Parallelizing Sparse Matrix-Vector Multiplication on GPUs

Olaolu Biggie Emmanuel Paul Vorobyev

Abstract

Sparse Matrix-Vector Multiplication (SPMV) is one of the most important operations in the computational sciences today. Because it is so ubiquitous, there is plenty of literature available on improving its performance. In this paper, we discuss two such possibly improvements. Because many-core GPUs are becoming increasingly common, there is much research being done into utilizing their inherent power. We look into two parallel algorithms: based on atomic and segment-scan based operations in order to achieve significant improvements over a sequential implementation.

1. Introduction

Matrix operations are pertinent to many areas of mathematics, computer science, as well as the natural and physical sciences. Because sparse matrix structures often arise in these fields, the multiplication thereof becomes absolutely crucial to the performance of many applications. For this reason, Sparse Matrix-Vector Multiplication is very prominent in the field of computational science [1]. SpMV remains a commonly-used primitive in a plethora of sparse linear algebra algorithms.

With the prevalence of multi-core CPUs and GPUs on the rise, there is much research being done into harnessing the power of parallelization to boost the performance of commonly-used algorithms. By exploiting fine-grained parallelism, it is possible to use the individual processor-cores on a many-core GPU to significantly improve the speed of this commonly-used operation. Of course, this is more easily said than done. The challenge in writing algorithms for many-core GPUs is that work is distributed among tens of thousands of threads of execution rather than just 4 or 8.

In this paper, we employ two different methods to parallelize the algorithm: one utilizing atomic operations, and the other based on segment scan.

1.1. Matrix Structure

Some of the complexity inherent in parallelizing sparse matrix algorithms specifically is the structure in which they are stored in. Because sparse matrices are primarily filled with zeros, they are often stored in formats that omit the zeros in order to save space. In our implementation, we used the Matrix Market format provided by the code package. The format keeps three separate arrays for For reference, the format is illustrated below:

A matrix

$$\mathbf{A} = \begin{matrix} 1 & 0 & 3 \\ 4 & 5 & 0 \\ 0 & 8 & 9 \end{matrix}$$

In *Matrix Market Format* becomes

rIndex = [1, 1, 2, 2, 3, 3]
cIndex = [1, 3, 1, 2, 2, 3]
val = [1, 3, 4, 5, 8, 9]

2. Target Platform

Our kernel programs were written in CUDA and are tested on Nvidia GPUs. We are using CUDA version 7.5 on an iLab machine with 198 CUDA cores. Neither of our implementations utilize the shared memory hierarchies provided by the GPU, but is plenty of shared memory provided between threads in a warp/block, as well as a *texture cache* for read-only data.

3. Atomic Implementation

In an atomics based approach, different threads can process the same matrix row and communicate by atomic read-modify-write instructions. To ensure a three-step read-modify-write operation, we used a CUDA-provided hardware instruction called atomic instruction. In our algorithm, each thread is assigned more or less the same number of multiplication operations and the communication between different threads is accomplished using the atomic instructions.

3.1. The Algorithm

We used the algorithm provided in the project description, however we adapted it to work with the Matrix Market format rather than the one it used in the provided listing [2].

Each thread is supposed to compute one element of the resultant vector. We start off by creating copies of the input matrix and vector and the output vector on the device and pass them into our kernel. In our kernel function, we start off by identifying the current thread. Then, we iterate over the input vector and begin multiplying it by the non-zero entries in the current row of the matrix. In the end, we perform an atomic addition operation in the relevant space in the resultant vector. This is done on each thread until the entire vector is computed.

4. Segmented Scan Implementation

Segment scan is a common primitive used in a myriad of different many-core algorithms. It's ubiquity and utility is largely due to it's simplicity and exploitation of fine-grained parallelism. In our implementation, as opposed to our atomic implementation, we greatly diverged from the example provided in the project description.

4.1. Segmented Scan

Given an input sequence and and a binary operator - since we are implementing prefix-sum, this would be addition - a scan operation produces an output sequence which applies the operation to each index.

Segmented scan takes this idea a step further by performing separate parallel scans on arbitrary contiguous partitions of the input vector. Grouping threads encourages a divide-and-conquer approach. Organizing algorithms to match these natural execution granularities to achieve maximum efficiency [3].

4.2. The Algorithm

First, we sorted the non-zeros to avoid sorting the individual val, rIndex, and cIndex arrays. We perform the multiplication operations sequentially; only addition is performed in our kernel. We then invoke our kernel.

In our kernel function, we start off by addressing the issue of splitting rows between threads. We do not want a row to be split between blocks, so we will edit any necessary chunk sizes so that no elements for the same row are split between two blocks. For example, if block n sees that the first e elements in block n+1 have the same row as the last elements in n, then n's chunk will be extended by e. Similarly, if n notices that the last elements in n-1 have the same row as the first e elements in n, then n's chunk size will be decreased by e and an offset of e will be added to the index of all the elements in the chunk for n. We then perform the segmented scan. We calculate the number of iterations required based on the chunk size.

We also take precaution by handling any possible extra elements in case the number of threads is less than the total number of elements.

References

[1]: N. Bell, M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors.

[2]: E. Zheng. CS 516/415 Programming Languages and Compilers II Project 1: Parallelizing SPMV on GPU.

[3]: S. Sengupta, M. Harris, Y. Zhang, J. D. Owens. Scan Primitives for GPU Computing.