Optimization For Algorithmic Intensive Systems On Constrained Embedded Systems

Mark Griffiths*

* Oulu University of Applied Sciences, School of Engineering, Oulu, Finland

t3grma00@students.oamk.fi

Abstract

In this paper I will experiment to see if there are some methodologies which can be incorporated into software which have intensive algorithmic operations and which must run often and quickly. The methodologies can be used anywhere and I will run the tests with large and small data structures, which are larger than the CPU's L1 cache.

This paper mainly focuses on an embedded systems L1 cache and how the design of this needs to be taken into consideration to avoid problems inherent to intensive operations, such as cache thrashing.

Keywords: *engineering, ARM, cache optimization, algorithm optimization,*

Introduction

All modern CPUs incorporate a hierarchical system of memory. The reason for this is to hide from the user the relatively high latency of the main memory accesses with respect to the performance of the CPU.

During algorithmic intensive operations, where the data structures are larger than the cache then data accesses and bandwidth issues may severely impact the efficiency of the execution of the algorithms.

Therefore, when designing algorithms of this nature concern needs to be given not just to the algorithm design but also to how these algorithms work in relation to the memory hierarchy.

In many cases, the compiler itself will provide many optimizations when running at -O2 and O3. Many modern CPUs, provide HW optimizations too, in the form of HW pre-fetching. This greatly eases the burden of the modern software engineer.

In this paper I will investigate if there are any other code changes a SW engineer can do in order to speed up their code..

Memory Caches

In most modern processors, there is a significant gap in performance between memory accesses that hit the cache and those that do not. Cache misses can take tens of cycles to resolve, cache hits return data within a few cycles. The most important improvements can be achieved by focusing on the L1 cache, as these caches are still very small, typically 32kb, although they can be 64kb. L2 caches are generally larger and most data structures can comfortably fit within these caches. Therefore most benefits in optimization occur by focusing on the L1 cache and that is what this paper focuses on.

In this paper, the ARM Cortex A9 will be used as the reference, although the procedures outlined here will work on any CPU, which has a hierarchical memory system.

Based on the Cortex A9, the examples given here assume a four-way set associative cache and a cache line length 32 bytes and 1024 of them. The Cortex A9, is frequently used in modern embedded systems and is found in Apple iPhones and Android devices, and so is relevant in many modern embedded systems. The general guidelines here can still be used, however some modifications to the actual examples may be needed.

In essence a L1 memory cache is a low latency memory block, which is used for accessing frequently used data. This low latency memory is still very expensive so, the L1 caches of modern CPUs still tend to be very small.

However, making use of this L1 cache and making sure the CPU does not need to get data from main memory frequently, which is significantly slower, can have a huge impact on the performance of software, in terms of speed of execution.

The relationship between the L1 cache and the main memory for a Cortex A9 is shown below.

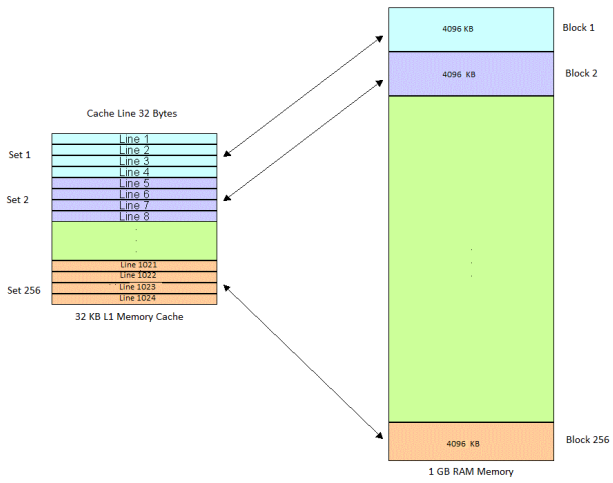

Figure 1 - The Cortex A9 Cache layout

More details on how exploiting the relationship between the main memory and the L1 cache will be explained later with cache thrashing.

Reusing the data that already resides in the L1 cache will speed up performance, making sure that data that is used together is stored together is known as data locality. As data is put into the cache on a cache line basis, data which is stored together will therefore be in the cache at the same time. A few techniques, which explore data locality can help in program performance are now explored.

