**Sparse matrix vector multiplication (spmv) based on segmented scan, with improvement of two packing algorithms**

Wangchao Sheng (ws333), *Rutgers University*

Abstract

Sparse matrix vector multiplication (spmv) is arguably the most important operation in sparse linear algebra. I exploited fine-grained parallelism in spmv and implement the algorithm in CUDA for project 1. In this project, I kept optimizing spmv by data reordering. Data reordering means that the data layout of the sparse matrix will be reorganized for better run-time spatial reuse.

1. System Design

1.1. Key Variables

|  |  |
| --- | --- |
| int M | # of rows of input matrix |
| int N | # of columns of input matrix |
| int nz | # of non-zeroes of input matrix |
| float X[N] | original input vector |
| float X’[N] | input vector after transforming |
| int reorderX[N] | reorderX[i]=j means that after transforming, X’[i]=X[j] |
| int  reorder\_reverseX[N] | if reorderX[i]=j, then  reorder\_reverseX[j]=i |
| float Y[M] | original output vector |
| float Y’[M] | output vector before transforming |
| int reorderY[M] | reorderY[i]=j means that before transforming,Y’[i]=Y[j] |
| HashMap graph | the whole graph stored as a hashmap, with the key the vertex index and the value another sub-hashmap |

1.2. Data Structure and Algorithms

First-touch-packing algorithm is simple. We just iterate all non-zeroes and put the column and row indexes in the order that we saw them, with no duplication. At the same time we keep the reorderX, reorder\_reverseX, and reorderY arrays so we could do the transformation later. The graph based packing algorithm would be discussed in detail in the following paragraphs.

1.2.1. Generating Graph (Hashmap)

In this project, I use a hashmap to generate the graph. For the whole graph, the (key, value) entries are basically pairs of (vertex index *u*, sub-hashmap). In the sub-hashmap, the (key, value) entries are pairs of (vertex index *v*, weight *w*), which means there exists an edge in the graph (*u, v*) weighted *w*. The pseudo code for generating graph is as below:

*void generatingGraph(graph, matrix->val){*

*for 32 consecutive non-zeroes in matrix->val{*

*for each (u, v) where u, v are in the set && u!=v{*

*if (u, v) exists in graph{*

*weight of (u, v) ++}*

*else{*

*put (u, v)=1 into graph}*

*same thing with (v, u);*

*}*

*move to next 32 elements;*

*}*

*}*

Since each put operation in a hashmap is constant time, and the number of (key, value) entries we need to add is roughly 32\*32\*nz/32. The running time of generating graph is

32\*32\*nz/32 \*O(1) = O(nz) (1)

Each different (key, value) entry takes the space of 3 integers (including a pointer variable), so in the worst case, such that all entries are distinct, the memory space we need is 3\*sizeof(int)\*32\*32\*nz/32 = 382\*nz bytes. Luckily, we won’t meet such extreme case normally. The true space complexity really depends on the pattern of the input matrix.

1.2.2. Sorting (Array, LinkedList)

After we get the whole graph unsorted, we will want to get the edge with the maximum weight, for plenty of times. Hashmap would not be a good choice, so we sort it to an array of linkedlist. First, we sort every sub-hashmap into a sorted linkedlist using weight as key, descendingly. We will have O(N) sorted linkedlists which represents the outgoing edges of these O(N) vertices. Then we sort the heads of those linkedlists with the weight. After this step is done, we will have an sorted array of O(N) head linked nodes, which point to next nodes. For example, this is the array we have after sorting:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| from | to | to | to |  |
| 4 | 2:19 | 7:11 | 3:8 | … |
| 2 | 4:19 | 8:13 | 9:3 | … |
| 8 | 2:13 | 3:7 | 1:5 | … |
| 1 | 12:9 | 8:5 | 4:2 | … |
| 13 | … | … | … | … |
| …. | … | … | … | … |

Note: the rows are sorted by the weight of the first node, keeping the algorithm still working because when we finish a cache line, we can switch the corresponding rows to NULLs because we won’t look at them anymore.

