Design and Implementation of a Database Filter for BLAST Acceleration

Panagiotis Afratis, Constantinos Galanakis, Euripides Sotiropoulos, Georgios-Grigoris Mplemenos, Grigorios Chrysos, Ioannis Papaefstathiou, Dionisos Pnevmatikatos

Department of Electronic and Computer Engineering, Technical University of Crete, Chania, GR 73100, Greece
{afratis,esot,chrysos, pnevmati}@mhl.tuc.gr

Abstract — BLAST is a very popular Computational Biology algorithm. Since it is computationally expensive it is a natural target for acceleration research, and many reconfigurable architectures have been proposed offering significant improvements.

In this paper we approach the same problem with a different approach: we propose a BLAST algorithm preprocessor that efficiently identifies the portions of the database that must be processed by the full algorithm in order to find the complete set of desired results. We show that this preprocessing is feasible and quick, and requires minimal FPGA resources, while achieving a significant reduction in the size of the database that needs to be processed by BLAST. We also determine the parameters under which prefiltering is guaranteed to identify the same set of solutions as the original NCBI software.

We model our preprocessor in VHDL and implement it in reconfigurable architecture. To evaluate the performance, we use a large set of datasets and compare against the original (NCBI) software. Prefiltering is able to determine that between 80 and 99.9% of the database will not produce matches and can be safely ignored. Processing only the remaining portions using software such as NCBI-BLAST improves the system performance (reduces execution time) by 3 to 15 times. Since our prefiltering technique is generic, it can be combined with any other software or reconfigurable acceleration technique.

I. INTRODUCTION

BLAST is considered as the most popular and widely used algorithm of Computational Biology. It is used for searching large genetic databases in order to find areas of high similarity (matches) between the database and an input query or it can be used as a part of other applications as bioinformatics algorithms.

Since the algorithm is inherently computational intensive, it has been a challenge during the last few years for several research groups to build FPGA-based systems to boost this algorithm performance. RC-BLAST [1] was the first effort to implement this algorithm using FPGA. Boston University [2], FPGA/FLASH project [3], TUC BLAST [4], Mercury BLAST [5-6], and BEE BLAST [7] follow offering significant results. These projects share common characteristics but each one differs giving a new architecture with its points of strength. In this paper we propose a Prefiltering architecture that does not directly implement the BLAST algorithm, but exploits its characteristics to tag portions of the database that for the particular query are deemed “interesting”. This “interesting” portion of the database is the portion that we should explore in detail since it is likely (but not certain) that there we will find areas of high similarity (BLAST matches). A threshold value can determine the selectivity of the filter. After the tagging, any BLAST algorithm implementation (software or hardware) can be used on the identified database subset to determine the full results. We compare the results of our prefiltering against NCBI BLAST results with several datasets and queries, and we show that for a threshold value of 2 our solution is lossless (i.e. it produces all the results BLAST reports). We also find that for very large queries pre-filtering is not effective, and we address these cases by partitioning the query in smaller pieces and processing them in parallel to achieve the correct overall results and better filtering behavior. Search space reduction is 80% in the worst case, and up to 99.9% for some of our experiments. Our pre-filtering approach is versatile and can be used in combination with any other hardware or software BLAST implementation.

We show the versatility and efficiency of our approach using a parallel BLASTn algorithm that has been implemented on M.PL.EM (Multiprocessor Platform for Embedded systems) [10]. M.PL.EM is an FPGA-based multiprocessor consisting of a large number of Xilinx MicroBlaze soft-cores together with a hierarchical interconnection scheme and a sophisticated memory subsystem. We use this platform to post-process the results of our prefiltering system in order to speed up the execution of BLAST-n algorithm on the M.PL.EM system.

In the next section we discuss the BLAST behaviour and characteristics that lead us to our prefiltering approach. Section 3 expands the description and evaluates BLAST prefiltering potential. Section 4 and 5 describe the architecture of the proposed system and present performance measurements. Finally, in section 6 conclusions and future work are discussed.

II. PREFILTERING FOR THE BLAST ALGORITHM

The BLAST algorithm takes as input a query and a database of genetic data. Its operation consists of three steps: (1) the first step of the algorithm is a preprocessing that breaks the query into w-mers, that is smaller parts of 12 character wide substrings (a character is a 2 bit value), (2) in the second step the database is searched in order to find an exact match (hit) between any part of the database and any of the w-mers,
and (3) the third step of BLAST is the expansion, which is executed only after an exact match has been found in step two for a w-mer. In this step the algorithm uses the matched w-mer as a starting point and tries to extend it in either way in order to determine the maximum similarity width according to a scoring scheme.

