JAUES

Journal of Al-Azhar University Engineering Sector

Vol.15, No. 55, April 2020, 604-613

CACHE MEMORY LOACLITY OPTIMIZATION FOR IMPLEMENTATION OF COMPUTER VISION AND IMAGE PROCESSING ALGORITHMS

A. Al-Marakeby
Systems and Computers Engineering Dept. , Faculty of Engineering , Al-Azhar University, Cairo, Egypt.
E-mail: a.marakeby@azhar.edu.eg

ABSTRACT

One of the main problems in developing fast image processing and computer vision systems is the memory speed. Memory speed represents the performance bottleneck due to the large gap between processor and memory speeds. Cache memory is very fast, but it is small to store all required data and instructions. In this paper , image processing and computer vision algorithms are optimized to enhance performance by increasing the cache memory utilization. This optimization increases the spatial locality and temporal locality and improves the system performance. The proposed optimization is applied on a set of image processing operations such as image intensity transformation, image filtering, geometric transformation, and CNN. The time analysis of the systems has shown a speed improvement of 30% to 70% compared with direct algorithm implementation.

KEYWORDS: Cache Memory, Image Processing, Computer Vision, Locality Of Reference, Code Optimization.

تحسين تركيز الذاكرة المخبأة لتنفيذ خوارزميات الرؤية بالحاسب و معالجة الصور

أشرف المراكبي
قسم هندسة النظم والحاسبات، كلية الهندسة ،جامعة الأزهر، القاهرة ، مصر
البريد الإلكتروني للباحث: E-mail: a.marakeby@azhar.edu.eg

الملخص

سرعة الذاكرة تمثل واحدة من أهم المشكلات التي تواجه عملية تطوير أنظمة سريعة لمعالجة الصور والرؤية بالحاسب. إن سرعة الذاكرة تمثل عنق الزجاجة في تحقيق أداء جيد نظرا للفجوة الكبيرة بين سرعة الذاكرة وسرعة المعالج. الذاكرة المخبأة هي ذاكرة سريعة جدا ولكنها صغيرة ولا يمكنها تخزين جميع البيانات والتعليمات المطلوبة. في هذا البحث يتم عمل تحسين لخوازميات الرؤية بالحاسب ومعالجة الصور من أجل تحسين الأداء عن طريق زيادة استغلال أفضل للذاكرة المخبأة. هذه الأمثلة تزيد من التركيز المكاني والتركيز الوقتي لتحسين الأداء. الأمثلة المقترحة تم تطبيقها على العديد من عمليات معالجة الصور مثل تحويلات الوضوح و مرشحات الصور و التحويلات الهندسية و الشبكات العصبية. إن تحليلات الوقت لهذه الأنظمة أثبتت تحسينات في السرعة تصل من ٣٠% إلى ٧٠% مقارنة بالطرق المباشرة لتنفيذ هذه الخوارزميات.

الكلمات المفتاحية: الذاكرة المخبأة ، معالجة الصور ، الرؤية بالحاسب ، تركيز المرجعية،تحسين الأكواد.

1. INTRODUCTION

Processors can consume data much faster than memory can supply it[4]. Main memory latency is around 200-300 processor cycles, causing a large speed impact [11]. The memory speed bottleneck can be solved by good utilization of cache memory. Cache memory operates between 10 to 100 times faster than main memory. The cache memory size represents the main problem in obtaining high performance specially when working with large data sets. Image processing and computer vision algorithms requires very large size memory buffers which can't be stored in cache memory. With the new high resolution cameras, a single frame can reach more than 10 M Pixels , and working with real time video streams requires more than 100 M Byte memory buffers per sec. The size of L1 (Level 1) is about 128 KB while the LLC (Last level cache) can be in the range of 4-16 MB. Exploiting locality of reference improves the hit ratio of cache memory and hence improves the total performance of the system. Locality of reference is the tendency to access the same set of memory locations repetitively over a specific period. Locality of reference can be divided into two types: temporal locality , and spatial locality. The temporal locality indicates that, recently referenced items are likely to be referenced in the near future. The spatial locality refers to the use of data elements stored in locations close to the currently referenced item. When memory locations are accessed in a regular method, the locality of reference is increased and the performance is improved. Some image processing and computer vision algorithms have a regular data access nature, while others have irregular and randomly access patterns of memory locations. In both cases, taking cache memory locality into consideration has a large impact on the system performance. Because, software developers have no direct access to cache memory and this memory is completely controlled by hardware, many cache issues are neglected while writing codes. This neglection leads to producing many inefficient vision systems. The best solution is to control cache memory indirectly. Indirect control of cache memory is applied using many optimization techniques. This optimization can be done on the algorithmic level, and also in the implementation level. Simple algorithm modification can lead to large impact on time. The modifications and optimizations are based on the good understanding of hardware architecture, memory operation, image format , and algorithm nature. Profiling tools are used to obtain many measurements and to analyze the time hot spots. While cache memory stores data and instructions, profiling tools helps in the prediction of the next instruction and branching to prefetch the expected instructions. In this work , the focus is on image processing and computer vision systems. These systems have a special nature of containing huge data , but the number of instructions is very small. A simple algorithm may contain only tens of instruction but run over several Megabytes of data. This make the optimization more directed on data not instructions profiling and optimization. In this work, the optimization of image processing and computer vision algorithms to increase cache memory locality is presented. Profiling tools are used to detect the bottlenecks and time hot spots. The comparison of direct algorithm implementation and optimized algorithm are given which indicates the performance improvements. In section 2 related work is given. Section 3 introduces the hardware architecture of cache memory and the metrics used to measure the performance. Image processing and computer vision algorithms optimizations are given in section 4 . Experimental results are illustrated in section 5. Finally, a conclusion is given in section 6.

