ECE60827 – SP'25 Programming Assign#3

Conor X Devlin, PUID: 0035599728

Part A: Convolutional Neural Networks

For Part A of this assignment, we needed to implement a convolutional kernel in CUDA optimized as much as I could and then present performance results across the various layers. I initially present my finished CPU implementation, Global Memory GPU implementation then Shared Memory implementation, briefly explaining them before presenting performance results. I then close the section discussing my overall strategy and drawbacks and room for further improvements in my implementation(s).

Part A: CPU Implementation

The CPU implementation required finished up the existing CPU code which entirely involved sorting out the indexes in accordance with a 1D array and then choosing an activation function (Figure 2). I've opted for the ReLU to maintain narrative consistency with the original paper.

```
for (uint32_t n = 0; n < oShape.count; ++ n) {
    for (uint32_t m = 0; m < oShape.channels; ++ m) {
        for (uint32_t x = 0; x < oShape.height; ++ x ) {
            for (uint32_t y = 0; y < oShape.width; ++ y) {
                //  For each output fmap value
                //  STUDENT: Set output fmap to bias
                //output[n][m][x][y] = bias[m];
                uint32_t out_idx = ((n * oShape.channels + m) * oShape.height + x) * oShape.width + y;

                output[out_idx] = bias[m];

                for (uint32_t k = 0; k < fShape.channels; ++k) {
                    for (uint32_t i = 0; i < fShape.height; ++i) {
                        for (uint32_t j = 0; j < fShape.width; ++j) {
                            //  STUDENT: Calculate
                            //  O[n][m][x][y] +=
                            //      I[n][k][args.strideH * x][args.strideW * y] *
                            //      W[m][k][i][j];
                            //output[n][m][x][y] += input[n][k][args.strideH * x][args.strideW * y] * filter[m][k][i][k];
                            uint32_t in_h = args.strideH * x + i;
                            uint32_t in_w = args.strideW * y + j;

                            if (in_h < iShape.height && in_w < iShape.width) {
                                uint32_t in_idx = ((n * iShape.channels + k) * iShape.height + in_h) * iShape.width + in_w;
                                uint32_t filter_idx = ((m * fShape.channels + k) * fShape.height + i) * fShape.width + j;
                                //uint32_t filter_idx = ((m * fShape.channels + k) * fShape.height + i) * fShape.width + j;
                                output[out_idx] += input[in_idx] * filter[filter_idx];
                            }
                        }
                    }
                }
```

```
//  STUDENT: Check by disabling activation
//  STUDENT: Apply Activation here
if (args.activation) {
    //  O[n][m][x][y] = Activation( O[n][m][x][y] );
    output[out_idx] = fmaxf(0.0f, output[out_idx]);
}
//
```

Figure 1,2: CPU ConvLayer Function with ReLU Activation

Part A: GPU Global Memory Implementation

The GPU Global Memory (GM) convLayer kernel I've implemented was just a CUDA-*fied* version of the CPU implementation, with ReLU Activation. The purpose of this kernel was to first have a functional CUDA kernel and secondly to have a kernel to compare my *optimized* shared memory implementation against. The code is depicted below in Figure 3 and should look entirely familiar to its CPU cousin. The host-side function call (Figure 7) just involves standard cudaMemCopies and grid and block allocations.

```cpp
__global__ void convLayer_gpu(float* input, TensorShape iShape, float* filter,
    TensorShape fShape, float* bias, float* output, TensorShape oShape, ConvLayerArgs args) {

    /*Coordinates*/
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int channels = blockIdx.z;

    if (col < oShape.width && row < oShape.height) {
        uint32_t out_idx = ((channels * oShape.height + row) * oShape.width + col);
        output[out_idx] = bias[channels];
        //CPU Code For-loop repeat for the filter window//
        for (uint32_t i = 0; i < fShape.height; ++i) {
            for (uint32_t j = 0; j < fShape.width; ++j) {
                for (uint32_t k = 0; k < fShape.channels; ++k) {
                    uint32_t in_h = args.strideH * row + i;
                    uint32_t in_w = args.strideW * col + j;

                    if (in_h < iShape.height && in_w < iShape.width) {
                        uint32_t in_idx = (k * iShape.height + in_h) * iShape.width + in_w;
                        uint32_t filter_idx = ((channels * fShape.channels + k) * fShape.height + i) * fShape.width + j;

                        output[out_idx] += input[in_idx] * filter[filter_idx];
                    }
                }
            }
        }
        if (args.activation) {
            output[out_idx] = fmaxf(0.0f, output[out_idx]);
        }
    }
}
```

Figure 3: GPU Global Memory ConvLayer Function with ReLU Activation

Part A: GPU Shared Memory Implementation

The shared memory (SM) implementation of the kernel was obviously much longer and more involved primarily due to the portion of the code (Figure 5) set aside for cooperative loading of the SM tile. The initial code of the kernel (Figure 4) is also modified to incorporate the SM tile indexing and bounds definition as that was a major stumbling block for myself while coding this, constantly delineating between global coordinates and local shared memory tile coordinates. The cooperative loading takes the thread's index within the local thread block then increments that by the number of threads in the thread block effectively striding across the global memory to load into this shared tile, once again this was the most difficult part for myself (due to shared/global indexing) and where the bulk of my time was spent trying to make sure this cooperative loading loop was functional. Lastly Figure 6 features the bulk of the convolution code which has been modified to consider that we're now operating inside a SM tile vice GM.

```cpp
__global__ void convLayer_gpu_SM_DM_v3(float* input, TensorShape iShape, float* filter,
    TensorShape fShape, float* bias, float* output, TensorShape oShape, ConvLayerArgs args, int TILE_SIZE) {

    extern __shared__ float tile[];

    const int tile_height = ((TILE_SIZE - 1) * args.strideH) + fShape.height;  // 6-1*4+11 = 31
    const int tile_width  = ((TILE_SIZE - 1) * args.strideW) + fShape.width;   // " "        = 31
    const int tile_depth = iShape.channels;                                    //            = 3

    /*Shared Memory Sizing: 31x31x3 = 2,883 elements, 961 elements per channel*/

    int tidx = threadIdx.x;
    int tidy = threadIdx.y;

    int out_x = blockIdx.x * blockDim.x + tidx;
    int out_y = blockIdx.y * blockDim.y + tidy;
    int out_z = blockIdx.z;

    int threadblock_thread_id = tidy * blockDim.x + tidx; // 0:35

    int base_x = blockIdx.x * blockDim.x * args.strideW; // 0:9*6*4 = 0 --> 216
    int base_y = blockIdx.y * blockDim.y * args.strideH; // 0:9*6*4 = 0 --> 216

    /*Load full 31x31 * 3 tile cooperatively*/
    /*First 12 threads load 80 elements per*/

    //int coop_tid = tidy * TILE_SIZE + tidx;
    int total_threads = TILE_SIZE * TILE_SIZE;            // 6*6  = 36
    //int threads_per_channel = total_threads / tile_depth; // 36/3 = 12
    int elements_per_channel = (tile_height * tile_width);
    int total_elements = tile_depth * elements_per_channel;

    if (threadblock_thread_id == 0 && blockIdx.x == 0 && blockIdx.y == 0 && blockIdx.z == 0) {
        printf("Block (0, 0, 0): Expected shared memory = %d bytes\n", total_elements * 4);
    }
```

Figure 4: GPU Shared Memory ConvLayer Function: initial indexing, bounding and variable setting.

```
/*Cooperative Loading*/
for (int idx = threadblock_thread_id; idx < total_elements; idx += total_threads) {
    int c = idx / elements_per_channel; //0:2
    int coop_idx = idx % elements_per_channel;
    int coop_y = coop_idx / tile_width; // row in shared tile (0 to tile_height-1)
    int coop_x = coop_idx % tile_width; // col in shared tile (0 to tile_width-1)

    // Compute the corresponding global input coordinates.
    // Incorporate any padding in the index calculation if needed.
    int input_x = base_x + coop_x; // PAD ?
    int input_y = base_y + coop_y; // PAD ?

    int shared_idx = (c * tile_height * tile_width) + coop_y * tile_width + coop_x;
    //printf("Channel: %d | Input_x: %d | Input_y: %d\n", c, input_x, input_y);
    if (input_x >= 0 && input_x < iShape.width &&
        input_y >= 0 && input_y < iShape.height) {
        int global_idx = (c * iShape.height + input_y) * iShape.width + input_x;
        tile[shared_idx] = input[global_idx];
        //tile[shared_idx] = 1.0f;
    }
    else {
        tile[shared_idx] = 0.0f;  // Handle boundaries (zero padding)

    }

}

__syncthreads();
```

