Data-Parallel Programming
using SaC
lecture 4

F21DP Distributed and Parallel Technology

Sven-Bodo Scholz
Looking at Entire Programs:
Multi-Threaded Execution on SMPs
SMP Ideal: Coarse Grain!
Multi-Threaded Execution on GPGPUs
GPGPU Ideal: Homogeneous Flat Coarse Grain
GPUs and synchronous memory transfers

LatticeBoltzmann CUDA vs. SaC Speedups (Tesla)

- Speedup
- Problem Size
- 10 Steps
- 25 Steps
- 50 Steps
- 100 Steps
- 200 Steps

Heriot-Watt University
GPGPU Naive Compilation

\[ PX = \{ [i,j] \rightarrow PX[ i,j ] + \text{sum} ( VY[.,i] \times CX[ i,j ] ) \}; \]

\[ PX_d = \text{host2dev}( PX); \]
\[ CX_d = \text{host2dev}( CX); \]
\[ VY_d = \text{host2dev}( VY); \]

\[ PX_d = \{ [i,j] \rightarrow PX_d[ i,j ] + \text{sum} ( VY_d[.,i] \times CX_d[ i,j ] ) \}; \]

\[ PX = \text{dev2host}( PX_d); \]
Hoisting memory transfers out of loops

```c
for( i = 1; i< rep; i++) {
    PX_d = host2dev( PX);
    CX_d = host2dev( CX);
    VY_d = host2dev( VY);
    PX = dev2host( PX_d);
}
```
Retaining Arrays on the Device

CX_d = { [i,j] -> ..... } ;

CX = dev2host(CX_d);

PX_d = host2dev(PX);
VY_d = host2dev(VY);

for( i = 1; i< rep; i++) {

The Impact of Redundant Memory Transfers
Challenge: Memory Management: What does the \( \lambda \)-calculus teach us?

\[
f(a, b, c) \rightarrow \{ \text{... a..... a....} \text{b.....b....c...} \}
\]
How do we implement this? – the scalar case

\[ f( a, b, c ) \]

\[
\begin{aligned}
\text{conceptual copies} \\
\end{aligned}
\]

\[
f( a, b, c ) \\
\{ \\
\quad \text{... a..... a....b.....b....c...} \\
\} \\
\]

<table>
<thead>
<tr>
<th>operation</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read from stack</td>
</tr>
<tr>
<td>funcall</td>
<td>push copy on stack</td>
</tr>
</tbody>
</table>
How do we implement this? – the non-scalar case

naive approach

conceptual copies

f(a, b, c)  

f(a, b, c) 

{  

... a..... a....b.....b....c...  

}

<table>
<thead>
<tr>
<th>operation</th>
<th>non-delayed copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>O(1) + free</td>
</tr>
<tr>
<td>update</td>
<td>O(1)</td>
</tr>
<tr>
<td>reuse</td>
<td>O(1)</td>
</tr>
<tr>
<td>funcall</td>
<td>O(1) / O(n) + malloc</td>
</tr>
</tbody>
</table>
How do we implement this? – the non-scalar case widely adopted approach

\[ f(a, b, c) \]

<table>
<thead>
<tr>
<th>operation</th>
<th>delayed copy + delayed GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>update</td>
<td>( O(n) + \text{malloc} )</td>
</tr>
<tr>
<td>reuse</td>
<td>( \text{malloc} )</td>
</tr>
<tr>
<td>funcall</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>
How do we implement this? – the non-scalar case reference counting approach

conceptual copies

\[ f(a, b, c) \]

\[
\begin{array}{l}
\text{operation} \\
\text{read} \\
\text{update} \\
\text{reuse} \\
\text{funcall}
\end{array}
\begin{array}{l}
\text{delayed copy + non-delayed GC} \\
O(1) + \text{DEC}_\text{RC}_\text{FREE} \\
O(1) / O(n) + \text{malloc} \\
O(1) / \text{malloc} \\
O(1) + \text{INC}_\text{RC}
\end{array}
How do we implement this? – the non-scalar case

A comparison of approaches

<table>
<thead>
<tr>
<th>operation</th>
<th>non-delayed copy</th>
<th>delayed copy + delayed GC</th>
<th>delayed copy + non-delayed GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>O(1) + free</td>
<td>O(1)</td>
<td>O(1) + DEC_RC_FREE</td>
</tr>
<tr>
<td>update</td>
<td>O(1)</td>
<td>O(n) + malloc</td>
<td>O(1) / O(n) + malloc</td>
</tr>
<tr>
<td>reuse</td>
<td>O(1)</td>
<td>malloc</td>
<td>O(1) / malloc</td>
</tr>
<tr>
<td>funcall</td>
<td>O(1) / O(n) + malloc</td>
<td>O(1)</td>
<td>O(1) + INC_RC</td>
</tr>
</tbody>
</table>
Avoiding Reference Counting Operations

a = [1,2,3,4];

b = a[1];

c = f(a, 1);

d = a[2];
e = f(a, 2);

We would like to avoid RC here!

Clearly, we can avoid RC here!

We cannot avoid RC here!
NB: Why don’t we have RC-world-domination?
Going Multi-Core

local variables do not escape!
relatively free variables can only benefit from reuse in 1/n cases!

=> use thread-local heaps
=> inhibit rc-ops on rel-free vars
Bi-Modal RC:

fork

local

join

norc
Conclusions

• There are still many challenges ahead, e.g.
  ➢ Non-array data structures
  ➢ Arrays on clusters
  ➢ Joining data and task parallelism
  ➢ Better memory management
  ➢ Application studies

• If you are interested in joining the team:
  ➢ talk to me 😊
Going Multi-Core II

Local variables do escape!
Relatively free variables can benefit from reuse in 1/2 cases!

=> use locking....
Going Many-Core

256 cores
500 threads in HW each

functional programmers paradise, no?! 

nested DP and TP parallelism
RC in Many-Core Times

computational thread(s)

rc-op

rc-op

rc-op

rc-op

RC-thread
and here the runtimes

Execution on 1 cores

pipeline utilization
active threads
exclusive queue size

Execution on 4 cores

pipeline utilization
active threads
exclusive queue size

Execution on 16 cores

pipeline utilization
active threads
exclusive queue size

Execution on 64 cores

pipeline utilization
active threads
exclusive queue size
Multi-Modal RC:

- Local
- Async
- Norc

- Spawn
- Rc==1
- Create/Sync
new runtimes: