Gimbatulûk[1]

Accelerating Pattern Matching with OpenCL

Fraser Adams

This paper describes an approach for accelerating the common problem of pattern matching with a parallel variant of the Aho-Corasick algorithm implemented in OpenCL.

1 Gimbatulûk means “To find them all” derived from J. R. R. Tolkien's Black Speech inscription upon the *One Ring*: “*Ash nazg durbatulûk, ash nazg gimbatul, ash nazg thrakatulûk agh burzum-ishi krimpatul”* meaning; *One ring to rule them all, one ring to find them, one ring to bring them all and in the darkness bind them.*

1. Introduction

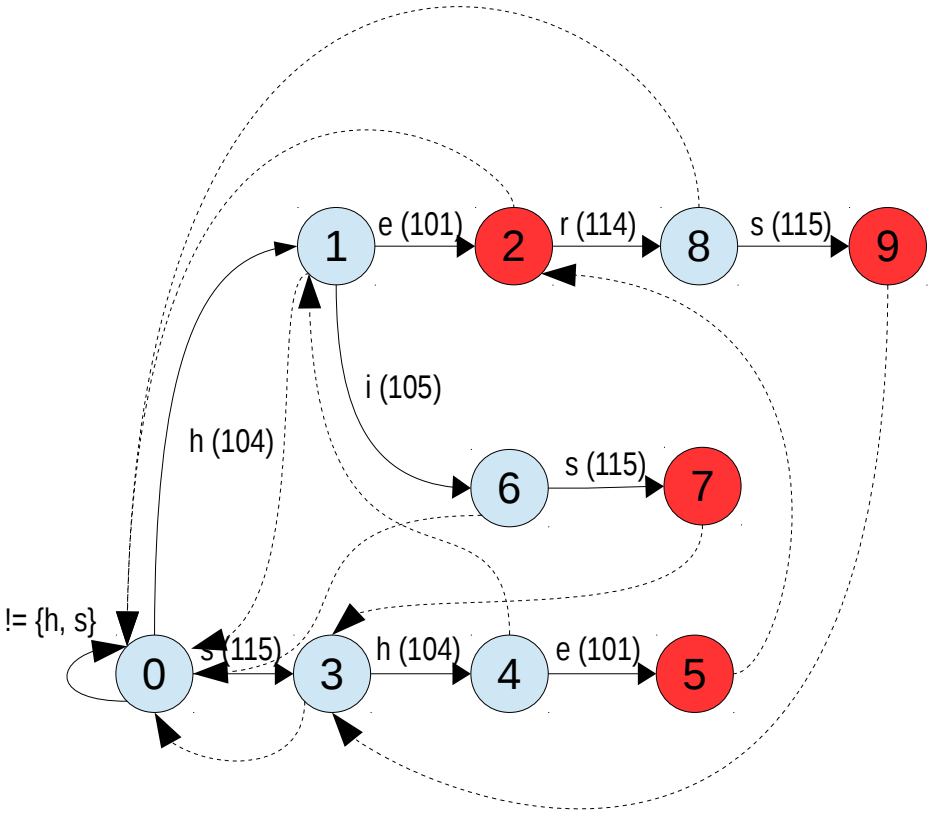
Pattern matching is the act of searching a given sequence of tokens for the presence of some other sequence of tokens, and represents one of the most common problems encountered in computer science, from simple string scanning and regular expressions through to lexers & parsers, virus scanners and Intrusion Detection Systems (IDS).

This paper describes an approach to pattern scanning that uses a parallel variant of the well known Aho-Corasick algorithm that has been implemented using OpenCL to enable heterogenous acceleration across a range of devices.

2. Aho-Corasick Algorithm Recap

The Aho-Corasick algorithm was first presented in 1975 by Alfred V. Aho and Margaret J. Corasick http://cr.yp.to/bib/1975/aho.pdf and is one of the most widely known and commonly adopted pattern scanning algorithms due to its property of matching all patterns in the target dictionary simultaneously.

The Aho-Corasick algorithm is an extension of a trie https://en.wikipedia.org/wiki/Trie (a prefix tree Deterministic Finite Automaton) with the addition of failure transitions to allow fast transitions between failed matches to other branches of the trie that share a common prefix. To illustrate, consider the following example that matches the patterns **he**, **she**, **his** and **hers**, the dotted lines in the diagram represent the failure transitions.

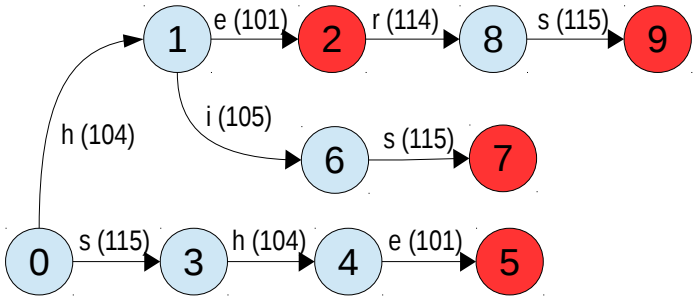


3. The Parallel Failureless Aho-Corasick Algorithm

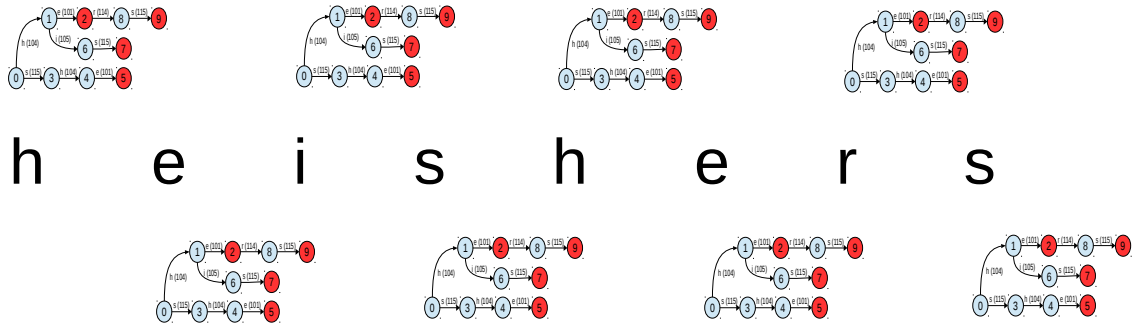
The conventional Aho-Corasick algorithm can be parallelised effectively via an approach known as the Data Parallel Aho-Corasick (DPAC) algorithm, which partitions the sequence to be scanned into multiple (overlapping) segments and allocates each segment to a processing thread http://csweb.cs.wfu.edu/~fulp/Papers/milcom07f.pdf.

As the number of processing threads reaches one per character of the input sequence we can however take a radically different approach and dispense with the failure transitions altogether. This approach is known as the Parallel Failureless Aho-Corasick (PFAC) algorithm where each thread is only required to match the pattern that starts at its own thread starting position http://www.ijoer.com/Paper-June-2016/IJOER-JUN-2016-11.pdf.

By allocating a thread to each character of the input sequence the state machine is able to terminate as soon as it fails to find a match *without requiring failure transitions*, moreover as the state machine is an acyclic directed graph each final state represents a unique pattern except for the case of wildcard transitions where we may match both the explicit match and the wildcarded path separately, but for simplicity we ignore wlldcards for now.



The following example illustrates the approach for the simple input string **heishers**.



The **first** thread starts with the “h” transition then the “e” finding a match for “he”, but the subsequent “i” character has no transition so will cause the first thread to terminate. The **second** thread has no initial transition for “e” and so will terminate immediately, similarly the **third** thread has no initial transition for “i” so it too will terminate immediately. The **fourth** thread starts with the “s” transition then the “h” and “e” finding a match for “she”. The **fifth** thread starts with the “h” transition then the “e” finding a match for “he” and also subsequently the “r” then “s” finding a match for “hers”. The **sixth** and **seventh** threads match no initial transitions so terminate immediately and although the **eighth** thread matches the “s” transition it will then terminate as it reaches the end of the input sequence.

It should be clear from the example that although the algorithm *potentially* creates large numbers of threads most of them have a high probability of early termination. It is actually possible to implement the PFAC algorithm sequentially with a smaller number of threads such that a given thread simply iterates through the input character sequence calling the state machine for each character, but the approach is especially well suited to highly data parallel architectures such as GPUs where many thousands of threads may be employed.

Another important consideration is that pattern matching is intrinsically a memory bound problem and the PFAC algorithm has a number of advantages with respect to memory access. In particular because threads are assigned to subsequent locations of the input sequence there are opportunities to make good use of GPU local/shared memory as a cache, moreover this access pattern is optimal from the perspective of coalescing reads from global memory, which significantly improves GPU memory bandwidth.

4. Optimising the State Transition Table

Although perhaps not obvious from the previous graphical illustrations of the Aho-Corasick state-machine most implementations make use of a state transition look-up table to allow the next state to be determined given the current state and input character.

A common naïve implementation of the state transition table is to simply use a two dimensional array with the rows of the table indexed by current state and columns, representing the next state, indexed by the input character. This lookup is very time efficient requiring only one memory access, however in general it is extremely space *inefficient* as most states have relatively few outbound transitions, especially towards the leaves of the table, resulting in a very sparse array.

Another common implementation idiom for Aho-Corasick state transition tables is the use of a second (output) table indexed by state that is used to identify whether the state is a match state and look up the index of the matching pattern. In the previous examples states 2, 5, 7 and 9 represent match states and the output table would therefore have values at indexes 2, 5, 7 and 9 pointing to the patterns “he”, “she”, “his” and “hers” respectively.

From the perspective of improving the efficiency of the state transition table two key complementary approaches may be employed 1) employ hashing to exploit the sparseness of the main state transition table 2) eliminate the output table.

Eliminating the Output Table

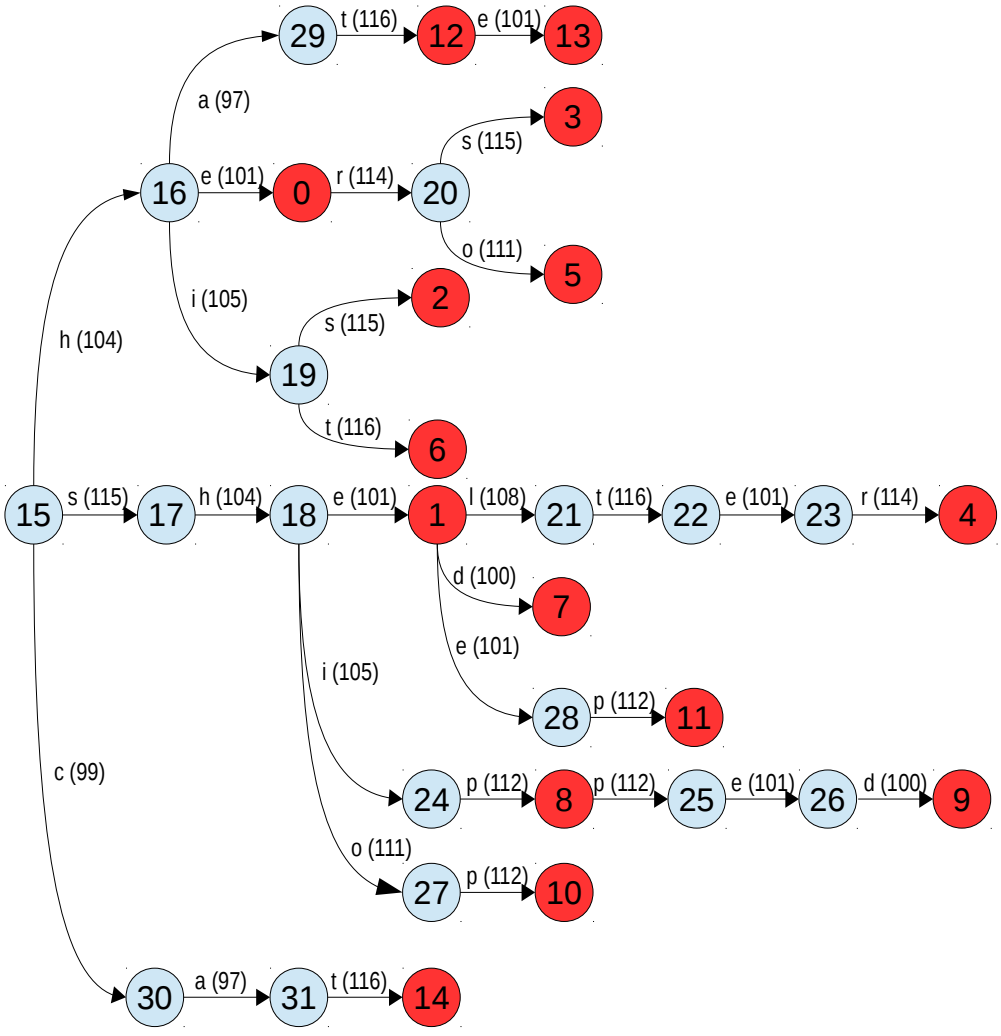
Eliminating the need for the secondary output table is surprisingly straightforward. Consider the earlier examples where the states of the state machine have been ordered in the “obvious” way by allocating state zero as the root of the state machine and the match states have arbitrary numbering, in the case of the examples 2, 5, 7 and 9.

If, however, we first consider the number of match states, in this case four, and make the initial state 4 instead of 0 we can then assign the match states numbers 0, 1, 2 and 3, each of which corresponds to the appropriate pattern number. We can now simply and cheaply identify a match state, as its state number is less than the initial state number, in this case a state number less than 4 corresponds to a match state.

Hashing the State Transition Table

The hashing approach is quite involved and best illustrated by using a slightly more complex example. The first step is to create a “raw” state transition table, consider for example the following patterns transformed into a state machine, with the match states allocated state values less than the initial state value as described previously.

```
 0: he
 1: she
 2: his
 3: hers
 4: shelter
 5: hero
 6: hit
 7: shed
 8: ship
 9: shipped
10: shop
11: sheep
12: hat
13: hate
14: cat
```



The raw state transition table is implemented for simplicity and compactness as an array of arrays (or vector of vectors) of Transitions where Transition is a simple struct.

```
struct Transition {
    std::int32_t ch;
    std::int32_t nextState;
};
```

The first step is to parse the dictionary to determine the number of patterns and for each pattern an empty column vector is inserted into the raw state transition table's row vector, after this initial parse we know that the number of the initial state (15 in this case) is the number of patterns in the dictionary and we add an empty column vector to row 15.

The second step is to parse the dictionary again, in this case each character represents a Transition. We first see “h” and as the current state is 15 we know the next state is 16, so we add an entry `{ch: 104, nextState: 16}` to the vector at row 15 and change the current state to 16. We next see “e”, but we also know that this is a match state so we know that the next state is the index of the pattern “he” so we add an entry `{ch: 101, nextState: 0}` to the vector at row 16 and reset the current state to the initial state. We next see “s” and add an entry `{ch: 115, nextState: 17}` to the vector at row 15 then we see “h” and add an entry `{ch: 104, nextState: 18}` to the vector at row 17 then we see “e” (another match state) and add an entry `{ch: 101, nextState: 1}` to the vector at row 18. This algorithm is followed until we reach the end of the dictionary and end up with a raw state transition table as follows, the initial state is highlighted in red below:

Raw State Transition Table

```
0   [{ch: 114, nextState: 20}],
1   [{ch: 108, nextState: 21}, {ch: 100, nextState:  7}, {ch: 101, nextState: 28}],
2   [],
3   [],
4   [],
5   [],
6   [],
7   [],
8   [{ch: 112, nextState: 25}],
9   [],
10  [],
11  [],
12  [{ch: 101, nextState: 13}],
13  [],
14  [],
15  [{ch: 104, nextState: 16}, {ch: 115, nextState: 17}, {ch:  99, nextState: 30}],
16  [{ch: 101, nextState:  0}, {ch: 105, nextState: 19}, {ch:  97, nextState: 29}],
17  [{ch: 104, nextState: 18}],
18  [{ch: 101, nextState:  1}, {ch: 105, nextState: 24}, {ch: 111, nextState: 27}],
19  [{ch: 115, nextState:  2}, {ch: 116, nextState:  6}],
20  [{ch: 115, nextState:  3}, {ch: 111, nextState:  5}],
21  [{ch: 116, nextState: 22}],
22  [{ch: 101, nextState: 23}],
23  [{ch: 114, nextState:  4}],
24  [{ch: 112, nextState:  8}],
25  [{ch: 101, nextState: 26}],
26  [{ch: 100, nextState:  9}],
27  [{ch: 112, nextState: 10}],
28  [{ch: 112, nextState: 11}],
29  [{ch: 116, nextState: 12}],
30  [{ch:  97, nextState: 31}],
31  [{ch: 116, nextState: 14}]
```

Given the raw state transition table we can now proceed to compile it into a hash table in order to provide the optimum balance between compactness and look-up performance, noting that although the raw state transition table is space optimal, the look-ups have been performed via a simple linear search, which is significantly sub-optimal especially in a memory bound application.

