Reconfigurable Hardware Acceleration of Canonical Graph Labelling

David B. Thomas[1], Wayne Luk[1], and Michael Stumpf[2]

[1] Department of Computing, Imperial College London
{dt10,wl}@doc.ic.ac.uk
[2] Centre for Bioinformatics, Imperial College London
m.stumpf@imperial.ac.uk

Abstract. Many important algorithms in computational biology and related subjects rely on the ability to extract and to identify sub-graphs of larger graphs; an example is to find common functional structures within Protein Interaction Networks. However, the increasing size of both the graphs to be searched and the target sub-graphs requires the use of large numbers of parallel conventional CPUs. This paper proposes an architecture to allow acceleration of sub-graph identification through reconfigurable hardware, using a canonical graph labelling algorithm. A practical implementation of the canonical labelling algorithm in the Virtex-4 reconfigurable architecture is presented, examining the scaling of resource usage and speed with changing algorithm parameters and input data-sets. The hardware labelling unit is over 100 times faster than a quad Opteron 2.2GHz for graphs with few vertex invariants, and at least 10 times faster for graphs that are easier to label.

1 Introduction

Computational biology is a key part of modern research into biological processes and drug development, allowing for virtual lab-work, large-scale biological simulation, and computer driven search processes. This reduces the number of traditional wet experiments required, as well as broadening the scope of experiments that can be considered. One of the enabling factors in computational biology has been the rapid increase in software processing speed over previous decades.

Now that single processor speed-increases are beginning to slow, it is necessary to consider technologies such as multi-processor computers and multi-node clusters. Unfortunately these solutions are large, expensive, and require large amounts of power, as well as presenting problems of application scaling. Another possibility is to use custom hardware, such as FPGAs, to provide acceleration, making each node more powerful, and so reducing the cost, power, and degree of scaling needed per cluster.

This paper presents an architecture, implementation and evaluation of matrix based canonical labelling in hardware. This is a key building-block for many graph based bioinformatics algorithms, and a computational bottleneck in software implementations. The key contributions of this paper are:

P.C. Diniz et al. (Eds.): ARC 2007, LNCS 4419, pp. 302–313, 2007.
© Springer-Verlag Berlin Heidelberg 2007

– an architecture for the implementation of canonical labelling in hardware;
– resource usage and performance evaluation in the Virtex-4 platform;
– a comparison of the performance of software and hardware canonical labelling units, with an xc4vlx60 showing a 100 times speed-up over a quad Opteron software implementation.

2 Motivation

Many biological and chemical processes are represented using graphs. Protein Interaction Networks (PINs) are one example, which represent biological processes using protein-protein interactions as paths between proteins [1]. These interactions are recovered experimentally, and the functions of different interactions within the graph are initially unknown. One approach for extracting information from these PINs is to find recurring motifs within the network [2].

Frequent Sub-Graph Mining [3] is one technique for looking for these motifs, by enumerating all sub-graphs less than a certain size. After all sub-graphs within the biological network have been counted, the frequency of each sub-graph is compared with the probability of its random occurrence. Any graphs that occur more often than chance would predict may indicate a unit of functionality, or building block, within the biological network. Attempting to find such graphs through human inspection would be almost impossible, and it is only through the use of large amounts of computation power that such techniques are possible.

One problem that occurs when searching for motifs is that a given sub-graph may be counted more than once, as it may be encountered at a number of different points during the search, and the in-memory representation of the sub-graph may not appear the same as when it was encountered before. One way in which the accuracy of searches can be improved is to use canonical labelling to uniquely identify the structure of each node. However, this represents a considerable cost in software, as billions of sub-graphs may be encountered during each search. The approach proposed in this paper is to use software to generate candidate sub-graphs, then to perform the canonical labelling in hardware.

