PARALLEL COMPUTING ASSIGNMENT 2

Matrix Multiplication and Parallel Computation:
Matrix Multiplication is an interesting parallelization problem. Each element of a matrix has a unique set of constituent elements/operations for each element. A simple 3x3 matrix is unrolled below:

$$\begin{pmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{pmatrix} \begin{pmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{pmatrix} =$$

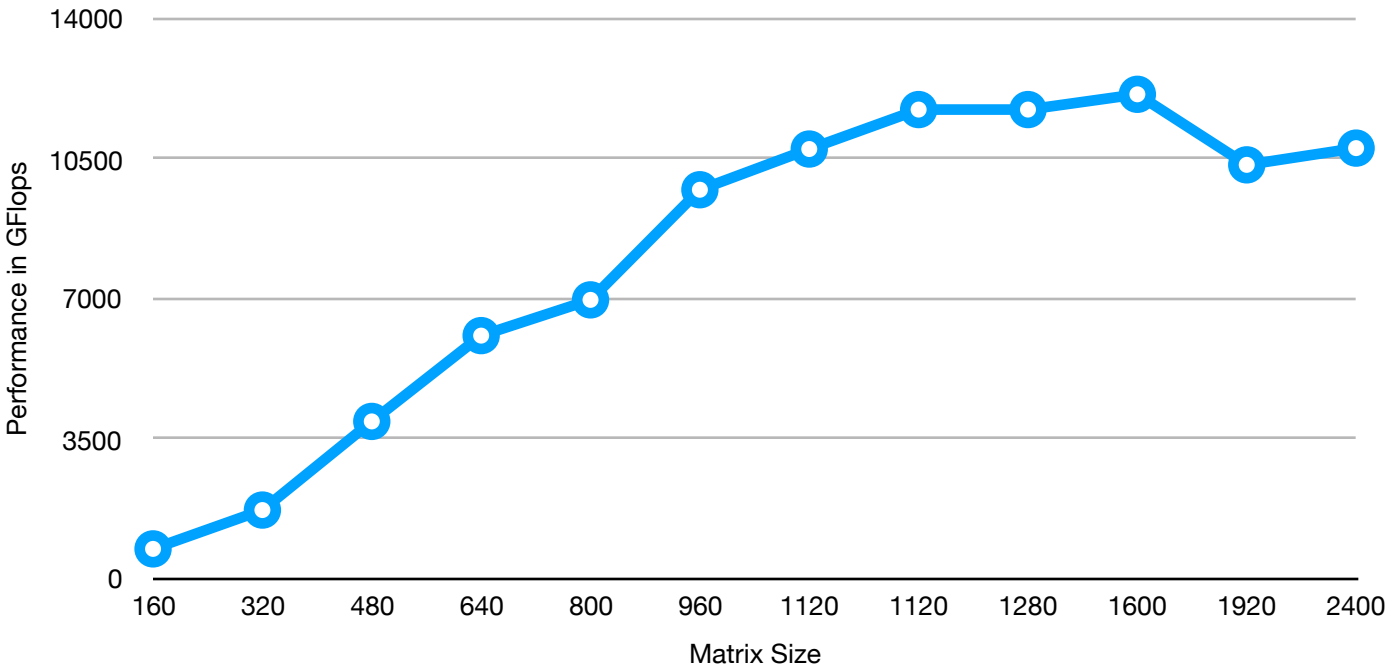
$$\begin{pmatrix} a11.b11 + a12.b21 + a13.b31 & a11.b12 + a12.b22 + a13.b32 & a11.b13 + a12.b23 + a13.b33 \\ a21.b11 + a22.b21 + a23.b31 & a21.b12 + a22.b22 + a23.b32 & a21.b13 + a22.b23 + a23.b33 \\ a31.b11 + a32.b21 + a33.b31 & a31.b12 + a32.b22 + a33.b32 & a31.b13 + a32.b23 + a33.b33 \end{pmatrix}$$

For example:
Let result matrix be C. For the result matrix element C[1][1]: a[1][1] is multiplied with b[1][1] as part of the calculation. a[1][1] is never multiplied with b[1][1] again for another element making caching the answer useless. Likewise, the constituents of the first element c[1][1] are never used together exclusively for another element.
Parallelizing a matrix multiplication is possible because each element can be independently calculated without dependencies. Hence, each processing unit can be given an element to process and since there is no need for memory/information sharing during calculation, the speed-up from parallel processing can be considerable. But the parallelization is affected by the fact that as the size of the matrix increases, each element's processing needs much more space and computation- unlike, for example, a SAXPY program (embarassingly parallel), where the space and computation is constant.

