Practical Work - Project : Optimization and Profiling

Youness El Houssaini

***Abstract*— In software engineering, profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization. In this practical work we achieved profiling by instrementing a program source code of a 1024×1024 matrix product and its binary executable throughout the Linux perf command (called profiler). Compiling with "DCMAKE BUILD TYPE=RelWithDebInfo" mode allowed such combination and allowed us to detect the concrete part of the source code which is time consuming. We based our profiling on time stats based on several tests using from one to eight processors. The results are dependent on a casual 8 core laptop performance and are only used as indicators of the proposed improvements. The optimization is done on three steps, starting with enhancing the memory access by respecting the rule of contiguous memory blocks. Then, parralelizing the lower loop using OpenMP library, assuming that each product can be calculated seperatly and summing all results in the end using reduction clause of $\#$omp pragma. Finally, we improved the code by vectorizing the inner loop using the OpenMP SIMD pragma. In order to structure the optimized versions we use the conditionnal compilation. Each enhancement is defined with a given definition to simplify the process of testing. In that regard, we also made a shell script in order to run the profiler command over several processors. The results are as expected, when enhancing the memory access we gained around 94%, parallelizing made it a bit higher while the last version of vectorizing enhanced the gain to around 99% (i.e. 8 processors).**

I. Measure the performance of the unoptimize kernel

A. *Preleminary steps*

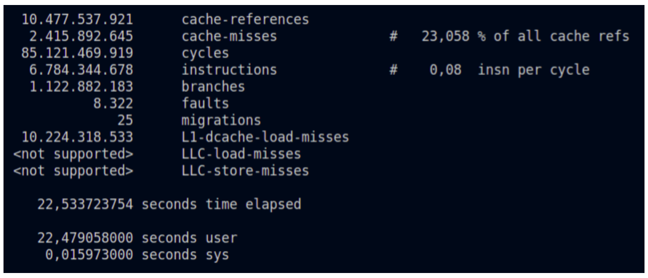
In previous practical works we compiled our codes using "DCMAKE BUILD TYPE=Release" or "DCMAKE BUILD TYPE=Debug" options. The first one is optimizing the compilation at maximum level while the second option is totally the oposite. In both cases, we don't have access to the name of functions. However, compiling with "DCMAKE BUILD TYPE=RelWithDebInfo" and using the "perf" command would allow a connection between the binary file and the code and thus an access to details about the source code functions. In fact, "perf" allow us to get some stats such as:

- cache-references: The number data query to the cache system.
- cache-misses: All cache misses (e.g. LLC,L1).
- cycles: Number of CPU cycles.
- instructions: Number of assembly instructions (many instructions take more than 1 cycle).
- branches: Number of branches (test/cmp).
- migrations: The number of times the process has migrated to a new CPU.
- L1-dcache-load-misses: Number of loads that are not served by the L1 cache (the smaller/faster cache).
- LLC-load-misses: Number of loads that are not served by the last level cache (the higher/larger cache).
- LLC-store-misses: Number of stores that are not served by the last level cache (the higher/larger cache).

For more information about perf stats, one can check the following reference [3].

B. *Get an overview*

After deploying "perf" repo with required packages and activating access to performance monitoring of my machine throughout: "sudo sh -c 'echo -1 > $/proc/sys/kernel/perf\_event\_paranoid\,setting$'", we get the following results :



```
   10.477.537.921      cache-references
    2.415.892.645      cache-misses              #   23,058 % of all cache refs
   85.121.469.919      cycles
    6.784.344.678      instructions              #    0,08  insn per cycle
    1.122.882.183      branches
            8.322      faults
               25      migrations
   10.224.318.533      L1-dcache-load-misses
  <not supported>      LLC-load-misses
  <not supported>      LLC-store-misses

     22,533723754 seconds time elapsed

     22,479058000 seconds user
      0,015973000 seconds sys
```

Fig. 1. First performances stats

The LLC-load/store-misses stats are not supported in my machine, which is not a big deal in my opinion as what matters to obserrve/inhance our code is the overall number of misses and the the lower cache L1 in particular. They are closer to use and thus provide quicker results when variables are hit (i.e accessed and not missed).

As expected, the given code is not optimized and the access to memory is expensive as shown by the big number of cache misses (i.e 2.5 billions). The whole matrix product calculation is done in about 22 seconds for a total number of cycles of around 79 billions cycles. The machine cycle is a 4 process cycle that includes reading and interpreting the machine language, executing the code and then storing that code.

