EE-147 Lab 2
Tiled Matrix Multiplication

Cody Simons
861177050

April 28, 2018

1 Introduction

This lab our goal was to implement a tiled matrix multiplication algorithm. The goal of a tiled algorithm is to increase the the memory locality of an operation. We do this by batching together operations that access the same memory. This gives almost a 4 times speed up on large matrices.

2 Questions

1. In your kernel implementation, how many threads can be simultaneously executing? Assume a GPU which has 30 streaming multiprocessors.
   Each multiprocessor executes 32 threads at a time, so there can be at a maximum 30*32 excuting at a time or 960 threads.

2. Use nvcc --ptxas-options="-v" to report the resource usage of your implementation. Note that the compilation will fail but you will still get a report of the relevant information. Experiment with the Nvidia visual profiler, which is part of the CUDA toolkit, and use it to further understand the resource usage. In particular, report your branch divergence behavior and whether your memory accesses are coalesced.
   My kernel used 25 registers, 2048 bytes of smem, and 360 bytes of cmem. Using nvprof I found that there are 7.8 Gloabel Load Transactions Per Request. I think this means that it is loading 7 bytes per request. There are 5.9 Global store Transactions Per Request. Nvprof also classified the Control-Flow Function Unit Utilization as low which I think indicates that there isnt a lot of branch divergence.

3. How many times is each element of the input matrices loaded during the execution of the kernel?
   While it will very depending on the tile size and the size of the input matrics. If A is an mxk matrix and B is and kxn matrix then every element in A will be loaded $\left\lceil \frac{n}{tile\_size} \right\rceil$ times and every element in B will be loaded $\left\lceil \frac{m}{tile\_size} \right\rceil$ times.

3 Code

3.1 Main.cu

The main file for this lab is very similar to the last lab. The main difference is that instead of taking the size of the vector as a parameter, we must calculate the length which is the total number of elements in the matrix.

```
/******************************************************************************
 *
 *            (C) Copyright 2010 The Board of Trustees of the
 *                        University of Illinois
 *                         All Rights Reserved
 *
 ******************************************************************************/
```

```c

#include <stdio.h>
#include <stdlib.h>
#include "kernel.cu"
#include "support.h"

int main (int argc, char *argv[])
{

    Timer timer;
    cudaError_t cuda_ret;

    // Initialize host variables ----------------------------------------------

    printf("\nSetting up the problem..."); fflush(stdout);
    startTime(&timer);

    float *A_h, *B_h, *C_h;
    float *A_d, *B_d, *C_d;
    size_t A_sz, B_sz, C_sz;
    unsigned matArow, matAcol;
    unsigned matBrow, matBcol;
    dim3 dim_grid, dim_block;

    if (argc == 1) {
        matArow = 1000;
        matAcol = matBrow = 1000;
        matBcol = 1000;
    } else if (argc == 2) {
        matArow = atoi(argv[1]);
        matAcol = matBrow = atoi(argv[1]);
        matBcol = atoi(argv[1]);
    } else if (argc == 4) {
        matArow = atoi(argv[1]);
        matAcol = matBrow = atoi(argv[2]);
        matBcol = atoi(argv[3]);
    } else {
        printf("\n    Invalid input parameters!"
      "\n    Usage: ./sgemm-tiled                # All matrices are 1000 x 1000"
      "\n    Usage: ./sgemm-tiled <m>            # All matrices are m x m"
      "\n    Usage: ./sgemm-tiled <m> <k> <n>    # A: m x k, B: k x n, C: m x n"
      "\n");
        exit(0);
    }

    A_sz = matArow*matAcol;
    B_sz = matBrow*matBcol;
    C_sz = matArow*matBcol;

    A_h = (float*) malloc( sizeof(float)*A_sz );
    for (unsigned int i=0; i < A_sz; i++) { A_h[i] = (rand()%100)/100.00; }

    B_h = (float*) malloc( sizeof(float)*B_sz );
    for (unsigned int i=0; i < B_sz; i++) { B_h[i] = (rand()%100)/100.00; }

    C_h = (float*) malloc( sizeof(float)*C_sz );

    stopTime(&timer); printf("%f s\n", elapsedTime(timer));
    printf("    A: %u x %u\n    B: %u x %u\n    C: %u x %u\n", matArow, matAcol,
        matBrow, matBcol, matArow, matBcol);

    // Allocate device variables ----------------------------------------------

    printf("Allocating device variables..."); fflush(stdout);
    startTime(&timer);

    //INSERT CODE HER
    cudaMalloc((void**) &A_d, sizeof(float)*A_sz);
    cudaMalloc((void**) &B_d, sizeof(float)*B_sz);
    cudaMalloc((void**) &C_d, sizeof(float)*C_sz);