Our main observation that leads to our prefiltering approach is that the BLAST algorithm finds and reports matches in the areas of high similarity between database and query, i.e. in areas where the third step of the algorithm is active and successfully processes a large number of extensions. These areas with high activity in the third step are also areas where the second step of the algorithm produces multiple hits between different w-mers of the query and different offsets of the database.

![Hit rate distribution](image)

**Fig. 1** Hit rate distribution for a window of 100 characters over the streaming database input. The two top circled areas are “of interest” and correspond to actual BLAST matches. A Threshold set to 5 is optimal and identifies all these areas. Thresholds less than 5 will produce more candidate regions without identifying more hits (drawn for Threshold=3), while thresholds greater than 5 will miss (some of) the hits reported by BLAST.

Our prefiltering formulates this observation: if within a particular portion (window) of the database the high hit rate between the database and the set of w-mers exceeds a Threshold, then there is high probability that this area will result to a high similarity (extensions) between database and query and we need to run the full BLAST algorithm. Portions of the database where the hit rate does not reach the Threshold are not processed further. Note that when the hit-rate does exceed the threshold, there is no guarantee that we will actually find a match in this window: multiple hits may be produced from different w-mers in an incorrect order or distance, so they may not correspond to actual extensions. Our approach is depicted in Figure 1 that plots the hit rate distribution for a window of 100 characters that slides over a streaming database input. According to the threshold, different portions of the database are considered interesting. The higher the threshold, the more selective the filtering, but if we exceed a certain threshold value we will miss (some of) the reported BLAST results.

Rate-based BLAST prefiltering approach is feasible and efficient since it involves a basic search of the query w-mers against the streaming database input. As we will show later, pre-filtering requires very modest FPGA resources to be implemented, making it very cost effective as an accelerator.

### III. BLAST Prefiltering Potential

To analyze the potential of BLAST prefiltering, we built a set of software tools that implement BLAST searching. We ran these tools using several data sets that were provided from NCBI site, and we compared the results against those of the original NCBI software [8]. In our experiments we compared parts of Homo Sapiens (Human) (queries) against Chimpanzee’s chromosomes (Pan Troglodytes) genome (database). The data exhibit a high degree of similarity which leads to high hit rate at the second step of the BLAST algorithm. We also used the BioPerf benchmark for BLASTn.

#### A. Prefiltering Window Size

First, we investigate the effect of the window size, i.e. the width of the database region in which we measure the hit rate. Figure 2 plots “Space” (i.e. the resulting percentage of the database that we need to process after prefiltering) versus window size: small values are better since they correspond to less input to the full BLAST processing. Since the query size may vary greatly, we express the window size as a percentage of the query length, ranging from 10% up to 100%. Intuitively, larger window sizes will produce more hits shifting the hit rate upwards. The results in Figure 2 lead to two conclusions. First, regarding window size, space is either unaffected or increases as the window size increases; hence a small window is both more effective and sufficient to capture the necessary information. Second, the effectiveness of pre-filtering varies greatly: we find cases where the results are excellent (space in the range of 3% or less of the database), while totally ineffective in other cases (chromosomes 12 and 13) with space 100%, i.e. the entire database is candidate for match. We address this limitation in section III.D.

![Database Space (%) vs. window Size](image)

**Fig. 2** Database Space (%) vs. window Size.

#### B. Prefiltering Threshold

The other main prefiltering parameter is the threshold. Figure 3 plots the database space versus a threshold that ranges between two and five. We see that as threshold increases there is a decrease in space, even for some of the “difficult” cases (chromosome 12) identified in the previous paragraph. However, the results for other queries, such as
chromosome 13, are insensitive to increasing the threshold. Note that the choice of the threshold value is not straightforward. Setting the threshold too low results in larger database space that needs to be processed. Setting the threshold too high we risk ignoring portions of the database that will produce actual hits. In the rest of this paper we use a threshold value of 2 based on the following observation: for the BLAST algorithm to begin the extension process we need at least one match. Since there will be at least one extension (otherwise the BLAST extension process stops), we will find another hit for a w-mer overlapping with the first. We tested all our results for all our runs and verified that indeed this threshold identifies all the reported NCBI BLAST results. To safely use larger threshold values we need to further understand the biological significance on the reported results. We believe that setting larger threshold values may omit only the least significant BLAST results while still report the high ranked ones.

