ASSIGNMENT 1

REPORT

Software Programming for Performance

Jivitesh Jain 2018101092

Tanmay Sachan 2018111023

Performance Analysers

Several performance analysers helped us study the performance of our code and the optimisations we applied. Here's a brief description of the ones we used.

Cachegrind

Cachegrind is a cache and branch-prediction profiler. It simulates how a program interacts with the machine's cache by running it in a sandboxed environment which simulates caching. It then measures the cache usage patterns of the program, the number of cache hits and misses and the spatial and temporal locality of the program.

Because of the difference in CPU and memory speeds, cache optimisation is an integral part of any optimisation attempt, and cachegrind helped us analyse our cache usage. For example, for mergesort, recursive mergesort clocked a miss rate of 0.3%, while iterative mergesort increased it to 0.6%. We could get the best of both worlds by using tiled mergesort, which uses an iterative approach in a different order and achieved a miss rate of 0.3%.

```
==28258== Cachegrind, a cache and branch-prediction profiler
==28258== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et
al.
==28258== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright
info
==28258== Command: ./test2
==28258==
--28258-- warning: L3 cache found, using its data for the LL simulation.
==28258== I   refs:      1,177,041,707
==28258== I1  misses:            1,298
==28258== LLi misses:            1,284
==28258== I1  miss rate:          0.00%
==28258== LLi miss rate:          0.00%
==28258==
==28258== D   refs:        399,933,170  (259,196,352 rd   + 140,736,818 wr)
==28258== D1  misses:        2,567,145  (  1,253,952 rd   +   1,313,193 wr)
==28258== LLd misses:        2,557,271  (  1,244,151 rd   +   1,313,120 wr)
==28258== D1  miss rate:           0.6% (        0.5%     +         0.9%  )
==28258== LLd miss rate:           0.6% (        0.5%     +         0.9%  )
==28258==
==28258== LL refs:           2,568,443  (  1,255,250 rd   +   1,313,193 wr)
==28258== LL misses:         2,558,555  (  1,245,435 rd   +   1,313,120 wr)
==28258== LL miss rate:            0.2% (        0.1%     +         0.9%  )
```

CACHEGRIND OUTPUT FOR ITERATIVE MERGESORT

Gprof

Gprof is a performance profiler tool that collects and arranges statistics on the program. It does this by inserting time-measuring macros at the start and end of each function call, generating a data file and then presenting the output in a human-readable form.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 76.85      4.62     4.62                             main
 23.56      6.03     1.42        1     1.42     1.42  matrix_multiply
. . .
                 Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.17% of 6.03 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    4.62    1.42                 main [1]
                1.42    0.00       1/1           matrix_multiply [2]
-----------------------------------------------
                1.42    0.00       1/1           main [1]
[2]     23.5    1.42    0.00       1         matrix_multiply [2]
-----------------------------------------------
```

GPROF OUTPUT FOR MATRIX MULTIPLICATION

Perf
Perf is another performance analysis tool that provides access to the Performance Monitoring Unit of the CPU, monitor software events and then generates reports based on them.

Clock_gettime
The `clock_gettime` function gets the time stored in the inputted clock and populates the `struct timespec`. It is used to time various parts of the program.

Problem 1:

The problem asks us to deal with matrix multiplication. We can use spatial locality to exploit the cache and get much higher running times.
On running the naive O(n^3) algorithm on our machine, we were getting a runtime of approximately 5.5 seconds.

Approach

We tried switching the loop orders to gain an improvement of about 2 seconds, bringing it down 3.3 seconds by exploiting the locality since data is stored in the row-major form on our machine.

Then we implemented the blocked tiling method, which uses strides to move the `i`, `j`, `k` of the outer loops, and multiply the smaller matrices generated by the inside loops.

```
for(i = 0; i < N; i += stridei){
      for(k = 0; k < N; k += stridek){
            for(j = 0; j < N; j += stridej){
                  // code goes here
                  // now we need to multiply two smaller matrices of sizes
                  // stridei X stridek and stridek X stridej
                  …
            }
      }
}
```

This decreased the runtime to about 1.5 seconds, by exploiting spatial locality.
Now to further reduces the runtime, we considered the approach to use the `i, j, k` loop order, and to preserve the locality while multiplying B, we decided to transpose B and multiply the row-major vectors obtained.

```
for(i = 0; i < N; i += stridei){
        for(j = 0; j < N; j += stridej){
            for(k = 0; k < N; k += stridek){
                for(ii = i; ii < min(i + stridei, N); ii++){
                    arow = a->matrix[ii];
                    for(jj = j; jj < min(j + stridej, N); jj++){
                        gotill = min(k + stridek, N);
                        tmp = 0;
                        brow = b->matrix[jj];
                        while(kk < gotill){
                            tmp += arow[kk] * brow[kk];
                            kk++;
                        }
                        c->matrix[ii][jj] += tmp;
                    }
                }
            }
        }
    }
```

Then we saw an opportunity in this code to use loop unrolling using strides of 16, which helped us bring down the runtime to roughly 1 second.

The final modification was the usage of register variables `arow`, `brow`, `ii`, `jj` and `kk`, since these variables were being accessed a lot.

Results
The final runtime was 0.85 seconds on our machine (Dell XPS 13, 128k L1D Cache, i7, 1.8GHz) with optimal values of `stridei`, `stridej` and `stridek`.

Problem 2

The problem requires us to optimise the mergesort algorithm using a single thread of execution.

Approach

One of the first things that come to mind while optimising mergesort is to convert the recursive calls to iterations. The recursion, though elegant, adds a significant function call overhead.
Hence we switched to an iterative approach in which at the ith iteration, sorted arrays of size 2^i are produced by merging arrays of size 2^(i-1) (which were sorted in the previous iteration). Thus the loop runs O(log(n)) times, with each iteration requiring a pass over the entire array, making it the same O(nlog(n)) mergesort. However, it removes the function call overhead.

