CPSC 3300 – Project 3 – Matrix Multiplication

Author: Rajat Sethi

Abstract:

This project looks through a program of matrix multiplication and applies several optimizations to improve performance, particularly through spatial and temporal locality. Optimizations used include register variables, loop interchanging, loop unrolling, and loop blocking. By combining all of these optimizations together, a matrix multiplication program can be speed-up by 904%. This factor can still be greatly expanded upon with other optimization techniques or improvements to the techniques used here.

Related Work:

Information about Register Variables:

https://www.tutorialspoint.com/register-keyword-in-c#:~:text=Register%20variables%20tell%20the%20compiler,to%20declare%20the%20register%20variables.

This article explains how the "register" keyword works in creating register variables. As shown in the Methodology section of this paper, the "register" keyword is used at the front of a variable declaration to clarify that the data is to be stored in a register. This is particularly helpful when it comes to loop iterations.

Information about Loop Unrolling:

https://www.geeksforgeeks.org/loop-unrolling/

This article explains how to utilize loop unrolling, and why it helps improve performance. If the iteration size is constant, then the entire loop can be "unrolled" so that all the calculations happen sequentially, and without branching. If the iteration size is variable, then loop unrolling can still occur by increasing the loop's increment (See Methodology for Example).

Information about Loop Blocking:

https://software.intel.com/content/www/us/en/develop/articles/loop-optimizations-where-blocks-are-required.html

This article explains in detail how loop blocking works. It explains how to create blocks, how the memory hierarchy works, and how loop blocking manipulates temporal locality through cache reuse.

Methodology:

T1 = Optimization with Register Variables

By using register variables instead of regular variables, data gets stored within the CPU, rather than memory. This provides a minor boost in performance, as the CPU no longer has to dig through caches or memory in order to pull variable data out.

Example Used:

```
register int l,m,n,k;
register int i,j, ii, jj, ll;
register double temp;
```

These variables are used to iterate through the for loops that multiply matrices. Storing them as registers helps improve performance as mentioned earlier by about 10%.

T2 = Optimization with Loop Interchange

This optimization takes advantage of spatial locality. In most modern languages, values that are stored in 2D arrays are stored by row first, then by column. As such, when iterating through a 2D array using a for loop, it is much more efficient to have columns in the outer loop and rows in the inner loop.

Example Used:

```
for(i=0;i<m;i++) {
  for(j=0;j<k;j++) {

    A[i][j] = i+j+4.0;
  }
}

for(i=0;i<k;i++) {
  for(j=0;j<n;j++) {
    B[i][j] = i+j+5.0;
  }
}

for(i=0;i<m;i++) {
  for(j=0;j<n;j++) {
    C[i][j] = 0.0;
  }
}
```

By placing ‘i’ before ‘j’, the rows are iterated first (j++), followed by the columns (i++).

T3 = Optimization with Loop Unrolling

Loop Unrolling is the act of optimizing a loop by “foreseeing” future iterations and doing them all at once, instead of iterating one by one. This is particularly helpful for reducing the branch penalties, as a smaller number of loops leads to a smaller number of instructions.

Example Used:

```
for(i=0;i<m;i+=2) {
  for(j=0;j<n;j+=2) {
    for(l=0;l<k;l+=2) {
      C[i][j] = C[i][j] + B[l][j]*A[i][l];
      C[i][j] = C[i][j] + B[l+1][j]*A[i][l+1];
      C[i][j+1] = C[i][j+1] + B[l][j+1]*A[i][l];
      C[i][j+1] = C[i][j+1] + B[l+1][j+1]*A[i][l+1];
      C[i+1][j] = C[i+1][j] + B[l][j]*A[i+1][l];
      C[i+1][j] = C[i+1][j] + B[l+1][j]*A[i+1][l+1];
      C[i+1][j+1] = C[i+1][j+1] + B[l][j+1]*A[i+1][l];
      C[i+1][j+1] = C[i+1][j+1] + B[l+1][j+1]*A[i+1][l+1];
    }
  }
}
```

In this example, the for loops go through eight separate iterations at once, rather that going one at a time. As such, the loop increments increase by +2, rather than +1 since they have already accounted for those elements in the array.