The general hashing approach is derived from the following paper: http://www.cs.umd.edu/~gasarch/BLOGPAPERS/FKS.pdf, this is usually known in the literature as FKS Perfect Hashing after the authors Fredman, Komlós and Szemerédi.

The idea of FKS hashing is to arrange the hash table in two levels:

- In level 1, hash n keys into m buckets by a universal hash function h.
  Bi is the set of items hashed to the ith bucket.
- In level 2, construct a perfect hash of size Bi^2 for each bucket Bi.

The Universal Hash function used is: h(k) = ((k*id mod p) mod s

FKS Hashing is, however, the *general* case, but has several key problems:

1. Two hash evaluations are required:
   row = ((k*id mod p) mod s where s is the number of states
   column = ((k*id mod p) mod Si where Si the is number of buckets in row i
2. The modulo operation is expensive on many architectures.
3. The approach requires extra data required by the hash function to be stored in both the row and column arrays.

Because Aho-Corasick state machines have a tree structure and the number of valid transitions generally decreases rapidly as the leaves are approached then it is possible to do perfect hashing on *each row* of the state transition table. That observation allows us to simplify the FKS approach into a single hash. In this case the row table size is simply the number of state transitions and the table contains an offset into the value table plus the k and Si values for that particular row. Because we are hashing every row of the state transition table we know that id is the transition character/octet, which is < 256 thus the prime number p can be 257 so that 1 <= k < 257.

From the FKS paper we note the *sufficient* condition for collision free hashing is Si >= Bi^2 and to reduce use of mod we choose the number of buckets Si to be a power of two, so for **((ki * ch) mod p) mod Si** we can instead do **((ki * ch) mod p) & (Si – 1)**, we shall later also eliminate mod p.

Note that we can also further optimise the hash table for space (at the cost of some increased computation during hash table compilation) as follows:

The maximum required value of Si is 256, but we can observe however that for:
[Bi = 1, Si = 1], [Bi = 2, Si = 4], [Bi = 3, 4, Si = 16], [Bi = 5, Si = 32],
[Bi = 6, 7, 8 Si = 64], [Bi = 9, 10, 11 Si = 128], [Bi = 12, 13, ..., 255 Si = 256]
and note therefore that the *sufficient* condition for collision free hashing adds a space penalty (though we know we are guaranteed a collision free hash). In the algorithm below we actually *relax* the sufficient condition, so for say Bi of 3 we first *try* computing ki values for an Si of 4 rather than 16, now it is entirely possible that *may* collide, however in very many cases it ***may not*** and would thus result in a *greatly compressed* transition table.

Intuitively one might expect that probing on a non-sufficient condition of Si would have a significant insert time penalty, because if we fail to find a ki value for the non-sufficient Si value we have to iterate again using the next power of two putative Si value until eventually we reach Si >= Bi^2, but in practice some initial timings suggest that the performance penalty to iterate though the non-sufficient Si values probing for a valid ki is modest and the table compression gain is significant, which may also help look up speed.

```
for (auto i = 0; i < numOfStates; i++) {
    hashRow.push_back({INVALID, INVALID}); // Initialise offset and k_sminus1.
    const std::vector<Transition>& currentState = stateTable[i];
    int Bi = currentState.size();
    if (Bi) {
        if (i == initialState) { // Copy initial transitions into separate look up table.
            for (auto transition : currentState) {
                initialTransitions[transition.ch] = transition.nextState;
            }
        } else { // Populate hashRow and hashVal.
            // Compute Si, the power of two greater than Bi.
            std::int32_t Si = 256;
            for (const auto Bi2 = Bi * 2; Si >= Bi2; Si >>= 1) {/*empty*/}
            std::int32_t sminus1 = Si - 1;

            // Iteratively compute ki which is the lowest ki < P257 where no collisions
            // occur for ((ki * ch) % p) % Si for all ch.
            std::int32_t ki = INVALID;
            while (ki < 0) {
                if (Si == 1 || Si == 256) {
                    ki = 1;
                } else {
                    for (std::int32_t k = 1; k < P257 && ki < 0; k++) {
                        ki = k;
                        std::vector<bool> bits(Si, false);

                        // Check if this value of k has any collisions.
                        for (auto transition : currentState) {
                            const auto p = mod257(k * transition.ch) & sminus1;
                            if (bits[p]) {
                                ki = INVALID; // Collision occurred, try next k.
                                break;
                            } else {
                                bits[p] = true;
                            }
                        }
                    }

                    if (ki == INVALID) { // No collision-free ki found, try next Si.
                        Si <<= 1;
                        sminus1 = Si - 1;
                    }
                }
            }

            for (auto j = 0; j < Si; j++) { // Reserve next block in hashVal table.
                hashVal.push_back({INVALID, INVALID});
            }

            for (auto transition : currentState) {
                const auto p = mod257(ki * transition.ch) & sminus1;
                hashVal[offset + p] = transition;
            }

            hashRow[i] = {offset, sminus1 | (ki << MASKBITS)};
            offset += Si;
        }
    }
}
```

The algorithm perhaps looks a little complicated, but is actually relatively straightforward, processing the raw state transition table “stateTable” and populating three related tables “initialTransitions”, “hashRow” and “hashVal” that comprise the final hash table.

initialTransitions is an array of integers that represents the initial transitions of the raw state transition table (in the case of the example state 15) indexed by input character. We use a simple array rather than hashing for the initial transitions as, in general, the first state has by far the most outbound transitions so hashing gains little space for non-trivial examples, but moreover as the PFAC algorithm generally terminates many threads on the first transition using a simple array can significantly optimise memory accesses for these cases. The test

```
if (i == initialState) {
    for (auto transition : currentState) {
        initialTransitions[transition.ch] = transition.nextState;
    }
}
```

causes initialTransitions to be populated when we reach row “initialState” in the raw state transition table and for our simple example we see initialTransitions set as follows:

initialTransitions Table

```
[ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, 30, -1, -1, -1, -1, 16, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, 17, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
  -1, -1, -1, -1, -1, -1, -1, -1, -1 ]
```

hashVal is a vector of Transitions that uses the same Transition struct as the raw state transition table and hashRow is a vector of structs that contain an offset into the hashVal table plus packed ki and s – 1 values used to compute the hash function at each row.

```
struct HashRow {
    std::int32_t offset;
    std::int32_t k_sminus1; // Packed using k << 16 | sminus1
};
```

In order to populate the hashRow and hashVal tables the first thing we do is to compute the *putative* Si and Si – 1 values, where Si the is number of buckets in row i and is the first power of two greater than the number of transitions in row i. This is the putative value of Si and we may have to increase it to a greater power of two if we subsequently find collisions.

```
std::int32_t Si = 256;
for (const auto Bi2 = Bi * 2; Si >= Bi2; Si >>= 1) {/*empty*/}
std::int32_t sminus1 = Si - 1;
```

The next code block is responsible for computing the ki value for the equation

p = ((ki * ch) mod 257) & (Si – 1)

where p represents the bucket index, Si is the number of buckets in row i and ch is the input character causing the transition. The algorithm iteratively computes ki where ki is the lowest value of ki less than the hash prime p, which is 257 in our case, for all values of ch such that no collisions occur for p.

In order to detect collisions we make use of a bit vector of size Si indexed by the computed p value thus if any value of ((ki * ch) mod 257) & (Si – 1) is repeated we know we have a hash collision and must try a different value of ki (or Si).

```
for (std::int32_t k = 1; k < P257 && ki < 0; k++) {
    ki = k;
    std::vector<bool> bits(Si, false);

    // Check if this value of k has any collisions.
    for (auto transition : currentState) {
        const auto p = mod257(k * transition.ch) & sminus1;
        if (bits[p]) {
            ki = INVALID; // Collision occurred, try next k.
            break;
        } else {
            bits[p] = true;
        }
    }
}
```

As a concrete example consider state 1 from our example (the first state with Si > 1)

```
1   [{ch: 108, nextState: 21}, {ch: 100, nextState:  7}, {ch: 101, nextState: 28}],
```

For k = 1 we compute p values of ((1*108) mod 257) & 3 = 0, ((1*100) mod 257) & 3 = 0 and note an immediate collision at p = 0
For k = 2 we compute p values of ((2*108) mod 257) & 3 = 0, ((2*100) mod 257) & 3 = 0 and note an immediate collision at p = 0
For k = 3 we compute p values of ((3*108) mod 257) & 3 = 3, ((3*100) mod 257) & 3 = 3 and note an immediate collision at p = 3
For k = 4 we compute p values of ((4*108) mod 257) & 3 = 3, ((4*100) mod 257) & 3 = 3 and note an immediate collision at p = 3
For k = 5 we compute p values of ((5*108) mod 257) & 3 = 2, ((5*100) mod 257) & 3 = 3, ((5*101) mod 257) & 3 = 0

So for ki of 5 and Si of 4 we have **no collisions for any value of ch**. Note that the number of transitions from state 1 is three so the *sufficient* value of Si would in theory be nine giving a next power of two value of Si of sixteen, but we have achieved collision free hashing with an Si of **four** buckets rather than sixteen.

For our simple example all states achieved collision free hashing for values of Si less than the theoretical sufficient value, giving a hashVal table size of 31 rather than 71.

hashRow Table

```
 0: [ { offset:  0, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
 1:   { offset:  1, k_sminus1: 327683 },  // k = 5,  sminus1 = 3
 2:   { offset: -1, k_sminus1: -1 },
 3:   { offset: -1, k_sminus1: -1 },
 4:   { offset: -1, k_sminus1: -1 },
 5:   { offset: -1, k_sminus1: -1 },
 6:   { offset: -1, k_sminus1: -1 },
 7:   { offset: -1, k_sminus1: -1 },
 8:   { offset:  5, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
 9:   { offset: -1, k_sminus1: -1 },
10:   { offset: -1, k_sminus1: -1 },
11:   { offset: -1, k_sminus1: -1 },
12:   { offset:  6, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
13:   { offset: -1, k_sminus1: -1 },
14:   { offset: -1, k_sminus1: -1 },
15:   { offset: -1, k_sminus1: -1 },
16:   { offset:  7, k_sminus1: 2424835 }, // k = 25, sminus1 = 3
17:   { offset: 11, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
18:   { offset: 12, k_sminus1: 327683 },  // k = 5,  sminus1 = 3
19:   { offset: 16, k_sminus1: 65537 },   // k = 1,  sminus1 = 1
20:   { offset: 18, k_sminus1: 589825 },  // k = 9,  sminus1 = 1
21:   { offset: 20, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
22:   { offset: 21, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
23:   { offset: 22, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
24:   { offset: 23, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
25:   { offset: 24, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
26:   { offset: 25, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
27:   { offset: 26, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
28:   { offset: 27, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
29:   { offset: 28, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
30:   { offset: 29, k_sminus1: 65536 },   // k = 1,  sminus1 = 0
31:   { offset: 30, k_sminus1: 65536 } ]  // k = 1,  sminus1 = 0
```

hashVal Table

```
 0: [ { ch: 114, nextState: 20 },  // offset 0,  bucket 0
 1:   { ch: 101, nextState: 28 },  // offset 1,  bucket 0
 2:   { ch:  -1, nextState: -1 },  // offset 1,  bucket 1
 3:   { ch: 108, nextState: 21 },  // offset 1,  bucket 2
 4:   { ch: 100, nextState:  7 },  // offset 1,  bucket 3
 5:   { ch: 112, nextState: 25 },  // offset 5,  bucket 0
 6:   { ch: 101, nextState: 13 },  // offset 6,  bucket 0
 7:   { ch:  97, nextState: 29 },  // offset 7,  bucket 0
 8:   { ch:  -1, nextState: -1 },  // offset 7,  bucket 1
 9:   { ch: 105, nextState: 19 },  // offset 7,  bucket 2
10:   { ch: 101, nextState:  0 },  // offset 7,  bucket 3
11:   { ch: 104, nextState: 18 },  // offset 11, bucket 0
12:   { ch: 101, nextState:  1 },  // offset 12, bucket 0
13:   { ch: 111, nextState: 27 },  // offset 12, bucket 1
14:   { ch:  -1, nextState: -1 },  // offset 12, bucket 2
15:   { ch: 105, nextState: 24 },  // offset 12, bucket 3
16:   { ch: 116, nextState:  6 },  // offset 16, bucket 0
17:   { ch: 115, nextState:  2 },  // offset 16, bucket 1
18:   { ch: 111, nextState:  5 },  // offset 18, bucket 0
19:   { ch: 115, nextState:  3 },  // offset 18, bucket 1
20:   { ch: 116, nextState: 22 },  // offset 20, bucket 0
21:   { ch: 101, nextState: 23 },  // offset 21, bucket 0
22:   { ch: 114, nextState:  4 },  // offset 22, bucket 0
23:   { ch: 112, nextState:  8 },  // offset 23, bucket 0
23:   { ch: 101, nextState: 26 },  // offset 24, bucket 0
25:   { ch: 100, nextState:  9 },  // offset 25, bucket 0
26:   { ch: 112, nextState: 10 },  // offset 26, bucket 0
27:   { ch: 112, nextState: 11 },  // offset 27, bucket 0
28:   { ch: 116, nextState: 12 },  // offset 28, bucket 0
29:   { ch:  97, nextState: 31 },  // offset 29, bucket 0
30:   { ch: 116, nextState: 14 } ] // offset 30, bucket 0
```

5. Key OpenCL Concepts

Note that this document shall primarily focus on optimisations for OpenCL high-end GPU devices, as those are currently the most common devices used for acceleration with OpenCL. it is likely that multicore CPU, Many Integrated Core (MIC) CPU, such as Xeon Phi, FPGA and even low-end/mobile GPU devices may require different optimisations to achieve maximum performance, though many of the principles are likely to be similar.

The two main things to consider when optimising OpenCL for particular compute devices is how the OpenCL Execution Model maps to device threads and processors and, most importantly for pattern matching, how the OpenCL Memory Model maps to device memory.

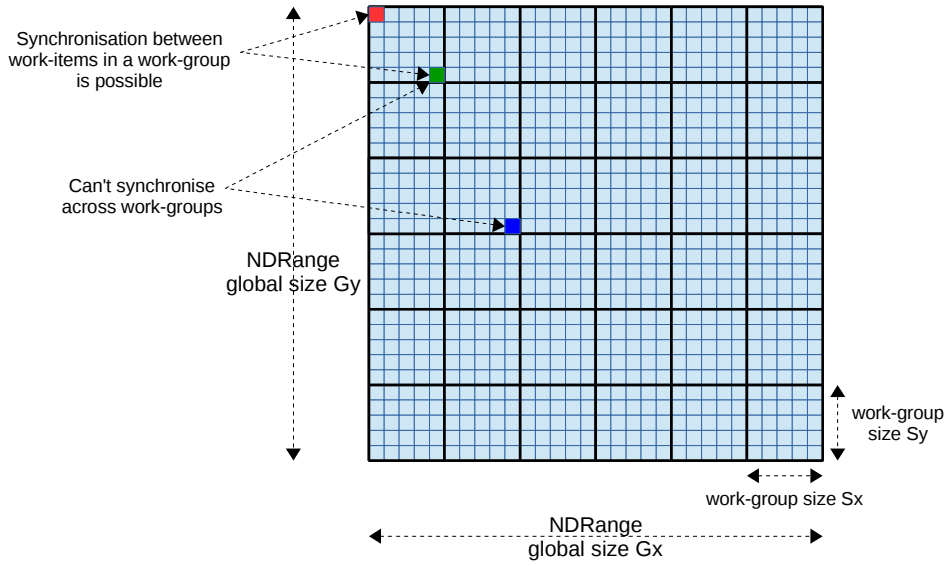
The OpenCL specification https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf provides the definitive reference for the Execution and Memory models, but it is helpful to summarise some of the key concepts here and illustrate how they map to the application. Note that OpenCL covers both host and device side programming, but from the perspective of optimisations we shall primarily consider the device side at this stage.