```
 int i = 0;
  for (int cur_size = 1; cur_size < n; cur_size = cur_size << 1) {
      for (int left_start = 0; left_start < n - 1; left_start += (cur_size
<< 1)) {
          int x = left_start + cur_size - 1;
          int y = n - 1;
          int mid = y ^ ((x ^ y) & -(x < y));

          x = x + cur_size;
          int right_end = y ^ ((x ^ y) & -(x < y));

          if (i & 1){
              fast_merge(aux, arr, left_start, mid, right_end);
          }else{
              fast_merge(arr, aux, left_start, mid, right_end);
          }
      }
      i++;
  }
  if (i & 1) {
      for (int i = 0; i < n; i++) {
          arr[i] = aux[i];
      }
  }
```

Unfortunately, this approach disturbed the spatial locality of the program, as it repeatedly passes over the entire array instead of finishing off sorting elements in one part of it. Hence, the cache miss rate increased from 0.4% to 0.6%, negating the benefits of the reduced overhead.

```
==28258== D   refs:        399,933,170  (259,196,352 rd   + 140,736,818 wr)
==28258== D1  misses:        2,567,145  (  1,253,952 rd   +   1,313,193 wr)
==28258== LLd misses:        2,557,271  (  1,244,151 rd   +   1,313,120 wr)
==28258== D1  miss rate:           0.6% (        0.5%     +         0.9%  )
==28258== LLd miss rate:           0.6% (        0.5%     +         0.9%  )
```

To fix this, we switched to tiled mergesort, which basically is the same as iterative mergesort, except done in a different order. It first divides the entire array into arrays of size `CACHE_SIZE/2` (i.e, each array contains `CACHE_SIZE/8` elements). This allows each of these arrays and the corresponding auxiliary array to fit in the cache. Each of these arrays is then sorted using iterative mergesort. Finally, the arrays are merged just like iterative mergesort, starting from a size of `CACHE_SIZE/8`, and going upwards, doubling at each step.

```
   for (int left_start = 0; left_start < n - 1; left_start += 4096) {
       int x = left_start + 4095;
       int y = n - 1;
       int right_end = y ^ ((x ^ y) & -(x < y));

       iterative_mergesort(arr + left_start, aux, right_end - left_start +
1);
   }

   // phase II
   int i = 0;
   for (int cur_size = 4096; cur_size < n; cur_size = cur_size << 1) {
       for (int left_start = 0; left_start < n - 1; left_start += (cur_size
<< 1)) {
           int x = left_start + cur_size - 1;
           int y = n - 1;
           int mid = y ^ ((x ^ y) & -(x < y));

           x = x + cur_size;
           int right_end = y ^ ((x ^ y) & -(x < y));
```

```
        if (i & 1) {
            fast_merge(aux, arr, left_start, mid, right_end);
        } else {
            fast_merge(arr, aux, left_start, mid, right_end);
        }
    }
    i++;
}
if (i & 1) {
    for (int i = 0; i < n; i++) {
        arr[i] = aux[i];
    }
}
```

This effectively uses the L1D cache in the first phase, which is to the best extent possible. This helped us bring down the cache miss rate back to 0.3%, as reported by cachegrind.

```
==28175== D   refs:       399,958,866  (259,217,848 rd   + 140,741,018 wr)
==28175== D1  misses:       1,239,957  (    600,058 rd   +     639,899 wr)
==28175== LLd misses:       1,057,128  (    494,018 rd   +     563,110 wr)
==28175== D1  miss rate:          0.3% (        0.2%     +         0.5%  )
==28175== LLd miss rate:          0.3% (        0.2%     +         0.4%  )
```

Several mergesort implementations use the auxiliary array inefficiently, by copying to and from the auxiliary array at the end of each call to merge. Because iterative mergesort goes level by level, we could alternate between the two arrays, at each layer, reducing the number of copy operations to half.

On an analysis of the assembly code generated by GCC with the -O0 optimisation option, we realised that GCC had done a really inefficient job in managing variables and the stack. There were far too many unnecessary memory accesses, and a lack of register and variable reuse. Hence we wrote the entire merge function, which according to profiling results, takes the most amount of time, using inline assembly. This helped us achieve a 2x speedup, with the runtime of the program on a reverse sorted input array of 10^7 elements dropping from 0.6 seconds to 0.3 on our machine.
Next, we introduced insertion sorting for arrays of size less than 20, to reduce the overhead of copying elements. The insertion sort was written in assembly again, and helped us achieve moderate speedups.

Results
The final code runs in 0.091 seconds on our machine (Dell Inspiron 14 7000, 32k L1D Cache, i7-7500U at 2.7 GHz) for a random input array of size 10^6 containing a permutation of numbers from 1 to 10^6.