AN INTRODUCTION TO PARALLEL COMPUTING

CHAPTER 5
HISTOGRAMS, DOT PRODUCTS, AND TRANSPOSING MATRICES

BY FRED ANNEXSTEIN, PHD

4. HISTOGRAMS, DOT PRODUCTS, AND TRANSPOSING MATRICES

CHAPTER OBJECTIVES

1. Understand thread addressing for multi-block kernels and for decomposing data into block-based tiles for better locality.
2. Understand and apply parallel algorithms for computing dot products and histograms on GPGPUs.
3. Compare and contrast a variety of parallel approaches to computing histograms.
4. Understand the communication bottlenecks inherent in matrix transpose operations.

5. Apply and code the usage of fast shared memory operations for problems such as transpose on GPGPUs.

LOCALITY OF REFERENCE

Maintaining the “locality of references” is critical for performance of both CPUs and GPUs. GPU has potentially much higher sequential memory access performance than the CPU. To get peak memory performance, computation must be structured around contiguous memory accesses. This aligns well for accelerating linear algebra operations such as adding two large vectors, and computing dot-products where data is naturally accessed sequentially. Algorithms that do a lot of pointer chasing will suffer from poor performance due to random-access patterns.

DATA TILING

Global memory resides in device memory (DRAM) results in much slower access than shared memory.

Tiling data takes advantage of faster shared memory, but first the global data must be partitioned into subsets that will fit into limited shared memory. Each thread block must manage its own partition: load from global memory to shared memory (called the cache), then perform several computations. Finally, copy results from shared memory to global memory. To map between global and local indices we need to explicitly list parameters…

```
#define NumSMPs 30    #Machine dependent see gpu_enum
#define N    8192     #problem size
#define ThreadsperBlock   1024
#define BlocksperGrid     N/ThreadsperBlock
```

The following is code for dot product where we use the shared cache array for tree-based reduction.

```
__global__ void dot( float *a, float *b, float *c ) {
__shared__ float cache[ThreadsPerBlock];

int tid = threadIdx.x + blockIdx.x * blockDim.x;
int cacheIdx = threadIdx.x;

float temp = 0;
while (tid < N) {
   temp += a[tid] * b[tid];
   tid += blockDim.x * gridDim.x;}

cache[cacheIdx] = temp;


// Recursive Halving per Block Across the Grid
//  add pairs of values offset by 2-powers

int offset = blockDim.x/2;
while (offset != 0) {
     if (cacheIdx < offset){
         cache[cacheIdx] += cache[cacheIdx +
offset];}
    __syncthreads();
    offset /= 2; }
if (cacheIdx == 0)
c[blockIdx.x] = cache[0];
}
//Finish sum–reduction of size gridDim.x on host
```

Note the code is valid when N is larger than the number of threads, since we can use thread-id as an offset into the N-item array. Hence, in effect using the total number of threads as a stride value. Finally we would need to do a final reduction across the set of all blocks, and this is usually computed on the host.

HISTOGRAMS AND DISTRIBUTIONS

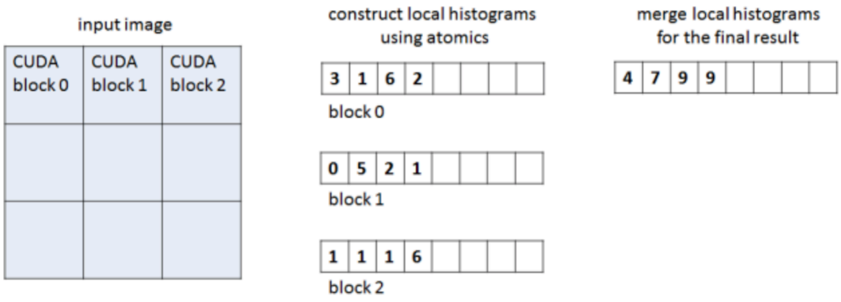
Given a vector a values, we wish to compute a histogram or frequency table with k bins. This means we are to divide the range into k (equal-sized) intervals and determine a count for number of items in each interval. For example, suppose the input is the vector [2 4 3 3 1 7 4 5 7 0 8 4 3 2]. For this the min value is 0, the max value is 8, the range is 9 values, and the histogram with three equal bins is [4 7 3]. Thus we have the cumulative distribution or scan vector is [4 11 14].

The general problem of computing a histogram is made difficult when we have little knowledge of the distribution. Depending on the input distribution, some bins will be used much more than others, so it is necessary to support efficient accumulation of the values across the full memory hierarchy. The main challenge is that the bin or output location for each element is not known prior to reading its value. Therefore, it is impossible to create a generic parallel scheme that completely avoids write conflicts or memory collisions.

3 APPROACHES TO HISTOGRAMS

There are three basic approaches using CUDA to the problem of Histogram. The first design is to use memory atomics on a per-block basis. The second design is to use only Shared-Memory Atomics. The third design is to avoid Atomics all together and use privatized per-thread.

