HPC Spring 2017 – Project 1

Robert van Engelen

Due date: March 7, 2017

1 Introduction

1.1 Login Procedure for this Project

You need an X11-capable machine to run Oracle Studio remotely. Oracle Studio is installed on the FSU RCC ”spear” cluster. To access the ”spear” cluster:

1. If you are connecting from a location not on campus, you must use VPN and set VPN AnyConnect to connect to `vpn.fsu.edu/hpc` with your RCC account credentials, see `https://rcc.fsu.edu/doc/off-campus-vpn-access` for details.
2. If you are currently using an X11-capable machine, then SSH to Spear with the command `ssh -Y yourname@spear-login.rcc.fsu.edu`. Option -Y is required to enable X11 forwarding, see `https://rcc.fsu.edu/doc/ssh` for details. To use X11 on a Mac, install XQuartz from `https://www.xquartz.org`. Start XQuartz and start a XQuartz X11 xterm terminal application from the XQuartz menu (do not use the Mac Terminal app).
3. If you are not using an X11-capable machine, then install the ”NoMachine” client on your machine, see `https://rcc.fsu.edu/doc/spear`. Use the NoMachine client app to log in your RCC account on Spear (I recommend the SSH protocol option, use host name `spear-login.rcc.fsu.edu`, and check that the NoMachine login option is selected in Advanced settings). When using NoMachine to connect, start a KDE virtual desktop. In case of trouble, consult the RCC documentation (don’t forget to use VPN to log in remotely, when you’re away from our campus).
4. Add the Oracle Studio (a.k.a. Sun Studio) compiler tools to your path by running `module load sunstudio`. Add this command to your login script, e.g. `.tcshrc` so it will run each time you log in.

1.2 Download the Project Files

Download the project source code:

```
[yourname@spear ~]$ wget http://www.cs.fsu.edu/~engelen/courses/HPC/Pr1.zip
```

The package bundles the following files:

- `Makefle`: a standard Makefile to build the project.
- `config.guess`: determines the platform
- `make.`*platform-comp*: platform- and compiler-specific files used by `Makefile`
- `timing.sh`: a script used by `make plot`, see Section 9
- `global.h`: global definitions
- `bench.c`: a benchmark wrapper program to time `sqmat_mult()` square matrix multiply using random matrices of varying dimensions
- `cputime.h` and `timeres.c`: `cputime()` timer
- `rdtsc.h`: Intel RDTSC timer used by `cputime()`
- `timeres.h` and `timeres.c`: determine timer resolution
- `timer.c`: timer precision test
- `sqmat.c`: simple version of square matrix multiply in C
- `sqmatb.c`: blocked version of square matrix multiply in C (incomplete)
- `sqmatf.f`: simple version of square matrix multiply `SQMULT` in Fortran (incomplete)
- `sqmatfc.c`: wrapper to invoke Fortran `SQMULT` from C
- `sqmatw.c`: Winograd version of square matrix multiply (incomplete)
- `sqblas.c`: wrapper to invoke BLAS3 DGEMM (incomplete)

1.3 Project Objectives

By completing this project you will be able to

- investigate the accuracy of timers for benchmarking and timing experiments.
- use advanced profiling techniques to identify performance issues and relate these to the source code.
- use compiler optimizations and compiler hints via program annotations to improve performance.
- compare and understand the performance differences of numerical programs written in C versus Fortran.
- apply loop blocking techniques to improve performance.
- use BLAS DGEMM libraries.
- understand the impact of algorithmic differences by implementing an alternative formulation of matrix multiply using Winograd's algorithm.

1.4 Write a Project Report

Write a report of your findings and submit this to the instructor for grading. When applicable, include performance graphs and explanations of your findings in your report. Your report should address the parts listed in the sections below and also list the source code of the programs that you wrote or modified. You should only submit one report. For this project you do not need to submit your source code separately for evaluation.

