Solving 0-1 KNAPSACK PROBLEM USING CUDA Platform

CSE 708 Seminar: Programming Massively Parallel Systems

Instructor: Professor Russ Miller

Author: Pushkar Pandey

University at Buffalo The State University of New York

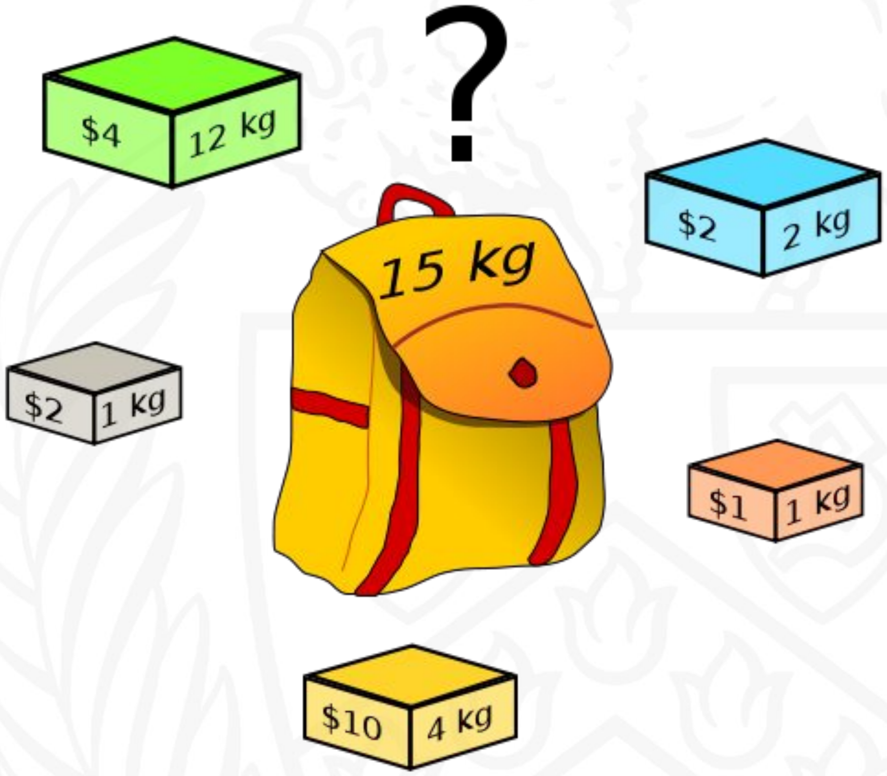
CONTENT:

- Introduction to 0-1 Knapsack Problem
- 0-1 Knapsack Problem Example
- Sequential Implementation
- Parallel Implementation: MPI
- CUDA Implementation
- Output Analysis and Graphs
- MPI vs Cuda
- Conclusion
- References

Introduction to 0-1 Knapsack Problem

- Problem of combinatorial optimization
- A set of items with a weight and a value given a knapsack with a maximum weight it can carry

Find which items to take to get the best value but not exceed the knapsack capacity



Example of Knapsack Problem

0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

Sequential Implementation

```
// Input:
// Values (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)
// NOTE: The array "v" and array "w" are assumed to store all relevant values starting at index 1.

array m[0..n, 0..W];
for j from 0 to W do:
    m[0, j] := 0
for i from 1 to n do:
    m[i, 0] := 0

for i from 1 to n do:
    for j from 0 to W do:
        if w[i] > j then:
            m[i, j] := m[i-1, j]
        else:
            m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])
```

