Homework # 3

Rohan Kapur, ECPE 293A

April 22, 2025

1. *Active learning problem 2*

   Reduction is a parallelizable operation where several related data points are combined into a single result in any dataset consisting of related values. For instance, we can reduce a list of given words to a mapping of each unique word to its frequency. Consider the list `{hi, hi, ok , ok, nice, nice, nice}`, which we can reduce to the mapping `{hi: 2, ok: 2, nice: 3}`. This task can be parallelized by independently considering each unique word and counting them in parallel, and then combining the frequencies into a single mapping.

   The prefix sum is an operation where, given a list of values, for instance `{1,2,3,4,5}`, each entry is replaced with the sum of the values up until that point, i.e. `{1,1+2=3,1+2+3=6,10,15}` for our example. This can be parallelized by independently considering each index in the list and summing the values up until that point in parallel.

   Scanning is a more generalized form of the prefix sum where the sum operation is replaced with some other operation. For instance, given a list of characters, e.g. `{a,b,c,d,e}`, it can be scanned so each position is replaced with the cumulative concatenation of all the characters up until that point, i.e. `{a,ab,abc,abcd,abcde}` for the example given. This is parallelized similar to how the prefix sum is, with the cumulative concatenation done independently in parallel for each position in the list.

   Sorting is another operation that can be parallelized depending on the algorithm used. For instance, with merge sort or quick sort, each of the smaller sub-lists of the given list that are to be sorted and merged together to form the sorted list can be independently sorted in parallel threads, and then combined thereafter.

   Active learning problem 4

   ```
   dim3 dimGrid3(2,2,2);
   ```

   Values of pertinent variables:

   ```
   gridDim.x=2, gridDim.y=2, gridDim.z=2
   0<=blockId.x<=1, 0<=blockId.y<=1, 0<=blockId.z<=1
   ```

   Active learning problem 5

   ```
   dim3 dimThreads(16,16,16);
   ```

   Values of pertinent variables:

   ```
   blockDim.x=16, blockDim.y=16, blockDim.z=16
   0<=threadIdx.x<=15, 0<=threadIdx.y<=15, 0<=threadIdx.z<=15
   ```

2. PyCUDA vector addition performance analysis

Listing 1: Comparing vector addition using the CPU vs. the GPU with pyCUDA

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy as np
import argparse
import time

# Define CUDA device kernel
module = SourceModule(
    r"""
    // C `long*` types used to properly handle np.int64 arrays!
    // Otherwise, with `int*` instead, the allocated memory boundaries overlap with individual elements in the
    // `np.int64` type array, resulting in `0` for every other entry in `a` and `b`!
    // Also, `double*` should be used for `np.float64` arrays!
__global__ void add_kernel(long* d_a, long* d_b, long* d_c, int vecsize) {
    int tid = threadIdx.x + blockIdx.x*blockDim.x; // Get id for current thread on GPU
    if (tid < vecsize) {
        d_c[tid] = d_a[tid] + d_b[tid];
    }
}
"""
)

# Reference: https://documen.tician.de/pycuda/tutorial.html
def vec_add_gpu(a: np.ndarray, b: np.ndarray, vecsize: int):
    """Add 2 vectors of the size given using pyCUDA on the device's Nvidia GPU selected

    Args:
        a (np.ndarray): First vector to add
        b (np.ndarray): Second vector to add
        vecsize (int): Size of the vectors being added

    Returns:
        np.ndarray: Result of vector addition (a+b) on the GPU
        float: Host-to-device data transfer time
        float: Kernel execution data transfer time
        float: Device-to-host data transfer time
    """

    # Allocate memory on and transfer vector data to the GPU device and
```

```
    # and obtain htod transfer time
    start_htod = time.time()
    a_gpu = cuda.mem_alloc(a.nbytes)
    cuda.memcpy_htod(a_gpu, a)
    b_gpu = cuda.mem_alloc(b.nbytes)
    cuda.memcpy_htod(b_gpu, b)
    time_htod = time.time() - start_htod

    # Allocate space and memory on the device for output
    c = np.empty_like(b)
    c_gpu = cuda.mem_alloc(c.nbytes)

    # Get instance of and execute device kernel and obtain kernel execution
        time
    func = module.get_function("add_kernel")
    # Types being passed MUST be objects of numpy types!
    # Create blocks of width 1024 threads
    # and grid of (vecsize // 1024 + 1) blocks
    # since blocks can have 1024 threads at most
    block_size = 1024
    grid_size = vecsize // block_size + 1
    start_kernel = time.time()
    func(
        a_gpu,
        b_gpu,
        c_gpu,
        np.int32(vecsize),
        block=(block_size, 1, 1),
        grid=(grid_size, 1),
    )
    kernel_time = time.time() - start_kernel

    # Get data from device and obtain dtoh transfer time
    start_dtoh = time.time()
    cuda.memcpy_dtoh(c, c_gpu)
    time_dtoh = time.time() - start_dtoh

    return c, time_htod, kernel_time, time_dtoh

def vec_add_cpu(a: np.ndarray, b: np.ndarray):
    """Perform and output result of ordinary vector addition on the device CPU
        .