Using a GPU for parallelizing the Matrix Multiplication is a viable idea because matrix multiplication is a compute intensive problem which can greatly benefit from the numerous cores of a GPU. A CPU's main draws such as superior cache size, branch evaluation and more powerful cores are less effective for Matrix Multiplication speed-up compared to that of the GPUs'.

All experiments are conducted on a 16 Thread CPU and a single Nvidia Tesla V100.

1. The CUBLAS library has subroutines for Basic Linear Algebra Subprograms. This library has a well optimized Matrix Multiplication routine that is used in program 1. The performance of the Matrix Multiplication is expected to provide significant performance improvement until a certain size then plateau as the matrix size becomes larger. In the table below, we can see the performance of the CUBLAS matrix multiplication routine for matrix sizes upto 2400 (ie C=A*B where each matrix A and B upto size 2400 x 2400).



The matrix multiplication routine is very effective in obtaining a substantial speed up, especially approaching the 1600x1600 matrix that requires 8192 x 10^6 operations and 256000 elements.

A V100 processor has 80 SMs, 32 Blocks per SM and 1024 Threads per Block Maximum.

Hence, 80 x 32 x 1024=2621440 which is just above the number of elements in the 1600 x 1600 matrix. This means that each of the elements are immediately scheduled to a kernel. But, post this size, the scheduling needs to be done again- thereby causing a fall in performance.

2. Basic CUDA library gives the user control over GPU (device) memory and allows launching of kernels. Kernels follow a SIMD (Single Instruction Multiple Data) class of parallel computing and are launched from the host. Once the data is on the device, the kernels can access it. We can also define the number of Threads and Blocks while launching the kernels. Some traits of kernels:

a. Each Kernel is executed by a Thread, that is part of a Block, that in turn is part of a Grid. Thousands of kernels are launched in parallel and it is impossible to predict exact order of kernel execution/completion.

b. Each Kernel has a unique id number that can be obtained by using the block Id, dimensions of the block and the thread Id of that kernel in the block.

Kernel order of execution is often unpredictable. However, using the id number of the kernel, we can assign jobs to each kernel -using the identity to predetermine the job of a kernel instead of the order of execution. So, for example, the kernel that has the Id 2 can process the 2nd element in the matrix. Using a 2D Block and 2D Thread implementation (x & y), we can implement matrix multiplication. The 2D aspect of the Kernel Id can help determine the row and column, respectively, of the element for execution. The code snippet is given below.

```
Col=blockIdx.x*blockDim.x+threadIdx.x;//Col and Row Ids of threads
Row=blockIdx.y*blockDim.y+threadIdx.y;
```

Optimisation:
Considering the thread, block dimensions' impact on performance, it might be worth attempting a 1D implementation as well, to check the performance. The performance of 1D kernel is faster but declines quickly after 1600 matrix similar to the CUBLAS subroutine. Logic for 1D matrix multiplication to determine row and column numbers:

```
i = blockIdx.x*blockDim.x + threadIdx.x;
int quot=i/row_num_b;//For Rows
int rem=i%row_num_b;//For Columns
```

The performance of the 2D Kernel implementation is significantly slower- almost by half. Since the processing units are the same, this disparity might need more analysis to understand. When doing so, we realize that memory reads are different for 2D and 1D kernels. When Kernels of a block read from GPU memory, they can coalesce their reads. For example,
If our threads need to read, from memory, the following matrix elements in order:

Thread 0: 0 1 2
Thread 1: 3 4 5
Thread 2: 6 7 8

Since these threads execute in parallel, the memory reads would be 0, 3 and 6 to begin. Then next 1, 4, 7 and so on. However, this is inefficient because threads can coalesce memory reads using contiguous memory spaces. Using the following memory read would be more appropriate:

Thread 0: 0 3 6
Thread 1: 1 4 7
Thread 2: 2 5 8

Since in a 1D implementation, the execution happens across rows, the threads are likely benefitted by contiguous memory access. However, the 2D kernel definitely has room for improvement.

