MATRIX MULTIPLICATION USING CACHE MEMORY

Name: Sravya.P
Dept name :School of Informatics, Computing, and Cyber System
Organization:Northern arizona university
City:Arizona state, flagstaff
Email: sp2379@nau.edu

***Abstract*—Optimizing the memory by Comparing the Matrix multiplication by main memory and matrix multiplication utilizing cache memory and Vectorization.**

Keywords—main memory, cache, Vectoriztion.

I. MATRIX MULTILICTION

In mathematics, particular in linear algebra, matrix multiplication is a binary operation that produces a matrix from two matrix. For matrix multiplication the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix known as matrix product has the number of rows of the first and the number of columns of the second matrix. The product of the matrix A and B is then denoted simply as AB.

II. EASE OF USE

Main memory

Main memory is where programs and data are kept when the processor is actively using them. When programs and data become active, they are copied from secondary memory into main memory where the processor can interact with them. A copy remains in secondary memory.

Main memory is sometimes called RAM. RAM stands for Random Access Memory. "Random" means that the memory cells can be accessed in any order. However, properly speaking, "RAM" means the type of silicon chip used to implement main memory.

When people say that a computer has "512 megabytes of RAM" they are talking about how big its main memory is. One megabyte of memory is enough to hold approximately one million (106) characters of a word processing document.

Cache memory

Cache Memory is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed.
Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations. There are various different independent caches in a CPU, which store instructions and data.

- Caches are the subsets of the main memory

Register Memory:

Register memory is the smallest and fastest memory in a computer. A register temporarily holds frequently used data, instructions, and memory address that are to be used by CPU.

Register holds a small amount of data around 32 bits or 64 bits. The speed f the CPU depends on the number and size of register that are build into the CPU.

Memory address register:

The memory address register is the cpu register that either stores the memory address from which data will be fetched to the cpu, or the address to which data will be sent and stored,

While writing location of data that needs to be accessed

Challenges

1.To perform a matrix multiplication in cache memory with memory and cpu, of a particular size.

2.To perform a matrix multiplication in main memory

3. To perform the temporal locality of the matrix such that the frequently operations are in cache and thus being fast.

4. To perform the special locality in short to perform a mini linked list operation such that the data next to it will also appear in the cache memory and would easily be fetched.
5. To divide the entire matrix into the blocks of matrix called the patterns that are stored in the cache memory for optimisation
6. Carefully loop the matrix such that the inner and outer loop will be organised to optimise the code, since they have the data dependencies.
7. To calculate the time by performing matrix multiplication through vectorization.

Preliminary Ideas

1. To use the register keyword and do the cache blocking for matrix multiplication.
2. To use the temporal locality :

This is referred to the data items that are used frequently. We can take this advantage because we need the resultant matrix to be used frequently to store the result.

3. To use the spatial locality :

This is referred to the address that are located near by in cache memory.

4. Since we are using iterative loops if we have the body of the code in the cache it will be very effective, because instead of going to the memory to execute the body of the loop, you can go to the cache and get the data

Miss rate:

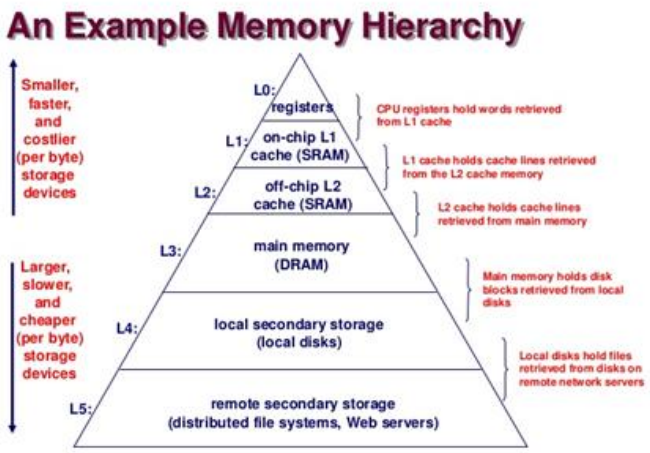
Fraction of memory reference not found in the cache = 1-hit rate

Hit Time :

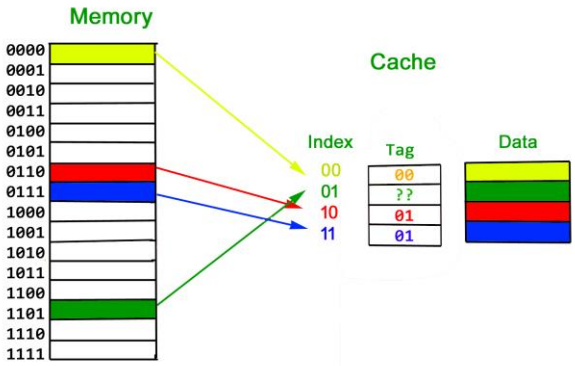
Time to deliver a line in the cache to processor

Includes the time to determine whether the line is in the cache or not

The cache hierarchy is :



Tags in cache memory:



The tags are used to specify the data, i.e,the tag along with the index allows use to find the exact data.

