263-2300-00: How To Write Fast Numerical Code

Assignment 2: 100 points

Due Date: Th, March 12th, 17:00

http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring15/course.html

Questions: fastcode@lists.inf.ethz.ch

Submission instructions (read carefully):

- (Submission)
  Homework is submitted through the Moodle system https://moodle-app2.let.ethz.ch/course/view.php?id=1317. Before submission, you must enroll in the Moodle course. Enrollment key is "263-2300".
- (Late policy)
  You have 3 late days, but can use at most 2 on one homework, meaning submit latest 48 hours after the due time. Note that each homework will be available for submission on the Moodle system 2 days after the deadline. However, if the accumulated time of the previous homework submissions exceeds 3 days, the homework will not count.
- (Formats)
  If you use programs (such as MS-Word or Latex) to create your assignment, convert it to PDF and name it homework.pdf. When submitting more than one file, make sure you create a zip archive that contains all related files, and does not exceed 10 MB. Handwritten parts can be scanned and included or brought (in time) to Alen's or Daniele's office. Late homeworks have to be submitted electronically.
- (Plots)
  For plots/benchmarks, be concise, but provide necessary information (e.g., compiler and flags) and always briefly discuss the plot and draw conclusions. Follow (at least to a reasonable extent) the small guide to making plots (soon in lecture).
- (Neatness)
  5% of the points in a homework are given for neatness.

Exercises:

1. *Short project info (10 pts)* Go to the list of mile stones for the projects. If you have not done that yet, please register your project there. Read through the different points and fill in the first two with the following about your project (be brief):

   Point 1) An exact (as much as possible) but also short, problem specification.

   For example for MMM, it could be like this:

   Our goal is to implement matrix-matrix multiplication specified as follows:

   Input: Two real matrices A, B of compatible size, $A \in \mathbb{R}^{n\times k}$ and $B \in \mathbb{R}^{k\times m}$. We may impose divisibility conditions on n, k, m depending on the actual implementation. *Output:* The matrix product $C = AB \in \mathbb{R}^{n\times m}$.

   Give the name of the algorithm you plan to consider for the problem and a precise reference (e.g., a link to a publication plus the page number) that explains it.

   Point 2) A very short explanation of what kind of code already exists and in which language it is written.

2. *Vandermonde determinant (25 pts)* Code needed
   The code in `vandermonde.cpp` contains a class for representing Vandermonde matrices. The internal representation of a Vandermonde matrix of size $N \times N$ consists of an array of length N for storing the second row of the matrix.

   (a) The method `Vandermonde::det()` is used to compute the determinant of a matrix. Inspect the method `det()` and determine its op count (double additions and multiplications only). Assign the computed value to the macro `OPCOUNT` in `vandermonde.cpp`.

   (b) Identify performance limitations in `det()` and implement an optimized version of the code in `Vandermonde::det_opt()`.

(c) Compile the code disabling vectorization and determine its performance. Choose values of N up to $4k$ doubles (you can select sizes and stride so to avoid leftovers in your computation). Collect the results in a table and briefly list your optimization choices.

(d) Modify the function `test()` to collect performance measurements for the method `det()`. Use the values of N previously chosen for `det_opt()` and add your results to the table mentioned in 2c.

Report compiler, version, and flags. Submit your modified version of `vandermonde.cpp` to Moodle.

Solution

A possible reference solution can be found here.

3. *Optimization Blockers (40 pts)* Code needed

Download, extract and inspect the code. Your task is to optimize the function called superslow (guess why it's called like this?) in the file **comp.c**. The function runs over an $n \times n$ matrix and performs some computation on each element. In its current implementation, *superslow* involves several optimization blockers. Your task is to optimize the code.

Run `make` to compile the code. For Windows users, we recommend using Cygwin as a developing environment. Edit the Makefile if needed (architecture flags specifying your processor). The generated executable verifies the code and outputs the performance (the flop count is underestimated, since the trigometric functions are ignored) of superslow. Proceed as follows

(a) Identify optimization blockers discussed in the lecture and remove them.

(b) For every optimization you perform, create a new function in `comp.c` that has the same signature and register it to the timing framework through the *register_function* procedure in *comp.c* Let it run and, if it verifies, determine the performance.

(c) In the end, the innermost loop should be free of any procedure calls and operations other than adds and mults.

(d) When done, rerun all code versions also with optimization flags turned off ($-O0$ in the Makefile).

(e) Create a table with the performance numbers. Two rows (optimization flags, no optimization flags) and as many columns as versions of superslow. Briefly discuss the table.

(f) Submit your `comp.c` to Moodle.

What speedup do you achieve?

Solution

We achieved a speed up of 25.07x over the initial `superslow` function by setting the proper flags and 12.05x in the `-O0` case. In the first case we reach 0.8374 F/C, and in the second 0.2182 F/C. Testing environment:

- CPU Intel(R) Core(TM) i7-3720QM CPU 2.60GHz
- Mac OS X 10.10 (Yosemite)
- `gcc 4.2.1` (Apple LLVM version 6.0 `(clang-600.0.56)`)
- `-O3 -m64 -march=corei7-avx`

Results can be obtained here and transformation which would yield full points can be found here

4. *Locality of Gaussian Elimination (20 pts)*

Consider the following C code, which computes Gaussian Elimination of a Nonsingular Matrix A of size $N \times N$.

```
double A[N][N],tmp;
for (int k = 0; k < N; k++)
  double max_p = abs(A[k][k]);
  int ind_p =  k;

  // Find Pivot
```

```
for(int i = k+1; i < N; i++){
    tmp = abs(A[i][k]);
    if(tmp > max_p){
            max_p = tmp;
            ind_p = i;
    }
}


// Swap row containing pivot with k-th row
for(int j = 0; j < N; j++){
    tmp = A[ind_p][j];
    A[ind_p][j] = A[k][j];
    A[k][j] = tmp;
}

// Main Loop for Gaussian Elimination
for(int i = k+1; i < N; i++){
    for(int j = k+1; j < N; j++){
            A[i][j] = A[i][j] - A[k][j] * (A[i][k]/A[k][k]);
    }
    A[i][k] = 0;
}
```

Inspecting the data accesses, where do you see

(a) Temporal locality?

(b) Spatial locality?

Solution

1. Temporal Locality: The element A[k][k] inside main loop is accessed $\mathcal{O}(N^2)$ times whereas element A[i][k] is accessed $\mathcal{O}(N)$ times. Thus, both show temporal locality inside the main loop. Elements A[ind_p][j] and Elements A[k][j] inside the loop for swapping rows also show temporal locality as both are accessed twice inside the loop.

2. Spatial Locality: The access to A[i][j], A[k][j] inside the main loop is sequential thus resulting in spatial locality. Elements A[ind_p][j] and Elements A[k][j] inside the loop for swapping rows are also accessed sequentially thus resulting in spatial locality.