MATH-GA 2012.002, Spring 2023, Homework 1
Nikola Janjušević, February 12, 2023

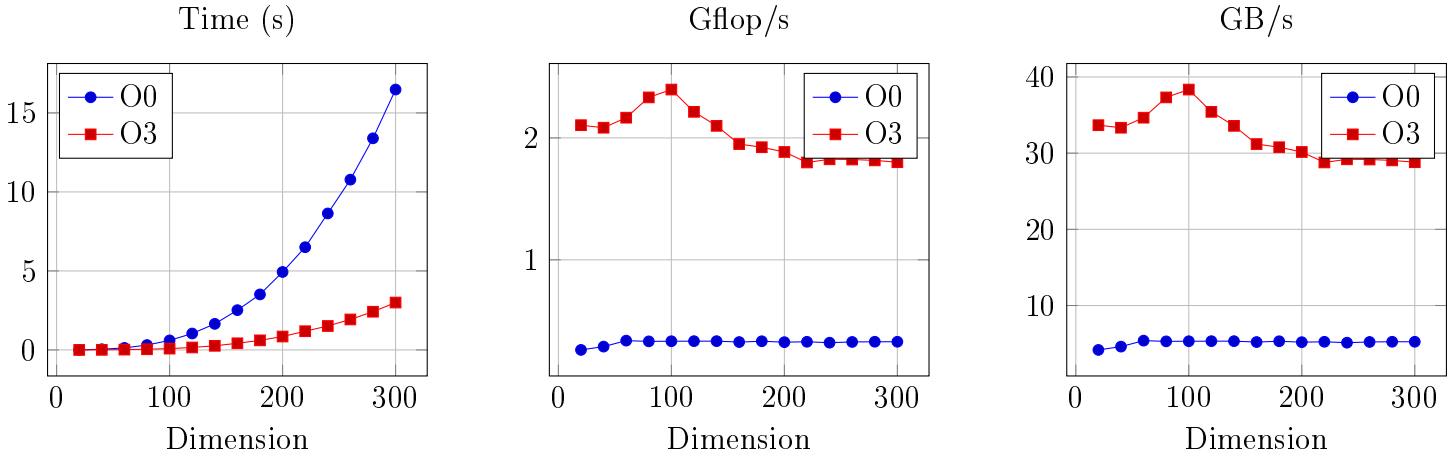
Problem 1 (Presentation)**.** On "KAISA: An Adaptive Second-Order Optimizer Framework for Deep Neural Networks". See `JANJUSEVIC_NIKOLA.pdf` for slides.

Problem 2 (Matrix-matrix multiplication (MMM))**.**
We edit and compile `MMutl0.cpp`, a program that does matrix-matrix multiplication naively via a tripple nested for loop. For multiplication matrices $C \leftarrow C + AB$ of shapes $C \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{m\times k}$, and $B \in \mathbb{R}^{k\times n}$, the inner iteration at indices (i, j, p) of the loop consists of:

1) reading A_{ip}, B_{pj}, and C_{ij} (i.e. 3 reads)
2) `C_ij += A_ip * B_pj` (i.e. 1 add and 1 multiply)
3) writing to C_{ij}

Hence, our total number of flops per MMM is `2mnk`, and our total memory read/write per MMM is `4mnk * sizeof(double) = 64mnk`. Note that this is not an optimal implementation of MMM, nor an optimal implementation of *naive* MMM.

The following tests were run on an Intel Core i5-8250U CPU, a 64 bit x86 processor with a max clock frequency of 3.4 GHz and a maximum bandwidth of 35.76 GB/s. The processor has an L1, L2, and L3 cache size of 256 kB, 1 MB, and 6 MB, respectively[1]. The code with compiled with `g++ 12.2.0`.



(A) Runtime (B) Flops (C) Bandwidth

FIGURE 1. `MMult0.cpp` output over different compliation optimization levels (O0, O3). Plotting code (`pgfplots`) generated by ChatGPT.

Figure 1 shows the output of the matrix multiplication program under different optimization levels. In this setting `M=N=K=dimension`. We see that without optimization, the code

[1] https://en.wikichip.org/wiki/intel/core_i5/i5-8250u

stagnates its performance in terms of flops and bandwidth. With optimization, the runtime increases while flops and bandwidth decrease with optimization level dimension. The use of compiler optimization also brings the performance of the program close to the manufacturer's specifications for the processor.

