

HPC Fall 2007 – Project 1

Fast Matrix Multiply

Robert van Engelen

Due date: October 11, 2007

1 Introduction

1.1 Account and Login

For this assignment you need an SCS account. The account gives you access to `pamd`, `prism006`, and `tempest`. Contact the instructor if you do not have an SCS account. To login to `prism006` and `tempest` from outside SCS you must first `ssh` to `pamd` (using `ssh name@pamd.scs.fsu.edu`) and then `ssh` to `prism006` or `tempest`. To login to a specific `tempest` node, e.g. `tempest04`, use `ssh tempest04`. For benchmarking it is best to find a lightly used node (run `top` to see how busy the machine is and run `users` to see how many users are logged in). Note that wall clock timings are affected by all other processes that share the CPU. Some timers (such as the UNIX `time` command) provide CPU user and system time of a process, not wall clock time.

1.2 Setup

If you use `tcsh` as your main shell then edit your `.tcshrc` file in your home directory, or edit the `.cshrc` file if you use `csh`. Add the following lines:

```
# Set paths to Intel compilers on prism006
setenv INTEL_CC /opt/intel/cce/9.1.047
setenv INTEL_FC /opt/intel/fce/9.1.043
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$INTEL_CC/lib
setenv MANPATH ${MANPATH}:$INTEL_CC/man
```

```
set path = ( $INTEL_CC/bin $INTEL_FC/bin $path )
# Set paths to Sun compilers
setenv MANPATH ${MANPATH}:/export/sun/sunstudio12/man:/opt/SUNWspro/man
set path = ( /export/sun/sunstudio12/bin $path )
```

1.3 Download

Next, download the project source code from

<http://www.cs.fsu.edu/~engelen/courses/HPC/Pr1.zip>

The package bundles the following files:

- `Makefile`: 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 6
- `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
- `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
- `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`

1.4 Getting Started

To get started with this assignment, follow these steps:

- Open a terminal and login to `prism006`
- Run `make` to build the binaries using the GNU C and Fortran compilers
- Run `./sqmat` which should produce a list of MFlop/s results for a series of $n \times n$ square matrix dimensions n
- Run `make plot` which executes a set of benchmark programs (this can take a while), with all incomplete programs generating “max error tolerance exceeded” (you can ignore this for now)
- Run `gnuplot -persist timing.gnuplot` to view the results

Note that the results of the incomplete programs and the BLAS results are not shown in the graph. BLAS results are only produced when the `sqblas` test is built with the Sun compiler. You are responsible for finishing the incomplete programs as explained in subsequent sections.

To verify the package content, run `make COMP=icc` and `make COMP=suncc` to build binaries on `prism006` with the Intel compilers and Sun compilers, respectively. But before you do so, please see the note below.

1.5 An Important Note

Whenever you decide to change compilers for tests, you **must** run `make clean` to clean up the builds before recompiling the codes. Whenever you change the content of the `make.platform-comp` files, the `Makefile`, or `.h` header files you **must** also run `make clean` before recompiling.

1.6 Your Assignment

Your assignment is subdivided in several parts: investigate CPU timers and their proper use for benchmarking (Section 2), use advanced compiler optimizations to improve performance of matrix multiply (Section 3), implement a blocked version of square matrix multiply (Section 4), and implement Winograd’s matrix multiply (Section 5).

Write a report of your findings and submit this to the instructor for grading. Include performance graphs and explanations of your findings in your report. Your report must address all

tasks listed in the sections below and also list the source code of the algorithms you wrote. You do not need to submit your source codes.

2 Timers

The `cputime()` function defined in `cputime.h` and `cputime.c` returns the elapsed time in seconds, which is measured from the last call to this function¹. We use the `cputime()` function to determine the number of floating point operations per second (MFlops “megaflops”) of our square matrix multiply routine `sqmat_mult()`, by measuring the elapsed time of k calls to `sqmat_mult()`:

$$\text{MFlops} = \frac{2kn^3}{10^6 \cdot \text{cputime}}$$

where n is the matrix dimension used in a benchmark test and $2n^3$ floating point operations are needed for the matrix multiply. In order to obtain accurate timings, we chose $k \geq \text{MINRUNS}$ and `cputime` \geq `MINSECS`, where `MINRUNS`=4 and `MINSECS`=0.1 as defined in `bench.c`. The benchmark driver automatically adjusts the number of runs k to meet these constraints.