2. RELATED WORK

Cache locality optimizing have been studied for different systems and applications. Tran , et. al. have developed a system for optimizing cache locality for irregular data accesses on many-core Intel Xeon Phi accelerator chip [9]. They studied a multiple patterns string matching algorithm commonly used in computer and network security, bioinformatics, using Aho-Corasick (AC) algorithm. They presented a cache locality optimizing parallelization on the many-core accelerator chip, the Intel Xeon Phi. A given set of pattern strings is partitioned into multiple sets of a smaller number of patterns so that multiple small DFAs are constructed instead of single large DFA. The accesses to multiple small DFAs lead to significantly smaller cache footprints in each core's cache and result in impressive performance improvements. Experimental results on the Intel Xeon Phi 5110P show that their approach delivers up to 2.00- times speedup compared with the previous approach using single large DFA. In the field of image processing , Tao and Shahbahrami [8] worked in data locality optimization based on comprehensive knowledge of the cache miss reasons. They developed a set of toolkits including data profiling, pattern analysis, and performance visualization tools to help the user to optimize the source program towards a better runtime data locality. They demonstrated how the toolset can be used step-by-step to understand the cache access behavior of the applications and then achieve optimized program code. The Discrete Wavelet Transform (DWT), is applied as an example. Kim and Kweon have proposed a rolling cache design optimized for image format and algorithms [2]. Their method reduced the miss penalty by moving the cache horizontally and vertically. They compared the suggested design with other types of caches and the average memory access time and the memory bandwidth are decreased by 28% and 74%, respectively, for a 2048 x 2048 image size. Petko et. al. have studied cache performance of video computation workloads[5] . The workloads studied include PEG, MPEG-2 and H263 and the impact of cache sizes, block sizes and associativity on cache performance. Pesterev et. al. have studied locating cache performance bottlenecks using data profiling[4]. They presented two case studies of using a profiling tool called DProf to find and fix cache performance bottlenecks in Linux and achieved throughput improvements of 16-57% . Other work can be found in [1][6][10].

3. CACHE HIERARCHY AND PERFORMANCE METRICS

There are many different architectures for cache memories, but usually the cache memory consists of 3 or 4 levels as shown in fig.1**.** For multi-core processors L1 and L2 caches are separated for each core, while L3 or LLC are usually shared and common for all cores. L1 cache is the fastest cache and the most close memory to CPU after registers and it has a latency of 4-5 cycles. L1 cache size is typically goes up to 256KB but some powerful processors have 1MB L1 cache. There are two types of L1 cache: L1 D for data , and L1 I for instructions. L2 cache is bigger than L1 and it can store more data, but the access time is a bit slower than L1 (about 7 cycles). L2 is used for the both of instructions and data[3][12].

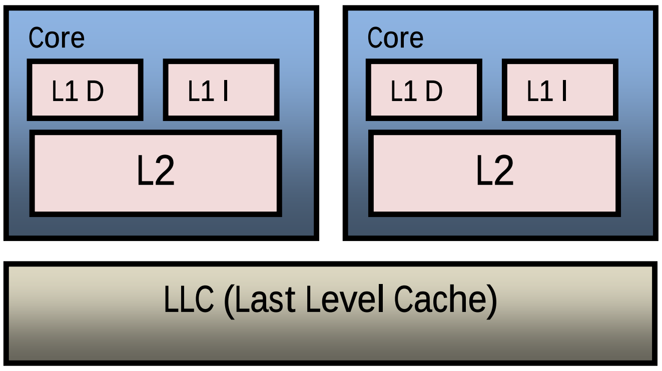
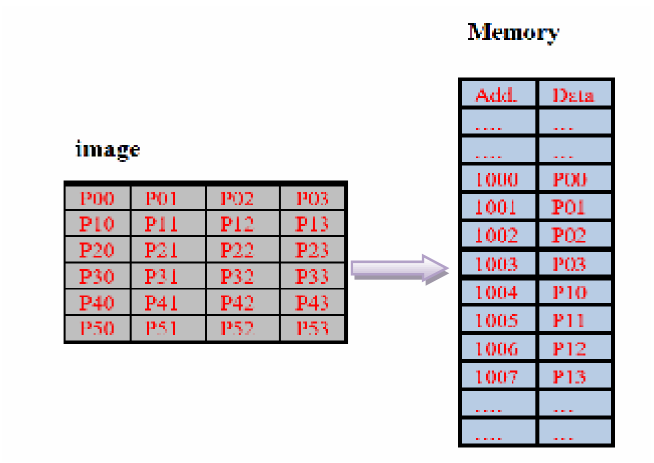


