Example 1: Optimizing C Code

```
#ifndef _GRAPHICS_H
#define _GRAPHICS_H



typedef struct
{
    unsigned char R;
    unsigned char G;
    unsigned char B;
    unsigned char alpha;
} pixel;



typedef struct
{
    pixel **map;
    int length, width;
} bitmap;



#endif
```

Optimizations/Speedups:

- Swap `i-j` order in `plot_bitmap`.
- Inline `plot_pixel`.
- Use local variable in `bitmap_checksum` so we don't always reference `result`.
- Try unrolling loops in `bitmap_checksum`.

```
#include "graphics.h"


void plot_pixel( pixel p , int posx, int posy )
{
    /* do some stuff to plot it */
    return;
}


void plot_bitmap( bitmap b )
{
    int i, j;

    for (j = 0; j < b.width; j++)
        for (i = 0; i < b.length; i++)
            plot_pixel( b.map[i][j], i, j );
}


int bitmap_checksum( bitmap b, unsigned *result )
{
    int i, j;

    *result = 0; /* initial checksum value is 0 */


    for (i = 0; i < b.length; i++)
        for (j = 0; j < b.width; j++)
        {
            pixel *p = &b.map[i][j];
            unsigned up = *(unsigned *)p;

            *result = *result ^ up;
        }

    return 0; /* no errors */
}


int main(int argc, char *argv[])
{
    return 0;
}
```

Example 2: Pipelined Processing

Consider the following function for computing the product of an array of n integers. We have unrolled the loop by a factor of 3.

```
int aprod(int a[], int n)
{
   int i, x, y, z;
   int r = 1;

   for (i = 0; i < n-2; i+= 3)
   {
      x = a[i]; y = a[i+1]; z = a[i+2];
      r = r * x * y * z; // Product computation
   }
   for (; i < n; i++)
      r *= a[i];

   return r;
}
```

For the line labeled Product computation, we can use parentheses to create 5 different associations of the computation, as follows:

```
r = ((r * x) * y) * z; // A1
r = (r * (x * y)) * z; // A2
r = r * ((x * y) * z); // A3
r = r * (x * (y * z)); // A4
r = (r * x) * (y * z); // A5
```

We express the performance of the function in terms of the number of cycles per element (CPE). As described in the book, this measure assumes the run time, measured in clock cycles, for an array of length n is a function of the form $C_n + K$ where C_n is the CPE.

We measured the 5 versions of the function on an Intel Pentium III. Recall from Figure 5.12 of the book that the integer multiplication operation on this machine has a latency of 4 cycles and an issue time of 1 cycle. The following table shows some values of the CPE, and other values missing. The measured CPE values are those that were actually observed. "Theoretical CPE" means that performance that would be achieved if the only limiting factor were the latency and issue time of the integer multiplier.

The Pentium III has only 2 Integer Multiplication units. You may neglect time used by load unit.

Version	Measured CPE	Theoretical CPE
A1	4.00	4.00
A2	2.67	$8/3 = 2.67$
A3	$5/3 = 1.67$	$4/3 = 1.33$
A4	1.67	$4/3 = 1.33$
A5	$8/3 = 2.67$	$8/3 = 2.67$

Fill in the missing entries. For the missing values of the measured CPE, you can use the values from other versions that would have the same computational behavior. For the values of the theoretical CPE, you can determine the number of cycles that would be required for an iteration considering only the latency and issue time of the multiplier, and then divide by 3.