CPSC 8200 Project 2 Report

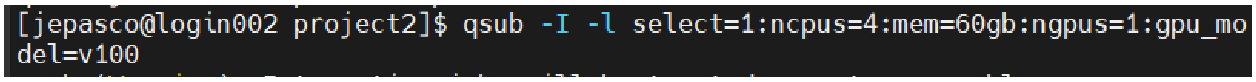
John Pascoe
jepasco@clemson.edu

1: Introduction

This writeup delves into the intricacies of CUDA programming, specifically focusing on matrix multiplication, a fundamental operation in various scientific and engineering applications. By following the guidelines of the instructions, I aimed to not only implement the matrix multiplication operations, but also analyze and optimize their performance. Through rigorous experimentation, I gained insights into the impact of matrix size on performance, as well as the several different kinds of optimization techniques that can be used to increase the performance of the operation. The writeup presents a comprehensive analysis of the observed results, shedding light on the challenges associated with different approaches to matrix multiplication.

2: Results

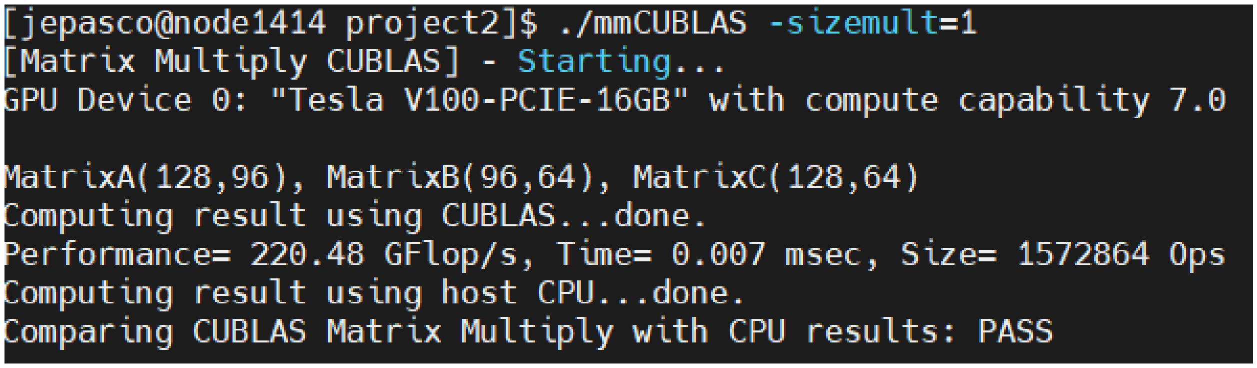
All results are based off of the following job node command:



```
[jepasco@login002 project2]$ qsub -I -l select=1:ncpus=4:mem=60gb:ngpus=1:gpu_mo
del=v100
```

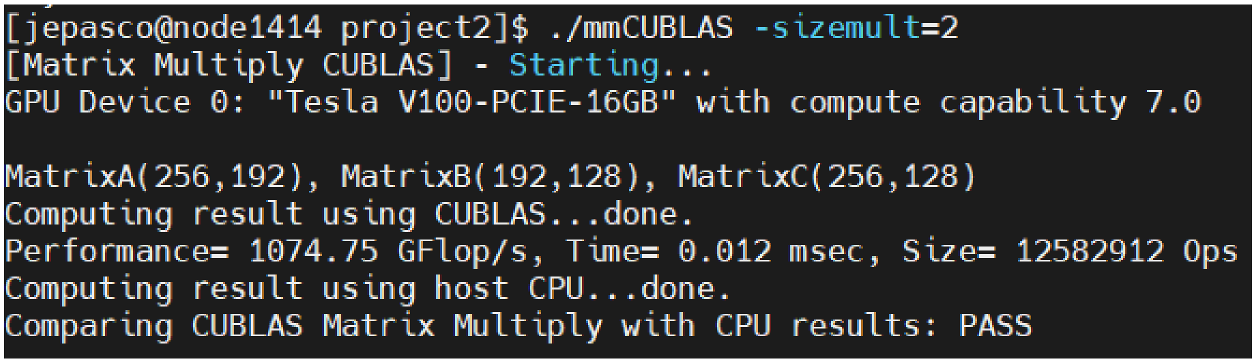
Qsub -I -l select=1:ncpus4:mem=60gb:ngpus1:gpu_model=v100

Part 1: Compute Matrix Multiplication w/ Cuda Library

- 

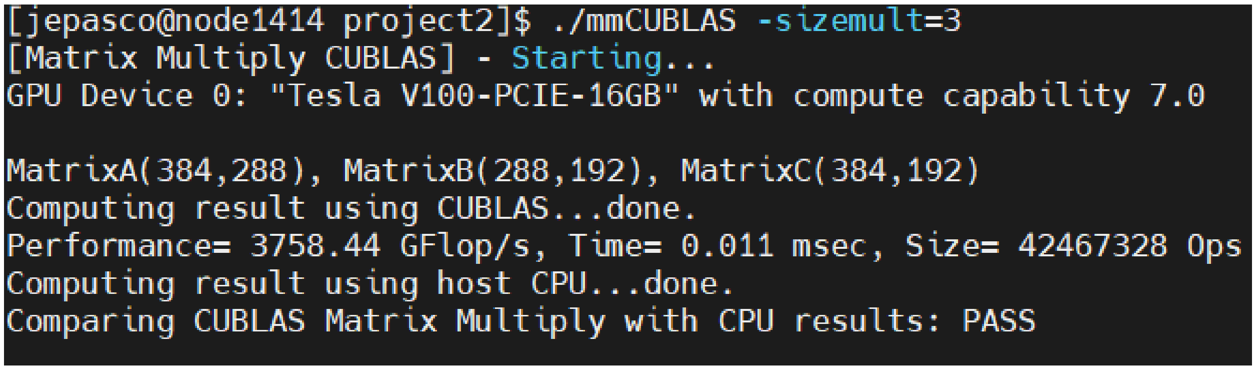
```
[jepasco@node1414 project2]$ ./mmCUBLAS -sizemult=1
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-16GB" with compute capability 7.0

MatrixA(128,96), MatrixB(96,64), MatrixC(128,64)
Computing result using CUBLAS...done.
Performance= 220.48 GFlop/s, Time= 0.007 msec, Size= 1572864 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
```

    - This image shows the output from mmCUBLAS using the -sizemult parameter set to 1. It took 0.007 msecs to perform the execution.