Execution Model

The key concept of the OpenCL Execution Model on the device is the kernel. When a kernel is submitted for execution by a host application an N-dimensional index space called an NDRange is defined. Each kernel instance represents a **thread** of execution that executes for a point in the index space and in OpenCL terms is known as a **work-item**.

Threads/work-items are themselves arranged into a more coarsely grained index space known as a **thread block** or **work-group**. Each work-item within a work-group executes concurrently within a compute unit (for GPUs a compute unit corresponds to Nvidia's Streaming Multiprocessor or AMD's SIMD Engine). On GPU devices work-groups are further decomposed into device specific blocks that represent the lowest level execution flow control can affect, known as warps by Nvidia and wavefronts by AMD. Nvidia warps comprise 32 work-items and AMD wavefronts comprise 64 work-items. **Best performance is achieved when work-group size is an integer multiple of the warp/wavefront size**.

The following diagram illustrates work-items and work-groups in a 2D Range.



The OpenCL C API provides a number of built-in functions to enable the kernel to query information about the global and work-group local work size as follows:

```
uint get_work_dim()                      // Number of dimensions in use
size_t get_global_size(uint dimension)   // Number of global work-items
size_t get_global_id(uint dimension)     // Global work-item ID
size_t get_local_size(uint dimension)    // Number of work-items in a work-group
size_t get_local_id(uint dimension)      // The work-item ID in a work-group
size_t get_num_groups(uint dimension)    // The number of work-groups
size_t get_group_id(uint dimension)      // The work-group ID
size_t get_global_offset(uint dimension) // The global offset
```

For the previous illustration we would see get_work_dim() = 2, get_global_size(0) = 30 and get_global_size(1) = 30 (corresponding to Gx and Gy respectively). For the red work-item get_global_id(0) = 0, get_global_id(1) = 0. For the green work-item get_global_id(0) = 4, get_global_id(1) = 4. For the blue work-item get_global_id(0) = 9, get_global_id(1) = 14. get_local_size(0) = 5 and get_local_size(1) = 5 (corresponding to Sx and Sy respectively). get_num_groups(0) = 6 and get_num_groups(1) = 6.

Each work-item is identifiable in two ways; the first in terms of its global ID as illustrated previously, the second is in terms of its work-group ID plus its local work-item ID within a work-group. For the red work-item get_local_id(0) = 0, get_local_id(1) = 0, get_group_id(0) = 0, get_group_id(1) = 0. For the green work-item get_local_id(0) = 4, get_local_id(1) = 4, get_group_id(0) = 0, get_group_id(1) = 0. For the blue work-item get_local_id(0) = 4, get_local_id(1) = 4, get_group_id(0) = 1, get_group_id(1) = 2.

We can compute the global ID in terms of the local ID, local size and work-group ID as:
get_global_id = get_group_id * get_local_size + get_local_id + get_global_offset
So for the blue work-item we see 9 = 1 * 5 + 4 + 0 and 14 = 2 * 5 + 4 + 0

Similarly given a global ID, local ID and the work-group size, the work-group ID for a work-item may be computed as:
get_group_id = (get_global_id – get_local_id – get_global_offset)/ get_local_size
So for the blue work-item we see 1 = (9 – 4 – 0)/5 and 2 = (14 – 4 - 0)/5

Similarly we can compute the global size in terms of local size and number of work-groups:
get_global_size = get_local_size * get_num_groups e.g. 30 = 5 * 6 or
get_num_groups = get_global_size/get_local_size e.g. 6 = 30/5

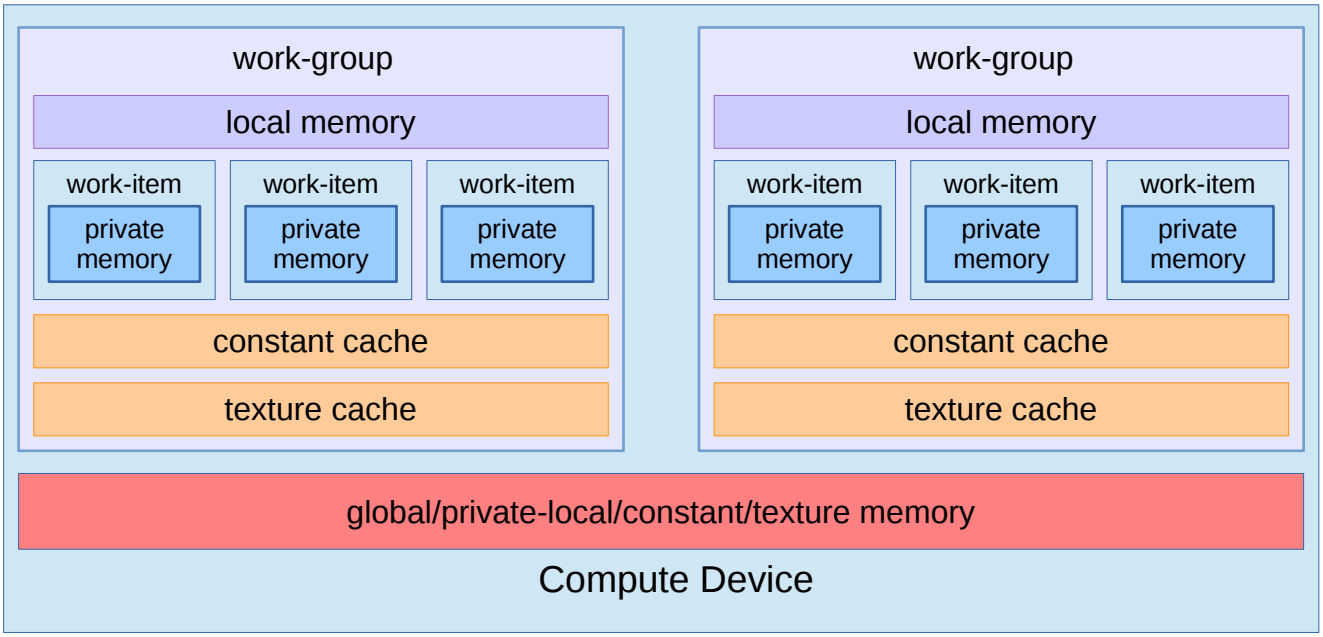
It is important to be able to understand how we can represent global ID in terms of the local ID, work-group size and work-group ID and vice-versa, because as we explore further we shall see that OpenCL device optimisations tend to occur within the context of a work-group (AKA thread block) where we are able to share certain resources, in particular high speed local memory between work-items in a work-group, but not between work-groups. These shared resources are indexed in terms of local ID as they are work-group specific, so be must therefore be ready to map from global IDs to local IDs and back.

Memory Model

The OpenCL memory model corresponds fairly closely to the physical memory configuration of Nvidia and AMD GPU hardware. For the case of Nvidia each Streaming Multiprocessor has on-chip memory of the following types (AMD SIMD units are similar):

- One set of local 32-bit *registers* per processor.
- A parallel data cache or shared memory that is shared by all scalar processor cores and is where OpenCL local memory resides.
- A read only *constant cache* that is shared by all scalar processor cores and speeds up reads from OpenCL constant memory.
- A read only *texture cache* that is shared by all scalar processor cores and speeds up reads from OpenCL image objects; each multiprocessor accesses the texture cache via a *texture unit* that implements the various addressing modes and data filtering specified by OpenCL sampler objects; the region of device memory addressed by image objects is referred to as *texture memory*.

In addition there is a global memory address space that is used for OpenCL global memory and a local memory address space private to each thread (not to be confused with OpenCL local memory). OpenCL global and private-local memory are read-write regions of device memory **and are not cached on GPUs**.



Private memory is not visible to the host nor to any other work-item, it is only visible to an individual work-item.

For local memory, the values that are seen by a set of work-items within a work-group are guaranteed to be consistent at work-group synchronisation points. For example a work-group barrier requires that all loads and stores defined before the barrier complete before any work-items in the group proceed past the barrier.

Global memory is also made consistent at a work-group barrier, however even though this memory is shared between work-groups there is no way to enforce consistency of global memory between different work-groups, only between work-items in a work-group.

6. OpenCL GPU Device Optimisations

The previous section described a number of key OpenCL concepts that we can now go on to employ in order to optimise the application for high-end GPU devices.

Use OpenCL Image Objects for the Hash Table

OpenCL defines two types of memory objects, buffer objects and image objects. A buffer object is simply a contiguous block of memory that is made available to kernels as global memory. Buffer objects are very flexible as application developers can map arbitrary data structures onto buffers and access these via pointers on the device.

Image objects, as the name suggests, are primarily intended for holding images and the image object is opaque with accessors provided to retrieve the RGBA pixel values.

Image objects have one key advantage over buffer objects for implementing a lookup table however, which is that on GPU devices access to image/texture memory is cached through the texture system (corresponding to the GPU L2 and L1 caches).

Fortunately encoding the hash table structures as image objects is not too convoluted as each of the tables comprises a simple struct of two integers for hashRow and hashVal and a single integer for initialTransitions. These map readily to the OpenCL Image Format (CL_RG, CL_SIGNED_INT32) for the case of a hashRow and hashVal and (CL_R, CL_SIGNED_INT32) for the case of initialTransitions. Moreover with OpenCL 1.2 we don't even have to play any tricks to index the data in the tables, as **Image1DBuffer** objects represent one dimensional images backed by linear memory and can be accessed by simple un-normalised indexes that map directly to array indexes.

```
cl::Buffer inTransBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                         sizeof(std::int32_t)*initialTransitionsH.size(),
                         const_cast<std::int32_t*>(initialTransitionsH.data()));

initialTransitions = cl::Image1DBuffer(context, CL_MEM_READ_ONLY,
                                       cl::ImageFormat(CL_R, CL_SIGNED_INT32),
                                       initialTransitionsH.size(),
                                       inTransBuffer);

cl::Buffer hashRowBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                         sizeof(HashRow)*hashRowH.size(),
                         const_cast<HashRow*>(hashRowH.data()));

hashRow = cl::Image1DBuffer(context, CL_MEM_READ_ONLY,
                            cl::ImageFormat(CL_RG, CL_SIGNED_INT32),
                            hashRowH.size(),
                            hashRowBuffer);

cl::Buffer hashValBuffer(context, CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                         sizeof(Transition)*hashValH.size(),
                         const_cast<Transition*>(hashValH.data()));

hashVal = cl::Image1DBuffer(context, CL_MEM_READ_ONLY,
                            cl::ImageFormat(CL_RG, CL_SIGNED_INT32),
                            hashValH.size(),
                            hashValBuffer);
```

On the device side we use **read_imagei** with appropriate swizzle to get the struct values.

```
const int2 row = read_imagei(hashRow, state).xy; // hashRow[state]
```

Eliminate the Second mod Operator

In section 4. Optimising the State Transition Table in order to reduce use of the modulo operator we chose the bucket size Si to be a power of two, so for **((ki * ch) mod p) mod Si** we can instead do **((ki * ch) mod p) & (Si – 1)**, however that still leaves the mod p term.

Because the maximum number of outbound transitions from a state is 256 in our case p may be 257, the first prime number above 256. 257 has the interesting property that we can do reduction modulo 257 by simple bitwise operations using (x & 255) - (x >> 8) http://mymathforum.com/number-theory/11914-calculate-10-7-mod-257-a.html. We can thus create an optimised mod257 function as follows:

```
static inline int mod257(int x) {
    int mod = (x & 255) - (x >> 8);
    if (mod < 0) {
        mod += 257;
    }
    return mod;
}
```

At first glance this function may have a small issue due to the “if (mod < 0)” test and branch, however the PTX assembly created by the Nvidia compiler appears to be optimised to avoid the branch. The test could be replaced by an OpenCL select function, but it seems unnecessary and makes the code somewhat less clear.

Combining the use of image1d_buffer_t objects as described in the previous section with the optimised mod257 gives us the following OpenCL C function for looking up the next state in the hash table given the current state and transition character, noting that the initial transition is treated separately and will be covered later.

```
static inline int lookup(image1d_buffer_t hashRow,
                         image1d_buffer_t hashVal,
                         int state,
                         int inputChar) {
    const int2 row = read_imagei(hashRow, state).xy;
    const int offset  = row.x;
    int nextState = INVALID;
    if (offset >= 0) {
        const int k_sminus1 = row.y;
        const int sminus1 = k_sminus1 & MASK;
        const int k = k_sminus1 >> MASKBITS;

        const int p = mod257(k * inputChar) & sminus1;
        const int2 value = read_imagei(hashVal, offset + p).xy;
        if (inputChar == value.x) {
            nextState = value.y;
        }
    }
    return nextState;
}
```

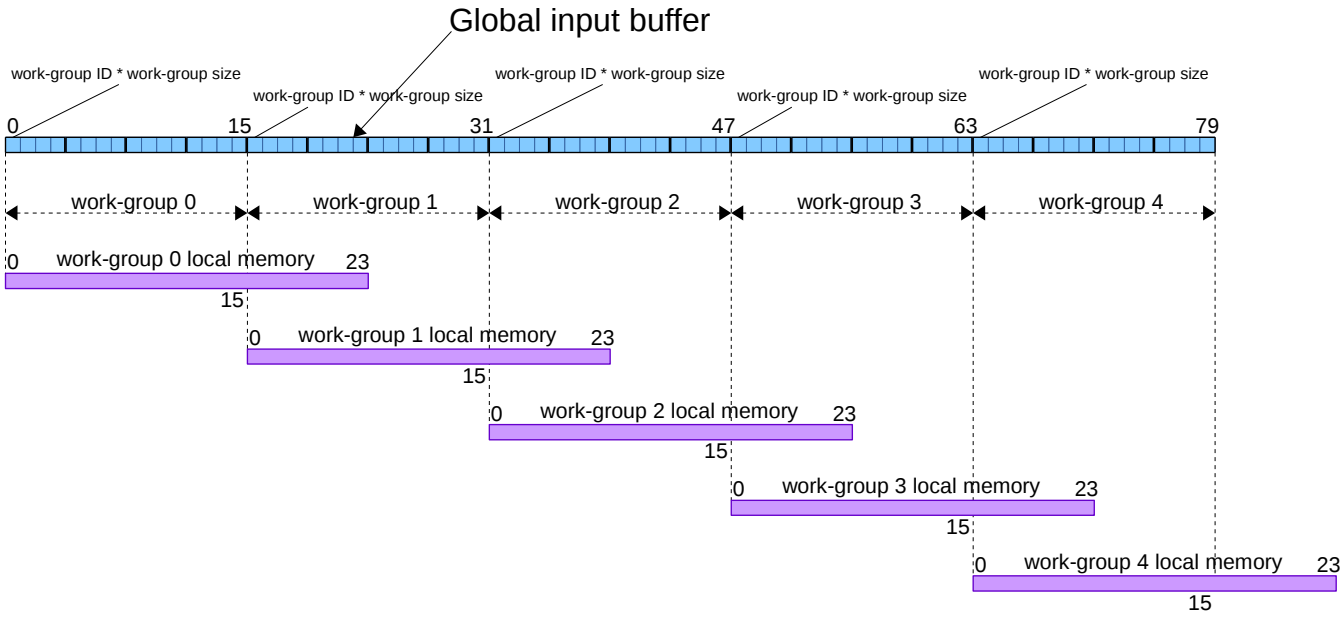
Copy Data to Local Memory to Reduce Global Memory Accesses

As the pattern matching is implemented by reading characters from the input buffer and using those to navigate a state transition table the application is really far more memory bound than compute bound and thus the overall system performance is primarily limited by memory access performance.