Fig.1 L1,L2, and LLC Cache memories

Last Level Cache (LLC) is the largest and slowest memory shared to all cores in the processor. LLC has a size within the range of 2-8 MB and it has a latency of 20-30 cycles. Because, cache memory can't store big data , a part of data is stored in the cache and the remaining still stored in the main memory. When the processor request a memory location (read or write), it first checks for a corresponding entry in the cache, and If the processor finds that the memory location is in the cache, a cache hit has occurred. If the processor does not find the memory location in the cache, a cache miss has occurred. The cache allocates a new entry and copies data from main memory in the miss case. Data and instructions are transferred between main memory and cache in blocks of specific size, called cache lines. The cache controller attempt to store data which will be accessed in the future by the processor [7]. This is done by loading additional data other than that being requested by the processor during a replacement process. Spatial locality is based on retrieving additional data from the neighboring address space of the requested data. Temporal locality refers to the reuse of specific data within a relatively small time duration. The performance of the cache memory is measured using different metrics. While the main target is to reduce the time of program execution, different measures and metrics are required for better understanding of latency reasons. Profiling tools are used to get different metrics. Intel VTune Profiler[11] is commonly used for intel processors profiling and can be used to catch different cache memory events. The ***hit ratio***, or percentage of accesses satisfied in the cache, can be used as a measure of cache performance. To get th\e hit ratio, **LLC Miss Count** metric is used which shows the total number of last-level cache misses. Some cache misses don't cause latency due to the pipelining in modern processors. So, another metric called **Memory Bound** is commonly used which has extra meaning and usage. **Memory Bound** metric shows a fraction of cycles spent waiting due to demand load or store instructions or data. **L1 Bound , L2 Bound, and L3 Bound,** are the detailed metric for each cache level and these metrics give better understanding of latency causes. The time analysis and the hardware events collected by the profiling tolls help in optimizing the algorithms to get better cache locality and hence improve the performance.

4. OPTIMIZATION OF VISION AND IMAGE PROCESSING ALGORITHMS

Storing multidimensional arrays in linear storage can be done using row-major order or column-major order. In the case of image pixels, using the row-major or column major affects the pixels adjacency patterns. Fig.2 shows a sample image stored in memory using row-major order.



```
for x=0:width
  for y=0:height
    Pixels[x,y]=NewVal
  end
end
```

Code 1 (Slow)

```
for y=0:height
  for x=0:width
    Pixels[x,y]=NewVal
  end
end
```

Code 2 (Fast)

Improved Spatial Locality

Fig.2 Major-Row ordered and spatial locality

Pixels in the same row are stored in sequential in adjacent memory locations, while pixels in the same column are stored far from each other's. column-major order has a reversed pattern. Simple image scan algorithms can be optimized by taking into account the major order used in representing images. Hence spatial locality can be increased by adjusting the inner and outer loops in the algorithm. Taking into consideration the cache memory issues, improves the performance and reduce the time required for many image processing operations. Not all operations are simple and have direct optimization solution like the previous example. Many operations and algorithms requires deep thinking , iterations , time analysis, and using profiling tools. For example image rotation algorithm requires reading from a memory buffer and writing in another buffer in a different sequence. Improving the reading sequence may cause inefficient spatial locality for writing , and vice versa. Fig.3 illustrates image tiling technique which divide the image into small parts and execute the operations over these small parts. Working with small parts increase the probability of cache hits due to reducing data size. The tile size depends on many factors such as cache size , type of operations , and image size.

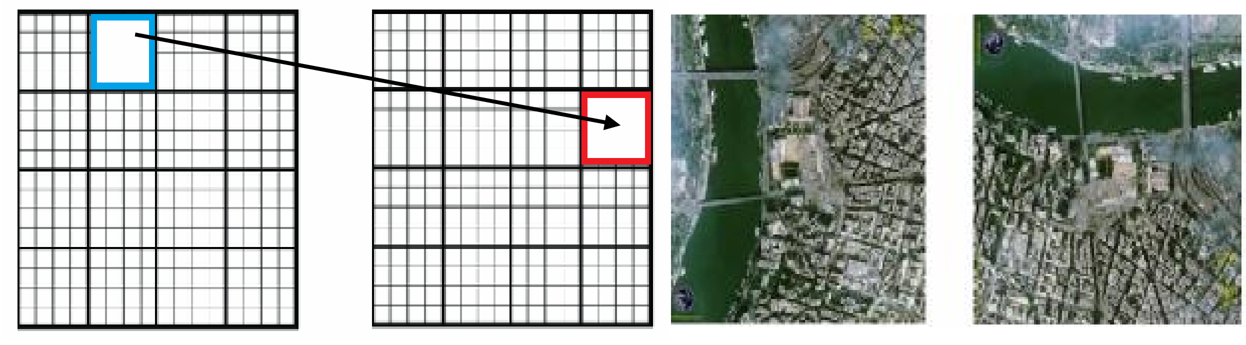


Fig.3 Using image tiling to improve locality of reference.

2D convolution and correlation are widely used for many applications in image processing and computer vision such as image filtering, image transformation, feature selection, feature extraction , and Convolutional Neural Networks (CNN) . When the kernel of convolution or correlation is small, there is no problem in cache operation. Using large size kernels or large number of kernels can increase the cache miss ratio and decrease the temporal locality.