Sorting an array of length n takes O(n\*log n) time. In our case, we need to sort O(N) small arrays with length 32\*nz/N and one big array with length O(N). Thus the total running time of this step is

O(N)\*32\*nz/N\*log(32\*nz/N) + N\*log N

= nz\*log(32nz/N) + N\*logN

= O(nz) (2)

Although the running time is linear to nz, the constant could be several thousand or even bigger.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Data Structure | | Matrix | Linkedlist | Hashmap | Self-balancing Binary Search Tree |
| Time Complexity | Generating Graph | O(nz) | O(nz^2/N) | O(nz) | O(nz) |
| Getting Reordering | O(N^3) | O(N) | O(N) | O(N) |
| Space Complexity | | O(N^2) | O(N) | O(N) | O(N) |
| \* avg\_out - Average outgoing degree for each vertex in the graph: 31\*nz/N, because each column index will appear nz/N times in average, and for each time of appearance, there would be 31 outgoing edges at most. This number can be seen as a constant in spmv because its range normally would be (1, 1000). | | | | | |

1.2.3. Getting Reorder

Since we already have the sorted graph, we can get the edges with the maximum weight quickly. The pseudo code for getting reorder is in Project3A.pdf. For getting the edge with maximum weight, we simply take the first non-NULL linkedlist in the array and take the first node, taking constant time; for getting the edge with maximum weight connected to some other nodes, we just need to look at those linkedlists corresponding to the vertices in the current cache line, taking constant time as well since cache line size is a constant and the length of each linkedlist is O(nz/N), which can be seen as a constant. We need O(N) operations of getting edge so the total running time of getting reorder is

O(N) (3)

1.3. Complexity Analysis

For first-touch packing algorithm, time complexity is O(nz); space complexity is O(N).

For graph based packing algorithm, time and space complexity really depends on the data structure we choose. There’re several choices that I’ve been either thinking of or implemented:

1). Matrix – the most naïve way is to use an N\*N matrix to store the whole graph. The advantage of this design is simplicity and speed. The time complexity of generating graph is O(nz). For getting reorder, since a whole traversal of O(N^2) is needed for each getting edge with maximum weight, the time complexity is O(N^3). The space complexity is O(N^2). Obviously neither O(N^3) time nor O(N^2) space is acceptable for input in Million level.

2). LinkedList – Realizing that the whole graph is sparse, although not as sparse as input matrix, using linkedlist to store the sub-graph is a better option. The outgoing degree of every vertex is 32\*nz/N. We can still give N slots so that each vertex can have its own linkedlist and it’s easier to find the linkedlist belonging to one vertex. We sort the sub-graphs and the whole graph so we could generate the reorder faster. The time complexity of this design is O(nz^2/N) for generating the graph and O(N); the space complexity is O(N).

3). HashMap – Instead of linkedlist, hashmap has the advantage of constant time accessing. We don’t even need the N rows exactly. Although the space complexity is still O(N), it’s actually several times smaller than the linkedlist version. As discussed in 1.2, the time complexity for generating graph (including sorting) and getting

reorder is O(nz) and O(N) respectively (see formula (1), (2) and (3). This is the design I implemented.

4). Self-balancing Binary Search Tree – Self-balancing binary search tree, also called treemap, is a sorted hashmap data structure based on red-black tree. It fits our demands in this project perfectly. Either put, get, or poll takes logarithmic time. The time and space complexity would not be better than the previous hashmap design, but it’s way simpler and more efficient. It should be best option to solve this problem.

The time and space complexity is summarized in the table.

2. Experiment Results

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Matrix | M | N | nz | Segment scan (ms) | First touch packing (ms) | Graph based packing (ms) | Graph preparation (s) |
| web-Google | 916428 | 916428 | 5105039 | 28 | 20 | 19 | 2839 |
| kneser\_10\_4\_1 | 349651 | 330751 | 992252 | 5 | 4 | 4 | 8 |
| il2010 | 451554 | 451554 | 1082232 | 4 | 4 | 4 | 10 |
| road central | 14081816 | 14081816 | 16933413 | 66 | 58 | 56 | 3117 |
| web-Stanford | 281903 | 281903 | 2312497 | 11 | 9 | 8 | 7429 |
| hugebubbles-00010 | 19458087 | 19458087 | 29179764 | 110 | 110 | N/A | N/A |
| hugetric-00020 | 7122792 | 7122792 | 10680777 | 41 | 37 | 36 | 513 |
| Road\_usa | 23947347 | 23947347 | 28854312 | 91 | 85 | N/A | N/A |
| pds-80 | 129181 | 434580 | 927826 | 4 | 3 | 3 | 9 |
| nc2010 | 288987 | 288987 | 708310 | 3 | 3 | 2 | 5 |

My program could run with all inputs except two huge matrices – “hugebubbles” and “Road\_usa”. The matrix information and kernal time of segment scan (without improvement), first touch packing, and graph based packing are kept in the following table. Experiment environment:

- block size: 256, block number: 8

- OS: CentOS 7.3 | CPU: Intel® Core™ i7-7700 @ 3.60GHz - 8 cores | Memory: 32GB

- GPU Specs: GT 730 | Memory:2 GB | CUDA Cores:384 | CUDA capability : 3.5 | Version: 9.0

Findings:

1). Both first touch packing algorithm and the graph based packing algorithm can improve the kernel time of segment scan, and graph based packing beats first touch packing by a little bit.

