CSEE4824 Course Project

Student Mengqi Zong < $mz2326@columbia.edu$ >

December 2011

Abstract

In this project, I use software optimization and hardware design to minimize the running time. In software optimization, I successfuly reduce the usage of memory and parallel the code. In hardware design, I use 5 samllcore. And this gaves me a sim time of 1.677 ms, 278.751 ms, 1301.463 ms for small, medium, large input.

1 Optimization flow

1.1 Software: reduce memory usage

In LCS, the data dependency is as follow [1]:

$$\text{S[i,j]} = \begin{cases} 0 & i \text{ or } j \text{ is } 0 \\ S[i-1,j-1]+1 & a_i = b_j \\ max(S[i-1,j], S[i,j-1]) & \text{o.w.} \end{cases}$$

It is clear that, in order to compute the current row, we just need to know the previous row. So we can use the bottom-up method to compute the LCS and it only requires two row's memory instead of the original n rows (n is the length of the input).

By doing this, large input is runnable. Also, the bottom-up method and the memory optimization can also double the performance. This should due to the no recursion overheads and a better cache hit rate for using less memory.

1.2 Design: singlecore design

1.2.1 Frequency

Increase clock frequency is the most efficient way to improve performance.

The ratio between frequency and performance is 1:1. Also, the power consumption is quite low. However, when frequency reaches 2.5G Hz, performance no longer improves. And when frequency is 2.5G Hz, performance has a 20 times improvement, and this only has an extara 6.2 watts power consumption.

1.2.2 Cache

Cache optimization is not necessary for singlecore.

The original L1 cache is enough. Large L1 cache and L2 cache can not improve performance at all. This is because the code's consecutive calculation has a very good locality.

1.2.3 CPU units

More units can slightly imporve the performance, but it is not energy efficient.

When using a largecore with IssueNumber = 10, it can give a 6.3% performance improvement, but it costs an extra 24 watts.

1.3 Software: parallel the code

1.3.1 Change the data dependency

When $S[i,j] = max(S[i-1,j], S[i,j-1])$, $S[i,j]$ can be made independent of ith row data as $S[i,j-1]$ can be replaced by the following recurrence equation [2]:

$$S[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j-1 = 0 \\ S[i-1,j-2]+1 & \text{if } a_i = b_{j-1} \\ max(S[i-1,j-1], & \\ S[i,j-2]) & \text{o.w.} \end{cases}$$

Again, only the third condition, $S[i,j-1]$ depends on the ith row data. Thus $S[i,j-2]$ can be re-

formulated in a similar way, and so on. This process ends when $j-k=0$ at the kth step, or $a_i = b_{j-k}$ at the kth step. Finally, we have:

$$S[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j-1=0 \\ S[i-1,j-1]+1 & \text{if } a_i = b_j \\ max(S[i-1,j], & \\ S[i-1,j-k-1]) & \text{if } a_i = b_{j-k} \\ max(S[i-1,j],0) & \text{if } j-k=0 \end{cases}$$

By changing the dependency, all ith row data can be calcuated just based on the $(i-1)$th row data. In this case, we can use at most N processors to run this program.

1.3.2 Two ways to parallel the code

There are two ways to parallel to code: balanced and range.

Suppose we have 2 threads and input length is 10. Balanced is to let thread 0 calculate data 0, 2, 4, 6, 8 and thread 1 calculate 1, 3, 5, 7, 9. The benefit of balaned is every thread's workload is balanced. The time to wait for all threads finish is shorter. The bad thing is locality is not easy to exploit as before, cache hit rate will be lower. And this also increase the power consumption.

As to range, thread 0 will calculate 0, 1, 2, 3, 4 and thread 1 will calculate 5, 6, 7, 8, 9. The benefit of range is a better cache hit rate and lower power consumption. However, every thread's workload is not balanced. In this example, thread 2 may need to execute much longer than thread 1 since the jth data's running time is $O(j)$. And this will affect the performance.

1.4 Design: mutlticore design

Having tried many designs, I find that cache, issue number still cannot boost performance significantly, which is the same as in singlecore design. The only main issue here is to choose the way to parallel the code based on the power requirement.

1.4.1 Choosing parallelization

Due to 30 watts power restriction, I find that balanced can only afford 5 smallcores with frequency 25 X. Range can afford 6 smallcores but has to reduce its frequency from 25 X to 20 X. However, the result of both implementation is quite close. On large input, balanced with 5 smallcores has a sim time of 1301.463 msec, and range with 6 smallcores has a sim time of 1319.399 msec. And range's power consumption is slightly higher than balanced due to its one extra core.

The parallelization has a performance boost of 40%. And the total boost is $2*20*1.4*100\% = 5600\%$

1.5 Final design

Here is the final configuration:
code: lcs_multi_balanced.c
config: 5 smallcore. Frequency 2.5G Hz.
The performance is shown in Table-1.

Input	Power (w)	Time (ms)
small	25.633	1.677
medium	29.087	278.751
large	28.287	1301.463

Table 1: Final performance.

2 Conclusion

Clock frequency is the most efficient way to boost performance. Boost performance by using multicore architecture is much harder. And multicore architecture is not always better than singlecore architecture.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[2] Yi Shang Jiaoyun Yang, Yun Xu. An efficient parallel algorithm for longest common subsequence problem on gpus. *Processings of the World Congress on Engineering 2010 Vol I*, 2010.