FPGA-Based K-Means Clustering Using Tree-Based Data Structures

Felix Winterstein, Samuel Bayliss, George Constantinides
2 September 2013
• Application in
  – Unsupervised learning
  – Data mining
  – Pattern recognition
  – Tracking
  – Image quantisation (colour space)
Clustering

- Application in
  - Unsupervised learning
  - Data mining
  - Pattern recognition
  - Tracking
  - Image quantisation (colour space)
- Automatic partitioning
- K-means clustering widely used

N data points
K-means clustering

- K is a known parameter (e.g. K=4)
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning

\[ \| x_i - z_1 \|^2 \]
K-means clustering

• K is a known parameter (e.g. K=4)
• Clusters represented by their centres
• Centre position determines cluster assignment
• Find optimal positioning
• Iterative refinement of initial placement

N data points
• K is a known parameter (e.g. K=4)
• Clusters represented by their centres
• Centre position determines cluster assignment
• Find optimal positioning
• Iterative refinement of initial placement
K-means clustering

- K is a known parameter (e.g. K=4)
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning
- Iterative refinement of initial placement

N data points
- K is a known parameter (e.g. K=4)
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning
- Iterative refinement of initial placement
Brute-force Algorithm

- Simple control flow
- Embarrassingly parallel
- Well-suited for FPGAs:
  

```plaintext
function CONVENTIONAL(
  for all $x_i \in \{x_1, x_2, ..., x_N\}$ do
    for all $z_j \in \{z_1, z_2, ..., z_K\}$ do
      $d^2 \leftarrow \|x_i - z_j\|^2$
      $z^* \leftarrow$ closest centre to $x_i$
    updateCentreBuffer($z^*$)
  end for
end for
end function
```
function FILTER(treeNode u, CentreSet Z)

\[ z^* \leftarrow \text{closest centre } \epsilon Z \text{ to } u.\text{midPoint} \]

if \( u \) is leaf then
  updateCentreBuffer(\( z^* \))
else
  \( newZ \leftarrow \text{pruneSet}(Z, z^*, u.\text{bndBox}) \)
  if \( |newZ| > 1 \) then
    FILTER(\( u.\text{left}, newZ \))
    FILTER(\( u.\text{right}, newZ \))
  end if
end if

end function

Filtering Algorithm

\[ Z = \{ z_1, z_2, z_3, z_4 \} \]
\[ Z' = \{ z_2, z_3 \} \]
\[ Z'' = \{ z_2 \} \]

- Recursive tree traversal
- Dynamic data structure

**function** FILTER(treeNode u, CentreSet Z)

\[ z^* \leftarrow \text{closest centre } \epsilon Z \text{ to } u.\text{midPoint} \]

if \( u \) is leaf then
  updateCentreBuffer(\( z^* \))
else
  \[ \text{newZ} \leftarrow \text{pruneSet}(Z, z^*, u.\text{bndBox}) \]
  if \( |\text{newZ}| > 1 \) then
    FILTER(\( u.\text{left}, \text{newZ} \))
    FILTER(\( u.\text{right}, \text{newZ} \))
  end if
end if

end function

---

Clear Advantage in SW

Node-centre interactions

Computational complexity (distance comp. equivalents)

N=16384, K=128, D=3

Tree traversal: 98.8% of total computation
Outline

• Pointer-based dynamic data structures on FPGA
• Pipelining and parallelisation
• Dynamic memory access
• Maintaining efficiency
• Conclusion
Why unusual in HW

- Recursion
  - Implement customised stack
- Benefits of a hardware implementation
  - Pipeline and parallelise recursive instances
- Each instance ‘creates’ a new new \( Z \) (under data-dependent condition)
  - Manage dynamic allocation of scratchpad memory

```python
function FILTER(treeNode \( u \), CentreSet \( Z \))
  \( z^* \leftarrow \) closest centre \( \epsilon \ Z \) to \( u. \) midPoint
  if \( u \) is leaf then
    updateCentreBuffer(\( z^* \))
  else
    \( newZ \leftarrow \) pruneSet(\( Z \), \( z^* \), \( u. \) bndBox)
    if \( |newZ| > 1 \) then
      FILTER(\( u. \) left, newZ)
      FILTER(\( u. \) right, newZ)
    end if
  end if
end function
```
Outline

• Pointer-based dynamic data structures on FPGA
• Pipelining and parallelisation
• Dynamic memory access
• Maintaining efficiency
• Conclusion
Overlap recursive instances in the pipeline

Stack content:
- \( \text{lt1: \{s1, s1\}} \)
- \( \text{lt2: \{s2, s2, s1\}} \)
- ...

