**Regexp**

Since DFAs have the potential of state explosion for complex regular expressions, generating DFA representations without exceeding the GPU memory size becomes an important problem. We propose techniques to reduce the DFA size and optimize regular expression matching on GPU. This chapter is structured as follows. In Section 3.1, we show the basic approach to implement regular expression matching on GPU. In Section 3.2, we discuss an optimization algorithm called alphabet reduction. We propose different approaches to implement alphabet reduction on GPU in Section 3.3 and 3.4. In Section 3.5, we describe two novel clustering algorithms for regular expressions. Our algorithms allow achieving smaller number of DFAs that fit GPU memory. Finally, in Section 3.6 we perform experiments on both real and synthetic pattern-sets using our DFA-based search engine

Basic Implementation

Since GPUs are memory-based hardware platforms. In this case, DFA representation is preferred. DFAs need to be stored in GPU global memory. We take advantage of the uncompressed DFA-based solution from [34]. In general, different threads process different DFAs in the same thread-block. Different input streams are again mapped onto different thread-blocks. Because the ASCII table size is 256, we will store 256 transitions for each state in the memory layout. However, if the regular expression is complex and causes the DFA to have large number of states, the memory requirement may exceed the 17 GPU memory capacity. In this case, we can apply the alphabet reduction algorithm described in Section 3.2 to reduce the size of transition table and we can implement the clustering algorithms discussed in Section 3.5 to divide the regular expressions into partitions and generate smaller number DFAs that fit memory size.

Alphabet Reduction

The idea at the basis of alphabet reduction is the following: in a DFA recognizing regular expressions over an alphabet Σ, each state has potentially |Σ| outgoing transitions, one for each symbol in Σ. However, Σ can be partitioned into classes of symbols C1,..,Ck which are indistinguishable for the purposes of the DFA operation. Two symbols ci and cj will fall into the same class if they are treated the same way in all DFA states. In other words, given the transition function δ(states, Σ)→states, δ(s,ci)= δ(s,cj) for each state s in the DFA. Once the class translation C(Σ)→ {1..k} has been computed, the alphabet is reduced from cardinality |Σ| to k. k next state transitions will therefore suffice at each procedure alphabet\_reduction (DFA dfa=(n, δ(states, Σ)), modifies set class); (1) int alphabet\_size = 0; (2) for state s ∈ states do (3) for state t ∈ states do (4) set char\_covered[|Σ|] = false; (5) set class\_covered[|Σ|] = false; (6) set remap[|Σ|] = 0; (7) for (char c ∈ Σ & δ(s,c)=t) do (8) char\_covered[c] = true; (9) class\_covered[class[c]]=true; (10) for (char c ∈ Σ) do (11) if (!char\_covered [c] & class\_covered[class[c]]) then (12) if (remap[class[c]]==0) then remap[class[c]]= ++alphabet\_size; (13) class[c]=remap[class[c]]; end; 18 state. An additional alphabet translation table encoding the symbol-to-class mapping is required to allow the pattern matching operation. In practical scenarios (ASCII alphabet) this table will contain 256 entries, with a maximum width of 1 byte (for heavily compressed alphabets 5-6 bits per character may suffice). This indexing table can be efficiently cached or stored in on-chip memory. The algorithm of alphabet reduction is shown in the pseudo code above. To compute the required alphabet translation tables, we use a parallel variant of the alphabet compression algorithm proposed in [4], which has O(n 2 ) time complexity. Specifically, we first construct a separate translation table for each state and then build a global alphabet translation table by progressively merging the state-specific tables from the first phase. On a 8-core processor, our implementation achieves a 4-5x speedup compared to the original single-threaded version [4]. Unfortunately, alphabet reduction becomes less effective as the size of the dataset (and of the corresponding DFA) increases. In fact, on large DFAs it is less likely for different symbols to cause transitions to the same target states. Therefore, in this study we combine alphabet compression with regular expression partitioning. In particular, we propose two new regular expression clustering methods in Section 3.5, and we compare them with the bisection-based scheme proposed

Selection of GPU Memory for Alphabet Transition Table In our implementation, each alphabet translation table consists of 256 1-byte entries, thus requiring 256 B of memory. Below, we discuss advantages and disadvantages of storing the alphabet translation tables in different GPU memories. 19 • Given its large size (from 1 to about 12 GB depending on the GPU), global memory can easily accommodate a large number of alphabet translation tables. The main disadvantage of global memory is its high access latency. • Shared memory offers low access latency at the cost of a limited capacity (from 16KB to 48KB per SM, depending on the configuration). The main limitation of shared memory is the following. Shared memory is SM-specific and has the scope of a single thread-block. If multiple thread-blocks with cumulative shared memory requirements exceeding the available capacity are mapped onto the same SM, their execution is serialized. Thus, storing a large number of alphabet translation tables in shared memory will limit the scalability in the number of packet flows. Specifically, given nAT alphabet translation tables, a shared-memory based implementation can scale up to 48KB/(256B×nAT) concurrent flows, and is more suited to pattern-sets that can be easily compiled into a small number of DFAs. • Constant memory is read-only, has a 64KB size, is shared by all the threadblocks, offers low access latency, and can be accessed in parallel to shared memory. If every thread in a half-warp requests data from the same address in constant memory, the GPU will generate only a single read request and subsequently broadcast the data to every thread. In addition, constant memory is cached, and therefore consecutive reads to the same address will not lead to any additional memory traffic. However, if the threads in a half-warp require different data, the corresponding 16 reads will be serialized. If multiple, per-DFA alphabet translation tables are used, this memory accesses serialization may impact the performance of our implementation, since different threads process different DFAs

4 Per-DFA vs. Shared Alphabet Translation Tables In general, alphabet translation tables can be either DFA-specific or shared across multiple DFAs. This design choice involves the following trade-off. Per-DFA alphabet translation tables typically result in smaller alphabets, thus reducing the amount of memory required to store the DFA state transition tables. However, as discussed above, multiple alphabet translation tables limit the flow scalability of shared-memory based implementations, and cause access serialization in constant-memory based implementations. On the other hand, sharing a single alphabet translation table across multiple DFAs generally leads to larger alphabets (it is more likely for a character to be treated differently in distinct DFAs), and thus to larger state transition tables (and memory requirements). As we will discuss in Section 3.6.3, we found the use of a single, shared alphabet translation table to be preferable.

5 Regular Expression Clustering Algorithm In this section, we propose two regular expression clustering schemes aimed to mitigate the state explosion problem [3, 17, 33, 25]. Recall that, in our implementation (Section 3.1), each thread is responsible for the traversal of one DFA, and branch and memory divergence are the main obstacles to achieving high processing throughput. Therefore, when performing regular expression partitioning, it is important to alleviate this performance degradation by limiting the number of DFAs. At the same time, the size of each DFA must be kept small, so to limit the DFA memory requirements to the 21 available GPU capacity. Since we need GPU global memory to store packets, we allocate 80% of global memory to store DFAs. In order to limit the number of DFAs encoding a particular pattern-set, we need to consider the complexity of each regular expression in that set. Combining many complex patterns in a single DFA can lead to state explosion and prohibitive memory requirements [3, 33]; on the other hand, equally distributing complex regular expressions into multiple DFAs allows limiting the size of each DFA. Smaller DFA have also the benefit of faster generation and compression (for example, alphabet compression has a time complexity which is quadratic in the number of DFA states). Below, we present two schemes to achieve this goal.