Cascaded Window Memoization for Medical Imaging

Farzad Khalvati, Mehdi Kianpour, and Hamid R. Tizhoosh

Department of Systems Design Engineering,
University of Waterloo,
Waterloo, Ontario, Canada
{farzad.khalvati,m2kianpo,tizhoosh}@uwaterloo.ca

Abstract. *Window Memoization* is a performance improvement technique for image processing algorithms. It is based on removing computational redundancy in an algorithm applied to a single image, which is inherited from data redundancy in the image. The technique employs a fuzzy reuse mechanism to eliminate unnecessary computations. This paper extends the window memoization technique such that in addition to exploiting the data redundancy in a single image, the data redundancy in a sequence of images of a volume data is also exploited. The detection of the additional data redundancy leads to higher speedups. The cascaded window memoization technique was applied to Canny edge detection algorithm where the volume data of prostate MR images were used. The typical speedup factor achieved by cascaded window memoization is 4.35x which is 0.93x higher than that of window memoization.

Key words: Fuzzy memoization, Inter-frame redundancy, Performance optimization

1 Introduction

While high volume data processing in medical imaging is a common practice, the high processing time of such data significantly increases computational cost. The urgent need for optimizing the medical imaging algorithms have been frequently reported from both industry and academia. As an example, the processing times of 1 minute [1], 7 minutes [2], and even 60 minutes [3] are quite common for processing the volume data in medical imaging. Considering the high volume of data to be processed, these reported long processing times introduce significant patient throughput bottlenecks that have a direct negative impact on access to timely quality medical care. Therefore, accelerating medical imaging algorithms will certainly have a significant impact on the quality of medical care by shortening the patients' access time.

Currently, the medical image processing end-product market is dominated by real-time applications with both soft and hard time constraints where it is imperative to meet the performance requirements. In hard real-time medical systems,

such as image guided surgery, it is crucial to have a high-performance system. In soft real-time systems, such as many diagnostic and treatment-planning tasks in medical imaging, although it is not fatal to not have a high-performance system, it is important to increase the speed of the computations to enable greater patient throughput. The combination of complex algorithms, high volume of data, and high performance requirements leads to an urgent need for optimized solutions. As a result, the necessity and importance of optimization of medical image processing is a hard fact for the related industry.

In general, the problem of optimizing computer programs are tackled with the following solutions:

- Fast Processors (e.g. Intel Core i7): The increasing speed of processors is mainly due to more transistors per chip, fast CPU clock speeds, and optimized architectures (e.g. memory management, out of order superscalar pipeline).
- Algorithmic Optimization (e.g. Intel IPP: Integrated Performance Primitives [4]): The optimization is done by decreasing the algorithm complexity by reducing the number of computations.
- Parallel Processing: It takes advantage of the parallel nature of hardware design in which the operations can be done in parallel. This can be exploited in both hardware (Graphics Processing Unit: e.g. NVIDIA [5]) or software (RapidMind Multi-core Development Platform, now integrated with Intel called as Intel Array Building Blocks [6]).

Among these optimization solutions, using fast processors are the most obvious and simplest one. In academia, in particular, this has been an immediate response in dismissing the need for optimization methods for image processing and medical imaging. Although using fast processors can always be an option to tackle the performance requirements of a medical imaging solution, it suffers from two major shortcomings; first, it is not always possible to use high-end processors. One example is embedded systems where the low-end embedded processors are used. Second, in the past 20 years, the processors' performance has increased by 55% per year [7]. For a medical imaging solution that currently takes 10 minutes to run on a high-performance computer, in order to reduce the processing time down to 1 minute, it will take more than 6 years for the processors to catch up with this performance requirement. Obviously, this is not a viable solution for many performance-related problems.

