ECE 4822: Engineering Computation IV

Homework No. 8: Managing GPU Resources

By Shahzad Khan

The goal of this assignment is to demonstrate that I can split a job into multiple tasks and run them on multiple GPUs in parallel to each GPU.

The first two tasks of the assignment were simple. Determine the number of GPUs and acquire the stats of said GPUs. CUDA has built commands that allow us to call them and figure out how many GPUs are available.

We can define a int to store the number of GPUs then use the built in command "cudaGetDeviceCount" and call that into our defined in and print it out.

```
 // Getting the number of GPUs on the node

 int deviceCount = 0;

 cudaGetDeviceCount(&deviceCount);

printf("Number of GPUs:  %d\n", deviceCount);

nedc_130_[1]: Number of GPUs: 4
```

Now that we have the number of GPUs available to us, we have to figure out the actual capabilities of each GPU. Once again CUDA has a built-in command that makes this pretty simple. We can get this by using the "cudaDeviceProp" command and to go through every GPU available to us we can use a for loop, and I just pulled this off of stack overflow.

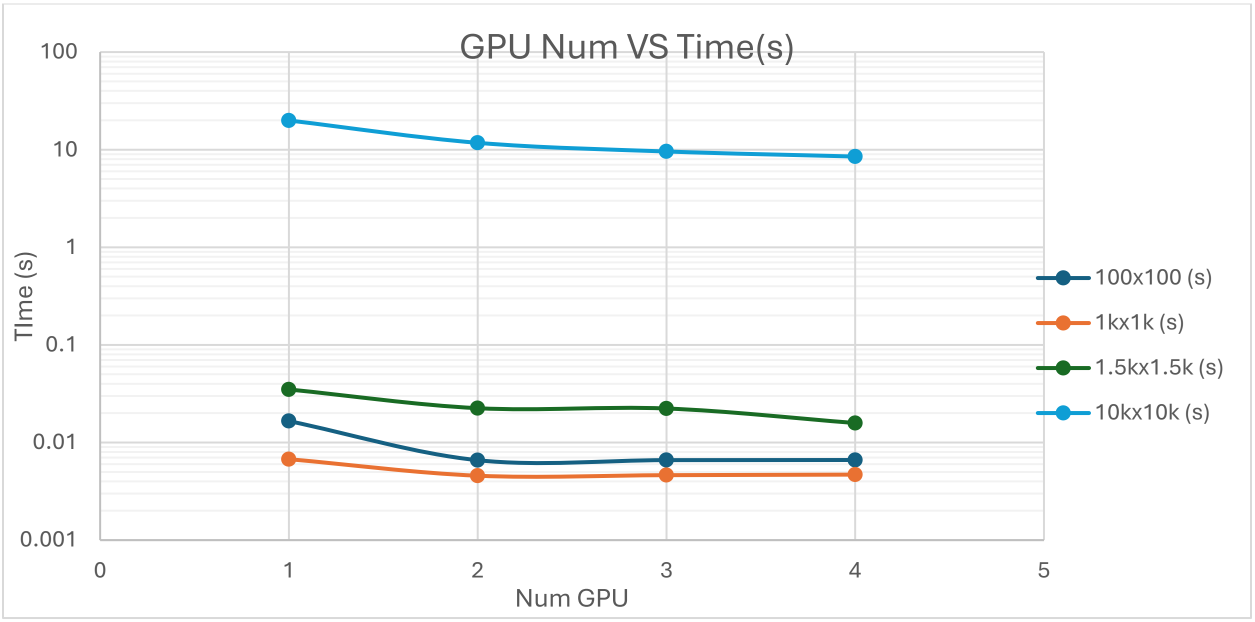
```
 for (int i = 0; i < deviceCount; i++) {
  cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, i);
  printf("Device Number: %d\n", i);

  printf("  Device name: %s\n", prop.name);

  printf("  Memory Clock Rate (MHz): %d\n",
      prop.memoryClockRate/1024);
  printf("  Memory Bus Width (bits): %d\n",
      prop.memoryBusWidth);
  printf("  Peak Memory Bandwidth (GB/s): %.1f\n",
      2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
  printf("  Total global memory (Gbytes)
%.1f\n",(float)(prop.totalGlobalMem)/1024.0/1024.0/1024.0);
  printf("  Shared memory per block (Kbytes) %.1f\n",(float)(prop.sharedMemPerBlock)/1024.0);
  printf("  minor-major: %d-%d\n", prop.minor, prop.major);
  printf("  Warp-size: %d\n", prop.warpSize);
  printf("  Concurrent kernels: %s\n", prop.concurrentKernels ? "yes" : "no");
  printf("  Concurrent computation/communication: %s\n\n",prop.deviceOverlap ? "yes" : "no");
 }
```