An interesting note is the relative plateu of performance in the `dimension=[10,100]` range. This can be explained by the size of the L1 cache being 256 kB, which can hold 3 $N \times N$ matrices of (roughly up) to size $N = 103$. After dimension 100, slower forms of memory are likely being used and we observe a sharp decrease in performance.

Problem 3 (Laplace 1D)**.**
`laplace.cpp` (given at the end of this document) implements Jacobi and Gauss-Seidel solvers for the 1D Laplace equation with Dirichlet boundary conditions. The program was compared against the analytical solution $u(x) = \frac{1}{2}x(1-x)$, $x \in [0, 1]$ for correctness. Table 1 shows the statistics of the algorithms when compiled and run on the processor described in the previous problem.

TABLE 1. `laplace.cpp` solver stats over dimension (N) and compiler optimization (O0, O3), `maxit=50000`.

Solver	Stats	Dimension (N)			
		10	100	1k	100k
Jacobi	O0-time	2e-4	0.080	1.85	156
	O3-time	2e-5	4e-3	0.095	11.4
	iters	222	18830	50k	50k
	residual	3e-4	1e-2	22.29	315.4
Gauss-Seidel	O0-time	6e-4	0.035	1.79	145
	O3-time	8e-6	3e-3	0.18	17.9
	iters	112	9416	50k	50k
	residual	3e-4	1e-2	22.29	315.4

After $N = 1,000$, neither solver is able to run to convergence. We observe that the Gauss-Seidel algorithm is per-iteration slower than Jacobi but has a faster runtime when given enough time to converge (as it uses less iterations). Enabling compilation optimzation speeds up both algorithms by an order of magnitude.

MMult0.cpp

```
1  // + Experiment with different optimization levels from -O0 to -O3 and
       report
2  //   the flop-rate and the bandwidth observed on your machine.
3  // + Specify the the compiler version (using the command: "g++ -v")
4  // + Try to find out the frequency, the maximum flop-rate and the maximum
       main
5  //   memory bandwidth for your processor.
6  // $ g++ -O3 -std=c++11 MMult0.cpp && ./a.out
7
8  #include <stdio.h>
9  #include "utils.h"
10
11 // Note: matrices are stored in column major order; i.e. the array
       elements in
12 // the (m x n) matrix C are stored in the sequence: {C_00, C_10, ...,
       C_m0,
13 // C_01, C_11, ..., C_m1, C_02, ..., C_0n, C_1n, ..., C_mn}
14 void MMult0( long m, long n, long k, double *a,
15                                      double *b,
16                                      double *c) {
17   for (int i = 0; i < m; i++) {
18     for (int j = 0; j < n; j++) {
19       for (int p = 0; p < k; p++) {
20         double A_ip = a[i+p*m];
21         double B_pj = b[p+j*k];
22         double C_ij = c[i+j*m];
23         C_ij = C_ij + A_ip * B_pj;
24         c[i+j*m] = C_ij;
25       }
26     }
27   }
28 }
29
30 int main(int argc, char** argv) {
31   const long NREPEATS = 100;
32   const long PFIRST = 20;
33   const long PLAST = 300;
34   const long PINC = 20;
35
36   printf(" Dimension       Time    Gflop/s       GB/s\n");
37   for (long p = PFIRST; p <= PLAST; p += PINC) {
38     long m = p, n = p, k = p;
39     double* a = (double*) malloc(m * k * sizeof(double)); // m x k
40     double* b = (double*) malloc(k * n * sizeof(double)); // k x n
41     double* c = (double*) malloc(m * n * sizeof(double)); // m x n
42
43     // Initialize matrices
44     for (long i = 0; i < m*k; i++) a[i] = drand48();
```

```
45      for (long i = 0; i < k*n; i++) b[i] = drand48();
46      for (long i = 0; i < m*n; i++) c[i] = drand48();
47
48      Timer t;
49      t.tic();
50      for (long rep = 0; rep < NREPEATS; rep++) {
51        MMult0(m, n, k, a, b, c);
52      }
53      double time = t.toc();
54      double flops = NREPEATS * m * n * 2*k / time / 1e9;
55      double bandwidth = NREPEATS * 4 * m * n * k * sizeof(double) / time /
            1e9;
56      printf("%10ld %10f %10f %10f\n", p, time, flops, bandwidth);
57
58      free(a);
59      free(b);
60      free(c);
61    }
62
63    return 0;
64  }
```

