CME213 Final Project Preliminary Report

Brad Huang

May 2016

1 CUDA Kernels Used

```
/**
 * Function: myAllocGEMM
 * ---------------------
 * This function computes GEMM the same way as myGEMM, but not in place. The
 * results would be stored in the pointer D, which is conditionally allocated
 * based on the allocate variable.
 */
int myAllocGEMM(const double* A, const double* B, const double* C, double*& D,
            double alpha, double beta, int M, int N, int K, bool allocate);

/**
 * Function: computeZ
 * ------------------
 * This function computes GEMM the same way as myGEMM, but not in place and
 * the second matrix b is now a vector instead of a matrix. The function is
 * required because in computing z of each layer, the bias is a vector and
 * doing so can avoid performing operations similar to repmat the bias by the
 * number of samples. The results would be stored in the pointer z, which is
 * allocated within the function.
 */
void computeZ(const double* a, const double* W, const double* b, double*& z,
    int N, int M, int H);

/**
 * Function: mySigmoid
 * -------------------
 * This function computes the sigmoid of a matrix with the given dimension
 * in-place.
 */
int mySigmoid(double* mat, int M, int N);

/**
 * Function: mySoftmax
 * -------------------
 * This function computes the softmax of a matrix with the given dimension
 * in-place, assuming that each row of the matrix is the outputs from one
 * sample.
```

```
 */
int mySoftmax(double* mat, int N, int C);

/**
 * Function: elemMults
 * -------------------
 * This function computes the elementwise multiplication as used when
 * computing the derivative of CE with z1, aka dCE/dz1 = dCE/da1 o a1 o
 * (1-a1).
 */
void elemMults(const double* A, const double* B, double*& C, int M, int N);

/**
 * Function: columnSums
 * --------------------
 * This function computes the column sum of the matrix and stores the results
 * in sums, assuming that the sums pointer has already been properly
 * allocated.
 */
void columnSums(const double* mat, double* sums, int M, int N);

/**
 * Function: matAdd, vecAdd
 * ------------------------
 * These functions compute the sum of two matrices or two vectors with the
 *given coefs and stores the results in the first matrix or vector
 */
void matAdd(double* X, double* Y, double a, double b, int M, int N);
void vecAdd(double* X, double* Y, double a, double b, int M);

/**
 * Function: transpose
 * -------------------
 * This function computes the transpose of the matrix A with dimension MxN
 * and stores the results AT, which is allocated within the function.
 */
void transpose(double* A, double*& AT, int M, int N);

/**
 * Function: freePtrs
 * ------------------
 * This function frees all the device pointers stored in a vector of
 * pointers.
 */
void freePtrs(std::vector<double*> ptrs);
```

