Hardware acceleration using OpenCL applications on FPGA

Abstract— **With increase in demand for high performance computing, research and development in the field of parallel computing and acceleration have increased a lot to satisfy the overgrowing demand for high speed of operation and performance. Different devices and architecture offer different performance speed up and benefits for different applications. In this paper, we choose to demonstrate the ability of using FPGA platform for hardware acceleration using OpenCL by exploiting the inherent flexibility and massive parallel computation capabilities of a field programming gate array. A sample bench mark of matrix multiplication is chosen as matrix operations are one of the very common and time consuming tasks that have to be performed for many applications. A 1000x1000 matrix multiplication is considered and performed on different devices like Intel Xeon CPU and Altera FPGA. Performance and speedup comparison is done between single core and multicore application ran on the Intel Xeon processors using pThreads and using OpenCL on the FPGA. Observations, output results and comparison graphs have also been presented in the paper.**

Keywords— OpenCL, FPGA, pThreads, Matrix Multiplication

I. INTRODUCTION

An increasing number of application areas today are going for high performance computing (HPC) solutions for their processing needs. Conventional processors have given way to various accelerator technologies to meet the computational demands. These range from Graphic Processing Units (GPU), Field Programmable Gate Arrays (FPGA), heterogeneous multicore processors. Different classes of applications employ different targets best suited for the problem under consideration. Reconfigurable systems like FPGAs provide promising opportunities for acceleration in many fields due to their inherent flexibility and massive parallel computation capabilities.

FPGAs can potentially deliver tremendous acceleration in high-performance server and embedded computing applications. Enormous acceleration can be delivered by FPGAs owing to the flexibility and parallelism provided by the fine grained architecture. However, being able to fully harness this potential presents challenges in terms of programmability. Implementation of applications on FPGAs involve cumbersome RTL programming and manual optimizations. To reduce this effort OpenCL has been taken under consideration.

The OpenCL programming model provides a standard framework that enhances portability across devices essentially rendering it platform independent. It enables the developer to focus on design of the application while abstracting the fine details of implementation. Being able to develop applications in OpenCL for FPGA platforms would result in faster implementation times and time-to-market.

This paper demonstrates a design of a hybrid system with CPU + FPGA. The design is a mix of software host application and hardware designs using OpenCL API.

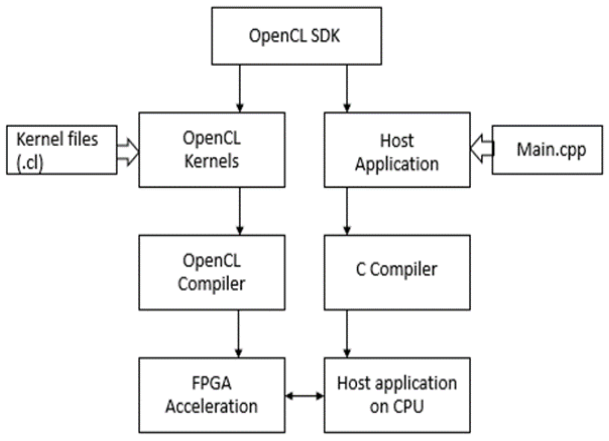


Fig 1: Methodology flow

II. Description

In this hardware acceleration design, we implemented an extensive matrix multiplication operation of two 1000x1000 matrices. The design was compared with the CPU performance for single core, dual core, quad core, octal core, 16 cores and 32 cores to show the acceleration that was gained on a FPGA device.

Altera FPGA was used with Altera OpenCL SDK to design hardware configuration files. The FPGA was plugged in the PCIe slot. The task included main host application coding in C, OpenCL coding as well as FPGA algorithm design. Using FPGA acceleration we tried to speed up the execution of the algorithm, this speed up was compared to the CPU performance giving us a solid result of how much acceleration is achieved. Results of all the designs and the comparison are attached in the later section of this document.

