Noah Hanks

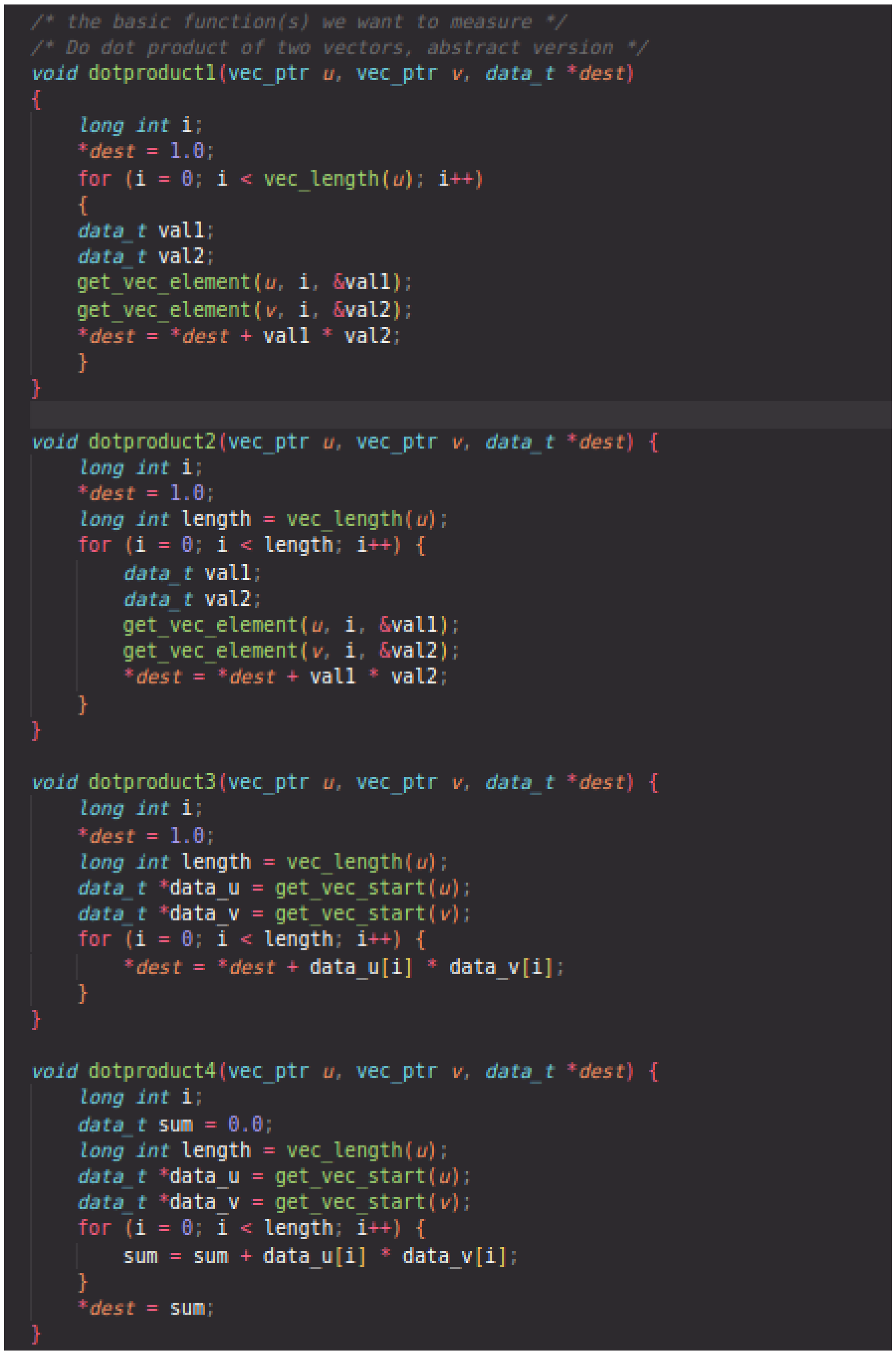
ECEN 424 HW 5

424-4)