- 

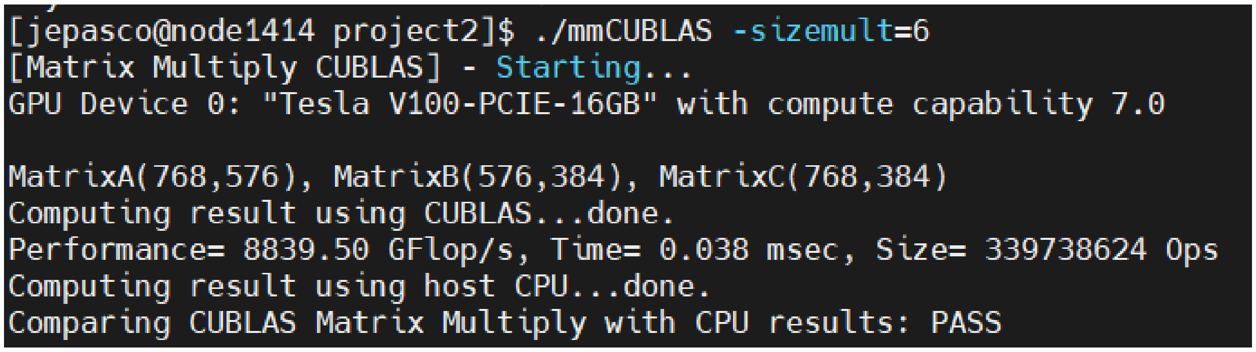
```
[jepasco@node1414 project2]$ ./mmCUBLAS -sizemult=2
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-16GB" with compute capability 7.0

MatrixA(256,192), MatrixB(192,128), MatrixC(256,128)
Computing result using CUBLAS...done.
Performance= 1074.75 GFlop/s, Time= 0.012 msec, Size= 12582912 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
```

  - This image shows the output from mmCUBLAS using the -sizemult parameter set to 2. It took 0.0012 msecs to perform the execution.

- 

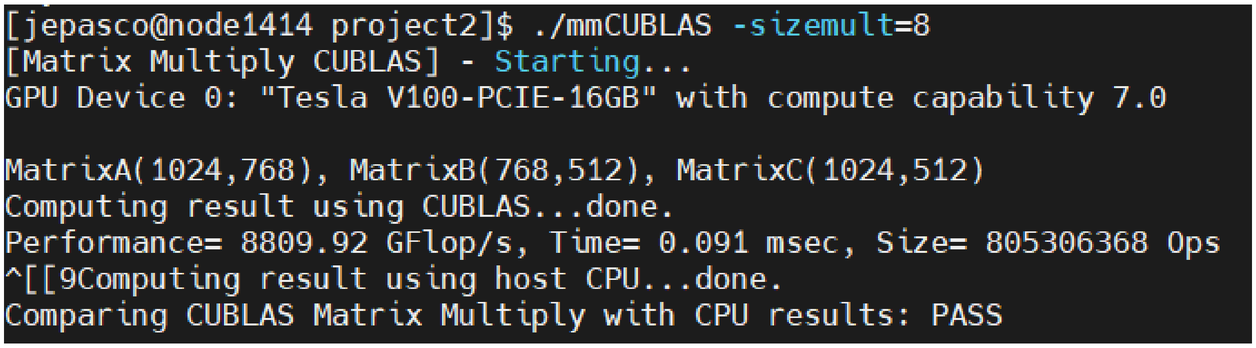
```
[jepasco@node1414 project2]$ ./mmCUBLAS -sizemult=3
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-16GB" with compute capability 7.0

MatrixA(384,288), MatrixB(288,192), MatrixC(384,192)
Computing result using CUBLAS...done.
Performance= 3758.44 GFlop/s, Time= 0.011 msec, Size= 42467328 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
```

  - This image shows the output from mmCUBLAS using the -sizemult parameter set to 3. It took 0.011 msecs to perform the execution.

- 

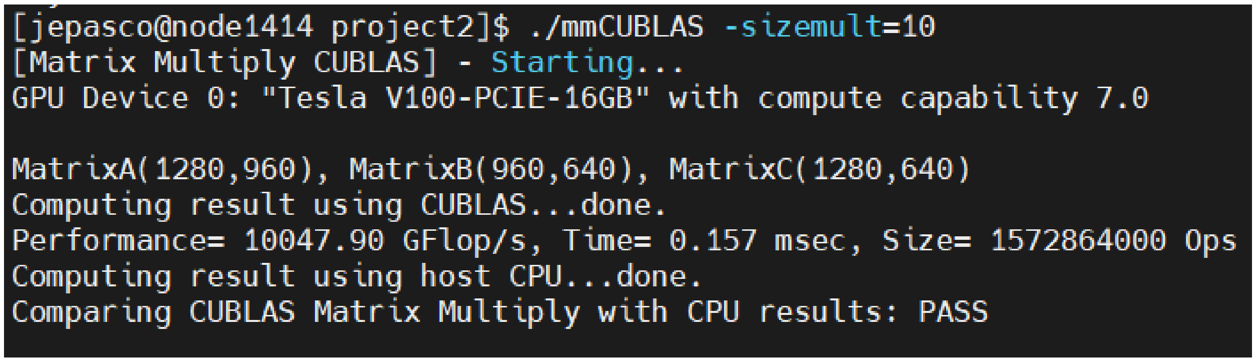
```
[jepasco@node1414 project2]$ ./mmCUBLAS -sizemult=6
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-16GB" with compute capability 7.0

MatrixA(768,576), MatrixB(576,384), MatrixC(768,384)
Computing result using CUBLAS...done.
Performance= 8839.50 GFlop/s, Time= 0.038 msec, Size= 339738624 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
```

  - This image shows the output from mmCUBLAS using the -sizemult parameter set to 6. It took 0.038 msecs to perform the execution.

- 

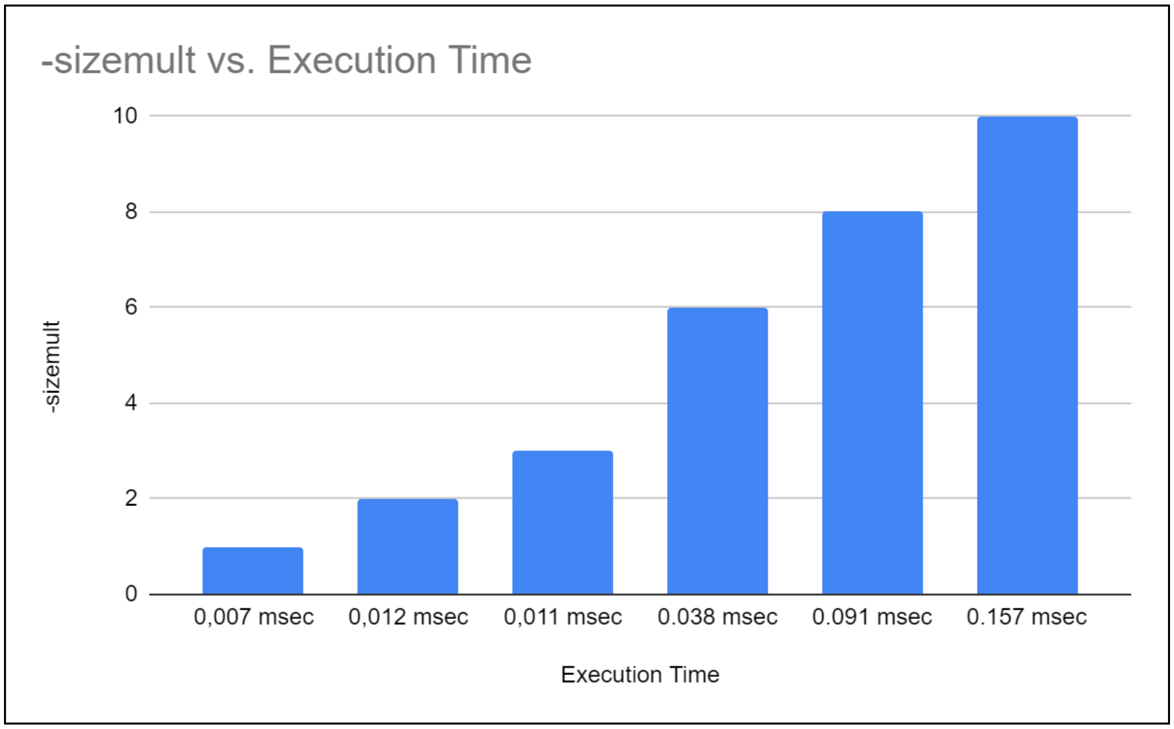
```
[jepasco@node1414 project2]$ ./mmCUBLAS -sizemult=8
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-16GB" with compute capability 7.0

MatrixA(1024,768), MatrixB(768,512), MatrixC(1024,512)
Computing result using CUBLAS...done.
Performance= 8809.92 GFlop/s, Time= 0.091 msec, Size= 805306368 Ops
^[[9Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
```