Figure 5: GPU Shared Memory ConvLayer Function: shared memory cooperative loading portion.

```
// Debug: Confirm execution
if (blockIdx.x == 0 && blockIdx.y == 0 && blockIdx.z == 0 && threadblock_thread_id == 0) {
    printf("Block (0, 0, 0): Loading completed\n");
}

// Output computing, somewhat similar to CPU/GM
if (out_x < oShape.width && out_y < oShape.height) {
    uint32_t out_idx = (out_z * oShape.height + out_y) * oShape.width + out_x;
    float shared_sum = bias[out_z];

    for (uint32_t k = 0; k < fShape.channels; ++k) {
        float channel_sum = 0.0f;
        for (uint32_t i = 0; i < fShape.height; ++i) {
            for (uint32_t j = 0; j < fShape.width; ++j) {
                int in_x = (out_x)*args.strideW + j;
                int in_y = out_y * args.strideH + i;
                int tile_x = in_x - base_x;
                int tile_y = in_y - base_y;

                //Bounds check with input shape
                if (in_x < iShape.width && in_y < iShape.height) {
                    uint32_t shared_idx = k * tile_height * tile_width + tile_y * tile_width + tile_x;
                    uint32_t filter_idx = (out_z * fShape.channels + k) * fShape.height * fShape.width + i * fShape.width + j;
                    channel_sum += tile[shared_idx] * filter[filter_idx];
                }
            }
        }
        shared_sum += channel_sum;
    }
    output[out_idx] = shared_sum;
    //Relu again-lu
    if (args.activation) {
        output[out_idx] = fmaxf(0.0f, output[out_idx]);
    }
}
```

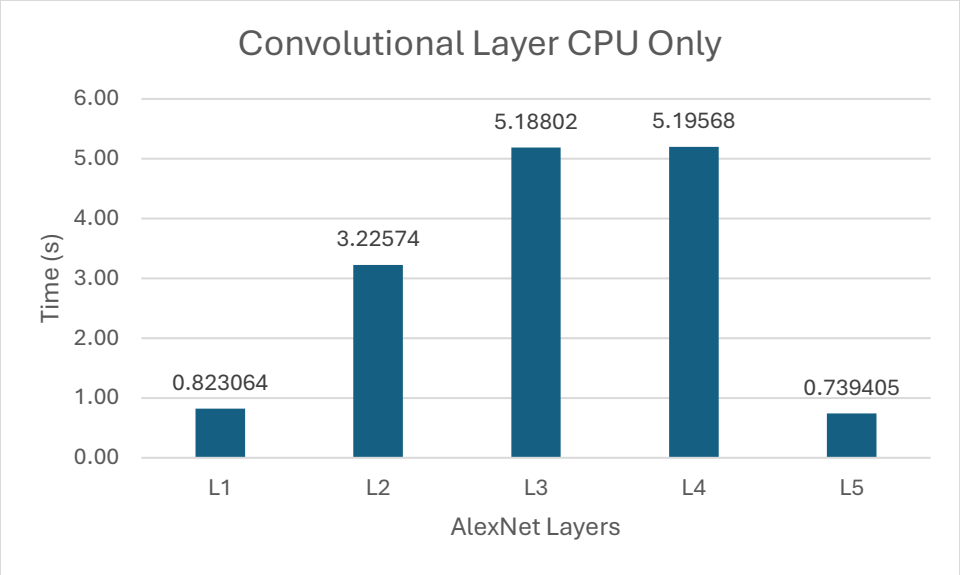
Figure 6: GPU Shared Memory ConvLayer Function: convolution logic.

```
/*Block and Grid Dims*/
dim3 blockDim(TILE_SIZE, TILE_SIZE, 1);
dim3 gridDim((oShape.width + TILE_SIZE - 1) / TILE_SIZE, (oShape.height + TILE_SIZE - 1) / TILE_SIZE, oShape.channels);

int shared_window_height = ((TILE_SIZE - 1) * args.strideH) + fShape.height; // 31
int shared_window_width = ((TILE_SIZE - 1) * args.strideW) + fShape.width;  // 31
size_t sharedMemSize = ((iShape.channels * (shared_window_height) * (shared_window_width))) * sizeof(float); // 4B*(3*31*31) = 11,5KB

/*ConvLayer Kernel Call*/
std::cout << "Memory Size: " << sharedMemSize << " Bytes! \n";

convLayer_gpu_SM_DM_v3 << <gridDim, blockDim, sharedMemSize >> > (d_in, iShape, d_filter, fShape, d_bias, d_out, oShape, args, TILE_SIZE);
```

Figure 7: GPU Shared Memory ConvLayer Function: host-side sizing and logic.

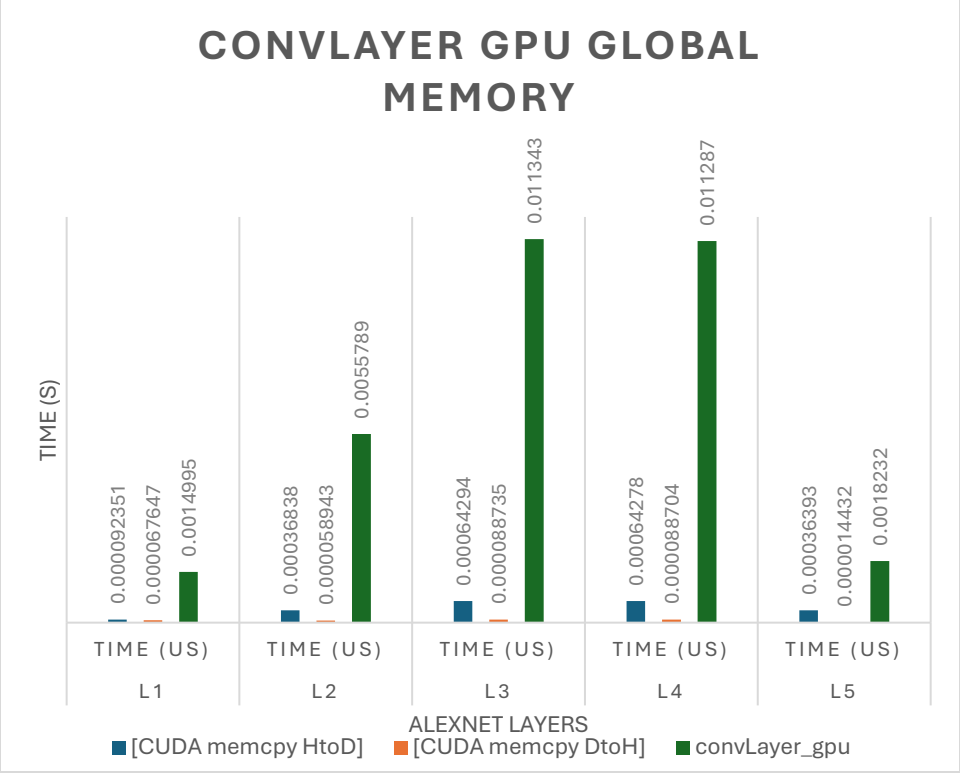
Part A: Performance Results

Graph 1 depicted below is just the time taken by the CPU function on the Purdue GPU Server with it's Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz which is a 12/24 (C/T) processor paired with our V100. As we see in Graph 1 (and repeated throughout the GM and SM implementations) L1 and L5 take the least amount of time due to both the density of the input shape and the filter sizing, they've got FLOPS around 100M while L2-L4 range from 150-450M FLOPs and hence take a larger amount of time. Regardless the CPU's slowness is evident through the function ranging from nearly 1.0 to 5.2 seconds.