Next, we have split a job between different GPUs the way I tackled this is by splitting the rows to each gpu depending on how many rows and GPUs. So GPU gets a part of the rows and the next GPU get a different amount of rows well sending the full B matrix.

```
for (int iter = 0; iter < niter; iter++) {
  // printf("\n--- Iteration %d ---\n", iter + 1);
  for (int g = 0; g < gpusToUse; g++) {

    cudaSetDevice(g);
    int startRow = g * rowsPerGPU;

    int nrowsA = (g == gpusToUse - 1) ? (nrows - startRow) : rowsPerGPU;

    size_t bytesA = nrowsA * ncols * sizeof(float);
    size_t bytesB = nrows * ncols * sizeof(float);
    size_t bytesC = nrowsA * ncols * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc(&d_A, bytesA);
    cudaMalloc(&d_B, bytesB);
    cudaMalloc(&d_C, bytesC);

    cudaMemcpy(d_A, h_A + startRow * ncols, bytesA, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, bytesB, cudaMemcpyHostToDevice);
    cudaMemset(d_C, 0, bytesC); // clear output for each iteration

    dim3 block(16, 16);
    dim3 grid((ncols + block.x - 1) / block.x, (nrowsA + block.y - 1) / block.y);

    // Time per GPU kernel
    cudaEvent_t iterStart, iterStop;

    cudaEventCreate(&iterStart);
    cudaEventCreate(&iterStop);
    cudaEventRecord(iterStart);
```

Each GPU gets all of B, computes its part of C, and

the host combines C0 + C1 + C2 = full matrix C.



Looking at the timing graph now, we can see that across all our matrix sizes, the runtimes from a single GPU were considerably slower and had a significant improvement when moving to two. This is understandable, since the workload is essentially halved and GPUs can work in parallel without too much communication overhead. However, as the number of GPUs is increased beyond two-for example, using four GPUs on a relatively small 100×100 matrix-the performance gain rapidly diminishes, and in some cases, the total execution time increases.

This happens because for smaller problem sizes, the overhead of managing multiple GPUs becomes comparable to, or even higher than the computation time itself. Each additional GPU requires the additional overhead of partitioning the data, allocation of memory, transfer of data from host to device, and synchronization of kernels. These do not have linear scaling costs in terms of the amount of computation involved; therefore, with small matrices, the GPUs do the majority of their work in setting up and communicating rather than performing useful work. In contrast, for larger matrices the computation time dominates the overheads, which is why we observe a clear speedup as more GPUs are added.

In other words, this is the typical trade-off between parallel efficiency and problem size: the multi-GPU configuration pays off when huge workloads saturate the hardware completely, but for small matrices, the coordination cost is too high compared to the benefit of parallelism.

In this assignment, I demonstrated how to manage and utilize multiple GPUs to perform computations in parallel. By first detecting the available GPUs and examining their properties, I was able to design a method to distribute a large matrix operation efficiently across several devices. The approach of splitting the input matrix by rows allows each GPU to handle a unique portion of the data while sharing the full secondary matrix, ensuring balanced workload distribution and minimal communication overhead.

This approach emphasizes the importance of knowledge in GPU architecture and memory management for scaling computations beyond a single device. The exercise made me revisit the main concepts of parallel programming in CUDA, such as the selection of devices, memory allocation, and synchronization. Overall, it showed how multi-GPU computing can significantly improve performance because its large computation tasks are divided into small and independent operations that run concurrently.