laplace.cpp

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <errno.h>
4  #include <math.h>
5  #include "utils.h"
6
7  void printvec(double* v, long n1, long n2)
8  {
9      for(long i=n1; i<n2; i++) printf("%f\n", v[i]);
10 }
11 void printvec(double* v, long N){ return printvec(v, 0, N); }
12
13 double* jacobi_iter(long N, double* u, double* u_old, double* f)
14 {
15     double h2 = 1/pow(N+1, 2);
16
17     // set previous value to current value
18     for(long i=0; i<N+2; i++) u_old[i] = u[i];
19
20     // boundary values u[0] = u[N+1] = 0
21     for(long i=1; i<N+1; i++)
22     {
23         u[i] = 0.5 * (h2*f[i] + u_old[i-1] + u_old[i+1]);
24     }
25     return u;
26 }
27
28 double* gauss_seidel_iter(long N, double* u, double* f)
29 {
30     double h2 = 1/pow(N+1, 2);
31
32     // boundary values u[0] = u[N+1] = 0
33     for(long i=1; i<N+1; i++)
34     {
35         u[i] = 0.5 * (h2*f[i] + u[i+1] + u[i-1]);
36     }
37     return u;
38 }
39
40 double normdiff(long N, double* u, double* v)
41 {
42     double err = 0;
43     for(long i=0; i<N; i++) err += pow(u[i] - v[i], 2);
44     return sqrt(err);
45 }
46
47 double residual(long N, double* u, double *f)
48 {
```

```
49      double h2 = 1/pow(N+1, 2);
50      double res = 0;
51      for(long i=1; i<N+1; i++){
52          res += pow((2*u[i] - u[i-1] - u[i+1])/h2 - f[i], 2);
53      }
54      return sqrt(res);
55  }
56
57  int main(int argc, char** argv) {
58      // argument parsing
59      if(argc < 3)
60      {
61          fprintf(stderr, "Usage: %s [N] [MAXIT=5000] [PRINT_SKIP=1]
                [GAUSS_SEIDEL=0]\n", argv[0]);
62          exit(-1);
63      }
64
65      errno = 0;
66      const double TOL = 1e-4;
67      const long N = strtol(argv[1], NULL, 10);
68
69      long MAXIT = 5000;
70      if(argc > 2) MAXIT = strtol(argv[2], NULL, 10);
71
72      long PRINT_SKIP = 1;
73      if(argc > 3) PRINT_SKIP = strtol(argv[3], NULL, 10);
74
75      bool GAUSS_SEIDEL = false;
76      if(argc > 4) GAUSS_SEIDEL = (strtod(argv[4], NULL) == 1) ? true :
            false;
77
78      if(errno != 0)
79      {
80          perror("strtol");
81          exit(EXIT_FAILURE);
82      }
83      printf("N=%ld, MAXIT=%ld, TOL=%f, PRINT_SKIP=%ld, GAUSS_SEIDEL=%s\n",
            \
84              N, MAXIT, TOL, PRINT_SKIP, GAUSS_SEIDEL ? "true" : "false");
85
86      double res, res0, x;
87      double* f;     // data vector
88      double* u;     // current solution vector
89      double* u_old; // previous solution vector
90      double* u_sol; // analytic solution vector
91
92      // allocate and init
93      f     = (double*) malloc((N+2) * sizeof(double));
94      u_sol = (double*) malloc((N+2) * sizeof(double));
95      u     = (double*) malloc((N+2) * sizeof(double));
```

```
96      u_old = (double*) malloc((N+2) * sizeof(double));
97      for(long i=0; i<N+2; i++)
98      {
99          f[i] = 1;
100         u[i] = 0;
101         x = (double) i / (N+1);
102         u_sol[i] = 0.5 * x * (1 - x);
103     }
104
105     Timer t;
106     t.tic();
107
108     // solve
109     res0 = residual(N, u, f);
110     printf("     k,      res,      solres\n");
111     printf("%10d, %10f, %10f\n", 0, res0, normdiff(N, u_sol, u));
112     for(long k=1; k<=MAXIT; k++)
113     {
114         GAUSS_SEIDEL ? gauss_seidel_iter(N, u, f) : jacobi_iter(N, u,
               u_old, f);
115         res = residual(N, u, f);
116         if(k % PRINT_SKIP == 0 || res < TOL*res0){
117             printf("%10ld, %10f, %10f\n", k, residual(N, u, f),
                   normdiff(N+2, u_sol, u));
118         }
119         if(res < TOL*res0) break;
120     }
121     printf("time = %f\n", t.toc());
122
123     free(f);
124     free(u);
125     free(u_old);
126     free(u_sol);
127     return 0;
128 }
```