T4 = Optimization with Blocking

To improve cache reuse and locality, loop blocking splits the iterations into “blocks” that keep data within the same cache.

Example Used:

```
for(i=0;i<m;i+=BS) {
  for(j=0;j<n;j+=BS) {
    for(l=0;l<k;l+=BS) {
      for (ii=i;ii<i+BS;ii+=2) {
        for (jj=j;jj<j+BS;jj+=2) {
          for (ll=l;ll<l+BS;ll+=2) {
            C[ii][jj] = C[ii][jj] + B[ll][jj]*A[ii][ll];
            C[ii][jj] = C[ii][jj] + B[ll+1][jj]*A[ii][ll+1];
            C[ii][jj+1] = C[ii][jj+1] + B[ll][jj+1]*A[ii][ll];
            C[ii][jj+1] = C[ii][jj+1] + B[ll+1][jj+1]*A[ii][ll+1];
            C[ii+1][jj] = C[ii+1][jj] + B[ll][jj]*A[ii+1][ll];
            C[ii+1][jj] = C[ii+1][jj] + B[ll+1][jj]*A[ii+1][ll+1];
            C[ii+1][jj+1] = C[ii+1][jj+1] + B[ll][jj+1]*A[ii+1][ll];
            C[ii+1][jj+1] = C[ii+1][jj+1] + B[ll+1][jj+1]*A[ii+1][ll+1];
          }
        }
      }
    }
  }
}
```

The example above combines loop unrolling and loop blocking. The outer three loops control the blocks that will help position data within the cache. The inner three loops are then tasked with iterating through those blocks and completing the calculations. Once again, +2 is used instead of +1 due to unrolling.

Note: The block size (BS) must be a factor of the matrix size. For example, if a square matrix has size 1000, then the block size must be some factor of 1000 (10, 20, 50, 100, etc.)

Results:

Timer Used: gettimeofday() from <sys/time.h>

Architecture: x86_64 Intel i7

Platform: Linux Terminal

Technique/Matrix Size	1000	2000	3000	4000	5000
Naïve + (-O0) (T_{Naive})	16.024 s	177.610 s	698.837 s	1667.381 s	3194.238 s
T1 + (-O0) (T_{opt})	13.609 s	163.220 s	621.857 s	1503.619 s	3021.749 s
(T_{Naive})/(T_{opt})	1.177	1.088	1.124	1.109	1.057
T1 & T2 + (-O0) (T_{opt})	7.113 s	85.317 s	332.652 s	827.757 s	1729.156 s
(T_{Naive})/(T_{opt})	2.253	2.082	2.101	2.014	1.847
T1 & T2 & T3 + (-O0) (T_{opt})	4.388 s	51.728 s	166.087 s	443.229 s	817.202 s
(T_{Naive})/(T_{opt})	3.652	3.434	4.208	3.762	3.909
T1 & T2 & T3 + T4 (blk = 50) (-O0) (T_{opt})	2.852 s	23.001 s	77.281 s	185.624 s	359.025 s
(T_{Naive})/(T_{opt})	5.619	7.722	9.043	8.983	8.897
T1 & T2 & T3 + T4 (blk = 100) (-O0) (T_{opt})	2.943 s	24.006 s	80.091 s	200.662 s	374.933 s
(T_{Naive})/(T_{opt})	5.445	7.399	8.726	8.309	8.519

Analysis: As expected, most optimization techniques applied led to a speed-up. However, one thing to notice is that creating more blocks slowed down the program, instead of speeding it up. This could be due to the size of the caches used in the system, as 100 blocks might not make complete use of the cache as well as 50 blocks.

Citations:

"Loop Optimizations Where Blocks are Required," *Intel*. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/loop-optimizations-where-blocks-are-required.html. [Accessed: 26-Apr-2021].

"Loop Unrolling," *GeeksforGeeks*, 19-Feb-2018. [Online]. Available: https://www.geeksforgeeks.org/loop-unrolling/. [Accessed: 26-Apr-2021].

"register" keyword in C. [Online]. Available: https://www.tutorialspoint.com/register-keyword-in-c#:~:text=Register%20variables%20tell%20the%20compiler,to%20declare%20the%20register%20variables. [Accessed: 26-Apr-2021].