Even though, as mentioned in the Analysis section of matrix multiplication, caching answers to operations for a matrix multiplication is useless- caching the memory reads to the matrices to be multiplied is useful.
The Nvidia CUDA library gives the user access to the block level shared memory. This shared memory can be used to cache the sections of the matrix that the block needs for execution. Each block processes one subsection of the matrix and writes to the final matrix elements' memory. A block writes as many elements are there are threads in the block- which in turn references the kernels. Setting the block dimension as 32x32 provides the best speed-up and uses 32x32=1024 max threads per block.

According to the technical report on Nvidia Volta architecture (link):

"The V100 GPU has up to 96 KiB of shared memory (configurable) with low latency and high memory bandwidth. "

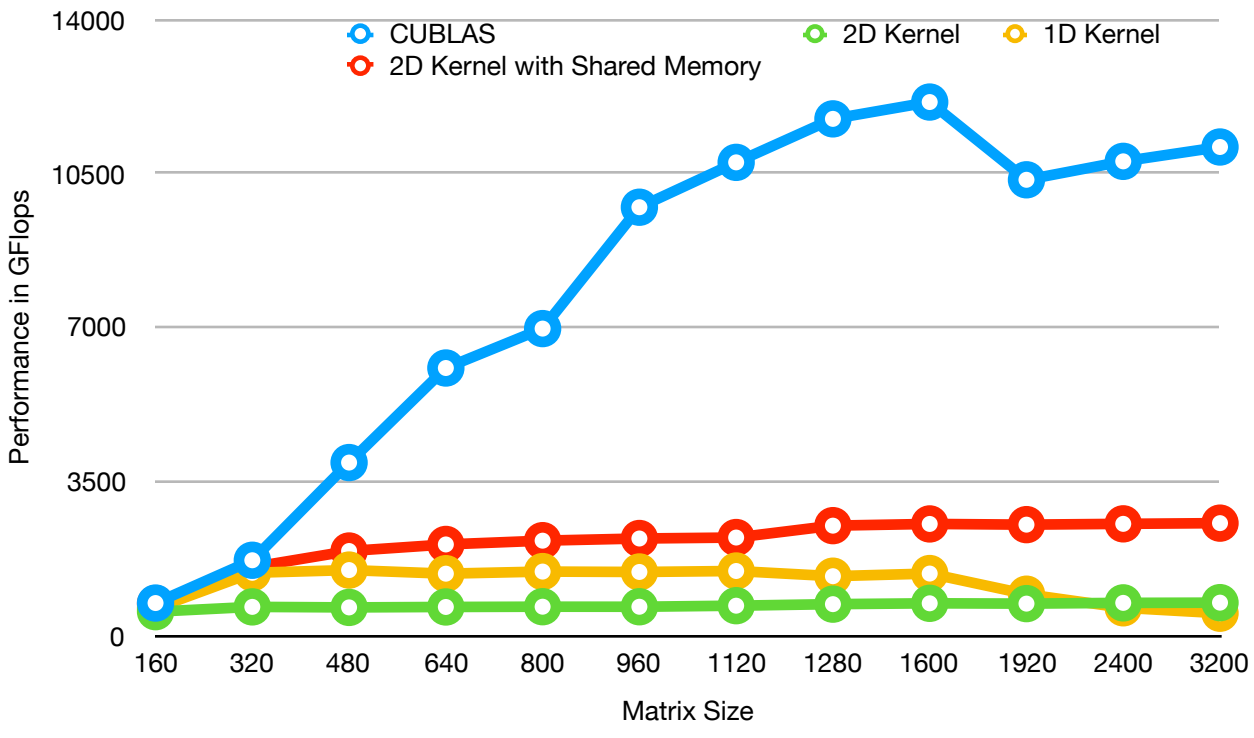
With 96KiB of shared memory per SM, and 2*32*32*4B= 8KB per thread block- there can be about 12 blocks in active execution at any point in a single Streaming Multiprocessor.