- All items on stack can be processed independently
- Dependence analysis shows we can pipeline recursive instances
- Further parallelism by splitting into sub-trees
Outline

• Pointer-based dynamic data structures on FPGA
• Pipelining and parallelisation
• Dynamic memory access
• Maintaining efficiency
• Conclusion
Scratchpad memory

Centre set 0
Centre set 1

...
Worst-case amount of memory

- Worst-case: N-1 sets
- N=16384, K=256:
  -> 32 Mbits (81% of BRAMs in medium-size Virtex 6 FPGA)
Average amount of memory

• Worst-case: N-1 sets
• N=16384, K=256:
  -> 32 Mbits (81% of BRAMs in medium-size Virtex 6 FPGA)
Average amount of memory

- Worst-case: N-1 sets
- \( N=16384, K=256 \):
  \[ \text{\textgreater~} 32 \text{ Mbits (81\% of BRAMs in medium-size Virtex 6 FPGA)} \]
Average amount of memory

- Worst-case: N-1 sets
- N=16384, K=256:
  -> 32 Mbits (81% of BRAMs in medium-size Virtex 6 FPGA)
Average amount of memory

- Worst-case: N-1 sets
- N=16384, K=256:
  -> 32 Mbits (81% of BRAMs in medium-size Virtex 6 FPGA)
- Reuse memory space
- Dynamic memory allocation
Dynamic memory allocation

- Worst-case: N-1 sets
- N=16384, K=256:
  -> 32 Mbits (81% of BRAMs in medium-size Virtex 6 FPGA)
- Reuse memory space
- Dynamic memory allocation
Dynamic memory allocation

- Worst-case: $N-1$ sets
- $N=16384, K=256$: 
  -> 32 Mbits (81% of BRAMs in medium-size Virtex 6 FPGA)
- Reuse memory space
- Dynamic memory allocation
- How to support the worst case?
Dynamic memory allocation

- Worst-case: N-1 sets
- N=16384, K=256:
  
  - \( \rightarrow 32 \text{ Mbits (81\% of BRAMs in medium-size Virtex 6 FPGA)} \)
- Reuse memory space
- Dynamic memory allocation
- How to support the worst case?
- Degrades pruning, degrades runtime
Memory size vs. Runtime

Bound on memory size [number of centre sets]

Run-time

[Node-centre pairs x100k]

Dynamic Alloc.
4 BRAMs

Static Alloc.
512 BRAMs

N=16384, K=128, \( \sigma=0.2 \)
Outline

• Pointer-based dynamic data structures on FPGA
• Pipelining and parallelisation
• Dynamic memory access
• Maintaining efficiency
• Conclusion
Performance comparison

**Node-centre interactions (x1M)**

- **SW**
  - Brute-f.: 0
  - Filt.: 0
  - Multiplier: x 24

- **FPGA**
  - Brute-f.: 0
  - Filt.: 0
  - Multiplier: x 21

**Distance comp. Equivalents (x1M)**

- **SW**
  - Brute-f.: 0
  - Filt.: 0
  - Multiplier: x 8

- **FPGA**
  - Brute-f.: 0
  - Filt.: 0

**Execution time [ms]**

- **SW**
  - Brute-f.: 10
  - Filt.: 0

- **FPGA**
  - Brute-f.: 10
  - Filt.: 0

**Resource consumption [%]**

- **SW**
  - Slices: 50
  - DSP Slices: 30
  - BRAMs: 20

- **FPGA**
  - Slices: 50
  - DSP Slices: 30
  - BRAMs: 20

Throughput constraint: 128us / clustering iteration

N=16384, K=128, D=3
• Pointer-based recursive algorithm using dynamic data structures on FPGA

• Specialised pipeline:
  – Benefit from hardware implementation
  – Maintain efficiency seen in SW comparison

• Custom dynamic memory allocation:
  – Use memory space efficiently

• Future work:
  – Interface to off-chip memory
  – Investigate implications for research on high-level synthesis tools
Thank you.
Questions?