3 Canonical Labelling Algorithm

Canonical labelling is the process of attaching a unique label to graphs, such that any two graphs with the same structure will receive the same label, and any two graphs with different structures will not receive the same label. It has many applications, such as in algorithms for finding graph-isomorphisms [4], and for chemistry and biological applications where repetitions of structures such as molecules and biological structures need to be identified [2].

In this paper the graphs will be considered to be undirected, vertex-labelled, and edge-labelled. The finite set of all vertex and edge labels are L_V^+ and L_E^+ respectively. Each graph is defined as a tuple $G = (V, E, \varphi_V, \varphi_E)$, where V is a finite set of vertices, and $E \subset V \times V$ is a finite set of edges. Graphs must be connected, i.e. for each graph there is a path between every vertex in V by

traversing one or more edges in E. The two total functions $\varphi_V : V \mapsto L_V^+$ and $\varphi_E : E \mapsto L_E^+$ assign labels to each vertex and node.

This definition reflects the most complex type of graph commonly encountered in applications, for example where distinct vertices within a molecular graph can represent different instances of the same atom (i.e. have the same vertex label), or where distinct edges between vertices represent one of a number of types of bonding. If $|L_V^+| = 1$ then there is only one edge label, so the graph edges are effectively unlabelled, and similarly if $|L_E^+| = 1$ then the edges are unlabelled. The mappings can also be made injective, in which case the label will uniquely identify each edge or node within the graph.

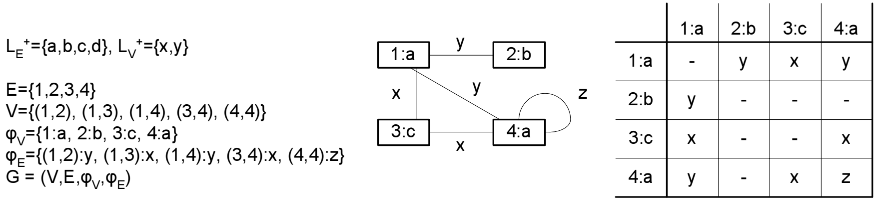
Canonical labelling is a function $\varphi_G : G \mapsto L_G^+$ that takes a graph G, and returns a label L_G for that graph, such that $\varphi_G(G_1) = \varphi_G(G_2)$ if and only if there exists an isomorphism between G_1 and G_2 [5]. For example, if G describes a protein interaction network, where L_V^+ identifies proteins, and L_E^+ describes different types of interactions between proteins, then the canonical label of G uniquely identifies that interaction structure. If the same canonical label is then observed in a different graph, then the two protein interaction networks must be the same. If the same interaction graph is observed in many different places then it is possible the structure is important in its own right, or forms a key building-block for larger structures.

One method for calculating the canonical label is use graph operations, by constructing sets of sub-graphs with certain characteristics, or partitioning the graph in some way. NAUTY [6] is a software package that uses this approach, and is able to label large graphs with many thousands of vertices.

This approach is effective for large graphs, as the asymptotic behaviour of the algorithms is good. However, there is a significant overhead involved in maintaining and manipulating the data structures, so for small graphs the cost of these algorithms is high. In applications searching for motifs the sub-graphs are generally small, only requiring graphs of size less than 24 to be labelled. However, the number of examined sub-graphs may be extremely high, thus the canonical labelling algorithm must be as fast as possible for small graphs.

A different way of calculating the canonical label is to first convert the graph into a matrix form, then to use matrix operations to calculate the canonical label. Figure 1 shows a matrix represented in three different ways, with the set based description on the left, the familiar graphical representation in the middle, and a matrix representation on the right. To allow easy comparison between the representations, the vertexes have been given a unique id number, so node $1 : a$ has the id 1 and label a. Note that the matrix representation is symmetric, as the graph is undirected.