2). First touch packing would be the optimal algorithm if each column index appears at most once in the input matrix, because it needn’t go back to look at some Xi that appeared before and was put in the former part of the reordered X’. However, it’s not a common case. When the same column index appears for several times and randomly, first touch packing, as a greedy algorithm, will show its limitation.

3). In the opposite, the graph based algorithm, is taking global information into account. It counts the total number that some column contacts with other column in the same cacheline, and chooses the most frequent one by one. It’s like a global greedy algorithm, because getting the edge with maximum weight doesn’t yield to global optimum actually. Still it wins first touch packing.

3. Future Work and Acknowledgement

1). For the graph based packing algorithm, implement it with self-balancing binary search tree.

2). For the graph based packing algorithm, use better strategy to get the reorder, yielding to better global improvement.

Acknowledgement:

This project cannot be done without Yi Hou (yh523). We discussed the algorithms and shared some basic data structures. I used her spmv\_optimize\_graph.c with her consent.

Reference

[1] website

<https://devblogs.nvidia.com/parallelforall/how-query-device-properties-and-handle-errors-cuda-cc/>

[2] CUDA programming guide

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>

[3] stackoverflow

<https://stackoverflow.com/questions/10460742/how-do-cuda-blocks-warps-threads-map-onto-cuda-cores?noredirect=1&lq=1>

[4] stackoverflow

<https://stackoverflow.com/questions/3519598/streaming-multiprocessors-blocks-and-threads-cuda?noredirect=1&lq=1>

[5] lecture slides

<https://www.cs.rutgers.edu/~zz124/cs515_fall2017/lectures/lec02.pdf>

[6] website

<https://www.geforce.com/hardware/desktop-gpus/geforce-gt-630/specifications>

[7] website

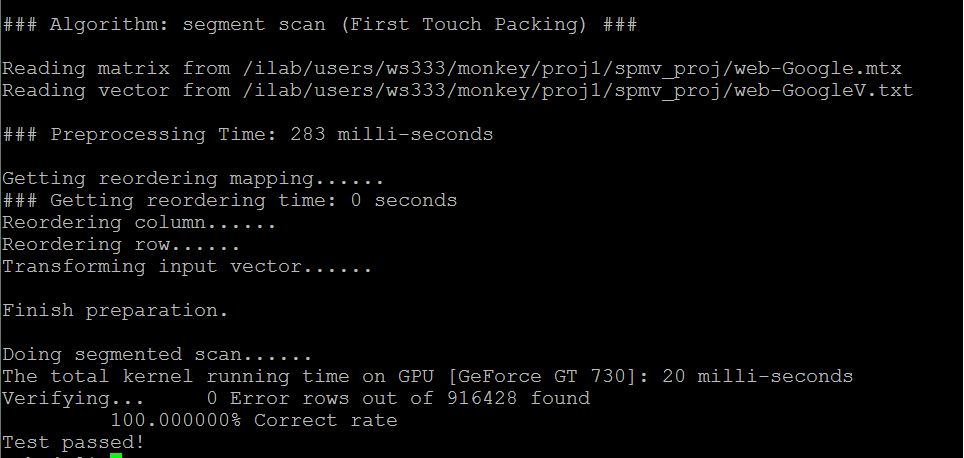
<https://www.geforce.com/hardware/desktop-gpus/geforce-gt-730/specifications>

[8] website

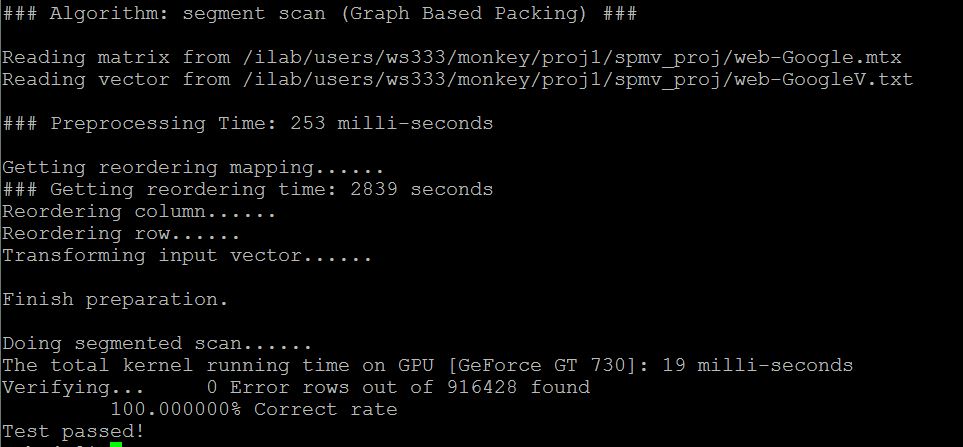
<https://developer.nvidia.com/cuda-gpus>

