Optimized and Naive Non-Square Matrix Multiplication With Cuda and OpenCL

Drew Afromsky
Email: daa2162@columbia.edu

***Abstract*—In this assignment, matrix multiplication in both CUDA and OpenCL was implemented, taking advantage of both shared memory and global memory. A matrix of size [M X N] and a second matrix of size [N X M] are initialized. The matrix is then iteratively increased in both the x- and y-dimensions (i.e. [itr*M X itr*N]; [itr*N X itr*M]). First, a naive implementation of matrix multiplication, in both PyCUDA and PyOpenCL, is used, where both matrices are stored in global memory. Second, an optimized implementation of matrix multiplication is used, in both PyCUDA and PyOpenCL, where both matrices are stored in shared memory. Matrices in the naive and optimized PyCuda code were initialized to size 16x14 and 14x16, with each dimension in each matrix was iteratively increased by a factor of 1 to 38. The dimension of the largest matrix was 608 x 608. Matrices in the naive and optimized PyOpenCl code were initialized to the same size as those in PyCuda for consistency. However, matrices in PyOpenCL naive and optimized implementations were able to compute matrices of size 8000x8000 without encountering memory errors. Matrices in the optimized PyCuda implementation could also compute matrices of this size. Finally, run times were compared for PyCuda, Python (Serial/CPU), and PyOpenCL for both naive and optimized matrix multiplication implementations. PyOpenCL optimized code was clearly able to compute the matrix multiplication much faster than PyOpenCL naive and Python (serial/CPU). PyCuda optimized and naive had similar run-times, with the optimized code barely outperforming the naive implementation.**

***Index Terms*—PyCuda,CUDA, PyOpenCL, OpenCL, naive, optimized, matrix multiplication**

I. Introduction

Matrix-matrix multiplication between two matrices of size [M X N] and [N X M] results in a new matrix of size [M X M]. This operation presents opportunities for reduction of global memory accesses. The execution speed of matrix multiplication functions can vary by orders of magnitude, depending on the level of reduction of global memory accesses. A CUDA device contains several types of memory that can help programmers improve compute-to-global-memory-access-ratio, defined as the number of floating-point calculations performed for each access to the global memory within a region of a program. Specifically, we focus on global and shared memory. Global memory can be written and read by the device. Shared memory is on-chip memory. Variables that exist in this type of memory can be accessed at very high-speed in a highly parallel manner. Shared memory locations are allocated to thread blocks; all threads in a block can access shared memory variables allocated to that block. Shared memory is an efficient way for threads to share their input data and intermediate results. The global memory is large, but slow, whereas shared memory is small, but fast. This causes a trade-off in the use of device memories. A common strategy is to partition the data into subsets called tiles, so that each tile fits into the shared memory.

II. Methods

A. Naive Matrix Multiplication PyCuda and PyOpenCL

Unlike an optimized method for matrix multiplication, a naive method does not take advantage of shared memory and tiling. The process for computing matrix multiplication, of two rectangular matrices, using a naive approach can be broken down into the following steps: (1.) initialize the matrices of arbitrary size following the format m1 = [M X N] and m2 = [N X M], where after multiplication the resulting matrix is of size [N X N], (2.) create the kernel code for matrix multiplication utilizing simple indexing for each thread and computing the dot product of the rows and columns of the corresponding matrices; global/device memory is used here only (3.) move input and expected output matrices to device memory, (4.) call kernel function to perform naive matrix multiplication, (5.) measure run-time and return the multiplied matrix. These steps were performed for matrices of various sizes created by iteratively increasing the dimension of the initial matrices in both dimensions.

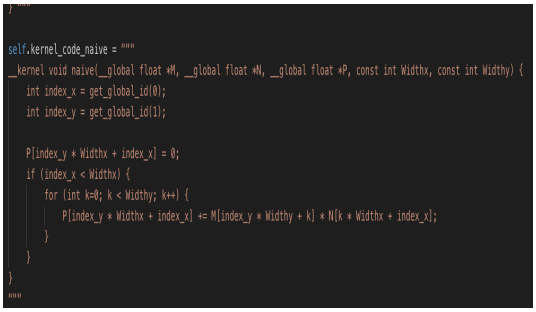
```
self.kernel_code_naive = """
__global__ void naive(float *M, float *N, float *P, const int Widthx, const int Widthy) {
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    P[Row * Widthx + Col] = 0;
    if (Col < Widthx) {
        for (int k=0; k < Widthy; k++) {
            P[Row * Widthx + Col] += M[Row * Widthy + k] * N[k * Widthx + Col];
        }
    }
}
"""
```