Once the graph is in matrix form, it is possible to define a label for each matrix. The top left of Fig. 2 shows the example matrix, and highlighted in grey are the elements used to define the label. A textual version of the label is shown below, consisting of first the vertex labels, followed by the lower triangle of the edge labels. The canonical label will be defined using the lexicographical minimum



	1:a	2:b	3:c	4:a
1:a	-	y	x	y
2:b	y	-	-	-
3:c	x	-	-	x
4:a	y	-	x	z

Fig. 1. Representation of matrix as a set, as a diagram, and in matrix form

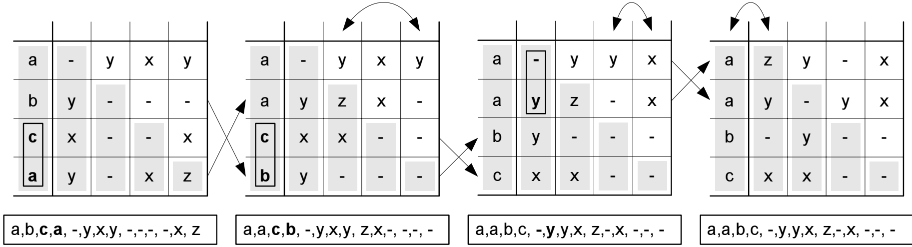


Fig. 2. Matrix method for canonical labelling

as an ordering. The canonical label is the label that comes first amongst all structure preserving orderings of the graph matrix.

Looking at the label for the top-left matrix it is clearly not the minimum label, as a occurs after c. A lower label can be reach by swapping these two vertexes, but to maintain the interconnection structure it is also necessary to swap the ordering of the edge matrix, by symmetrically swapping the rows and columns containing a and c. Note that after swapping both rows and columns the z from the lower right corner has moved diagonally within the matrix. Another swap moves b and c into the correct order, shown in the bottom right of the figure. Now all the vertex labels are in the correct order, but because there are two a labels it is still possible to achieve a lower label. The two symbols to the right of the a vertices are $-$ and y, and here y is considered to be less than $-$. Swapping these two rows will not affect the relative ordering of the preceding part of the label, as the two a elements can appear in either order. After this swap the matrix label in now in its canonical form.

The canonical label can be determined by using a brute-force row swapping algorithm, by using row swaps to examine every permutation of the matrix, and comparing the label of the new matrix against the best matrix after each swap. This will eventually find the correct label, but will take $n!$ swap-compares, so even for relatively small matrices this becomes impractical.

A technique known as vertex invariants can be used to limit the number of combinations that are needed. A vertex invariant is a property that can be assigned to a vertex no matter what order the graph is in. A good example of a vertex invariant is the vertex degree (the number of edges that are incident

on the vertex). This information can be used to partition the column of vertices into a number of sorted sub-ranges. Then for any sub-range where all the vertex invariants are equal the brute-force computation can be used.

Another type of vertex invariant is the label of the vertex, which was used implicitly in the example of Fig. 2. The two invariants can be combined using some deterministic function to create a single hybrid vertex invariant: the more distinct invariant labels that a vertex can be assigned, the more likely it is that no other vertex will have that invariant label. Ideally all the vertex invariants will be different, reducing the canonical labelling process to that of sorting the list of invariants. If the corresponding set of swaps is applied to the edge matrix then the resulting matrix will be in canonical form.

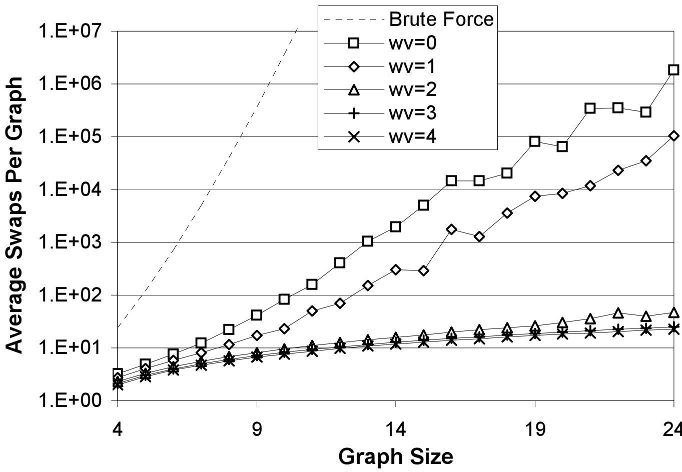


Fig. 3. Average number of swaps required to label random graphs. Brute force is the number of swaps when vertex invariants are not used, $w_v = 0$ uses vertex degrees only, and $w_v = 1, 2, 3$ use vertex degrees and a vertex label randomly chosen from amongst 2, 4 or 8 labels respectively.