Loop Interchange

Loop interchange essentially means changing the order of two iteration variables used in a nested loop. In the C programming language, array elements in the same row are stored consecutively in memory in row-major order. Following the standard matrix notation, rows are numbered by the first index of a two dimensional array and columns by the second. Accessing arrays that are contiguous in memory is usually faster than accessing elements which are not due to caching.

Loop Fusing

Loop fusing is almost always carried out by the compiler and so the SW developer does not need to pay too much attention to this.

Essentially if two loops have the same iteration parameters, these can be fused together in order to speed up the operations as they can be processed together..

Loop Tiling

Loop tiling partitions data into equal-sized subsets, which are known as tiles, so as to help ensure data used in a loop stays in the cache until it is reused. The partitioning of loop iteration space leads to partitioning of large arrays into smaller blocks, thus fitting accessed array elements into cache size, enhancing cache reuse and eliminating cache size requirements.

It is not always easy to decide what value of tiling size is optimal for one loop, experiments and some guesstimation based on the array regions in the loop and the cache size are the best way to get this value. The value of 16 was chosen when doing tests as four doubles can fit in one cache line and because the cache is four way associative.

The GCC flag is -floop-block to use this code transformation; GCC has to be configured with --with-ppl and --with-cloog to enable the Graphite loop transformation infrastructure.

Data Layout

These kinds of optimizations modify how structures and variables are arranged in memory. Like other cache optimisations, the point is to avoid cache misses as often as possible. In data layout optimisation this is done by taking into consideration the program's memory performance. Usually, memory performance is characterized in terms of the fault rates a program incurs on various memory resources.

Identifying the memory resources, which would be considered bottle necks in the program are best done using profiling tools. Some popular programs, which will do this are Cachegrind and Callgrind. The documentation for these programs is located in [6] and [7].

If data is commonly used together then it should be located together too in memory in order to utilise cache space. It should also be laid out in such a way that as much as possible of the data in each cache line loaded into the cache is used; data being used in the same code context should be placed contiguously in memory.

Reuse of cache lines comes in two distinct flavours: spatial reuse, where the accessed data is close to other data that has already been fetched into the cache, and temporal reuse, where the same data is revisited multiple times. Minimizing the time before the reuse will lower miss ratios.

Cache Thrashing

Cache thrashing on the Cortex A9 will only be expected with very large matrices that just happen to be the same size as the stride. As is explained with the following example:

```
double max = 8192; //8192*8 = 65536 bits.
double a(max), b(max), c(max), d(max), e(max);
        for( i = 1; i< max;i++)
          a(i) = b(i)*c(i) + d(i)*e(i);
```

A particular memory address cannot be loaded into an arbitrary cache line. Only one of the 256 sets can be used, but any of the 4 ways in the set can be used. We can calculate which set of cache lines to use for a particular memory address by the formula:

Set Number = (memory address/line size) mod (number of sets)

Where Line size = 256 kilobits
Number of sets = 256

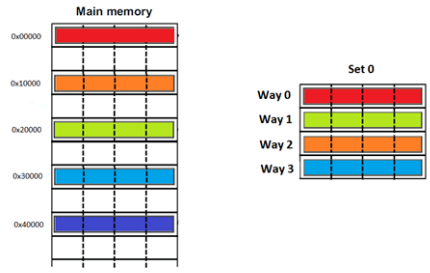

Figure 2 – The four way cache

Based on a 32bit address space, 32KB cache and 32 byte cache line, four way. We can represent the above example as memory addresses a[0] = 0x0000, b[0]=0x10000, c[0]=0x20000, d[0]=0x30000 and e[0]=0x40000.

The problem here is that the addresses in the RAM map exactly to one possible set. These addresses all belong to set number 0. There are only four cache lines in each set. The problem only occurs because the addresses are spaced a multiple of 0x10000 apart. This distance (for our cache) is known as the critical stride. Variables whose distance in memory are a multiple of the critical stride will contend for the same cache lines. The critical stride can be calculated as

$$(\textit{critical stride}) = (\textit{number of sets}) \times (\textit{line size})$$

$$65536 = 256*256 = 0x10000$$

So, our code goes into the same set:

a[0] =(0x00/0x100) mod 0x100 = 0. Set 0.
b[0] =(0x10000/0x100) mod 0x100 = 0. Set 0.
c[0] =(0x20000/0x100) mod 0x100 = 0. Set 0.
d[0] =(0x30000/0x100) mod 0x100 = 0. Set 0.
e[0] =(0x40000/0x100) mod 0x100 = 0. Set 0.