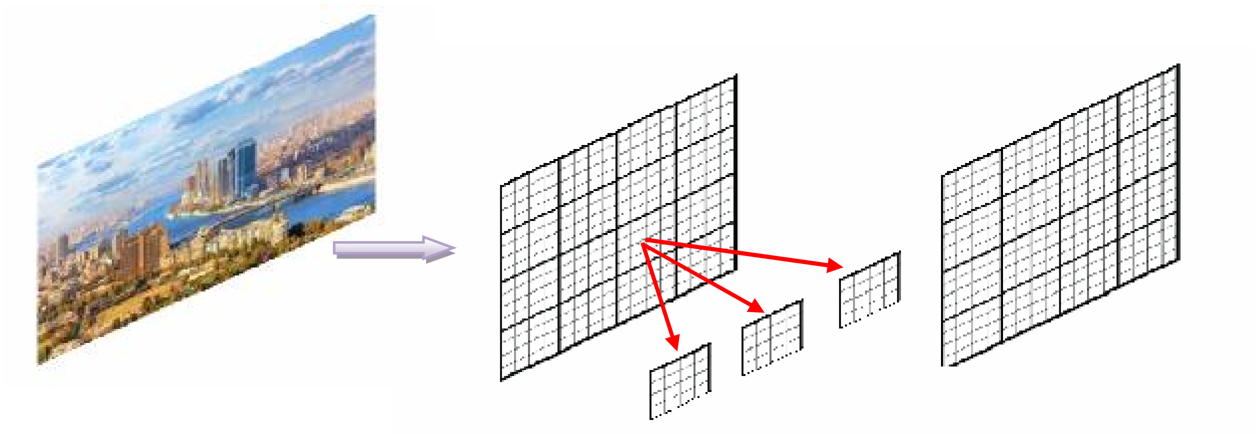
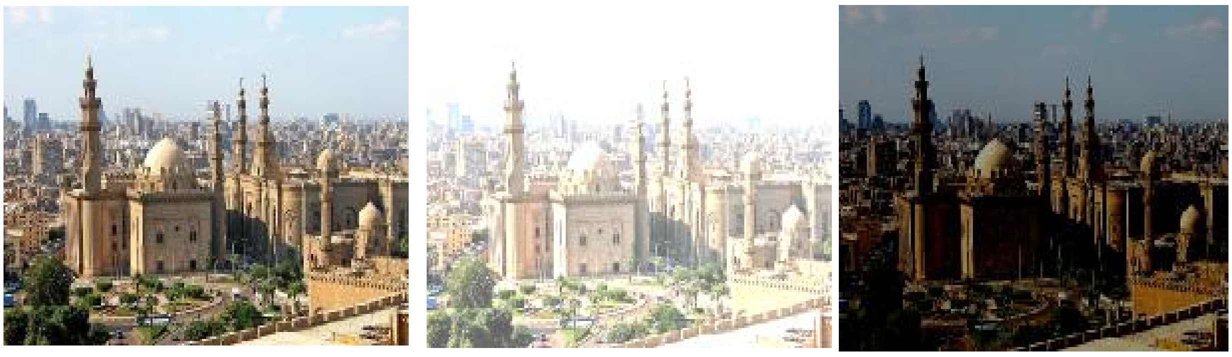


Fig.4 multiple convolutional kernels and temporal locality (CNN).

For some applications like simple edge detection, image smoothing or sharpening, convolution kernels from small 3 x 3 up to 7 x 7 are typically enough. For other applications like object tracking, estimation filtering, or pattern recognition, larger kernels can be required[10]. CNN is widely used in image analysis, video analysis, and computer vision with high accuracy. The number of kernels in CNN can be in the range of 16 kernels to 64 kernels. To increase the temporal locality for such problems, the kernels should be stored in cache memory. Putting the kernel loop as outer loops improves the temporal locality of the kernel data, but can lower the spatial locality of image data. Optimization of algorithms based on profiling tools, hardware events , and other metrics help in designing and implementing high performance systems by increasing temporal and spatial locality.

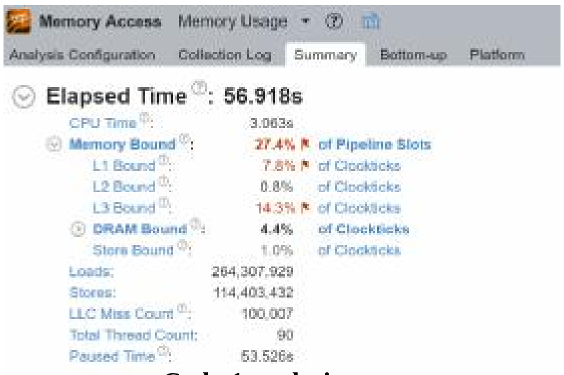
5. EXPERIMENTAL RESULTS

A computer with 128 KB L1 cache, 512 KB L2 cache , 3MB L3 cache is used to test and analyze different image processing and computer vision algorithms. Different images with sizes are used to test the effects of cache memory capacity. Intel VTune Profiler software [12] is used to collect different hardware events and memory metrics and to analyze the code to detect different hot spot points and to get different statistics. In Fig.1 a simple brightness adjustment is applied to an image with different resolutions and using two different codes shown in fig.2 (code-1 and code-2). The image is stored in memory in row-major order, so using vertical index as outer loop improves the spatial locality.