Figure 3 examines the computational cost of canonical labelling when applied to random graphs. The random graphs are constructed by creating a set of n nodes, then adding edges between randomly selected nodes until the graph is connected (i.e. there is a path from each node to every other node). Each vertex is also assigned a random w_v bit vertex label, i.e. $2^{w_v} = |L_V^+|$.

The dashed line shows the number of swap-compares for the brute force case, and due to the factorial growth this quickly becomes infeasible to apply. When $w_v = 0$ the only thing that distinguishes between vertices is their degree, and there are likely to many nodes with the same number of degree. This means many ranges must be brute-forced , so the number of swap-compares is high, growing approximately exponentially and on average requiring 10^6 swap-compares for a 24 vertex matrix. In the case of $w_v = 1$ the situation is improved, as some

of the ranges of similar degree are now split by the random 1 bit vertex label. However, the growth is still exponential. For larger widths the number of sub-ranges that need to be brute-forced is low on average, so the number of swap compares reduces to an $n \log n$ trend, dominated by the process of sorting the list of vertex invariants.

4 Hardware Implementation

The matrix based canonical labelling algorithm described in the previous section is very simple, but requires irregular memory reads and writes when implemented in software. In this section an architecture for implementing the algorithm in hardware is presented, taking maximum advantage of the parallelism available in the algorithm.

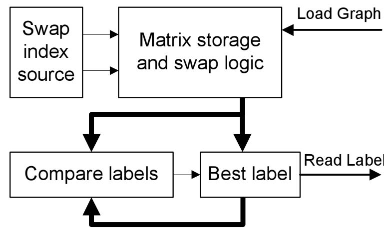


Fig. 4. High level architecture of a canonical labelling unit

Figure 4 shows a high level view of a hardware canonical labelling unit. At the centre is the matrix storage and swap logic, which is responsible both for maintaining the local copy of the graph matrix, using either registers or RAMs, and for swapping rows. Graphs are first loaded into the matrix storage, then the swap index source produces pairs of rows to be swapped. After each swap the label of the current graph is extracted and compared with that of the best known graph. If the current graph's label is lower then the old label is replaced, and once the search space has been explored the minimum label is retrieved.

Within this architecture there is considerable freedom in terms of implementation. For example, the matrix storage could be implemented using registers, distributed RAMs, or block RAMs, and the swap process might take one or many cycles. The number of cycles per swap-compare might be different from turn to turn, for example if the comparison unit searched sequentially for the first element of the label that matched. In such cases the swap and compare units might be decoupled using a FIFO.

In this paper the swap-compare unit is designed to be a single cycle process, so on every cycle a full symmetric matrix swap occurs, and the minimum of the old and new label is captured. There are two clear bottlenecks in this approach: routing data from one position within the matrix to another, and performing

the label comparison within a single cycle. It is possible to buffer the label with registers between the swap-compare unit and the comparison, so the two can be considered independently.

Considering the comparison unit first, the length of each label can be determined from the widths of the vertex and edge labels, and the size of the graph. Let $n = |V|$, the size of the graph, and $w_v = \lceil log_2(|L_E^+|) \rceil$, the number of bits per vertex label (allowing that w_v might be zero). The number of bits per edge label is $w_e = \lceil log_2(1 + |L_E^+|) \rceil$, which allows enough bits to hold all edge labels, plus one more state to represent the lack of an edge. At a minimum $w_e = 1$, which allows one label for all edges that exist, and another for edges that don't exist.