Algorithmic optimization is a fundamental method which aims at reducing the number of operations used in an algorithm by decreasing the number of steps that the algorithm requires to take. Parallel processing breaks down an algorithm into processes that can be run in parallel while preserving the data dependencies among the operations. In both schemes, the building blocks of the optimization methods are the operations where the former reduces the number of operations and the latter runs them in parallel. In both cases, the input data is somewhat ignored. In other words, the algorithmic optimization will be done in the same way regardless of the input data type. Similarly, running the operations in parallel is independent of the input data. Window memoization, introduced

in [8], is a new optimization technique which can be used in conjunction with either of these two optimization methods

Window memoization is based on fuzzy *reuse* or *memoization* concept. Memoization is an optimization method for computer programs which avoids repeating the function calls for previously seen input data. Window memoization applies the general concept of memoization to image processing algorithms by exploiting the image data redundancy and increases the performance of image processing. The reuse method used by window memoization is a simple fuzzy scheme; it allows for the reuse of the results of similar but non-identical inputs. Thus far, window memoization has been applied to single frame images and therefore, it has been able to exploit the intra-frame data redundancy of image data to speed up the computations significantly [8]. In this work, we extend the window memoization technique to be applicable to sets of volume data where in addition to intra-frame data redundancy, the inter-frame data redundancy is exploited as well. This extra data redundancy (i.e. inter-frame) leads to higher speedups compared to intra-frame redundancy based window memoization. We call this new window memoization technique that exploits both inter-and intra-frame data redundancy *Cascaded Window Memoization*.

The organization of the remaining of this paper is as follows. Section 2 provides a brief background review on the window memoization technique. Section 3 presents the cascaded window memoization technique. Sections 4 and 5 present the results and conclusion of the paper, respectively.

2 Background

In this section, a background review of the window memoization technique and its fuzzy reuse mechanism is given.

2.1 Window Memoization

Introduced in [8], window memoization is an optimization technique for image processing algorithms that exploits the data redundancy of images to reduce the processing time. The general concept of memoization in computer programs has been around for a long time where the idea is to speed up the computer programs by avoiding repetitive/redundant function calls [9] [10] [11]. Nevertheless, in practice, the general notion of memoization has not gained success due to the following reasons: 1) the proposed techniques usually require detailed profiling information about the runtime behaviour of the program which makes it difficult to implement [12], 2) the techniques are usually generic methods which do not concentrate on any particular class of input data or algorithms [13], and 3) the memoization engine used by these techniques are based on non-fuzzy comparison where two input data is considered equal (in which case one's result can be reused for another) if they are identical. In contrast, in designing the window memoization technique in [8], the unique characteristics of image processing algorithms have been carefully taken into account to enable window memoization

to both be easy to implement and improve the performance significantly. One important property of image data is the fact that it is tolerant to small changes that a fuzzy reuse mechanism would produce. This has been exploited by window memoization to increase the performance gain.

The window memoization technique minimizes the number of redundant computations by identifying similar groups of pixels (i.e. window) in the image using a memory array, reuse table, to store the results of previously performed computations. When a set of computations has to be performed for the first time, they are performed and the corresponding results are stored in the reuse table. When the same set of computations has to be performed again, the previously calculated result is reused and the actual computations are skipped. This eliminates the redundant computations and leads to high speedups.

2.2 Fuzzy Memoization

Although window memoization can be applied to any image processing algorithm, local algorithms, in particular, are the first to adopt the technique. Local processing algorithms mainly deal with extracting local features in image (*e.g.* edges, corners, blobs) where the input to the algorithm is a window of pixels and the output is a single pixel. When a new window of pixels arrives, it is compared against the ones stored in the reuse table. In order for a window to find the exact match in the reuse table (i.e. hit), all pixels should match:

$$\forall pix \in win_{new}, \forall pix' \in win_{reuse},\ pix = pix' \implies win_{new} = win_{reuse} \quad (1)$$

where pix and pix' are corresponding pixels of the new window win_{new} and the one already stored in the reuse table win_{reuse}, respectively. In the above definition, a hit occurs if all the corresponding pixels of the windows are identical. This limits the *hit rate*: the percentage of windows that match the one stored in the reuse table. In order to increase the hit rate, we use fuzzy memoization. The human vision system cannot distinguish small amounts of error in an image, with respect to a reference image. Therefore, small errors can usually be tolerated in an image processing system. The idea of using this characteristic of image data to improve performance was introduced by [14] where a fuzzy reuse scheme was proposed for microprocessor design in digital hardware.

