Exam 1

Author: Gabriel Hofer

Course: CSC-410 Parallel Computing

Instructor: Dr. Karlsson

Due: October 19, 2020

Computer Science and Engineering
South Dakota School of Mines and Technology

How to Make the Project

```
7356111@linux09 CSC-410-Exam-1 >>make
gcc openmp.c -fopenmp -lm -o openmp
nvcc cuda.cu -o cuda
```

Usage Statement

```
Usage: ./cuda [-r low_power high_power] [-c low_power high_power]
-r: runs Floyd's algorithm in parallel on range of powers of two
-c: runs correctness tests on range of powers of two
```

Usage Examples (Performance Tests for a Range of n)

```
  7356111@linux09 CSC-410-Exam-1 >>./cuda -r 1 12
  2, 0.000023
  4, 0.000031
  8, 0.000059
  16, 0.000113
  32, 0.000224
  64, 0.000483
  128, 0.001060
  256, 0.003196
  512, 0.014511
  1024, 0.113596
  2048, 0.846598
  4096, 6.665819
```

```
  7356111@linux09 CSC-410-Exam-1 >>./openmp -r 1 12
  2, 0.000549
  4, 0.000271
  8, 0.000282
  16, 0.000430
  32, 0.001399
  64, 0.001146
  128, 0.008198
  256, 0.033451
  512, 0.168379
  1024, 1.088660
  2048, 7.835085
  4096, 60.018689
```

Data in a Table

n	CUDA (sec)	OpenMP (sec)
2	0.000023	0.000549
4	0.000031	0.000271
8	0.000059	0.000282
16	0.000113	0.000430
32	0.000224	0.001399
64	0.000483	0.001146
128	0.001060	0.008198
256	0.003196	0.033451
512	0.014511	0.168379
1024	0.113596	1.088660
2048	0.846598	7.835085
4096	6.665819	60.018689

Usage Examples (Correctness Tests)

```
7356111@linux09 CSC-410-Exam-1 >>./cuda -c 1 10
SAME
SAME
SAME
SAME
SAME
SAME
SAME
SAME
SAME
SAME
ALL SAME:)
```

```
7356111@linux09 CSC-410-Exam-1 >>./openmp -c 1 10
SAME
SAME
SAME
SAME
SAME
SAME
SAME
SAME
SAME
SAME
ALL SAME:)
```

For both options -r and -c, two integers are required to follow. These two integers are the range for p. The two listings above show how to run cuda.cu and openmp.c from the command line using the -c option (which runs correctness tests). Each output line that prints "SAME" is a successful test. And, "ALL SAME:)" means that all tests passed. Each test is run on a different value of n calculated as follows:

$$n = 2^p$$

Where p is a number between 1 and 10 which were the low_power and high_power arguments passed to the program.

Contents of exam1.tgz

1. Makefile
2. cuda.cu
3. openmp.c
4. exam1.pdf

Functions

- **main** calls either usage or range or correctness depending on the command line arguments.
- **Usage** prints a Message to standard output about how to run the program.
- **makeMatrix** makes a random matrix. the probability of that there is an edge for any two vertices is equal to 0.25. We use the rand() C language function to “generate” random integers. we also set the seed value before any rand() calls.
- **serial** is our implementation of Floyd’s algorithm without any parallelization. We use it to check the correctness of our parallelized functions.
- **Correctness** tests whether our parallelized code is correct. We make the assumption that the function called serial is a correct implementation of Floyd’s algorithm. So, we compare the output of our parallelized function to the output of serial.
- **Range** runs Floyd’s algorithm in parallel for a range of values of n. we iterate from small power of 2 to a greater power of 2.
- **printA** simply prints the 2D array, with a tab separating each column.

Files

- **cuda.cu** contains entire CUDA implementation of floyd’s algorithm.
- **openmp.c** constains entire OpenMP implementation of floyd’s algorithm.

CUDA Floyd

Listing 1: CUDA Floyd

```
__global__ void aux(int * dA, const int n, const int k){
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if(index >= n*n) return;
  __syncthreads();
  int i = index / n, j = index % n;
  dA[i*n+j] = dA[i*n+j] < (dA[i*n+k]+dA[k*n+j]) ?
    dA[i*n+j] : dA[i*n+k]+dA[k*n+j];
  __syncthreads();
}

void floyd(int * dA, const int n){
  for(int k=0;k<n;k++){
    aux<<<(n*n+THREADS_PER_BLOCK)/(THREADS_PER_BLOCK),
      THREADS_PER_BLOCK>>>(dA,n,k);
    cudaDeviceSynchronize();
  }
}
```

In CUDA, we use indexing to get rid of the two inner for loops in Floyd's algorithm.

OpenMP Floyd

Listing 2: OpenMP Floyd

```
void floyd(int * A, const int n){
  for(int k=0;k<n;k++)
    #pragma omp parallel for
    for(int i=0;i<n;i++)
      #pragma omp parallel for
      for(int j=0;j<n;j++)
        A[i*n+j] = A[i*n+j] < (A[i*n+k]+A[k*n+j]) ?
          A[i*n+j] : A[i*n+k]+A[k*n+j];
}
```

In the OpenMP version of Floyd's algorithm, we insert two pragmas for the second and third nested for loops.

Serial (Non-Parallelized)

Listing 3: Serial Floyd

```
void serial(int * A, const int n){
  for(int k=0;k<n;k++)
    for(int i=0;i<n;i++)
      for(int j=0;j<n;j++)
        A[i*n+j] = A[i*n+j] < (A[i*n+k]+A[k*n+j]) ?
          A[i*n+j] : A[i*n+k]+A[k*n+j];
}
```

Testing and Verification

Listing 4: CUDA Correctness Testing

```
void correctness(const int low, const int high){
  for(int n = pow(2,low); n <= pow(2,high); n*=2){
    int * A = makeMatrix(n);
    int * B = (int *)malloc(n*n*sizeof(int));
    int Asize = n*n*sizeof(int);
    memcpy(B, A, Asize);
    serial(B,n);

    int * dA=NULL;
    cudaMalloc((void **)&dA, Asize);
    cudaMemcpy(dA, A, Asize, cudaMemcpyHostToDevice);
    floyd(dA,n);
    cudaMemcpy(A, dA, Asize, cudaMemcpyDeviceToHost);

    bool foundDiff=false;
    for(int i=0;i<n;i++)
      for(int j=0;j<n;j++)
        if(B[i*n+j]!=A[i*n+j]){
          foundDiff=true;
          return;
        }
    cudaFree(dA);
    free(A);
    free(B);
    cudaDeviceSynchronize();
    if(foundDiff){
      printf("FOUND DIFFERENCE:(\n\n");
      return;
    }
    printf("SAME\n");
  }
  printf("ALL SAME:)\n\n");
}
```

Experimental Serial Fraction

$$e = \frac{\frac{1}{\psi} - \frac{1}{p}}{1 - \frac{1}{p}}$$