Fig. 1. PyCuda Naive Matrix Multiplication Kernel Code

B. Optimized Matrix Multiplication PyCuda and PyOpenCL

The optimized matrix multiplication was conducted in a similar manner to the naive method, however, the kernel code incorporates tiles and shared memory. The benefit for using shared memory and tiling is that through tiling we are able to partition the data that will be processed by threads and with utilizing shared memory (on-chip memory) we can access these variables quickly. With each tile fitting into



```
"""

self.kernel_code_naive = """
__kernel void naive(__global float *M, __global float *N, __global float *P, const int Widthx, const int Widthy) {
   int index_x = get_global_id(0);
   int index_y = get_global_id(1);

   P[index_y * Widthx + index_x] = 0;
   if (index_x < Widthx) {
      for (int k=0; k < Widthy; k++) {
         P[index_y * Widthx + index_x] += M[index_y * Widthy + k] * N[k * Widthx + index_x];
      }
   }
}
"""
```

Fig. 2. PyOpenCL Naive Matrix Multiplication Kernel Code

the shared memory, all the threads in that tile access these memory variables and we avoid global memory accesses (slower access). Therefore, steps 1-5 above were also followed for the optimized matrix multiplication method. The kernel code differs, however, in that shared memory is declared and shared memory locations are allocated to tiles of thread blocks. The kernel code follows the following structure: (1.) declare shared memory variables, where all threads in a block have access to the elements loaded into these memory variables (one pair for each block), (2.) save threadIdx and blockIdx values into automatic variables, and thus, registers, for fast access, (3.) identify the row and column of the output matrix element to work on/produced by the thread - the horizontal position, or the column index of the P element to be produced by a thread is calculated using bx*BLOCKSIZE + tx because each block covers BLOCKSIzE elements in the horizontal (and this logic is also similar for the vertical position, using by*BLOCKSIZE + ty) (4.) iterate through all the phases of calculating the P element, with each iteration corresponding to one calculation of a tiled matrix multiplication, (5.) load the appropriate input matrix element into shared memory, (6.) have each thread load an element 'tx' positions away from the beginning point, (7.) ensure all threads have finished loading the tiles of input matrices into shared memory matrix variables before any can move forward (using __syncthreads), (8.) perform one phase of the dot product on the basis of tile elements, (9.) ensure all threads have finished using the input matrix elements in the shared memory before any move on to the next iteration and load elements from next set of tiles, avoiding loading elements too early and corrupting input values for other threads.

III. Results

A. Run-Time Comparisons, Code Equality, Profiling (Naive Matrix Multiplication)

The size of the input matrices in each dimension were iteratively increased to produce a new matrix. For each iteration the CPU, PyCuda kernel, and PyOpenCl kernel execution times for matrix multiplication were recorded. PyCuda optimized and naive matrix multiplication kernel execution times were compared to CPU/serial code. PyOpenCL optimized and naive matrix multiplication kernel execution times were compared to CPU/serial code. The plots for run-time using PyOpenCL and PyCuda can be seen in figures 5 and 6. For consistency

```
# Write the kernel code
self.kernel_code_optimized = """
__global__ void optimized(float*M, float*N, float*P, const int Widthx, const int Widthy)

{
    __shared__ float Mds[%(BLOCK_SIZE)s][%(BLOCK_SIZE)s];
    __shared__ float Nds[%(BLOCK_SIZE)s][%(BLOCK_SIZE)s];

    int tx=threadIdx.x;
    int ty=threadIdx.y;
    int bx=blockIdx.x;
    int by=blockIdx.y;

    int Row = ty + by * %(BLOCK_SIZE)s;
    int Col = tx + bx * %(BLOCK_SIZE)s;

    float Pvalue=0;

    for(int ph=0; ph<= Widthy/%(BLOCK_SIZE)s; ph++) {
        if (Row < Widthx && ph+%(BLOCK_SIZE)s + tx < Widthy){
            Mds[ty][tx]=M[Row * Widthy + ph * %(BLOCK_SIZE)s + tx];
        }
        else{
            Mds[ty][tx]=0;
        }
        if (Col < Widthx && ph+%(BLOCK_SIZE)s + ty < Widthy){
            Nds[ty][tx]=N[(ph * %(BLOCK_SIZE)s + ty)* Widthx + Col];
        }
        else{
            Nds[ty][tx]=0;
        }
        __syncthreads();

        for (int k = 0; k < %(BLOCK_SIZE)s; ++k) {
            Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();
        }
    if (Row < Widthx && Col < Widthx) {
        P[Row * Widthx + Col] = Pvalue;
    }
} """
```

