Parallel Matrix Multiplication

A Comparison of OpenCL, CUDA, and Python

Submitted by Alexander Stein

EECSE4750
Dr. Zoran Kostic
Assignment 2
Columbia University
October 19th, 2016

Problem Statement:

The problem for this assignment is essentially broken into 3 parts:

1. Implement a matrix transpose in each language and compare results
2. Implement the multiplication of a matrix with its transpose in each language and compare results, verify that the result is symmetric.
3. Optimize the OpenCL and CUDA implementations with methods learned from class.

Platform Information:

GPU: Nvidia GTX 660M
CPU: Intel Core i7 3630QM @ 2.4GHz
OS: Ubuntu 14.04 LTS
RAM: 24 GB SODIMM DDR3 Synchronous 1333 MHz (0.8 ns)

Overall Structure Pseudocode:

```
Function TRANSPOSE, MULTIPLY (MxN Matrix A):

    For (Python):
        Use numpy built-in function

    For (OpenCL, CUDA):
        Obtain Platform and Device Information
        Create Context
        Build C Kernel
        Queue Commands
        Allocate device memory and transfer from host
        Specify desired hardware dimensions on device
        Run Program
        Copy outputs from device back to host
```

```
Function MAIN():
    For (i = 1 to K):
        Launch TRANSPOSE, MULTIPLY(Matrix size (k * M, k * N))
        Store Results
    Plot Results
```

OpenCL Transpose Kernel

```
__kernel void transpose(__global const unsigned int *A, __global unsigned int *T,
unsigned int M, unsigned int N)
        {

              unsigned int idx = get_global_id(0); // ROW id
              unsigned int idy = get_global_id(1); // COLUMN id
              unsigned int dimx = get_global_size(0); // num ROWs

              unsigned int id = dimx * idy + idx;


              // Transpose: tij = aji = A[j][i] = A_flat[i*N + j]
              unsigned int i, j, aji;
              i = idx % M;
              j = idx / M;
              aji = A[i * N + j];

              if(idx < N * M){
              T[idx] = aji;
              }

 }// end kernel
```

OpenCL Naive Multiply Kernel

```
__kernel void naive_multiply(__global const unsigned int *A, __global unsigned int
*T, __global unsigned int *X, unsigned int M, unsigned int N)
        {
              //Naive Multiply: Just iterate over every single element

              unsigned int j , k;
              unsigned int i = get_global_id(0); //ROW index
              unsigned int xvalue;

              for(j = 0; j < M; j++){

              xvalue = 0;
```

```
            // Input Matrix 'A' is MxN, and 'T' is NxM, so X is MxM

            for(k = 0; k < N; k++){
                  xvalue += A[i*N + k]*T[k*M + j];
            }

            X[i*M + j] = xvalue;
            }

}// end kernel
```

OpenCL Optimized Multiply Kernel

```
__kernel void opt_multiply(__global const unsigned int *A, __global unsigned int *T,
__global unsigned int *X, unsigned int M, unsigned int N)
       {
            // Indices needed for optimization
            unsigned int j , k;
            unsigned int i = get_global_id(0); // ROWs
            unsigned int xvalue = 0;
            unsigned int Awrk[1024];

            // Pull Row of A into private memory for faster accesses
            for(k = 0; k < N; k++){
            Awrk[k] = A[i*N + k];
            }//end k-for

            // Perform the Multiplication for a whole row
            // Here M - num rows in A, N - num columns in A
            for(j = 0; j < M; j++){

            xvalue = 0;
            for(k = 0; k < N; k++){
                  xvalue += Awrk[k]*T[k*M+j];
            }//end k-for

            X[i*M + j] = xvalue;

            }//end j-for

 }// end kernel
```

CUDA Naive Multiply Kernel

```
 __global__ void multiply(const unsigned int *A, unsigned int *T, unsigned int *X,
```

```
unsigned int M, unsigned int N)
	{

		unsigned int tidx = threadIdx.x;
		unsigned int bidx = blockIdx.x;
		unsigned int bdimx = blockDim.x;

		unsigned int id = bidx * bdimx + tidx;

		// Multiplication:
		// X[j*M + i] = Sum[k = 0 --> M]{A[i*M + k] * A_t[k*M + j]}

		// Accumulate Sum of Row-A * Column-T into X[i][j]
		unsigned int i, j, k, xij;
		i = id % M;
		j = id / M;
		xij = 0;
		// Input Matrix 'A' is MxN, and 'T' is NxM, so X is MxM
		if(id < M * M ){
			for (k = 0; k < N; k++){
			X[id] += A[j*N + k] * T[k*M + i];
			}
		}

		// Input Matrix 'A' is MxN, and 'T' is NxM, so X is MxM
		//if(id < M * M){
		//	X[id] = xij;
		//}
}
```

