CS 491-004 Project 1 Report

Sean Gillen

February 6, 2019

Introduction

In this project, I implemented and compared square matrix multiplication algorithms with various levels of register reuse and tested the performance of these algorithms.

Part 1

The implementation of dgemm0 is reproduced below:

```
void dgemm0(double *a, double *b, double *c, int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            for(int k = 0; k < n; k++) {
                c[i*n+j] += a[i*n+k] * b[k*n+j];
            }
        }
    }
}
```

Note that with each iteration of the innermost loop, three memory accesses are made to retrieve the values a[i*n+k], b[k*n+j], and c[i*n+j], two floating-point operations are performed, and one memory access is made to store the result in c[i*n+j].

Assuming that both floating-point operations can be completed in one cycle and each memory access adds a delay of 100 cycles, each iteration of the inner loop uses 401 cycles. For $n = 1000$, the body of the inner loop is executed 1000^3 times, corresponding to a total of 4.01×10^{11} processor cycles. At 2 GHz, this is equal to $\boxed{\text{200.5 seconds}}$.

The implementation of `dgemm1` is reproduced below:

```
void dgemm1(double *a, double *b, double *c, int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            register double r = c[i*n+j];
            for(int k = 0; k < n; k++) {
                r += a[i*n+k] * b[k*n+j];
            }
            c[i*n+j] = r;
        }
    }
}
```

In this algorithm, the innermost loop includes only two memory accesses, with the other two memory accesses taking place only with every iteration of the second-level loop. Using the assumptions noted above, this means the innermost loop body completes in 201 cycles and the body of the second-level loop completes in $201n + 200$ cycles. For $n = 1000$, the total execution time of `dgemm1` is then 2.012×10^{11} cycles or, at 2 GHz, $\boxed{\text{100.6 seconds}}$, 49.8% faster than `dgemm0`.

On my computer, `dgemm1` completes up to 32% faster than `dgemm0` on tests with $n \leq 1024$, but 1% slower than `dgemm0` when $n = 2048$ (see pages 6–7 for complete results).

Part 2

My implementation of `dgemm2` is reproduced below:

```
void dgemm2(double *a, double *b, double *c, int n) {
    int i, j, k;
    for(i = 0; i < n; i += 2) {
        for(j = 0; j < n; j += 2) {
            register double c0 = c[i*n + j];
            register double c1 = c[i*n + j+1];
            register double c2 = c[(i+1)*n + j];
```

```
            register double c3 = c[(i+1)*n + j+1];
            for(k = 0; k < n; k += 2) {
                register double a0 = a[i*n + k];
                register double a1 = a[i*n + k+1];
                register double a2 = a[(i+1)*n + k];
                register double a3 = a[(i+1)*n + k+1];
                register double b0 = b[k*n + j];
                register double b1 = b[k*n + j+1];
                register double b2 = b[(k+1)*n + j];
                register double b3 = b[(k+1)*n + j+1];
                c0 += a0*b0 + a1*b2;
                c2 += a2*b0 + a3*b2;
                c1 += a0*b1 + a1*b3;
                c3 += a2*b1 + a3*b3;
            }
            c[i*n + j] = c0;
            c[i*n + j+1] = c1;
            c[(i+1)*n + j] = c2;
            c[(i+1)*n + j+1] = c3;
        }
    }
}
```

On my computer (see page 4 for system informattion), `dgemm2` completes 56–70% faster than `dgemm0` (see page 7 for complete results).

Part 3

I implemented `dgemm3` as follows:

```
void dgemm3(double *a, double *b, double *c, int n) {
    register int i, j, k;
    register int rn = n;
    for(i = 0; i < rn; i += 2) {
        register int in = i*rn;
        for(k = 0; k < rn; k += 2) {
            register int kn = k*rn;
            register double a0 = a[in + k];
```

```
                register double a1 = a[in + k+1];
                register double a2 = a[in+rn + k];
                register double a3 = a[in+rn + k+1];
                for(j = 0; j < rn; j += 2) {
                    register double b0 = b[kn + j];
                    register double b1 = b[kn + j+1];
                    register double b2 = b[kn+rn + j];
                    register double b3 = b[kn+rn + j+1];
                    c[in + j] += a0*b0 + a1*b2;
                    c[in+rn + j] += a2*b0 + a3*b2;
                    c[in + j+1] += a0*b1 + a1*b3;
                    c[in+rn + j+1] += a2*b1 + a3*b3;
                }
            }
        }
}
```

On my computer, dgemm3 completes up to 97% faster than dgemm0 (see page 7 for complete results).

