COL380: Assignment 4

Parallel Matrix Multiplication using CUDA

Ishita Chaudhary
2019CS10360
April 28, 2023

We use CUDA C++ to create a massively parallel Matrix Multiplication application for CUDA-capable GPU, NVIDIA TESLA K40, in our case.

Detailed Approach

The approach is broadly divided into three stages which are described below.

1. **Reading the input matrices**
   We are given two input matrices, A and B, as binary files. Only the blocks with at least one non-zero element are given in the input file. The other blocks are assumed to have all 0s. We read these binary files and store the list of non-zero blocks in array *blocks* and the values of these blocks in array *mat.* We sort this data by block indices (i,j) and then store the starting index for each i = x in another array *indices.* This index array will help us parallelize matrix multiplication operations on the GPU.
   These arrays are allocated using *cudaMallocManaged()*, which returns a pointer that you can access from the host (CPU) code or device (GPU) code, to avoid any further copying. We also allocate *blocksOutput* and *matOutput* arrays to store blocks and their values of resultant matrix C' where MAX VAL = $2^{32} - 1$ (4 bytes).

$$C'_{ij} = min(C_{ij}, MAX\_VAL) \qquad C_{ij} = \sum_{k=0}^{r-1} A_{ik} \times B_{kj}$$

2. **Matrix Multiplication on GPU**
   We write a CUDA kernel function called *calculateResult* for matrix multiplication. This function takes as input the *blocks*, *indices,* and *mat* arrays for both the matrices and fills the *blocksOutput* and *matOutput* arrays for the resultant matrix.
   Let *n* be the dimension of the matrices, and *m* be the dimension of each block. Let *d=n/m* be the dimension of the block matrix. We launch **1024** threads per block and **d^2/1024** blocks in total so that we can calculate the result for each of the d^2 blocks on separate threads in parallel on the GPU. For every (i,j)th block, the kernel function uses the *indices* array to identify the range of *blocks* in both matrices that could contribute to the final values in the resultant product matrix. Since we sorted the input blocks, these ranges are contiguous. This makes the complexity of the kernel call O(d^2) in the worst case when the matrices are dense.

3. **Outputting the resultant matrix**
   We use the *blocksOutput* and *matOutput* arrays to generate the output binary file in the required format. We only output those blocks of the resultant matrix with at least one non-zero value.

Key Optimizations

- We use *cudamallocmanaged()* to allocate arrays required by both CPU and GPU, allowing for easy data transfer between the two. The unified memory is managed by the CUDA runtime system and can be accessed by both the host (CPU) and device (GPU) code without the need for explicit memory copies.
- We sort the list of blocks obtained in the input for each matrix so that we can just iterate over a small contiguous range of the blocks in the kernel function to get the final output for some blocks in the resultant matrix. It is important to note that the ability to calculate each block separately and quickly is essential to truly realize the benefits of massive GPU parallelization.
- We build an extra *indices* array for each of the two input matrices so that we can quickly reach the relevant range of blocks for some particular block in the resultant matrix. This allows us to efficiently compute each of the blocks of the resultant matrix in parallel on the GPU device.

Observations

- We observe that for smaller values of n, the allocation time of arrays on unified memory between the CPU and GPU is the bottleneck, whereas, for larger values of n, file I/O becomes the bottleneck because it happens sequentially on the CPU.
- For n ~ 2^{15} and k ~ 10^5, it was observed that the time for outputting the resultant matrix into a binary file was over 70% of the total runtime. In contrast, the GPU function for matrix multiplication took only about 10% of the total runtime, the remaining time taken by the reading stage. Thus 90% of the total program time is file I/O.
- The above example uses over 50,000 blocks with 1024 threads in each block. This shows that the GPU is being used efficiently for computation by the program.

References

[1] NVIDIA technical blog: An Even Easier Introduction to CUDA

[2] NVIDIA slides: CUDA Basics

[3] NVIDIA slides: CUDA Optimizations

[4] NVIDIA slides: CUDA Streams

[5] Programming Massively Parallel Processors by D. Kirk and W. Hwu