On high-end GPUs it is difficult to optimise the performance of global memory accesses as only certain memory access patterns enable the hardware to coalesce groups of data items to be written and read in one operation. The Nvidia best practices guide http://www.nvidia.com/content/cudazone/cudabrowser/downloads/papers/nvidia_opencl_bestpracticesguide.pdf states that perhaps the single most important performance consideration in programming for the CUDA architecture is coalescing global memory accesses. Global memory loads and stores by threads of a half warp (16 threads) are coalesced by the device in as few as one transaction (or two transactions in the case of 128-bit words) when certain access requirements are met. To understand these access requirements, global memory should be viewed in terms of aligned segments of 16 and 32 words. AMD architectures have similar requirements for coalesced accesses.

Trying to arrange memory access patterns in an application to be optimal for memory coalescence is difficult, especially if the application really requires random access. One common pattern to optimise global memory access is for the application to perform coalesced reads on each thread from global memory to local memory and then perform subsequent reads from local memory, which is thus essentially behaving as a cache shared between all threads in a work-group (AKA thread block).

By reading from global memory to local memory and performing subsequent memory accesses *in terms of local memory* it becomes important to think of the application in terms of its work-groups, to do so we must reconsider the OpenCL Execution Model. Consider the following example where we have five work-groups each with sixteen work-items and recall from the previous discussion of the Execution Model that:
get_global_id = get_group_id * get_local_size + get_local_id + get_global_offset



Copying the Input Data to Local Memory

Recall from Section 3. that the PFAC algorithm assigns a thread to each input character and each thread continues reading characters from that point either until a final state or invalid transition is reached or we reach the end of the input stream. This is a relatively obvious procedure when global memory is being accessed as all accesses are contiguous and the end of the input stream is the end of the global memory buffer, but it requires more thought when mapping to work-groups and local memory. Consider the code below where inputSize is the *actual* size of the input buffer (as opposed to get_global_size), cache is a local memory array size WORK_GROUP_SIZE + MAX_PATTERN_SIZE and input is the global memory buffer.

```
if (get_global_id(0) < inputSize) {
    cache[get_local_id(0)] = input[get_global_id(0)];
}
```

This is the normal way to copy data from global memory to local memory, however if we were to simply copy data in this way we would have a boundary problem as we would only have get_local_size characters available in local memory, so threads close to the boundary will terminate before reaching the end of the state transition table and valid matches may get missed by the scanner.

To solve this problem we increase the size of the local memory buffer as illustrated in the previous diagram, the extra size *should be at least as big as the maximum pattern size*. We can therefore modify the previous call to include an additional copy from global memory to the extra local memory space. Note that this read is also coalesced as it is a work-group size (a multiple of warp/wave front size) from the original copy, note too that not all threads will perform the second copy only those with a local ID less than the largest pattern size. The final code to copy from global memory to local memory looks like this:

```
const int tid = get_local_id(0); // Thread (Work Item) ID
int inputIndex = get_global_id(0);

if (inputIndex < inputSize) {
    cache[tid] = input[inputIndex];
}

inputIndex += WORK_GROUP_SIZE;
if ((inputIndex < inputSize) && (tid < MAX_PATTERN_SIZE)) {
    cache[tid + WORK_GROUP_SIZE] = input[inputIndex];
}
```

Copying the Initial Transition to Local Memory

As the PFAC algorithm generally terminates many threads on the first transition using a simple array can significantly optimise memory accesses, so we load the initialTransitions table to local memory as follows.

```
const int tid = get_local_id(0); // Thread (Work Item) ID
initialTransitionsCache[tid] = read_imagei(initialTransitions, tid).x;
```

Note that doing this requires that **the work-group size must be set to 256** in the call to enqueueNDRangeKernel (256 being the maximum number of transitions) so that the range of local ID (AKA thread ID, AKA work-item ID) is between 0 and 255. A work-group size of 256 happens to be a good value anyway as it is a multiple of warp/wave-front size.

Process Input Characters in Groups of Four (as an OpenCL int)

Although the input stream to be scanned is clearly a sequence of characters/bytes it is more efficient for the kernel to treat the data, initially at least, as a sequence of integers. This is because coalescing is achieved for any pattern of accesses that fits into a segment size of 32 bytes for 8-bit words, 64 bytes for 16-bit words, **or 128 bytes for 32- and 64-bit words**. Moreover as illustrated in the previous sections on copying data from global memory to local memory each thread copies the memory item associated with its own index irrespective of the size of that item (char, short or int), so again it is more efficient to copy integers from global memory to local memory than characters.

The approach taken therefore is to have a WORK_GROUP_SIZE = 256 and MAX_PATTERN_SIZE = 128, which will result in a maximum of 384 integers or 1536 bytes being processed by a given work-group and allow a maximum pattern of 512 characters.

In order to achieve this we need to ensure that the global size of the NDRange is set to the correct multiple of WORK_GROUP_SIZE so we must first calculate the number of OpenCL integers that would completely contain the input bytes and then round up if necessary to a multiple of WORK_GROUP_SIZE. The host size code to achieve this is as follows:

```
const cl_int size = input.size(); // input is a std::vector<char>

queue.enqueueWriteBuffer(inputBuffer, CL_TRUE, 0, size, input.data());

/**
 * The kernel processes the input characters in groups of four (OpenCL int),
 * so we therefore need to calculate our global work size in terms of how
 * many OpenCL integers it comprises. The computed global work size is then
 * rounded up to a multiple of the local work-group size (thread block size).
 */

// Number of OpenCL integers that would completely contain the input bytes.
const cl_int n = (size + sizeof(cl_int) - 1)/sizeof(cl_int);

// Given n round up if necessary to a multiple of WORK_GROUP_SIZE.
const auto r = n % WORK_GROUP_SIZE;
const auto global = (r == 0) ? n : n + WORK_GROUP_SIZE - r;

const cl_int initialState = dictionary->initialState;

pfacKernel.setArg(0, initialTransitions);
pfacKernel.setArg(1, hashRow);
pfacKernel.setArg(2, hashVal);
pfacKernel.setArg(3, initialState);
pfacKernel.setArg(4, inputBuffer);
pfacKernel.setArg(5, outputBuffer);
pfacKernel.setArg(6, size);
pfacKernel.setArg(7, n);

queue.enqueueNDRangeKernel(pfacKernel,
                           cl::NullRange, // Offset value is zero.
                           cl::NDRange(global),
                           cl::NDRange(WORK_GROUP_SIZE));

queue.enqueueReadBuffer(outputBuffer, CL_TRUE, 0,
                        size*sizeof(cl_int), output.data());
```

On the OpenCL kernel the complete code to load the data into local memory, compute the local buffer size available to a work-group and to synchronise the work-group threads to the local memory load is as follows, which represents roughly the first half of the kernel.

```
__kernel void pfac(image1d_buffer_t initialTransitions,
                   image1d_buffer_t hashRow,
                   image1d_buffer_t hashVal,
                   int initialState,
                   global int* input,
                   global int* output,
                   int inputSize, // Input size in bytes.
                   int n) {
    // Calculate the index of the first character in the Work Group.
    const int firstCharInWorkGroup = get_group_id(0)*WORK_GROUP_SIZE*sizeof(int);

    // Calculate remaining characters, starting from firstCharInWorkGroup.
    const int remaining = inputSize - firstCharInWorkGroup;

    // Calculate the local memory buffer size in bytes, noting that the last
    // work-group may contain fewer characters than the maximum buffer size.
    const int MAX_BUFFER_SIZE = (WORK_GROUP_SIZE + MAX_PATTERN_SIZE)*sizeof(int);
    const int bufferSize = min(remaining, MAX_BUFFER_SIZE);

    const int tid = get_local_id(0); // Thread (Work Item) ID

    int inputIndex  = get_global_id(0);
    int outputIndex = firstCharInWorkGroup + tid;

    // Local (AKA shared) memory arrays.
    local int initialTransitionsCache[WORK_GROUP_SIZE];
    local int cache[WORK_GROUP_SIZE + MAX_PATTERN_SIZE];
    local unsigned char* buffer = (local unsigned char*)cache;

    // Load the initialTransitions table to local (shared) memory.
    initialTransitionsCache[tid] = read_imagei(initialTransitions, tid).x;

    // Read input data from global memory to local (shared) memory.
    if (inputIndex < n) {
        cache[tid] = input[inputIndex];
    }

    // Read extra input data as we need to overlap to mitigate boundary condition.
    inputIndex += WORK_GROUP_SIZE;
    if ((inputIndex < n) && (tid < MAX_PATTERN_SIZE)) {
        cache[tid + WORK_GROUP_SIZE] = input[inputIndex];
    }

    // Block until all Work Items in the Work Group have reached this point
    // to ensure correct ordering of memory operations to local memory.
    barrier(CLK_LOCAL_MEM_FENCE);


    . . . more of the kernel code to follow
```

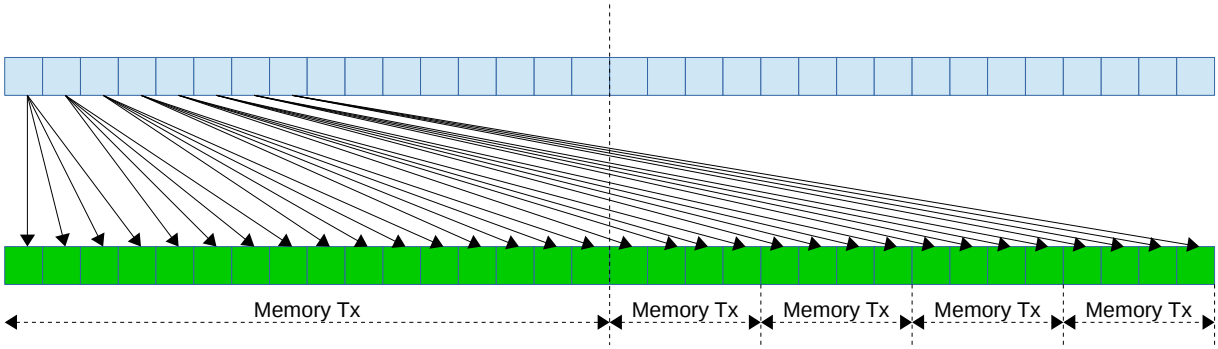
Much of this code should be familiar from the previous sections on copying data from global to local memory, but it is useful to see it in context.

Coalesce Global Memory Writes

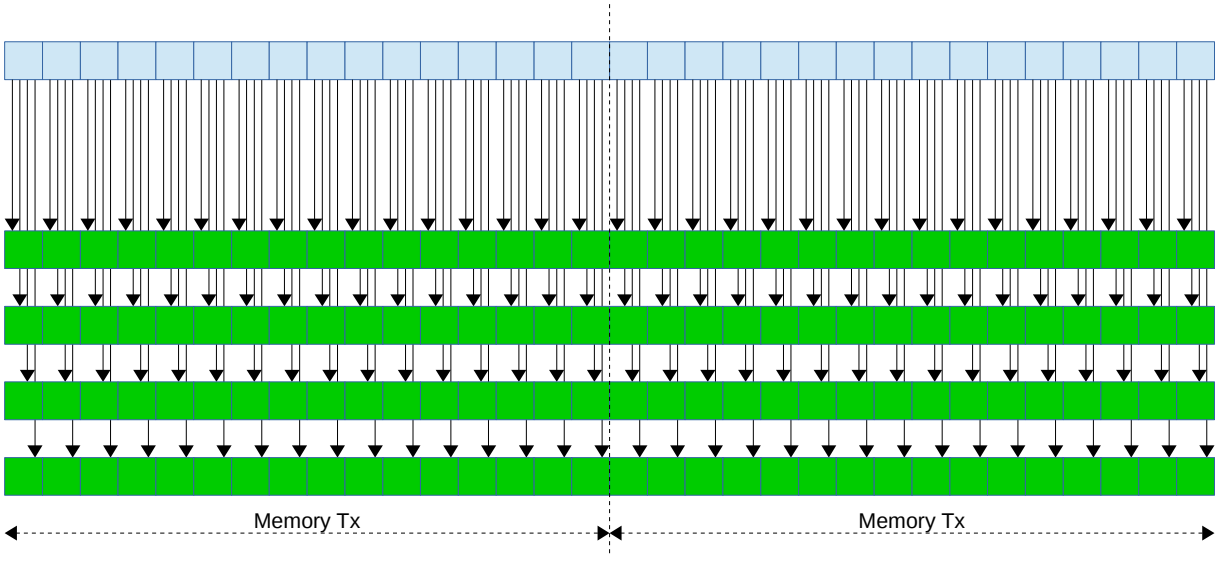
Although each thread processes four input characters as described in the previous section it is not optimal to simply process the input characters in the “obvious” way as sequential characters, this is because although our input data is a sequence of characters/bytes (albeit mapped to integers to optimise memory read coalescence) our output data *is* actually a sequence of integers. Our output is the index of the longest matching pattern recorded at the location in the stream where the pattern occurred, thus if we require a dictionary of size > 256 we need to write our output to a sequence of shorts or of ints.

As we actually need to output integers not characters we can't simply pack the output data for each group of four characters into a single integer as we did for the input data, we therefore have to think about how best to achieve coalesced memory writes.

The problem is that coalescing memory reads and writes is done on the basis of threads in a half warp (16 threads) and if each thread processes each character of its input group of four in simple sequence although the writes of the first four threads would be coalesced the remainder would not be. Each subsequent thread in the half warp would result in a further 32 byte memory transaction being issued, a total of 13 transactions per half warp.



Clearly that memory access pattern is sub-optimal and we would like all our memory writes to be coalesced. The way to achieve this is to ensure that each write aligns with the half warp, that is to say accesses are on some multiple of 16. As our work-group size is 256 threads we therefore arrange for each write to be 256 bytes apart.



Recall that we have a WORK_GROUP_SIZE = 256 and MAX_PATTERN_SIZE = 128, which will result in a maximum of 384 integers or 1536 bytes being processed by a given work-group. This data has been copied into local memory, so we are in a position to efficiently access it however we wish, so by arranging each thread to access the character associated with its own local ID (e.g. local work-item/thread ID within the work-group) and the characters at multiples of the work-group size we can therefore achieve coalesced writes and minimise memory transactions to four per half warp. The second half of the kernel therefore is as follows:

```
. . . from first part of the kernel code.

    // Perform state machine look-up with each thread processing four characters.
    #pragma unroll
    for (int i = 0; i < 4; i++) {
        const int j = tid + i * WORK_GROUP_SIZE;
        int pos = j;

        if (pos >= bufferSize) return;

        int match = -1;
        int inputChar = buffer[pos];
        int nextState = initialTransitionsCache[inputChar];
        if (nextState != INVALID) {
            if (nextState < initialState) {
                match = nextState;
            }
            pos = pos + 1;
            while (pos < bufferSize) {
                inputChar = buffer[pos];
                nextState = lookup(hashRow, hashVal, nextState, inputChar);
                if (nextState == INVALID) {
                    break;
                }

                if (nextState < initialState) {
                    match = nextState;
                }
                pos = pos + 1;
            }
        }

        output[outputIndex] = match;
        outputIndex += WORK_GROUP_SIZE;
    }
}
```

This is relatively straightforward; the loop is set to four as we are processing four characters per thread and we unroll the loop for efficiency. The next step is to find the index of the initial character of each scan.

```
const int j = tid + i * WORK_GROUP_SIZE;
```

This results in initial indexes into the input data cached in local memory of tid, tid + 256, tid + 512 and tid + 768. We then check if the index is greater than the remaining number of characters, which allows unused threads in the last Work Group to terminate early.

```
if (pos >= bufferSize) return;
```

At this point for each iteration of the loop we have the index of the first character to be processed by the PFAC state machine. The next step is to check if the first character corresponds to a state transition by looking up the cached array of initial transitions.

```
int inputChar = buffer[pos];
int nextState = initialTransitionsCache[inputChar];
```

If no transition is found (nextState is INVALID) then the output data is set to -1, the output index is increased by WORK_GROUP_SIZE and the next input character is processed. If however a valid transition is found we check it it is a match state and then go on to traverse the remainder of the state transition table until either an invalid state is reached (representing no transition for the current character in the state transition table) or we reach the end if the input buffer.

```
while (pos < bufferSize) {
    inputChar = buffer[pos];
    nextState = lookup(hashRow, hashVal, nextState, inputChar);
    if (nextState == INVALID) {
        break;
    }

    if (nextState < initialState) {
        match = nextState;
    }
    pos = pos + 1;
}
```

The lookup function performs a lookup of the state transition table hash table and its operation was covered in the previous section on eliminating the second mod operator.

