CA Project: Data level parallelism

**Abstract:**

The idea is to implement matrix and vector multiplication algorithm using OpenCL programming concept and explore Data level Parallelism. It is a basic component of image processing and we implemented basic of filtration which is matrix to matrix and matrix to vector multiplication.

**Description:**

OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. OpenCL specifies programming languages (based on C99 and C++11) for programming these devices and application programming interfaces (APIs) to control the platform and execute programs on the compute devices. OpenCL provides a standard interface for parallel computing using task- and data-based parallelism.

**Compute Device**  
Compute devices are the target of computation to be off-loaded from the host CPU. Command-queues are created in OpenCL applications and are tied to a specific compute device. Internally, a compute device is a collection of computing units. OpenCL compute devices typically correspond to a GPU, a multi-core CPU, or multi-core DSP.

**Compute Unit**  
A compute device contains one or more compute units. For multi-core devices, a compute unit often corresponds to one of the cores. A work-group executes on a single compute unit and multiple work-groups can execute concurrently on multiple compute units within a device. A compute unit will have local memory that is accessible only by the compute unit.

**Command Queue**  
An object created by OpenCL APIs in the host application. A command-queue is created for a specific device in a context. Command queues hold commands that will be executed on that specific device. Commands to a commandqueue are queued in-order but may be executed in-order or out-of-order depending on the attributes specified during the command queue’s creation. A compute device may have many command queues associated with it, but a command queue will only associate with one compute device.

**Kernel**  
A kernel is a function declared in an OpenCL C program and executed on an Compute Device. A kernel is identified by the \_\_kernel or kernel qualifier. Kernels are enqueued to compute devices through command queues.

**Work-item**  
When a kernel is enqueued to a command queue, the enqueue command specifies the number of work-items to be completed. For enqueueNDRangeKernel, the number of work-items is explicitly specified by the global size argument. For enqueueTask, the number of work-items is implicitly specified as 1.  
  
One of a collection of parallel executions of a kernel invoked on a device by a command. A work-item is executed by one or more processing elements as part of a work-group executing on a compute unit. A work-item is distinguished from other executions within the collection by its global ID and local ID.  
  
**Work-group**  
A collection of related work-items that execute on a single compute unit. The work-items in the group execute the same kernel and share local memory.

**Global ID**  
A global ID is used to uniquely identify a work-item and is derived from the number of global work-items specified when executing a kernel. The global ID is an N-dimensional value that starts at 0 in all dimensions.

**Local ID**  
A local ID specifies a unique work-item ID within a given work-group that is executing a kernel. The local ID is an N-dimensional value that starts at 0 in all dimensions.

**Problem Statement:**

The time complexity of matrix-matrix multiplication is O(n^3) and matrix-vector multiplication is O(kn^2). We applied data level parallelism using OpenCL to reduce execution time.

.

**Conclusion:**

Hence we conclude that GPU is much better than CPU when we have a large dataset. There is a huge performance difference in the Graph on large dataset but for small datasets (<1M) sequential programming provides almost the same results and is easy to implement. Whereas parallel programming using GPU proves to be very complex and hard to implement. Hence concluding GPU is better for large dataset while CPU is better for small dataset.

**Code Snippets:**

void PerformCalculationOnDevice(cl::Device device) {  
 vector<cl::Device> contextDevices;  
 contextDevices.push\_back(device);  
 cl::Context context(contextDevices);  
 cl::CommandQueue queue(context, device);  
 std::fill\_n(result, row, 0);  
 cl::Buffer cl\_matrix\_A = cl::Buffer(context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, row \* col \* sizeof(int), matrix\_A);  
 cl::Buffer cl\_vector\_B = cl::Buffer(context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, col \* sizeof(int), matrix\_B);  
 cl::Buffer cl\_result\_vector = cl::Buffer(context, CL\_MEM\_WRITE\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, col \* sizeof(int), result);  
   
 std::ifstream sourceFile("kernal.cl");  
 std::string sourceCode(std::istreambuf\_iterator<char>(sourceFile), (std::istreambuf\_iterator<char>()));  
 cl::Program::Sources source(1, std::make\_pair(sourceCode.c\_str(), sourceCode.length() + 1));  
 cl::Program program = cl::Program(context, source);  
 program.build(contextDevices);  
 cl::Kernel kernel(program, "matrixVectorMul");  
 int iArg = 0;  
 kernel.setArg(iArg++, cl\_result\_vector);  
 kernel.setArg(iArg++, cl\_matrix\_A);  
 kernel.setArg(iArg++, cl\_vector\_B);  
 kernel.setArg(iArg++, col);  
 queue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(row), cl::NDRange(row));  
 queue.finish();  
  
 queue.enqueueReadBuffer(cl\_result\_vector, CL\_TRUE, 0, col \* sizeof(int), result);  
}