2 Determine Machine Timer Accuracy

The `cputime()` function defined for you in `cputime.h` and `cputime.c` returns the CPU time or wall-clock time in seconds, which is measured from the previous call to this function to the next[1]. We use the `cputime()` function to determine the number of floating point operations

[1]Important: You cannot invoke `cputime()` from multiple threads in a multi-threaded application because it is not thread safe. This is not a problem in this assignment in which all of our code is single threaded.

per second (`MFlops` “megaflops”) of our square matrix multiply routine `sqmat_mult()`, by measuring the elapsed time of k calls to `sqmat_mult()`:

$$\texttt{MFlops} = \frac{2kn^3}{10^6 \cdot \texttt{cputime}} \tag{1}$$

where n is the matrix dimension used in a benchmark test and $2n^3$ floating point operations are needed for the matrix multiply. We chose `MINRUNS`=2 and `MINSECS`=0.1 as defined in `bench.c` to force at least two runs and an elapsed time of 0.1 seconds. The benchmark driver automatically adjusts the number of runs k of the benchmark code to meet these constraints.

What follows are some helpful observations about MFlops.

> As a measure of performance we use MFlop/sec (or MFlops). MFlops is actually a measure of *useful work* performed, i.e. we don’t count integer arithmetic and address calculations needed to access the arrays. Though MFlops is a really a misnomer, because it puts the emphasis on the floating point operations, not the speed by which the output data is obtained. In fact, slower algorithms with a high MFlop count may look better than faster algorithms with a lower MFlop count if we are not careful to define a fair MFlop formula.
>
> For example, suppose that we optimize the square matrix multiply algorithm for the case of diagonal matrices. A check for diagonality takes n^2 comparisons and computing the result takes only n floating point operations. Thus, if we would compare the performance of the optimized algorithm to the naïve algorithm based on Flop counts per second, then the naïve algorithm would always win! But that would not be desirable as it is very slow in this case.
>
> Because algorithms have different floating point operation counts, the performance of an algorithm with fewer Flops could look worse even when the algorithm runs in the same time or faster. Therefore, it is more fair to keep using the general MFlops formula for matrix multiply (1) and use it as a *scaled measure of performance* of matrix multiply, which is only dependent on the data size n and `cputime`, and not dependent on the actual operations performed, where n is the matrix dimension and k is the number of benchmark runs that are timed.

To investigate timer accuracy, for example to determine execution times and to calculate MFlops, follow these steps to test the `TIMES` timer:

- Log in and start an xterm.
- Chdir to the directory where you installed the content of `Pr1.zip`
- Do `make timer` to build a timer test program.

- Run `./timer`

You will get:

```
It took ... iterations to generate a nonzero time on platform ...
compiled with ...
Timer resolution is  0.010000000 seconds
With MINSEC=0.1 timing precision is at least 1 digit(s)
```

The precision of the default timer is only one digit when our benchmarks ran for 0.1 second! By keeping `MINSECS` small we avoided very long waiting times for the benchmarks to complete. However, if the resolution of `cputime` is too low then the timing precision in digits p_{digits} might be insufficient, since:

$$p_{\text{digits}} \geq \lfloor \log_{10}(t/r) \rfloor \tag{2}$$

where r is the resolution of `cputime()` in seconds and t_{sec} is the elapsed time in seconds. When $t =$ `MINSECS` $= 0.1$ and $r = 0.01$ in Eq.(2) we have $p_{\text{digits}} = 1$.

To ensure a reasonable accuracy of our timings we can increase `MINSECS` or use a timer with a higher resolution. This should be done with care since high-resolution timers tend to roll over when the maximum elapsed time that they can represent is exceeded. Timers have a fixed bit-width (e.g. 32 bit or 64 bit) that limits the maximum elapsed time that can be represented.

Investigate the applicability and resolution of the following timers defined in `cputime.c` enabled with the `-D` option:

- `-DUSE_TIMES`: uses `times()` to obtain CPU time (user + system time).
- `-DUSE_GETRUSAGE`: uses `getrusage()` to obtain CPU time (user + system time).
- `-DUSE_GETTIMEOFDAY`: uses `gettimeofday()` to obtain wall-clock time.
- `-DUSE_RDTSC`: uses the Intel RDTSC instruction to obtain a very high resolution wall-clock time. This is not portable: only available with Intel IA32 and IA64 and may not work on Spear. Also, RDTSC may not be accurate on multicore processors, since these may have RDTSC clocks per core and a context switch to another core gives a different readout. To avoid this we need to lock the task's thread affinity with a core.

To test a timer, modify the `make.x86_64-unknown-linux-gnu-suncc` make file by setting macro flags `CMFLAGS` to one of the `-D` compiler options listed above. Then recompile and run with:

```
[yourname@spear ~]$ make clean timer
[yourname@spear ~]$ ./timer
```

For each timer tested, report its resolution and precision. **Always make sure you do a clean build as shown above after editing the make file.**

Report the timer that gives the best wall-clock time accuracy and works naturally in a multi-threaded machine. We will prefer wall-clock time over CPU time, especially when we deal with parallel programs as we will do in the next project. **You should use that timer for all timing experiments from now on**.

Make sure the usage load on the machine on which you are benchmarking is low. To check the CPU usage of users use the `top` command on Spear. Rerun your experiments at different times if necessary.

3 Profiling

Profiling is an important aid to determine hot spots and bottlenecks for performance in your code. We will use hardware counter profiling to determine the impact of a matrix multiply loop interchange with respect to memory access.

In an xterm execute

```
[yourname@spear ~]$ make clean sqmat
[yourname@spear ~]$ collect -h cycles,on,dcr,on,dcm,on -o sample.er ./sqmat
[yourname@spear ~]$ analyzer sample.er
```

and select the "Source" tab. Then select "Metrics" from the menu bar and enable hardware counters "CPU Cycles", "L1 D-Cache Refs", and "L1 D-Cahce Misses". What is the CPU cycle count of line 16 and the and the L1 D-cache miss ratio (misses / refs) for line 16?

Note 1: compilation fails if Oracle Studio was not enabled with `module load sunstudio`

Note 2: `analyzer` requires X11, which fails if you're not using an xterm or NoMachine.

Note 3: `collect -h` lists the hardware counter profiling options. The three counters used in our experiment are:

- `cycles` measures CPU cycles
- `dcr` measures L1 D-cache references

- `dcm` measures L1 D-cache misses

Note that the matrix multiply loop next ordering is from outermost to innermost j, i, k:

```
for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
        for (k = 0; k < n; k++)
            C[j*n + i] = C[j*n + i] + A[k*n + i]*B[j*n + k];
```

As you can see, the A elements are accessed with a large stride in the innermost k loop, thus spatial locality is poor. The goal is to modify the loop to improve this access pattern. For the same reason it would not make sense to pick j as the innermost loop. Rather, we should pick i for the innermost loop:

```
for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
        for (i = 0; i < n; i++)
            C[j*n + i] = C[j*n + i] + A[k*n + i]*B[j*n + k];
```

Change the code as shown and in an xterm execute

```
[yourname@spear ~]$ make clean sqmat
[yourname@spear ~]$ rm -rf sample.er
[yourname@spear ~]$ collect -h cycles,on,dcr,on,dcm,on -o sample.er ./sqmat
[yourname@spear ~]$ analyzer sample.er
```

and select the "Source" tab. What is the CPU cycle count of line 16 and the and the L1 D-cache miss ratio (misses / refs) for line 16? Was there any improvement in this ratio given that we changed the memory access pattern in the loop nest?

Explain the impact of the loop change based on the profiling results. What is the FP:M ratio of the code that spans the code of the innermost loop and its body[2]? How did the change improve the spatial and/or temporal locality of memory access?

From now on for the remainder of the project, we will use the reordered loop nest with i as the innermost loop, a decision based on the results of our experiment.

[2]do not count FP:M over the outer loops, since the performance is largely determined by the inner loop, so the performance of the inner loops dominates.

4 Compiler Optimizations

In the previous experiment we solely focussed on data access pattern optimization, without any compiler optimizations to improve code execution. We will combine data access and compiler optimizations to improve performance further.

Compiler optimizations differ from platform to platform. Instruction scheduling (such as modulo scheduling) is especially important for VLIW, EPIC (Itanium), and RISC architectures. Loop restructuring is especially important to optimize code. Loop restructuring requires dependence testing to verify the absence of cross-iteration dependences (other than the dependences of the loop counter variables). If the compiler cannot prove absence of a cross-iteration dependence, a (nested) loop cannot be reordered for optimization. The onus of the proof is on the compiler. If it cannot disprove dependence, the loops are not optimized. Hints provided by the programmer in the code or as compiler options can help. In this part of the assignment we will experiment with compiler optimizations and program annotations to speed up matrix multiply.

In an xterm execute (this may take a while):

```
[yourname@spear ~]$ make clean plot
[yourname@spear ~]$ gnuplot -persist timing.gnuplot
```

This can take some time. The MFlops plot for sqmat is shown (errors will be shown and plots for the other programs do not show up, which is normal as we will work on them later). Save the graph for your report with a screen capture or edit `timing.gnuplot` and modify to `set term png; set output "myplot.png";` (see also Section 9). Rerun `gnuplot timing.gnuplot` to generate a PNG file.

This plot shows the baseline performance of sqmat, without optimizations, compared to an optimized BLAS-based version sqblas. The plot should be in your report. Determine the performance difference between the two (how many times is sqmat slower than sqblas, for which problem "Dim"-ension(s), i.e. matrix rank(s)?).

Change the `make.x86_64-unknown-linux-gnu-suncc` by adding the `-fast` optimization option for the C and Fortran compilers, `COFLAGS` and `FOFLAGS`, respectively. This option is similar to `-O3` to optimize code. Then generate a new performance plot. **Always use `make clean plot` when modifying the makefiles and program sources.**

Put the new plot in your report and estimate the speedup obtained with the `-fast` option compared to the baseline performance.

In the xterm execute `er_src sqmat`. The Oracle Studio `er_src` command lists the source code and optimizations that have been applied to it, if any. What optimizations have been applied? Which optimization(s) do you believe have contributed significantly to the performance increase?

The compiler appears to perform optimization under "dynamic-alias-disambiguation", which means the code is optimized by checking for the absence of aliases at runtime. We can either add `restrict` qualifiers to the arguments as follows:

```
sqmat_mult(const double *restrict A,
           const double *restrict B,
           double *restrict C,
           int n)
```

which is safe, since we do not use this function with overlapping input/output arrays (in fact, `A` and `B` are still permitted to overlap or even be the same because there are no flow/anti dependences with `C`, since `A` and `B` are input only!). Or we can ask the compiler to figure out if there are any aliases based on structure layout using the optimization flags `-xrestrict -xalias_level=layout`. Add these to `COFLAGS` and `FOFLAGS`. It is fine to do both (use `restrict` in code and compiler options).

Generate a new plot for and determine if any speedups were obtained with "`restrict`" compared to the previous experiment.

In the xterm execute `er_src sqmat`. What optimizations have been applied? Which optimization(s) do you believe have contributed significantly to the performance increase?

Edit `sqmat.c` to modify the test sizes as follows for three profiling experiments we will conduct:

- Small : `int sqmat_test_size[] = { 50, 60, 70, 80, 90, 100 };`
- Medium: `int sqmat_test_size[] = { 200, 300, 400, 500, 600, 700, 800 };`
- Large: `int sqmat_test_size[] = { 900, 1000 };`

For each of the three size experiments Small, Medium, and Large, profile the `sqmat` program with hardware counters. For each experiment, determine the CPU cycle count, the L1 D-cache miss ratio (cache misses / refs) and FPU stall ratio (FPU stall cycles / total CPU cycles) for line 16. (Use `collect -h` to list the available hardware counters to use as options, up to 4 counters can be used per experiment).

From now on, we will continue to use the `-fast`, `-xrestrict -xalias_level=layout` flags for C (`COFLAGS` in the make file) and the `-fast` flag for Fortran (`FOFLAGS` in the make file).

5 Fortran vs C

Fortran and C code compiles differently, mainly because of the underlying programming language properties that the compiler can exploit. A Fortran compiler that translates Fortran to C first to compile the resulting C code loses some of these properties along the way.

Edit `sqmatf.f` and complete the function's implementation of square matrix multiply. For the multiply loop, use the loop nest ordering `j` (outer), `k` (middle), and `i` (inner) loop.

In an xterm execute

```
[yourname@spear ~]$ make clean plot
[yourname@spear ~]$ gnuplot -persist timing.gnuplot
```

Use `er_src sqmatf` to take a look at function `sqmat_`. Explain what the compiler has done here since it refers to a library function `__f95_dgemm_` (see also next section).

To understand the raw differences between C and Fortran without dgemm, try a different loop ordering in `sqmatf.f` to compile and run as a separate experiment and report what happens in that case, i.e. compiler optimizations and timings.