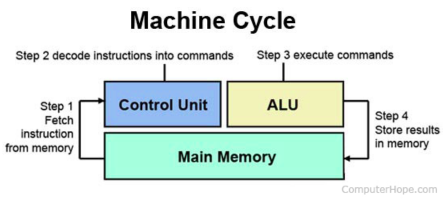
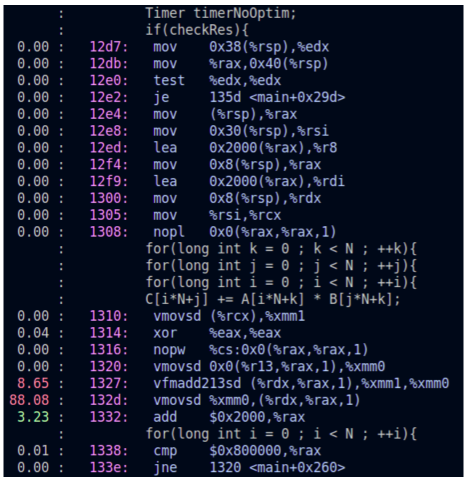

Machine Cycle
Step 2 decode instructions into commands
Step 3 execute commands
Step 1 Fetch instruction from memory
Control Unit
ALU
Step 4 Store results in memory
Main Memory
ComputerHope.com

Fig. 2. Machine cycle [1]

C. Detailed records

In order to avoid any ”perf” restrictions one can run as root. Thanks to the ”DCMAKE BUILD TYPE=RelWithDebInfo” compiling mode, it is possible to record the information related to the execution and to generate reports to see the result per function (or even per instruction). By running the command perf record -e cache-misses ./matrix -no-optim followed by perf annotate, we can look at the longest instructions in the given code.



```
       :            Timer timerNoOptim;
       :            if(checkRes){
  0.00 :   12d7:    mov    0x38(%rsp),%edx
  0.00 :   12db:    mov    %rax,0x40(%rsp)
  0.00 :   12e0:    test   %edx,%edx
  0.00 :   12e2:    je     135d <main+0x29d>
  0.00 :   12e4:    mov    (%rsp),%rax
  0.00 :   12e8:    mov    0x30(%rsp),%rsi
  0.00 :   12ed:    lea    0x2000(%rax),%r8
  0.00 :   12f4:    mov    0x8(%rsp),%rax
  0.00 :   12f9:    lea    0x2000(%rax),%rdi
  0.00 :   1300:    mov    0x8(%rsp),%rdx
  0.00 :   1305:    mov    %rsi,%rcx
  0.00 :   1308:    nopl   0x0(%rax,%rax,1)
       :            for(long int k = 0 ; k < N ; ++k){
       :            for(long int j = 0 ; j < N ; ++j){
       :            for(long int i = 0 ; i < N ; ++i){
       :            C[i*N+j] += A[i*N+k] * B[j*N+k];
  0.00 :   1310:    vmovsd (%rcx),%xmm1
  0.04 :   1314:    xor    %eax,%eax
  0.00 :   1316:    nopw   %cs:0x0(%rax,%rax,1)
  0.00 :   1320:    vmovsd 0x0(%r13,%rax,1),%xmm0
  8.65 :   1327:    vfmadd213sd (%rdx,%rax,1),%xmm1,%xmm0
 88.08 :   132d:    vmovsd %xmm0,(%rdx,%rax,1)
  3.23 :   1332:    add    $0x2000,%rax
       :            for(long int i = 0 ; i < N ; ++i){
  0.01 :   1338:    cmp    $0x800000,%rax
  0.00 :   133e:    jne    1320 <main+0x260>
```

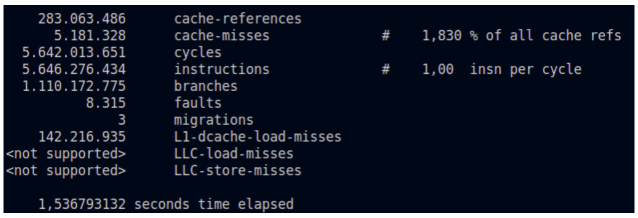
Fig. 3. Performance detailed stats

As expected, the nested loop is the cause of such bad performance. For each loop we can see in the right side how many percentage of the cache misses compared to the other loops and the final addition.

II. Improve the Kernel

A. The memory access

After reversing the indixes in the nested loop (i.e i and k) we obtain better results, as shown the figure below.



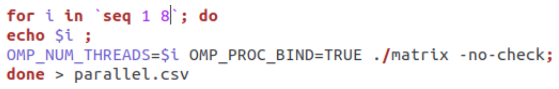
```
    283.063.486      cache-references
      5.181.328      cache-misses                #    1,830 % of all cache refs
  5.642.013.651      cycles
  5.646.276.434      instructions                #    1,00  insn per cycle
  1.110.172.775      branches
          8.315      faults
              3      migrations
    142.216.935      L1-dcache-load-misses
<not supported>      LLC-load-misses
<not supported>      LLC-store-misses

    1,536793132 seconds time elapsed
```

Fig. 4. Performance stats of the first improvement

Actually, The arrays can be imagined to be stored in contiguous way in the memory so for better performance the order of loops matters. The execution time fall down to 1.53 seconds, around 94% improvement. Can we obtain better throughout vectorization and parallelization?