The logic behind the Shared-Memory implementation is as follows

a. Load the necessary data for block from GPU memory to shared memory.
b. Wait till all threads Synchronize.
C. Calculate element value using data in shared memory.

The results and comparison are presented below:

Matrix/ Algo	160	320	480	640	800	960	1120	1280	1600	1920	2400	3200
CUBLAS	736	1708	3927	6075	6970	9727	10746	11734	12116	10350	10769	11095
2D Kernel	540	649	637	647	653	652	676	711	730	723	740	745
1D Kernel	616	1423	1485	1403	1451	1442	1464	1346	1406	937	628	505
2D Kernel with Shared Memory	722	1553	1918	2067	2153	2202	2224	2494	2533	2517	2535	2551



The performance of the shared memory implementation is consistent in terms of performance unlike the previous implementations, but the performance does not scale as well as the CUBLAS library for larger matrix models. The memory access mapping optimization is likely the biggest reason for the superior performance of CUBLAS. Minimum GPU memory reads along with optimal algorithmic implementation of the kernel (eg; reduced branch evaluations), effective use of L2 cache and registers probably contributes to improved performance.

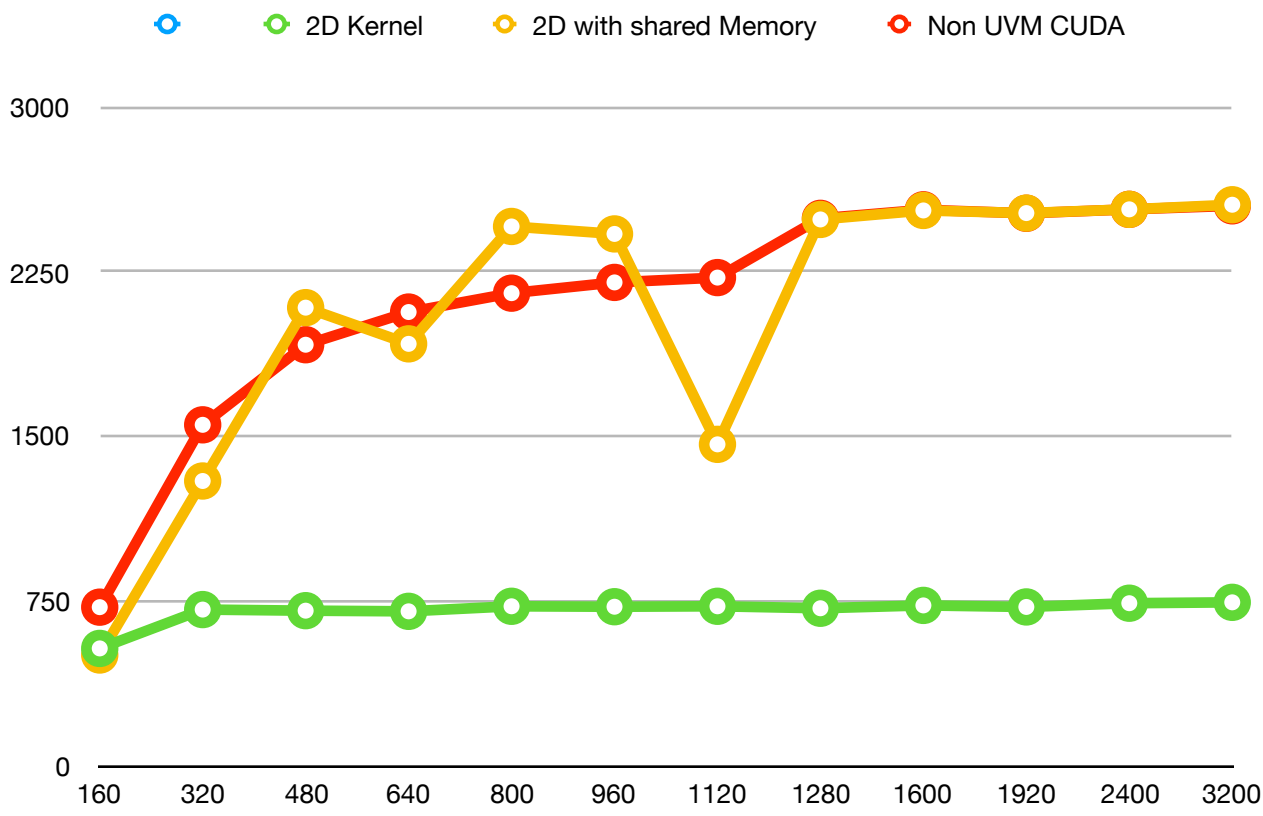
3. CUDA library allows the user to manage the memory on the host. However, the the CUDA library has Unified Memory access- meaning that both device and host memory can be allocated as a shared pool. This removes the need to explicitly move data between the host and device or Deep Copies.