The size of each label is $w_v n + w_e n(n+1)/2$, and in current FPGA architectures the comparison performance will be limited by the linear carry chain propagation. Thus the cycle time of the comparison unit will vary quadratically with the size of the graph. Increasing the size of w_e will also add a large amount of delay. However, w_v will have less affect, which is important if large vertex invariants are to be used.

The matrix storage unit must have at least $w_v n + w_e n^2$ bits of storage to hold the current state of the graph. Because swap-compares occur in a single cycle, no other storage is needed. However, implementing the parallel element routing to allow the swaps will require significant logic resources. Figures 6 and 5 show one way in which this can be achieved. The array consists of two types of cells: diagonal cells, which provide a special case for the behaviour of diagonal elements, and standard cells, which implement the general row and column swapping logic.

Each data bus within the array is composed of an upper and lower bus, which provide the two data-paths required for swapping to take place within a cycle. Figure 5 shows these data buses as thick lines, except for in the top row of the row bus, where the two halves are shown explicitly. During a cycle where indexes a and b (where $a < b$) are to be swapped, the standard cells with index $(i = a, 1 \leq j \leq n)$ will drive the lower half of row bus i and read from the upper half, while the standard cells with index $(i = b, 1 \leq j \leq n)$ will drive the upper half of row bus i and read from the lower half. Similarly the cells in columns a and b drive the lower and upper halves of the column bus.

Diagonal elements require special handling. First, because the matrix is symmetric, element $(a, b) = (b, a)$, so it is not necessary to reflect these elements across the matrix diagonal. It is sufficient to simply disable the two standard cells corresponding to these positions when a cell is in the low row and the high column, or the high row and the low column. Second, in symmetric row swaps the elements running along the matrix diagonal (representing self-connected vertices) are completely independent of the rest of the matrix. Thus the diagonal elements use an independent diagonal bus, with a pair of upper and lower buses as before to allow swaps in a single cycle.

Figure 6 shows the internal arrangement of the standard and diagonal cells. Standard cells are controlled using inputs $colEnHigh$, $colEnLow$, $rowEnHigh$,

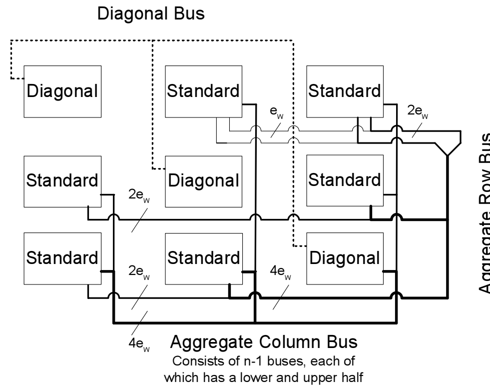


Fig. 5. Data buses between the cells in a swap-compare unit. Note that all data-paths actually contain two buses, as shown for the first row of the row bus, corresponding to the two indices to be swapped.

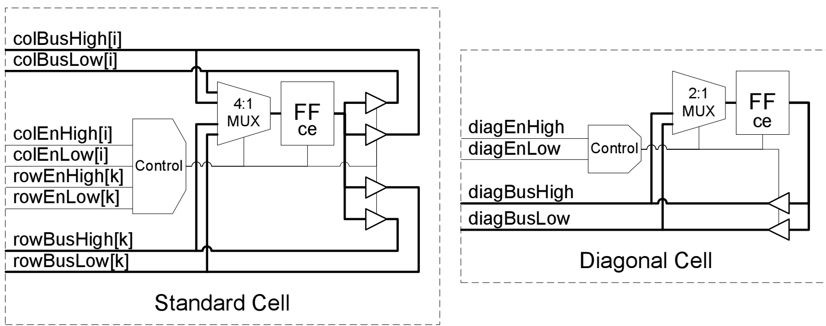


Fig. 6. Edge storage cell used in a swap-compare unit. The four control signals colEnHigh..rowEnLow determine whether each standard cell is in one of the rows or columns to be swapped, and if so which of the upper and lower buses to read and write to. The diagonal cell only needs to know whether it is the upper or lower edge of the swap, and reads and writes to the corresponding upper and lower buses.

