An OpenCL Implementation of Wait-Free Sets

Sudarshan Shinde
Bangalore, INDIA.
Email:sudarshan_shinde@iitbombay.org

Abstract

Index Terms

wait-free programming, multithreading, gpgpu, opencl.

I. Implementation Details

The algorithms described in the literarure do not describe memory management. In perticular, they do not take into consideration, what happens if a node is physically deleted, while another process it using that node.

OpenCL 2.0 defines Shared Virtual Memory (SVM), that could be used to share data between host and the GPU device, without explicit data transfer. However OpenCL does not support dynamic data allocation on GPU side, and requires that size of SVM be pre-defined. This restricts maximum size of a set.

Since an implementation needs to take care of memory management, in this implementation we allocate a node pool of size N in SVM. Though this restricts set size to be less than N, it still highlights many implementation aspects of wait-free set implementation in heterogenous computing setting.

II. The Algorithm

The algorithm consists of a modification of the wait-free linked list to support replacing a node.

A *packed* reference consists of an unmarked reference, a *mark* bit to indicate if the node is marked and an *retain* bit to indicate that unmarked reference could not be changed. It also has a *free* bit to indicate that whether the node is occupied or free. These three bits are arranged as $Bits = [fBit, rBit, mBit]$.

We also have the following methods

1) $[ref, bits] = unpackRef(pref)$.

Algorithm 1 Snips next node if it is marked and could be snipped

```
1:  function SNIP(startRef)
2:      pPRef = startRef:next;
3:      [nRef,pBits] = unpackRef(startRef:next);
4:      if (pBits = [0,0,x]) then
5:          [nnRef,nBits] = unpackRef(nRef:next):
6:          if (nBits = [0,x,1]) then
7:              status = CAS(startRef:next,pPRef,[nnRef,pBits]);
8:              if (status = true) then
9:                  FREE(nRef);
10:                 return snipped;
11:             else
12:                 return failed;
13:             end if
14:         else
15:             return next_unmarked;
16:         end if
17:     else
18:         return start_invalid;
19:     end if
20: end function
```

Algorithm 2 Replaces next node with a new node

```
1:  function REPLACE(startRef, newRef)
2:      pPRef = startRef:next;
3:      [nRef,pBits] = unpackRef(startRef:next);
4:      if ([pBits] = [0,0,0]) then
5:          nPRef = nRef:next;
6:          [nnRef,nBits] = unpackRef(nRef:next);
7:          if (nBits = [0,x,0]) then
8:              status = CAS(nRef:next,nPRef,[nnRef,[0,1,0]]);
9:              if (status = false) then
10:                 return retain_failed;
11:             end if
12:         else
13:             return invalid_nbits;
14:         end if
15:         newRef:next = [nnRef,[0,0,0]];
16:         status = CAS(startRef:next,pPRef,[newRef,pBits]);
17:         if (status = true) then
18:             FREE(nRef);
19:             return replaced;
20:         else
21:             return failed;
22:         end if
23:     else
24:         return invalid_pbits;
25:     end if
26: end function
```

Algorithm 3 cleans all the nodes that are logically deleted and could be physically deleted

```
1:  function CLEAN(startRef, nextRef&)
2:      pRef = startRef;
3:      nextRef = null;
4:      while (true) do
5:          status = snip(pRef);
6:          [nRef,pBits] = unpackRef(pRef:next);
7:          if (status = next_unmarked) then
8:              nextRef = nRef;
9:              return status;
10:         else if status = start_invalid then
11:             if (pBits = [0,1,x]) then
12:                 pRef = nRef;
13:             else
14:                 if (pRef = startRef) then
15:                     return start_invalid
16:                 else
17:                     pRef = startRef;
18:                 end if
19:             end if
20:         end if
21:     end while
22: end function
```

