Solving 0-1 KNAPSACK PROBLEM USING CUDA Platform

CSE 708 Seminar: Programming Massively Parallel Systems
Instructor: Professor Russ Miller
Author: Pushkar Pandey
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

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

Solution: 220

\[\text{Weight} = 10; \text{Value} = 60;\]
\[\text{Weight} = 20; \text{Value} = 100;\]
\[\text{Weight} = 30; \text{Value} = 120;\]
\[\text{Weight} = (20+10); \text{Value} = (100+60);\]
\[\text{Weight} = (30+10); \text{Value} = (120+60);\]
\[\text{Weight} = (30+20); \text{Value} = (120+100);\]
\[\text{Weight} = (30+20+10) > 50\]
Sequential Implementation

```java
// 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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>v</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Capacity** = 6

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.
CUFAQ Parallel Implementation

Anti-Diagonal Approach

- We iterat in 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 $W(100000/10000)$

<table>
<thead>
<tr>
<th>No of Threads</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.04461</td>
</tr>
<tr>
<td>32</td>
<td>0.03264</td>
</tr>
<tr>
<td>64</td>
<td>0.02343</td>
</tr>
<tr>
<td>128</td>
<td>0.01838</td>
</tr>
<tr>
<td>256</td>
<td>0.01375</td>
</tr>
<tr>
<td>512</td>
<td>0.01108</td>
</tr>
</tbody>
</table>
Output Analysis for $W(500000/10000$
**MPI vs Cuda for W(500000/10000)**

<table>
<thead>
<tr>
<th>No of Threads/Nodes</th>
<th>Time(s)</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.5073</td>
<td>101.475</td>
</tr>
<tr>
<td>32</td>
<td>0.4654</td>
<td>64.479</td>
</tr>
<tr>
<td>64</td>
<td>0.4481</td>
<td>43.489</td>
</tr>
</tbody>
</table>
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:

Thanks You