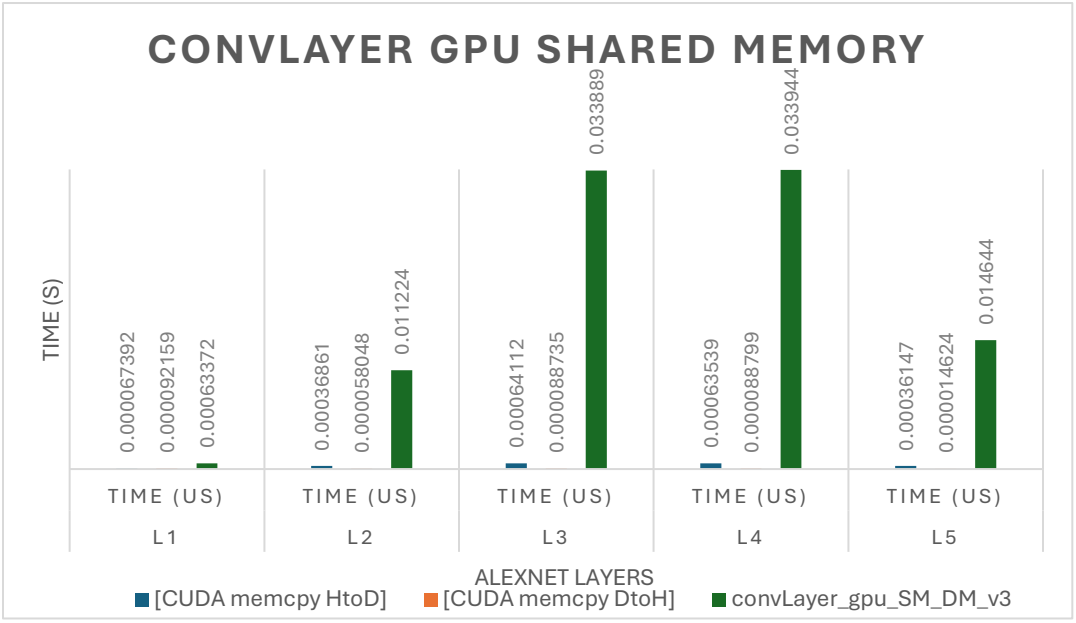
Graph 1: CPU ConvLayer, ranging from AlexNet L1-L5.

In Graph 2 we depict the GM (convLayer_gpu) implementation of ConvLayer and see the same trend from the CPU repeat here as well though notably we obtain a 450x to 550x speedup comparing our kernel (Green) to the CPU function.



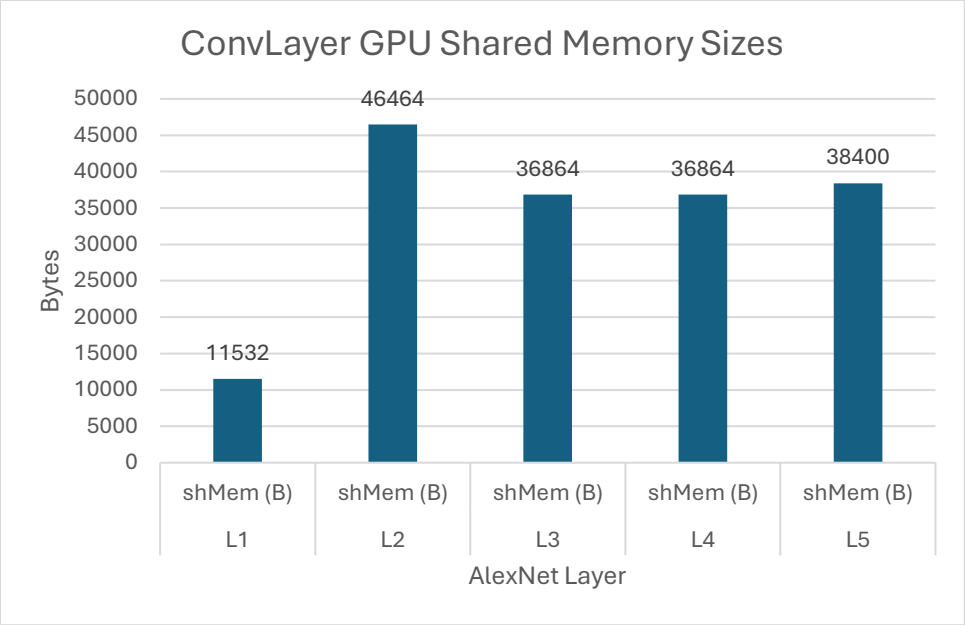
Graph 2: GPU ConvLayer Global Memory - GPU Activities time (s) spent varying AlexNet Layers.

In Graph 3 we depict the SM implementation (convLayer_gpu_SM_DM_v3) of ConvLayer, again seeing the same trend from the CPU and GPU-GM repeat here as well though we find that the SM implementation encounters significant slowdowns across L3-5 while maintaining parity in L2 and actually surpassing in L1. Within L1 our SM obtains a 1,159x speedup over the CPU and 2.11x speedup over the GPU-GM implementations. For L2 we achieve a 288x speedup over the CPU but ultimately lose out to the GM implementation (GM is 2x faster than SM). We repeat this pattern across L3 and L4, though with GM 3x faster than SM and see in L5 that GM is 8x faster than the SM.



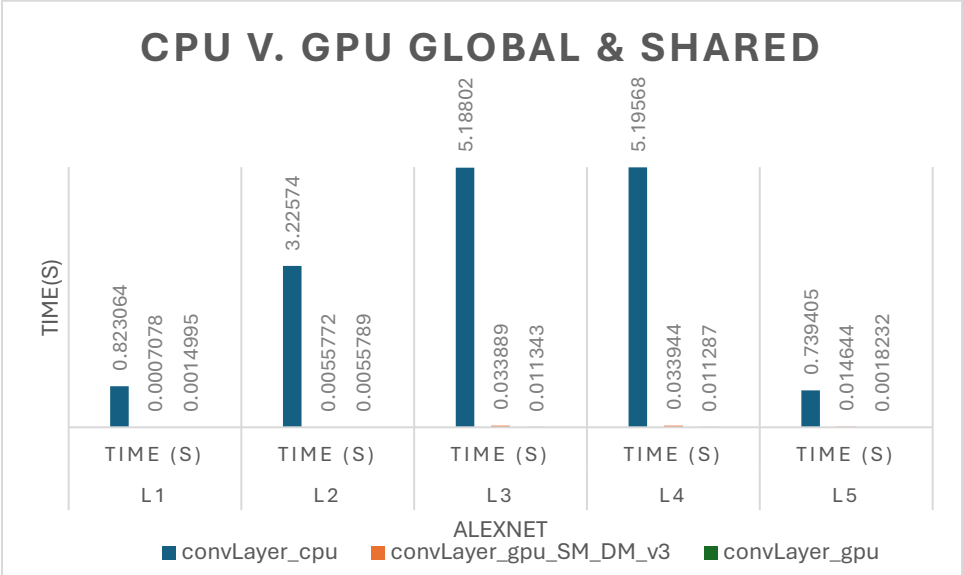
Graph 3: GPU ConvLayer Shared Memory - GPU Activities time (s) spent varying AlexNet Layers.

Graph 4 depicts the shared memory sizing (calculations in figure 7) across the various Alexnet layers and we find that the second layer takes the largest amount of space though its execution time doesn't necessary synchronize with its space demands and then L3-5 are relatively consistent in their SM use. All the data is float-based, 4B, so it was rather straightforward to size the SM needed by the kernel per-layer.



Graph 4: GPU ConvLayer Shared Memory – Shared Memory Sizes (B) varying AlexNet Layers.

Graph 5 simply compares the CPU v SM v GM kernels and their execution times (seconds) we find that our GM implementation beats out my SM and the CPU implementations.