- This image shows the output from mmCUBLAS using the -sizemult parameter set to 8. It took 0.091 msecs to perform the execution.

- 

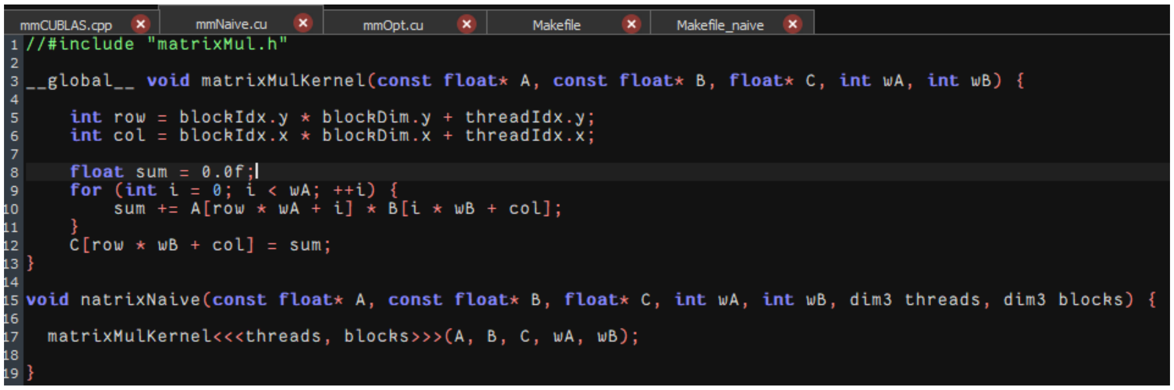
```
[jepasco@node1414 project2]$ ./mmCUBLAS -sizemult=10
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-16GB" with compute capability 7.0

MatrixA(1280,960), MatrixB(960,640), MatrixC(1280,640)
Computing result using CUBLAS...done.
Performance= 10047.90 GFlop/s, Time= 0.157 msec, Size= 1572864000 Ops
Computing result using host CPU...done.
Comparing CUBLAS Matrix Multiply with CPU results: PASS
```

  - This image shows the output from mmCUBLAS using the -sizemult parameter set to 10. It took 0.157 msecs to perform the execution.

- 

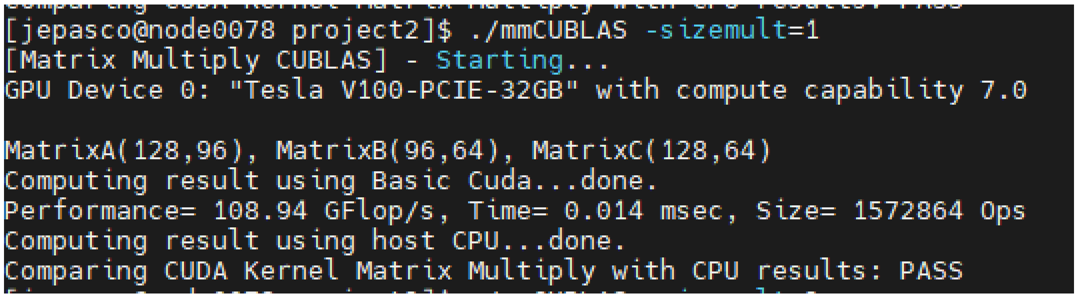
  - This bar graph shows the differences in execution time as the -sizemult parameter value increases which indicates larger matrices to multiply.

<u>Part 2: Implement Matrix Multiplication w/ basic Cuda</u>

- 

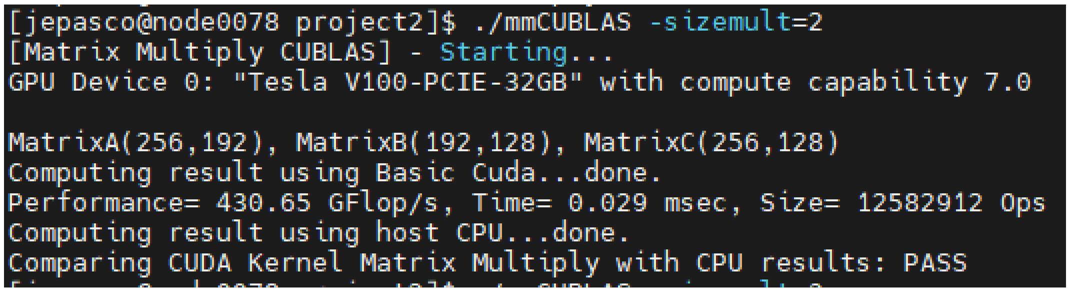
```
//#include "matrixMul.h"

__global__ void matrixMulKernel(const float* A, const float* B, float* C, int wA, int wB) {

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0.0f;
    for (int i = 0; i < wA; ++i) {
        sum += A[row * wA + i] * B[i * wB + col];
    }
    C[row * wB + col] = sum;
}

void natrixNaive(const float* A, const float* B, float* C, int wA, int wB, dim3 threads, dim3 blocks) {

  matrixMulKernel<<<threads, blocks>>>(A, B, C, wA, wB);

}
```

  - This image shows the code for the matrix multiplication function used by mmCUBLAS.cpp. The function is in mmNaive.cu.

- 

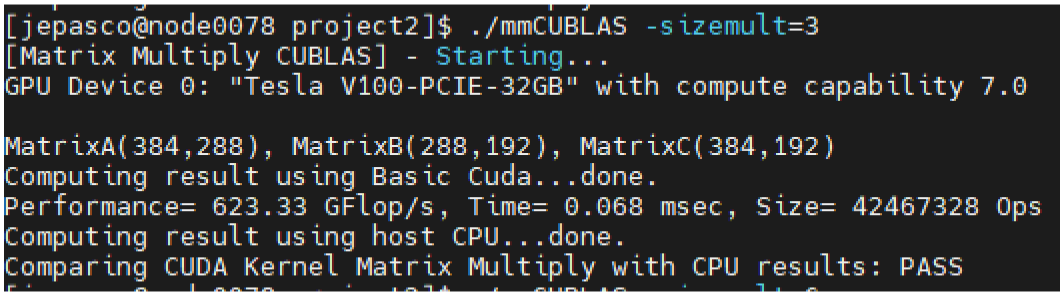
```
[jepasco@node0078 project2]$ ./mmCUBLAS -sizemult=1
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-32GB" with compute capability 7.0

MatrixA(128,96), MatrixB(96,64), MatrixC(128,64)
Computing result using Basic Cuda...done.
Performance= 108.94 GFlop/s, Time= 0.014 msec, Size= 1572864 Ops
Computing result using host CPU...done.
Comparing CUDA Kernel Matrix Multiply with CPU results: PASS
```

  - This image shows the output from mmCUBLAS using the -sizemult parameter set to 1 and the mmNaive.cu matrix multiply function. It took 0.014 msecs to perform the execution.