C. Sensitivity on Query Size

To understand the behavior of the “difficult” cases such as of the chromosome 12 and 13 queries, we analyzed our results and observed that all these cases correspond to very long queries in the order of many thousand characters. In Figure 4 we plot the effect of the query size on the resulting database space that must be searched for the queries that are not amenable to prefiltering. To produce small queries we use a prefix of the original query at a particular size. The trend in Figure 4 is very clear: large queries are not amenable to prefiltering, while small queries show great potential. A possible explanation for this behavior is that a large query contains more distinct w-mers than a smaller one, so the probability of finding multiple hits between the database and any w-mer is large. Prefiltering works very well for queries a few hundred characters long, and offers no improvement for queries longer than 5 thousand characters.

D. Partitioned Queries

The results from Figure 4 made clear that long queries, although very useful for biologists, cannot be handled effectively by prefiltering. However, the same results offer the solution to the problem: if the query is partitioned in smaller pieces and is processed in parallel, we may achieve operation in the effective prefiltering region. Figure 5 evaluates the partitioning potential. Starting with the original query size, we subdivide it to pieces of one thousand, 500, 250 characters and so on, evaluating the resulting database space that we need to search. As indicated from Figure 4, as the query size becomes smaller, the effectiveness of prefiltering increases. The best results are achieved for small sub-queries less than 250 characters, and for all the difficult queries pre-filtering achieves a 5-fold decrease in the space that needs to be explored (space = 20% of the database). More important is the correlation of query and prefiltering potential: given the database and the query, we can determine the effectiveness of prefiltering, and the extent of required partitioning.

IV. PREBLAST IMPLEMENTATION

A. Implementation Issues

All the previous described properties are based on the number of the hits that are produced on the second step of the algorithm. In order to find hits, comparisons should be performed between every w-mer and the complete database. These comparisons are 24 bit-wide (12 characters x 2 bits/character) and their number is almost equal to the size of query: the number of w-mers is equal to \((query \text{ length } - w\text{-mer length} + 1)\). For a 1,000 character query 989 w-mers are produced and need to be compared to the database input at every location. There are several implementations proposed for this problem.

1) The comparisons can be performed in parallel using for example 989 distinct comparators. This method consumes a significant amount of reconfigurable resources, and it is not scalable to larger query sizes.

2) Another approach is to use a Content Access Memory (CAM) which will have to be 24 bits-wide due to w-mer size,
and should have many thousands of entries to support larger queries, making it very expensive in terms of area.

3) A memory cache-like scheme can also be used. Using memories has the advantage that the size of the designed hardware is proportional to w-mer size which is constant and not to query size which varies. However, a single memory cannot be implemented due to its size (24 bits address) that can not fit to any reconfigurable device.

Due to hardware implementation limitations, an alternative method is proposed that uses a Bloom Filter [9] to determine the occurrence of a w-mer of a query. By the properties of Bloom filters, this approach can produce false positives, hence we count probable hits instead of actual hits. We attempted to identify the optimum number of the hash functions and the optimum depth of the filter memories while taking into account the implementation idiosyncrasies of the Xilinx FPGAs, and after thorough experimental research we concluded that 4 distinct prime polynomials used as hash functions to address 4 filter memories with address 14 bits was a good implementation trade-off. The 4 hash functions reduce the 24-bit wide w-mers to four 14-bit wide addresses, which are used to address 4 independent, 1-bit wide memories. These memories are initialized by setting to 1 all the locations identified by the hashing of all the query w-mers. On database lookup, a (probable) hit is determined when all the addressed memory locations are set to 1. Those matches are called probable hits because such a match might not necessarily produce a hit in the original second step of BLAST. However, in Bloom filters negatives are always true, so if a match does not exist, it can never be reported falsely. The structure of the Bloom filter is illustrated at Figure 6. Hence a simple lookup in the memory and an AND gate (not shown in Figure 6) identify if the w-mer portion is a sub-match with some w-mer of the query.

Since Virtex-5 BRAM blocks store 32kbits, are dual ported, and in their 32kx1 configuration need 15 bits for addressing, we combined two independent filter memories into a single BRAM block. In this way, the Bloom Filter can be efficiently implemented using just two embedded BRAM blocks.