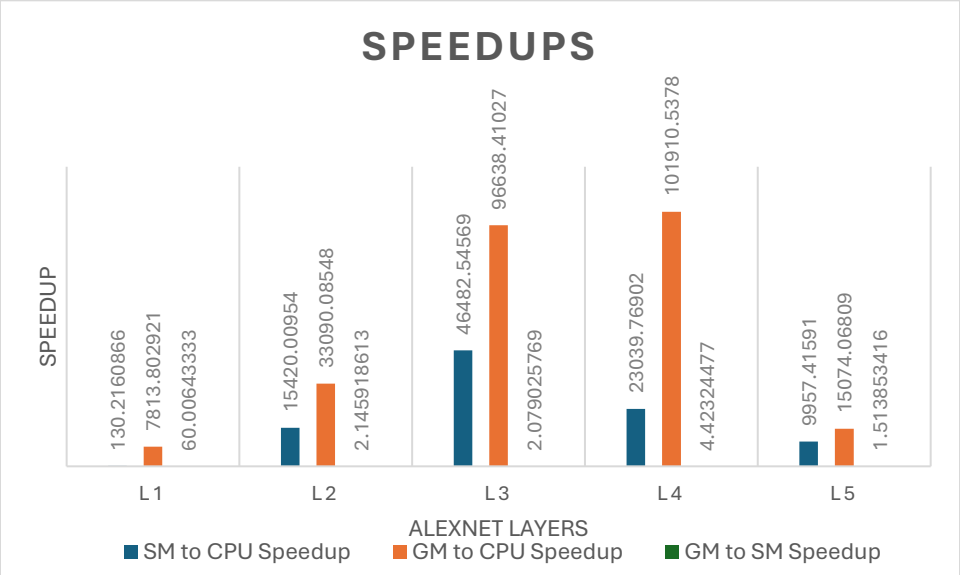
Graph 5: CPU v GPU ConvLayer Global & Shared Memory – Kernel/Function time (s) spent varying AlexNet Layers.

Lastly I re-run the performance metrics locally and due to having a GTX4070 which has compute capability 11.0 (not <8.0 which would enable the nvprof function) and an Intel(R) Core(TM) i7-12700H @ 2.30GHz with 14/20 (C/T). I implemented a basic timer from the *chrono* library as depicted in Figure 8 while I understand this isn't an entirely accurate way of gauging the function I wanted to run it locally to obtain performance and speedup metrics found in Graph 6 and Table 1. Noticeably these speedups are actually much higher than on Purdue's GPU server (likely due to running on a virtualized backup and having to share GPUs and other

infrastructure nuances) but the theme remains: GPU-GM is faster than GPU-SM is faster than CPU.

```
auto start = std::chrono::high_resolution_clock::now();
//convLayer_gpu << <gridDim, blockDim >> > (d_in, iShape, d_filter, fShape, d_bias, d_ou
convLayer_gpu_SM_DM_v3 << <gridDim, blockDim, sharedMemSize >> > (d_in, iShape, d_filter
auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed = end - start;
std::cout << "GPU execution time: " << elapsed.count() << " seconds\n";
```

Figure 8: GPU ConvLayer Kernel calls with simple timer.



Graph 6: CPU v GPU GM and GPU SM Speedups and GPU GM v GPU SM speedup.

	L1	L2	L3	L4	L5
Raw Times	Time (us)	Time (us)	Time (us)	Time (us)	Time (us)
GPU-SM	0.004383890	0.000142592	0.000067691	0.000137846	0.000046825
GPU-GM	0.000073057	0.000066448	0.000032559	0.000031164	0.000030931
CPU	0.570853	2.19877	3.14645	3.17594	0.466256
Speedups	L1	L2	L3	L4	L5
SM to CPU Speedup	130.2160866	15420.00954	46482.54569	23039.76902	9957.41591
GM to CPU Speedup	7813.802921	33090.08548	96638.41027	101910.5378	15074.06809
GM to SM Speedup	60.00643333	2.145918613	2.079025769	4.42324477	1.513853416

Table 1: Full times and speeds up for Graph 6 run locally on my machine.

Part A: Strategy, Drawbacks and Improvement Points

For the SM implementation, my general strategy revolved around the 49KB of memory available (which was particularly evident given that L2 was more memory-bound than anything else). I structured the kernel into two main parts: loading shared memory and executing the actual convolution logic.

In the first part, I used cooperative thread loading but failed to properly utilize memory coalescing. Instead, I employed a striding approach, where global loads were incremented by the number of threads per thread block (which varied depending on shared memory sizing). This was a clear drawback where I could have improved—I didn't implement effective memory coalescing where I should have and missed the opportunity to have each thread load multiple elements. However, doing so would have added complexity to my logic for managing shared memory usage within the kernel.

Additionally, I could have split the kernel into two separate __device__ kernels—one dedicated to loading data and another for convolution. This would have allowed for finer control over performance optimizations. Increasing the number of threads per block would have exacerbated stalls caused by __syncthreads(), as well as the global memory accesses in the preceding loop, further slowing down my kernel due to the lack of coalescing.

Another potential optimization I tried, but failed at, to explore was filter preloading, which could have been handled in its own __device__ kernel to further minimize global memory reads. My code also requires improvements in dynamic tile logic to ensure that every configuration maximally utilizes shared memory (49KB). Additionally, breaking the kernel into segmented __device__ sub-sections and refactoring the cooperative loading mechanism to use coalesced accesses and filter preloading would reduce global memory accesses significantly.

Ultimately, these inefficiencies explain why my GM implementation was much faster than my SM implementation. I still struggle with effectively using shared memory and ensuring optimal bounding within it, but this experience has highlighted several areas for improvement.

Part B: General Matrix Multiplication (GeMM)

For Part B of this assignment, we needed to implement a GeMM kernel in CUDA optimized as much as I could using both CUDA memory copies and unified virtual memory and then present performance results across the various layers. As outlined by the directions, for the input size, I utilize the fully-connected layer size from the 2nd fully connected of Alexnet and sweep the batch size from 1 to 3. I initially present my finished CPU implementation, 'Basic' memcpy shared memory GPU implementation then 'Speed' memcpy shared memory implementation and finally the unified virtual memory (UVM) implementation, briefly explaining them before presenting performance results. I then close the section discussing my overall strategy and drawbacks and room for further improvements in my implementation(s).

Part B: CPU Implementation

The CPU implementation remained largely the same save for my modification of the original function to support batch sizes greater than one. The structural change was another for-loop specifically for the batch and then a batch offset for each of the matrices: A, B and C, which involved the batch multiplied with the matrices' count, height and width. The code is depicted below in Figure 8.

```
int gemmLayer_cpu_batchsize(float* a, TensorShape aShape,
    float* b, TensorShape bShape,
    float* c, TensorShape& cShape,
    GemmLayerArgs& args) {

    int tilesAlongW = (cShape.width + args.tileW - 1) / args.tileW;
    int tilesAlongH = (cShape.height + args.tileH - 1) / args.tileH;
    int subTilesAlongK = (aShape.width + args.tileH - 1) / args.tileH;

    // Loop over batches
    for (uint32_t batch = 0; batch < cShape.count; ++batch) {
        int batchOffsetA = batch * aShape.channels * aShape.height * aShape.width;
        int batchOffsetB = (bShape.count == 1) ? 0 : batch * bShape.channels * bShape.height * bShape.width;
        int batchOffsetC = batch * cShape.channels * cShape.height * cShape.width;

        int tileId = 0;
        while (tileId < tilesAlongW * tilesAlongH) {
            int offsetH = (tileId / tilesAlongW) * args.tileH;
            int offsetW = (tileId % tilesAlongW) * args.tileW;
            int rowIdx, colIdx;

            for (int subTile = 0; subTile < subTilesAlongK; ++subTile) {
                for (int row = 0; row < args.tileH; ++row) {
                    for (int col = 0; col < args.tileW; ++col) {
                        rowIdx = row + offsetH;
                        colIdx = col + offsetW;

                        if (rowIdx < cShape.height && colIdx < cShape.width) {
                            if (subTile == 0)
                                c[batchOffsetC + IDX2R(rowIdx, colIdx, cShape.width)] = 0;

                            for (int subTileK = 0; subTileK < args.tileH; ++subTileK) {
                                int k = subTile * args.tileH + subTileK;
                                if (k < aShape.width) {
                                    c[batchOffsetC + IDX2R(rowIdx, colIdx, cShape.width)] +=
                                        a[batchOffsetA + IDX2R(rowIdx, k, aShape.width)] *
                                        b[batchOffsetB + IDX2R(k, colIdx, bShape.width)];
                                }
```

Figure 8: gemmLayer_cpu with Batch Size functionality!