Appendix A – Screenshots of Running

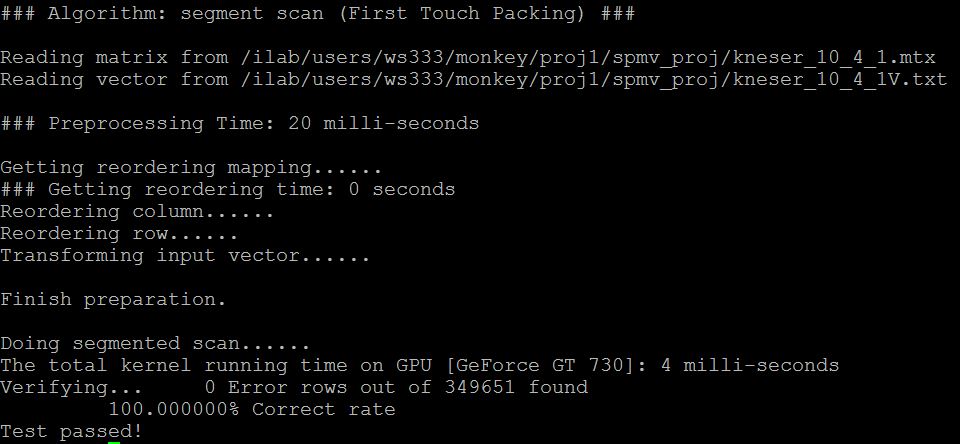
web-Google: First Touch Packing



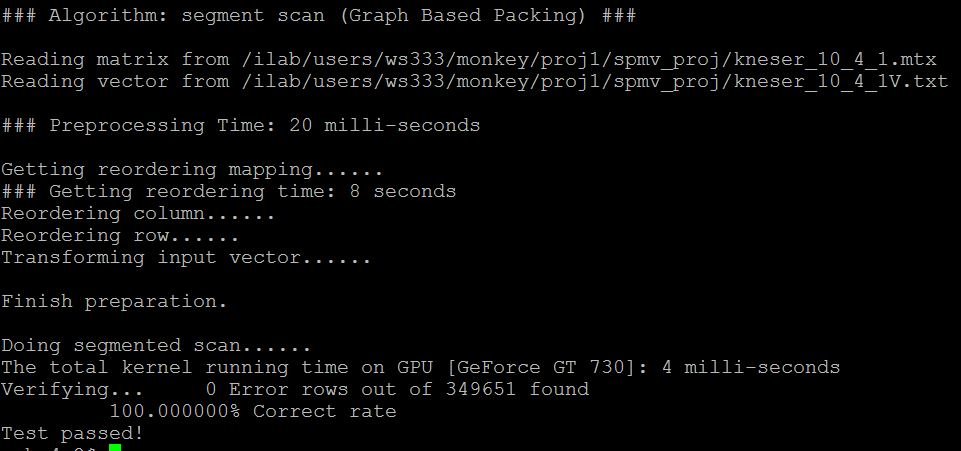
web-Google: Graph Based Packing



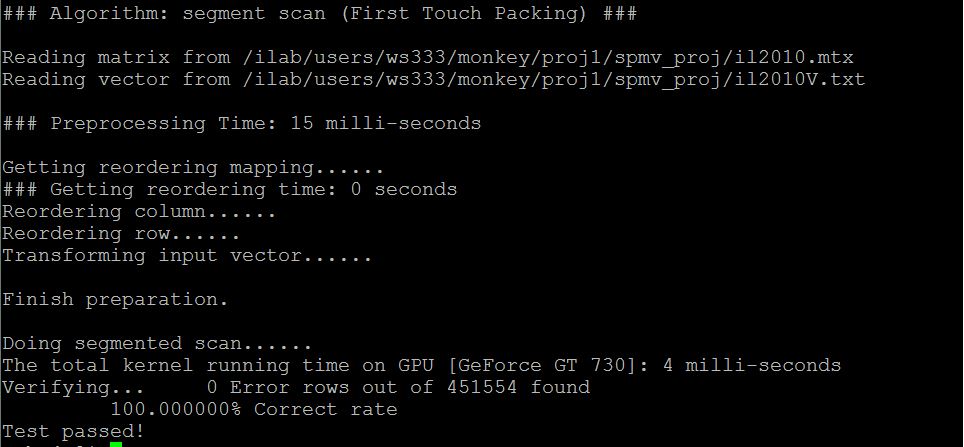
kneser\_10\_4\_1: First Touch Packing