The Cortex A9 is four way set associative, so it can tolerate only four lines using the same address bits. In order to carry out the calculation one cache line must be displaced. When a(i) is to be calculated, b(i) will be thrown out. Then as the loop iterates, b(i) is needed and so another vector is displaced. In the example above, there is no cache reuse. This behaviour is known as cache thrashing and it results in poor performance, essentially reducing the program to un-cached use of memory. The cause is the alignment of all the vectors, they all map to the same cache location. Which of the ways is chosen depends on the cache replacement policy.

To solve this problem, you can introduce padding between the vectors in order to space out their beginning address. Ideally, each padding should be at least the size of a full cache line (32 bytes).

The above problem can be solved by using padding of 32 bytes. As shown below.

32bytes = 256bits = 0x100

int a(max), b(max), c(max), d(max),
pad1(0x100), e(max)

This would give:
a[0] =(0x00/0x100) mod 0x100 = 0. Set 0.
b[0] =(0x10000/0x100) mod 0x100 = 0. Set 0.
c[0] =(0x20000/0x100) mod 0x100 = 0. Set 0.
d[0] =(0x30000/0x100) mod 0x100 = 0. Set 0.
e[0] =((0x40000+0x100)/0x100) mod 0x100 = 1. Set 1.

For multidimensional arrays, it is enough to make the leading dimension an odd number.

Array Merging

The idea behind array merging is that it will combine two separate arrays into a single interleaved array. This will only be useful if the two blocks in the array might fight for the same block in the cache.

The result is that it brings together corresponding elements in both arrays, which are likely to be referenced together. The reduction in spatial locality reduces cache misses. Of course mainly a benefit if the two arrays are frequently used together.

IRQ & Cache Line Locking

When an interrupt occurs it is important for the SW that interrupt is handled as quickly as possible. Otherwise the CPU is held up and other interrupts can not be processed. If the IRQ handler is no longer present in the cache, then a cache miss will occur and the handler will need to be got from main memory. Hence slowing down the operation. On the A9, you can "lock" the IRQ into the cache, so that it will always be present. Of course by doing this you have less space for other data. However if your program has frequent interrupts then this may be one way in which you could speed up program execution.

Data Pre-fetching and Monitoring

After all algorithms and data structures have been optimised, there may still be some places in the code that cause cache misses that cannot be avoided. You can sometimes further improve performance by carefully adding pre-fetch instructions to make sure that this data is loaded into the cache before it is actually needed by the program, so that the program does not have to stall waiting for the data to arrive from memory.

Analysis tools are best used to determine the correct place to insert a pre-fetch instruction or how far ahead in the data structure to pre-fetch. If you insert a pre-fetch

instruction and reanalyze the program with the right analysis tools, they will provide feedback on whether the pre-fetch instruction is doing useful work and whether it is pre-fetching data too close to or too far ahead of the use of the data.

The GCC function __builtin_prefetch (*const void *addr, ...*) is used to minimize cache-miss latency by moving data into a cache before it is accessed. You can insert calls to __builtin_prefetch into code for which you know addresses of data in memory that is likely to be accessed soon. If the pre-fetch is done early enough before the access then the data will be in the cache by the time it is accessed.

The value of *addr* is the address of the memory to prefetch. There are two optional arguments, *rw* and *locality*. The value of *rw* is a compile-time constant; one or zero; one means that the prefetch is preparing for a write to the memory address and zero, the default, means that the prefetch is preparing for a read. The value *locality* must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it need not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. The default is three.

The Cortex-A9 data cache implements an automatic pre-fetcher that monitors cache misses done by the processor. This unit can monitor and pre-fetch two independent data streams. It can be activated in software using a CP15 Auxiliary Control Register bit.