- 

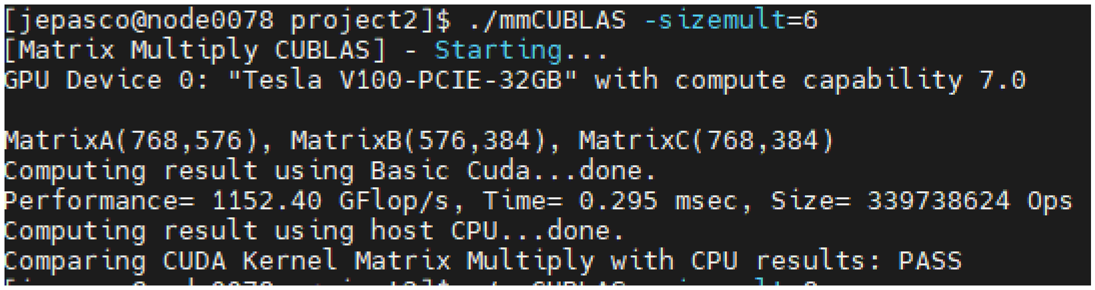
```
[jepasco@node0078 project2]$ ./mmCUBLAS -sizemult=2
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-32GB" with compute capability 7.0

MatrixA(256,192), MatrixB(192,128), MatrixC(256,128)
Computing result using Basic Cuda...done.
Performance= 430.65 GFlop/s, Time= 0.029 msec, Size= 12582912 Ops
Computing result using host CPU...done.
Comparing CUDA Kernel Matrix Multiply with CPU results: PASS
```

  - This image shows the output from mmCUBLAS using the -sizemult parameter set to 2 and the mmNaive.cu matrix multiply function. It took 0.029 msecs to perform the execution.

- 

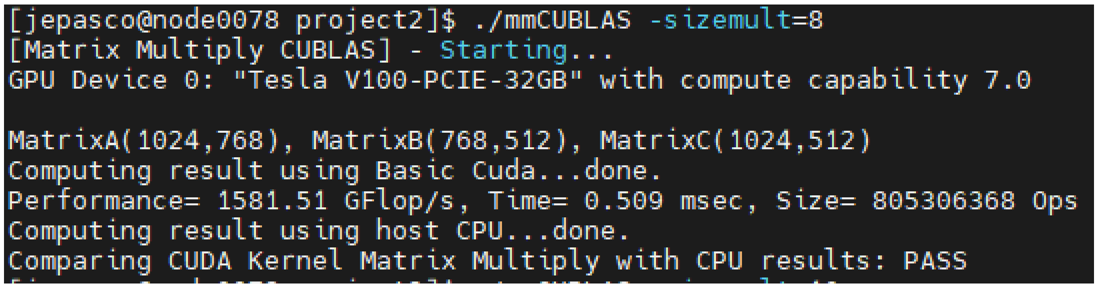
```
[jepasco@node0078 project2]$ ./mmCUBLAS -sizemult=3
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-32GB" with compute capability 7.0

MatrixA(384,288), MatrixB(288,192), MatrixC(384,192)
Computing result using Basic Cuda...done.
Performance= 623.33 GFlop/s, Time= 0.068 msec, Size= 42467328 Ops
Computing result using host CPU...done.
Comparing CUDA Kernel Matrix Multiply with CPU results: PASS
```

  - This image shows the output from mmCUBLAS using the -sizemult parameter set to 3 and the mmNaive.cu matrix multiply function. It took 0.068 msecs to perform the execution

- 

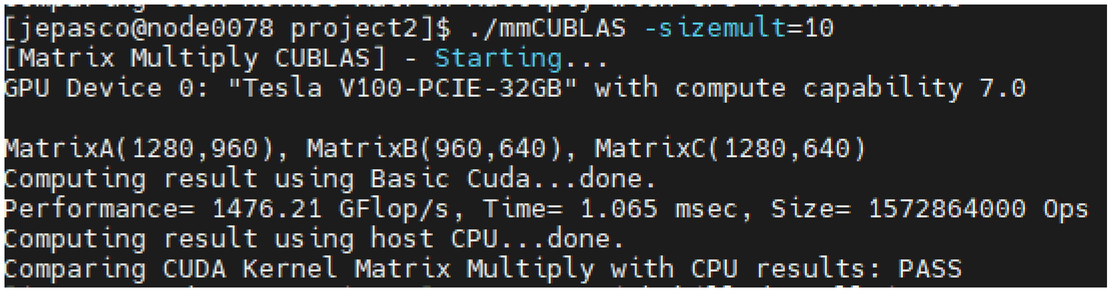
```
[jepasco@node0078 project2]$ ./mmCUBLAS -sizemult=6
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-32GB" with compute capability 7.0

MatrixA(768,576), MatrixB(576,384), MatrixC(768,384)
Computing result using Basic Cuda...done.
Performance= 1152.40 GFlop/s, Time= 0.295 msec, Size= 339738624 Ops
Computing result using host CPU...done.
Comparing CUDA Kernel Matrix Multiply with CPU results: PASS
```

  - This image shows the output from mmCUBLAS using the -sizemult parameter set to 6 and the mmNaive.cu matrix multiply function. It took 0.295 msecs to perform the execution

