C332 - Advanced Computer Architecture - CW2

Qiang Feng, Nikolay Smirnov, William Seddon
{qf316, ns4516, ws716}@ic.ac.uk

Abstract

In this report we will attempt to improve the performance of a `kdtree` program by identifying critical areas where speed-ups can be made. The areas we explore include parallelisation through threading, manual software prefetching, vectorised operations and compiler optimisations.

1 Hardware and Software Choices

We chose to use one of the DoC lab computers to run our benchmarks and optimised the `kdtree` program specifically for the machine. The machine has a quad-core Intel Core i7-6700 running at 3.4GHz with 16 GB of RAM which is running Ubuntu 18.04.1 LTS (Linux kernel version 4.15.0-43).

This particular processor organises its cache into 3 layers, with a 256 KiB, 8-way set associative L1 cache (split equally into two separate 128 KiB caches for the instruction and data). The L2 cache is 1 MiB in size, and is 4-way set associative. Finally, the L3 cache is 8 MiB in size and is a direct-mapped cache. In addition, as this processor is part of Intel's Skylake micro-architecture, there is a DCU prefetcher which fetches the next cache line into the L1-data cache [1]. We will further explore the impact of manual software prefetching.

2 Software Prefetching

In order to investigate cache efficiency for potential speedups, we start by running cachegrind with a program argument size of 10,000,000. From analysing the `cg-annotate` output, we find the largest source of data cache misses to be caused in `insert_rec` by the two lines:

```
new_dir = (node->dir + 1) % dim
if (pos[node->dir] < node->pos[node->dir])
```

Figure 1: Program lines associated with large amount of data cache misses

These two lines respectively miss the level 1 data cache 511,057,087 out of 927,663,988 times and 720,477,356 out of 2,782,991,964 times. After checking that the total number of level 1 data cache misses is 1,548,080,946, we can see that these two lines amount to 79.6% of all the level 1 data cache misses in the program.

In an attempt to investigate these caches misses further, we used Godbolt [2] to translate the C code into x86 assembly to view the cache access instructions. We can see the `mov` instructions that are accessing the memory at addresses referenced by offsets of the frame pointer register - and assume that these are the ones causing the frequent cache misses. However, the only way we saw to modify and improve the assembly was to add software prefetch instructions. Rather than writing assembly and using x86 instructions such as `PREFETCHT0`, we chose to look at C prefetch instructions for simplicity.

```
mov     rax, QWORD PTR [rbp-8]
mov     eax, DWORD PTR [rax+8]
add     eax, 1
cdq
idiv    DWORD PTR [rbp-48]
mov     DWORD PTR [rbp-12], edx
mov     rax, QWORD PTR [rbp-8]
mov     eax, DWORD PTR [rax+8]
cdqe
lea     rdx, [0+rax*8]
mov     rax, QWORD PTR [rbp-32]
add     rax, rdx
movsd   xmm1, QWORD PTR [rax]
mov     rax, QWORD PTR [rbp-8]
mov     rdx, QWORD PTR [rax]
mov     rax, QWORD PTR [rbp-8]
mov     eax, DWORD PTR [rax+8]
cdqe
sal     rax, 3
add     rax, rdx
movsd   xmm0, QWORD PTR [rax]
ucomisd xmm0, xmm1
jbe     .L9
```

Figure 2: Assembly code associated with large amount of data cache misses

With these observations, the first hypothesis we questioned was whether we could reduce cache misses and improve program speed using software prefetch instructions in C. Normally this strategy may be ineffective due to hardware prefetching or compiler software prefetches already doing this. However, we can assume that the Intel DCU prefetching hardware does not perform well here, due to the irregular pointer access patterns the tree structure creates making it difficult to fetch the next cache line accurately. We further assume that the current default compiler is failing to prevent the cache misses with software prefetching due to the fact that so many cache misses are occurring. As such, we introduce two GCC prefetch instructions before the cache accesses, as shown below.

```
static int insert_rec(struct kdnode **nptr, const double *pos, void *data, int dir, int dim)
{
  int new_dir;
  struct kdnode *node;

  /********** Prefetch 1 **********/
  __builtin_prefetch(*nptr, 0, 2);

  if(!*nptr) {
    if(!(node = malloc(sizeof *node))) {
      return -1;
    }
    if(!(node->pos = malloc(dim * sizeof *node->pos))) {
      free(node);
      return -1;
    }
    memcpy(node->pos, pos, dim * sizeof *node->pos);
    node->data = data;
    node->dir = dir;
    node->left = node->right = 0;
    *nptr = node;
    return 0;
  }

  node = *nptr;

  /********** Prefetch 2 **********/
  __builtin_prefetch(node->pos, 0 , 2);

  new_dir = (node->dir + 1) % dim;
```