Part B: 'Basic' Memcpy Implementation

Compared to the CPU implementation my initial hack at the GPU implementation by way of the 'Basic' memcpy implementation is effectively a modified version of the textbook's (*Programming Massively Parallel Processors 3rd Ed.)* with 2D shared Arrays (Mds, Nds). We make changes in the initial declarations to support this batch-based GeMM model and then have a single global bounds check before heading into a single for-loop that has each thread iterate over a selection of value based on a globally defined GEMM_TILE_SIZE (32 appear to be the most optimal and least likely to cause the kernel to fail due to memory). Mds is set with the

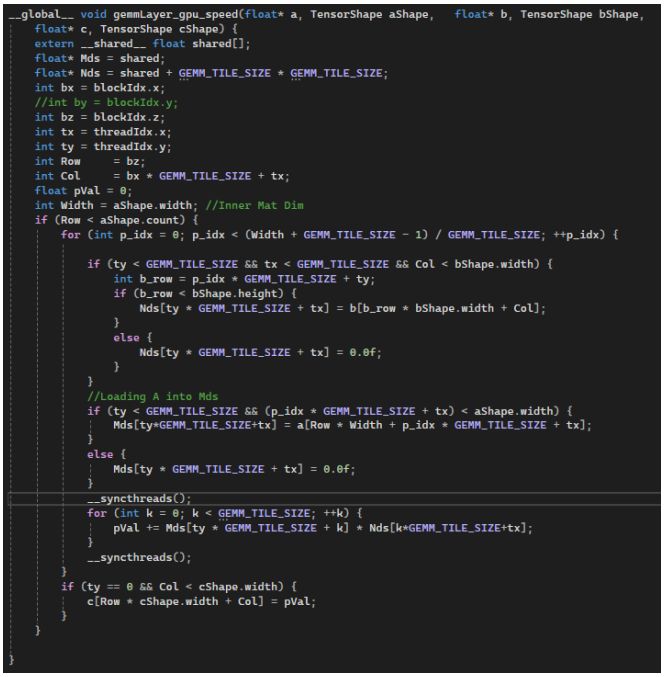
source A matrix and Nds with B using ty and tx as the y-x, x-y respectively. Then once the indices are set theirs a __syncthread() and then a for-loop to determine the local value, *pVal*, and then inevitably assign that *pVal* to the output C matrix.

```
__global__ void gemmLayer_gpu(float* a, TensorShape aShape, float* b, TensorShape bShape,
    float* c, TensorShape cShape) {
    __shared__ float Mds[GEMM_TILE_SIZE][GEMM_TILE_SIZE];
    __shared__ float Nds[GEMM_TILE_SIZE][GEMM_TILE_SIZE];
    int bx = blockIdx.x;
    //int by = blockIdx.y;
    int bz = blockIdx.z; //Batching idx 0:2
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int Row = bz;
    int Col = bx * GEMM_TILE_SIZE + tx;
    float pVal = 0;
    int Width = aShape.width; //Inner Mat Dim
    if (Row < aShape.count) {
        for (int p_idx = 0; p_idx < (Width + GEMM_TILE_SIZE - 1) / GEMM_TILE_SIZE; ++p_idx) {
            //Loading A into Mds
            if (ty < GEMM_TILE_SIZE && (p_idx * GEMM_TILE_SIZE + tx) < aShape.width) {
                Mds[ty][tx] = a[Row * Width + p_idx * GEMM_TILE_SIZE + tx];
            }
            else {
                Mds[ty][tx] = 0.0f;
            }
            //Loading B into Nds
            if (Col < bShape.width && (p_idx * GEMM_TILE_SIZE + ty) < bShape.height) {
                Nds[ty][tx] = b[(p_idx * GEMM_TILE_SIZE + ty) * bShape.width + Col];
            }
            else {
                Nds[ty][tx] = 0.0f;
            }
            __syncthreads();
            for (int k = 0; k < GEMM_TILE_SIZE; ++k) {
                pVal += Mds[ty][k] * Nds[k][tx];
            }
            __syncthreads();
        }
        if (ty == 0 && Col < cShape.width) {
            c[Row * cShape.width + Col] = pVal;
        }
    }
```

Figure 9: 'Basic' Copy gemmLayer_gpu kernel

Part B: 'Speed' Memcpy Implementation

The *gemmLayer_gpu_speed* is an alternative to the 'basic' iteration of a copy-based gemm kernel in Figure 9 and instead of utilizing the 2D shared dual SM arrays opts instead for a traditional 1D shared array that I'm much more familiar with. I Load the initial part of the input (A) into Mds and then load the second portion (B) into Nds giving it part of the SM array but with the offset of the tile area to account for the initial part of the *shared* array being loaded into Mds. The rest of the function proceeds similarly to the above but accounts for the fact that we're now operating with a 1D array. The designation of *'speed'* is a slight misnomer and it's honestly better to differentiate *'basic'* and *'speed'* as *'SM-2D'* and *'SM-1D'*.

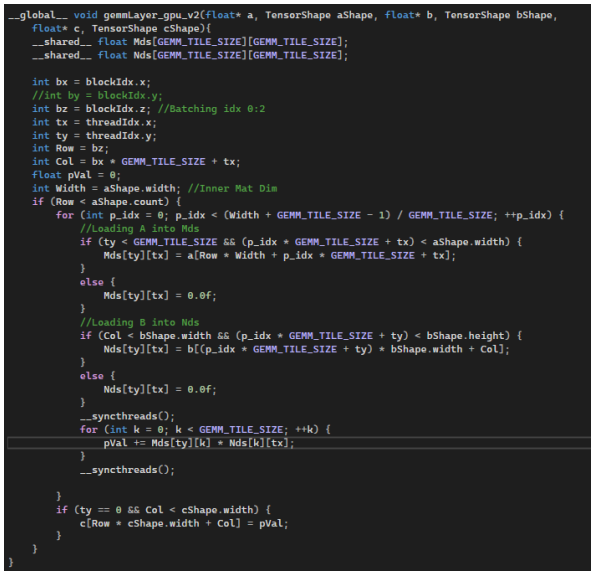


```
__global__ void gemmLayer_gpu_speed(float* a, TensorShape aShape,   float* b, TensorShape bShape,
    float* c, TensorShape cShape) {
    extern __shared__ float shared[];
    float* Mds = shared;
    float* Nds = shared + GEMM_TILE_SIZE * GEMM_TILE_SIZE;
    int bx = blockIdx.x;
    //int by = blockIdx.y;
    int bz = blockIdx.z;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int Row     = bz;
    int Col     = bx * GEMM_TILE_SIZE + tx;
    float pVal = 0;
    int Width = aShape.width; //Inner Mat Dim
    if (Row < aShape.count) {
        for (int p_idx = 0; p_idx < (Width + GEMM_TILE_SIZE - 1) / GEMM_TILE_SIZE; ++p_idx) {

            if (ty < GEMM_TILE_SIZE && tx < GEMM_TILE_SIZE && Col < bShape.width) {
                int b_row = p_idx * GEMM_TILE_SIZE + ty;
                if (b_row < bShape.height) {
                    Nds[ty * GEMM_TILE_SIZE + tx] = b[b_row * bShape.width + Col];
                }
                else {
                    Nds[ty * GEMM_TILE_SIZE + tx] = 0.0f;
                }
            }
            //Loading A into Mds
            if (ty < GEMM_TILE_SIZE && (p_idx * GEMM_TILE_SIZE + tx) < aShape.width) {
                Mds[ty*GEMM_TILE_SIZE+tx] = a[Row * Width + p_idx * GEMM_TILE_SIZE + tx];
            }
            else {
                Mds[ty * GEMM_TILE_SIZE + tx] = 0.0f;
            }
            __syncthreads();
            for (int k = 0; k < GEMM_TILE_SIZE; ++k) {
                pVal += Mds[ty * GEMM_TILE_SIZE + k] * Nds[k*GEMM_TILE_SIZE+tx];
            }
            __syncthreads();
        }
        if (ty == 0 && Col < cShape.width) {
            c[Row * cShape.width + Col] = pVal;
        }
    }
}
```