and *rowEnLow*, which identify whether the cell is in the upper or lower row or column. It is impossible for either $colEnHigh \wedge rowEnHigh$ or $rowEnHigh \wedge rowEnLow$ to occur in a standard cell, as this could only occur in a diagonal location. If only one of the control inputs is true then this identifies that the cell lies only within a row or column. The register's clock-enable is asserted, and the asserted control input determines which of the input buses to select, and which of the output buses to drive. If none of the control inputs are true, or if more than one is true (indicating the cell is on the anti-diagonal), then the register's clock-enable is disabled. The diagonal cell is very similar, but only has one bus.

This architecture was implemented in Handel-C, targeting the Virtex-4 architecture (although the description is platform independent), synthesized using Xilinx ISE 8.1 with default effort and optimisation settings. The code directly reflects the structure outlined above, using two modules containing the swap and comparison modules, with the swap module implemented as a grid of modules representing standard and diagonal cells. The buses were implemented using the *signal* language feature of Handel-C, which act as tri-state buses, but are implemented in hardware as multiplexors. The implementation also contains a graph load facility, allowing a new graph state to be loaded column by column in n cycles, and a label extraction facility, also taking n cycles. The code is also parametrised to allow different widths of w_e and w_v (including $w_v = 0$).

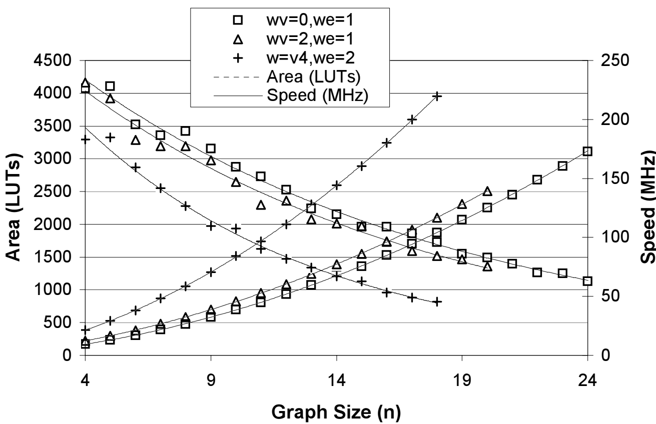


Fig. 7. LUT usage and speed for different combinations of vertex and edge label width

Figure 7 shows the resulting resource usage in LUTs, and speed in MHz for three different combinations of vertex and edge label width. The results are shown for increasing graph size, varying between 4 and 24. The results for the larger label widths are truncated due to tool-chain limitations. Quadratic curves are fitted through the resource usage, showing that as expected LUT usage is almost exactly quadratic with increases in graph size. Exponential curves are fitted to the frequency results, and provide a reasonable fit given the level of noise from the tool-chain.

Figure 8 focusses on the case where $s_v = 0$ and $w_e = 1$, i.e. an unlabelled graph. This represents the hardest case to label, as there are few vertex invariants, and also occurs in many applications, so it is of particular interest. Resource usage is now broken down into registers, LUTs and slices, and again the quadratic fit is near perfect. The quality of the quadratic and exponential model as predictors of resource usage and speed make it possible to get a good estimate for the performance of a hardware implementation without necessarily synthesising it.