CUDA Optimized Multiply Kernel

```
__global__ void multiply(const unsigned int *A, unsigned int *T, unsigned int *X,
unsigned int M, unsigned int N)
	{
		// Define Constants
		#define TILE_WIDTH 2

		// Indices needed for optimization
		unsigned int tx = threadIdx.x;
		unsigned int bx = blockIdx.x;
		unsigned int ty = threadIdx.y;
		unsigned int by = blockIdx.y;

		// Multiplication:
		// X[j*M + i] = Sum[k = 0 --> M]{A[i*M + k] * A_t[k*M + j]}

		// Tiling Optimiziation: Pull tiles of size TILE_WIDTH x TILE_WIDTH
into local memory	before operation
```

```
        __shared__ unsigned int ds_A[TILE_WIDTH][TILE_WIDTH];
        __shared__ unsigned int ds_T[TILE_WIDTH][TILE_WIDTH];

        unsigned int Row = by * blockDim.y + ty;
        unsigned int Col = bx * blockDim.x + tx;
        unsigned int xvalue = 0;

        for(int t = 0; t < (N-1)/TILE_WIDTH + 1; t++){

               if(Row < M && (t*TILE_WIDTH + tx) < N){
               ds_A[ty][tx] = A[Row*N + t*TILE_WIDTH + tx];
               } else {
               ds_A[ty][tx] = 0;
               }

               if( (t*TILE_WIDTH + ty) < N && Col < M){
               ds_T[ty][tx] = T[(t*TILE_WIDTH + ty)*M + Col];
               } else {
               ds_T[ty][tx] = 0;
               }

               __syncthreads();

               for(int i = 0; i < TILE_WIDTH; i++){
               xvalue += ds_A[ty][i] * ds_T[i][tx];
               }// multiply and sum the tiled indices

        }// for all tiles

        __syncthreads();

        if ( (Row * M + Col) < M * M){
               X[Row*M+ Col] = xvalue;
        }

}// end kernel
```

Figure 1 : Python vs. PyOpenCL : Multiply

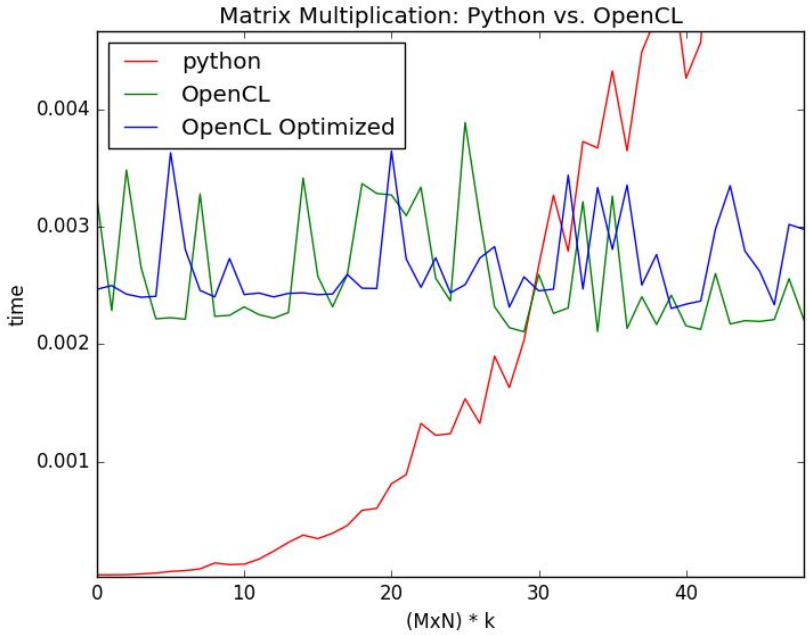
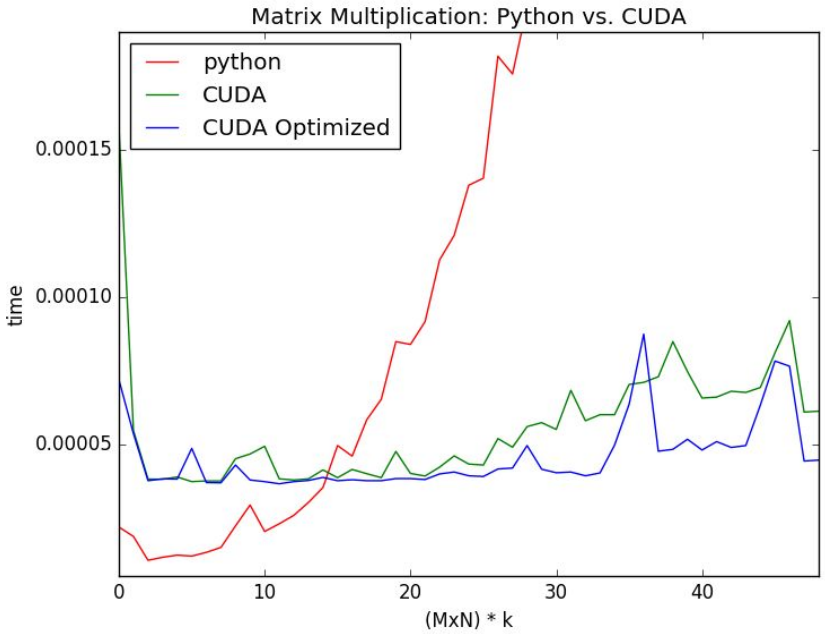