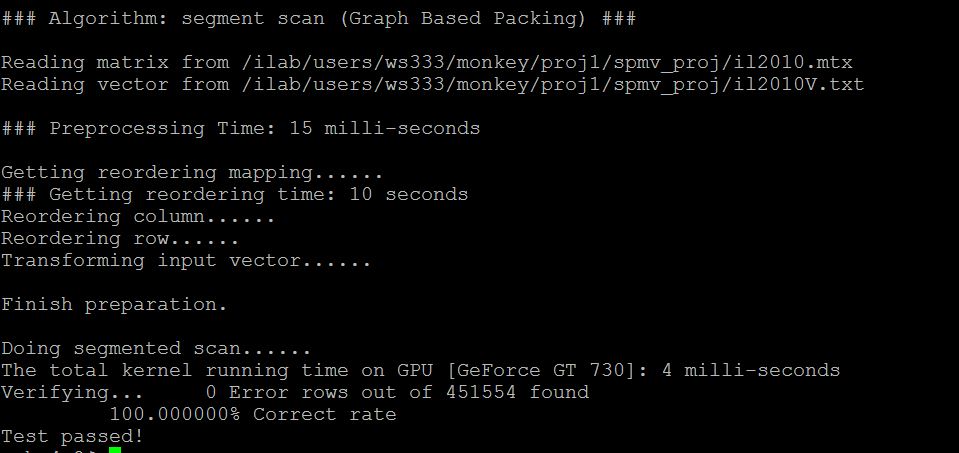
Kneser\_10\_4\_1: Graph Based Packing



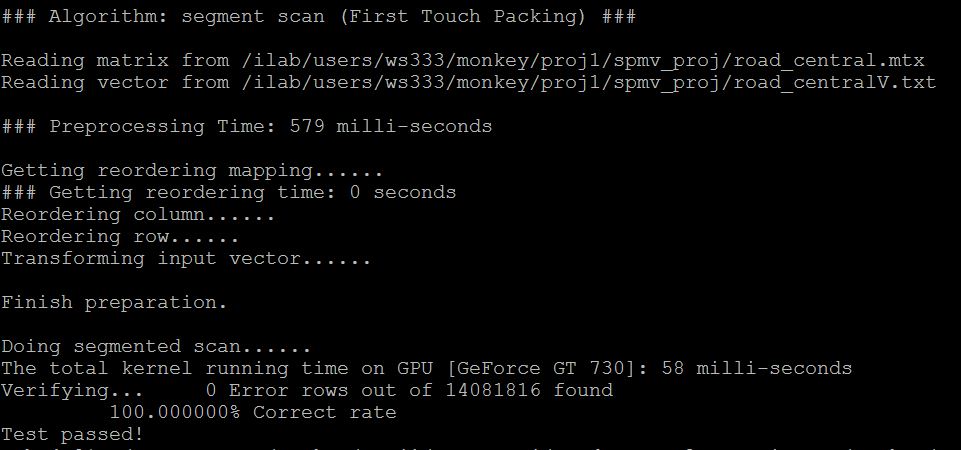
il2010: First Touch Packing



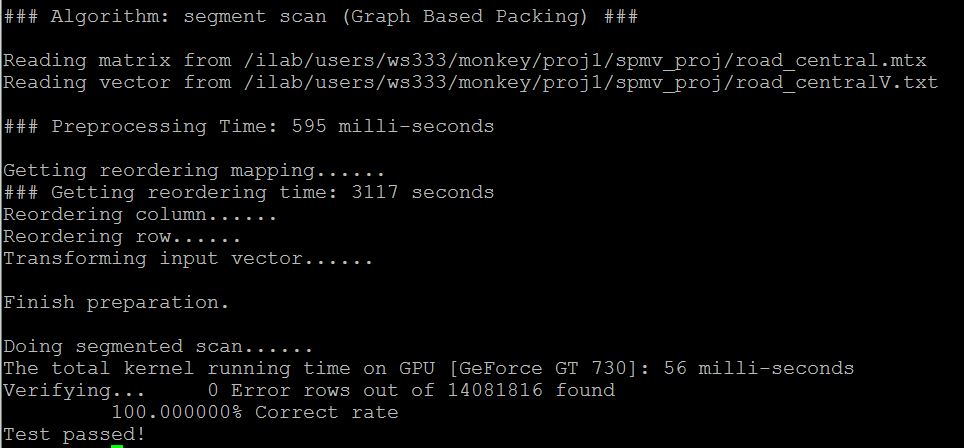
il2010: Graph Based Packing



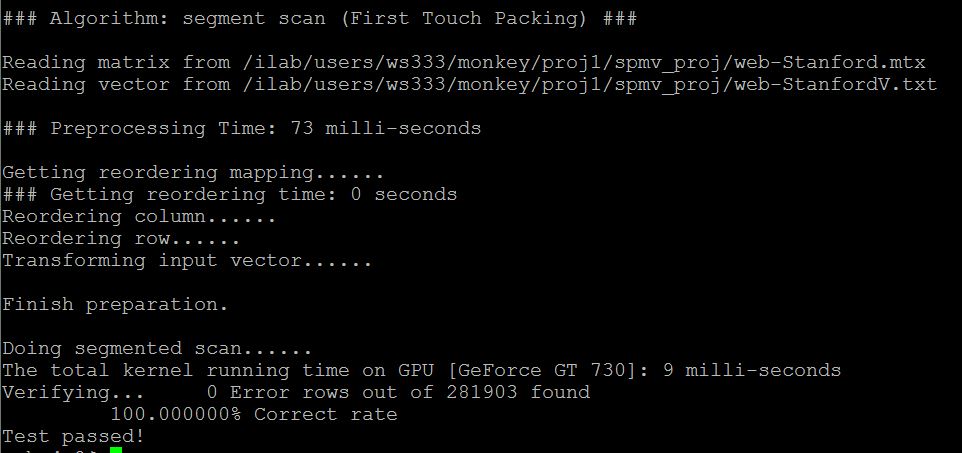