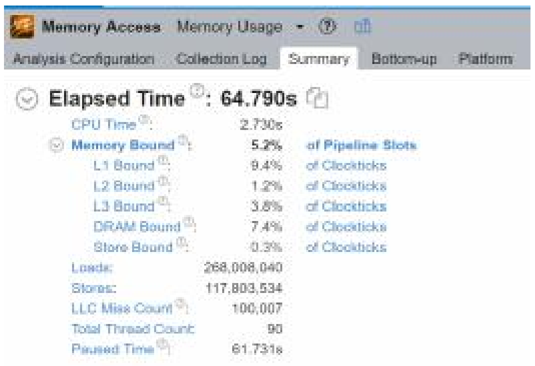
a) original image **b) increasing brightness** **c) decreasing brightness**

Fig.4 intensity transformation



```
Memory Access   Memory Usage
Analysis Configuration   Collection Log   Summary   Bottom-up   Platform

Elapsed Time: 56.918s
  CPU Time:                 3.063s
  Memory Bound:             27.4%  of Pipeline Slots
    L1 Bound:                7.8%  of Clockticks
    L2 Bound:                0.8%  of Clockticks
    L3 Bound:               14.3%  of Clockticks
    DRAM Bound:              4.4%  of Clockticks
    Store Bound:             1.0%  of Clockticks
  Loads:              264,307,929
  Stores:             114,403,432
  LLC Miss Count:         100,007
  Total Thread Count:          90
  Paused Time:            53.526s
```



```
Memory Access   Memory Usage
Analysis Configuration   Collection Log   Summary   Bottom-up   Platform

Elapsed Time: 64.790s
  CPU Time:                 2.730s
  Memory Bound:              5.2%  of Pipeline Slots
    L1 Bound:                9.4%  of Clockticks
    L2 Bound:                1.2%  of Clockticks
    L3 Bound:                3.8%  of Clockticks
    DRAM Bound:              7.4%  of Clockticks
    Store Bound:             0.3%  of Clockticks
  Loads:              268,008,040
  Stores:             117,803,534
  LLC Miss Count:         100,007
  Total Thread Count:          90
  Paused Time:            61.731s
```

Code-1 analysis **Code-2 analysis**

Fig.5 Performance Metrics for code-1 and code-2

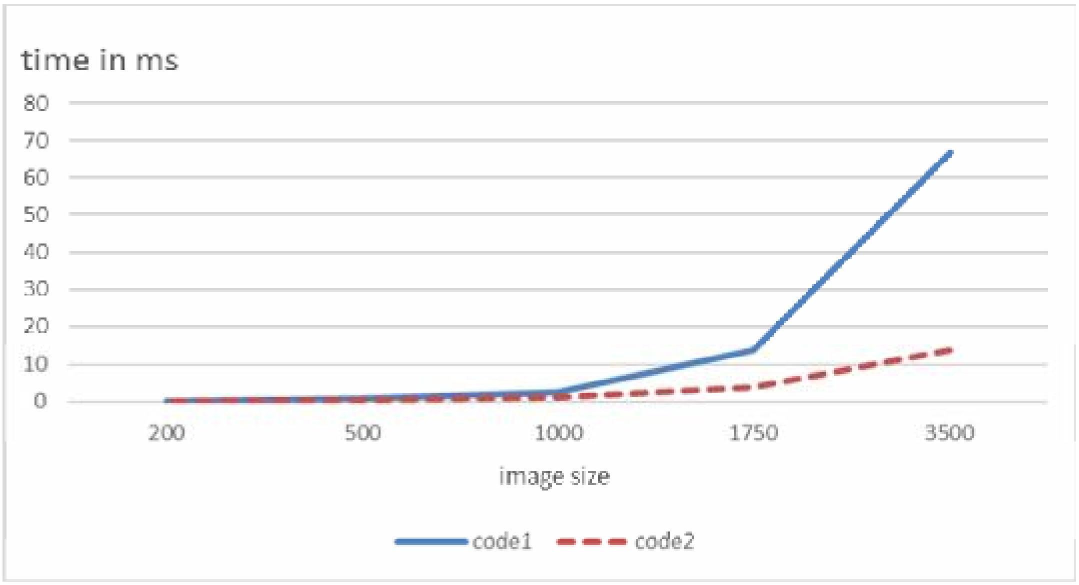


Fig.6 Time required for brightness adjustment for different image sizes

Fig.5 illustrates some metrics measured using profiling tools for the two different codes. It is clear that memory bound is high in code-1 due to the bad locality of reference. Code-2 has better memory bound rates and it is faster. Not only memory bound metrics are used but also, there are many other metrics used to get better understand of the reasons of latency such as L1D_Miss_Count, L2_Miss, L3_Miss, and other time analysis metrics. The time consumed by the brightness adjustment algorithms are shown in fig.6 for code-1 and code-2. For small image sizes the cache memory can store all data and hence , there is no big difference between the two codes. For large image sizes the difference is very clear, while the cache memory speed with better spatial locality improves the performance. For image size 3500 (3500 x 3500) code-1 requires 67 ms, while code-2 requires only 17 ms for the same operation. Working with image rotation and other geometric transformation , simple loop interchange is not enough to optimize the algorithm for better locality of reference. For example, the simple rotation shown in fig.3 copies data from input buffer in horizontal sequence and store in output in vertical sequence. The loop arrangement will be efficient for reading operation and will be inefficient for the writing or vice versa. Image tiling is used to improve the spatial locality for both operation and keep active part in cache memory for the both of input and output buffers. Fig.7 shows the time required for a rotation using tiling optimization and without tiling for fixed image size and using different tile sizes.it is clear that tile size selection affects the time and the best selection depends on many factors such as image size , cache size , and many other factors. Hotspots analysis is used in addition to many other metrics captured using profiling tools to get the best parameters for efficient algorithm implementation. Fig.8 shows the hotspots analysis which indicates the functions and line codes which consumes more CPU time. For convolution operation commonly used in many image processing and computer vision the temporal locality improvement is required to store the kernels in cache memory. Table.1 shows the values of a Gaussian kernel of size 9x9. For such sizes the temporal locality is high, but an optimization is required when the size is large or when the number of kernels is large.

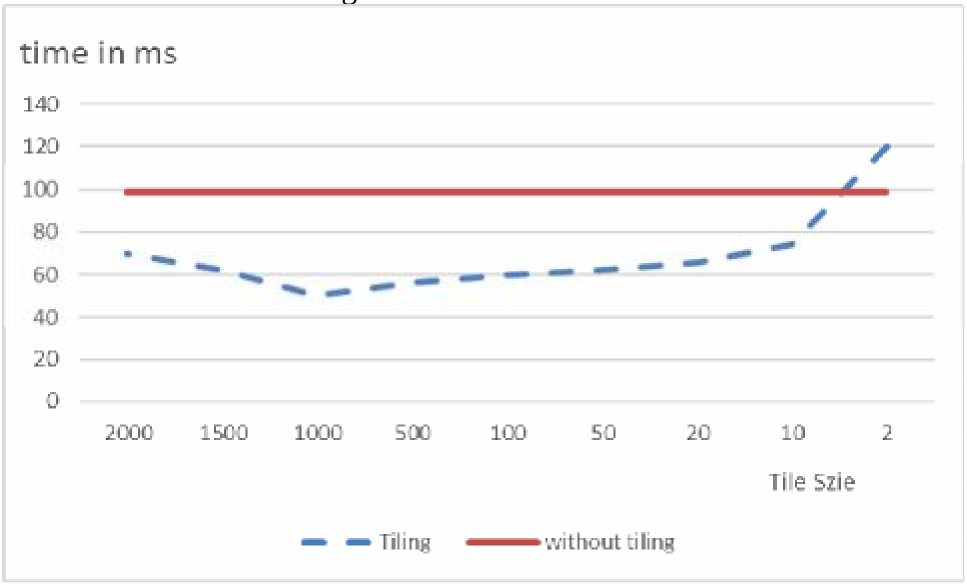
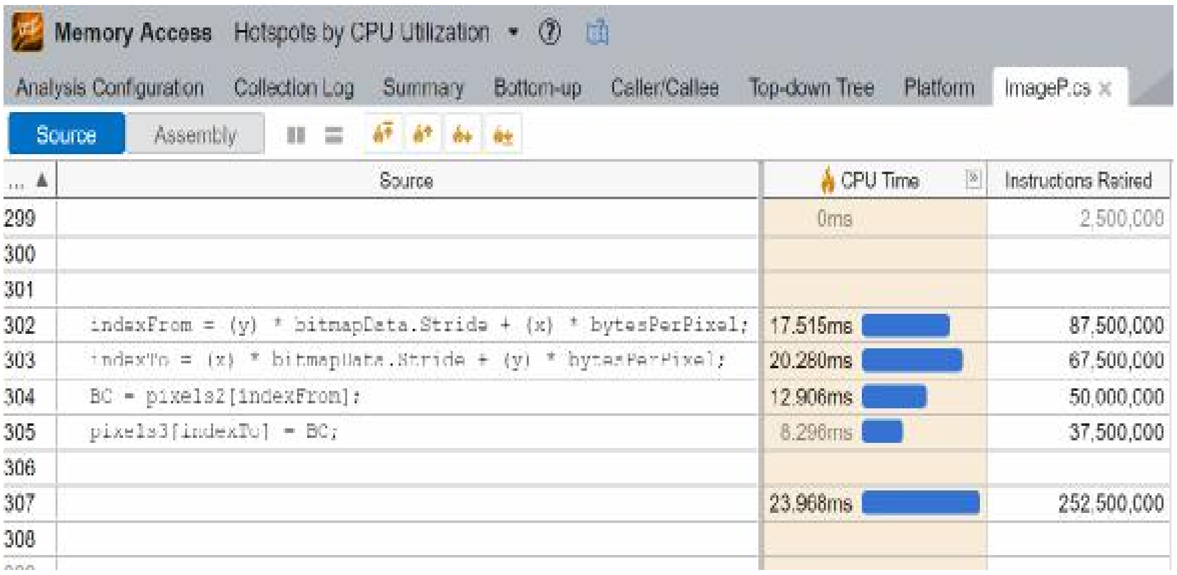
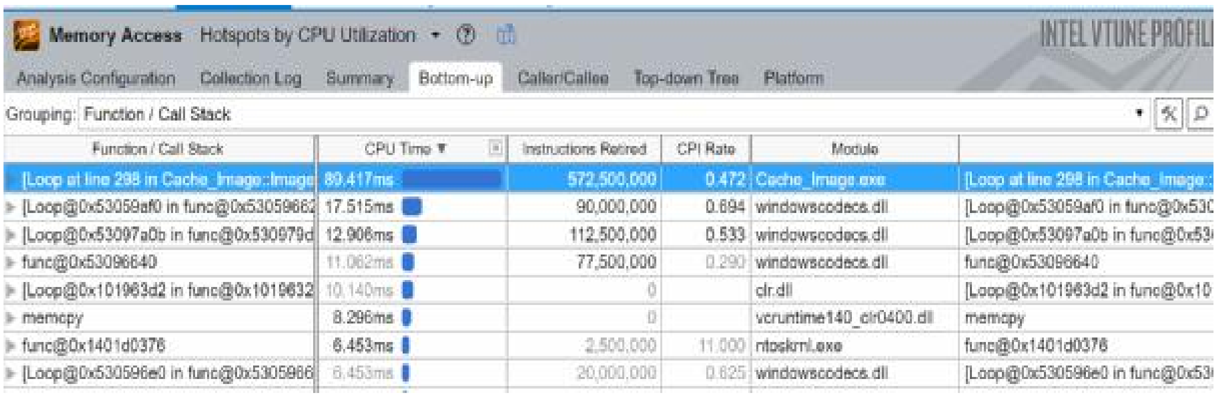


Fig.7 Rotation algorithm time with different tiling sizes



... ▲	Source	CPU Time	Instructions Retired
299		0ms	2,500,000
300			
301			
302	`indexFrom = (y) * bitmapData.Stride + (x) * bytesPerPixel;`	17.515ms	87,500,000
303	`indexTo = (x) * bitmapData.Stride + (y) * bytesPerPixel;`	20.280ms	67,500,000
304	`BC = pixels2[indexFrom];`	12.906ms	50,000,000
305	`pixels3[indexTo] = BC;`	8.296ms	37,500,000
306			
307		23.968ms	252,500,000
308			



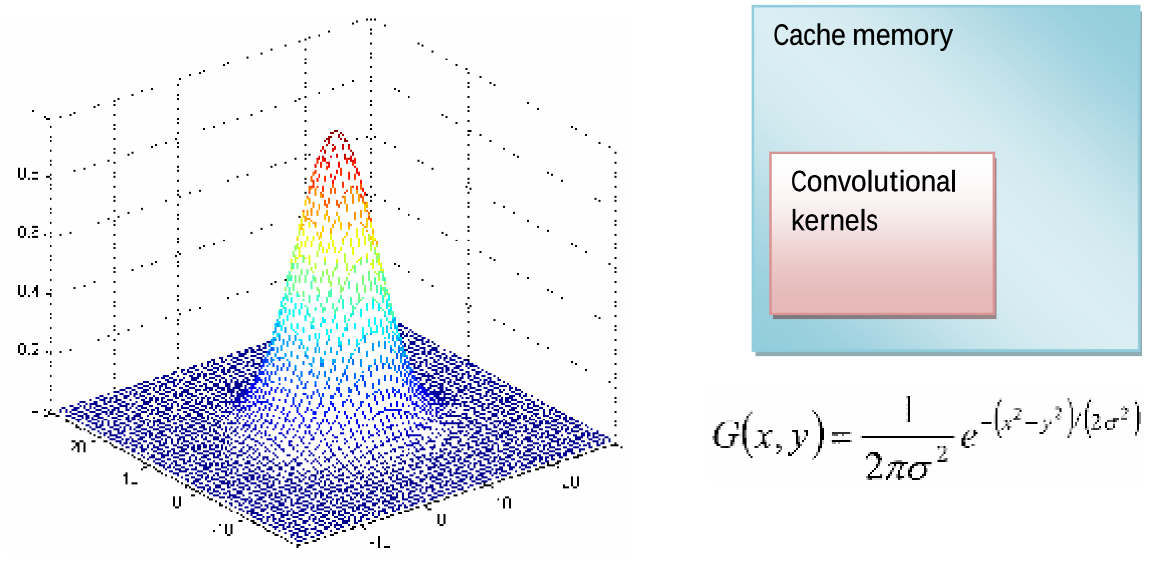
Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Instructions Retired	CPI Rate	Module	
[Loop at line 298 in Cache_Image::Image	89.417ms	572,500,000	0.472	Cache_Image.exe	[Loop at line 298 in Cache_Image::
[Loop@0x53059af0 in func@0x53059662	17.515ms	90,000,000	0.694	windowscodecs.dll	[Loop@0x53059af0 in func@0x530
[Loop@0x53097a0b in func@0x530979d	12.906ms	112,500,000	0.533	windowscodecs.dll	[Loop@0x53097a0b in func@0x53
func@0x53096640	11.062ms	77,500,000	0.290	windowscodecs.dll	func@0x53096640
[Loop@0x101963d2 in func@0x1019632	10.140ms	0		clr.dll	[Loop@0x101963d2 in func@0x10
memcpy	8.296ms	0		vcruntime140_clr0400.dll	memcpy
func@0x1401d0376	6.453ms	2,500,000	11.000	ntoskrnl.exe	func@0x1401d0376
[Loop@0x530596e0 in func@0x5305986	6.453ms	20,000,000	0.625	windowscodecs.dll	[Loop@0x530596e0 in func@0x53