Parallel Feedforward and Backprop

```
void myFeedforward(const NNPointers& nn, double* XT, CachePtr& cache) {
   std::vector<double*> tempPtrs;

   cache.X = XT;
   double* X;
   transpose(XT, X, cache.H[0], cache.N);
   tempPtrs.push_back(X);

   double* a1;
   computeZ(X, nn.W[0], nn.b[0], a1, cache.N, cache.H[0], cache.H[1]);
   mySigmoid(a1, cache.N, cache.H[1]);
   cache.a1 = a1;

   double* yc;
   computeZ(a1, nn.W[1], nn.b[1], yc, cache.N, cache.H[1], cache.H[2]);
   mySoftmax(yc, cache.N, cache.H[2]);
   cache.yDiff = yc;

   freePtrs(tempPtrs);
}

void myBackprop(const NNPointers& nn, double* yT, double reg, const CachePtr&
    cache, GradPtr& deviceGrad, int totalSize, int num_procs) {

   std::vector<double*> tempPtrs;

   // compute yDiff, an NxC matrix
   double* y;
   transpose(yT, y, cache.H[2], cache.N);
   matAdd(cache.yDiff, y, 1.0 / totalSize, -1.0 / totalSize, cache.N,
       cache.H[2]);
   tempPtrs.push_back(y);

   // compute dW2 = a1^T * yDiff + reg * W2
   double* a1T;
   transpose(cache.a1, a1T, cache.N, cache.H[1]);
   myAllocGEMM(a1T, cache.yDiff, nn.W[1], deviceGrad.dW[1], 1.0, reg /
       num_procs, cache.H[1], cache.H[2], cache.N, false);
   tempPtrs.push_back(a1T);

   // compute db2 = colSum (yDiff)
   columnSums(cache.yDiff, deviceGrad.db[1], cache.N, cache.H[2]);

   // compute gradients for W1 and b1
   // first, compute dCE/da1 = yDiff * W2T
   double* da1;
   double* W1T;
   transpose(nn.W[1], W1T, cache.H[1], cache.H[2]);
```

```
    myAllocGEMM(cache.yDiff, W1T, cache.a1, da1, 1.0, 0.0, cache.N,
        cache.H[1], cache.H[2], true);
    tempPtrs.push_back(W1T);
    tempPtrs.push_back(da1);

    // then compute dCE/dz1 = da1 o a1 o (1 - a1)
    double* dz1;
    elemMults(da1, cache.a1, dz1, cache.N, cache.H[1]);
    tempPtrs.push_back(dz1);

    // lastly, dW1 = X^T * dz1 + reg * W1
    myAllocGEMM(cache.X, dz1, nn.W[0], deviceGrad.dW[0], 1.0, reg / num_procs,
        cache.H[0], cache.H[1], cache.N, false);

    // db1 = colSums (dz1)
    columnSums(dz1, deviceGrad.db[0], cache.N, cache.H[1]);

    freePtrs(tempPtrs);
}
```

2 CUDA Kernel Analysis

All runs across the three configurations pass the correctness test. The overall runtime for the different configurations are as follows:

Config	CPU Runtime	GPU Runtime
1	117.216	45.7335
2	29.4631	12.1609
3	19.1442	2.14016

There are still a lot of room for improvements on runtime, especially when the number of epochs increases.

From the timeline of NVidia Visual Profiler, I extracted these information on the percentage of time taken by each kernel for the total amount of computation time on GPU:

Wrapper	Kernel	Calls per Proc per Batch	Percentage of Time
myAllocGEMM	simpleAllocGEMM	3	74.3
computeZ	gpuVecGEMM	2	18.7
mySigmoid	gpuSigmoid	1	0.1
mySoftmax	gpuExp	1	0.0
mySoftmax	gpuRowSums	1	0.0
mySoftmax	gpuRowDivide	1	0.0
elemMults	gpuElemMult	1	0.1
columnSums	gpuColSums	2	4.9
matAdd	gpuMatAdd	2	0.0
vecAdd	gpuVecAdd	2	0.0
transpose	gpuTranspose	4	1.7

From this table, we can see that the `simpleAllocGEMM` occupies most of the GPU execution time, so it would be most efficient to optimize that kernel. For similar reasons, `computeZ`, which also uses the naive matrix multiplication in the CUDA kernel, `columnSums` and `transpose` are ones that should be focused on when optimizing.

This is confirmed in the `Kernel Optimization Priorities` analysis of NVidia Visual Profiler, which produced the following priorities of kernels to optimize:

Rank	Description
100	[6 kernel instances] simpleAllocGEMM(double const *, double const *, double const *, double*, double, doul
33	[6 kernel instances] gpuVecGEMM(double const *, double const *, double const *, double*, int, int, int)
32	[6 kernel instances] gpuColSums(double const *, double*, int, int)
18	[6 kernel instances] gpuColSums(double const *, double*, int, int)
10	[1 kernel instances] simpleAllocGEMM(double const *, double const *, double const *, double*, double, doul
7	[6 kernel instances] gpuTranspose(double*, double*, int, int)
7	[6 kernel instances] simpleAllocGEMM(double const *, double const *, double const *, double*, double, doul
4	[1 kernel instances] gpuColSums(double const *, double*, int, int)
4	[1 kernel instances] gpuVecGEMM(double const *, double const *, double const *, double*, int, int, int)
4	[6 kernel instances] gpuVecGEMM(double const *, double const *, double const *, double*, int, int, int)
2	[1 kernel instances] gpuColSums(double const *, double*, int, int)
1	[1 kernel instances] gpuElemMult(double const *, double const *, double*, int, int)
1	[1 kernel instances] gpuExp(double*, int, int)
1	[1 kernel instances] gpuRowDivide(double*, double const *, int, int)
1	[1 kernel instances] gpuSigmoid(double*, int, int)
1	[1 kernel instances] gpuTranspose(double*, double*, int, int)

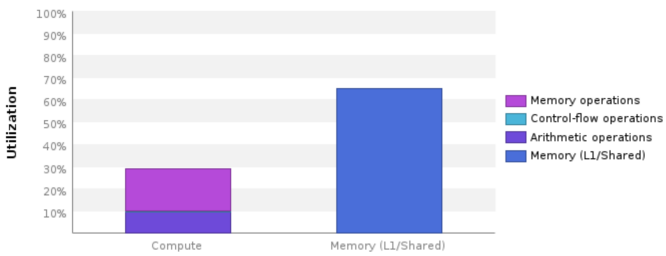
Next I'll discuss the analysis of each of these kernels and how to optimize them.

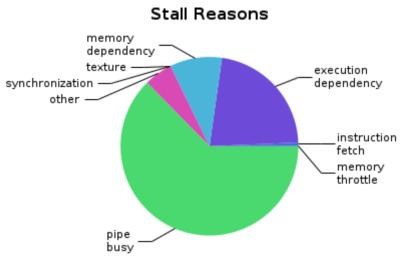
2.1 Summarization of Memory Usage

Here I summarize the memory usage of the four kernels to be optimized. Two of them are memory bounds, so I were able to extract the number of load/store transactions per request information for those two kernels.

Kernel	Transaction per Request	Memory Bandwidth (GB/s)	Bound
simpleAllocGEMM	114.3	7.121	Memory
gpuVecGEMM	N/A	6.526	Latency
gpuColSums	N/A	2.852	Latency
gpuTranspose	16	92.203	Memory

2.2 Profiling of simpleAllocGEMM





This kernel is memory bandwidth bound. This is quite obvious because the implementation is a naive one where one thread computes the final value for one element in the final result:

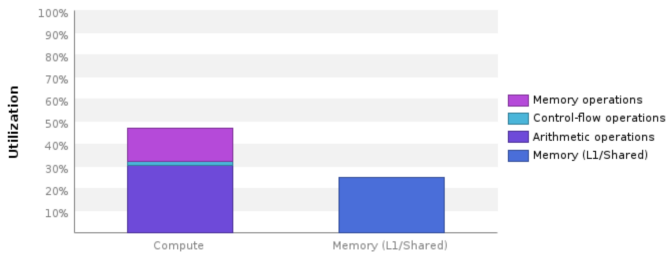
```
/*
 * Routine to allocate for D and perform a GEMM operation,
 * i.e., D := alpha*A*B + beta*C (A: MxK, B: KxN, C/D: MxN)
 */
__global__
void simpleAllocGEMM(const double* A, const double* B, const double* C,
    double* D,
                 double alpha, double beta, int M, int N, int K) {
   const int col = threadIdx.x + blockDim.x * blockIdx.x;
   const int row = threadIdx.y + blockDim.y * blockIdx.y;

   if (col >= N || row >= M) return;

   double sum = 0.0;
   for (int i = 0; i < K; ++i) {
      sum += A[i * M + row] * B[col * K + i];
   }
   D[col * M + row] = sum * alpha + C[col * M + row] * beta;
}

int myAllocGEMM(const double* A, const double* B, const double* C, double*& D,
            double alpha, double beta, int M, int N, int K, bool allocate) {
   /* TODO: Write an efficient GEMM implementation on GPU */
   if (allocate) {
      checkCudaErrors( cudaMalloc((void**) &D, M * N * sizeof(double)) );
```