Figure 10: 'Speed Copy gemmLayer_gpu kernel

Part B: UVM Implementation

The UVM implementation utilizes a kernel (Figure 11) that is practically the same as the *'basic'* copy kernel found in Figure 9 and with my implementation of UVM the real difference lies in the host-side function call found in Figure 12. The matrices A, B and C are implemented using CUDA UVM then passed to *gemmLayer_gpu_v2* where they follow the same execution path as discussed for the *'basic'* kernel.



```
__global__ void gemmLayer_gpu_v2(float* a, TensorShape aShape, float* b, TensorShape bShape,
    float* c, TensorShape cShape){
    __shared__ float Mds[GEMM_TILE_SIZE][GEMM_TILE_SIZE];
    __shared__ float Nds[GEMM_TILE_SIZE][GEMM_TILE_SIZE];

    int bx = blockIdx.x;
    //int by = blockIdx.y;
    int bz = blockIdx.z; //Batching idx 0:2
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int Row = bz;
    int Col = bx * GEMM_TILE_SIZE + tx;
    float pVal = 0;
    int Width = aShape.width; //Inner Mat Dim
    if (Row < aShape.count) {
        for (int p_idx = 0; p_idx < (Width + GEMM_TILE_SIZE - 1) / GEMM_TILE_SIZE; ++p_idx) {
            //Loading A into Mds
            if (ty < GEMM_TILE_SIZE && (p_idx * GEMM_TILE_SIZE + tx) < aShape.width) {
                Mds[ty][tx] = a[Row * Width + p_idx * GEMM_TILE_SIZE + tx];
            }
            else {
                Mds[ty][tx] = 0.0f;
            }
            //Loading B into Nds
            if (Col < bShape.width && (p_idx * GEMM_TILE_SIZE + ty) < bShape.height) {
                Nds[ty][tx] = b[(p_idx * GEMM_TILE_SIZE + ty) * bShape.width + Col];
            }
            else {
                Nds[ty][tx] = 0.0f;
            }
            __syncthreads();
            for (int k = 0; k < GEMM_TILE_SIZE; ++k) {
                pVal += Mds[ty][k] * Nds[k][tx];
            }
            __syncthreads();

        }
        if (ty == 0 && Col < cShape.width) {
            c[Row * cShape.width + Col] = pVal;
        }
    }
}
```

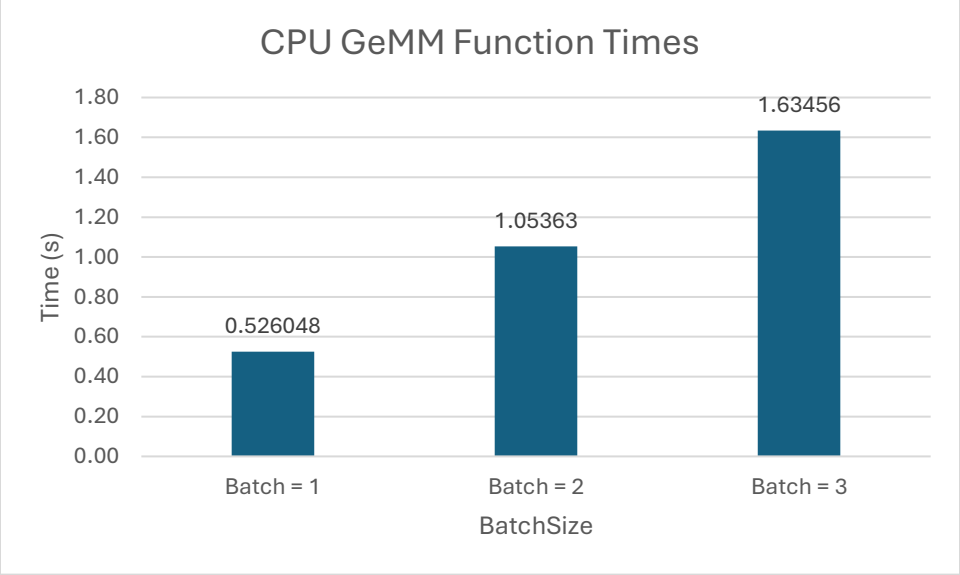
Figure 11: UVM gemmLayer_gpu kernel

```cpp
int evaluateGpuGemm_uvm(TensorShape aShape, TensorShape bShape,
    TensorShape& cShape, GemmLayerArgs args, uint32_t BatchSize) {

    cShape.height = aShape.height;
    cShape.width = bShape.width;
    cShape.channels = aShape.channels;
    cShape.count = BatchSize;

    float* uvm_a = nullptr;
    float* uvm_b = nullptr;
    float* uvm_c = nullptr;

    /*CUDA UVM Babyyy*/
    cudaMallocManaged(&uvm_a, aShape.count * aShape.width * sizeof(float));
    cudaMallocManaged(&uvm_b, bShape.height * bShape.width * sizeof(float));
    cudaMallocManaged(&uvm_c, cShape.count * cShape.width * sizeof(float));

    int retVal;
    retVal = makeTensor_uvm(&uvm_a, aShape);
    if (retVal != 0) {
        std::cout << "Unable to make tensor \n";
        return -1;
    }
    retVal = makeTensor_uvm(&uvm_b, bShape);
    if (retVal != 0) {
        std::cout << "Unable to make tensor \n";
        return -1;
    }

    /*Block and Grid Dims*/
    dim3 blockDim(GEMM_TILE_SIZE, GEMM_TILE_SIZE);
    dim3 gridDim((cShape.width + GEMM_TILE_SIZE - 1) / GEMM_TILE_SIZE, 1, cShape.count);

    /*gpuGemm Kernel Call*/
    std::cout << "UVM GPU Starting!\n";

    gemmLayer_gpu_v2 << <gridDim, blockDim >> > (uvm_a, aShape, uvm_b, bShape, uvm_c, cShape);
```

Figure 12: UVM gemmLayer_gpu host-side function

Part B: Performance Results

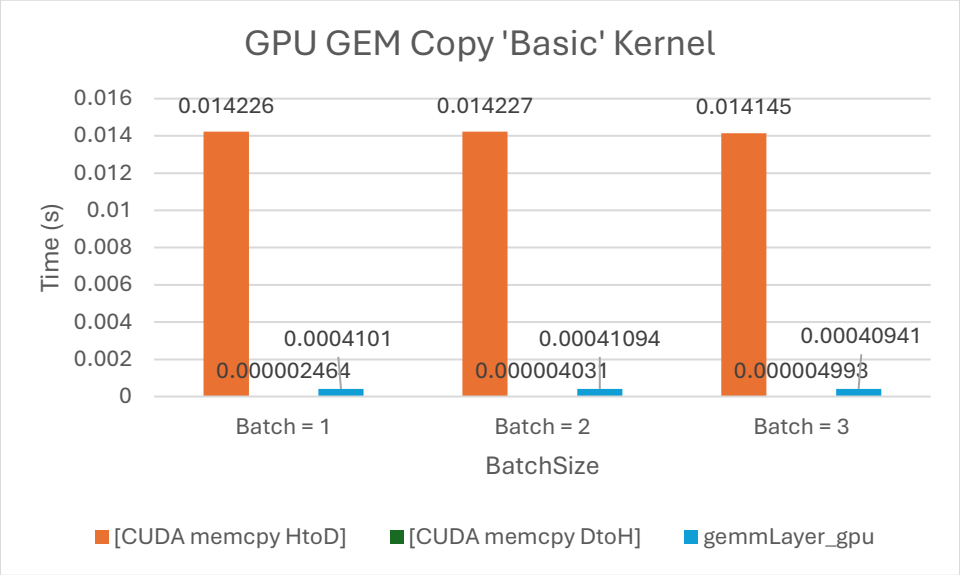
With the second part we once again have the same Purdue GPU server execution environment and begin with the results of the CPU implementation, Graph 7, and observe a roughly 0.52 second time per batch size.



Graph 7: CPU GeMM function timing per batch size.