III. pThreads Overview

POSIX or Pthreads is a standardized C API for multithreading applications. In shared memory multiprocessor architectures, threads can be used to implement parallelism. Each thread is an independent stream of instructions that share the process resources with other threads. It has its own independent flow of control as long as its parent process exists and the OS supports it. Multiple threads work together to complete a task thereby achieving parallelism.

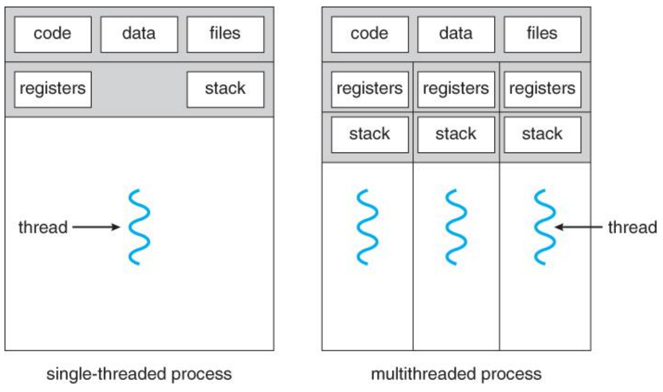


Fig 2: pThreads (Multithreading process)

A. *Why pThreads was chosen?*

1) Light Weight: When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

2) Efficient Communications/Data Exchange: For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process. There is no data transfer, per se. It can be as efficient as simply passing a pointer.

3) Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

4) Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.

In this paper, we use Pthreads to achieve acceleration of matrix operations in an INTEL Xeon 16 core Processor. We will be comparing the performance obtained against hardware acceleration achieved using Altera FPGA and Altera OpenCL SDK.

In general matrix multiplicat*ion is denoted in the following* way. C=A*B, where “A” and “B” are two matrices and “C” is the product resultant of the two.

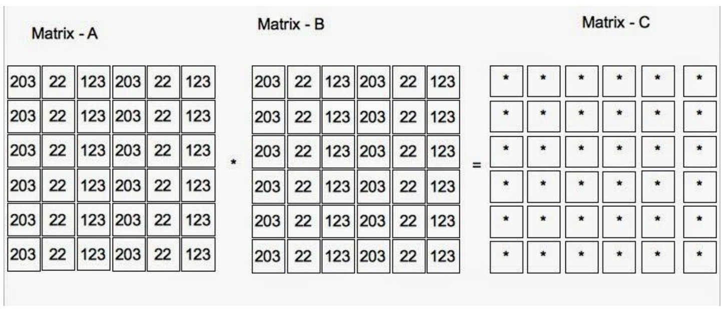
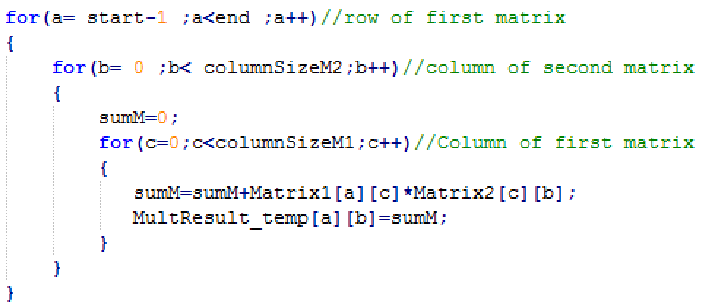


Fig 3: Matrix Multiplication

In the above figure matrix C denotes the resultant matrix which is obtained by multiplying each element in the row of matrix A with each element in the column of the matrix B and summing the result. For example: The result of the cell C (0,0) of the matrix C can be obtained by performing the following operation. C(0,0) = A(0,0) * B(0,0)+ A(0,1)*B(1,0) + ...A(0,n) *B(n,0);

