Program(ctx, open("my_LUP.cl").read() % kernel_params, ).build()

	n = len(A)
	m = len(A[0])
	# correct size
	rem = m % block_size
	if rem:
		A = [row + [0.0]*(block_size - rem) for row in A]
		m = len(A[0])

	a_buf = np.array(A).astype(np.float32)
	L_buf = np.empty_like(a_buf).astype(np.float32)

	print("a_buf input")
	print(a_buf)

	mf = cl.mem_flags
	d_a_buf = cl.Buffer(ctx, mf.READ_WRITE | mf.COPY_HOST_PTR, hostbuf=a_buf)
	d_L_buf = cl.Buffer(ctx, mf.READ_WRITE | mf.COPY_HOST_PTR,
			    size=L_buf.nbytes, hostbuf=L_buf)

	p_act = []
	for k in range(n-1):
		s = max_col(d_a_buf, m, k)
		if s != k:
			p_act.append((s,k))
			swap_row(d_a_buf, m, s, k)
			swap_row(d_L_buf, m, s, k)
			swap_col(d_L_buf, m, s, k)
			print('swap', k, s)

		print("one_step", k)
		LU_one_step(d_a_buf, d_L_buf, a_buf.shape, n, k)

		cl.enqueue_copy(queue, a_buf, d_a_buf)
		cl.enqueue_copy(queue, L_buf, d_L_buf)

	# construct permute vector
	p = list(range(n))
	for (s,k) in p_act:
		p[s],p[k] = p[k],p[s]

	cl.enqueue_copy(queue, a_buf, d_a_buf)
	cl.enqueue_copy(queue, L_buf, d_L_buf)

	print("a_buf")
	print(a_buf)
	print("L_buf")
	print(L_buf)
	print("p = " + str(p))

	if rem:
		m -= block_size - rem

	res = [[ round(a+l, 5) for (a,l) in zip(row_a[:m], row_l[:m])]
		for (row_a, row_l) in zip(a_buf, L_buf)]

	return res
```

1. Эталонный код прототипа на Python3:

```
	def build_LU(self):
		n = self.n
		A = Matrix()
		A.M = deepcopy(self.M)
		A.n = self.n
		A.m = self.m
		self.p = [x for x in range(n)]
		self.p_count = 0

		U = Matrix("", n, n)
		L = Matrix("", n, n)

		for k in range(n-1):
			# Find max
			s = k
			for j in range(k+1, n):
				if (abs(A[s][k])< abs(A[j][k])) : s = j
			A.swap_rows(k, s)
			L.swap_rows(k, s)
			L.swap_cols(k, s)

			# construct vector p
			z = self.p[k]   #equal k
			self.p[k] = self.p[s]
			self.p[s] = z
			if (k!=s): self.p_count += 1

			for i in range(k+1, n): #col number
				koef = A[i][k]/A[k][k]
				A.M[i] = [(xx-xo*koef) for (xx,xo) in zip(A.M[i],A.M[k])]
				L[i][k] = koef
		self.L = L
		self.U = A
```

2. Решатель СЛАУ:

```
def shift_b(self, b):
	v = [0]*self.n
	for i in range(self.n):
		v[i] = b[self.p[i]]
	return v

def solve(self,b):
	n = len(b)
	z = [0]*n
	for i in range(1,n):
		z[i] = b[i]
		for j in range(i):
			z[i] -= self.L[i][j]*z[j]
	x = [0]*n
	i = n-1
	for i in range(n-1,-1,-1):
		x[i] = z[i]
		for j in range(i+1,n):
			x[i] -= self.U[i][j]*x[j]
		x[i] /= self.U[i][i]
	return x
```

3. OpenCL kernels:

```
__kernel void
one_step(__global float* A,
         __global float* L,
         int n, int k)
{
       int i = get_global_id(1) + k+1;
       int j = get_global_id(0) + k;
       // exit 2d condition like for(;;)
       if (j >= n || i >= n) return;

       float koef = A[n*i + k] / A[n*k + k];

       L[n*i + k] = koef;
       A[i*n + j] = A[i*n + j] - koef * A[n*k + j];
}

__kernel void
max_col(__global float* A,
       __global float* B,
       int m, int col, int row)
{
       __local float a_local[BLOCK_SIZE];
       const int k = BLOCK_SIZE;

       int i_g = get_global_id(0);
       int i_l = get_local_id(0);
       if (i_g >= m - row) return;

       B[i_g + 0] = fabs(A[m*(i_g+row+0) + col]);
       B[i_g + 1] = fabs(A[m*(i_g+row+1) + col]);
       B[i_g + 2] = fabs(A[m*(i_g+row+2) + col]);
       B[i_g + 3] = fabs(A[m*(i_g+row+3) + col]);
       barrier(CLK_GLOBAL_MEM_FENCE);

       while (m > 0 && i_g < m - row) {
              a_local[i_l] = max(max(B[i_g/4+0],B[i_g/4+1]), max(B[i_g/4+2], B[i_g/4+3]));
              barrier(CLK_LOCAL_MEM_FENCE);
              B[i_g] = a_local[i_l];
              barrier(CLK_GLOBAL_MEM_FENCE);
              m /= k;
       }
}

__kernel void
index_col(__global float* A,
          __global float* B,
          __global int* RES,
         int m, int i, int row)
{
       float val = B[0];
       int j = get_global_id(0) + row;
       if (j >= m) return;

       if (fabs(A[m*j + i]) == val)
              RES[0] = j;
}

__kernel void
swap_row(__global float* A,
         int n, int i1, int i2)
{
       int j = get_global_id(0);
       if (j >= n) return;
       float z1 = A[n*i1 + j];
       float z2 = A[n*i2 + j];
       A[n*i2 + j] = z1;
       A[n*i1 + j] = z2;
}
```

