Producer-Consumer Model for Massively Parallel Zero-Sum Games on the GPU

Avi Bleiweiss
NVIDIA Corporation
Zero-Sum

- One’s gain ⇒ other’s loss
- Perfect info
- Multi player game
  - Simple and involved

<table>
<thead>
<tr>
<th>Matching Pennies</th>
<th>Head</th>
<th>Tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head</td>
<td>1, -1</td>
<td>-1, 1</td>
</tr>
<tr>
<td>Tail</td>
<td>-1, 1</td>
<td>1, -1</td>
</tr>
</tbody>
</table>
Motivation

- Play as you go
  - Thousands players
- Game cloud
  - GPU computing
- Mobile computer
  - Commodity

NVIDIA Tesla

NVIDIA Tegra
Problem

Game
- Two player
- Maximize look ahead
- Rapid node expansion
  - $10^{25}$ for 4x4x4 Tic-Tac-Toe
  - $10^{120}$ for Chess

Search
- Efficient parallel
  - Space split
  - Statistical simulation
  - Simultaneous matches
- Many thousands threads
Parallelism

Principal Variation Split
- Strongly ordered tree
- Synchronization bound
- Load imbalance

Young Brothers Wait Concept
- Parallel at any node
- Processor owns node
- Up to 1024 processors

Dynamic Tree Splitting
- Processors share node
- Global job list
- Reasonable speedup

No massive parallel solution in shared memory settings!
Cut Nodes

- PV Bounds
- Equal weight
- Static split
Heuristic Search

\[ \text{AlphaBeta}(v, \alpha_l, \beta_l, \alpha_g, \beta_g) \]

\[
\begin{align*}
\alpha_l & \leftarrow \max(\alpha_l, \alpha_g) \\
\beta_l & \leftarrow \min(\beta_l, \beta_g) \\
\text{if} \ (\alpha_l > \beta_l) & \text{ return } \text{rank} \\
\text{if} \ (s(v) = \emptyset) & \text{ return } f(v) \\
\text{foreach} \ (v_j \ni s(v)) & \text{ do} \\
 & \text{ value } \leftarrow \text{AlphaBeta}(v_j, \alpha_l, \beta_l, \alpha_g, \beta_g) \\
 & \text{if} \ (p(v) == \text{First}) \text{ then} \\
 & \quad \text{rank } \leftarrow \max(\text{rank}, \text{value}) \\
 & \quad \alpha_l \leftarrow \max(\alpha_l, \text{rank}) \\
 & \quad \text{if} \ (\alpha_l > \alpha_g) \text{ then } \alpha_g \leftarrow \alpha_l \\
 & \text{else} \\
 & \quad \text{rank } \leftarrow \min(\text{rank}, \text{value}) \\
 & \quad \beta_l \leftarrow \min(\beta_l, \text{rank}) \\
 & \text{if} \ (\beta_l < \beta_g) \text{ then } \beta_g \leftarrow \beta_l \\
\text{return} \ \text{rank}
\end{align*}
\]
Statistical Simulation

- Monte Carlo method
  - Annealing process
- \( k \)-simulation move error
  \[
  \Delta f(v) \sim \frac{1}{\sqrt{k}}
  \]
- Many thousands games
  - for 19x19 Go game
Challenges

Deep recursion, limited stack

Divergent, irregular threads

Dynamic parallelism

Low arithmetic intensity
Implementation

- Kernel for each
  - Alpha-Beta, Monte Carlo
- Board C++ class
  - Rules specific
- Games

<table>
<thead>
<tr>
<th>Heuristic Search</th>
<th>3D Tic-Tac-Toe</th>
<th>Connect-4</th>
<th>Reversi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monte Carlo Simulation</td>
<td>Go</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
State Abstraction

Cells

Successors

Player

Move

Manipulate

Update

Query

Undo

Winner

Full

Pitched 2D/3D

Player

{none, 1st, 2nd}

Rank

{Win, Lose, Draw}

Player

struct Move {
int3 position;
int rank;
};
Stack

- Recursion depth >1000
- Greedy allocation

\[ \text{StackBytes/Thread} \times \text{SM #} \times \text{MaxWarps/SM} \times \text{Threads/Warp} \]

- Hybrid design

Local Memory
- Local variables
- Function parameters

Runtime/Compiler

User
- Successors

Global Memory
Random trial moves
Parallel games
Scoring trial moves
Find highest cut nodes
Parallel node search
Rank Reduction-Max

producer
consumer

Game Tree
Thousands of working threads

foreach move
game init
game over

CPU
GPU

Producer-Consumer

Producers and consumers work together to process the game.

Thousands of working threads process the game tree.
Shared αβ

```c
__device__ int *galpha, *gbeta;

__device__ void
resolve(int& alpha, int& beta)
{
    if(alpha <= *galpha) alpha = *galpha;
    else atomicMax(galpha, alpha);

    if(beta >= *gbeta) beta = *gbeta;
    else atomicMin(gbeta, beta);
}
```
Limitations

- Stack allocation
  - Bounds parallelism
- Static split constraints

<table>
<thead>
<tr>
<th>Depth</th>
<th>Tic-Tac-Toe Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3x3x3</td>
</tr>
<tr>
<td>1</td>
<td>650</td>
</tr>
<tr>
<td>2</td>
<td>15600</td>
</tr>
</tbody>
</table>
Methodology

- CUDA Toolkit 3.1, Windows
- Single processor

<table>
<thead>
<tr>
<th>GPU</th>
<th>SMs</th>
<th>Warps/SM</th>
<th>Clocks(MHz)</th>
<th>L1/Shared (KB)</th>
<th>L2(KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX480</td>
<td>15</td>
<td>2</td>
<td>723/1446/1796</td>
<td>48/16</td>
<td>640</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
<th>Cores</th>
<th>Clocks(MHz)</th>
<th>L1/L2 (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I7-940</td>
<td>8</td>
<td>2942/(3*1066)</td>
<td>32/8192</td>
</tr>
</tbody>
</table>
Space Split

4x4x4 Tic-Tac-Toe

- Naïve Split
- Shared αβ

lower is good
Monte Carlo

Average Seconds/Move vs Board Dimension

Go Running Time

- 128
- 1024
- 4096
- 8192
- 16384

lower is good
Simultaneous Matches

Multi Game Running Time

- 3D Tic-Tac-Toe
- Connect-4
- Reversi

Average Seconds / Move

Matches

lower is good
Future Work

- Data packing optimization
- Per-SM transposition table
- Predictable node ranks
- Dynamic space split
# GPU Performance

<table>
<thead>
<tr>
<th>Metric</th>
<th>Game</th>
<th>Dimension</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared αβ vs. Naïve Split</td>
<td>3D Tic-Tac-Toe</td>
<td>4x4x4</td>
<td>13.37X mean</td>
</tr>
<tr>
<td>Monte Carlo vs. CPU</td>
<td>Go</td>
<td>19x19</td>
<td>121.64X @16K</td>
</tr>
</tbody>
</table>
Summary

- Efficient GPU based
  - Heuristic search
  - Statistical Simulation
- Economical solution
  - Mobile-Cloud
Thank You!

Questions?
Info

▪ Base

▪ GPU AI for Board Games
  – Technology Preview

▪ Toolkit
  – CUDA Zone

▪ Debugger
  – Parallel Nsight