Example: Histograms on Images. Images have 3-4 channels with range of 256 values. We use a two Phase Strategy: In the first kernel phase each CUDA thread block processes a region of the image and accumulates a corresponding local histogram, storing the per-block histogram in global memory at the end of the phase. The second kernel reduces/accumulates all per-



block histograms into the final histogram stored in global memory. The work separation between blocks in the first phase reduces contention when accumulating values into the same bin.

```
__global__ void histogram_gmem_atomics(const uchar4*
const rgbaImage, int numRows, int numCols, unsigned
int *out )

{  // pixel coordinates

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

  int t = threadIdx.x + threadIdx.y * blockDim.x
  int g = blockIdx.x + blockIdx.y * gridDim.x;

// initialize temporary accumulation array in
//    global memory

  unsigned int *gmem = out + g * NUM_BINS;
// process pixels and update block's partial
histogram in global memory

  unsigned int r = (unsigned int)(rgbaImage[y *
numRows + x].x);

  unsigned int g = (unsigned int)(rgbaImage[y *
numRows + x].y);

  unsigned int b = (unsigned int)(rgbaImage[y *
numRows + x]].z);

  atomicAdd(&gmem[NUM_BINS * 0 + r], 1);
  atomicAdd(&gmem[NUM_BINS * 1 + g], 1);
  atomicAdd(&gmem[NUM_BINS * 2 + b], 1); }}
```

ASSIGNMENT #5

In linear algebra, the transpose of a matrix is an operator which flips a matrix over its diagonal; that is, it switches the row and column indices of the matrix A by producing another matrix, often denoted by A^T . The transpose of a matrix was introduced in 1858 by the mathematician Arthur Cayley.

On the CUDA platform matrices are stored as one dimensional arrays in row-major order and indexed as i + j*N for the element A[j,i], where N is number of rows.

You are to write two tiled versions of transpose operation, one using global memory and one using shared memory. These operations are to be launched with one thread per element, in KxK threadblocks.

Your code should enable each thread in a tile to determine and write the element (i,j) of global output matrix.

There is starter code for this assignment and your code should use the gputimer code from Canvas to allow you to experiment with several transpose executions in order to find the best way on CUDA to transpose a large matrix.

The deliverables for this assignment is submission of CUDA code along with a short writeup on the results of your experiments.

```
//transpose.cu
#include <stdio.h>
#include "gputimer.h"
// #include "utils.h"

const int N= 1024;      // matrix size will be NxN

int compare_matrices(float *gpu, float *ref, int N)
{
        int result = 0;
        for(int j=0; j < N; j++)
        for(int i=0; i < N; i++)
                if (ref[i + j*N] != gpu[i + j*N])
                   {result = 1;}
        return result;
}

// fill a matrix with sequential numbers in the range
0..N-1
void fill_matrix(float *mat, int N)
{
        for(int j=0; j < N * N; j++)
                mat[j] = (float) j;
}

void
transpose_CPU(float in[], float out[])
{
      for(int j=0; j < N; j++)
```

```
          for(int i=0; i < N; i++)
                  out[j + i*N] = in[i + j*N]; //
implements flip out(j,i) = in(i,j)
}

// to be launched on a single thread
__global__ void
transpose_serial(float in[], float out[])
{
      for(int j=0; j < N; j++)
            for(int i=0; i < N; i++)
                  out[j + i*N] = in[i + j*N];

// to be launched with one thread per row of output
matrix
__global__ void
transpose_parallel_per_row(float in[], float out[])
{
      int i = threadIdx.x + blockDim.x * blockIdx.y;

      for(int j=0; j < N; j++)
            out[j + i*N] = in[i + j*N];
}



__global__ void
transpose_parallel_per_element_tiled(float in[],
float out[])
{
      //YouToDo
}

__global__ void
transpose_parallel_per_element_tiled_shared(float
in[], float out[])
{
      //YouToDo
}

int main(int argc, char **argv)
{
      int numbytes = N * N * sizeof(float);
      float *in = (float *) malloc(numbytes);
      float *out = (float *) malloc(numbytes);
      float *gold = (float *) malloc(numbytes);
      fill_matrix(in, N);
      transpose_CPU(in, gold);

      float *d_in, *d_out;

      cudaMalloc(&d_in, numbytes);
      cudaMalloc(&d_out, numbytes);
      cudaMemcpy(d_in, in, numbytes,
cudaMemcpyHostToDevice);

      GpuTimer timer;
        timer.Start();
      transpose_serial<<<1,1>>>(d_in, d_out);
      timer.Stop();
        for (int i=0; i < N*N; ++i){out[i] = 0.0;}
        cudaMemcpy(out, d_out, numbytes,
cudaMemcpyDeviceToHost);
      printf("transpose_serial: %g ms.\nVerifying ...
%s\n",
               timer.Elapsed(), compare_matrices(out,
gold, N) ? "Failed" : "Success");


        cudaMemcpy(d_out, d_in, numbytes,
cudaMemcpyDeviceToDevice); //clean d_out
        timer.Start();
      transpose_parallel_per_row<<<1,N>>>(d_in,
d_out);
      timer.Stop();
        for (int i=0; i < N*N; ++i){out[i] =
0.0;}  //clean out
      cudaMemcpy(out, d_out, numbytes,
cudaMemcpyDeviceToHost);
      printf("transpose_parallel_per_row: %g ms.
\nVerifying ...%s\n",
                timer.Elapsed(),
compare_matrices(out, gold, N) ? "Failed" :
"Success");

        cudaMemcpy(d_out, d_in, numbytes,
cudaMemcpyDeviceToDevice); //clean d_out
        // Tiled versions
        const int K= 16;
        dim3 blocks_tiled(N/K,N/K);
      dim3 threads_tiled(K,K);
      timer.Start();

transpose_parallel_per_element_tiled<<<blocks_tiled,t
hreads_tiled>>>(d_in, d_out);
      timer.Stop();
        for (int i=0; i < N*N; ++i){out[i] = 0.0;}
      cudaMemcpy(out, d_out, numbytes,
cudaMemcpyDeviceToHost);
      printf("transpose_parallel_per_element_tiled
%dx%d: %g ms.\nVerifying ...%s\n",
               K, K, timer.Elapsed(),
compare_matrices(out, gold, N) ? "Failed" :
"Success");

        cudaMemcpy(d_out, d_in, numbytes,
cudaMemcpyDeviceToDevice); //clean d_out
```

```
        dim3 blocks_tiled_sh(N/K,N/K);
      dim3 threads_tiled_sh(K,K);
        timer.Start();

transpose_parallel_per_element_tiled_shared<<<blocks_
tiled_sh,threads_tiled_sh>>>(d_in, d_out);
      timer.Stop();
        for (int i=0; i < N*N; ++i){out[i] = 0.0;}
      cudaMemcpy(out, d_out, numbytes,
cudaMemcpyDeviceToHost);

printf("transpose_parallel_per_element_tiled_shared
%dx%d: %g ms.\nVerifying ...%s\n",
               K, K, timer.Elapsed(),
compare_matrices(out, gold, N) ? "Failed" :
"Success");

      cudaFree(d_in);
      cudaFree(d_out);
}
```

TRANSPOSE IN PYTHON/NUMBA

Optional Assignment. Run experiments with the posted Python/Numba code and provide a comparison of timing of that code with your CUDA solution. Include a write-up which attempts an explanation of the performance differences.

This following Python/Numba code implements the matrix transpose algorithm documented in `http://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/`

```
# transpose.py
from numba import cuda
from numba.cuda.cudadrv.driver import driver
```

```
import math
from numba.np import numpy_support as nps

def transpose(a, b=None):
    """Compute the transpose of 'a' and store it into
'b', if given,
    and return it. If 'b' is not given, allocate a
new array
    and return that.
    :param a: an `np.ndarray` or a
`DeviceNDArrayBase` subclass. If already on
        the device its stream will be used to perform
the transpose (and to copy
        `b` to the device if necessary).
    """
    # prefer `a`'s stream if
    stream = getattr(a, 'stream', 0)

    if not b:
        cols, rows = a.shape
        strides = a.dtype.itemsize * cols,
a.dtype.itemsize
        b = cuda.cudadrv.devicearray.DeviceNDArray(
            (rows, cols),
            strides,
            dtype=a.dtype,
            stream=stream)

    dt=nps.from_dtype(a.dtype)

    tpb = driver.get_device().MAX_THREADS_PER_BLOCK
    # we need to factor available threads into x and
y axis
    tile_width = int(math.pow(2, math.log(tpb, 2)/2))
    tile_height = int(tpb / tile_width)

    tile_shape=(tile_height, tile_width + 1)

    @cuda.jit
    def kernel(input, output):

        tile = cuda.shared.array(shape=tile_shape,
dtype=dt)

        tx = cuda.threadIdx.x
        ty = cuda.threadIdx.y
        bx = cuda.blockIdx.x * cuda.blockDim.x
        by = cuda.blockIdx.y * cuda.blockDim.y
        x = by + tx
        y = bx + ty

        if by+ty < input.shape[0] and bx+tx <
input.shape[1]:
            tile[ty, tx] = input[by+ty, bx+tx]
        cuda.syncthreads()
        if y < output.shape[0] and x <
output.shape[1]:
            output[y, x] = tile[tx, ty]


    # one block per tile, plus one for remainders
    blocks = int(b.shape[0]/tile_height + 1),
int(b.shape[1]/tile_width + 1)
    # one thread per tile element
    threads = tile_height, tile_width
    kernel[blocks, threads, stream](a, b)

    return b
```