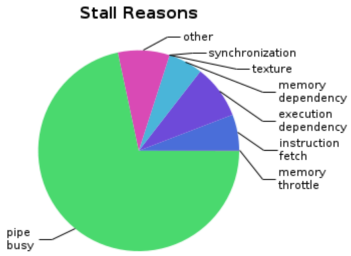
```
    }

    // launch kernel
    dim3 threadDims(16, 48);
    dim3 blockDims((N + 15) / 16, (M + 47) / 48);
    simpleAllocGEMM<<<blockDims, threadDims>>>(A, B, C, D, alpha, beta, M, N,
        K);
    check_launch ("device_alloc_gemm");

    return 0;
}
```

This naive implementation has little computation but many memory accesses - a whole row of `A` (one cache line request per row) and a whole column of `B` (`n_rows`/16 cache line requests). Since a warp is of size 16x2 here, `n_cols` cache line requests are necessary for `A` (each cache line contains 16 adjacent cells in the row direction) and `n_rows`/8 cache line requests for `B`, leading to the high number of transactions per request.

Optimizing for this kernel requires improving the coalescing of the memory access pattern, in particular the matrix multiplcation part of the kernel. The optimizations described in the project handout address this aspect using shared memory.

2.3 Profiling of computeZ





The kernel is latency bound. If we observe the kernel code:

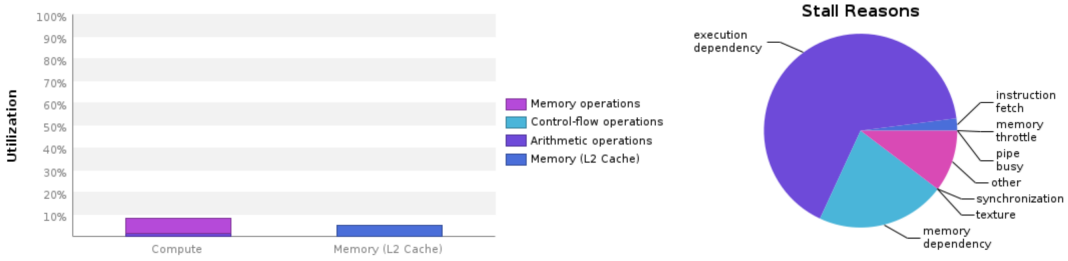
```
// NxH = NxM x MxH + N x H
__global__
void gpuVecGEMM(const double* a, const double* W, const double* b, double* z,
    int N, int M, int H) {
    const int row = threadIdx.x + blockDim.x * blockIdx.x;
    const int col = threadIdx.y + blockDim.y * blockIdx.y;
    if (row >= N || col >= H) return;

    double sum = b[col];
    for (int i = 0; i < M; ++i) {
        sum += a[i * N + row] * W[col * M + i];
    }
    z[col * N + row] = sum;
}
```

```
void computeZ(const double* a, const double* W, const double* b, double*& z,
    int N, int M, int H) {
   checkCudaErrors( cudaMalloc((void**) &z, N * H * sizeof(double)) );
   dim3 threadDims(16, 48);
   dim3 blockDims((N + 15) / 16, (H + 47) / 48);
   gpuVecGEMM<<<blockDims, threadDims>>>(a, W, b, z, N, M, H);
   check_launch ("device_vec_gemm");
}
```

This is a very similar kernel to the `simpleAllocGEMM` kernel, so the optimizations for the kernel would also hold. The kernel has a reduced memory utilization because the access pattern to `b` is different from the access pattern to `C` in the `simpleAllocGEMM` kernel - each warp is of size 16x2, so one cache line request to `b` is enough for the warp, making the portion of usable memory in the cache line smaller than that in `simpleAllocGEMM`. As a result, even though we're having fewer number of cache line requests per warp, the utilization is deemed less by the profiler.