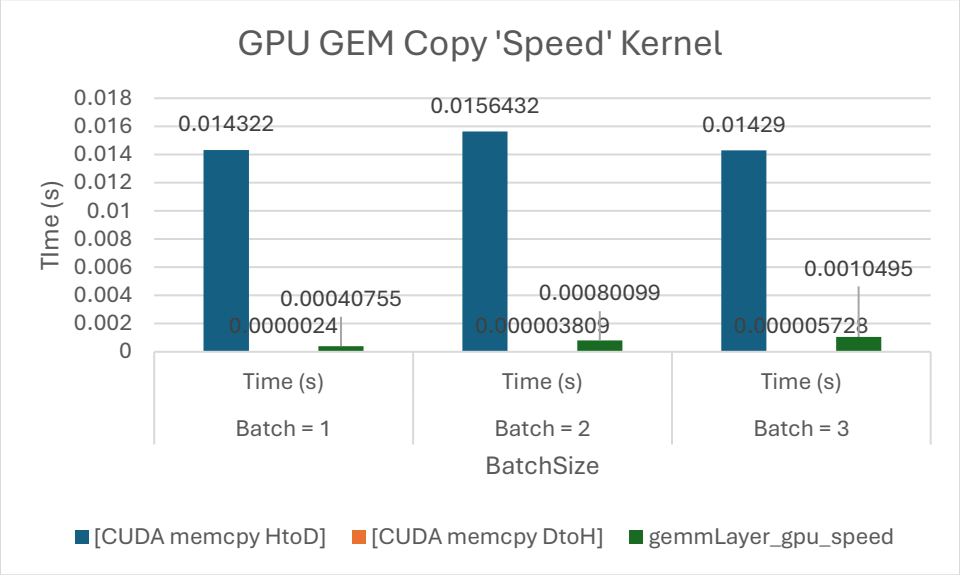
The next is Graph 8 which considers our ‘basic’ copy CUDA kernel that depicts an expected dramatic speedup over it’s CPU counterpart. Across batch sizes of 1, 2 and 3 we observe consistent speedup of 1280x compared to the CPU’s relative 0.52s time for it’s function, the

general Host to Device copying is consistent but we observe an increasing size of the Device to Host memory copy as we increase the batch size expectedly.



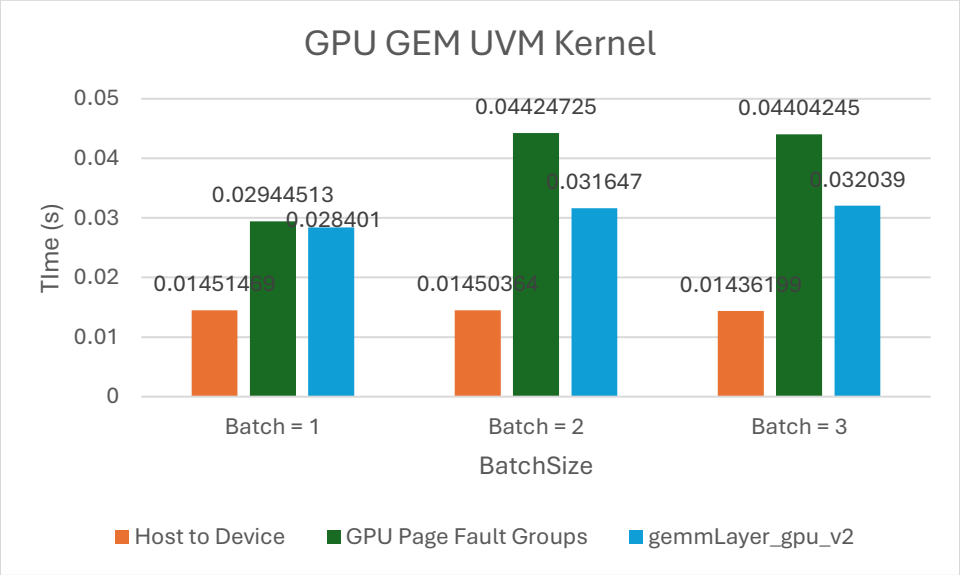
Graph 8: 'Basic' Copy GeMM kernel- GPU Activities time (s) spent varying batch size.

With Graph 9, which is supposedly my *'speed'* copy kernel (I say supposedly because the name is the opposite of reality). The kernel itself is still faster than the CPU implementation but is slower than its 'basic' counterpart and observes the same trends of memory copy times as well. This kernel sees an initial speedup for the first batch but diverges from the 'basic' implementation in that it slows down over increasing batch sizes due primarily to the way I wrote/structured the kernel. The speedup starts off at 1290x compared to the CPU.

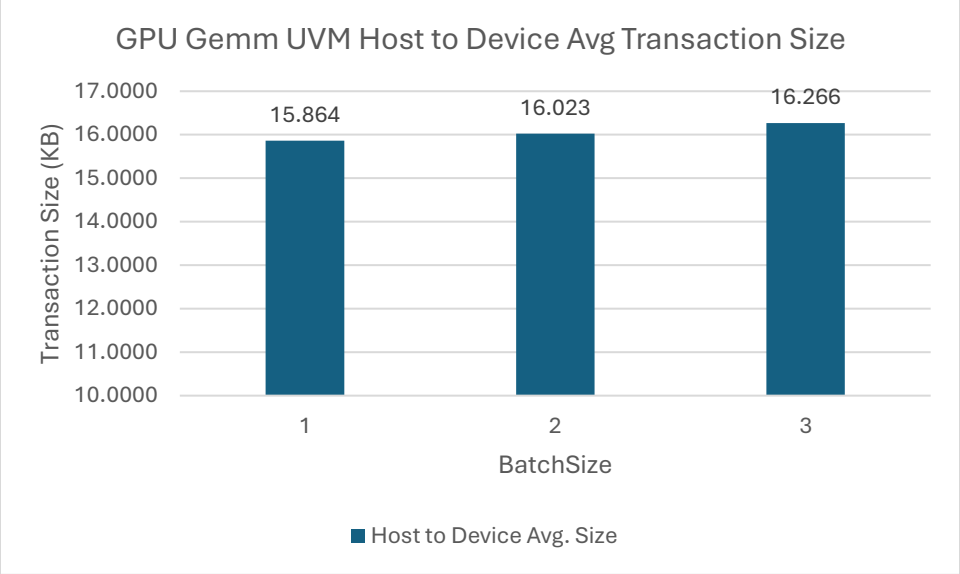


Graph 9: 'Speed' Copy GeMM kernel- GPU Activities time (s) spent varying batch size.

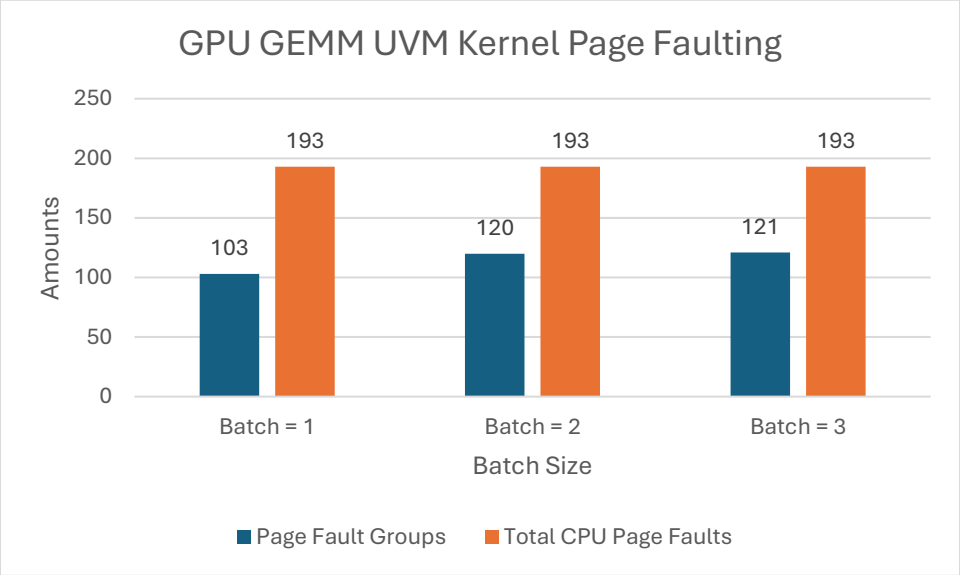
In Graph 10 we observe the UVM performance which introduces UVM functions instead of the cudaMemcpy functions but utilizes a near identical kernel found in the *'basic'* copy iteration. Considering that this function doesn't utilize memcpy we use other means of measurement presented by the *nvprof* tool, 'Host to Device' and 'GPU Page Fault Groups' and purely comparing speedups over the CPU we observe a 16-18x speedup and the *'basic'* version has a 69-78x speedup over this UVM counterpart. Depicted in Graph 11 is the average memory transaction size across batch size iterations which is relatively consistent at 16KB (4096 elements*4B) and supports its consistency in timing in Graph 10, of about 14ms. Graph 12 likewise depicts page faults which remain consistent across batch sizes and then the page fault groups which tick upwards—they're both consistent in the sense that the kernel itself is the ultimate cause of the faulting but the data itself is relatively iterated on with increasing batch size so these metrics 'make sense' but this is something that is a failing of my kernels.