Finally we are able to write the result of the match back out to global memory

```
// Output results to global memory
output[outputIndex] = match;
outputIndex += WORK_GROUP_SIZE;
```

We note again that each thread processes four characters, where each input character being processed is separated by WORK_GROUP_SIZE in order to achieve coalesced writes, so we similarly have to increment our outputIndex by WORK_GROUP_SIZE.

7. OpenCL Data Transfer Optimisations

Although high-end GPU devices have extremely high compute capability (~6.5 TFLOPS for the Maxwell powered Titan X and ~11 TFLOPS for the Pascal powered Titan X) as well as correspondingly high coalesced memory bandwidth (336 GB/s for Maxwell Titan X and 480 GB/s for Pascal Titan X) those headline figures, although impressive, need something of a reality check in the context of real-world applications where the overall **system** performance must be considered, not just the performance of the device in isolation.

In practice the PFAC pattern scanning algorithm actually has fairly low computational density and, with the optimisations described previously, it is also relatively memory bandwidth efficient. The **real** limiting factor for all discrete acceleration devices therefore turns out to be the PCIe bandwidth where 8xPCIe3 lanes offer a *theoretical* bandwidth of 7.877 GB/s and in for real world cases is usually closer to ~6.5 GB/s.

Furthermore, if one considers the approach taken for data transfers in the PFAC scan:

```
const cl_int size = input.size(); // input is a std::vector<char>

queue.enqueueWriteBuffer(inputBuffer, CL_TRUE, 0, size, input.data());

/**
 * The kernel processes the input characters in groups of four (OpenCL int),
 * so we therefore need to calculate our global work size in terms of how
 * many OpenCL integers it comprises. The computed global work size is then
 * rounded up to a multiple of the local work-group size (thread block size).
 */

// Number of OpenCL integers that would completely contain the input bytes.
const cl_int n = (size + sizeof(cl_int) - 1)/sizeof(cl_int);

// Given n round up if necessary to a multiple of WORK_GROUP_SIZE.
const auto r = n % WORK_GROUP_SIZE;
const auto global = (r == 0) ? n : n + WORK_GROUP_SIZE - r;

const cl_int initialState = dictionary->initialState;

pfacKernel.setArg(0, initialTransitions);
pfacKernel.setArg(1, hashRow);
pfacKernel.setArg(2, hashVal);
pfacKernel.setArg(3, initialState);
pfacKernel.setArg(4, inputBuffer);
pfacKernel.setArg(5, outputBuffer);
pfacKernel.setArg(6, size);
pfacKernel.setArg(7, n);

queue.enqueueNDRangeKernel(pfacKernel,
                           cl::NullRange, // Offset value is zero.
                           cl::NDRange(global),
                           cl::NDRange(WORK_GROUP_SIZE));

queue.enqueueReadBuffer(outputBuffer, CL_TRUE, 0,
                        size*sizeof(cl_int), output.data());
```

We transfer a **char** sequence to the device representing the data to be scanned, but we transfer an **int** sequence from the device representing the matched pattern ID at a given index. For the synchronous scan we have followed a fairly standard pattern where we do enqueueWriteBuffer→enqueueNDRangeKernel→enqueueReadBuffer using blocking transfers, so each byte scanned actually requires five bytes of data to be transferred (one Host→Device plus four Device→Host) so we can only achieve around 6.5/5 or ~1.3 GB/s.

The real world situation is more subtle than this, as the input and output data used in the scan is in the form of C++ std::vector<char> and std::vector<std::int32_t> respectively, *using the default allocator*. Although using standard vectors with default allocators is the most flexible approach from an API perspective it imposes an inefficiency, especially for small transfers, as PCIe transfers generally perform better from pinned (page locked) memory and using non-pinned memory tends to result in the driver transparently doing an additional copy to a pinned buffer. For a description of pinned memory see section 3.1 of http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

In practice the actual observed implications of this somewhat depend on available CPU performance; in the author's case using an i7-5820K CPU the observed throughput over 8xPCIe3 lanes and a buffer size of 1.5MB is 1.136 GB/s and for 16xPCIe3 lanes this rises to 1.97 GB/s. In both cases this uses a single CPU thread at 100% and clocked at 4.1GHz.

The CPU running at 100% indicates, at least in part, the penalty of the OpenCL driver copying from “normal” memory to pinned memory (it needs to use page locked memory to perform the DMA across PCIe) it also suggests that there may be several options available for improving throughput (some of which may be complementary):

- Use overlapped non-blocking transfers and multiple Command Queues.
- Use multiple client threads.
- Reduce/compact the data being transferred.
- Use pinned memory.

Use overlapped non-blocking transfers and multiple Command Queues

The scan code illustrated previously used **blocking** data transfers, for example:

```
queue.enqueueWriteBuffer(inputBuffer, CL_TRUE, 0, size, input.data());
```

From the OpenCL spec https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf#page=73 we note the following:

If blocking_read is CL_TRUE i.e. the read command is blocking, clEnqueueReadBuffer does not return until the buffer data has been read and copied into memory pointed to by ptr.

If blocking_read is CL_FALSE i.e. the read command is non-blocking, clEnqueueReadBuffer queues a non-blocking read command and returns. The contents of the buffer that ptr points to cannot be used until the read command has completed. The event argument returns an event object which can be used to query the execution status of the read command. When the read command has completed, the contents of the buffer that ptr points to can be used by the application.

If blocking_write is CL_TRUE, the OpenCL implementation copies the data referred to by ptr and enqueues the write operation in the command-queue. The memory pointed to by ptr can be reused by the application after the clEnqueueWriteBuffer call returns.

If blocking_write is CL_FALSE, the OpenCL implementation will use ptr to perform a non-blocking write. As the write is non-blocking the implementation can return immediately. The memory pointed to by ptr cannot be reused by the application after the call returns. The event argument returns an event object which can be used to query the execution status of the write command. When the write command has completed, the memory pointed to by ptr can then be reused by the application.

By making use of non-blocking writes and reads it is possible to have the Host→Device transfer, the Kernel execution and the Device→Host transfer overlap such that, for example, the Host→Device transfer for the second scan can begin as soon as the Host→Device transfer for the first scan has completed rather than having to wait until the Kernel execution and Device→Host transfer has completed as is the case with the previous synchronous/blocking scan implementation. With all other things being equal overlapping transfers *should* make the overall throughput converge on the Device→Host transfer throughput giving a theoretical throughput of ~6.5/4 or 1.625 GB/s over 8xPCIe3 lanes, where the divide by four is again due to the use of std::vector<std::int32_t>.

Making effective use of overlapping transfers turns out to be somewhat fiddly and awkward and suffers from a distinct lack of useful examples, moreover if one wishes to hide the intricacies from client code the best approach is to use an asynchronous model (for which even fewer examples appear to exist), despite OpenCL supporting clSetEventCallback. Some important observations made when implementing overlapped transfers are:

1. It is necessary to use multiple Command Queues.
2. It is necessary to use multiple Device I/O buffers.
3. It is necessary to prevent premature reuse of client input/output data pointers.
4. It is necessary to “wrap” callback state.

As the name suggests each Command Queue is a queue, therefore without multiple Command Queues no overlapping can take place, fortunately it is fairly easy to make use of multiple Command Queues using a simple array.

Similarly, but perhaps less obvious than the need to use multiple Command Queues, is the need to use multiple Device buffers. This is necessary because it is possible, for example, for say multiple Host→Device transfers to be pending whilst a Device→Host transfer is in progress. Without multiple Device buffers it is possible for the second Host→Device transfer to overwrite the Device buffer, especially for the case of large transfers.

The following code snippet illustrates the use of multiple Command Queues and Buffers.

```
// In the scanner-opencl.h header
static constexpr auto COMMAND_QUEUES = 3;
static constexpr auto BUFFERS = 2;
int scanCount; // Count of async scan calls, used to identify CommandQueue.
std::array<cl::CommandQueue, COMMAND_QUEUES> queue;
std::array<cl::Buffer, BUFFERS> inBuffer;
std::array<cl::Buffer, BUFFERS> outBuffer;

…

// In scan(const std::vector<char>& input, std::vector<std::int32_t>& output, Callback f)
auto qid = scanCount % COMMAND_QUEUES; // CommandQueue ID
auto bid = scanCount % BUFFERS;        // Buffer ID
scanCount++;

queue[qid].enqueueWriteBuffer(inBuffer[bid], CL_FALSE, 0, size, input.data());

…
```

Each call to the asynchronous scan obtains a queue and buffer ID from the scanCount by taking the scanCount modulo the number of Command Queues or Buffers and uses this ID to select which Command Queue and Buffer to use for the current transfers.

One of the fiddliest aspects of using overlapping transfers is the need to prevent premature reuse of client input/output buffers, because the OpenCL specification says: “The memory pointed to by ptr cannot be reused by the application after the call returns.” “When the write command has completed, the memory pointed to by ptr can then be reused by the application.”.

This implies that some form of synchronisation between the client application and the asynchronous scan is necessary as, by default, calls to scan using non-blocking transfers called in a tight loop could quickly exceed the number of available Command Queues in use (and thus the available concurrency). Rather than forcing the client code to explicitly include synchronisation the approach that has been taken is to implement blocking when the available level of concurrency (number of Command Queues) has been exceeded.

Another place where it is necessary to manage reuse of data is when implementing the callback mechanism. The OpenCL clEventCallback uses a fairly standard C callback idiom even when using the OpenCL C++ wrapper, the signature for setCallback is:

```
cl_int setCallback(cl_int type,
                   void (CL_CALLBACK * pfn_notify)(cl_event, cl_int, void *),
                   void * user_data = NULL)
```

In C++ it is more useful to be able to use callable objects (including lambdas) as callbacks, so the approach taken is to “wrap” the C++ callback in a CallbackWrapper and use the CallbackWrapper instances as the user_data of the OpenCL setCallback call as follows.

```
// In pfac.h
using Callback = std::function<void(const std::vector<char>& input,
                                    std::vector<std::int32_t>& output)>;

…

// In scanner-opencl.h
template<std::size_t N>
struct CallbackWrapper {
    Callback callback = [](const std::vector<char>& input,
                           std::vector<std::int32_t>& output) {};

    const std::vector<char>* input;
    std::vector<std::int32_t>* output;
    CircularStore<CallbackWrapper<N>, N>* store;

    cl::Event bufferReadEvent;
};

…

// In scan(const std::vector<char>& input, std::vector<std::int32_t>& output, Callback f)
callback.bufferReadEvent.setCallback(CL_COMPLETE,
                                     [](cl_event event, cl_int status, void* c) {
    auto& callback = *static_cast<CallbackWrapper<COMMAND_QUEUES>*>(c);
    callback.callback(*callback.input, *callback.output);
    callback.store->release(callback);
}, static_cast<void*>(&callback));
```

In the asynchronous scan we obtain instances of CallbackWrapper objects and assign the addresses of the input and output buffers and C++ callback to the wrapper.

Because we need an instance of CallbackWrapper for each pending callback (one for each Command Queue) we implement a CircularStore mechanism for obtaining CallbackWrapper instances, this behaves somewhat like a blocking circular queue except that it uses pre-allocated instances, using a get() method to obtain a CallbackWrapper instance and a release() method to release the instance back to the store (the release method is illustrated above in the setCallback lambda).

The blocking get() method of the CircularStore and the associated release() method also provide the synchronisation needed to avoid premature reuse of client input/output buffers. The complete code for the asynchronous scan is as follows:

```
const cl_int size = input.size();

auto& callback = callbackStore.get();
callback.callback = std::move(f);
callback.input = &input;
callback.output = &output;
callback.store = &callbackStore;

auto qid = scanCount % COMMAND_QUEUES; // CommandQueue ID
auto bid = scanCount % BUFFERS;        // Buffer ID
scanCount++;

queue[qid].enqueueWriteBuffer(inBuffer[bid], CL_FALSE, 0, size, input.data());

/**
 * The kernel processes the input characters in groups of four (OpenCL int),
 * so we therefore need to calculate our global work size in terms of how
 * many OpenCL integers it comprises. The computed global work size is then
 * rounded up to a multiple of the local work-group size (thread block size).
 */

// Number of OpenCL integers that would completely contain the input bytes.
const cl_int n = (size + sizeof(cl_int) - 1)/sizeof(cl_int);

// Given n round up if necessary to a multiple of WORK_GROUP_SIZE.
const auto r = n % WORK_GROUP_SIZE;
const auto global = (r == 0) ? n : n + WORK_GROUP_SIZE - r;

const cl_int initialState = dictionary->initialState;

pfacKernel.setArg(0, initialTransitions);
pfacKernel.setArg(1, hashRow);
pfacKernel.setArg(2, hashVal);
pfacKernel.setArg(3, initialState);
pfacKernel.setArg(4, inBuffer[bid]);
pfacKernel.setArg(5, outBuffer[bid]);
pfacKernel.setArg(6, size);
pfacKernel.setArg(7, n);

queue[qid].enqueueNDRangeKernel(pfacKernel,
                                cl::NullRange, // Offset value is zero.
                                cl::NDRange(global),
                                cl::NDRange(WORK_GROUP_SIZE));

queue[qid].enqueueReadBuffer(outBuffer[bid], CL_FALSE, 0,
                             size*sizeof(cl_int), output.data(),
                             NULL, &callback.bufferReadEvent);

callback.bufferReadEvent.setCallback(CL_COMPLETE,
                            [](cl_event event, cl_int status, void* c) {
    auto& callback = *static_cast<CallbackWrapper<COMMAND_QUEUES>*>(c);
    callback.callback(*callback.input, *callback.output);
    callback.store->release(callback);
}, static_cast<void*>(&callback));
```

Running a similar benchmark to that used for the synchronous scan, in the author's case using an i7-5820K CPU the observed throughput over 8xPCIe3 lanes and a buffer size of 1.5MB is 1.293 GB/s and for 16xPCIe3 lanes this rises to 2.28 GB/s. In both cases this appears to use multiple CPU threads with top showing just over 200% and the CPU cores clocked at 4.1GHz.

Although the overall throughput is clearly somewhat greater than for the single threaded synchronous scan it is not quite as high as one might hope, which can probably be attributed to thread creation and synchronisation implemented in the OpenCL driver, which is transparent giving the user no control over how the threads are managed. Moreover the asynchronous callback based approach makes client code somewhat more complex and although the callbacks themselves are fairly straightforward the need to maintain (and not reuse) the input an output buffers until the callback for any give call to the asynchronous scan has completed can make managing the buffers correctly a little fiddly.

Use multiple client threads

An alternative approach for overlapping transfers is to implement threading in the client code. There are two main approaches, the first is to have a single scanner instance and have each thread call scan() on that instance, the second approach is to create multiple scanner instances, one for each thread, and have each thread use its own instance.