By using the fuzzy window memoization, those windows that are similar but not identical are assumed to provide the same output for a given algorithm.

$$\forall pix \in win_{new}, \forall pix' \in win_{reuse},\ MSB(d, pix) = MSB(d, pix') \implies win_{new} = win_{reuse} \quad (2)$$

where $MSB(d, pix)$ and $MSB(d, pix')$ represent d most significant bits of pixels pix and pix' in windows win_{new} and win_{reuse}, respectively. By reducing d, similar but not necessarily identical windows are assumed to have identical results for a given algorithm. This means that the response of one window may

be assigned to a similar but not necessarily identical window. As d decreases, more windows with minor differences are assumed equal and thus, the hit rate of window memoization increases drastically. Assigning the response of a window to a similar but not necessarily identical window introduces inaccuracy in the result of the algorithm to which window memoization is applied. However, in practice, the accuracy loss in responses is usually negligible.

We perform an experiment, using our memoization mechanism, to pick an optimal d that gives high hit rates with small inaccuracy in results. For our experiments, we use an ideal algorithm which outputs the central pixel of the input 3×3 window as its response. We use different values for d from 1 to 8. As the input images, we use a set of 40 natural images of 512×512 pixels. The accuracy of the results is calculated as SNR[1]. In calculating SNR, we replace an infinite SNR with 100, in order to calculate the average of SNRs of all images.

Figure 1 shows the average hit rate and SNR of the result for each value of d. It is seen that as d decreases, hit rate increases and at the same time, SNR decreases. The error in an image with SNR of $30dB$ is nearly indistinguishable by the observer [15]. Therefore, we pick d to be 4 because it gives an average SNR of $29.68dB$, which is slightly below the value $30dB$. Reducing d from 8 to 4 increases the average hit rate from 10% to 66%. For our experiments in the remaining of this chapter, we will choose d to be 4.

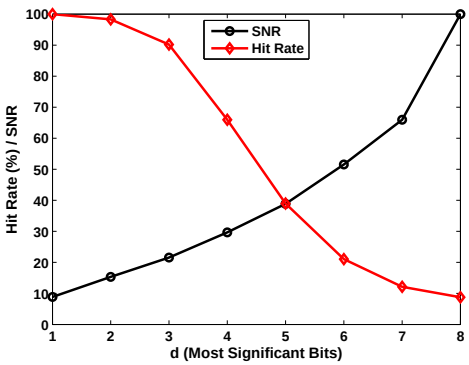

Fig. 1. Average hit rate and SNR versus the number of the most significant bits used for assigning windows to symbols. Infinite SNRs have been replaced by SNR of 100.

In the previous work [8], an optimized software architecture for window memoization has been presented where the typical speedups range from 1.2x to 7.9x with a maximum factor of 40x.

[1] The error in an image (Img) with respect to a reference image (R_{Img}) is usually measured by *signal-to-noise ratio (SNR)* as $SNR = 20log_{10}(\frac{A_{signal}}{A_{noise}})$ where A_{signal} is the RMS (root mean squared) amplitude. A^2_{noise} is defined as: $A^2_{noise} = \frac{1}{rc}\sum_{i=0}^{r-1}\sum_{j=0}^{c-1}(Img(i,j) - R_{Img}(i,j))^2$ where $r \times c$ is the size of Img and R_{Img}.

3 Cascaded Window Memoization

In this paper, we improve the the window memoization technique by extending it from exploiting only intra-frame data redundancy to inter-frame data redundancy as well. The main objective is that for a given set of images and an image processing algorithm, to further speed up the computations. The actual speedup achieved by the window memoization technique is calculated by equation 3 [8].

$$speedup = \frac{t_{mask}}{t_{memo} + (1 - HR) \times t_{mask}} \tag{3}$$

where:

- t_{mask} is the time required for the actual mask operations of the algorithm at hand.
- t_{memo} or the memoization overhead cost is the extra time that is required by the memoization mechanism to actually reuse a result.
- HR or hit rate determines the percentage of the cases where a result can be reused.

In the equation above, for a given processor, t_{mask} depends on the complexity of the algorithm to be optimized. For a fixed t_{mask}, to achieve high speedups, the hit rate must be maximized and the memoization overhead cost must be minimized. We increase the hit rate by extending the window memoization technique from exploiting only intra-frame data redundancy to utilize both intra- and inter-frame data redundancy.

While window memoization minimizes the number of redundant computations by identifying similar groups of pixels in a single image (i.e. intra-frame data redundancy), cascaded window memoization discovers the data redundancy not only in a single frame, but also among the multiple frames of the same volume data (i.e. inter-frame data redundancy). In other words, the similar groups of pixels are compared against the ones that are from all frames of a volume data that have been previously processed.

Figure 2 shows the flowchart of the the cascaded window memoization technique. In cascaded window memoization, the reuse table is initialized only once for the entire volume data inputs; that is when the first frame arrives. After the first initialization, for the rest of the frames in the same volume data, it is assumed that the reuse table contains valid data and needs not to be reset again. As a result, not only the similar neighborhoods in a singe frame are detected and the corresponding results are reused, the same scenario occurs for the neighborhoods which belong do different frames of the same volume data. This, in fact, exploits the inter-frame data redundancy, which is in addition to the intra-frame data redundancy. Thus, the cascaded window memoization technique increases the average hit rate and hence, the average speedup for each frame increases compared to regular window memoization which is based on intra-frame redundancy only.

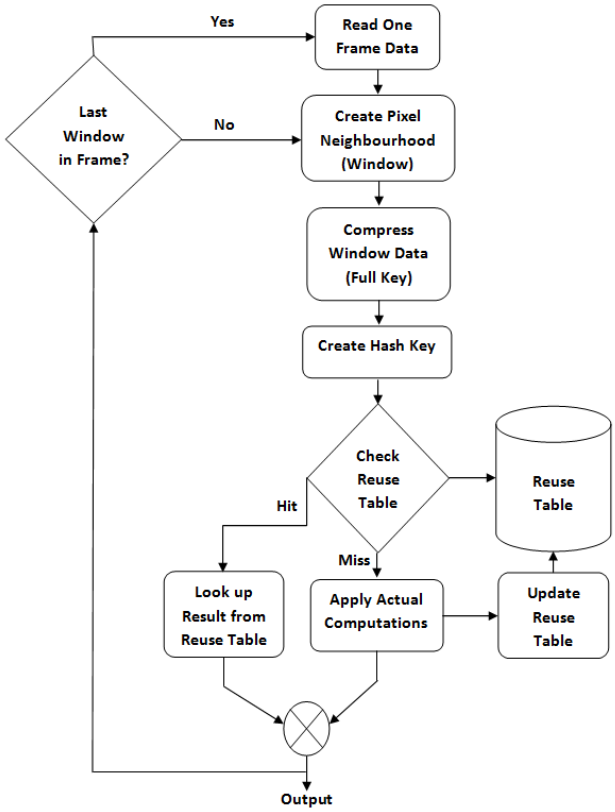

Fig. 2. Cascade Window Memoization: The flowchart

In order to translate the extra hit rates to actual speedups, it is crucial that exploiting inter-frame redundancy does not introduce extra memoization overhead time. Otherwise, the increase in hit rate will be compensated by the extra overhead time, leading to no increase (or decrease) in the speedup.

At each iteration, a window of pixels are compressed (using equation 2) to create $full\_key$. A $full\_key$ is a unique number that represents similar groups of pixels which are spread among different frames of a volume data. Compressing $full\_key$ is a necessary step; it causes that similar but not necessarily identical groups of pixel to be considered equal. This introduces a small amount of inaccuracy in the results, which is usually negligible. Moreover, the compression reduces the reuse table size which holds the $full\_keys$ and the results.