From the pie chart of `Stall Reasons`, we can see that the kernel is structurally similar to the `simpleAllocGEMM` kernel - both are significantly stalled because of pipe busy, aka that the kernels are using a lot of low throughput instructions like double multiplication and addition.

2.4 Profiling of columnSums



This kernel is latency bound and has very low utilization in both compute and memory. Let's observe the code:

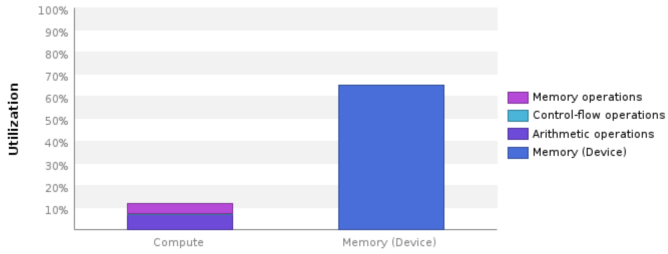
```
__global__
void gpuColSums(const double* mat, double* sums, int M, int N) {
   const int id = threadIdx.x + blockIdx.x * blockDim.x;
   if (id >= N) return;

   sums[id] = 0.0;
   for (int i = 0; i < M; ++i) {
      sums[id] += mat[id * M + i];
   }
}

void columnSums(const double* mat, double* sums, int M, int N) {
   int threadDim1D = 192;
   int blockDim1D = (N + 191) / 192;
   // compute column sums
   gpuColSums<<<blockDim1D, threadDim1D>>>(mat, sums, M, N);
```

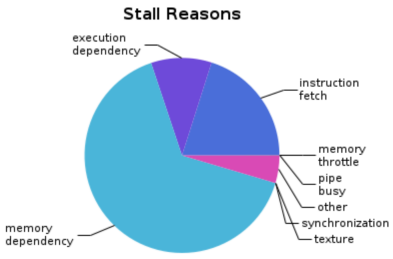
```
    check_launch ("device_col_sum");
}
```

This kernel has very bad memory coalescing - each thread is responsible for one column, and the storage is in column major, so memory coaslescing is essentially nonexistent.

Looking at the reasons for low compute utilization, the biggest reason for stalling is execution dependency. This means the code can be further optimized by introducing more threads to handle one column, and use parallelism and reduction when summing along the column. This can be done through a kernel similar to the shared memory example for summing a vector in the lecture.

2.5 Profiling of transpose





This kernel is memory bound, and also quite fittingly stalls a lot for memory dependencies. Let's inspect the code:

```
__global__
void gpuTranspose(double* A, double* AT, int M, int N) {
    const int row = threadIdx.x + blockDim.x * blockIdx.x;
    const int col = threadIdx.y + blockDim.y * blockIdx.y;

    if (row >= M || col >= N) return;

    AT[row * N + col] = A[col * M + row];
}

// transpose MxN matrix A into AT, both column major storage
void transpose(double* A, double*& AT, int M, int N) {
    // launch kernel
    dim3 threadDims(8, 4);
    dim3 blockDims((M + 7) / 8, (N + 3) / 4);

    checkCudaErrors( cudaMalloc((void**) &AT, M * N * sizeof(double)) );

    gpuTranspose<<<blockDims, threadDims>>>(A, AT, M, N);
    check_launch("device_transpose");
}
```

Indeed, one warp for this kernel consists of an 8x4 thread block. Each warp would read from a 16x4 block in A (4 cache lines) and write to a 16x8 block in AT (8 cache lines), so the total amount of usable memory is only 8x4 / (16x4 + 16x8) = 1 / 6, making the kernel memory bound.