Fig. 3. PyCuda Optimized Matrix Multiplication Kernel Code

```
# Write the kernel code
self.kernel_code_optimized = """
__kernel void optimized(__global float*M, __global float*N, __global float*P, const int Widthx, const int Widthy)

{
    __local float Mds[%(BLOCK_SIZE)s][%(BLOCK_SIZE)s];
    __local float Nds[%(BLOCK_SIZE)s][%(BLOCK_SIZE)s];

    int tx=get_local_id(0);
    int ty=get_local_id(1);
    int bx=get_group_id(0);
    int by=get_group_id(1);

    int Row = ty + by * %(BLOCK_SIZE)s;
    int Col = tx + bx * %(BLOCK_SIZE)s;

    float Pvalue = 0;

    for(int ph = 0; ph <= Widthy / %(BLOCK_SIZE)s; ph++) {
        if (Row < Widthx && ph * %(BLOCK_SIZE)s + tx < Widthy){
            Mds[ty][tx]=M[Row * Widthy + ph * %(BLOCK_SIZE)s + tx];
        }
        else{
            Mds[ty][tx]=0;
        }
        if (Col < Widthx && ph * %(BLOCK_SIZE)s + ty < Widthy){
            Nds[ty][tx]=N[(ph + %(BLOCK_SIZE)s + ty)* Widthx + Col];
        }
        else{
            Nds[ty][tx]=0;
        }
        __syncthreads();

        for (int k = 0; k < %(BLOCK_SIZE)s; ++k) {
            Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads();
        }
    if (Row < Widthx && Col < Widthx) {
        P[Row * Widthx + Col] = Pvalue;
    }
} """
```

Fig. 4. PyOpenCL Naive Matrix Multiplication Kernel Code

between PyOpenCL and PyCuda, the figures below were produced using output matrices beginning with dimension 16 x 16 iterating each dimension by a factor 1 to 38, with the largest matrix of size 608 x 608. The horizontal axis is the number of elements in the matrix during each iteration. The vertical axes are the run times in seconds. As stated earlier, the kernel execution time for matrices of much larger size, i.e. 8000x8000, was able to be recorded in PyOpenCL, without encountering memory errors. PyCuda optimized kernel code was able to compute matrices of this size as well without encountering memory issues, however, the naive method was limited to matrices of size 608 x 608.

Figure 7 is a screenshot of the profiling for PyCuda. The Visual Profiler is a graphical profiling tool that displays a timeline of CPU and GPU activity, and that includes an automated analysis engine to identify optimization opportunities.

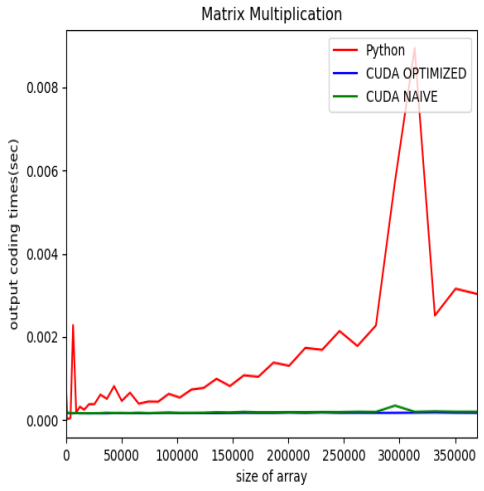


Fig. 5. PyCuda Execution Times for Naive and Optimized Matrix Multiplication Kernel Code

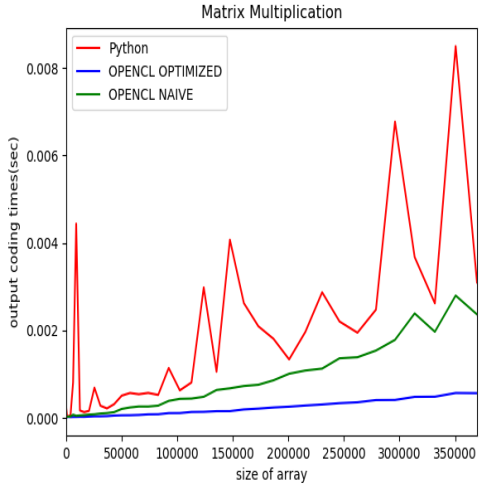


Fig. 6. PyOpenCL Execution Times for Naive and Optimized Matrix Multiplication Kernel Code