road central: First Touch Packing



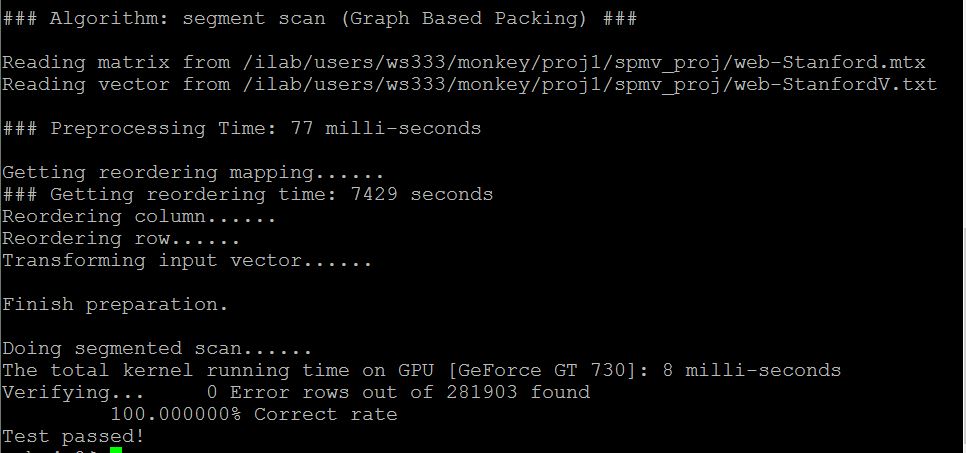
road central: First Touch Packing



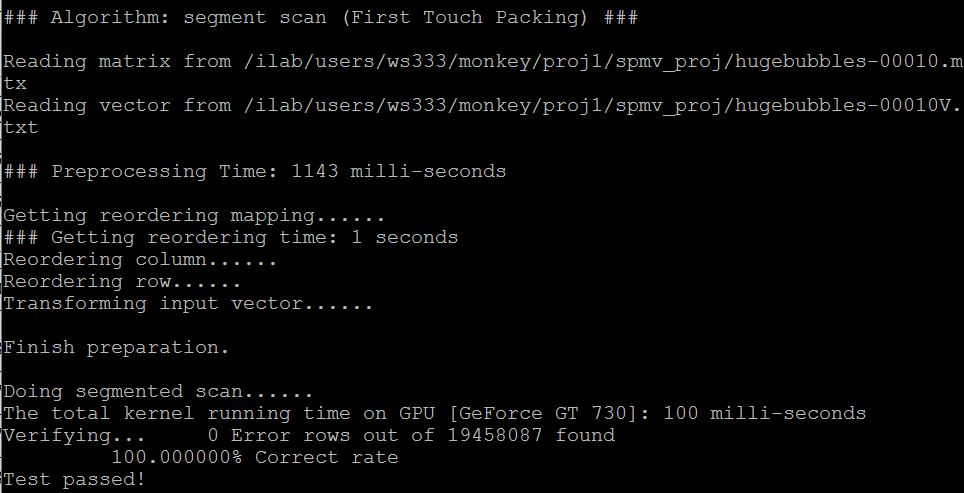
web-Stanford: First Touch Packing



web-Stanford: Graph Based Packing

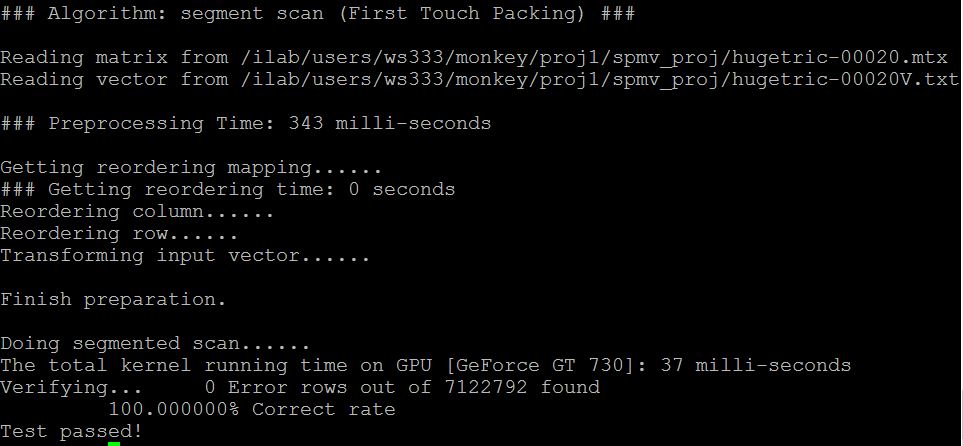