In order to map a $full\_key$ to the reuse table, a hash function is used to generate $hash\_key$. As discussed in [8], the multiplication method is used as

the hash function. In order to maximize the HR and minimize t_{memo} (i.e. to minimize the collisions), a constant number is used for calculating the $hash\_key$. It is important to note that this optimal constant number is different for 32-bit and 64-bit processor architectures [16][2]. Thus, it is imperative that the window memoization technique detects (automatically or by user input) the architecture type and set the appropriate constant number. Using 64-bit processor reduces the memoization overhead time in comparison to a 32-bit processor. The reason is that for a given $full\_key$, in the 64-bit processor less number of memory accesses is required compared to that of the 32-bit processor. Less number of memory accesses means lower memoization overhead time and hence, higher speedup (equation 3)[3].

4 Results

In order to measure the performance of the cascaded window memoization technique, the following setups were used:

- Case study algorithm: Canny edge detection algorithm that uses windows of 3×3 pixels.
- Processor: Intel (R) Core (TM)2 Q8200 processor; CPU: 2.33GHz, Cache size 4MB.
- Reuse Table size: 64KB.
- Input images: Volume data for Prostate MR Images of 5 patients. The MR images are T2 weighted with endorectal coil (180 frames in total)[4].

We applied Canny edge detection algorithm to all five patients volume data. Figure 3 shows the hit rate of the frames of a sample patient volume data for regular and cascaded window memoization applied to the Canny Edge detector. As it is seen, exploiting inter-frame redundancy by cascaded window memoization increases the hit rate, on average, by 4.44%.

Figure 4 shows the actual speedup of the frames of the sample patient volume data for regular and cascaded window memoization applied to the Canny Edge detector. As it is seen, exploiting inter-frame data redundancy by cascaded window memoization increases the speedup, on average, by a factor of 1.01x. All the reported speedup factors are based on running the actual C++ code implemented for (cascaded) window memoization and measuring the CPU time. The average accuracy of the results is above 96%.

As for the entire data, on average, cascaded window memoization increased the hit rate and speedup for Canny edge detection algorithm from 84.88% and 3.42x to 90.03% and 4.35x respectively; an increase of 5.15% and 0.93x, respectively (Table 1).

[2] This constant number for 32-bit and 64-bit processor architecture is '2,654,435,769' and '11,400,714,819,323,198,485', respectively [16].

[3] This was verified by experimental data.

[4] Taken from online Prostate MR Image Database:(http://prostatemrimagedatabase.com /Database/index.html).