```
 __kernel void
swap_col(__global float* A,
         int n, int j1, int j2)
{
       int i = get_global_id(0);
       if (i >= n) return;

       float z1 = A[n*i + j1];
       float z2 = A[n*i + j2];
       A[n*i + j2] = z1;
       A[n*i + j1] = z2;
}
```

Протокол

```
oleg@spetz:~/CL/lab4$ python lab_LUP.py
a_buf input
[[ 2.  7. -8.  6.]
 [ 4.  4.  0. -7.]
 [-1. -3.  6.  3.]
 [ 9. -7. -2. -8.]]
(4, 4)
a_buf
[[  9.00000000e+00  -7.00000000e+00  -2.00000000e+00  -8.00000000e+00]
 [  0.00000000e+00   8.55555534e+00  -7.55555534e+00   7.77777767e+00]
 [  0.00000000e+00  -4.76837158e-07   7.16883135e+00  -9.90909100e+00]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   8.92028999e+00]]
L_buf
[[  0.00000000e+00   8.98127835e-38   0.00000000e+00   8.99919703e-38]
 [  2.22222224e-01   1.27442246e-23   4.55982520e-41   7.88566474e-38]
 [  4.44444448e-01   8.31168890e-01   0.00000000e+00   1.20928694e-38]
 [ -1.11111112e-01  -4.41558450e-01   3.40579689e-01   4.62428493e-44]]
p = [3, 0, 1, 2]
result:
[[9.0,    -7.0,  -2.0,  -8.0],
 [0.22,   8.56, -7.56,  7.78],
 [0.44,   0.83,  7.17, -9.91],
 [-0.11, -0.44,  0.34,  8.92]]

oleg@spetz:~/CL/lab2$ python lab2_2.py
oleg@spetz:~/CL/lab4$ python3 1.1_LUP.py
Input matrix:
Matrix 4x4 :
       1.00   2.00   3.00   4.00
       3.00   4.00   3.00   4.00
       5.00   6.00   8.00   9.00
       5.00   6.00   3.00   7.00
-----
[2, 0, 3, 1]
U:
Matrix 4x4 :
       5.00   6.00   8.00   9.00
       0.00   0.80   1.40   2.20
       0.00   0.00   -5.00  -2.00
       0.00   0.00   0.00   -1.50
-----
L:
Matrix 4x4 :
       1.00   0.00   0.00   0.00
       0.20   1.00   0.00   0.00
       1.00   0.00   1.00   0.00
       0.60   0.50   0.50   1.00
-----
L*U:
Matrix 4x4 :
       5.00   6.00   8.00   9.00
       1.00   2.00   3.00   4.00
       5.00   6.00   3.00   7.00
       3.00   4.00   3.00   4.00
```

```
oleg@spetz:~/CL/lab4$ python lab_LUP.py
p = [3, 0, 1, 2]
result:
[[9.0, -7.0, -2.0, -8.0],
 [0.22, 8.56, -7.56, 7.78],
 [0.44, 0.83, 7.17, -9.91],
 [-0.11, -0.44, 0.34, 8.92]]

oleg@spetz:~/CL/lab4$ python3 1.1_LUP.py
Input Matrix 4x4 :
       2.00   7.00   -8.00  6.00
       4.00   4.00   0.00   -7.00
       -1.00  -3.00  6.00   3.00
       9.00   -7.00  -2.00  -8.00
-----
[3, 0, 1, 2]
U:
Matrix 4x4 :
       9.00   -7.00  -2.00  -8.00
       0.00   8.56   -7.56  7.78
       0.00   0.00   7.17   -9.91
       0.00   0.00   0.00   8.92
L:
Matrix 4x4 :
       1.00   0.00   0.00   0.00
       0.22   1.00   0.00   0.00
       0.44   0.83   1.00   0.00
       -0.11  -0.44  0.34   1.00

Проверим решатель СЛАУ:
Vector b:
[-39.0, 41.0, 4.0, 113.0]
Solve:
x =  [8.0, -3.0, 2.0, -3.0]
Check A*x = b:
-39.00  41.00  4.00  113.00
Determinant: -4924.00
A^-1:
Matrix 4x4 :
       0.10   0.07   0.16   0.07
       0.04   0.11   0.03   -0.05
       -0.00  0.09   0.15   -0.02
       0.08   -0.04  0.11   0.01
A*A^-1:
Matrix 4x4 :
       1.00   -0.00  0.00   0.00
       0.00   1.00   0.00   0.00
       0.00   0.00   1.00   0.00
       -0.00  0.00   0.00   1.00
```

Вывод

Алгоритм LU разложения позволяет эффективно находить решения СЛАУ, а также вычислять обратную матрицу и детерминант. Алгоритм особенно полезен в случае, когда необходимо найти решение нескольких СЛАУ с одинаковыми коэффициентами, и различными свободными членами. LUP-разложение используется для вычисления обратной матрицы по компактной схеме, решая СЛАУ. По сравнению с алгоритмом LU-разложения алгоритм LUP-разложения может обрабатывать любые невырожденные матрицы и при этом обладает более высокой численной устойчивостью.

В процессе решения была выявлена сложность реализации reduce для поиска maximum в векторе с использованием GPU. А также невозможность распараллеливания (ускорение незначительно) прямого и обратного хода метода Гаусса по готовому LU разложению.

Замечание: реализация операции reduce требует глобальной синхронизации потоков. Для устранения коллизий на больших объемах данных(больше кол-ва вычислителей) желательно использовать 2 буфера глобальной памяти.