CMPT 886: Introductory Example

Nigel Medforth, Wenqiang Peng

February 6, 2016

Inverted Index Problem

When search engines query their database to determine which documents contain a particular search term, they rely on a data structure that maps each term to a set of web pages containing it. The standard approach to create this mapping is to first generate a forward index, which lists the set of unique terms in each document and then inverting it to indicate the set of documents containing a particular term.

Document	Words
1	keep,night,old,town
2	big,gown,house,old
3	big,had,house,keep,old,town
4	did,night,sleep,where

(a) Forward Index

Word	Documents
big	2,3
did	4
gown	2
had	3
house	2,3
keep	1,3
night	1,4
old	1,2,3
sleep	4,
town	1,4
where	4

(b) Inverted Index

For this project, we focus on the creation of the forward index by building upon the example in $s2^k$ (Ch. 6). Our intent is to build a data parallel hash map data structure and incorporating it into the Pablo to LLVM Compiler within icGrep.

Data Parallel Hashing using Pablo and Gather/Scatter

To perform data parallel hashing we must be able to pack multiple words into a vector and simultaneously hash them. But words can differ in length so if we gather arbitrary-lengthed

words, we'd need (1) sentinels to mark the end of each word, and (2) to repeatedly test whether we've hashed up to the sentinel of every word. Both of these issues would severely impact SIMD efficiency but we can avoid them by using the length partitioning technique discussed in s2[k]. This technique utilizes the Pablo functions:

- `Advance`(C, n): moves each 1-bit in bit stream C forward by n positions.
- `ScanThru`(C, M): moves any 1-bit in C zero or more positions to the first subsequent 0-bit in M.
- `Lookahead`(C, n): moves each 1-bit in bit stream C backwards by n positions. (Currently, this function does not exist in icGrep but was implemented in the PyParabix toolchain.)

We begin by generating the bitstreams S and E, marking the start and end positions of each word in a particular document. For example,

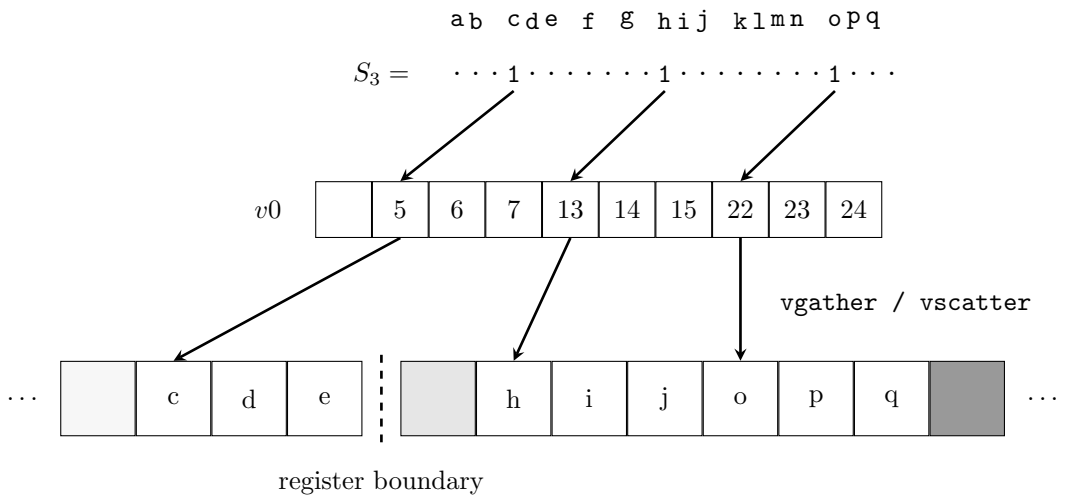
Document	`But I must explain`
$A =$ `[a-zA-Z]`	`111..1.1111.1111111..`
$T =$ `Advance`$(A, 1)$	`.111..1.1111.1111111.`
$S = A \wedge \neg T$	`1....1.1....1........`
$E =$ `ScanThru`(S, A)	`...1..1....1.......1.`

Next, we identify which start positions belongs to a word of a particular size. This will allow us to efficiently "bucket" tokens of a similar length into the same group and eliminate the need for a sentinel byte. Three approaches are discussed in s2[k] but here we focus only on the simplest one, identity length partitioning, in which every word of length k belongs to the k^{th} group. Using the bitstreams S and E, we can identify the start positions, S_i, of each word in the i^{th} group as follows:

Document	`But I must explain`
$T_1 =$ `Advance`$(S, 1)$	`.1....1.1....1.......`
$T_1 \wedge E$	`......1.............`
$S_1 =$ `Lookahead`$(T_1 \wedge E, 1)$	`.....1..............`
$T_2 =$ `Advance`$(T_1 \wedge \neg E, 1)$	`..1......1....1......`
$S_2 =$ `Lookahead`$(T_2 \wedge E, 2)$	`....................`
$T_3 =$ `Advance`$(T_2 \wedge \neg E, 1)$	`...1......1....1.....`
$S_3 =$ `Lookahead`$(T_3 \wedge E, 3)$	`1...................`
...	

From these, we can compute our sets of offsets into the `vgather` instruction. This may require scalar code (utilizing the `bitscan` intrinsic) as we have not found way to convert

the set of bit positions to indicies in parallel. Whenever the length of the group is not a multiple of the `vector width`, we can employ `vscatter` to ensure that the hashing function is strictly a SWAR operation. Similarly, to support division length and logarithmic length partitioning, both of which have groups in which multiple lengths map to the same group, either `vscatter` may be required to ensure each word is properly aligned to the appropriate boundary within the register.



Once all of the data is packed within vector register aligned memory, it can be loaded using a simple `vload` and hashing performed in parallel. Each length group will have its own hashing function designed specifically for words of that length. The exact function for each is still to be defined but will use a combination of masking (against constant bitmasks), fixed-length shifts, and SIMD arthmatic to compute a final hash code. Ideally, if the set of words is finite and known a priori, a **perfect hash** function could be generated in the compiler in which the final hash code is also the unique global identifier of the word. Otherwise, we will have to defer to scalar code as the hash code will only refer to which slot within the length-specific hash table to begin probing on.

Using `vgather` alone, we can only support parallel hashing for groups of length at most `vector width`/2. When handling larger words groups, we can use the `pext` and `pdep` instructions to extract a subset of the bits of each word (using the transposed byte stream) but the exact process for this is still to be explored.