Here the index are the lower 2 bits od the memory

For example: index=00, tag =00 → data of (0000)

Data positioning :

For example if we take 1110(blue)

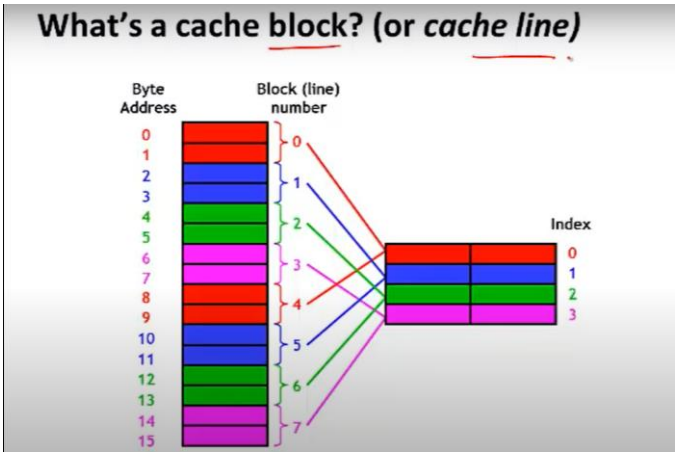
we know that the last two bits goes to the index

So, the tag along with the index we can say exactly what data is set in the cache.

So, there is no confusion on multiple memory address stores into the same position on the cache.

Cache blocking/Cache lines :

For example if I have a block size of 2 bytes and the byte address has 16 bits(ranges from 0-15) and the block line number of 7 and the cache memory takes the data that is in the byte address.



Blocking refers to a class of techniques that aim to break the original problem into small chunks, so that all data needed to process each chunk fits in the cache. This way that data can be Optimally reused before the next chunk is processed.

Unoptimized matrix multiplication:

The numerical algorithms can be reformulated to maintain accumulators to hold partial results between processing of different blocks.

```
for (int i = 0; i < SIZE; i++)
  for (int j = 0; j < SIZE; j++)
    c[i][j] = 0;

for (int i = 0; i < SIZE; i++) // "outer" loop
  for (int j = 0; j < SIZE; j++)
    for (int k = 0; k < SIZE; k++)
      c[i][j] += a[i][k] * b[k][j];
```

Here, The array b is traversed in the wrong way, and we should try to block with respect to it.

The i loop level is the outer loop which causes all b elements to be revisited, for j and k, are therefore candidate lops to blocks

This can be done by splitting the k loop into blocks:

Blocking :

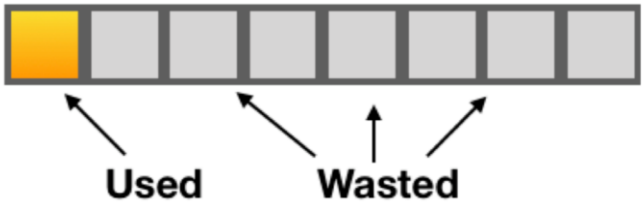
When blocking, the loops are nested in certain way for data dependencies. If one iteration depends on values produced in an earlier iteration, that one must loop in a certain way for boundary zones of the block to preserve the application semantics.

The blocking factors should be chosen to make the active problem size fit inside the target cache size. Different lop levels can be blocking to fit different cache levels.

When two loops use the same data it is beneficial to reduce the amount of data accessed between the loops. Cache lines fetched into the cache by the first loop may still be in the cache when the second lop is run, so that it does not have to fetch them. The less data touched between the two loops, the best optimized. So If it is possible to completely merge the loop, then all potential misses from the second loop will disappear.