Algorithm 4 checks for valid window.

```
1:  function WINDOW(key, prevRef, nextRef, index&)
2:      status = false;
3:      if (prevRef != Null) then
4:          [nRef, pBits] = unpackRef(prevRef:next);
5:          if (pBits != [0,x,0]) then
6:              return invalid_prev_bits; ;
7:          end if
8:          if (nRef != nextRef) then
9:              return invalid_next_ref;
10:         end if
11:         pMaxVal = simdMAX(prevRef:val);
12:         if (key <= pMaxVal) then
13:             return window_not_found;;
14:         end if
15:     end if
16:     [nnRef, nBits] = unpackRef(nextRef:next);
17:     if (nBits != [0,x,0]) then
18:         return invalid_next_bits;
19:     end if
20:     pMaxVal = simdMAX(nextRef:val);
21:     if (key <= pMaxVal) then
22:         return window_not_found;
23:     else
24:         if (simdANY((key = nextRef:Val), index)) then
25:             return key_found
26:         else
27:             return window_found
28:         end if
29:     end if
30: end function
```

Algorithm 5 finds a key and returns a window to it.

```
1:  function FIND(key, prevRef&, nextRef&, index&)
2:      startRef = hash(key);
3:      prevRef = nextRef = Null;
4:      index = 0;
5:      while (true) do
6:          pRef = Null;
7:          nRef = startRef;
8:          status = window_not_found;
9:          while (status = window_not_found) do
10:             status = WINDOW(key, pRef, nRef, index);
11:             if (status = window_not_found) then
12:                 pRef = nRef;
13:                 status2 = CLEAN(pRef,nRef);
14:                 if (status2 = start_invalid) then
15:                     status = window_not_found
16:                 end if
17:             end if
18:         end while
19:         if (status = window_found) then
20:             return status;
21:         end if
22:         if (status = key_found) then
23:             return status;
24:         end if
25:     end while
26: end function
```

Algorithm 6 clones a node and adds a key to it.

```
1:  function CLONE(ref, key)
2:      newRef = ALLOC();                          ▷ assumption is that ALLOC always returns a node
3:      newRef:next = ref:next;
4:      keyIndex = simdIndex(ref:val = key);
5:      if keyIndex = 0 then
6:          minIndex = min(simdIndex(ref:val = empty));      ▷ simdIndex returns zero if condition is not met
7:          if (minIndex ≠ 0) then
8:              simdCopy(ref:val, newRef:val);
9:              newRef:val[minIndex] = key;
10:             return newRef;
11:         else
12:             prevNewRef = ALLOC();
13:             simdCopy(ref:val, prevNewRef:val, (ref:val ¡ key));
14:             simdCopy(ref:val, newRef:val, (ref:val ¿ key));
15:             prevNewRef:next = [newRef,[0,0,0]];
16:             minIndex = min(simdIndex(newRef:val = empty));
17:             newRef:val[minIndex] = key;
18:             return prevNewRef;
19:         end if
20:     else
21:         simdCopy(ref:val, newRef:val);
22:         newRef:val[keyIndex] = empty;
23:         return newRef;
24:     end if
25:     return null;
26: end function
```

Algorithm 7 traverses the list from prev ref to next ref.

```
1:  function TRAVERSE(prevRef, nextRef)
2:      pRef = prevRef;
3:      [nRef, pBits] = unpackRef(pRef:next);
4:      while (pBits = [0,x,x]) do
5:          if (nRef ≠ nextRef) then
6:              pRef = nRef;
7:              [nRef, pBits] = unpackRef(pRef:next);
8:          else
9:              return pRef;
10:         end if
11:     end while
12:     return null;
13: end function
```

Algorithm 8 adds a key to the set.

```
1: function ADD(key)
2:     while (true) do
3:         status = FIND(key, prevRef nextRef,index);
4:         if status = window_found then
5:             pRef = TRAVERSE(prevRef, nextRef);
6:             if (pRef ≠ null) then
7:                 cloneRef = CLONE(nextRef,key);
8:                 status = REPLACE(pRef,cloneRef);
9:                 if (status = replaced) then
10:                    return true;
11:                end if
12:            end if
13:        else
14:            return false
15:        end if
16:    end while
17: end function
```

Algorithm 9 removes a key from the set.

```
1: function REMOVE(key)
2:     while (true) do
3:         status = FIND(key, prevRef nextRef,index);
4:         if status = key_found then
5:             pRef = TRAVERSE(prevRef, nextRef);
6:             if (pRef ≠ null) then
7:                 cloneRef = CLONE(nextRef,key);
8:                 status = REPLACE(pRef,cloneRef);
9:                 if (status = replaced) then
10:                    return true;
11:                end if
12:            end if
13:        else
14:            return false
15:        end if
16:    end while
17: end function
```