hugebubbles-00010: First Touch Packing

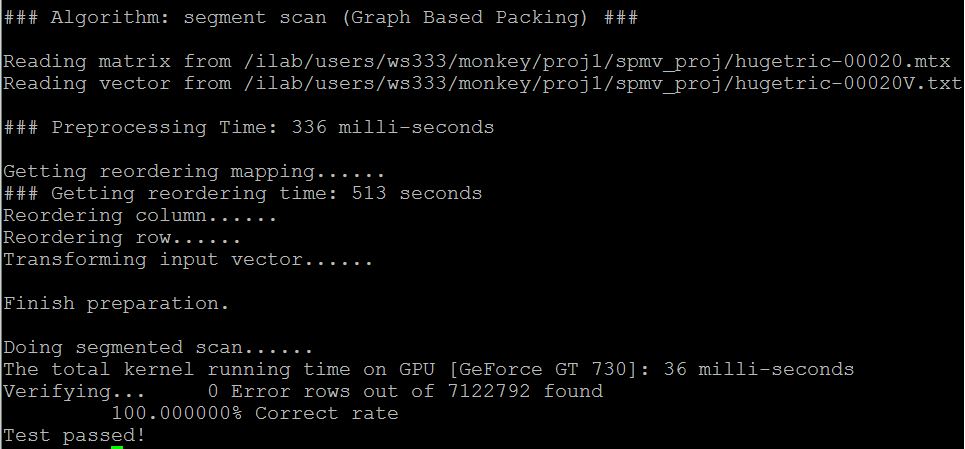


hugebubbles-00010: Graph Based Packing

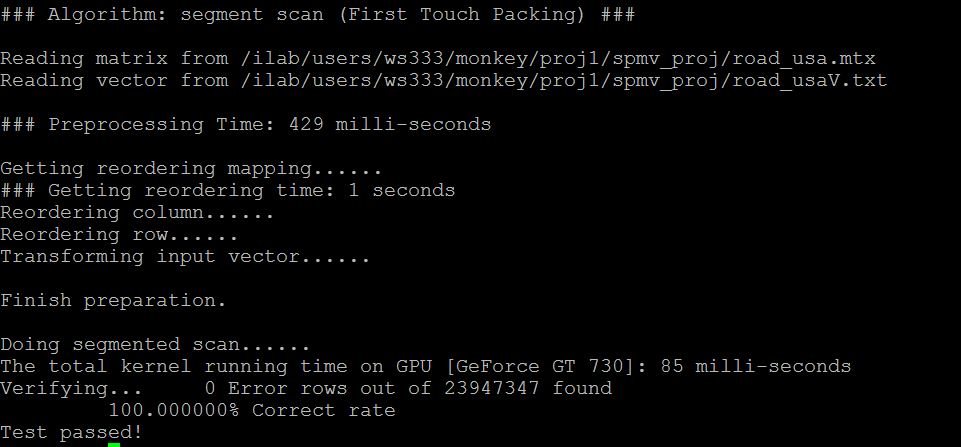
hugetric-00020: First Touch Packing



hugetric-00020: Graph Based Packing

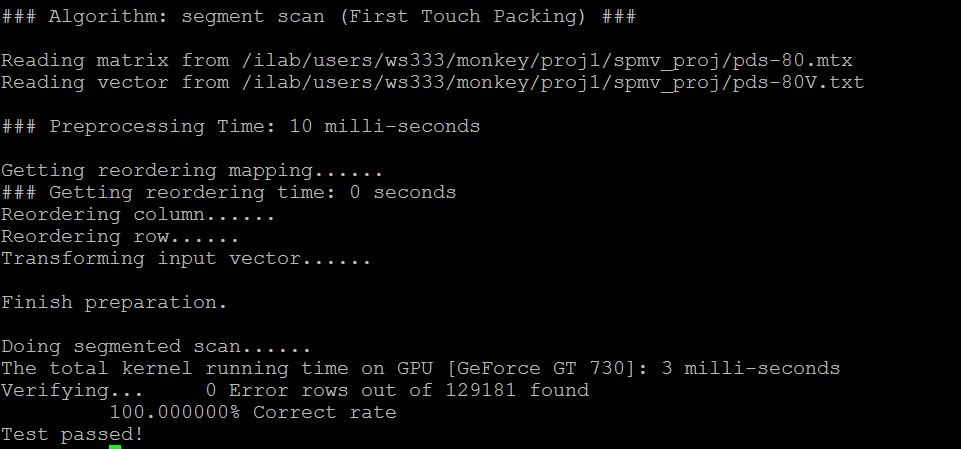