At face value the first approach looks appealing as only a single set of Device resources (such as Buffers) need to be allocated, however although most OpenCL calls are thread safe calls to clSetKernelArg are not unless concurrent calls operate on different cl_kernel objects https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/clSetKernelArg.html which implies that the pfacKernel.setArg() calls would need to be protected by a mutex.

Another, more significant, issue with the first approach is that multiple client threads calling the same scanner instance **will use the same Command Queue**, which will greatly limit the available throughput of this method. Some initial prototyping of this approach did indeed show that the throughput was similar to, and indeed a little worse than, calling the synchronous scan from a single thread, which supports the hypothesis that **it is essential** to use multiple Command Queues.

The second approach of using a scanner instance per thread is relatively straightforward and simply requires that each thread creates and uses its own scanner instance populated with the same dictionary data as the other threads, each thread can then run using its own scanner instance relatively independently of the other threads.

The main disadvantage of the second approach is that each scanner instance will consume its own Device resources (such as Buffers and Command Queues). In practice however, in order to actually achieve useful concurrency it is necessary to use multiple Command Queues and Device Buffers (and probably Kernel instances too), so in reality the only *actual* disadvantage of using multiple scanner instances is that the Device Buffer and Image1DBuffer objects used to hold the Dictionary hash table are duplicated.

The following code snippet from simple-benchmark-threaded.cpp illustrates a trivial example showing multiple threads, each calling the synchronous scan method of its own scanner instance to illustrate the raw data transfer and scan throughput.

```
std::vector<std::thread> threads;

// Read the text we want to scan into memory.
auto input = std::vector<char>(text.begin(), text.end());

auto start = std::chrono::steady_clock::now();
std::atomic<bool> first(true);

for (auto t = 0; t < numThreads; t++) {
    threads.push_back(std::thread([&]() {
        // Create output vector.
        std::vector<std::int32_t> output(input.size());

        // Create scanner instance.
        gimbatuluk::PFAC pfac(device, input.size());

        // Read entire dictionary file into memory.
        pfac.loadDictionary(gimbatuluk::readFile(dictionary));

        // Compile and install dictionary onto Device.
        pfac.installDictionary();

        if (first) {
            start = std::chrono::steady_clock::now();
            first = false;
        }

        for (auto i = 0; i < iterations; i++) {
            pfac.scan(input, output);
        }
    }));
}

for (auto& thread : threads) {
    thread.join();
}

auto end = std::chrono::steady_clock::now();
auto duration = std::chrono::
    duration_cast<std::chrono::milliseconds>(end - start).count()/1000.0;
```

Running a similar benchmark to that used for the single threaded synchronous scan, in the author's case using an i7-5820K CPU and two threads the observed throughput over 8xPCIe3 lanes and a buffer size of 1.5MB is 1.53 GB/s and for 16xPCIe3 lanes this rises to 2.927 GB/s, in both cases top shows 200% and the CPU cores clocked at 4.1GHz.

For a relatively sparse input vector, that is to say one with few patterns matching the dictionary, increasing the number of threads above two gives no improved throughput, which is to be expected as the kernel should run extremely quickly as most threads will terminate almost immediately on the initial transition. A much more aggressive test was also run where the 5MB English words file used as the test dictionary was also used as the text to be scanned. This dictionary contains 479829 patterns (words) and when used as the text being scanned clearly each word will match the dictionary *at least* once (though some words match multiple patterns, for example hers will match he, her and hers). Even with such an aggressive test we see two threads give a throughput of 1.444 GB/s with 8 lanes and 2.525 GB/s with 16 lanes, with three threads we see 1.552 GB/s with 8 lanes and 2.47 GB/s with 16 lanes and with four threads we see 1.554 GB/s with 8 lanes and 2.66 GB/s with 16 lanes suggesting that the PCIe transfers remain the bottleneck.

Reduce/compact the data being transferred

The transfer + compute throughput results observed from the multi-threaded tests above, where we see that using 16 PCIe3 lanes gives a throughput between 1.6 and 1.9 times greater than that observed when using 8 PCIe3 lanes, suggests that the PCIe transfers represent the main system bottleneck.

This observation points to a hypothesis that implementing some mechanism to reduce the volume of data being transferred over PCIe should increase the overall system throughput, even if some additional processing cost is incurred in the OpenCL Kernel. It is however worth noting that the reduced increase over 16 lanes for the case of the very aggressive test case of using the 5MB dictionary as the data being scanned (giving 479829 matching patterns) might point to the pfacKernel beginning to become more heavily exercised, which we should bear in mind when considering any potential optimisations.

As the Host→Device transfer represents the vector of bytes that we wish to scan a useful approach for reducing the volume of data transferred from Host→Device isn't immediately obvious. The Device→Host transfer on the other hand seems a much more obvious candidate as it is a vector of **integers**, that is to say it has **four times** the transfer cost than the data we are actually scanning because this vector represents the ID of the matched pattern at the index within the scanned data where the pattern was found.

Rather than returning a std::vector<int>, where the index represents the position of the matched pattern in the scanned data and the value represents the pattern ID, an alternative approach would be to return a vector of structs comprising the position and ID of the matched pattern. For cases where relatively few patterns in the input vector are expected to match the dictionary this approach should significantly reduce the cost of the Device→Host transfer, however if the number of matching patterns exceeds half the size of the input vector then the transfer cost will start to exceed that of the simpler approach.

The hypothesis of reducing the sparse vector of ints to a dense vector of structs seems a sound one as although we have previously illustrated a very aggressive test where we match multiple patterns at every index, in practice a more typical use case is where only a modest number of patterns match, for example in a Network Intrusion scenario where we may be scanning packets and matching them against a Virus database or Snort patterns.

Before spending too much energy implementing a compaction Kernel it's worth getting a feel for any potential improvements by modifying the scan to first remove Device→Host transfer completely and then see the throughput with different output transfer sizes.

A set of throughput tests were carried out to simulate the potential gains of implementing a reduce Kernel. The approach taken was to use a range of input data sizes containing different numbers of matching patterns (12, 256 and 1024) in addition a more aggressive test scanning the entire dictionary and the dictionary truncated to 1.5MB were included. The output sizes reflect the number of patterns multiplied by the size of a struct of two ints and the tests include an additional read of one int to simulate retrieving the number of matching patterns from the device prior to the main data read. To get a “best case” result a test where the Device→Host transfer was completely removed was also carried out.

The following table illustrates the observed results.

	Output size 0							
	8xPCIe3 lanes8 PCIe3 lanes				16xPCIe3 lanes			
	1 thread	2 threads	3 threads	4 threads	1 thread	2 threads	3 threads	4 threads
Input size 1500000 12 matches	4.236 GB/s	5.658 GB/s	5.701 GB/s	5.666 GB/s	6.426 GB/s	9.157 GB/s	10.34 GB/s	10.54 GB/s
Input size 4953699 12 matches	5.011 GB/s	5.98 GB/s	5.968 GB/s	5.987 GB/s	8.424 GB/s	11.6 GB/s	11.6 GB/s	11.63 GB/s
Input size 1500000 479829 matches	1.829 GB/s	2.722 GB/s	2.722 GB/s	2.73 GB/s	2.092 GB/s	2.686 GB/s	2.712 GB/s	2.712 GB/s
Input size 4953699 479829 matches	1.981 GB/s	2.559 GB/s	2.813 GB/s	2.78 GB/s	2.339 GB/s	2.796 GB/s	2.762 GB/s	2.768 GB/s
	Output size 96 (matched 12 patterns returning a vector of structs of two ints)							
Input size 1500000 12 matches	4.046 GB/s	5.659 GB/s	5.724 GB/s	5.722 GB/s	5.983 GB/s	8.544 GB/s	10.33 GB/s	10.28 GB/s
Input size 4953699 12 matches	4.955 GB/s	5.985 GB/s	6.000 GB/s	5.98 GB/s	8.201 GB/s	11.54 GB/s	11.59 GB/s	11.62 GB/s
Input size 1500000 479829 matches	1.795 GB/s	2.713 GB/s	2.730 GB/s	2.687 GB/s	2.099 GB/s	2.692 GB/s	2.686 GB/s	2.688 GB/s
Input size 4953699 479829 matches	1.967 GB/s	2.557 GB/s	2.786 GB/s	2.799 GB/s	2.318 GB/s	2.795 GB/s	2.778 GB/s	2.76 GB/s
	Output size 2048 (matched 256 patterns returning a vector of structs of two ints)							
Input size 1500000 256 matches	3.865 GB/s	5.613 GB/s	-	-	5.547 GB/s	8.147 GB/s	-	-
Input size 4953699 256 matches	4.915 GB/s	5.976 GB/s	-	-	8.106 GB/s	11.53 GB/s	-	-
Input size 1500000 479829 matches	1.790 GB/s	2.718 GB/s	-	-	2.091 GB/s	2.686 GB/s	-	-
Input size 4953699 479829 matches	1.965 GB/s	2.55 GB/s	-	-	2.317 GB/s	2.790 GB/s	-	-
	Output size 8192 (matched 1024 patterns returning a vector of structs of two ints)							
Input size 1500000 1024 matches	3.771 GB/s	5.617 GB/s	-	-	5.524 GB/s	8.073 GB/s	-	-
Input size 4953699 1024 matches	4.861 GB/s	5.963 GB/s	-	-	8.065 GB/s	11.55 GB/s	-	-
Input size 1500000 479829 matches	1.786 GB/s	2.723 GB/s	-	-	2.077 GB/s	2.685 GB/s	-	-
Input size 4953699 479829 matches	1.964 GB/s	2.554 GB/s	-	-	2.319 GB/s	2.789 GB/s	-	-

Looking in particular at the results for two threads we see that the number of matching patterns present in the input test data makes only a modest difference to the throughput and indeed a further test where the input contained 16384 matching patterns (and thus an output data size of 131072 bytes) yielded a throughput of 5.490 GB/s when the input size was 1500000 and 5.92 GB/s when the input size was 4953699. Where the throughput fell off was only for cases of fairly extreme numbers of matching patterns, for example the dictionary of size 4953699 contains 479829 patterns, all of which would match at least once when used as input test data, which is non-representative for realistic use cases.

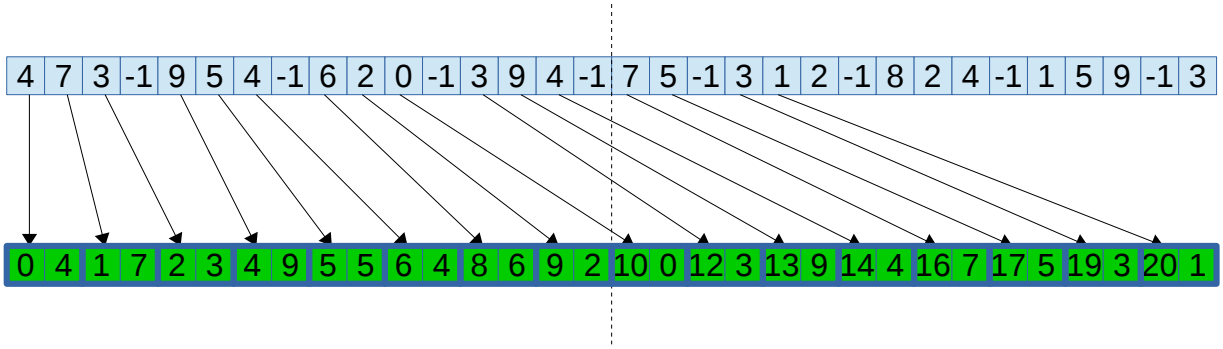
The mechanics of stream compaction are covered in detail in the later section OpenCL Device Stream Compaction.

Use pinned memory and memory mapped data transfers

See TODO section.

8. OpenCL GPU Device Stream Compaction Optimisations

Stream compaction is conceptually straightforward and represents the act of scanning a vector of data to remove invalid/unwanted elements and producing a dense output vector that contains only wanted elements which represent the position and value of the data in the original vector. In other words it represents a mechanism for packing sparse data.



In the example above we may see that the first 20 elements of the original array have been transformed into a 16 element array, noting that for data which isn't particularly sparse, such as the example above, we could actually see an overall data **expansion** because the output array comprises index + value information. In general however sparse data such as found in *realistic* pattern scanning scenarios will yield significant compaction.

Performing the transformation illustrated above on a CPU as a sequential algorithm is relatively trivial, where an obvious approach would be to iterate through the original array recording the index and value of each element as index/value pair structures in the new array and skipping any invalid elements of the original array.

Unfortunately, for the case of massively parallel architectures such as GPUs such a simplistic sequential approach would be very inefficient, so we have to consider an algorithm more suitable for parallel implementation.

The basis for an approach more amenable to parallel implementation is the prefix sum (AKA partial sum or scan) operation https://en.wikipedia.org/wiki/Prefix_sum, where, for the case of compaction, we first perform a boolean scan of the input data using a predicate where valid items are 1 and invalid items are 0. Taking the original array above and including some additional elements for illustration we start with:

```
4 7 3 -1 9 5 4 -1 6 2 0 -1 3 9 4 -1 7 5 -1 3 1 2 -1 8 2 4 -1 1 5 9 -1 3
9 1 8 9 -1 2 8 2 4 -1 9 -1 3 5 7 4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

This corresponds to a bit pattern as follows:

```
11101110111011101101110111011101 11110111101011110000000000000000
```

Performing a boolean inclusive scan operation on the above data will yield:

```
1  2  3  3  4  5  6  6  7  8  9  9  10 11 12 12
13 14 14 15 16 17 17 18 19 20 20 21 22 23 23 24
25 26 27 28 28 29 30 31 32 32 33 33 34 35 36 37
37 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37
```

Thus the value of each scan entry (minus one) yields the index into the compacted output array so we can see that, for example, the 9th element of the original array (in red above) maps to the 7th entry of the compacted array.

Efficient Parallel Prefix Sum (Scan)

Given the existence of such a prefix sum array we may see how a mapping from the original array to the compact array may be performed in a highly parallel manner, which leaves the question of how one may actually perform the prefix sum scan in parallel.

Fortunately as the prefix sum is a key building block for many other parallel algorithms such as stream compaction (as noted above), tree traversal and building, parallel sorting, etc. there has been considerable research, most notably the papers by Blelloch https://pdfs.semanticscholar.org/c94b/96c5f80e5326c172fa1a8875c6a897478bb8.pdf and http://www.cs.cmu.edu/afs/cs/academic/class/15750-s11/www/handouts/PrefixSumBlelloch.pdf

Implementations have become fairly widespread, with an early work-efficient prefix sum presented here http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html. More recent work has however observed that work-efficiency isn't *necessarily* the only factor to consider and that as many GPU Devices are based on SIMT architectures https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads. It is necessary to take into account that SIMT Devices group blocks of threads into a set of “lanes” than can run in true SIMD https://en.wikipedia.org/wiki/SIMD called a warp (32 lanes) by Nvidia and wavefront (64 lanes) by AMD. The implication of this is that as the threads within a warp or wavefront execute **in lock-step** in a SIMT fashion, there is actually no advantage in decreasing work at the expense of increasing the number of steps taken. That is to say each instruction executed by a warp has the same cost, whether executed by a single thread or all threads of the warp. This observation was explained in the paper https://research.nvidia.com/sites/default/files/publications/nvr-2008-003.pdf leading to the intra-block (or intra-warp) scan algorithm, implemented in the pfacCompact Kernel thus:

```
static inline int warpScanInclusive(const int idata, int id, volatile local int* data) {
    int pos = 2 * id - (id & (WARP_SIZE - 1));
    data[pos] = 0; // Padding to enable indexing without conditionals.

    pos += WARP_SIZE; // 32 on Nvidia, 64 on AMD
    data[pos] = idata;

    for (int offset = 1; offset < WARP_SIZE; offset <<= 1) {
        data[pos] += data[pos - offset];
    }

    return data[pos];
}
```

This function is optimised to operate within a warp in three ways. First, it assumes that the number of threads in a warp is 32 (or 64), which limits the number of parallel steps to five. Second, because it assumes that the threads of a warp execute synchronously, it omits barrier(CLK_LOCAL_MEM_FENCE) that would otherwise be necessary when using shared memory. Finally, by padding the intermediate storage for each warp with zeros, it ensures legal indexing of the shared memory array without conditionals (note that this requires extra shared memory). The data variable is assumed to be a pointer to shared memory, and it must be declared volatile as shown to ensure that the stores to shared memory are executed otherwise, in the absence of barrier synchronization, the compiler may choose to keep intermediate values in registers. Because data is declared as volatile, it must be reread from shared memory on every reference.

Boolean Prefix Sum

In a number of applications, the input elements to scan are single bits with a value of zero or one. Alternatively, we can consider them to be Boolean predicates. There are two common operations on sequences of elements with corresponding Boolean true/false predicates: stream compaction (also known as filter), and split.

Because of the importance of binary scans, the architects of the Nvidia Fermi GPU added some new operations for improving their efficiency. These operations allow the threads of a CUDA warp to compute cooperatively. The first is a new __ballot() intrinsic function for collecting “votes” from a warp of threads, and the second is a new scalar intrinsic called __popc() (short for “population count”). Using these new instructions we can construct efficient binary scans that execute fewer instructions and require much less shared memory than equivalent code that uses full 32-bit operations. The background is described in this paper http://booksite.elsevier.com/samplechapters/9780123859631/Chapter_3.pdf.

Unfortunately, although an equivalent for the CUDA __popc() intrinsic is available in OpenCL 1.2 (popcount()) there is no *portable* equivalent of the CUDA warp voting instructions, so we implement a non-portable version for Nvidia using conditional blocks.

This uses the OpenCL 1.2 popcount (population count) to count the number of bits set and ballot, which evaluates the predicate for all active threads of the warp and returns a uint whose Nth bit is set if and only if predicate evaluates to non-zero for the Nth thread of the warp and the Nth thread is active. This is restricted to Nvidia >= Fermi because of ballot.