You may have noticed when you followed the instructions in Section 1 that the precision of the results reported by the benchmark programs was only one digit! Hardly trustworthy. By keeping `MINSECS` small we avoided very long waiting times for the benchmarks. However, if the resolution of `cputime` is too low then the timing precision in digits p_{digits} might be insufficient, since:

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

where r is the resolution of `cputime()` in seconds. To ensure a reasonable precision of our timings we can increase `MINRUNS` 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.

To study the use of timers for benchmarking, you will need to investigate the applicability and accuracy of the following timers:

- `USE_TIMES`: use `times()` to obtain CPU time (user + system time).
- `USE_GETRUSAGE`: use `getrusage()` to obtain CPU time (user + system time).

¹A footnote for experts: 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.

- `USE_GETTIMEOFDAY`: use `gettimeofday()` to obtain wall-clock time.
- `USE_RDTSC`: use the Intel RDTSC instruction to obtain high resolution wall-clock time. Only available with Intel IA32 and IA64. Be careful when using this with multicore processors, since these may have RDTSC clocks per core and a context switch to another core gives a different readout. To avoid this you must set the processor affinity.
- `USE_GETHRTIME`: use `gethrtime()` to obtain high resolution wall-clock time. Available on Sun Solaris only.

To select a timer for `cputime()` use one of the above “defines” (C/C++ `#define` macros). A macro (`#define`) is set with compiler option `-D`, see the `make.platform-comp` files for platform- and compiler-specific options that are used with the `Makefile`.

Task: your task is to investigate the applicability of timers and their accuracy (with `MINRUNS=4` and `MINSECS=0.1`) on three platforms: `pamd2` (Pentium-4 2.8GHz), `prism006` (dual core Xeon 5160 3GHz), and `tempest` (UltraSparc IIIi 1.2GHz). To this end, run one of the square matrix multiply benchmark tests. Don’t write new code for this part of the assignment. You can simply compile and run `sqmat` for this test by adjusting the appropriate `make.platform-comp` files before running `make COMP=comp sqmat`, where `comp` is the compiler you want to use which is either `gcc`, `icc`, or `suncc` (depending on availability on the platform).

Important: always do a `make clean` whenever you change one of the `make.platform-comp` files or when you change a header file (`.h` file). Then recompile with `make COMP=comp prog` to compile `prog` using compiler `comp`.

3 Optimization

In this part of the assignment we will experiment with advanced compiler optimizations to improve the speed of matrix multiply implementations. Do not get disappointed that your implementations can’t beat BLAS3 DGEMM of the Sun high-performance library which is highly optimized for multicore processors. Getting within 50% is already very good with one processor core, for which you need to try advanced compiler optimizations and inspect the results.

For these assignments you should make use of the `man` pages of `gcc/g77`, `icc/ifort`, and `cc/f95` (Sun), and other documentation sources for these compilers if needed:

Intel C++ and Fortran compilers documentation:

<http://www.intel.com/cd/software/products/asm-na/eng/346158.htm>
<http://www.intel.com/cd/software/products/asm-na/eng/346152.htm>

Sun Studio 12 compilers documentation:

<http://developers.sun.com/sunstudio/documentation/product/compiler.jsp>

3.1 BLAS Level 3: DGEMM

Task: compile the programs on `prism006` with `suncc` (again using `make clean` then `make COMP=suncc`). Make sure that the `suncc -fast -g` options are used. Now run `er_src sqmatf` to see how the compiler optimized the code with advanced loop optimizations. Run `make COMP=suncc plot` and display the data plot with `gnuplot -persist timing.gnuplot`. Explain the results you get for `sqmatf`.

Task: Determine the peak performance of `sqblas` on `prism006` and `tempest` (based on the benchmark output) and compare this performance to the peak performance advertised for these machines (Xeon 5160 3GHz and UltraSparc IIIi 1.2GHz).