Performance measurements

I tested each matrix multiplication algorithm on an Intel Core i7-6700HQ processor running at 3.5 GHz after compiling the following test program with gcc 6.3.0 at optimization level 3. The GFLOPS performance for each algorithm is calculated based on an assumption that each algorithm includes $3n^3$ floating-point operations.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>

// implementations of dgemm0 through dgemm3 omitted on this page

void timeMultiplication(char *name, void (*mmm)(double*, double*,
    double*, int), double *a, double *b, double *c, int n) {
    clock_t start = clock();
```

```
    (*mmm)(a, b, c, n);
    double seconds = (double) (clock() - start) / CLOCKS_PER_SEC;
    double gflops = ((double) 3 * n*n*n) / 1000000000.0 / seconds;
    printf("%s completed in %.6f seconds (%.4f GFLOPS)\n", name,
        seconds, gflops);
}

int main(int argc, char *argv[]) {
    srand(time(0));

    int n = 1000;
    if(argc > 1) {
        n = atoi(argv[1]);
    }
    double *a = malloc(sizeof(double) * n*n);
    double *b = malloc(sizeof(double) * n*n);
    double *c0 = malloc(sizeof(double) * n*n);
    double *c1 = malloc(sizeof(double) * n*n);
    double *c2 = malloc(sizeof(double) * n*n);
    double *c3 = malloc(sizeof(double) * n*n);

    for(int i = 0; i < n*n; i++) {
        // Initialize A and B to random values and C to zero
        a[i] = (double) rand() / RAND_MAX * 1024;
        b[i] = (double) rand() / RAND_MAX * 1024;
        c0[i] = c1[i] = c2[i] = c3[i] = 0;
    }

    timeMultiplication("dgemm0", dgemm0, a, b, c0, n);
    timeMultiplication("dgemm1", dgemm1, a, b, c1, n);
    timeMultiplication("dgemm2", dgemm2, a, b, c2, n);
    timeMultiplication("dgemm3", dgemm3, a, b, c3, n);

    double maxdiff1 = 0;
    double maxdiff2 = 0;
    double maxdiff3 = 0;
    for(int i = 0; i < n*n; i++) {
        double diff1 = fabs(c1[i] - c0[i]);
```

```
        if(diff1 > maxdiff1) {
            maxdiff1 = diff1;
        }
        double diff2 = fabs(c2[i] - c0[i]);
        if(diff2 > maxdiff2) {
            maxdiff2 = diff2;
        }
        double diff3 = fabs(c3[i] - c0[i]);
        if(diff3 > maxdiff3) {
            maxdiff3 = diff3;
        }
    }
    printf("Maximum difference for dgemm1 is %.6f\n, maxdiff1);
    printf("Maximum difference for dgemm2 is %.6f\n, maxdiff2);
    printf("Maximum difference for dgemm3 is %.6f\n, maxdiff3);

    free(a);
    free(b);
    free(c0);
    free(c1);
    free(c2);
    free(c3);
    return 0;
}
```

The same tests I performed may be run with the attached source code using `make`. To compile without running the tests, run `make compile`. The results of these tests are listed below.

dgemm0

n	seconds	GFLOPS
64	0.001060	0.7419
128	0.011381	0.5528
256	0.045656	1.1024
512	0.371264	1.0845
1024	9.678723	0.3328
2048	127.853019	0.2016

`dgemm1`

			compared to `dgemm0`	
n	seconds	GFLOPS	speedup	max. difference
64	0.000892	0.8817	16%	0.000000
128	0.009744	0.6457	14%	0.000000
256	0.039746	1.2663	13%	0.000000
512	0.250638	1.6065	32%	0.000000
1024	6.884244	0.4679	29%	0.000000
2048	129.088407	0.1996	−1%	0.000000

`dgemm2`

			compared to `dgemm0`	
n	seconds	GFLOPS	speedup	max. difference
64	0.000333	2.3617	69%	0.000000
128	0.004979	1.2636	56%	0.000000
256	0.017665	2.8492	61%	0.000000
512	0.111821	3.6009	70%	0.000000
1024	2.942621	1.0947	70%	0.000001
2048	48.432833	0.5321	62%	0.000004

`dgemm3`

			compared to `dgemm0`	
n	seconds	GFLOPS	speedup	max. difference
64	0.000224	3.5109	79%	0.000000
128	0.001752	3.5910	85%	0.000000
256	0.004753	10.5894	90%	0.000000
512	0.039791	10.1192	89%	0.000000
1024	0.348491	9.2434	96%	0.000001
2048	3.395735	7.5889	97%	0.000004

Conclusion

These tests show that register reuse greatly impacts the runtime of square matrix multiplication on modern hardware. Speed improvements of up to 97% were obtained when multiplying 2048×2048 matrices of floating-point values by maximizing register reuse.