B. Parallelize the code

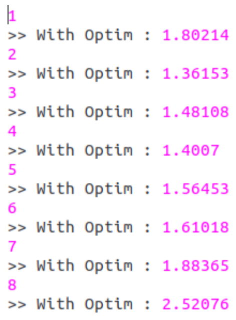
A matrix product can be divided in several dot products (i.e line $\times$ column). Thus, we can separate the computation of each dot product between multiple processors working independantly. We use the library OpenMP [2] and its omp pragma instructions. We execute `OMP_NUM_THREADS=i OMP_PROC_BIND=TRUE ./matrix -no-check` 8 times, where i will variate between 1 and 8. This is done throughout a script `par.sh` (see figure below) executed using first chmod +x par.sh then ./par.sh.



```
for i in `seq 1 8`; do
echo $i ;
OMP_NUM_THREADS=$i OMP_PROC_BIND=TRUE ./matrix -no-check;
done > parallel.csv
```

Fig. 5. Bash script for runnning several commands

The results are mainly satisfying but not decreasing at each additional processor parallelization. See figure below.



```
1
>> With Optim : 1.80214
2
>> With Optim : 1.36153
3
>> With Optim : 1.48108
4
>> With Optim : 1.4007
5
>> With Optim : 1.56453
6
>> With Optim : 1.61018
7
>> With Optim : 1.88365
8
>> With Optim : 2.52076
```

Fig. 6. Parallelization execution time for 1 to 8 processors

This unmonotone result might come from the fact that i am using a casual laptop with limited resources.

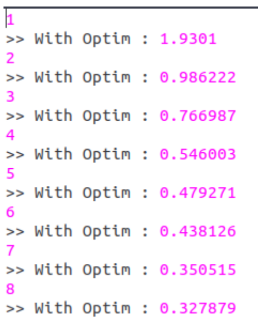
C. Vectorizing

For additional enhancement, we use the OpenMP SIMD pragma to vectorize the inner loop. We also do some tests with 1 to 8 processors using the same script (i.e par.sh).

The results are even better than the previous optimization. A final execution time of 0.32 seconds using 8 processors.

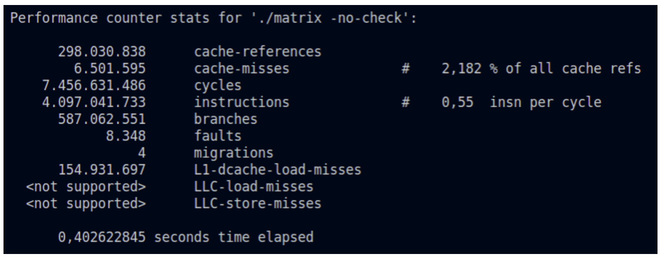
III. Conclusion

Using 8 processors, we gained around 99% compared to the badly implemented matrix product version by optimizing the memory access, parallelizing and then vectorizing. My machine is not a good demonstration of such optimization but still one can see the difference for a simple 1024×1024 matrix product. Here is a final overview of the perf stats regarding the proposed vectorized solution using only 2 processors.



```
1
>> With Optim : 1.9301
2
>> With Optim : 0.986222
3
>> With Optim : 0.766987
4
>> With Optim : 0.546003
5
>> With Optim : 0.479271
6
>> With Optim : 0.438126
7
>> With Optim : 0.350515
8
>> With Optim : 0.327879
```

Fig. 7. vectorization execution time for 1 to 8 processors



```
Performance counter stats for './matrix -no-check':

     298.030.838      cache-references
       6.501.595      cache-misses              #    2,182 % of all cache refs
   7.456.631.486      cycles
   4.097.041.733      instructions              #    0,55  insn per cycle
     587.062.551      branches
           8.348      faults
               4      migrations
     154.931.697      L1-dcache-load-misses
 <not supported>      LLC-load-misses
 <not supported>      LLC-store-misses

     0,402622845 seconds time elapsed
```

Fig. 8. Final performances

It is clear that we have less instructions, less cycles, less migration and most of all less cache misses. Using only 2 processors we reached en execution time of 0.4 seconds instead of 22 seconds for the bad version. Optimizing might take too much time from the developper but it is worth it!

References

[1] Machine cycle,https://www.computerhope.com/jargon/m/machcycl.htm.
[2] Openmp,https://www.openmp.org/wp-content/uploads/openmp-4.0-c.pdf.
[3] perf tutorial,https://perf.wiki.kernel.org/index.php/tutorial.