```

```
    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Copy host variables to device ------------------------------------------

    printf("Copying data from host to device..."); fflush(stdout);
    startTime(&timer);

    //INSERT CODE HERE
    cudaMemcpy(A_d, A_h, sizeof(float)*A_sz, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, sizeof(float)*B_sz, cudaMemcpyHostToDevice);

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Launch kernel using standard sgemm interface ---------------------------
    printf("Launching kernel..."); fflush(stdout);
    startTime(&timer);
    basicSgemm('N', 'N', matArow, matBcol, matBrow, 1.0f, \
                A_d, matArow, B_d, matBrow, 0.0f, C_d, matBrow);

    cuda_ret = cudaDeviceSynchronize();
        if(cuda_ret != cudaSuccess) FATAL("Unable to launch kernel");
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Copy device variables from host ----------------------------------------

    printf("Copying data from device to host..."); fflush(stdout);
    startTime(&timer);

    //INSERT CODE HERE
    cudaMemcpy(C_h, C_d, sizeof(float)*C_sz, cudaMemcpyDeviceToHost);

    cudaDeviceSynchronize();
    stopTime(&timer); printf("%f s\n", elapsedTime(timer));

    // Verify correctness -----------------------------------------------------

    printf("Verifying results..."); fflush(stdout);

    verify(A_h, B_h, C_h, matArow, matAcol, matBcol);


    // Free memory ------------------------------------------------------------

    free(A_h);
    free(B_h);
    free(C_h);

    //INSERT CODE HERE
    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);

    return 0;

}
```

3.2 Kernel.cu

In the kernel file I implemented the naive implementation and the tiled algorithm. The naive one is fairly straight forward. The tiled one takes a bit more work becuase we must load each element of the inputs in chunks. We must make sure to take care of the edge cases as well.

```
/***************************************************************************
 *cr
 *cr            (C) Copyright 2010 The Board of Trustees of the
 *cr                        University of Illinois
 *cr                         All Rights Reserved
```

```
 *cr
 ******************************************************************************/

#include <stdio.h>
#define TILE_SIZE 16

__global__ void mysgemm(int m, int n, int k, const float *A, const float *B, float* C) {

    /********************************************************************
     *
     * Compute C = A x B
     *   where A is a (m x k) matrix
     *   where B is a (k x n) matrix
     *   where C is a (m x n) matrix
     *
     * Use shared memory for tiling
     *
     ********************************************************************/

    // INSERT KERNEL CODE HERE

    // Calculate position of row and col
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure it is inside range
    if (row<m && col<n) {
        float val=0;
        for(int i=0;i<k;i++) {
            val += A[row*k+i]*B[i*n+col];
        }
        C[row*n+col] = val;
    }
}

__global__ void mysgemm_tiled(int m, int n, int k,
                              const float *A, const float *B, float *C) {
    __shared__ float ds_M[TILE_SIZE][TILE_SIZE];
    __shared__ float ds_N[TILE_SIZE][TILE_SIZE];

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

    int row = by*blockDim.y+ty;
    int col = bx*blockDim.x+tx;

    float val = 0;

    for(int i=0;i<(k-1)/TILE_SIZE+1;++i) {
        if(row<m && TILE_SIZE*i+tx<k) {
            ds_M[ty][tx] = A[row*k+tx+i*TILE_SIZE];
        } else {
            ds_M[ty][tx] = 0;
        }
        if(col<n && TILE_SIZE*i+ty<k) {
            ds_N[ty][tx] = B[(ty+i*TILE_SIZE)*n+col];
        } else {
            ds_N[ty][tx] = 0;
        }

        __syncthreads();

        if(row<m && col<n)
            for(int j=0;j<TILE_SIZE;++j)
                val+=ds_M[ty][j]*ds_N[j][tx];

        __syncthreads();
    }

    if(row<m && col<n)
        C[row*n+col] = val;
```

```
}

void basicSgemm(char transa, char transb, int m, int n, int k,
                float alpha, const float *A, int lda,
                const float *B, int ldb, float beta,
                float *C, int ldc)
{
    if ((transa != 'N') && (transa != 'n')) {
        printf("unsupported value of 'transa'\n");
        return;
    }

    if ((transb != 'N') && (transb != 'n')) {
        printf("unsupported value of 'transb'\n");
        return;
    }

    if ((alpha - 1.0f > 1e-10) || (alpha - 1.0f < -1e-10)) {
        printf("unsupported value of alpha\n");
        return;
    }

    if ((beta - 0.0f > 1e-10) || (beta - 0.0f < -1e-10)) {
        printf("unsupported value of beta\n");
        return;
    }

    // Initialize thread block and kernel grid dimensions ----------------------

    const unsigned int BLOCK_SIZE = TILE_SIZE;

    //INSERT CODE HERE
    dim3 dimGrid((n-1)/BLOCK_SIZE+1,(m-1)/BLOCK_SIZE+1,1);
    dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE,1);

    // Invoke CUDA kernel -------------------------------------------------------
    //INSERT CODE HERE
    mysgemm_tiled<<<dimGrid, dimBlock>>>(m,n,k,A,B,C);
}
```

4 Program Output

Below is the program output. This one is with zero parameters so it defaults to m=n=k=1000.

```
 ./sgemm-tiled

Setting up the problem...0.026839 s
    A: 1000 x 1000
    B: 1000 x 1000
    C: 1000 x 1000
Allocating device variables...0.086983 s
Copying data from host to device...0.003272 s
Launching kernel...0.007087 s
Copying data from device to host...0.002299 s
Verifying results...TEST PASSED
```