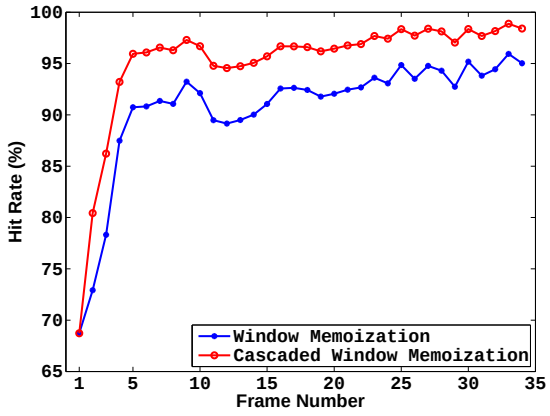

Fig. 3. Hit Rate: Cascaded Window Memoization versus Regular Window Memoization

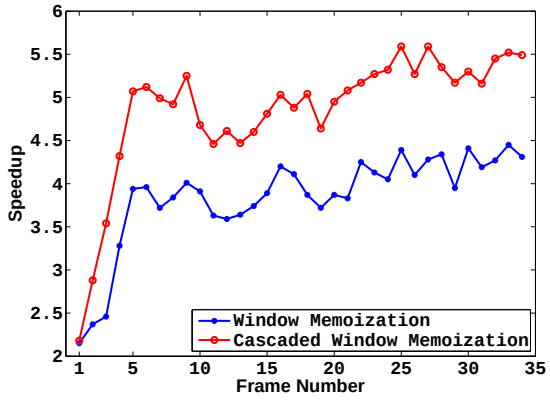

Fig. 4. Speedup: Cascaded Window Memoization versus Regular Window Memoization

5 Conclusion

Window memoization is an optimization technique that exploits the intra-frame data redundancy of images to reduce the processing time. It uses a fuzzy memoization mechanism to avoid unnecessary computations. In this paper, we extended the window memoization technique to exploit the inter-frame data redundancy as well. It was shown that the elimination of the extra data redundancy leads to higher speedup factors. For a case study algorithm applied to Prostate MR Images of 5 patients volume data, on average, the original window memoization technique yielded the speedup factor of 3.42x. Cascaded window memoization increased the speedup factors by, on average, 0.93x yielding average speedup factor of 4.35x.

Table 1. Hit Rate and Speedup Results

Patient data	Hit Rate	Hit Rate (Cascaded)	Speedup	Speedup (Cascaded)
1	84.07%	89.88%	3.24x	4.12x
2	79.96%	85.36%	3.09x	3.94x
3	82.45%	87.60%	3.29x	4.21x
4	87.20%	92.17%	3.62x	4.61x
5	90.70%	95.14%	3.85x	4.86x
Average	84.88%	90.03%	3.42x	4.35x

References

1. B. Haas et al., "Automatic segmentation of thoracic and pelvic CT images for radiotherapy planning using implicit anatomic knowledge and organ-specific segmentation strategies," *Phys. Med. Biol*, vol. 53, pp. 1751–1771, 2008.
2. A. C. Hodgea et al., "Prostate boundary segmentation from ultrasound images using 2D active shape models: Optimisation and extension to 3D," *Computer methods and programs in biomedicine*, vol. 84, pp. 99–113, 2006.
3. A. Gubern-Merida and R. Marti, "Atlas based segmentation of the prostate in MR images," in *MICCAI: Segmentation Challenge Workshop*, 2009.
4. Intel Integrated Performance Primitives, "`http://software.intel.com/en-us/articles/intel-ipp/`," .
5. NVIDIA, "`http://www.nvidia.com/`," .
6. RapidMind, "software.intel.com/en-us/articles/intel-array-building-blocks/," .
7. J. L. Hennessy and D. A. Patterson, *Computer Architecture - A quantitative approach*, Morgan Kaufmann Publishers, fourth edition, 2007.
8. F. Khalvati, *Computational Redundancy in Image Processing*, Ph.D. thesis, University of Waterloo, 2008.
9. D. Michie, "Memo functions and machine learning," *Nature*, vol. 218, pp. 19–22, 1968.
10. R. S. Bird, "Tabulation techniques for recursive programs," *ACM Computing Surveys*, vol. 12, no. 4, pp. 403–417, 1980.
11. W. Pugh and T. Teitelbaum, "Incremental computation via function caching," in *ACM Symposium on Principles of Programming Languages*, 1989, pp. 315–328.
12. W. Wang, A. Raghunathan, and N. K. Jha, "Profiling driven computation reuse: An embedded software synthesis technique for energy and performance optimization," in *IEEE VLSID-04 Design*, 2004, p. 267.
13. J. Huang and D. J. Lilja, "Extending value reuse to basic blocks with compiler support," *IEEE Transactions on Computers*, vol. 49, pp. 331–347, 2000.
14. E. Salami C. Alvarez, J. Corbal and M. Valero, "On the potential of tolerant region reuse for multimedia applications," in *International Conference on Supercomputing*, 2001, pp. 218–228.
15. Carlos Alvarez, Jesus Corbal, and Mateo Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922–927, July 2005.
16. B. R. Preiss, *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*, John Wiley and Sons, 1999.