This parallel implementation considers the delegation of tasks among different processor threads based on the concept of partitioning the rows of matrix A in a row-wise fashion while on the other hand, multiplying it with all columns of matrix B and calculating the resultant. In this implementation we read 2 text files as input that contain the matrix data. Each text file consists of 1 million elements (1000 x 1000) as input data. The performance is measured based on the time taken for the matrix multiplication operation and does not include the time taken to read the input files. The implementation snippet is given below. Start and end are determined depending on the number of processors used.



```
for(a= start-1 ;a<end ;a++)//row of first matrix
{
    for(b= 0 ;b< columnSizeM2;b++)//column of second matrix
    {
        sumM=0;
        for(c=0;c<columnSizeM1;c++)//Column of first matrix
        {
          sumM=sumM+Matrix1[a][c]*Matrix2[c][b];
          MultResult_temp[a][b]=sumM;
        }
    }
}
```

Fig 4: Matrix Multiplication Parallel implementation

B. *Results of pThread Implementation*

This implementation was carried out for 1, 2, 4, 8, 16 and 32 cores and the results are as follows:

Core 1: 3.769908071 sec

Core 2: 2.291178580 sec

Core 4: 1.289262632 sec

Core 8: 0.779504963 sec

Core 16: 0.423824383 sec

Core 32: 0.366285009 sec

IV. OPENCL OVERVIEW

OpenCL (Open Computing Language) is a specification introduced for parallel programming purposes across platforms. Applications written in OpenCL consist of two parts - a host program for initialization and management, and kernels that define the compute intensive tasks.

A. *OpenCL Platform model*

A host is connected to one or more OpenCL devices. OpenCL devices are a set or collection of one or more compute units (cores). Each compute unit is made up of one or more processing elements. The processing elements execute code as SIMD or SPMD.

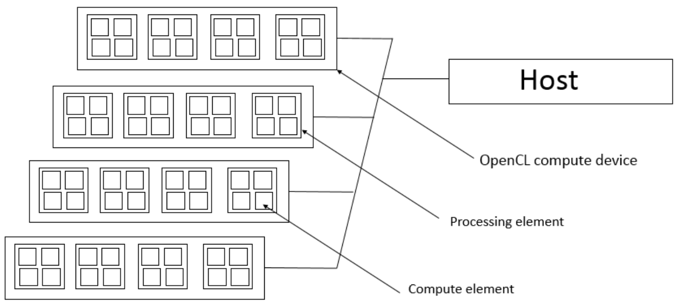


Fig 6: OpenCL platform model

B. *OpenCL Memory model*

It consists of Host memory, constant memory and local memory. The memory management here is explicit. You must move data from host to global to local and back. The memory division in this model is according to the following diagram.

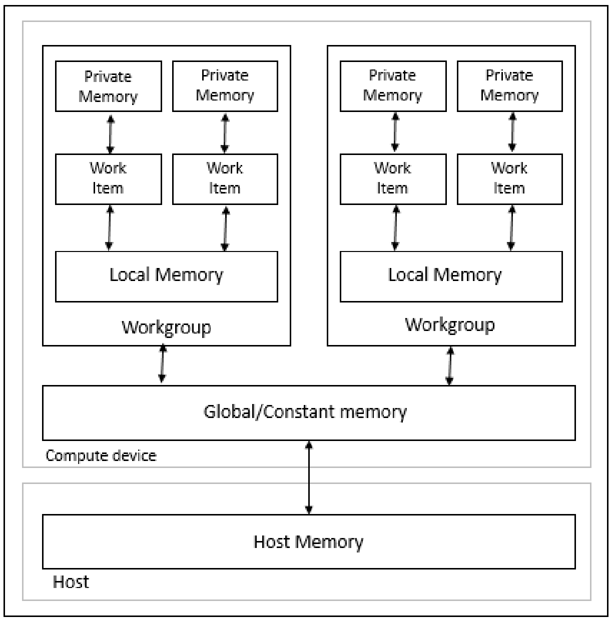


Fig 7: Memory model

C. *OpenCL Objects*

OpenCL consists of 4 basic type of objects:

1) Setup: Devices like GPU, CPU, and FPGA. Contexts, which are collection of devices. Queues, which are used to submit work to the devices.
2) Memory: Buffers, which are blocks of memory. Images, which are similar to blocks of memory but are used for 2D or 3D formatted images.
3) Execution: Kernel, which are Argument/execution instances of the specific function or operation. Programs, which are a collection of kernels.
4) Synchronization/profiling like Events.