Sequential Implementation Example

i	*v*	*w*
1	5	4
2	4	3
3	3	2
4	2	1

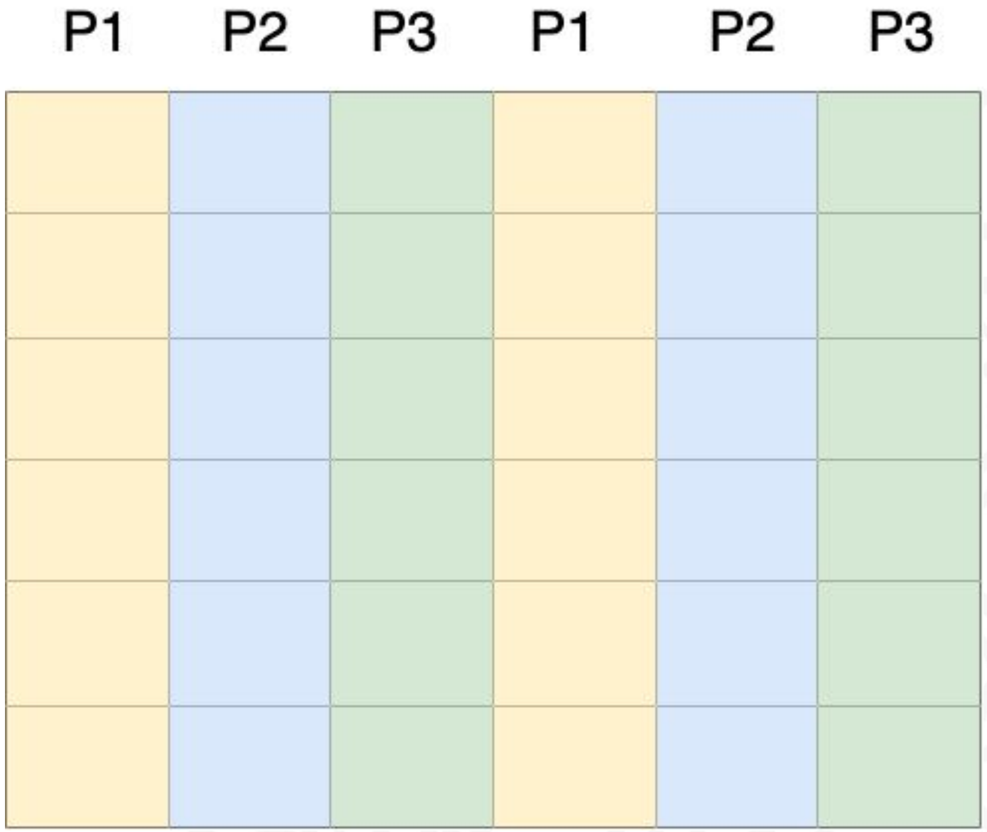
Capacity=6

	w						
	0	1	2	3	4	5	6
i 0							
1							
2							
3							
4							

MPI Parallel Implementation

We do column parallelization

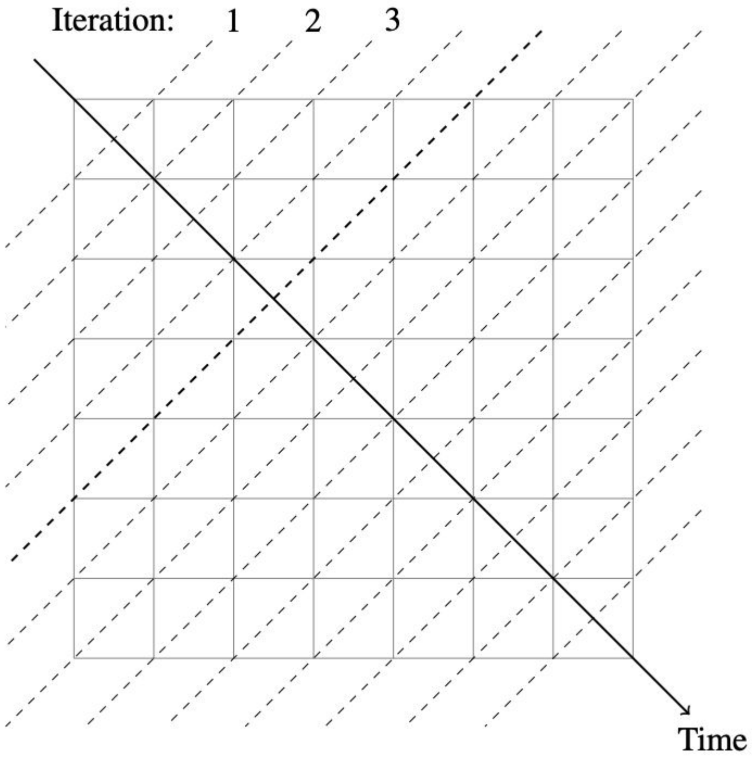
- Compute the maximum value achievable using the item of the row
- Compute the value without the new item. This value is the value just above in the matrix or 0 if it is the first item.
- Save in the cell the maximum value achievable using or not the new item
- Send to all the processors that could need it in future iteration the new value.