**B. PreBLAST Architecture**

The implemented architecture, shown in Figure 7, takes as input the database stream. At every clock cycle a new character (2bits for BLASTn) is inserted in the shift register generating a new word of the database that has to be examined against all the w-mers. This 24-bit word is processed through the Bloom filters with the same hash functions that have been used to initialize the BRAMs four new 15-bit words are produced. The hashed values are used to address the four lookup tables and if all have a ‘1’ stored at these positions, a probable hit is reported.

Figure 8 shows how the probable hits are counted for a certain window size. At every clock cycle the output of the probable hit is inserted in a shift register with length of window size. If a ‘1’ is inserted then the Up/Down counter counts up and when an ‘1’ is shifted out of the register the Up/Down counter counts down. With this simple design the Up/Down counter has always the number of the possible hits for the certain time window.

![Fig. 6 Example of BRAMs preloading](image)

![Fig. 8 Control path of the designed system](image)
In this design there is also a position counter which counts the number of the characters that have been processed which is translated to the position of the database which is processed at the certain time. If the value of the Up/Down counter exceeds the predefined threshold then the position of database that this happens is stored in a memory. When the value of the Up/Down counter decreases under the threshold then the position of database is stored again. Consequently every pair of the stored values in the memory is the tagged part of the database.

C. PreBLAST combination with M.PL.EM

In order to show the effectiveness of prefiltering we combined it with the Multiprocessor Platform for Embedded systems (M.PL.EM). We modified the way the M.PL.EM processor reads the database from its memory. Instead of initializing the memory controller and get the data in the row the memory controller starts from the points where the output memory of the PreBLAST filter shows, up to the points that the “interesting” parts of the database ends. The M.PL.EM platform consists of MicroBlaze embedded processors which are not able to run the original NCBI software. For that reason a new software version of BLASTn algorithm was implemented and several experiments where tested. To verify the correctness, we run tests on a fully post-place and route simulation with up to 4 parallel MicroBlaze processors.

V. PERFORMANCE MEASUREMENTS

PreBLAST design was synthesized, placed, routed and post P&R simulated for the Virtex 5 family device XC5VLX330T, as it is the one with the largest number of BRAMs. Table 1 shows the allocation of the resources for two different designs, for one double preprocessor and for 81 double parallel preprocessors working in a single chip. The design is bounded up to 162 parallel preprocessors due to the total number of available BRAMs. On the other hand very few LUTs are used.

<table>
<thead>
<tr>
<th>Database</th>
<th>1000</th>
<th>500</th>
<th>250</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChrX</td>
<td>3,298</td>
<td>4,314</td>
<td>4,190</td>
</tr>
<tr>
<td>Chr12</td>
<td>1,958</td>
<td>3,556</td>
<td>2,653</td>
</tr>
<tr>
<td>Chr13</td>
<td>1,006</td>
<td>1,562</td>
<td>1,604</td>
</tr>
<tr>
<td>Chr14</td>
<td>1,324</td>
<td>1,850</td>
<td>2,031</td>
</tr>
<tr>
<td>Chr17</td>
<td>1,342</td>
<td>1,670</td>
<td>1,898</td>
</tr>
</tbody>
</table>

Table 2: Projected system speedups using TUC PreBLAST combined with an 3GHz Intel Core 2 DUO

Table 2 shows the potential improvement of combining prefiltering with the original BLAST software. Depending on the particular query and for the smaller query sizes, the improvement ranges from 2 to almost 5 times, verifying that prefiltering is effective. These results indicate that using the entire FPGA resources to implement multiple prefiltering engines is beneficial, while having just two units in most cases achieves performance within 85% of what can be achieved using the entire FPGA device.

VI. CONCLUSIONS – FUTURE WORK

In this paper we exploit a property of the BLAST algorithm to collect simple measurements and filter the database that needs to be considered for a query. We show that BLAST prefiltering offers significant search space reduction that ranges from a factor of 5 for long queries, up to 3 orders of magnitude for short queries, and a significant system-level acceleration of 2 to 5 times. Prefiltering is very compact in terms of logic, and requires few memory blocks and low bandwidth (less than 1 Gbps) posing no real I/O problems. The filtered database can be subsequently processed with any existing software or hardware BLAST processing system in a streaming fashion.

In this work we have considered mainly a small window size of 10% of the query, and a threshold of 2. We are investigating ways to dynamically determine these parameter using sampling methods to achieve better filtering without losing accuracy in the results.
ACKNOWLEDGMENTS

This work was funded in part by the Greek Ministry of Education GSRT project “Irakleitos: Research Scholarships for TUC”, Program. TUC, sub-program #10, “Structures for Reconfigurable Computing”.

REFERENCES