Fig.8 Hotspots by CPU utilization for rotation algorithm

Table.1 Gaussian kernel for size 9x9

0	0.000001	0.000014	0.000055	0.000088	0.000055	0.000014	0.000001	0
0.000001	0.000036	0.000362	0.001445	0.002289	0.001445	0.000362	0.000036	0.000001
0.000014	0.000362	0.003672	0.014648	0.023205	0.014648	0.003672	0.000362	0.000014
0.000055	0.001445	0.014648	0.058434	0.092566	0.058434	0.014648	0.001445	0.000055
0.000088	0.002289	0.023205	0.092566	0.146634	0.092566	0.023205	0.002289	0.000088
0.000055	0.001445	0.014648	0.058434	0.092566	0.058434	0.014648	0.001445	0.000055
0.000014	0.000362	0.003672	0.014648	0.023205	0.014648	0.003672	0.000362	0.000014
0.000001	0.000036	0.000362	0.001445	0.002289	0.001445	0.000362	0.000036	0.000001
0	0.000001	0.000014	0.000055	0.000088	0.000055	0.000014	0.000001	0



$$G(x,y)=\frac{1}{2\pi\sigma^2}e^{-(x^2-y^2)/(2\sigma^2)}$$

Fig. 9 Gaussian filter and temporal locality optimization for large kernels

Table.2 Convolution time for different kernel sizes

Kernel Size	Time in ms (no optimization)	Time in ms (after optimization)
9x9	155	153
15x15	415	299
51x51	3,960	1,900
101x101	15,160	10,700

In fig.9 the temporal locality of large kernels is analyzed to store these kernels in cache memory by different loop transformations based on the metrics collected by profiling tools. Table.2 show the results after improving temporal locality for different kernel sizes. It is clear that for small kernel sizes there is no big differences. While, this work is little different from other works, but comparing totally these results with the results in 3 and 4 , it is found that the proposed optimization in this work has achieved better results.

6. CONCLUSION

Direct implementation of image processing and computer vision algorithms without taking cache memory operation into account, leads to producing low performance and inefficient systems. Algorithm and code optimization achieved speed improvements from 30% to 70% compared with direct implementation. Profiling tools are helpful in determining latency causes, time hot spots, cache misses and hits, and other performance metrics. Using loop transformation, image titling and other algorithmic and implementation optimizations improve the cache hit ratio and increase the performance.

REFERENCES:

1. Gupta, S., & Zhou, H. (2015, September). Spatial locality-aware cache partitioning for effective cache sharing. In *2015 44th International Conference on Parallel Processing* , IEEE (pp. 150-159).
2. Kim, Y. G., & Kweon, I. S. (2013). Image-optimized rolling cache: Reducing the miss penalty for memory-intensive vision algorithms. *IEEE Transactions on Circuits and Systems for Video Technology*, *24*(3), 539-551
3. Pandey, A., Tesfay, D., & Jarso, E. (2018, January). Performance analysis of Intel Ivy Bridge and Intel Broadwell microarchitectures using Intel VTune amplifier software.

In *2018 2nd International Conference on Inventive Systems and Control (ICISC)* , IEEE ,(pp. 423-426).

4. Pesterev, A., Zeldovich, N., & Morris, R. T. (2010, April). Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems* (pp. 335-348)
5. Petko, S., Kudithipudi, D., & John, E. (2002, November). Cache performance of video computation workloads. In *Third International Workshop on Digital and Computational Video, 2002. DCV 2002. Proceedings.* IEEE, (pp. 169-175).
6. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, *48*(6), 519-530
7. Samdani, Q. G., & Thornton, M. A. (2000, September). Cache resident data locality analysis. In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No. PR00728)* , IEEE ,(pp. 539-546).
8. Tao, J., & Shahbahrami, A. (2008, September). Data locality optimization based on comprehensive knowledge of the cache miss reason: A case study with DWT. In *2008 10th IEEE International Conference on High Performance Computing and Communications*, IEEE, (pp. 304-311).
9. Tran, N. P., Choi, D. H., & Lee, M. (2014, August). Optimizing cache locality for irregular data accesses on many-core Intel Xeon Phi accelerator chip. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, IEEE, (pp. 153-156).
10. Toledo-Moreo, F. J., Martinez-Alvarez, J. J., Garrigos-Guerrero, J., & Ferrandez-Vicente, J. M. (2012). FPGA-based architecture for the real-time computation of 2-D convolution with large kernel size. *Journal of Systems Architecture*, *58*(8), 277-285
11. Weidendorfer, J., & Trinitis, C. (2005, May). Collecting and exploiting cache-reuse metrics. In *International Conference on Computational Science* , Springer, Berlin, Heidelberg. (pp. 191-198).
12. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2020, https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual accessed at 15/3/2020