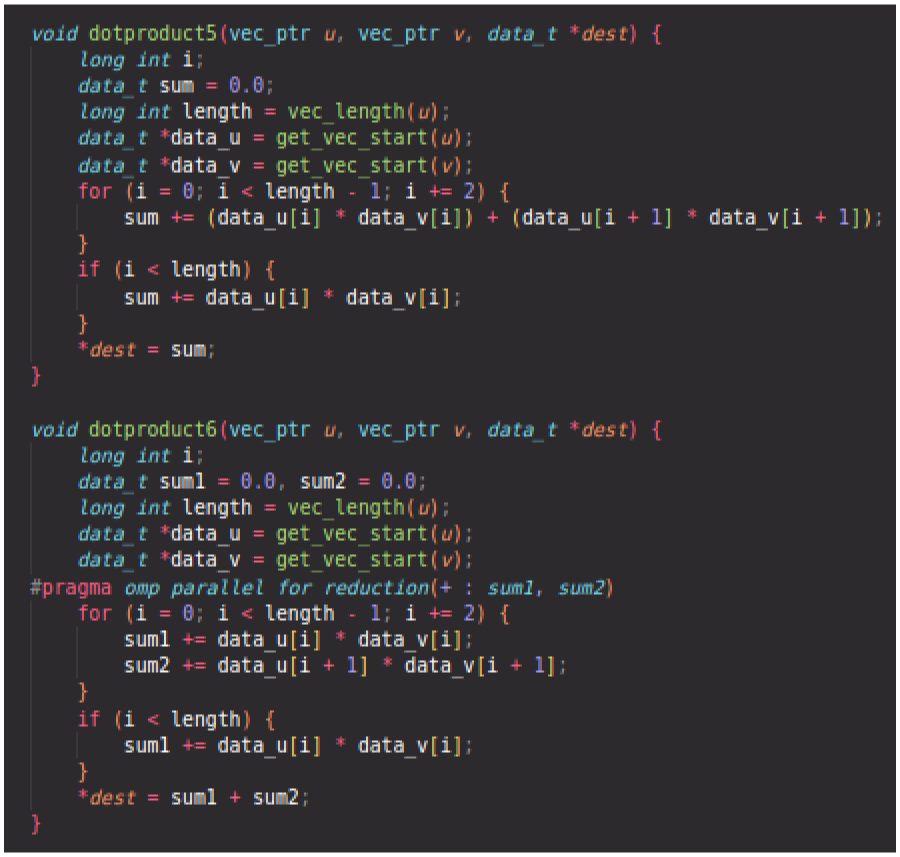
function	CPE
dotproduct1	11.3
dotproduct2	8.3
dotproduct3	7.3
dotproduct4	3.2
dotproduct5	1.6
dotproduct6	1.6
dotproduct7	1.7

function	CPE
combine1	12
combine2	8.09
combine3	8.01
combine4	3.0
combine5	1.5
combine6	1.5
combine7	1.51