- 

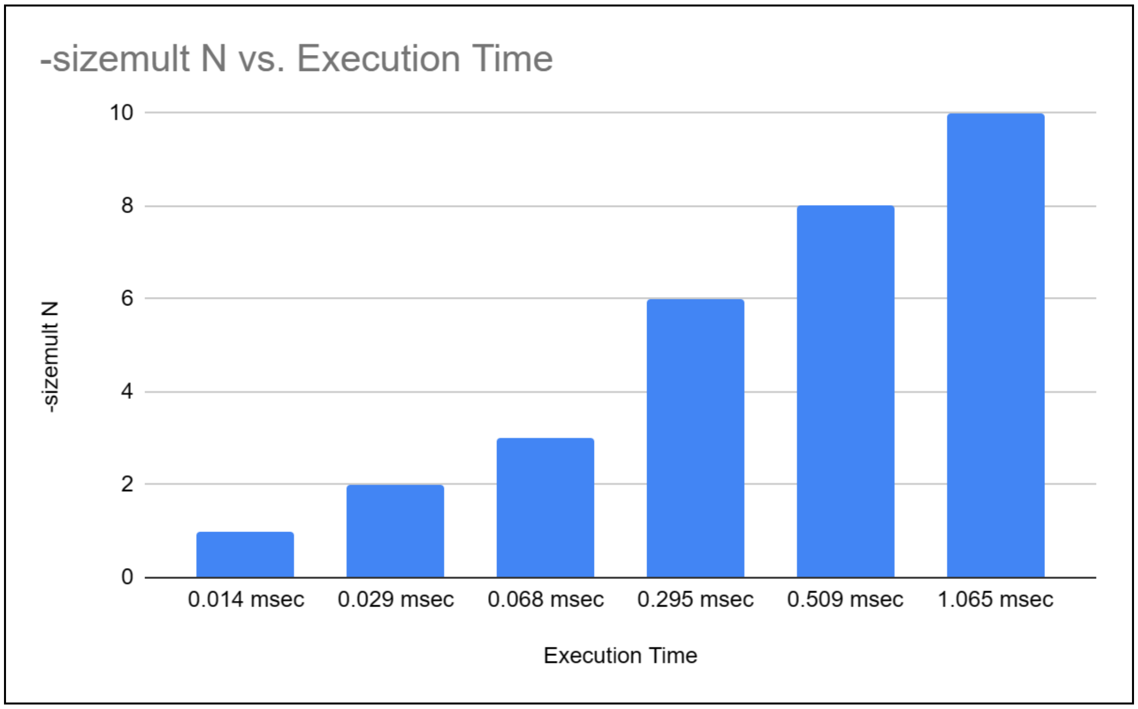
```
[jepasco@node0078 project2]$ ./mmCUBLAS -sizemult=8
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-32GB" with compute capability 7.0

MatrixA(1024,768), MatrixB(768,512), MatrixC(1024,512)
Computing result using Basic Cuda...done.
Performance= 1581.51 GFlop/s, Time= 0.509 msec, Size= 805306368 Ops
Computing result using host CPU...done.
Comparing CUDA Kernel Matrix Multiply with CPU results: PASS
```

  - This image shows the output from mmCUBLAS using the -sizemult parameter set to 8 and the mmNaive.cu matrix multiply function. It took 0.509 msecs to perform the execution

- 

```
[jepasco@node0078 project2]$ ./mmCUBLAS -sizemult=10
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-32GB" with compute capability 7.0

MatrixA(1280,960), MatrixB(960,640), MatrixC(1280,640)
Computing result using Basic Cuda...done.
Performance= 1476.21 GFlop/s, Time= 1.065 msec, Size= 1572864000 Ops
Computing result using host CPU...done.
Comparing CUDA Kernel Matrix Multiply with CPU results: PASS
```

  - This image shows the output from mmCUBLAS using the -sizemult parameter set to 10 and the mmNaive.cu matrix multiply function. It took 1.065 msecs to perform the execution

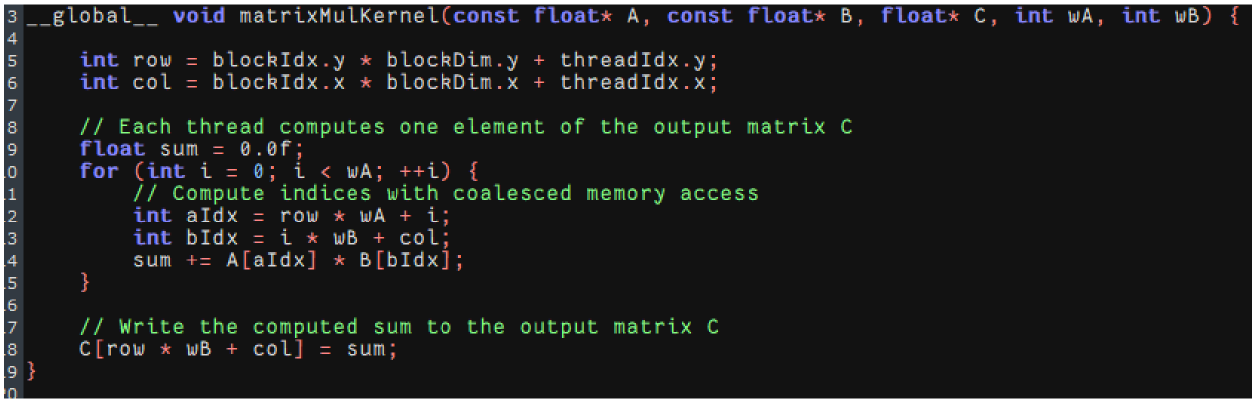
- 

    - This bar graph shows the differences in execution time as the -sizemult parameter value increases which indicates larger matrices to multiply.

-sizemult	ExTime Cublas	ExTime Naive
1	0.007 msec	0.014 msec
2	0.012 msec	0.029 msec
3	0.011 msec	0.068 msec
6	0.038 msec	0.295 msec
8	0.091 msec	0.509 msec
10	0.157 msec	1.065 msec

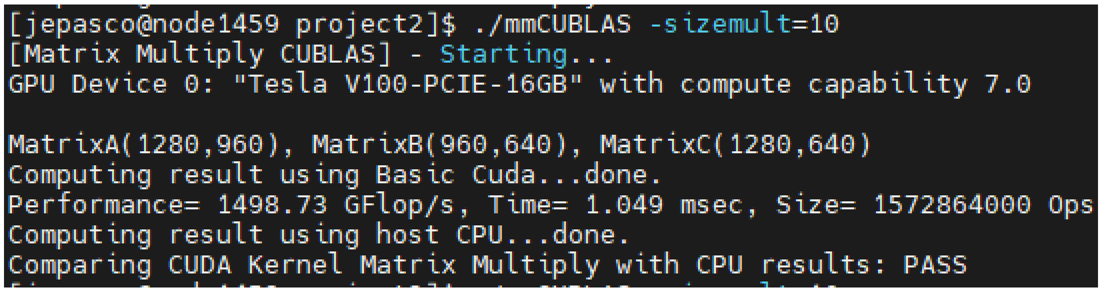
    - This table shows the difference in execution times based on the parameter -sizemult and what function was used. The first column of times utilizes cublas and the second column of times utilizes the naive cuda approach shown above.