IV. Discussion

In this assignment, matrix multiplication was successfully implemented using three different methods: 1. CPU/Serial, 2. PyCuda (naive + optimized), and 3. PyOpenCL (naive + optimized). From the results, it is clear that CPU matrix multiplication operation is much slower compared to PyCuda and PyOpenCL. The naive PyCuda method and the optimized PyCuda method had similar execution times for matrices of dimension 16 x 16 as large as 608 x 608. It would be interesting to see if the execution times would begin to diverge with much larger matrices. However, the execution time for PyOpenCL naive was longer than PyCuda. This is to be expected, since reducing global memory accesses and utilizing tiled matrix multiplication with shared memory accesses should speed up

```
==23200== Profiling application: python PyCuda_matmul.py
==23200== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 62.65%  78.113ms       114  685.21us  5.5680us  2.4758ms  naive
 15.23%  18.990ms       456  41.643us     960ns  133.09us  [CUDA memcpy HtoD]
 14.11%  17.598ms       114  154.37us  3.7440us  539.17us  optimized
  8.00%  9.9765ms       228  43.756us  1.8880us  125.92us  [CUDA memcpy DtoH]

==23200== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 27.28%  216.49ms         1  216.49ms  216.49ms  216.49ms  cuCtxCreate
 14.30%  113.49ms       456  248.88us  27.686us  573.57us  cuMemcpyHtoD
 13.19%  104.65ms         1  104.65ms  104.65ms  104.65ms  cuCtxDetach
 11.71%  92.911ms       228  407.51us  4.2210us  2.4633ms  cuEventSynchronize
 10.40%  82.565ms       684  120.71us  6.7920us  571.81us  cuMemAlloc
  9.66%  76.662ms       684  112.08us  9.2130us  1.4315ms  cuMemFree
  9.52%  75.562ms       228  331.41us  36.294us  1.3028ms  cuMemcpyDtoH
  1.63%  12.919ms        76  169.99us  130.66us  576.94us  cuModuleLoadDataEx
  0.94%  7.4424ms       228  32.642us  27.768us  72.321us  cuLaunchKernel
  0.46%  3.6778ms        76  48.392us  35.346us  100.72us  cuModuleUnload
  0.43%  3.4130ms       456  7.4840us  5.9870us  16.209us  cuEventRecord
  0.16%  1.2939ms       456  2.8370us  1.3830us  14.150us  cuEventCreate
  0.09%  747.78us       456  1.6390us     742ns  12.152us  cuEventDestroy
  0.08%  625.56us       761     822ns     476ns  5.7430us  cuCtxGetDevice
  0.05%  418.83us       228  1.8360us  1.5690us  7.0160us  cuFuncSetBlockShape
  0.03%  255.64us       228  1.1210us     931ns  5.0180us  cuModuleGetFunction
  0.03%  224.76us       384     585ns     284ns  7.6360us  cuDeviceGetAttribute
  0.02%  191.90us       304     631ns     296ns  8.8920us  cuDeviceComputeCapability
  0.00%  15.520us         6  2.5860us     780ns  9.4810us  cuCtxPopCurrent
  0.00%  11.650us         6  1.9410us     469ns  8.3070us  cuCtxPushCurrent
  0.00%  7.6530us         3  2.5510us     636ns  4.6230us  cuDeviceGetCount
  0.00%  4.2000us         3  1.4000us     863ns  1.7740us  cuDeviceGet
```