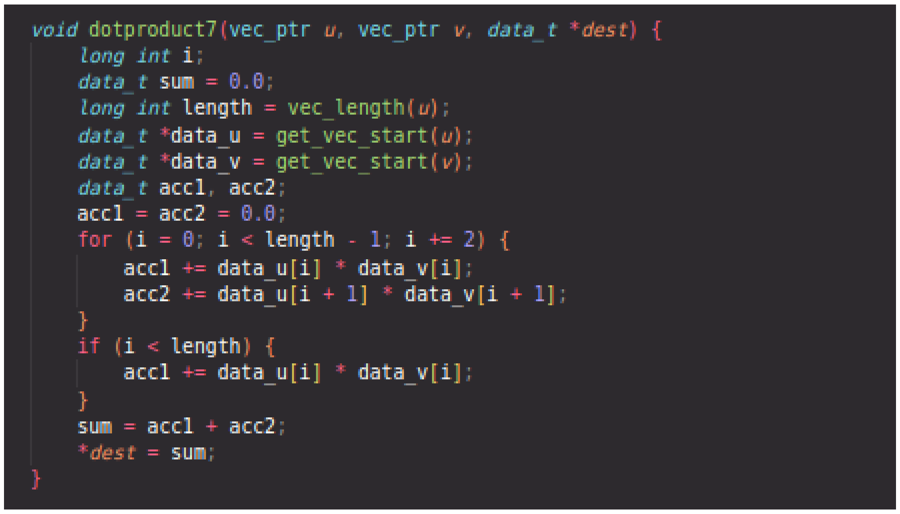
In general, the CPE for dotproduct is faster. This is likely due to the different memory access patterns and computation requirements of the two functions. The dotproduct functions perform a large number of arithmetic operations, which can be compute-bound and place high demands on the processor's arithmetic logic unit (ALU). The combine functions, on the other hand, primarily perform memory operations, which can be memory-bound and place high demands on the processor's memory hierarchy. From these results, it is difficult to infer much about the functional units in the processor of the system used from just looking at CPE.



```c
/* the basic function(s) we want to measure */
/* Do dot product of two vectors, abstract version */
void dotproduct1(vec_ptr u, vec_ptr v, data_t *dest)
{
    long int i;
    *dest = 1.0;
    for (i = 0; i < vec_length(u); i++)
    {
    data_t val1;
    data_t val2;
    get_vec_element(u, i, &val1);
    get_vec_element(v, i, &val2);
    *dest = *dest + val1 * val2;
    }
}

void dotproduct2(vec_ptr u, vec_ptr v, data_t *dest) {
    long int i;
    *dest = 1.0;
    long int length = vec_length(u);
    for (i = 0; i < length; i++) {
        data_t val1;
        data_t val2;
        get_vec_element(u, i, &val1);
        get_vec_element(v, i, &val2);
        *dest = *dest + val1 * val2;
    }
}

void dotproduct3(vec_ptr u, vec_ptr v, data_t *dest) {
    long int i;
    *dest = 1.0;
    long int length = vec_length(u);
    data_t *data_u = get_vec_start(u);
    data_t *data_v = get_vec_start(v);
    for (i = 0; i < length; i++) {
        *dest = *dest + data_u[i] * data_v[i];
    }
}

void dotproduct4(vec_ptr u, vec_ptr v, data_t *dest) {
    long int i;
    data_t sum = 0.0;
    long int length = vec_length(u);
    data_t *data_u = get_vec_start(u);
    data_t *data_v = get_vec_start(v);
    for (i = 0; i < length; i++) {
        sum = sum + data_u[i] * data_v[i];
    }
    *dest = sum;
}
```



```c
void dotproduct5(vec_ptr u, vec_ptr v, data_t *dest) {
    long int i;
    data_t sum = 0.0;
    long int length = vec_length(u);
    data_t *data_u = get_vec_start(u);
    data_t *data_v = get_vec_start(v);
    for (i = 0; i < length - 1; i += 2) {
        sum += (data_u[i] * data_v[i]) + (data_u[i + 1] * data_v[i + 1]);
    }
    if (i < length) {
        sum += data_u[i] * data_v[i];
    }
    *dest = sum;
}

void dotproduct6(vec_ptr u, vec_ptr v, data_t *dest) {
    long int i;
    data_t sum1 = 0.0, sum2 = 0.0;
    long int length = vec_length(u);
    data_t *data_u = get_vec_start(u);
    data_t *data_v = get_vec_start(v);
#pragma omp parallel for reduction(+ : sum1, sum2)
    for (i = 0; i < length - 1; i += 2) {
        sum1 += data_u[i] * data_v[i];
        sum2 += data_u[i + 1] * data_v[i + 1];
    }
    if (i < length) {
        sum1 += data_u[i] * data_v[i];
    }
    *dest = sum1 + sum2;
}
```



```c
void dotproduct7(vec_ptr u, vec_ptr v, data_t *dest) {
    long int i;
    data_t sum = 0.0;
    long int length = vec_length(u);
    data_t *data_u = get_vec_start(u);
    data_t *data_v = get_vec_start(v);
    data_t acc1, acc2;
    acc1 = acc2 = 0.0;
    for (i = 0; i < length - 1; i += 2) {
        acc1 += data_u[i] * data_v[i];
        acc2 += data_u[i + 1] * data_v[i + 1];
    }
    if (i < length) {
        acc1 += data_u[i] * data_v[i];
    }
    sum = acc1 + acc2;
    *dest = sum;
}
```

424-5) To measure branch misprediction penalty, there are a lot of small benchmarks used. They have with a conditional branch set up to have a high misprediction rate. To measure the penalty, it calculates the time it takes to execute the loop with and without the mispredicted branches. The accurate measurement of the cost of branch mispredictions helps improve performance of processors.

With predictable branches, the best time was 10 and 27 with unpredictable branches. I was on an intel x86 CPU and the calculated penalty was 37. I then compiled using the -O2 flag and the best times for predictable and unpredictable were 4 and 3 respectively. By using the conditional move, the performance improved by 528.6%.