Figure 2 : Python vs. PyCUDA : **Square** Multiply



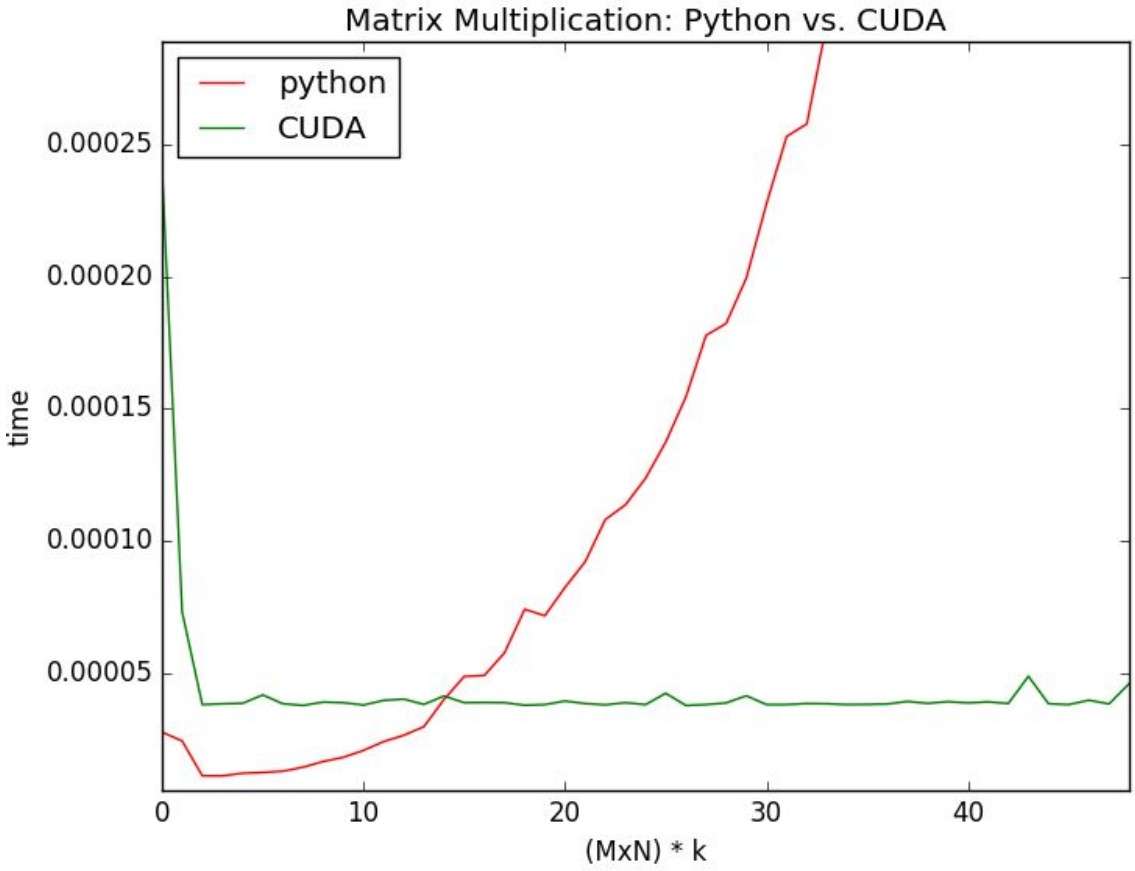
Results: Analysis

Each of figures 1-2 above show clearly that the time taken to execute the Mutliplication in Python (on the host CPU) increases exponentially in time when we scale the matrix size MxN up by K for each dimension, while the execution of the same function takes little additional time with that increase. Comparing Python with CUDA, we can see that for K > 15, CUDA yields the best performance. Comparing Python with OpenCL, we see that for K > 30, OpenCL yields the best performance. It is likely that CUDA is performing better than OpenCL because the platform used in this instance is an Nvidia GPU, for which CUDA is optimized. Also Nvidia does not support OpenCL past version 1.1, which may exclude many optimizations. Regardless, the benefits of parallelism are clear in both cases.

For both CUDA and OpenCL, the added optimizations seem to have very little performance benefit. Please see the kernel codes above for reference. Reasons for this could be: 1) incorrect implementation of optimizations in kernel; 2) incorrect selection of block/grid size for CUDA and global/local size for OpenCL. The first optimization (applied in both OpenCL and CUDA) was to calculate an entire row * column of in a single thread/work-item as opposed to doing a single element-element multiplication per thread/work-item. The second optimization in OpenCL was to pull the current row being used from an input matrix into private memory.

The second optimization used for CUDA was to tile the input matrices into shared memory. However, this implementation largely failed. Due to complexity in the indexing, and the attached complexity of choosing block and grid dimensions that did not botch the matrix calculation, I was only able to have the tiling optimization working for square, even-dimensioned matrices. The naive CUDA implementation of multiplication, however, was able to handle non-square, odd dimensional matrices (you can test this yourself by running the turned in code). See these results below

Figure 3 : Python vs. PyCUDA : **Nonsquare** Multiply (Not Optimized)



Theory Questions

1. Let there be a task whose latency is 100 microseconds. 90% of this task is parallelizable. A GPU can speed up this task 3 times. If we introduce a GPU in the system, what is the overall speedup? Theoretically, what is the maximum speed up this task can achieve? What is the speed up if the original system (without GPU) is replaced with 2 CPU cores?