3.2 Loop Optimization, Loop Vectorization, and Math Instructions

Task: compile the code with standard loop optimizations enabled. Note that loop optimizations such as loop unrolling, unroll and jam, fusion, and fission are already enabled with `icc -O3` and `suncc -fast`. To see how effective these optimizations are, compare the performance of the `-O3` and `-fast` optimized code to non-optimized code on both `prism006` and `tempest`. With `suncc` use option `-g` and after compilation run `er_src sqmatf` to see the optimizations that the compiler has applied. Note the differences produced by `suncc` between `prism006` and `tempest`. For `sqmat` and `sqmatf` make a list of the names of optimizations that `suncc` has applied and briefly describe them.

Task: investigate the use of alias disambiguation with “restrict-based” compiler options to ensure the compiler can optimize code without limitations of assumed dependences between unrelated function arguments such as pointers to arrays. These options differ between compilers. Find the appropriate ones and check if your code runs any faster or not.

Task: compile your code with loop vectorization to utilize Xeon’s SSE, SIMD-like, vector instructions on `prism006`. To do so with `gcc`, use `gcc -mfpmath=sse`. With `icc` use `icc -axW -vec-report3`. For `suncc` use `suncc -fast -g -xvector=simd` and `er_src prog` to view the results. Use other optimization switches/flags as needed, e.g. `-O3` and `-fast`. Compare the performance of the non-vectorized code with the vectorized code. Recall that alias disambiguation may enable further vectorization of loops.

Task: investigate the effect of using FMA instructions `suncc -fma=fused` on `tempest`.

Investigate `gcc -ffast-math` on `prism006`.

Hint: to run initial performance tests to find good combinations of compiler options, use a limited number of data samples to produce fewer test points by editing `bench.c` and use your own `sqmat_test_size` array. However, all of the results in your report must use the original `sqmat_test_size`.

3.3 Prefetching

Task: investigate the effect of prefetching on `tempest` by turning software prefetching on/off with `-fast -xprefetch=auto` and `-fast -xprefetch=no%auto`. Check the code with `er_src`. Explain the results, i.e. prefetching may or may not be possible and/or effective to hide memory latencies. What is the likely cause of the dips in performance at certain data points?

4 Blocking

Blocking improves locality of memory accesses to speed up the multiplication of large matrices. Implement square matrix multiply blocking in `sqmatb.c` (see also the instructions included in this file). If you prefer to implement this in Fortran, then create a new file `sqmatb.f` and modify the `Makefile`. You don't need to implement copy with blocking, but feel free to experiment.

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 $C = AB$ block-wise for each $BLKSIZE \times BLKSIZE$ block located at $C_{i,j}$, $A_{i,k}$, and $B_{k,j}$.

Task: implement the blocking algorithm and find two block sizes `BLKSIZE` that work well on `prism006` and `tempest`, respectively. Report how you found your block sizes and show your results with `gcc`, `icc`, and `suncc` on `prism006` and `suncc` on `tempest`. Experiment with advanced loop optimizations as explained in the Section 3, such vectorization. Compare the results to the non-blocked version. Explain the results.

5 Reducing the Number of Operations

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

5.1 Winograd's Algorithm

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

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

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.

Task: implement Winograd's algorithm in Fortran (or C if you prefer) in `sqmatw.f`.

5.2 Is MFlops Really a Fair Measure of Performance?

As a measure of performance we used MFlop/sec (or MFlops, Flop = floating point operation). 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 an output 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 diagonal matrices. A check for diagonality takes n^2 comparisons and computing the output 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 definitely win!

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

$$\text{MFlops} = \frac{2kn^3}{10^6 \cdot \text{cputime}}$$

and use it as a *scaled measure of performance* of square 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.

Task: it is actually more important to find an algorithm with a better FP:M ratio than saving more floating point operations, since memory access is expensive. Thus, reducing the number of floating point operations should (hopefully) also reduce the number of distinct memory locations referenced. 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.

Task: compare the performance of `sqmatw` to the other algorithms using vectorization and other (loop) optimizations enabled as explained in Section 3.

6 Plotting Help

To plot the data files of a set of benchmark programs, run `./timing.sh prog1 prog2...` followed by `gnuplot -persist timing.gnuplot`. Note that `gnuplot` is not available on `tempest`, so you should run `gnuplot` on `pamd` via another `xterm`.

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.