Figure 3: Modified code with GCC prefetch instructions

We ran the code 5 times for statistical accuracy, with and without the prefetch instructions and a parameter size of 10,000,000 points.

	1	2	3	4	5	Average
Time (s) without prefetch	49.74	49.14	49.13	49.09	48.79	49.18
Time (s) with prefetch	43.88	43.28	43.25	43.89	43.73	43.61

Table 1: Software prefetching results with a parameter size of 10,000,000

As shown above, the software prefetching clearly reduces the runtime, and the average run time by 11.3%. This confirms our proposed hypothesis that software prefetching can improve performance for this program.

We then went a little further and used callgrind to profile the application, one of our main observations was that these cache misses contributed disproportionately to the total program running time. We then went back to cachegrind and saw that the `pos` array was causing cache misses. To help solve this we added a `__buitin_prefetch(pos, 0, 2)` line, the third argument of which is the "locality" (from 0 to 3, with 3 being very local) of the data we are prefetching. We chose the constant `2` because if we recurse again into `insert_rec` we need to use the `pos` array again, however we did not choose `3` as it is not clear if we will recurse, and we don't want to keep the `pos` array in cache if we don't need it. If we do not recurse then hopefully it will be flushed from the cache pretty quickly.

One of things we saw in callgrind is that calls to malloc take a long time. Unfortunately there is no easy way to speed up malloc; we decided against using jemalloc instead of glibc malloc partially because jemalloc is designed to improve performance and reduce fragmentation with highly parallel workloads; however k-d trees is not such a workload.

The program could be potentially improved further by more thoroughly investigating the optimal data and timing to prefetch.

3 Inlining `kd_insert()`

Reading the source code for `kdtreetest.c`, we can see that the `kd_insert3()` function is called `num_pts` number of times, which in our case is 10 million. When examining the `kd_insert3()` function source, we can see that it performs a function call to `kd_insert()`, which suggests that we have 5 million additional function calls, hence the obvious optimisation here was to inline the `kd_insert()` function inside `kd_insert3()` to reduce the number of function calls, shown in figure 4.

```
int kd_insert3(struct kdtree *tree, double x, double y, double z, void *data)
{
  double buf[3];
  buf[0] = x;
  buf[1] = y;
  buf[2] = z;

  if (insert_rec(&tree->root, buf, data, 0, tree->dim)) {
    return -1;
  }

  if (tree->rect == 0) {
    tree->rect = hyperrect_create(tree->dim, buf, buf);
  } else {
    hyperrect_extend(tree->rect, buf);
  }

  return 0;
}
```

Figure 4: `kd_insert3()` code snippet with inlined `kd_insert()`

Applying this optimisation yielded only a very small performance improvement - over a course of 10 runs with a problem size of 10 million, we achieved an average time of 45.738s before, and 45.469s after - a minuscule execution time decrease of 0.588%. At first we thought that `gcc` automatically inlines this function for us when using the `-O3` flag. To check, we used the `perf` tool to profile the call stack of our `kdtree` program[1]. However, as shown in figures 5 and 6, this was not the case. This lead us to believe that `gcc` deemed the potential performance improvement of inlining this particular function was not worth the increased size of the output executable.