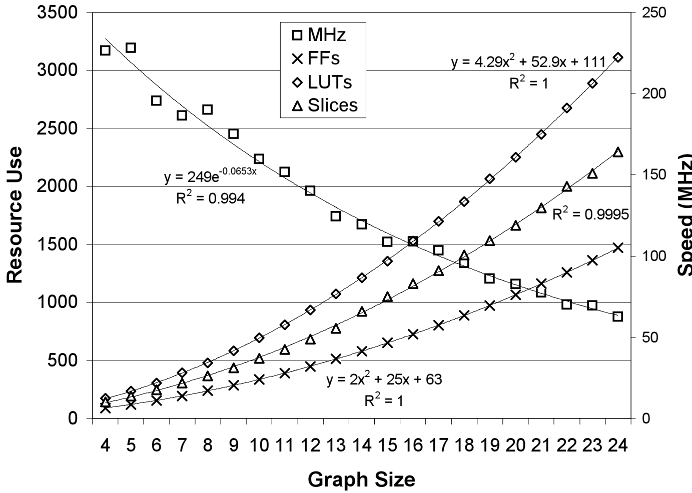


Fig. 8. Resource usage and frequency for an unlabelled graph swap unit (i.e. wv=0,el=1). Quadratic curves are fitted to the resources used, and an exponential curve to the clock frequency, showing an extremely good fit in all cases.

5 Evaluation

In this section we compare the performance of the hardware graph labelling architecture with a software implementation. The software was implemented in C++, and is templatised on the size of the graph, allowing as much constant propagation as possible. The code was compiled using g++ 3.4.5 with -O3, then executed on all four processors of a quad Opteron 2.2GHz machine. Each datapoint reflects measurements over a run of at least 10 seconds.

Figure 9 compares the performance of software and hardware purely in terms of the raw number of swap-compare steps per second. The software provides around 27MSteps/sec for the smallest graph size, decreasing down to 3.5MSteps/sec for 24 vertex graphs. By comparison a single graph labelling hardware instance starts at 226MSteps/sec, decreasing down to 61MSteps/sec, providing about a 10x speed-up over the quad Opteron in general.

The graph also estimates the performance if an entire xc4vlx60 device is devoted to canonical labelling. This figure was obtained by estimating the number of replicated instances that could be supported using slice counts, then scaling the speed by the number of instances. Although this ignores many practical problems, such as the amount of hardware dedicated to IO, and the drop in clock rate when targeting a congested chip, it does give some idea of the maximum speed-up possible. For small graphs, where the degree of replication is high, the estimated xc4vlx60 raw performance is around 1000 times that of software, and remains over 100 times faster over all the graph sizes tested.

Figure 10 provides a more practical measurement of performance, as it was derived from cycle accurate simulation of the algorithm using random graphs.

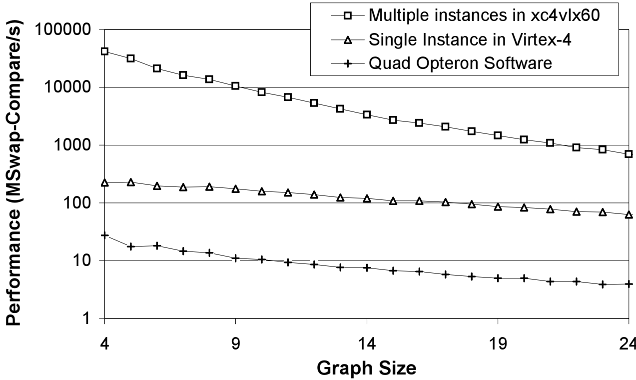


Fig. 9. A comparison of the raw swap-compare performance between the Virtex-4 hardware and a Quad Opteron software implementation. Hardware performance is shown both for a single graph labelling unit, and for an xc4vlx60 device filled with the maximum number of units that will fit.