CUDA Parallel Implementation

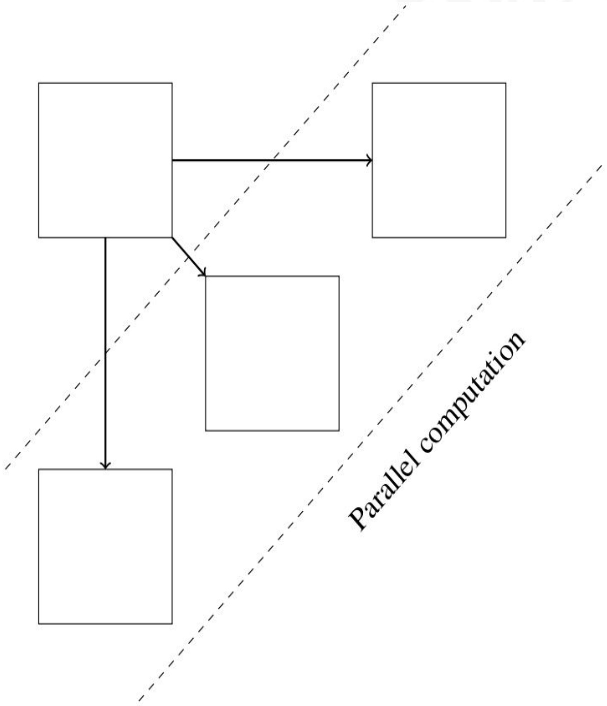
Anti-Diagonal Approach

- We Iterating through the dynamic programming scoring grid in an anti diagonal process.
- Each dotted line represents an iteration that is processed in parallel.