Part 3: Optimize You Matrix Multiplication

- 

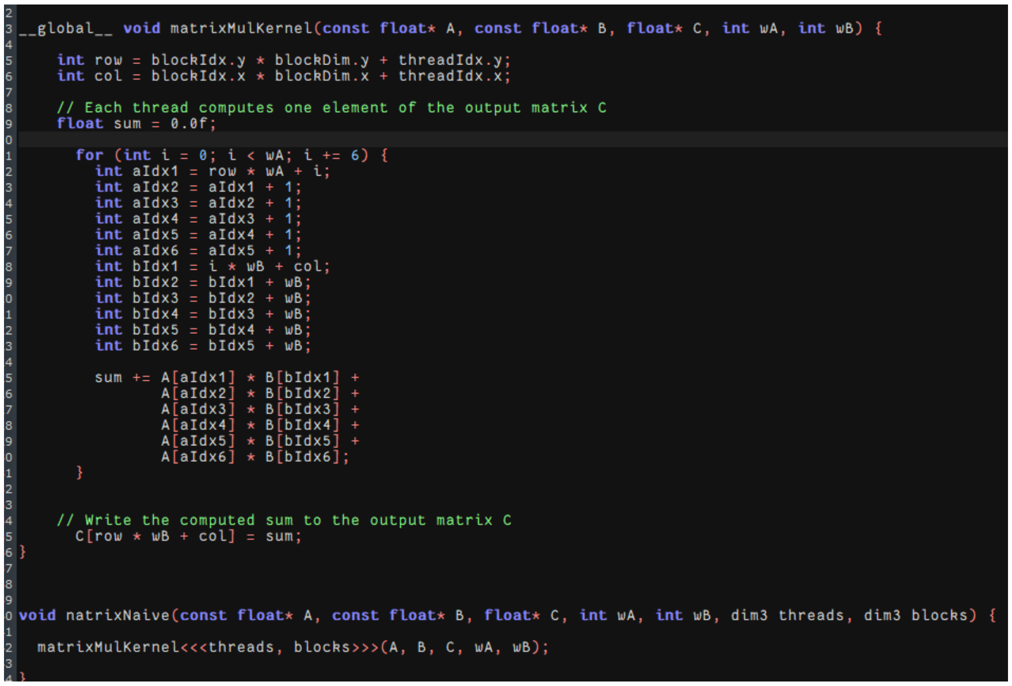
```
__global__ void matrixMulKernel(const float* A, const float* B, float* C, int wA, int wB) {

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Each thread computes one element of the output matrix C
    float sum = 0.0f;
    for (int i = 0; i < wA; ++i) {
        // Compute indices with coalesced memory access
        int aIdx = row * wA + i;
        int bIdx = i * wB + col;
        sum += A[aIdx] * B[bIdx];
    }

    // Write the computed sum to the output matrix C
    C[row * wB + col] = sum;
}
```

    - This image shows the mmOpt.cu code for matrix multiplication with coalescing memory utilized to optimize the function.

- 

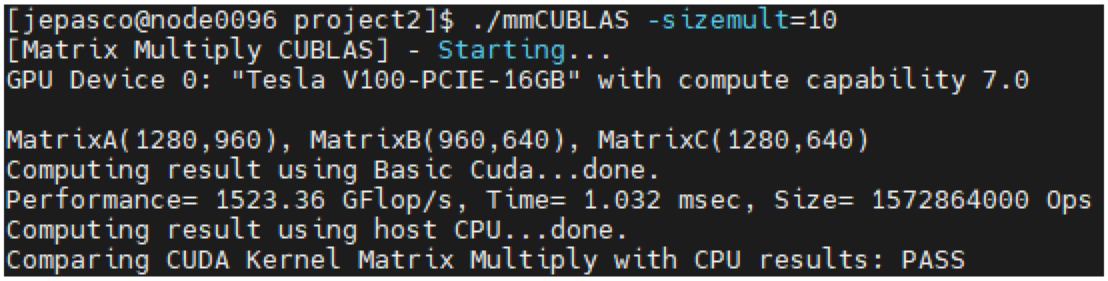
```
[jepasco@node1459 project2]$ ./mmCUBLAS -sizemult=10
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-16GB" with compute capability 7.0

MatrixA(1280,960), MatrixB(960,640), MatrixC(1280,640)
Computing result using Basic Cuda...done.
Performance= 1498.73 GFlop/s, Time= 1.049 msec, Size= 1572864000 Ops
Computing result using host CPU...done.
Comparing CUDA Kernel Matrix Multiply with CPU results: PASS
```

    - This image shows the output from mmCUBLAS using the -sizemult parameter set to 10 and the mmOpt.cu matrix multiply function with memory coalescing optimization. It took 1.065 msecs to perform the execution.

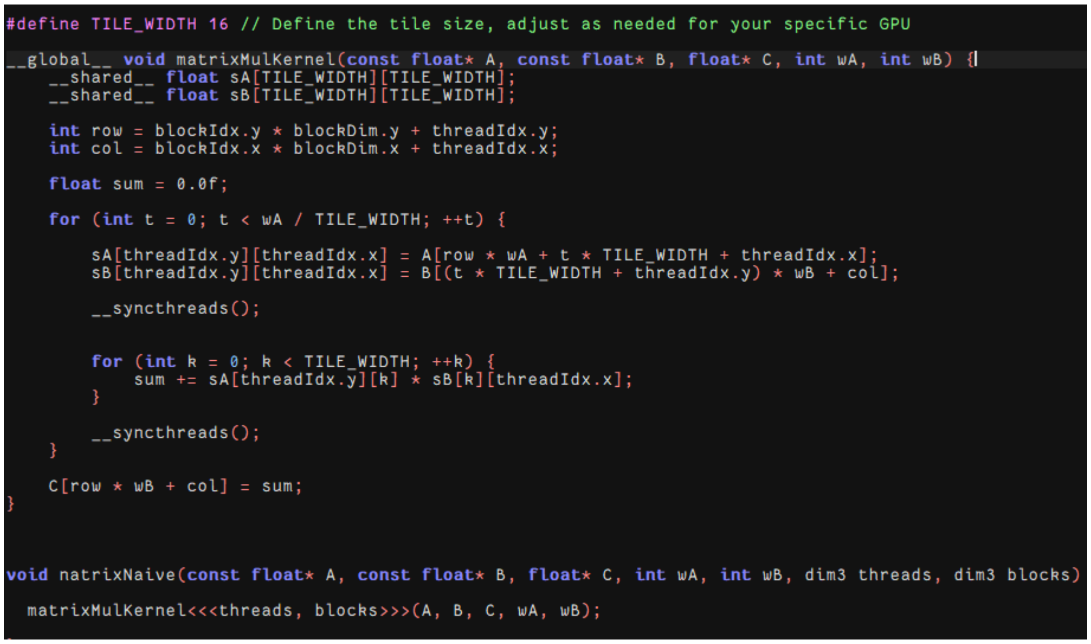
- 

```
__global__ void matrixMulKernel(const float* A, const float* B, float* C, int wA, int wB) {

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Each thread computes one element of the output matrix C
    float sum = 0.0f;

      for (int i = 0; i < wA; i += 6) {
        int aIdx1 = row * wA + i;
        int aIdx2 = aIdx1 + 1;
        int aIdx3 = aIdx2 + 1;
        int aIdx4 = aIdx3 + 1;
        int aIdx5 = aIdx4 + 1;
        int aIdx6 = aIdx5 + 1;
        int bIdx1 = i * wB + col;
        int bIdx2 = bIdx1 + wB;
        int bIdx3 = bIdx2 + wB;
        int bIdx4 = bIdx3 + wB;
        int bIdx5 = bIdx4 + wB;
        int bIdx6 = bIdx5 + wB;

        sum += A[aIdx1] * B[bIdx1] +
               A[aIdx2] * B[bIdx2] +
               A[aIdx3] * B[bIdx3] +
               A[aIdx4] * B[bIdx4] +
               A[aIdx5] * B[bIdx5] +
               A[aIdx6] * B[bIdx6];
      }


    // Write the computed sum to the output matrix C
      C[row * wB + col] = sum;
}



void natrixNaive(const float* A, const float* B, float* C, int wA, int wB, dim3 threads, dim3 blocks) {

  matrixMulKernel<<<threads, blocks>>>(A, B, C, wA, wB);

}
```

