HIGH PERFORMANCE COMPUTING

Assignment

Do data access analysis for project problem.

Propose optimal data access technique for your project problem.

Provide a detailed analysis report.

Project problem - Using Discrete Fourier Transform for finding the normalized frequency of a random wave and parallelizing the code using Openmp.

Bhaavanaa Thumu - CED17I021

Machine balance

The machine balance of a processor chip is the ratio of possible memory bandwidth in GWords/sec to peak performance in GFlops/sec.

Memory bandwidth = 34.1 / 8 = 4.2625

No. of cores = 2

Processor speed = 2.7 GHz

Flops = 4

Peak performance = no. of cores x processor speed x flops

= 2 x 2.7 x 4 = 21.6

Machine balance = memory bandwidth / peak performance

= 4.2625 / 21.6

= 0.1973

Hence, machine balance is 0.1973.

Code balance

The code balance of a loop is the ratio of data traffic to the number of floating point operations.

No. of load operations in the program = 4

No. of store operation in the program = 7

Total number of memory accesses = 4 + 7 = 11

No. of floating point operations = 18

Code balance = total no. of memory accesses / no. of floating point operations

= 11 / 18

= 0.6111

Hence, code balance is 0.6111.

Lightspeed of loop

machine balance / code balance = 0.1973 / 0.6111 = 0.3228

Lightspeed of loop = min(1, machine balance/code balance)

= min(1, 0.3228)

= 0.3228

Hence, the lightspeed of loop is 0.3228.

Algorithm classification and access optimizations

The optimization potential of many loops on cache-based processors can easily be estimated just by looking at basic parameters like the scaling behavior of data transfers and arithmetic operations versus problem size.

O(N) / O(N)

If both the number of arithmetic operations and the number of data transfers (loads/stores) are proportional to the problem size (or “loop length”) N, optimization potential is usually very limited. Scalar products, vector additions, and sparse matrix-vector multiplication are examples for this kind of problems. They are inevitably memory-bound for large N, and compiler-generated code achieves good performance because O(N)/O(N) loops tend to be quite simple and the correct software pipelining strategy is obvious. Loop nests, however, are a different matter. But even if loops are not nested there is sometimes room for improvement. As an example, consider the following from the code:

```
for (int k = 0; k < N; k++)                                  // finding the dft
{
        for (int n = 0; n < N; n++)                              // dft value
        {
                dft_val[k].real += samples[n] * cos((2 * PI * k * n) / N);
                dft_val[k].img += samples[n] * sin((2 * PI * k * n) / N);
        }

        k_by_N[k] = (double)k/N;                                 // k/N value

        power_spectrum[k] = dft_val[k].real*dft_val[k].real +
dft_val[k].img*dft_val[k].img;                     // power spectrum value

        sum_power_spectrum = sum_power_spectrum +
power_spectrum[k];                           // sum of power spectrum values
        sum_freq_power = sum_freq_power +
2*PI*k_by_N[k]*power_spectrum[k];        // sum of freq*power_spectrum
values
}
```

The above code removes the need for loading dft_val[k].real and dft_val[k].img, as it is used for computing as soon as it is updated. We are fusing the loops to reduce the loads. Similarly, for producing the arrays, sum_power_spectrum and sum_freq_power_spectrum, also use the recently used data for their computation. This gives a room for improvement.