    a. By Ahmdal's Law (S = 1 / ((1-p) + (p/s))) where S is speedup, p is % parallelism of the task under consideration, and s is the hardware speedup capability… we have::
        i. For GPU : S(3) = 2.5 -> Task will take 40 microseconds
        ii. For 2 CPUs : S(2) = 1.82 -> Task will take 54.95 microseconds

2. Describe in brief (1 line), what do the following statements do:
    a. - import numpy as np

        i. This brings the “numpy” package into python and gives it the alias name “np”
    b. - prg = cl.Program(ctx, kernel).build()
        i. Create a program for the given context, and build the kernel (using pyopencl)
    c. - ctx = cl.Context(devs)
        i. Create a context for the selected devices (pyopencl)
    d. - c_buf = cl.array.empty(queue, a.shape,a.dtype)
        i. This creates an empty array on the buffer in the shape and datatype of a.
    e. - prg.func(queue, name.shape, None, a_buf.data, b_buf.data)
        i. This launches the function “func” on the queue using all the devices allocated buffers.

3. Briefly explain the difference between private memory, local memory & global memory. What happens when you use too much private memory?

    a. Private memory belongs to each work-item in OpenCL and has a small size (a few B). Local memory belongs to each work-group in OpenCL, and has medium size (a few kB). Global memory is available to all work-items on the GPU, and is very large (a few GB). When you use too much private memory, it spills over into global memory.

4. Suppose you plan to multiply two matrices, and given the fact you know one of those is going to be a diagonal matrix (where only diagonal elements are non-zero), is there a way you can manipulate the inputs for faster and easier calculations before sending them to the Kernel? (Assume you can manipulate the product to form the original output later.)
    a. You could store the diagonals of each matrix into 1-D arrays, and just multiply each element of the first array by its corresponding element in the second array. The output would be also a 1-D array of the same length as the inputs. It would be re-manipulated later as the diagonal of an otherwise-zero matrix just like the inputs.

References

http://stackoverflow.com/questions/3042717/what-is-the-diff-between-a-thread-process-task
http://stackoverflow.com/questions/1050222/concurrency-vs-parallelism-what-is-the-difference
https://www.quora.com/What-is-the-difference-between-concurrency-and-parallelism