Graph 10: UVM GeMM kernel- GPU Activities time (s) spent varying batch size.

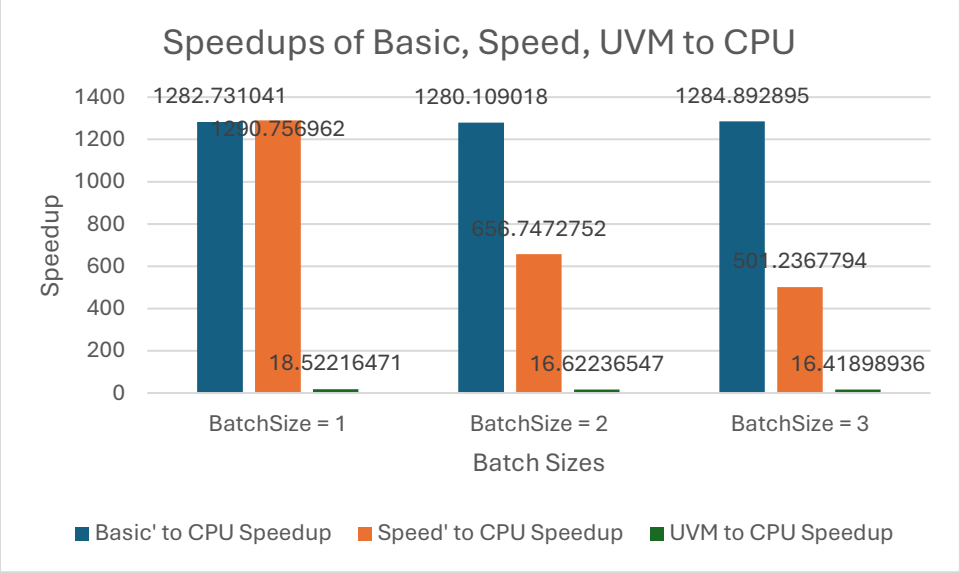


Graph 11: UVM GeMM kernel- Average memory transaction size Host to Device.



Graph 12: UVM GeMM kernel- Page fault details across batch sizes.

I conclude this section with Graph 13 and Table 2 which depict the speedups of my kernels compared to the CPU implementation and consolidate the findings outlined above noting that the *'basic'* copy implementation wins out against the competition.



Graph 13: CUDA GeMM kernel speedups compared to my CPU implementation.

	BatchSize = 1	BatchSize = 2	BatchSize = 3
Raw Times	Time (us)	Time (us)	Time (us)
‘Basic' Copy	0.0004101	0.00041094	0.00040941
‘Speed' Copy	0.00040755	0.00080099	0.0010495
UVM	0.028401	0.031647	0.032039
CPU	0.526048	1.05363	1.63456
Speedups	BatchSize = 1	BatchSize = 2	BatchSize = 3
‘Basic' to CPU Speedup	1282.731041	1280.109018	1284.892895
‘Speed’ to CPU Speedup	1290.756962	656.7472752	501.2367794
UVM to CPU Speedup	18.52216471	16.62236547	16.41898936

Table 2: CUDA GeMM kernel speedups compared to my CPU implementation.

Part B: Strategy, Drawbacks and Improvement Points

Initially, when analyzing the basic and speed copy kernels, I had to contend with my functional yet inefficient memory access patterns. While some threads within a thread block achieved coalescing, the overall implementation remained inefficient in both thread and block utilization. The *Nds* portions of my code proved challenging to optimize while maintaining accuracy—an issue that ultimately stemmed from my own skill issues. Additionally, I believe I could have further optimized the global memory reads for both *Mds* and *Nds*, potentially reducing kernel execution times to well below 400µs.

For my UVM implementation, the primary issue was a lack of optimization and a deeper understanding of how to properly leverage UVM. The sheer number of page faults I encountered was a direct consequence of inefficient memory access patterns and poor coalescing and striding, particularly in the following access pattern: *b[p_idx * 16 + ty * 4096 + Col]*

This was the main bottleneck of my kernel. If I had found a more efficient way to index this memory, the kernel's performance would have improved significantly. I also suspect I encountered *cold start issues*, where the first execution of the kernel caused page faults when accessing matrices *A, B, and C*, as they were not initially resident on the GPU. Furthermore, every write into *C* triggered write-allocate faults, further delaying execution—an issue exacerbated by my inefficient memory access patterns. Toward the end of my debugging process, I realized that prefetching techniques could have helped mitigate cold start delays by warming up the data in advance, significantly reducing initial page faults. Additionally, restructuring my kernel to better coalesce memory accesses and chunk data loads into *B/Nds* would have decreased the total number of faults.

From this assignment, it's clear to me that UVM requires a different programming paradigm (or mental model) compared to the standard *memcpy*-based approach. Despite my poorly optimized UVM kernel, I believe that for the GeMM use case, *memcpy* would still outperform UVM due to the sheer size of the matrices. Regardless of how much optimization or cache warming is done, UVM will eventually struggle with large matrix sizes, leading to excessive page faults (unless specifically delt with but that seems untenable at scale). In contrast, the *memcpy* approach allows for more controlled and predictable memory management, making it better suited for this scenario.

Part C: Verification

For Part A, I utilize the code found in Figure 13 which provides results as in Figure 14, and for Part B, Figure 15 contains the code with results in Figure 16 depicted below. Both functions are called from *runGpuConv* and *runGpuGemm*, respectively.

```
int verifyVector_convLayer(float* a, float* b, int size) {
    float tolerance = 0.01f;
    int errorCount = 0;
    for (int idx = 0; idx < size; ++idx) {
        float delta = a[idx] - b[idx];
        if (abs(delta) > tolerance) {
            ++errorCount;
            std::cout << "Idx " << idx << " expected (CPU):  " << a[idx] << " found (GPU):" << b[idx] << " Delta: " << abs(delta) << "\n";
        }
        if (idx >= 0 && idx < 20) {
            /*Range of IDX Checking*/
            //std::cout << "Idx " << idx << " expected (CPU):  " << a[idx] << " found (GPU):" << b[idx] << "\n";
        }
    }
    return errorCount;
}
```

Figure 13: ConvLayer verification function.

```
Choice selected - 13

Running ConvLayer on GPU!

GPU execution time: 2.5974e-05 seconds
CPU execution time: 0.541647 seconds

Found 0 Errors...
GPU execution time: 6.1552e-05 seconds
CPU execution time: 0.564043 seconds

Found 0 Errors...
Found 0 / 290400 errors


 ... Done!
```

Figure 14: ConvLayer verification function output.

```
int verifyVector_gemm(float* a, float* b, float* c, int size) {
    float tolerance = 0.01f;
    int errorCount = 0;
    for (int idx = 0; idx < size; ++idx) {
        float delta = a[idx] - b[idx];
        float delta2 = a[idx] - c[idx];
        float delta3 = b[idx] - c[idx];
        if (abs(delta) > tolerance || abs(delta2) > tolerance || abs(delta3) > tolerance) {
            ++errorCount;
            std::cout << "Idx " << idx << " expected (CPU):  " << a[idx] << " found (GPU):" << b[idx] << " Delta: " << abs(delta) << "\n";
        }
    }
    return errorCount;
}
```

Figure 15: GeMM verification function.

```
Running GemmLayer on GPU!

Executing GPU COPY GEMM with BatchSize: 3
'Speed-up' COPY GPU Starting!
        Memory Size: 8192 Bytes!
GPU execution time: 2.3257e-05 seconds
'Speed-up' COPY Batch 3 Time: 2.58454 ms
Executing GPU UVM GEMM with BatchSize: 3
UVM GPU Starting!
GPU execution time: 0.000340396 seconds
UVM Batch 3 Time: 20.3606 ms
CPU execution time: 1.59541 seconds

Found 0 Errors...


 ... Done!
```

Figure 16: GeMM verification function output.