6 BLAS Level 3: DGEMM

BLAS Level 3 is an efficient implementation of matrix-matrix linear algebra operations. Several implementations are available by vendors and as open source.

Determine how to use DGEMM in C by perusing the online manual pages of `dgemm`. Edit `sqblas.c` to call `dgemm` to perform a square matrix multiply.

Run the experiment with `make clean plot` and `gnuplot -persist timing.gnuplot`.

Add the plot to your report. From the plots, compare performances of the `sqmat`, `sqmatf`, and `sqblas` benchmarks.

7 Blocking

Loop blocking can be very effective to enhance performance by increasing the locality of memory access. Blocking improves locality of memory accesses to speed up the multiplication of large matrices. It is tricky however to find a good block size.

The block multiply is performed as follows:

```
for (i = 0; i < n; i += BLKSIZE)
  for (j = 0; j < n; j += BLKSIZE)
    for (k = 0; k < n; k += BLKSIZE)
      block_mult(A, B, C, i, j, k, n);
```

where `block_mult` multiplies $C = A\,B$ block-wise for each `BLKSIZE`$\times$`BLKSIZE` block located at $C_{i,j}$, $A_{i,k}$, and $B_{k,j}$.

Edit `sqmatb.c` and implement a blocked version of matrix multiply. More information can be found in the file itself. Show your code in your report.

Determine a block size `BLKSIZE` that works well. Report how you found your block size.

Run `collect -h dcr,on,dcm,on,l2dr,on,l2dm,on -o sample.er ./sqmat` to investigate the L1 and L2 cache miss ratios for `sqmat` in the hotspot loop that assigns `C` in `sqmat`. Repeat this for `sqmatb` to analyze the hotspot loop that assigns `Cb` in `sqmatb`. Report the profile results for `sqmat` and `sqmatb`.

There are certain matrix sizes for which the `sqmatb` code runs faster or slower as you can see from spikes in the performance graphs. What is a possible explanation of the slower parts and what could be causing these spikes?

8 Winograd's Algorithm

There are several algorithms that (theoretically) improve the speed of matrix multiply, such as Winograd's algorithm and Strassen's algorithm.

Winograd proposed the following formulas (here rewritten for $n \times n$ square matrices):

$$\begin{aligned}
x_i &= \sum_{k=1}^{\lfloor n/2 \rfloor} A_{i,2k-1}A_{i,2k} \\
y_j &= \sum_{k=1}^{\lfloor n/2 \rfloor} B_{2k-1,j}B_{2k,j} \\
C_{i,j} &= -x_i - y_j + \sum_{k=1}^{\lfloor n/2 \rfloor} (A_{i,2k} + B_{2k-1,j})(A_{i,2k-1} + B_{2k,j}) \\
&\quad + A_{i,n}B_{n,j} \qquad \textbf{if } n \text{ is odd}
\end{aligned}$$

The number of floating point operations performed in an $n \times n$ square matrix multiply is $2n^3$ (one add and one multiply per iteration of the $n \times n \times n$ loop). Winograd's method uses $2n^3 + 3n^2$ operations when n is even and $2n^3 + 5n^2$ when n is odd, but with only half the number of multiplications.

Implement Winograd's algorithm in Fortran in `sqmatw.f`.

Run the experiment with

```
[yourname@spear ~]$ make clean plot
[yourname@spear ~]$ gnuplot -persist timing.gnuplot
```

It is actually more important to find an algorithm with a better FP:M ratio than to reduce the total floating point operations, since memory access patterns have a critical impact on performance. Reducing the total number of floating point operations should also reduce the number of distinct memory locations referenced, but may not improve the memory access patterns. Determine the FP:M ratio of the basic square matrix multiply and compare it to the FP:M ratio of your implementation of Winograd's algorithm.

9 Plotting Help

To plot the data files of a set of benchmark programs, run `./timing.sh` *prog1 prog2* . . . followed by `gnuplot -persist timing.gnuplot`.

To produce PNG graphics files of the plots for your report, edit `timing.gnuplot` and change the first part to:

```
    set term png; set output 'myplot.png'; set grid; set xlabel 'Dim'; ...
```

Then run `gnuplot timing.gnuplot` to create the `myplot.png` file.

- *End*