Disable pre-fetch.
0 = disabled. This is the reset value.
1 = enabled.

When the software issues a PLD instruction the PLD pre-fetch unit always takes precedence over requests from the data pre-fetch mechanism. Pre-fetched lines in the speculative pref-etcher can be dropped before they are allocated. PLD instructions are always executed and never dropped.

However in many cases the HW pre-fetcher will do nothing where the SW could. This occurs where the HW is just not able to handle the logic and hence not predict well enough. Benchmarking on a case by case basis is needed to determine the best approach.

Results

Each test was run a hundred times and the mean average was taken. Here are the results with the optimization level at O0.

Name of Test	Matrix Size	Times /s
Tiling Small	100x100	0
Untiling Small	100x100	0
Tiling Large	1000x1000	0.15
Untiling Large	1000x1000	0.17
Loop Row First Small	100x100	0
Loop Column First Small	100x100	0
Loop Row First Large	1000x1000	0.05
Loop Column First Large	1000x1000	0.11

Here are the results with the optimization level at O3.

Name of Test	Matrix Size	Times /s
Tiling Small	100x100	0
Untiling Small	100x100	0
Tiling Large	1000x1000	0.09
Untiling Large	1000x1000	0.11
Loop Row First Small	100x100	0
Loop Column First Small	100x100	0
Loop Row First Large	1000x1000	0.04
Loop Column First Large	1000x1000	0.11

Conclusions

Due to the fact that the ARM Cortex A9 is four way associative and the fact it has a built in pre-fetcher make SW optimisations like the ones given in this paper not worth their while. The pre-fetcher can get the needed data from RAM before a cache miss occurs and so does not impact too much on the performance. If the procedures given here would be followed then the code would become more complex and this may lead to extra errors. So, for this reason the changes may cause more problems than they solve.

Many of the optimisations given here are also implemented as part of the GCC compiler and so this too eases the burden of the SW engineer.

Some things are important to follow, like using the column first for looping, also binding an IRQ may give performance improvements. One other thing to follow is that all optimisations like the ones given here should only be implemented after profiling tools have done their job.

References

[1] Cortex-A9 Technical Reference Manual, DDI 0388I
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388i/DDI0388I_cortex_a9_r4p1_trm.pdf

[2]Options that Control Optimization
http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[3] An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms
http://www.cc.gatech.edu/~bader/COURSES/UNM/ece637-Fall2003/papers/KW03.pdf

[4] Loop Tiling
http://en.wikipedia.org/wiki/Loop_tiling

[5] An Efficient Profile-Analysis Framework for Data-Layout Optimizations
http://research.microsoft.com/en-us/groups/rad/framework_popl.pdf

[6] Callgrind Manual
http://valgrind.org/docs/manual/cl-manual.html

[7] Cachegrind Manual
http://valgrind.org/docs/manual/cg-manual.html

[8] Cortex A Series Programmers Guide
https://silver.arm.com/download/Software/BX100-DA-98001-r0p0 01rel3/DEN0013D_cortex_a_series_PG.pdf

[9] Professional Embedded ARM Development
ISBN: 978-1-118-78894-3

[10] Chapter 6. Optimizing Cache Utilization
http://techpubs.sgi.com/library/dynaweb_docs/0640/SGI_Developer/books/OrOn2_PfTune/sgi_html/ch06.html#id14937

[11] Algorithms for Memory Hierarchies: Advanced Lectures
http://books.google.fi/books?id=Ok_T4PiwmUoC&pg=PA224&lpg=PA224&dq=Array+Merging+cache&source=bl&ots=hzLLElDbZw&sig=GpQ33fzXGggoZRdnyrrraykDlGk&hl=fi&sa=X&ei=utjwUprdOsnTtQbgmoHIAw&redir_esc=y#v=onepage&q=Array%20Merging%20cache&f=false

[12] Eliminate False Sharing
http://www.drdobbs.com/parallel/eliminate-false-sharing/217500206

[13] Array Padding Example
http://sc.tamu.edu/help/power/powerlearn/html/ScalarOptnw/sld017.htm

[14] Optimizing
http://www.agner.org/optimize/optimizing_cpp.pdf