//intel gpu  
long LoadOpenCLKernel(char const\* path, char \*\*buf)  
{  
 FILE \*fp;  
 size\_t fsize;  
 long off\_end;  
 int rc;  
 /\* Open the file \*/  
 fp = fopen(path, "r");  
 if (NULL == fp) {  
 return -1L;  
 }  
 /\* Seek to \*/  
 rc = fseek(fp, 0L, SEEK\_END);  
 if (0 != rc) {  
 return -1L;  
 }  
 /\* Byte offset to the end of the file (size) \*/  
 if (0 > (off\_end = ftell(fp))) { //points end of the file  
 return -1L;  
 }  
 fsize = (size\_t)off\_end;  
 /\* Allocate a buffer to hold the whole file \*/  
 \*buf = (char \*)malloc(fsize + 1);  
 if (NULL == \*buf) {  
 return -1L;  
 }

/\* Rewind file pointer to start of file \*/  
 rewind(fp);  
 /\* Slurp file into buffer \*/  
 if (fsize != fread(\*buf, 1, fsize, fp)) {  
 free(\*buf);  
 return -1L;  
 }  
 /\* Close the file \*/  
 if (EOF == fclose(fp)) {  
 free(\*buf);  
 return -1L;  
 }  
 /\* Make sure the buffer is NUL-terminated, just in case \*/  
 (\*buf)[fsize] = '\0';  
 /\* Return the file size \*/  
 return (long)fsize;  
}

Void main()

{   
 cl\_uint dev\_cnt = 0; //clockFrequency  
 clGetPlatformIDs(0, 0, &dev\_cnt);  
 cl\_platform\_id platform\_ids[2];  
 clGetPlatformIDs(dev\_cnt, platform\_ids, NULL);  
 // Connect to a compute device  
 int gpu = 1;  
 err = clGetDeviceIDs(platform\_ids[0], CL\_DEVICE\_TYPE\_GPU , 1, &device\_id, NULL);  
  
 // Create a compute context   
 context = clCreateContext(0, 1, &device\_id, NULL, NULL, &err);  
 // Create a command commands  
 commands = clCreateCommandQueue(context, device\_id, 0, &err);  
 // Create the compute program from the source file  
 char \*KernelSource;  
 long lFileSize;  
 lFileSize = LoadOpenCLKernel("matrixmul\_kernel.cl", &KernelSource);  
 program = clCreateProgramWithSource(context, 1, (const char \*\*)& KernelSource, NULL, &err);;  
 // Build the program executable  
 err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);  
 // Create the compute kernel in the program we wish to run  
 kernel = clCreateKernel(program, "matrixMul", &err);  
  
 // Create the input and output arrays in device memory for our calculation  
 d\_C = clCreateBuffer(context, CL\_MEM\_READ\_WRITE, mem\_size\_A, NULL, &err);  
 d\_A = clCreateBuffer(context, CL\_MEM\_READ\_WRITE | CL\_MEM\_COPY\_HOST\_PTR, mem\_size\_A, matrix\_A, &err);  
 d\_B = clCreateBuffer(context, CL\_MEM\_READ\_WRITE | CL\_MEM\_COPY\_HOST\_PTR, mem\_size\_B, matrix\_B, &err);  
 //Launch OpenCL kernel  
 size\_t localWorkSize[2], globalWorkSize[2];  
 int wA = row;  
 int wC = row;  
 err = clSetKernelArg(kernel, 0, sizeof(cl\_mem), (void \*)&d\_C);  
 err |= clSetKernelArg(kernel, 1, sizeof(cl\_mem), (void \*)&d\_A);  
 err |= clSetKernelArg(kernel, 2, sizeof(cl\_mem), (void \*)&d\_B);  
 err |= clSetKernelArg(kernel, 3, sizeof(int), (void \*)&wA);  
 err |= clSetKernelArg(kernel, 4, sizeof(int), (void \*)&wC);  
 localWorkSize[0] = 16;  
 localWorkSize[1] = 16;  
 globalWorkSize[0] = 1024;  
 globalWorkSize[1] = 1024;  
 err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL); //executes kernel  
 //Retrieve result from device  
 err = clEnqueueReadBuffer(commands, d\_C, CL\_TRUE, 0, mem\_size\_C, result, 0, NULL, NULL);  
 //AMD START  
 start\_t = clock();  
 std::vector<cl::Platform> platforms;  
 cl::Platform::get(&platforms);  
 std::vector<cl::Device> devices;  
 for (int iPlatform = 0; iPlatform < platforms.size(); iPlatform++) {  
 platforms[iPlatform].getDevices(CL\_DEVICE\_TYPE\_GPU, &devices);  
 for (int iDevice = 0; iDevice < devices.size(); iDevice++) {  
  
 PerformCalculationOnDevice(devices[iDevice]);  
  
 }  
 }

}

**References:**

[**https://stackoverflow.com/questions/15597299/matrix-vector-multiplications-using-opencl**](https://stackoverflow.com/questions/15597299/matrix-vector-multiplications-using-opencl)

[**http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmopencl**](http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmopencl)

[**https://www.youtube.com/watch?v=YU\_pRT-Be0c&list=PLzy5q1NUJKCJocUKsRxZ0IPz29p38xeM-**](https://www.youtube.com/watch?v=YU_pRT-Be0c&list=PLzy5q1NUJKCJocUKsRxZ0IPz29p38xeM-)