Fig. 7. PyCuda Profiling for Naive and Optimized Matrix Multiplication Kernel Code

```
==23200== NVPROF is profiling process 23200, command: python PyCuda_matmul.py
('CUDA NAIVE SHAPE:', (16, 16))
('CUDA OPTIMIZED SHAPE:', (16, 16))
('Code equality for CPU and naive GPU computation:', False)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (32, 32))
('CUDA OPTIMIZED SHAPE:', (32, 32))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (48, 48))
('CUDA OPTIMIZED SHAPE:', (48, 48))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (64, 64))
('CUDA OPTIMIZED SHAPE:', (64, 64))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (80, 80))
('CUDA OPTIMIZED SHAPE:', (80, 80))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (96, 96))
('CUDA OPTIMIZED SHAPE:', (96, 96))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (112, 112))
('CUDA OPTIMIZED SHAPE:', (112, 112))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (128, 128))
('CUDA OPTIMIZED SHAPE:', (128, 128))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (144, 144))
('CUDA OPTIMIZED SHAPE:', (144, 144))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (160, 160))
('CUDA OPTIMIZED SHAPE:', (160, 160))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (176, 176))
('CUDA OPTIMIZED SHAPE:', (176, 176))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (192, 192))
('CUDA OPTIMIZED SHAPE:', (192, 192))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)

('CUDA NAIVE SHAPE:', (512, 512))
('CUDA OPTIMIZED SHAPE:', (512, 512))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (528, 528))
('CUDA OPTIMIZED SHAPE:', (528, 528))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (544, 544))
('CUDA OPTIMIZED SHAPE:', (544, 544))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (560, 560))
('CUDA OPTIMIZED SHAPE:', (560, 560))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (576, 576))
('CUDA OPTIMIZED SHAPE:', (576, 576))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (592, 592))
('CUDA OPTIMIZED SHAPE:', (592, 592))
('Code equality for CPU and naive GPU computation:', False)
('Code equality for CPU and optimized GPU computation:', True)
('CUDA NAIVE SHAPE:', (608, 608))
('CUDA OPTIMIZED SHAPE:', (608, 608))
('Code equality for CPU and naive GPU computation:', True)
('Code equality for CPU and optimized GPU computation:', True)
```

Fig. 8. PyCuda Code Equality Between CPU and Naive and Optimized Matrix Multiplication Kernel Code

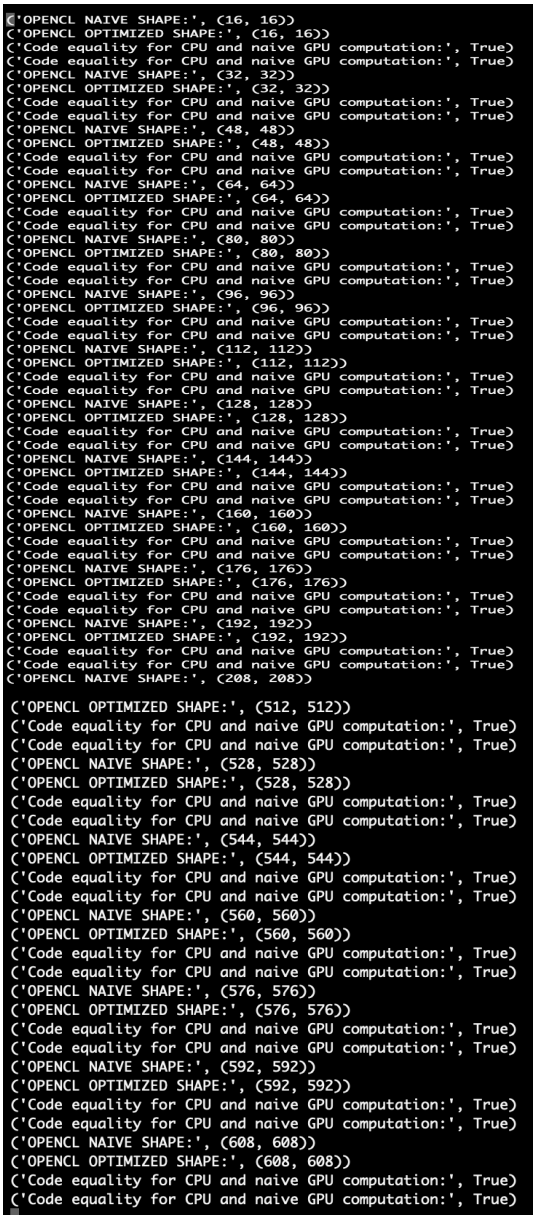


Fig. 9. PyOpenCL Code Equality Between CPU and Naive and Optimized Matrix Multiplication Kernel Code

kernel execution time.

REFERENCES

[1] David B. Kirk and Wen-mei W. Hwu. 2016. Programming Massively Parallel Processors, Third Edition: A Hands-On Approach (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.