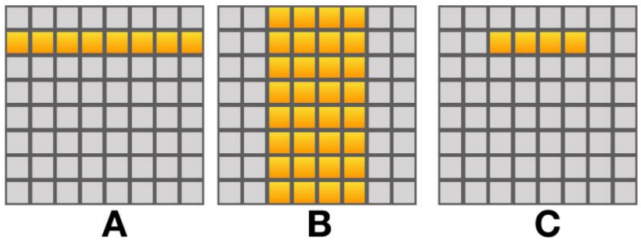
Register Blocking:

The CPU in the normal matrix is not fully utilized because it is memory bound. The Processor is capable of loading wide single instruction multiple data(SIMD) vectors, In the case of matrix A, the processr loads a cache line from the main memory into the L1 cache. Then it loads the scalar from the cache line one at a time. This is not ideal, But the situation f matrix B is much worst. In matrix B, we scan the matrix down a column and use a single scalar from every cache line that we load. By the time we finish scanning the column and start the next column, the processor had already flushed the cache. Here I assume that the length of column times the size of the cache line is greater than our L1 cache.



So, we need to improve the memory utilization of matrix B.

One possible implementation is to use multiple scalar values from each cache line in matrix B at once. We need to multiply the values that we load a scalar from matrix A and broadcast it 8 times to fill the SIMD register. Then we can load 8 consecutive columns from matrix B and perform the vectorized element wise computation. This means that we process and calculate 9 results in matrix C at once. Here instead of accumulating into a single scalar we'll be accumulating into a short vector of 8 scalars. The vectorized method will calculate in matrix C together



Milestones:

1. To take a correct subset for the cache memory, so that the frequently used data can be kept in cache
2. To determine a hit or a miss of the data

3. To calculate the matrix multiplication
4. To determine the locality of the data.
5. To perform cache miss analysis of the matrix multiplication
6. To perform the blocked matrix multiplication
7. To perform the cache miss analysis of the blocked matrix multiplication .
8. To perform vectorization.

Algorithm

//normal matrix multiplication.

```
C = (double *) calloc(size of(double), n*n);
//Multiply n*n matrix of a and b
for (int i = 0; i < SIZE; i++)
  for (int j = 0; j < SIZE; j++)
    c[i][j] = 0;

for (int i = 0; i < SIZE; i++) // "outer" loop
  for (int j = 0; j < SIZE; j++)
    for (int k = 0; k < SIZE; k++)
      c[i][j] += a[i][k] * b[k][j];
```

```
//Blocked matrix multiplication
for (int i = 0; i < SIZE; i++)
  for (int j = 0; j < SIZE; j++)
    c[i][j] = 0;

for (int ii = 0; ii < SIZE; ii += BLOCK_I)
  for (int kk = 0; kk < SIZE; kk += BLOCK_K)
    for (int jj = 0; jj < SIZE; jj += BLOCK_J)
      for (int i = ii; i < ii + BLOCK_I && i < SIZE; i++)
        for (int k = kk; k < kk + BLOCK_K && k < SIZE; k++)
          for (int j = jj; j < jj + BLOCK_J && j < SIZE; j++)
            c[i][j] += a[i][k] * b[k][j];
```

//vectorization

```
int m = A.length;
int n=A[0].length
int p=B[0].length
int [][] c =new int[m][p];
for (int i=0; i<m; i++) {
    for(int k=0;k<n;k++) {
        for(int j=0; j<p;j++) {
          c[i][j]+ = A[i][k] *B[k][j];
        }
    }
}
```

Preliminary Result

Matrix multiplication : 6.56
Cache Blocking time: 3.45
Vectorization : 6.01

Cache blocking:

The Cache blocking times is the time that is required to divide the entire matrix and then perform the operation individually and then combine it to produce the final result.

The matrix multiplication time is the time that the normal matrix multiplication requires.

Vectorization:

This transformation reduced the memory bandwidth of matrix B by a factor of 8(the SIMD width). Instead of performing N*N reads(most likely miss the cache) we perform N*N/8 reads.

We can load two elements from memory each cycle and perform two arithmetic operations each cycle. Here we multiply the elements from matrix A with an element from matrix B. This means that we have two memory operations for every arithmetic operations. Through this we can achieve at most 50% utilization

REFERENCES

Memory hierarchy :

https://www.brightways.org/computer-organization-memory-hierarchy.php

Tags in cache memory:

https://www.geeksforgeeks.org/cache-organization-set-1-introduction/

Introduction to cache memory:

https://www.youtube.com/watch?v=b_js5-gkZUI&t=459s

Reference Code:

https://docs.roguewave.com/threadspotter/2011.2/manual_html_linux/manual_html/ch05s02.html

Vectorization:

https://www.varsitytutors.com/hotmath/hotmath_help/topics/multiplying-vector-by-a-matrix