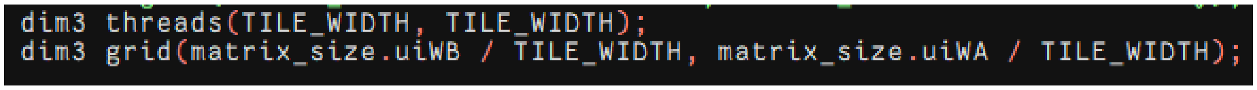
    - This image shows the mmOpt.cu code with coalescing memory and loop unrolling to optimize the function.

- 

```
[jepasco@node0096 project2]$ ./mmCUBLAS -sizemult=10
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-16GB" with compute capability 7.0

MatrixA(1280,960), MatrixB(960,640), MatrixC(1280,640)
Computing result using Basic Cuda...done.
Performance= 1523.36 GFlop/s, Time= 1.032 msec, Size= 1572864000 Ops
Computing result using host CPU...done.
Comparing CUDA Kernel Matrix Multiply with CPU results: PASS
```

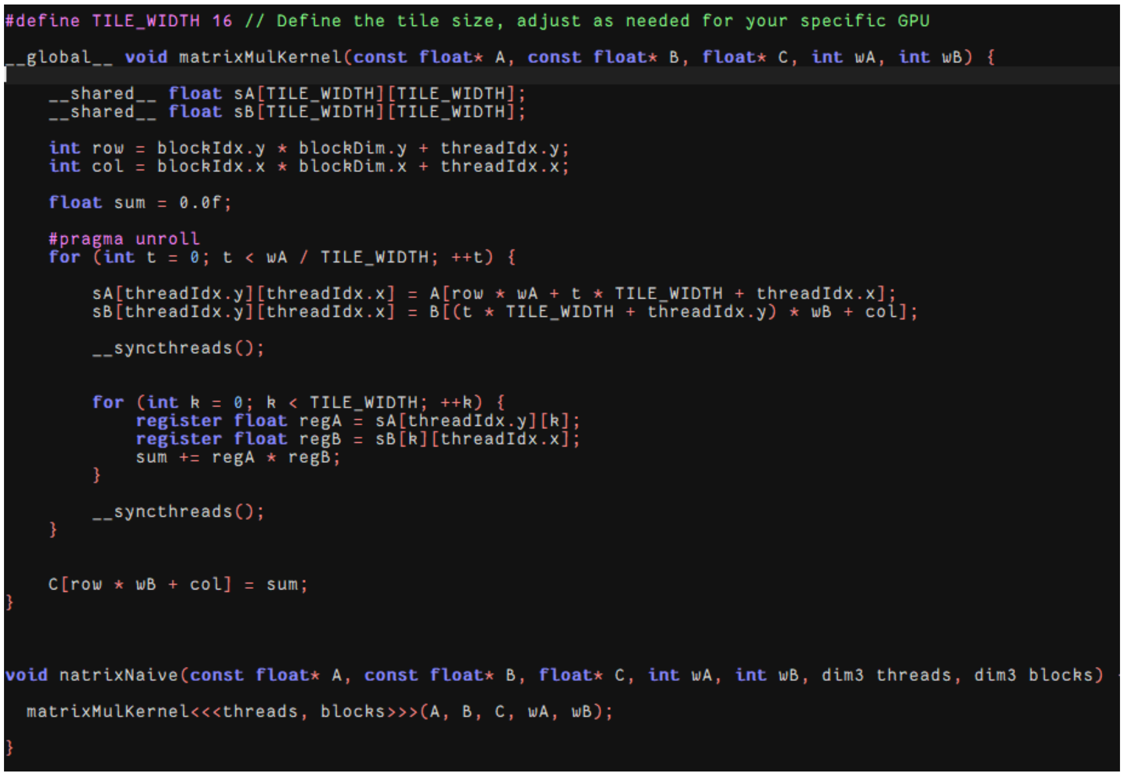
    - This image shows the output from mmCUBLAS using the -sizemult parameter set to 10 and the mmOpt.cu matrix multiply function with memory coalescing and loop unrolling optimization. It took 1.032 msecs to perform the execution.
- 

```
#define TILE_WIDTH 16 // Define the tile size, adjust as needed for your specific GPU

__global__ void matrixMulKernel(const float* A, const float* B, float* C, int wA, int wB) {
    __shared__ float sA[TILE_WIDTH][TILE_WIDTH];
    __shared__ float sB[TILE_WIDTH][TILE_WIDTH];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0.0f;

    for (int t = 0; t < wA / TILE_WIDTH; ++t) {

        sA[threadIdx.y][threadIdx.x] = A[row * wA + t * TILE_WIDTH + threadIdx.x];
        sB[threadIdx.y][threadIdx.x] = B[(t * TILE_WIDTH + threadIdx.y) * wB + col];

        __syncthreads();


        for (int k = 0; k < TILE_WIDTH; ++k) {
            sum += sA[threadIdx.y][k] * sB[k][threadIdx.x];
        }

        __syncthreads();
    }

    C[row * wB + col] = sum;
}



void natrixNaive(const float* A, const float* B, float* C, int wA, int wB, dim3 threads, dim3 blocks)

  matrixMulKernel<<<threads, blocks>>>(A, B, C, wA, wB);
```

    - This image shows the mmOpt.cu code with shared memory and tiling added to optimize the function.
- 

```
 dim3 threads(TILE_WIDTH, TILE_WIDTH);
 dim3 grid(matrix_size.uiWB / TILE_WIDTH, matrix_size.uiWA / TILE_WIDTH);
```

    - This image shows the slight change to mmCUBLAS.cpp to account for using tiling as an optimization technique.

- 

```
#define TILE_WIDTH 16 // Define the tile size, adjust as needed for your specific GPU
__global__ void matrixMulKernel(const float* A, const float* B, float* C, int wA, int wB) {

    __shared__ float sA[TILE_WIDTH][TILE_WIDTH];
    __shared__ float sB[TILE_WIDTH][TILE_WIDTH];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float sum = 0.0f;

    #pragma unroll
    for (int t = 0; t < wA / TILE_WIDTH; ++t) {

        sA[threadIdx.y][threadIdx.x] = A[row * wA + t * TILE_WIDTH + threadIdx.x];
        sB[threadIdx.y][threadIdx.x] = B[(t * TILE_WIDTH + threadIdx.y) * wB + col];

        __syncthreads();


        for (int k = 0; k < TILE_WIDTH; ++k) {
            register float regA = sA[threadIdx.y][k];
            register float regB = sB[k][threadIdx.x];
            sum += regA * regB;
        }

        __syncthreads();
    }


    C[row * wB + col] = sum;
}



void natrixNaive(const float* A, const float* B, float* C, int wA, int wB, dim3 threads, dim3 blocks)

  matrixMulKernel<<<threads, blocks>>>(A, B, C, wA, wB);

}
```

    - This image shows the mmOpt.cu code with thread divergence reduction used to further optimize the function.

3: Discussion

<u>Part 1: Compute Matrix Multiplicaiton w/ Cuda Library</u>

The timing for the matrix multiplication commences precisely at the initiation of the multiplication event and ceases immediately after its completion. This approach allows the program to eliminate any extraneous computations like the calculation of the CPU multiplication so that it focused solely on the core matrix multiplication process.