D. OpenCL Framework

The following block diagram shows the basic blocks in OpenCL programming and the flow of the process. Refer figure 8.

E. OpenCL Implementation

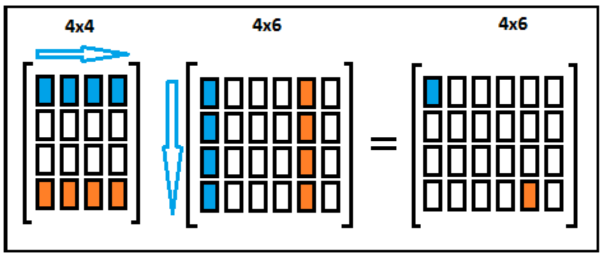
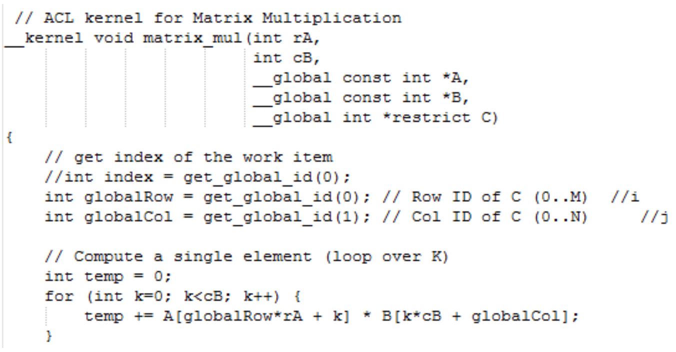


Fig 9: Data Parallelism

The OpenCL makes use of data level parallelization, in which workload is distributed among work group. In our implementation, matrix is multiplied with naïve approach and then compared the result with the matrix multiplication implemented in OpenCL. The OpenCL kernel performs multiplication of one row and one column and adds them together to form a single element of resultant matrix. So, two buffers, one for the row of matrix "A" and other for column of Matrix B, are passed to the OpenCL kernel. To pass the row and column to kernel matrix are converted into vector form, which makes it easy to transfer data easy between host and device.

We are using , for 1000x1000 multiplication, 1000x1000 workers (global_work_size = [1000,1000]), in way that each worker will produce one element of resultant array :

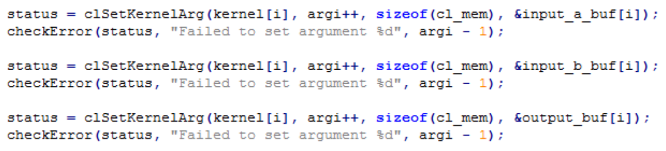


```
// ACL kernel for Matrix Multiplication
__kernel void matrix_mul(int rA,
                         int cB,
                         __global const int *A,
                         __global const int *B,
                         __global int *restrict C)
{
    // get index of the work item
    //int index = get_global_id(0);
    int globalRow = get_global_id(0); // Row ID of C (0..M)  //i
    int globalCol = get_global_id(1); // Col ID of C (0..N)     //j

    // Compute a single element (loop over K)
    int temp = 0;
    for (int k=0; k<cB; k++) {
        temp += A[globalRow*rA + k] * B[k*cB + globalCol];
    }
```

Fig 10: Kernel code snippet

At the host side, buffers are created to pass the Row vector of matrix A and column vector of matrix B to the OpenCL device.