[1] To get the call stack, we ran `perf record -call-graph dwarf ./kdtreetest 10000000` to first profile our program, then viewed the captured data using `perf report -call-graph -stdio -G`

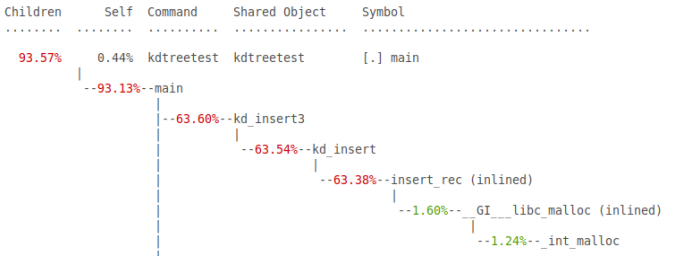

Figure 5: Call stack before manually inlining

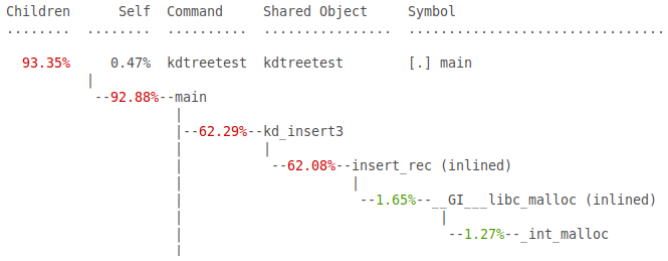

Figure 6: Call stack after manually inlining

Interestingly though, `gcc` did inline the `insert_rec()` function inside `kd_insert()`.

4 GCC Compiler Flags

We used `gcc` version 7.3.0, which is the default version installed for the version of Ubuntu we are running on the test machine. Note that by default, the only optimisation specific flag passed to `gcc` is setting the optimisation level to 3 (`-O3`).

Since our test machine has a Skylake i7-6700 CPU, which supports many extensions to the x86 ISA such as SSE and AVX we should be able to see performance improvement by specifying the target architecture as Skylake using the flag `-march=skylake`. This lead to a marginal decrease the average running time for a problem size of 10 million down from 43.094s to 42.892s (over a course of 10 runs).

To confirm this was down to architecture specific optimisations performed by `gcc`, we use the Godbolt Compiler Explorer [2] to view the generated assembly. A shown in figures 7 and 8, particular assembly instructions are replaced with the equivalent vectorised instructions (such as `movsd` being replaced with `vmovsd`, which is part of SSE) where deemed necessary by the compiler.

```
kd_insert3:
    unpcklpd xmm0, xmm1
    sub      rsp, 40
    mov      rdx, rsi
    mov      rsi, rsp
    movsd    QWORD PTR [rsp+16], xmm2
    movaps   XMMWORD PTR [rsp], xmm0
    call     kd_insert
    add      rsp, 40
    ret
```

Figure 7: `kd_insert3` assembly snippet before specifying target architecture

```
kd_insert3:
    sub      rsp, 40
    mov      rdx, rsi
    vunpcklpd xmm0, xmm0, xmm1
    mov      rsi, rsp
    vmovaps  XMMWORD PTR [rsp], xmm0
    vmovsd   QWORD PTR [rsp+16], xmm2
    call     kd_insert
    add      rsp, 40
    ret
```

Figure 8: `kd_insert3` assembly snippet after specifying target architecture

5 Parallelisation

Since the i7-6700 CPU follows a chip multiprocessor architecture with 4 cores, we decided to test the hypothesis of whether running the k-d trees program in different threads can speed up the execution speed. The reasoning behind this is that a chip multiprocessor has limited utilisation when running on one thread, and so by running the program over 4 threads, we can maximise utilisation of all 4 cores.

Having profiled the program using `callgrind`, we found that around 35% of the program run time was spent inside `kd_insert`. Therefore, we decided to thread the insertion section of the program. Our initial attempt involved creating `pthreads` for each point insertion in batches of 4, and then waiting for the 4 threads to join. However, running this with 10,000,000 points lead to a run time upwards of 100 seconds. As such, we decided to look for a more efficient way to thread the program.