The implementation on 1D, 2D and shared memory 2D results are presented below:

	160	320	480	640	800	960	1120	1280	1600	1920	2400	3200
CUBLAS	736	1708	3927	6075	6970	9727	10746	11734	12116	10350	10769	11095
2D Kernel	534	711	706	703	726	724	726	717	730	723	740	744
2D with shared Memory	504	1297	2087	1922	2457	2423	1464	2489	2530	2518	2535	2556
Non UVM CUDA	722	1553	1918	2067	2153	2202	2224	2494	2533	2517	2535	2551

The shared memory implementation closely mimics the shared memory implementation of the previous question without UVM. the difference in performance is They seem to be closer in performance for larger matrix sizes. Even the non shared memory versions are similar in performance- so there doesn't seem to be a large impact of using UVM on performance- only the algorithm seems to be different.

Considering the smaller size of the code (as explicit data transfers were unnecessary), this implementation is more easier on the coder.



4. OpenMP is a library that helps with parallelization on both CPUs and GPUs together. We are likely to see a speed-up on the Matrix Multiplication execution by using appropriate parallelization commands available in OpenMp.

The OpenMp library allows the data to be offloaded to the GPU using the 'device' command. Using the simple CPU matrix multiplication algorithm in the assignment code, and then running it with open MP parallelization for both the CPU and GPU, we get the following results:

Device\Size	160	320	480	640	800	960	1120	1280
CPU	0.55	0.47	0.46	0.47	0.47	0.47	0.48	0.27
Open MP CPU	1.10	4.45	5.15	4.99	5.48	5.44	5.58	3.28
Open Mp GPU	3.10	4.42	3.69	5.28	2.99	1.89	3.35	3.13

The library clearly provides a substantial speed up over the non-parallelized version but is not as good as the previous GPU based CUDA implementations by a significant margin. For example the speed-up provided by CUDA shared memory implementation is 809.2 over the OpenMp version for 1280x1280 matrix.

Analysis:
Among all the implementations, the CUBLAS library is an obvious choice due to ease of use, since the routine is readily available, and its superior performance compared to any other matrix multiplication method.
However, in case of manual implementation, the CUDA UVM is easier to work with. Considering that the same optimizations such as the shared memory for blocks and grid size- work on the UVM model just as well as the basic CUDA- it appears to be a good option.

Despite OpenMp being the easiest to program, since it only needs the addition of some library specific statements to a conventional Matrix Multiplication code, it does not provide a speed-up on par with CUDA programming. The power of programming at the hardware level that the CUDA library is unlikely to be matched by the OpenMp library even if the directives are used properly.

The advantages of implicit data sharing makes the use of UVM programming the best choice for future programming tasks that can benefit from using the GPU.

Sources:
1. https://ncalculators.com/matrix/3x3-matrix-multiplication-calculator.htm
2. https://devblogs.nvidia.com/unified-memory-in-cuda-6/
3. https://arxiv.org/pdf/1804.06826.pdf
4. http://users.umiacs.umd.edu/~ramani/cmsc828e_gpusci/Lecture5.pdf