```
#if __NV_CL_C_VERSION >= 120
static inline uint ballot(const int predicate) {
    uint result;
    asm volatile(
        "setp.ne.u32 %%p1, %1, 0;"
        "vote.ballot.b32 %0, %%p1;"
        : "=r"(result)
        : "r"(predicate)
    );
    return result;
}

static inline int warpScanInclusiveBool(const int idata, int id) {
    return popcount(ballot(idata) << (31 - id));
}
#endif
```

The operation of warpScanInclusiveBool is best explained by returning to the earlier example where the first part of the data set is as follows:

```
4 7 3 -1 9 5 4 -1 6 2 0 -1 3 9 4 -1 7 5 -1 3 1 2 -1 8 2 4 -1 1 5 9 -1 3
```

That data represents the result of the PFAC lookup and it is important to remember that each result has been computed by its *own thread*, so those 32 results represent the output from the first warp of processing. Running the ballot call on each of these threads with a predicate of idata >= 0 will result in **each thread** returning 3149625207. Each thread because with warp vote functions the results are combined (reduced) across the active threads of the warp, broadcasting a single return value to each participating thread and 3149625207 because the binary equivalent (noting the LSB is on the right hand side) is:

```
10111011101110110111011101110111
```

By shifting the result of the ballot by 31 minus the warp lane ID (because it is the *least significant bit* of the ballot result that represents lane zero) the results for each lane are:

```
lane 0:   3149625207 << (31 – 0) =  2147483648 1000 0000 0000 0000 0000 0000 0000 0000
lane 1:   3149625207 << (31 – 1) =  3221225472 1100 0000 0000 0000 0000 0000 0000 0000
lane 2:   3149625207 << (31 – 2) =  3758096384 1110 0000 0000 0000 0000 0000 0000 0000
lane 3:   3149625207 << (31 – 3) =  1879048192 0111 0000 0000 0000 0000 0000 0000 0000
lane 4:   3149625207 << (31 – 4) =  3087007744 1011 1000 0000 0000 0000 0000 0000 0000
lane 5:   3149625207 << (31 – 5) =  3690987520 1101 1100 0000 0000 0000 0000 0000 0000
lane 6:   3149625207 << (31 – 6) =  3992977408 1110 1110 0000 0000 0000 0000 0000 0000
lane 7:   3149625207 << (31 – 7) =  1996488704 0111 0111 0000 0000 0000 0000 0000 0000
lane 8:   3149625207 << (31 – 8) =  3145728000 1011 1011 1000 0000 0000 0000 0000 0000
lane 9:   3149625207 << (31 – 9) =  3720347648 1101 1101 1100 0000 0000 0000 0000 0000
lane 10:  3149625207 << (31 – 10) = 4007657472 1110 1110 1110 0000 0000 0000 0000 0000
lane 11:  3149625207 << (31 – 11) = 2003828736 0111 0111 0111 0000 0000 0000 0000 0000
lane 12:  3149625207 << (31 – 12) = 3149398016 1011 1011 1011 1000 0000 0000 0000 0000
lane 13:  3149625207 << (31 – 13) = 3722182656 1101 1101 1101 1100 0000 0000 0000 0000
lane 14:  3149625207 << (31 – 14) = 4008574976 1110 1110 1110 1110 0000 0000 0000 0000
lane 15:  3149625207 << (31 – 15) = 2004287488 0111 0111 0111 0111 0000 0000 0000 0000
lane 16:  3149625207 << (31 – 16) = 3149627392 1011 1011 1011 1011 1000 0000 0000 0000
lane 17:  3149625207 << (31 – 17) = 3722297344 1101 1101 1101 1101 1100 0000 0000 0000
lane 18:  3149625207 << (31 – 18) = 1861148672 0110 1110 1110 1110 1110 0000 0000 0000
lane 19:  3149625207 << (31 – 19) = 3078057984 1011 0111 0111 0111 0111 0000 0000 0000
lane 20:  3149625207 << (31 – 20) = 3686512640 1101 1011 1011 1011 1011 1000 0000 0000
lane 21:  3149625207 << (31 – 21) = 3990739968 1110 1101 1101 1101 1101 1100 0000 0000
lane 22:  3149625207 << (31 – 22) = 1995369984 0111 0110 1110 1110 1110 1110 0000 0000
lane 23:  3149625207 << (31 – 23) = 3145168640 1011 1011 0111 0111 0111 0111 0000 0000
lane 24:  3149625207 << (31 – 24) = 3720067968 1101 1101 1011 1011 1011 1011 1000 0000
lane 25:  3149625207 << (31 – 25) = 4007517632 1110 1110 1101 1101 1101 1101 1100 0000
lane 26:  3149625207 << (31 – 26) = 2003758816 0111 0111 0110 1110 1110 1110 1110 0000
lane 27:  3149625207 << (31 – 27) = 3149363056 1011 1011 1011 0111 0111 0111 0111 0000
lane 28:  3149625207 << (31 – 28) = 3722165176 1101 1101 1101 1011 1011 1011 1011 1000
lane 29:  3149625207 << (31 – 29) = 4008566236 1110 1110 1110 1101 1101 1101 1101 1100
lane 30:  3149625207 << (31 – 30) = 2004283118 0111 0111 0111 0110 1110 1110 1110 1110
lane 31:  3149625207 << (31 – 31) = 3149625207 1011 1011 1011 1011 0111 0111 0111 0111
```

By counting the ones (popcount) of the result of `ballot(idata) << (31 – id)` we see the values 1, 2, 3, 3, 4, 5, 6, 6, 7, 8, 9, 9, 10, 11, 12, 12, 13, 14, 14, 15, 16, 17, 17, 18, 19, 20, 20, 21, 22, 23, 23, 24 as expected, which correctly represent the final warp scan results for each thread of the first warp.

Intra Work Group Prefix Sum Tree

The warp-efficient prefix sum operations described in the previous sections have the obvious disadvantage that they only perform a scan over the range of 32 elements in a warp and moreover each warp scan resets its sum to zero.

As an example consider again the bit patterns covering two warps from the original example.

```
11101110111011101101110111011101 11110111101011110000000000000000
```

Performing a *complete* boolean inclusive scan operation on the above data would yield:

```
1  2  3  3  4  5  6  6  7  8  9  9  10 11 12 12
13 14 14 15 16 17 17 18 19 20 20 21 22 23 23 24
25 26 27 28 28 29 30 31 32 32 33 33 34 35 36 37
37 37 37 37 37 37 37 37 37 37 37 37 37 37 37 37
```

As described previously.

Performing **warp-optimised** scans will however yield multiple scans, in this case two separate scans as follows, representing each warp under consideration:

```
1  2  3  3  4  5  6  6  7  8  9  9  10 11 12 12
13 14 14 15 16 17 17 18 19 20 20 21 22 23 23 24
```

and

```
1  2  3  4  4  5  6  7  8  8  9  9  10 11 12 13
13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13
```

In order to yield the correct result for a complete scan over the full input stream we must update the result of the warp-optimised scan with the result of a scan of the individual warp scans. In this case the elements of second scan need to have 24 added to them and the results for any subsequent third warp scan would need 37 (24 + 13) added, and so on.

In the pfacCompact Kernel we implement a tree of prefix sums based on the fact that the PFAC algorithm partitions the input into blocks of 1024 characters each processed within an OpenCL Work Group (thread block). Each Work Group comprises 256 threads so each thread processes four characters and there are eight warps per Work Group. The processing required for each thread to compute a Work Group scan is as follows:

```
int scan[4]; // We need an array as we have four results per thread.
local int* warpSum = initialTransitionsCache; // Reuse shared memory.
barrier(CLK_LOCAL_MEM_FENCE);

const int WARPS_PER_WORK_GROUP = WORK_GROUP_SIZE/WARP_SIZE;
const int wid = tid >> WARP_SHIFT;     // Warp ID, WARP_SHIFT = 5
const int lid = tid & (WARP_SIZE - 1); // Lane ID, WARP_SIZE = 32

// Step 1: Perform boolean inclusive scan for each warp.
#pragma unroll
for (int i = 0; i < 4; i++) {
#if __NV_CL_C_VERSION >= 120
    scan[i] = warpScanInclusiveBool(match[i] >= 0, lid);
#else
    scan[i] = warpScanInclusive(match[i] >= 0, tid + i*WORK_GROUP_SIZE, cache);
#endif
    if (lid == (WARP_SIZE - 1)) {
        warpSum[wid + i*WARPS_PER_WORK_GROUP] = scan[i];
    }
}
barrier(CLK_LOCAL_MEM_FENCE);

// Step 2: Perform inclusive scan over the 32 warpSum results.
if (wid == 0) {
    warpScanInclusive(warpSum[lid], lid, cache);
}
barrier(CLK_LOCAL_MEM_FENCE);

// cache entry 63 holds the overall sum of the scan of warpSum results.
int workGroupSum = cache[2*WARP_SIZE - 1];

// Step 3: Update the scan values with the total computed for each warp.
#pragma unroll
for (int i = 0; i < 4; i++) {
    scan[i] += cache[(WARP_SIZE - 1) + wid + i*WARPS_PER_WORK_GROUP] - 1;
}
```

Inter Work Group Prefix Sum Tree

The previous sections have described how a prefix sum within a Work Group may be computed by composing the results of a scan performed over the threads of each warp with a scan over the elements comprising the final sum of the warp scans. Recall, however, that the OpenCL Execution Model partitions the index space based on the number of threads in a Work Group, so for non-trivial data sizes there may therefore be many Work Groups present.

In order to cater for the index space being decomposed over multiple Work Groups we have to extend the prefix sum tree concept over all the Work Groups and calculate a prefix sum of the final workGroupSum computed for each Work Group and use that as a “global offset” to add to the intra Work Group indexes (which each have an origin starting at zero).

Computing the inter Work Group prefix sum has an additional complication however as in OpenCL (and indeed CUDA) synchronisation primitives only exist intra Work Group.

Current state-of-the-art algorithms on GPUs (such as in CUDPP, Thrust and the SDK examples) consist of three phases, the first is the intra Work Group scan as described previously, the second is a global scan of the final sum of each Work Group scan and the third is an “update” stage that computes the final global values from steps one and two. Because each step of this process requires the results of the preceding step, in conventional approaches global barrier synchronisation is required, which in practice requires multiple Kernel launches and ~4n global data movement.

Some of the most recent research has however taken a different approach to the problem: https://research.nvidia.com/sites/default/files/publications/nvr-2016-002.pdf. This paper describes a **single-pass** scan algorithm with “decoupled look-back” between Work Groups.

The method presented is a generalisation of the “Stream Scan” chained-scan approach https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxzaGVuZ2VueWFufGd4OjQ3MjhiOTU3NGRhY2ZlYzA. The aim is to significantly reduce prefix propagation latencies by decoupling the singular dependence of each processor on its immediate predecessor at the expense of progressively redundant computation. Whereas the chained-scan approach has a fixed “look-back” of one Work Group the method presented by Merrill and Garland allows Work Groups to inspect the status of predecessors that are increasingly further away.

The method works as follows (paraphrased from the original paper):
1. Initialise the workGroupSum and inclusivePrefix descriptor values for each Work Group to -1 where the initial value of -1 acts as a flag to indicate that a valid workGroupSum or inclusivePrefix has not yet been written by the Work Group. These values are held in global memory and thus visible to all Work Groups indexed by the Work Group ID.
2. Synchronise processors to ensure a consistent view of the initialised descriptors. In practice steps 1 & 2 are in the form of a clEnqueueCopyBuffer followed by Kernel launch.
3. Compute and record the intra Work Group workGroupSum as described previously.
4. Determine the Work Group's exclusive prefix sum using decoupled look-back. Each Work Group maintains a running exclusivePrefix as it progressively inspects the descriptors of increasingly antecedent Work Groups, beginning with the immediately preceding Work Group.

For each predecessor the current Work Group will conditionally perform the following, based on the state of their workGroupSum and inclusivePrefix descriptors.

- While predecessorWorkGroupSum == -1 continue polling.
- If the predecessorInclusivePrefix == -1 add the predecessorWorkGroupSum to the exclusivePrefix value and continue on to inspect the next preceding Work Group.
- Else add the predecessorInclusivePrefix to the running exclusivePrefix value and terminate the look-back phase.

5. Compute and record the Work Group wide **inclusive** prefixes. Each Work Group adds the running exclusivePrefix to the previously computed workGroupSum and records the result to the current Work Group's inclusivePrefix descriptor in global memory. It then executes a memory fence to ensure the write will be committed to memory and readable by other Work Groups.
6. Update the Work Group wide scan using the Work Group's exclusivePrefix value as a global offset to be added to every output value.

The computation of each Work Group can proceed independently and in parallel with the other Work Groups throughout steps 1, 3, 5 and 6. In step 4 (decoupled look-back) each Work Group must wait on its predecessor(s) to finish step 3 (record the workGroupSum). The code for steps 4 and 5 as used in the pfacCompact Kernel is below.

```
cache[0] = 0; // Reuse first local/shared memory entry to store global offset.
barrier(CLK_LOCAL_MEM_FENCE);
if (tid == 0) { // Thread ID == 0
    /**
     * Store the workGroupSum for the current Work Group and if that is the
     * first Work Group also store workGroupSum to inclusivePrefix. After the
     * write(s) to Global Memory have been made the write_mem_fence ensures
     * the loads will be committed to memory & readable by other Work Groups.
     */
    smem[gid].workGroupSum = workGroupSum;
    write_mem_fence(CLK_GLOBAL_MEM_FENCE);
    if (gid == 0) { // If first Work Group.
        smem[gid].inclusivePrefix = workGroupSum;
    } else { // If not the first Work Group.
        int exclusivePrefix = 0;
        for (int id = gid - 1; id >= 0; id--) {
            // Poll (spinlock) until predecessorWorkGroupSum is set.
            int predecessorWorkGroupSum = -1;
            do { // Force atomic load from global memory.
                predecessorWorkGroupSum = atomic_add(&smem[id].workGroupSum, 0);
            } while (predecessorWorkGroupSum == -1);

            int predecessorInclusivePrefix = smem[id].inclusivePrefix;
            if (predecessorInclusivePrefix == -1) {
                exclusivePrefix += predecessorWorkGroupSum;
            } else {
                exclusivePrefix += predecessorInclusivePrefix;
                break;
            }
        }
        cache[0] = exclusivePrefix; // Store global offset to shared memory.
        smem[gid].inclusivePrefix = workGroupSum + exclusivePrefix;
    }
    write_mem_fence(CLK_GLOBAL_MEM_FENCE);
}
barrier(CLK_LOCAL_MEM_FENCE);

int globalOffset = cache[0];
barrier(CLK_LOCAL_MEM_FENCE);
```

For the final compaction the matching entries from the PFAC lookup are first copied into the local/shared memory cache at the scan indexes computed previously (which are relative to the Work Group). Finally the entries from the cache are copied to the output array, which uses globalOffset to provide the correct global index.