We found the open-source C-Thread-Pool library on GitHub, which allowed us to create a threadpool with a set number of workers, which we configured to be 4. We then added work to the threadpool in each iteration of the original insertion `for` loop, and then wait for the threadpool to finish. The modified insertion for loop can be seen below. Since threads in C take a function to run and a void pointer containing parameters, we had to create and allocate memory to a structure for

each thread, containing the tree and data parameters for each thread to use. This is unfortunate as the `malloc` command significantly slows the program. In addition, to this, we can expect the other overhead of creating threads to also slow the program - as was the case without a threadpool.

```
threadpool thpool = thpool_init(4);
  for( i=0; i<num_pts; i++ ) {
    data[i] = i;

    struct insert_params *params;
    if(!(params = malloc(sizeof(struct insert_params)))) {
      perror("malloc_failed");
      free( data );
      return 1;
    }
    params->ptree = ptree;
    params->data = &data[i];

    thpool_add_work(thpool, insert_proc, params);
  }

  thpool_wait(thpool);
```

Figure 9: Modified tree insertion loop

However, despite the extra overhead of creating threads, we still found that threading point insertion created a significant speed up in the run time of the program, confirming our parallelisation hypothesis. The results below show the program being run with and without threading for 10,000,000 data points. Analysing the results, we find that the program runs 12.6% faster with the point insertion being threaded, a significant improvement. A similar speedup might be achieved by configuring a compiler instead to parallelise a program to run more efficiently on multiple cores.

	1	2	3	4	5	Average
Time (s) without threads	43.90	43.29	43.43	43.62	43.39	43.53
Time (s) with threads	38.32	37.81	37.71	38.41	37.9	38.03

Table 2: Threading results with a parameter size of 10,000,000

After the success in threading `kd_insert`, we tried to look for other potential parts of the program to thread. A potential function we found to thread was `find_nearest`. However, rather than being called in a `for` loop like `kd_insert` is, this function is recursive. This makes threading harder, and although possible, we then came across race conditions and found the runtime to be much slower. Thus, we decided this section of the code was not worth threading.

6 The Intel C Compiler

Intel's C Compiler (ICC henceforth) is the opposite of GCC: a closed source compiler which unashamedly tries to eek as much performance as possible out of Intel CPUs. This in the past included checking if the CPU was a "GenuineIntel", and therefore giving AMD/VIA CPUs a slower code path. We hypothesised that using some of the Intel C Compilers advanced performance features could help us improve the K-D tree performance. We were especially keen on using the reporting features of ICC, which can generate reports identifying which loops were vectorised and parallelised and why some where not. Our idea was to use some of the special options and reporting offered by the ICC to reduce the time taken.

In its default -O2 state we found that the ICC did not have that much to offer over GCC: both generated SIMD instructions, vectorised loops and had similar -O3 performance characteristics. We used both the Godbolt compiler tool to see some differences in interesting parts of the assembly code, which showed that ICC was slightly more aggressive in applying optimisations such as unrolling loops. Figure 10 shows the GCC output where branching is used to run a loop despite the number of iterations being known at compile time, whereas Figure 11 shows how the ICC unrolled this exact loop. While this example will not bring much performance improvement we found it interesting to see how the ICC was noticeably more aggressive than GCC.

```
.L59:
        movsd   xmm1, QWORD PTR [r15+rax]
        subsd   xmm1, QWORD PTR [rsi+rax]
        add     rax, 8
        mulsd   xmm1, xmm1
        addsd   xmm2, xmm1
        cmp     rax, 24
        jne     .L59
```

Figure 10: Loop assembly snippet not being unrolled with GCC

```
mov     rcx, QWORD PTR [r14]
movsd   xmm2, QWORD PTR [8+rsp]
mulsd   xmm2, xmm2
movsd   xmm3, QWORD PTR [rcx]
movsd   xmm0, QWORD PTR [8+rcx]
movsd   xmm1, QWORD PTR [16+rcx]
subsd   xmm3, QWORD PTR [r13]
subsd   xmm0, QWORD PTR [8+r13]
subsd   xmm1, QWORD PTR [16+r13]
mulsd   xmm3, xmm3
mulsd   xmm0, xmm0
mulsd   xmm1, xmm1
addsd   xmm3, xmm0
addsd   xmm3, xmm1
```