    Args:
        a (np.ndarray): First vector to add
        b (np.ndarray): Second vector to add

    Returns:
```

```
        np.ndarray: Result of vector addition (a+b) on the CPU
    """
    return a + b # Return result of vector addition

def main():
    # Initialize CUDA driver
    cuda.init()

    # Query number of GPU's
    num_gpus = cuda.Device.count()
    print("Skeleton code for all gpu projects. MUST USE IT.")
    print(f"Number of available GPUs: {num_gpus}")

    # Prompt user for Pacific ID
    user_id = int(input("Enter your Pacific ID: "))

    # Determine which GPU to use based on the ID
    gpu_index = user_id % num_gpus
    print(f"Using GPU index: {gpu_index}")
    selected_gpu = cuda.Device(gpu_index)
    print(f"Selected GPU: {selected_gpu.name()}")

    # Initialize argument parser
    parser = argparse.ArgumentParser(
        description="Compare performance of simple vector addition using many
            parallel CUDA threads on the Nvidia GPU vs. a single thread on the
            CPU"
    )
    parser.add_argument(
        "--vecsize", type=int, required=True, help="Length of the vectors
            being added"
    )

    # Get command line argument
    args = parser.parse_args()
    vecsize = args.vecsize

    # Initialize random vectors of integers to add
    # THE C DATATYPE OF THE FOLLOWING IS np.int64 (C long*), NOT C int* (np.
        int32)
    # THE DATA TYPE USED FOR THE PARAMETER TYPES IN THE KERNEL PARAMETERS IS
        IMPORTANT
    # FOR PROPER OPERATION!!!
    a = np.random.randint(vecsize, size=(vecsize,))
    b = np.random.randint(vecsize, size=(vecsize,))

    # Invoke vector addition on the CPU and time it
    start = time.time()
```

```
    cpu_result = vec_add_cpu(a, b)
    cpu_time = time.time() - start
    print(f"CPU vector addition time: {cpu_time:.6f} s")

    # Invoke vector addition GPU and time it
    gpu_result, time_htod, kernel_time, time_dtoh = vec_add_gpu(a, b, vecsize)
    gpu_time = time_htod + kernel_time + time_dtoh
    print(f"GPU vector addition time")
    print(f"Host to device transfer time: {time_htod:.6f} s")
    print(f"Kernel execution time: {kernel_time:.6f} s")
    print(f"Device to host transfer time: {time_dtoh:.6f} s")
    print(f"Overall GPU time: {gpu_time:.6f} s")

    # Output speedup factor for GPU kernel execution vs CPU
    print(f"Execution speedup factor: {cpu_time / kernel_time:.2f}x")

    # Verify results are the same
    if np.allclose(cpu_result, gpu_result):
        print("The results are the same!")
    else:
        print("The results are different!")
        print(f"CPU result: {cpu_result}")
        print(f"GPU result: {gpu_result}")

if __name__ == "__main__":
    main()
```

Performance Comparison of CPU vs GPU for Vector Addition

Vector Size	CPU time (ms)	GPU Time (ms)				
		H2D Transfer	Kernel	D2H transfer	Total	Speedup
10,000	0.048	21.39	1.41	0.087	22.887	0.034x
100,000	1.017	1.592	1.325	2.389	5.307	0.768x
1,000,000	9.094	5.158	1.036	9.502	15.697	8.78x
10,000,000	42.03	36.93	0.689	38.618	76.236	61x

Table 1: A CUDA block size of 1024 linear threads and grid size of $\left\lceil \frac{\text{Vector Size}}{1024} \right\rceil$ linear blocks was used for this analysis. The CPU time consistently increased for larger vectors, whereas the GPU kernel execution time consistently decreased. However, transferring data to and from the GPU remained a significant bottleneck and made the GPU time worse overall for performing a single operation. This means that the benefits of GPU parallelism are best realized for multiple expensive operations being performed on data transferred a minimal number of times. The speedup factor is given from dividing the CPU time by the GPU kernel time.

The kernel execution time decreased significantly as the vectors being added increased in size due to the benefits offered by several thousand threads performing addition for corresponding vector entries in parallel. On the other hand, as expected, the CPU execution time consistently increased as the vectors grew. However, a consistent bottleneck remained the time taken for transferring the vector data from the host to the GPU device and back, which when combined

with the kernel execution time made the GPU time worse overall for a single operation being performed on the data transferred. So the benefits of parallelism seem to be best realized when multiple operations are to be performed on the data being transferred a minimal number of times, as the kernel takes significantly less time to execute than the single-threaded CPU does for the same operation.