**Part 1**

1. **Double, Multiply**
2. **Double, Add**
3. **Float, Add**
4. **Float, Multiply**
5. **Int, Add**
6. **Int, Multiply**

The results seem to be very similar to the one in the book, with the CPE’s being lowest for combine6 and combine7 where parallelization is implemented.

**Partb)**

According to my data, the CPE gets slightly worse after Unroll factor 7. The performance could potentially be getting worse as the number of elements being referenced per inner loop iteration could be greater than the cache block size. Hence the processor may be accessing memory multiple times in a single inner loop iteration, which could be reducing performance.

**Partc)**

Although the CPE with the Accumulator and the CPE with reassociative transformation are almost indistinguishable, the CPE with accumulator is slightly lower for greater unrolling factors.

**Part 2**

Optimized using loop unrolling with 6 separate accumulators.

**Part 3**

Modified Code (vec 1 always greater than vec 2). For this I just made vector 1 data 1 greater than the vector index, and vector 2 data equal to the vector index-

int init\_vector1(vec\_ptr v, long int len)

{

long int i;

if (len > 0) {

v->len = len;

for (i = 0; i < len; i++) v->data[i] = i+1 ; // Modify this line

return 1;

}

else return 0;

}

int init\_vector2(vec\_ptr v, long int len)

{

long int i;

if (len > 0) {

v->len = len;

for (i = 0; i < len; i++) v->data[i] = i; // Modify this line

return 1;

}

else return 0;

}

Modified Code- Vec1 and Vec2 randomized – Using rand() function.

/\* initialize vector with first pattern \*/

int init\_vector1(vec\_ptr v, long int len)

{

long int i;

if (len > 0) {

v->len = len;

for (i = 0; i < len; i++) v->data[i] = rand() ; // Modify this line

return 1;

}

else return 0;

}

/\* initialize vector with another \*/

int init\_vector2(vec\_ptr v, long int len)

{

long int i;

if (len > 0) {

v->len = len;

for (i = 0; i < len; i++) v->data[i] = rand(); // Modify this line

return 1;

}

else return 0;

}

It can be seen both cases, where input is predictable and randomized, that branch2 is faster than branch1 and has a lower CPE. The reason for this can be identified when the assembly language code for branch1 and branch2 are analyzed. It can be clearly seen in the assembly language that branch2 has a much smaller loop than branch1, and branch2 uses a maxsd instruction instead of je or jne instruction to compare the two values. It can also be seen from the final graph, that as length of the vectors increase, the predictable branches have a lower CPE than the randomized branches. This is owing to the fact the processor is more accurate in branch prediction.

**Part 4**

1. 6 hours
2. No
3. None
4. There was a minor mistake with the branch1 code, which I fixed.