Figure 11: Loop snippet being unrolled with ICC

The power of the ICC comes when deciding to go beyond the default set of performance flags, which is what we decided to do. We started with `-march=skylake` [3] which we hypothesised would have a rather minimal impact as the GCC `-march=skylake` flags only had a marginal increase in performance. We found this to be the case, and sadly our Skylake CPUs did not support AVX512, because we found that setting `-march=skylake-avx512` generated quite a few AVX instructions which in theory would have sped up our program more (even though there are trade-offs between the clock speed boost possible and the amount of AVX512 instructions run).

We looked into how to improve vectorisation of loops using the Intel manual [4], however due to the recursive nature of the code it was difficult to apply the useful pieces of advice in the manual.

We attempted to see if ICC could parallelize some loops for us using `-parallel -par-threshold99` [5], our hypothesis was that some simple loops could be optimised in such as way. When benchmarking the difference we sadly saw that the impact was negligible, and when reading through the output assembly code it was clear to see why: there was no parallelisation taking place. The compiler was reordering assembly code and using different registers but we did not see any invocation of OpenMP, the library Intel uses to thread loops with ICC. We suspect that the loops that the Intel compiler found and could prove were safe to parallelise were not long enough to warrant the spinning up of a new thread.

Reading through Intel's official guidance on improving C performance with ICC we decided to use some of the suggested flags [3] [6] which helped speed up the program. Our final flags are as follows: `-std=c89 -Wall -march=skylake -ipo -fargument-noalias -qopt-report -qopt-report-phase=vec -flto -no-prec-div -qopt-prefetch=3 -prof-use -prof-dir=./ -O3 -DUSEMALLOCNOTALLOCA -DUSEOWNSQRT` . Some interesting ones include:

1. `-no-prec-div` which reduces floating point accuracy very slightly but increases performance
2. `-qopt-prefetch=3` which makes ICC issue prefetching instructions quite aggressively in order to prefetch as much as possible.
3. `-flto` which enables link time optimisations
4. `-prof-use` which makes ICC use the profiling information generated by running the program to focus on optimising the parts of the program that take the longest.

Overall the optimisations provided by the Intel C Compiler gave us benchmarking results of `40.876` over an average of 5 runs for a problem size of 10 million. In comparison with gcc with the `-O2` flag, which takes 43.775 seconds this is a 7% improvement.

7 Group Working

7.1 Allocation

We decided to parallelise the work for this report by each investigating different potential improvements. The contributions of each group member were:

- Nikolay - software prefetching and thread parallelisation
- Qiang - abstract, hardware and software choices, function inlining and GCC compiler flags
- William - the Intel C Compiler and profiling

7.2 Combining

We then attempted to merge all of our improvements into one executable program. While some investigations may be promising, we did expect some to perform not as well with other optimisations - for example for software prefetching to be less important when compiling with the ICC and using software prefetching flags.

8 Final Result

Unexpectedly, using the Intel C compiler along with the parallelisation optimisations actually decreased performance down from an average of 38.03s when compiling with `gcc`, to 40.21s (using `icc`). Hence, we decided to use `gcc` along with the optimal flags found[2] for this final benchmark. For a problem size of 10 million over a course of 10 runs, we achieve an average of 35.14s.

References

[1] V. Viswanathan, "Disclosure of h/w prefetcher control on some intel processors." https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, Sep 2014.

[2] M. Godbolt, "Compiler explorer." https://godbolt.org/.

[3] Yang-wang, "Step by step performance optimization with intel c++ compiler." https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler, 2014.

[4] "Intel 64 and ia-32 architectures optimization reference manual," *Intel Software*, no. 248966-040.

[5] "Automatic parallelization with intel compilers." https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers, 2014.

[6] "Quick reference guide to optimization intel c++ and fortran compilers v15." https://polyhedron.com/web_images//intel/productbriefs/Quick-Reference-Card-Intel-Compilers-v15.pdf.

[2] -march=skylake -flto