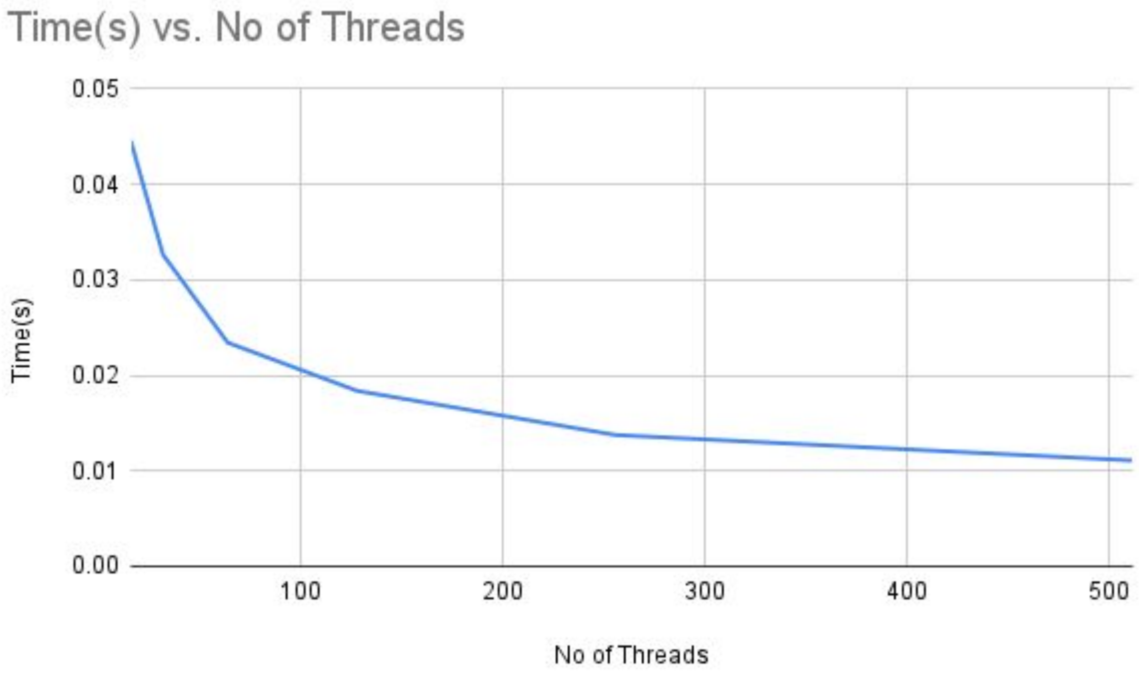
CUDA Parallel Implementation

- As a cell being filled satisfies the dependencies of future cells, it allows the elements of a diagonal iteration of the current grid, to be calculated and filled in parallel.
- An example of a cell in the current grid only having data dependencies to the previous iterations.



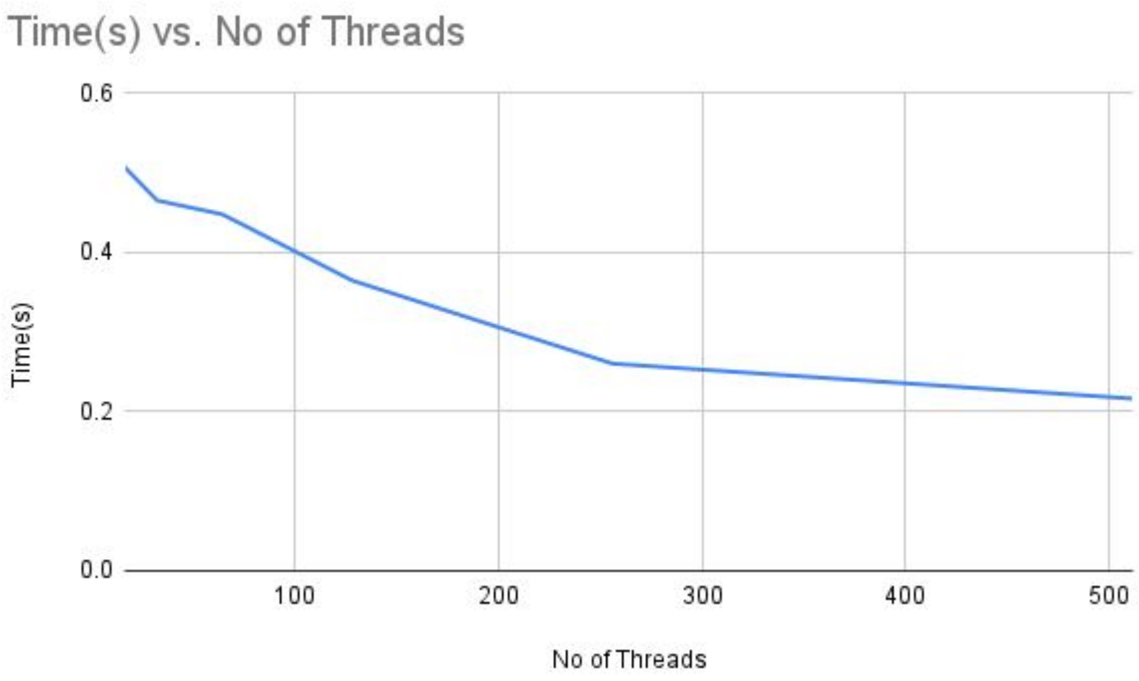
Output Analysis for W(100000/10000)

No of Threads	Time(s)
16	0.04461
32	0.03264
64	0.02343
128	0.01838
256	0.01375
512	0.01108



Output Analysis for W(500000/10000

No of Threads	Time(s)
16	0.5073
32	0.4654
64	0.4481
128	0.3648
256	0.2604
512	0.2164



MPI vs Cuda for W(500000/10000)

No of Threads/Nodes	Time(s)	Time(s)
16	0.5073	101.475
32	0.4654	64.479
64	0.4481	43.489

Conclusion

- As the thread count increases per block the code executing becomes faster.
- Cuda is a shared memory paradigm, which makes the algorithm easy and faster. MPI is a distributed memory and all synchronization and communication are explicit.

References:

- https://en.wikipedia.org/wiki/Knapsack_problem
- https://developer.nvidia.com/cuda-toolkit
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

Thanks You