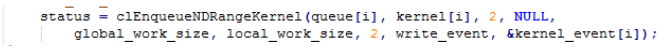
```
status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem), &input_a_buf[i]);
checkError(status, "Failed to set argument %d", argi - 1);

status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem), &input_b_buf[i]);
checkError(status, "Failed to set argument %d", argi - 1);

status = clSetKernelArg(kernel[i], argi++, sizeof(cl_mem), &output_buf[i]);
checkError(status, "Failed to set argument %d", argi - 1);
```

Fig 11: Setting kernel arguments

Once the kernel argument are set to the input buffers, next step would be to launch the kernel, which will do the operation on OpenCL device and return the result.



```
    status = clEnqueueNDRangeKernel(queue[i], kernel[i], 2, NULL,
        global_work_size, local_work_size, 2, write_event, &kernel_event[i]);
```

Fig 12: Launching kernel

F. *Implementation results*

The figure below shows the results of time taken by the OpenCL implementation on FPGA.

FPGA Time: 0.094ms

Time including communication: 27945.585ms

CPU Time: 6747.362ms

Verification: Passed (Comparing results of sequential implementation with FPGA results)

V. COMPARING RESULTS

As it can be seen from the diagram below the speedup obtained on FPGA is much more than obtained on the most threaded system i.e. with 32 cores. Although, this FPGA implementation doesn't include communication overhead of the PCIe bus. But if used still a very remarkable speedup will be obtained for high computation applications with compared to the ones implemented on a CPU.

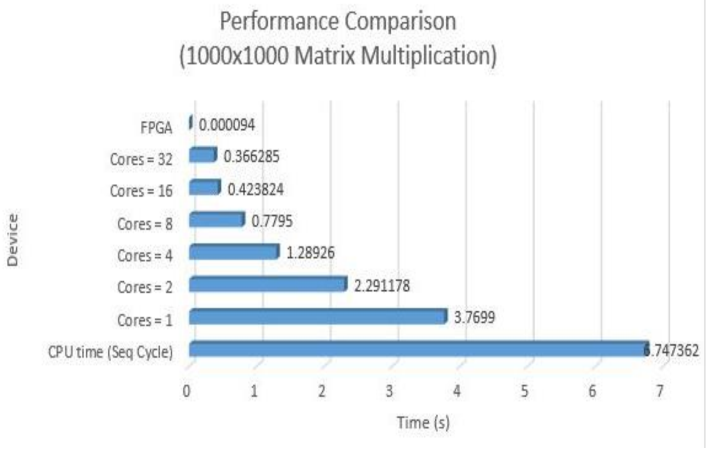


Fig 13: Result Comparison

VI. CONCLUSION

This paper successfully demonstrates the high computation power and acceleration obtained on an FPGA with compared to a multithreaded program run on a CPU. We have discussed and produced the results on how the matrix multiplication operation was performed using pThreads and parallel computing was achieved on the Intel Xeon CPU. We have also discussed and produced implementation results on how OpenCL was used to develop matrix multiplication kernels on the Altera FPGA and high performance with great speedup was achieved. The paper also presents the design and implementation of an architecture on the reconfigurable fabric to support the execution of the computation kernels by interfacing the cores with host and memory modules. Both the implementations were compared and OpenCL implementation on FPGA was concluded to produce high level speedup as compared to multithreaded ones. Although, there is a communication overhead for communicating with the FPGA via the PCIe connection but it was concluded theoretically that if heavy commutation was to be done using this technique, huge speed up and performance gain will be obtained. OpenCL provides a good abstraction from the low level details of hardware implementation. This guarantees smaller development times and faster time to market.

VII. REFERENCES

[1] *Enabling development of OpenCL applications on FPGA platforms by* Kavya S Shagrithaya.

[2] *Wikipedia OpenCL*

[3] What is OpenCL™? By Developer Central

[4] Higher level programming abstractions for FPGAs using OpenCL by Desh Singh

[5] Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks by Naveen Suda, Vikas Chandra,Ganesh Dasika*, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, Yu Cao

[6] From opencl to high-performance hardware on FPGAS by Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras and Deshanand P. Singh

[7] http://www.cmsoft.com.br/opencl-tutorial/case-study-matrix-multiplication/

[8]https://www.altera.com/support/support-resources/design-examples/design-software/opencl/vector-addition.html