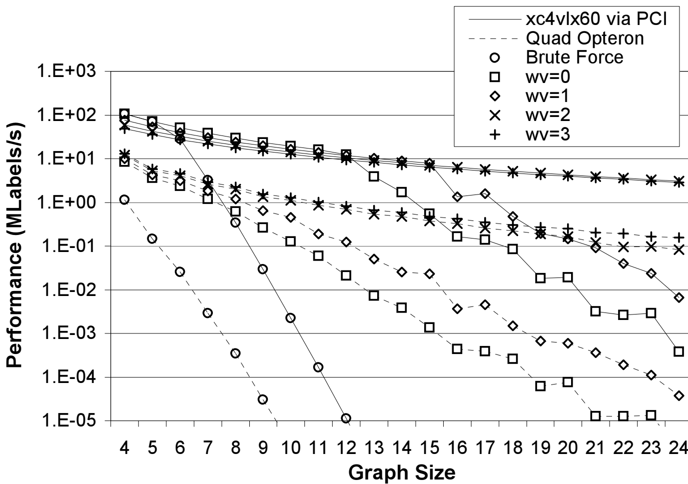


Fig. 10. Practical performance when labelling random graphs for a Virtex-4 xc4vlx60 device compared with that of a Quad Opteron. The FPGA performance includes cycles lost due to loading the graph and extracting the label, and is also bandwidth limited to 133MBytes/s to simulate a connection over a PCI bus. For all graphs of size less than 12 the FPGA is bandwidth limited (except in the Brute Force case), and for $wv = 2$ and $wv = 3$ the computational load stays so low relative to IO, due to the small number of swaps per graph, that the FPGA is bandwidth limited for all graph sizes.

The hardware costs also include the time taken to load input graphs and read back labels between labelling operations. The hardware is also assumed to be connected to a PC via a 133MByte/sec PCI bus, so hardware performance is

limited by the bandwidth required to transfer the graphs. The small number of swap-compares required when $w_v \geq 2$ mean that the hardware becomes completely IO limited, and for these graphs can only achieve about a ten times speed-up over the quad processor software. However, for $w_v \leq 1$ the computational load quickly increases to the point where bandwidth is not the limiting factor, and for graph sizes between 12 and 15 the hardware becomes computationally limited. Even before this point is reached the software speed degrades exponentially, so practical speed-ups over 100 times are possible.

6 Conclusion

This paper has presented an architecture, implementation and evaluation of matrix based canonical labelling in hardware. The raw processing rate of a single hardware labelling unit is approximately 10 times that of a quad processor software implementation, and by utilising multiple labelling units within an FPGA, the raw performance is over 100 times that of software.

In a simulation of practical graph labelling, a minimum speed-up of around 10 times is predicted. However, for graphs with few vertex invariants, such as unlabelled graphs, a speed-up of around 100 times is expected. This is important, as it means that the largest speed-up is expected in those cases that are the most computationally expensive in software.

Future work will focus on benchmarking the real-world performance once the hardware labelling unit is integrated into a software application. This will explore the effect that software to hardware bandwidth has on performance, and how to store and transmit graphs to allow efficient processing in both software and hardware.

References

1. Bu, D., Zhao, Y., Cai, L., Xue, H., Zhu, X., Lu, H., Zhang, J., Sun, S., Ling, L., Zhang, N., Li, G., Chen, R.: Topological structure analysis of the proteinprotein interaction network in budding yeast. Nucleic Acids Research **31**(9) (2003) 2443–2450
2. Huan, J., Wang, W., Washington, A., Prins, J., Shah, R., Tropsha, A.: Accurate classification of protein structural families based on coherent subgraph analysis. In: Proc. Pacific Symposium on Biocomputing (PSB). (2004) 411–422
3. Huan, J., Wang, W., , Prins, J.: Efficient mining of frequent subgraph in the presence of isomorphism. In: Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM). (2003) 549–552
4. Kuramochi, M., Karypis, G.: Frequent subgraph dscovery. Technical Report TR-01-028, Department of Computer Science, University of Minnesota (2001)
5. Fortin, S.: The graph isomorphism problem. Technical Report TR-96-20, University of Alberta (1996)
6. McKay, B.: Practical graph isomorphism. Congressus Numerantium (1981) 45–87