Road\_usa: First Touch Packing

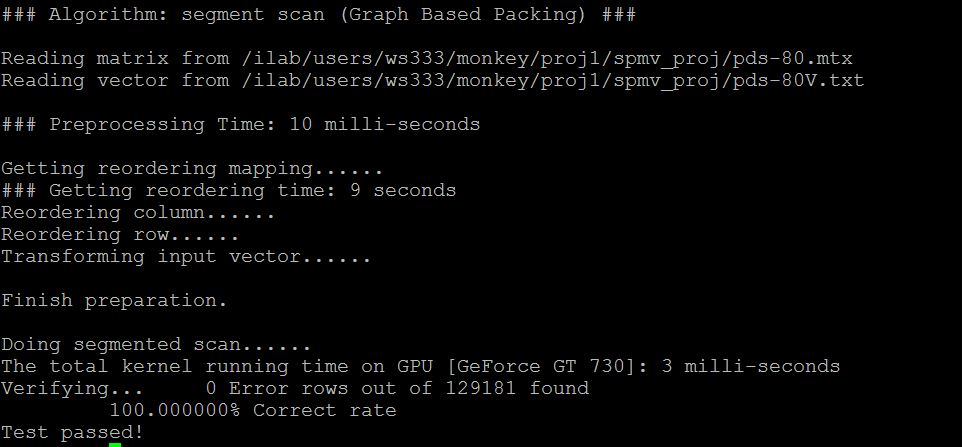


Road\_usa: Graph Based Packing

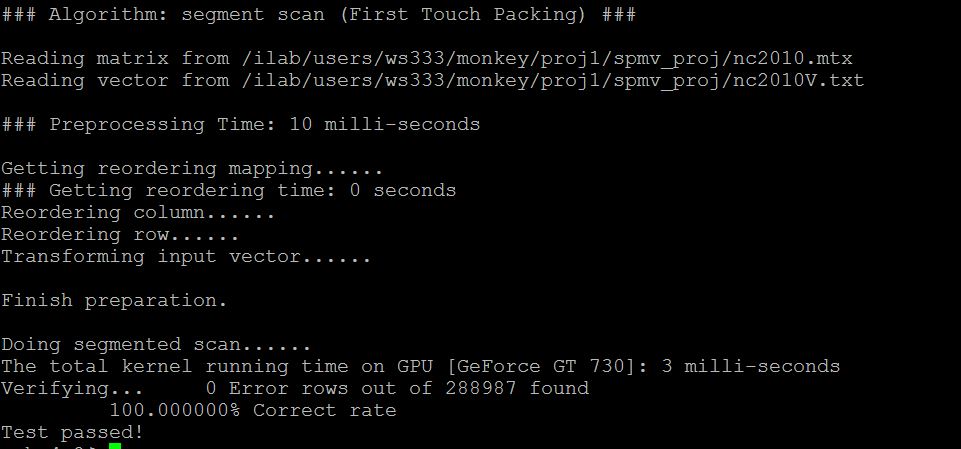
pds-80: First Touch Packing



pds-80: First Touch Packing



nc2010: First Touch Packing



nc2010: Graph Based Packing