```
local MatchEntry* outputCache = (local MatchEntry*)cache;

#pragma unroll
for (int i = 0; i < 4; i++) {
    if (match[i] >= 0) {
        const int index = firstCharInWorkGroup + tid + i*WORK_GROUP_SIZE;
        outputCache[scan[i]].index = index;
        outputCache[scan[i]].value = match[i];
    }
}
barrier(CLK_LOCAL_MEM_FENCE);

for (int i = tid; i < workGroupSum && globalOffset + i < limit; i += WORK_GROUP_SIZE) {
    output[globalOffset + i] = outputCache[i];
}
```

The following table illustrates the observed results of using the stream compaction algorithm, illustrating significant system (transfer + compute) throughput improvements when compared with using the basic PFAC algorithm and returning the match results in a simple vector of integers.

These results show the importance of considering the overall *system* performance, illustrating the perhaps counter-intuitive notion that it can sometimes be worth expending additional compute resources in order to reduce transfer bottlenecks.

	Output size 96 (matched 12 patterns returning a vector of structs of two ints)							
	8xPCIe3 lanes8 PCIe3 lanes				16xPCIe3 lanes			
	1 thread	2 threads	3 threads	4 threads	1 thread	2 threads	3 threads	4 threads
Input size 1500000 12 matches	3.598 GB/s	5.448 GB/s	5.683 GB/s	5.753 GB/s	5.02 GB/s	7.591 GB/s	9.23 GB/s	7.92 GB/s
Input size 4953699 12 matches	4.262 GB/s	5.948 GB/s	5.941 GB/s	5.964 GB/s	6.502 GB/s	11.32 GB/s	11.58 GB/s	11.54 GB/s
Input size 1500000 479829 matches	1.621 GB/s	2.295 GB/s	2.145 GB/s	2.142 GB/s	1.851 GB/s	2.266 GB/s	2.145 GB/s	2.142 GB/s
Input size 4953699 479829 matches	1.782 GB/s	2.357 GB/s	2.340 GB/s	2.348 GB/s	2.061 GB/s	2.348 GB/s	2.321 GB/s	2.324 GB/s
	Output size 2048 (matched 256 patterns returning a vector of structs of two ints)							
Input size 1500000 256 matches	3.440 GB/s	5.206 GB/s	5.669 GB/s	5.680 GB/s	4.755 GB/s	7.249 GB/s	8.341 GB/s	7.198 GB/s
Input size 4953699 256 matches	4.195 GB/s	5.923 GB/s	5.924 GB/s	5.951 GB/s	6.34 GB/s	11.12 GB/s	11.54 GB/s	11.62 GB/s
Input size 1500000 479829 matches	1.620 GB/s	2.284 GB/s	2.149 GB/s	2.146 GB/s	1.844 GB/s	2.264 GB/s	2.138 GB/s	2.134 GB/s
Input size 4953699 479829 matches	1.782 GB/s	2.359 GB/s	2.341 GB/s	2.341 GB/s	2.06 GB/s	2.345 GB/s	2.321 GB/s	2.323 GB/s
	Output size 8192 (matched 1024 patterns returning a vector of structs of two ints)							
Input size 1500000 1024 matches	3.398 GB/s	5.181 GB/s	5.650 GB/s	5.680 GB/s	4.690 GB/s	7.085 GB/s	8.063 GB/s	6.997 GB/s
Input size 4953699 1024 matches	4.146 GB/s	5.918 GB/s	5.944 GB/s	5.951 GB/s	6.269 GB/s	11.00 GB/s	11.47 GB/s	11.46 GB/s
Input size 1500000 479829 matches	1.617 GB/s	2.290 GB/s	2.148 GB/s	2.146 GB/s	1.847 GB/s	2.267 GB/s	2.136 GB/s	2.131 GB/s
Input size 4953699 479829 matches	1.784 GB/s	2.361 GB/s	2.342 GB/s	2.343 GB/s	2.06 GB/s	2.34 GB/s	2.323 GB/s	2.328 GB/s
	Output size 131072 (matched 16384 patterns returning a vector of structs of two ints)							
Input size 1500000 16384 matches	2.657 GB/s	4.290 GB/s	5.517 GB/s	5.553 GB/s	3.728 GB/s	5.979 GB/s	7.710 GB/s	6.734 GB/s
Input size 4953699 16384 matches	3.793 GB/s	5.874 GB/s	5.884 GB/s	5.916 GB/s	5.709 GB/s	10.05 GB/s	11.26 GB/s	11.35 GB/s
Input size 1500000 479829 matches	1.571 GB/s	2.272 GB/s	2.162 GB/s	2.157 GB/s	1.810 GB/s	2.254 GB/s	2.127 GB/s	2.134 GB/s
Input size 4953699 479829 matches	1.763 GB/s	2.355 GB/s	2.340 GB/s	2.342 GB/s	2.017 GB/s	2.341 GB/s	2.317 GB/s	2.315 GB/s

9. TODO

Use pinned memory

The data transfers used in the PFAC scan use the OpenCL enqueueWriteBuffer and enqueueReadBuffer methods to transfer data from host to device. The default approach has been for the client to pass standard C++ vectors that used the default allocator. That simple approach however has some inefficiencies, especially for small transfers, as PCIe transfers generally perform better from pinned (page locked) memory and using non-pinned memory tends to result in the driver transparently doing an additional copy to a pinned buffer. For a description of pinned memory see section 3.1 of http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

The way to use pinned memory in OpenCL is fairly subtle as OpenCL applications do not have direct control over whether memory objects are allocated in pinned memory or not, but they can create objects using the CL_MEM_ALLOC_HOST_PTR flag and such objects are likely to be allocated in pinned memory by the driver for best performance.

The steps normally needed to use pinned memory are summarised below and described more fully in the Best Practice Guide and the SDK oclBandwidthTest program.

1. Declare cl::Buffer objects for the pinned Host memory and the GPU Device memory and also standard pointers on the Host to reference the pinned Host memory.

2. Allocate the cl::Buffer objects. Noting that these are fairly expensive calls normally best made during application initialisation rather than within performance critical paths. The pinned buffer allocation code would look like this:

```
cl::Buffer pinnedBuffer = cl::Buffer(context,
                                     CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR,
                                     size);
```

3. Map the standard pointers to reference the pinned Host memory:

```
char* data = static_cast<char*>queue.enqueueMapBuffer(pinnedBuffer, CL_TRUE
                                                      CL_MAP_READ | CL_MAP_WRITE, 0,
                                                      size);
```

4. Access the pinned memory created in step 3 using the standard Host pointer and standard Host code exactly like any other raw block of memory.

5. Read or write data from the pinned Host memory to the GPU device using the normal OpenCL enqueueWriteBuffer and enqueueReadBuffer methods to transfer data from host to device in exactly the same way is if non-pinned memory were being used.

The mechanism for creating blocks of pinned memory in itself is relatively straightforward, the more fiddly part is using this memory effectively from client applications. The most effective approach is likely to be to write a custom C++ Allocator.

Use memory mapped data transfers

In many cases high performance OpenCL accelerators are in the form of discrete Devices, connected to a Host by a bus such as PCIe or NVLink. For discrete Devices it is obviously necessary to transfer data from the Host memory space to the Device memory space, which is exactly what OpenCL enqueueWriteBuffer and enqueueReadBuffer do.

There are however a number of cases where the Host and Device are not separate, for example Intel's HD/Iris Graphics https://en.wikipedia.org/wiki/Intel_HD_and_Iris_Graphics, AMD's APU https://en.wikipedia.org/wiki/AMD_Accelerated_Processing_Unit, Nvidia's Jetson http://www.nvidia.co.uk/object/jetson-tk1-embedded-dev-kit-uk.html among others. Indeed the OpenCL CPU device made available by AMD and Intel's OpenCL drivers is also clearly a case where the Host and Device are not separated via a bus.

For cases where the Host and Device are integrated it is generally the case that they share the same physical memory and therefore in those cases the approach of using enqueueWriteBuffer and enqueueReadBuffer is inefficient, as it implies what is in effect an unnecessary copy from Host to Device memory space.

Where the Host and Device share physical memory it is possible to make use of the OpenCL enqueueMapBuffer and enqueueUnmapMemObject methods to memory map buffers between the Host and Device. Memory mapping in this way should yield improved transfer performance for cases where Host and Device share physical memory, but the enqueueMapBuffer and enqueueUnmapMemObject approach seems to yield poorer performance with discrete Devices, which is unsurprising as at some point it is clearly necessary to transfer the data physically between the separate address spaces (I believe that the enqueueUnmapMemObject method does this).

More investigation is required as there is relatively little documentation available on making efficient use of enqueueMapBuffer, moreover the Nvidia SDK example is not correct, it does:

```
//create a buffer cmPinnedData in host
cmPinnedData = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR, memSize,
NULL, &ciErrNum);

....(initialize cmPinnedData with some data)....

//create a buffer in device
cmDevData = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE, memSize, NULL, &ciErrNum);

// get pointer mapped to host buffer cmPinnedData
h_data = (unsigned char*)clEnqueueMapBuffer(cqCommandQueue, cmPinnedData, CL_TRUE, CL_MAP_READ, 0,
memSize, 0, NULL, NULL, &ciErrNum);

// get pointer mapped to device buffer cmDevData
void* dm_idata = clEnqueueMapBuffer(cqCommandQueue, cmDevData, CL_TRUE, CL_MAP_WRITE, 0, memSize, 0,
NULL, NULL, &ciErrNum);

// copy data from host to device by memcpy
for(unsigned int i = 0; i < MEMCOPY_ITERATIONS; i++) {
    memcpy(dm_idata, h_data, memSize);
}
//unmap device buffer.
ciErrNum = clEnqueueUnmapMemObject(cqCommandQueue, cmDevData, dm_idata, 0, NULL, NULL);
```

which superficially looks fine except that the memcpy loop doesn't *actually* transfer data to the Device until the enqueueUnmapMemObject call and the SDK benchmark results for memory mapping a discrete Device are somewhat over-optimistic as a result.

Implement Kernel optimisations for a wider range of Devices

Throughout this paper the references to Kernel code have primarily focussed on the optimisations necessary to run well on GPU Devices and in particular the focus has been on Nvidia SIMT https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads Devices.

One of the features of OpenCL is that it has been designed to be platform neutral, so the need for device specific optimisations are unfortunate. Pragmatically however different architectures have fairly wide variations in the trades that have been made in order to gain performance advantages so some level of tuning for each device type must be expected.

In practice for true Embarrassingly Parallel Problems there are relatively few differences required in order to optimise for different Device types and the basic (non compact) PFAC Kernel runs well on a GPU and also relatively well on the OpenCL CPU Device.

PFAC is essentially an Embarrassingly Parallel, memory bound application and as it happens the main optimisations that have been implemented for the GPU have been put in place to optimise memory bandwidth, for example using texture memory for the state machine and caching the input data from global memory to local memory to increase locality of reference and maximise coalescence of reads and writes. These optimisations result in Kernel code that runs fairly well on a CPU, however the explicit copying/caching from global to local memory isn't necessary on a CPU due to the L1, L2 and L3 caches. Indeed, OpenCL local memory is implemented as a normal block of memory on a CPU, so the copying from global to local memory for the *purpose of caching* is likely to actually reduce performance on a CPU (though for other problems local memory may be necessary as the mechanism required to share state between threads).

Things get more complicated when it becomes necessary to share state between threads. The way SIMT and MIMD architectures behave is fundamentally different, for example SIMT Devices group blocks of threads into a set of “lanes” than can run in true SIMD https://en.wikipedia.org/wiki/SIMD called a warp by Nvidia and wavefront by AMD.

In the pfacCompact Kernel the basis of the stream compaction is a parallel prefix sum, which has been implemented as a tree of prefix sums based on the fact that the PFAC algorithm partitions the input into blocks of 1024 characters each of which are processed within an OpenCL Work Group (thread block). Each Work Group comprises 256 threads so each thread processes four characters and there are eight warps per Work Group. For efficiency the building block of the tree of prefix sums is the warpScanInclusive function, which relies on the fact that all threads in a warp run in lock-step in order to minimise synchronisation points. Each run of warpScanInclusive results in a total sum for the entire warp (32 in all) and those sums are themselves processed in another warpScanInclusive to generate the offsets to be added to the prefix sums generated for each warp.

The warpScanInclusive optimisation hasn't yet been tried on an AMD Device. AMD Devices are detected and WARP_SIZE is set to 64, but it hasn't been tested.

It is not yet clear the best way to optimise for a CPU Device and further research is required. The inter Work Group synchronisation spin-lock may be sub-optimal on a CPU.

Note too that for OpenCL 2.0 Devices the work_group_scan_inclusive_add function could likely be used to replace the tree of warpScanInclusive calls.

Re-order Input Dictionary Lexicographically

Although it hasn't been explicitly spelled out the input dictionary is compiled into the Aho-Corasick state-machine in the order it is parsed. Whilst this makes no difference to the correct operation of the state-machine it should be noted that the state number, and thus its position in memory, is allocated incrementally. Given a dictionary that might contain “overlapping” entries such as he, her, hers it is very likely that cache performance of the state-machine look-up may be improved if the additional states for the “r” then the “s” are located close to those for the “h” and the “e”.

In order to optimise the cache performance for the look-up it may therefore be worthwhile preprocessing the input dictionary such that it is placed in lexicographical order, noting that this will clearly have an adverse effect on dictionary load performance.

Handle Overlapping Matches

Although the Aho-Corasick state-machine correctly supports overlapping matches the results are exported in the form of a matching pattern ID at a given index in the input stream, with only a single result available for each index. What this means in practice is that the returned result is the longest matching pattern. By way of an illustration if we assume a dictionary containing the entries he, her, hers, if we scan input data that simply comprises the word “hers” the result will be 2, -1, -1, -1 indicating pattern ID two matched at input index zero.

Although it may be possible to modify the Kernel to return multiple results this is likely to be inefficient and complex. A better option is to implement additional preprocessing of the input dictionary such that a look-up of pattern ID two also implies that patterns zero and one have matched, for example by using a form of linked list.

Support Regular Expressions

The focus of this work has been on implementing a high-performance Aho-Corasick state-machine look-up and result compaction in OpenCL and so has only considered basic pattern matching.

In real world scenarios it is often desirable to perform scans where the dictionary is in the form of Regular Expressions (RegEx). The implications of this are yet to be explored however given that the OpenCL look-up is in the form of a state-machine the most likely avenue to explore is the use of https://en.wikipedia.org/wiki/Thompson%27s_construction to transform the Regular Expression into an equivalent Nondeterministic Finite Automaton (NFA) followed by the https://en.wikipedia.org/wiki/Powerset_construction to transform the NFA into an equivalent Deterministic Finite Automaton (DFA).

As the PFAC state-machine lookup is a generic state-machine traversal algorithm it is expected that the changes required to support Regular Expressions would be limited to dictionary construction and compilation and should require no change to the Kernels.

Support Dictionary Updates

At the moment the API only supports an entire Dictionary being loaded and installed, it might be useful to support finer grained Dictionary manipulation. It should currently be possible to load and install a new dictionary on a running instance, however that hasn't yet been extensively tested.