Various execution times were collected for varying values of the -sizemult parameter. As the parameter increased, so did the size of the matrices that were multiplied. As the matrices got larger, the execution time of the program also dramatically increased.

-sizemult	Execution Time
1	0.007 msec
2	0.012 msec
3	0.011 msec
6	0.038 msec

8	0.091 msec
10	0.157 msec

The above table shows the exact values of the execution times for the program as the -sizemult parameter increased. This trend is reasonable because matrix multiplication is inherently a computationally intensive operation. The time complexity of matrix multiplication is said to be O(n^3), which indicates that the number of operations required increases cubically with the size of the matrices. Therefore, larger matrices demand significantly more computational resources.

Part 2: Implement Matrix Multiplication w/ basic Cuda

As with Part 1, my naive implementation of matrix multiplication with cuda saw dramatic increases in execution time as the matrices became larger. My naive implementation, however, saw much more dramatic increases in execution time than the cublas approach.

-sizemult	ExTime Cublas	ExTime Naive
1	0.007 msec	0.014 msec
2	0.012 msec	0.029 msec
3	0.011 msec	0.068 msec
6	0.038 msec	0.295 msec
8	0.091 msec	0.509 msec
10	0.157 msec	1.065 msec

The above table shows the execution times of both the cublas and naive implementation as the size of the matrices increased. Whereas the cublas implementation had much smaller increments in execution time, my naive implementation was drastically slower, with an ending time almost 8x greater when -sizemult was equal to 10. There are various reasons as to why this is the case. Cublas is a highly optimized library for matrix operations and is able to handle larger matrix sizes more efficiently compared to a naive implementation that follows a basic algorithmic approach with no optimization. Therefore, my naive approach is also worse compared to cublas because it lacks algorithmic efficiency, parallelization, efficient memory access patterns, and hardware acceleration unlike cublas.

Part 3: Optimize You Matrix Multiplication

Memory Coalescing

For this part, I optimized my naive matrix multiplication function with several techniques. The first technique I used was memory coalescing.This technique ensures that global memory accesses are coalesced, meaning that threads access contiguous memory locations. This allows for improved memory bandwidth.

-sizemult	ExTime Cublas	ExTime Naive	ExTime Opt
10	0.157 msec	1.065 msec	1.049 msec

The above table shows that memory coalescing improved the execution time of the maximum size matrices by 0.016 msecs. This is reasonable because improving memory bandwidth will improve execution times as the matrices get larger, but this is still not a great means of improving performance.

Loop Unrolling

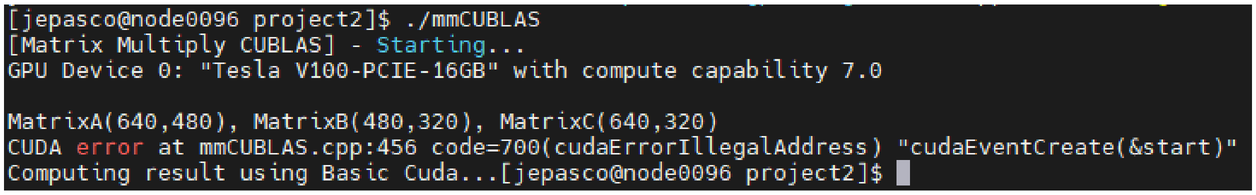
The next technique I used to optimize the function was loop unrolling. This technique reduces loop control overhead, and allows multiple arithmetic operations to be performed simultaneously which maximizes throughput.

-sizemult	ExTime Cublas	ExTime Naive	ExTime Opt
10	0.157 msec	1.065 msec	1.032 msec

The above table shows that loop unrolling slightly improved the execution time of the maximum size matrices by 0.033 msecs from the naive and 0.017 msecs from the previous optimized implementation. This is a reasonable improvement because loop unrolling depends heavily on the architecture of the GPU and the size of the matrices to be most effective. Even then, the improvement expected by this technique is not intended to be large.

Shared Memory & Tiling

The next technique I utilized was shared memory accompanied with tiling. With shared memory, I loaded parts of matrices A and B into shared memory to reduce global memory accesses. The effect of this is that shared memory has a much lower latency compared to global memory. By utilizing shared memory, threads in a block can cooperate and share data. Tiling is also used in this technique to divide the matrices into smaller tiles and load them into shared memory. This technique allows for better utilization of shared memory by breaking down the matrix multiplication into smaller chunks. This allows for reducing loop overhead and improving instruction-level parallelism.

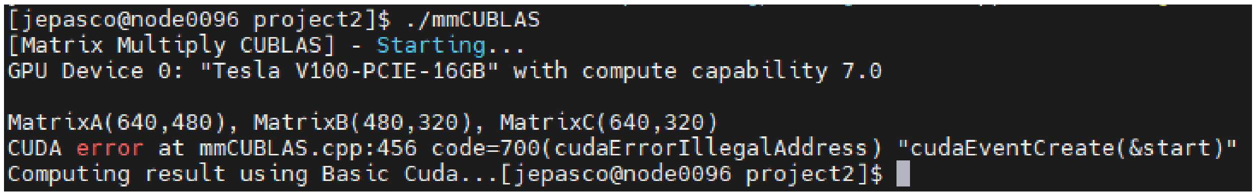


```
[jepasco@node0096 project2]$ ./mmCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-16GB" with compute capability 7.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
CUDA error at mmCUBLAS.cpp:456 code=700(cudaErrorIllegalAddress) "cudaEventCreate(&start)"
Computing result using Basic Cuda...[jepasco@node0096 project2]$
```

I was unable to get my code to run properly after implementing this technique, but the speedup from incorporating shared memory can significantly enhance performance, especially for large matrices. Experimentation with different tile sizes could also lead to further improvements in performance. The code for this implementation can be found under Results.

Thread Divergence Reduction

The final technique I utilized was thread divergence reduction. Thread divergence occurs when threads within a warp take different execution paths. This can significantly impact the performance of the cuda kernel. To optimize the function for this technique, I modified the kernel to ensure that threads within a warp execute the same instructions whenever possible. I did this through loop boundaries, coalesced memory access, register variables,and avoiding bank conflicts.



```
[jepasco@node0096 project2]$ ./mmCUBLAS
[Matrix Multiply CUBLAS] - Starting...
GPU Device 0: "Tesla V100-PCIE-16GB" with compute capability 7.0

MatrixA(640,480), MatrixB(480,320), MatrixC(640,320)
CUDA error at mmCUBLAS.cpp:456 code=700(cudaErrorIllegalAddress) "cudaEventCreate(&start)"
Computing result using Basic Cuda...[jepasco@node0096 project2]$
```

I was also unable to get my code to run properly after implementing this technique, but the speedup from incorporating thread divergence reduction can be very significant when using large matrices. The code for this implementation can be found under Results.

cuBLAS Comparison

Several key differences became apparent when comparing my optimized implementation against the utilization of cuBLAS which led to an observed performance gap between the two despite the incorporation of several optimizations like memory coalescing, loop unrolling, shared memory and tiling, and thread divergence reduction. Given the complexities and specialized nature of cuBLAS, it's common for custom implementations, even with extensive optimizations, to fall short in performance when compared to cuBLAS. Understanding and leveraging the unique optimizations and algorithms embedded within cuBLAS is often the key to achieving optimal performance.