# Performance x64 Caches Notes

## Terminology

Cache Hit: The data we need is cached.

Cache Miss: The data we need is not cached.

Data Cache: A cache used for storing data for arithmetic and computation.

Instruction Cache: A cache used for storing instructions for the CPU to execute.

Clock Cycle: One tick for the CPU's timer. Clock cycles are the smallest possible duration of time for the device.

L1: Smallest, but fastest caches (x86/x64 have both data and instruction L1)

L2: Medium size, medium speed

L3: Largest, but slowest cache

| Memory Type | Latency (Clock Cycles) |
|---|---|
| Register | 1 |
| L1 Cache | 3 |
| L2 Cache | 15 |
| L3 Cache | 60 |
| Main Memory | 150 |
| Hard Drive | Millions and millions |

Usually for multi-core CPU, the L1 and L2 are per core so each core has its own L1 and L2. L3 is shared among all cores.

## Naïve Matrix Multiplication

```
void Matrix::MUL(Matrix& dest, Matrix& src1, Matrix& src2) {
    int dim = dest.dimension;
    Matrix* tmp = new Matrix(dest.dimension);

    // Compute the matrix product
    for(int row = 0; row < dim; row++) {
        for(int col = 0; col < dim; col++) {
            double result = 0.0;

            // Compute the dot product of row src1 and col src2
            for(int dot = 0; dot < dim; dot++)
                result += src1.Get(dot, row) * src2.Get(col, dot);

            tmp->Set(col, row, result);
        }
    }

    // Move the tmp into the dest
    Matrix::MOV(dest, *tmp);
}
```

```
$ ./matrix_multiplication
Product computed in 12276296
Product computed in 12128504
Product computed in 12585935
Product computed in 12528847
Product computed in 12535354
Product computed in 12507618
Product computed in 12466965
Product computed in 12531697
Product computed in 12855040
Product computed in 12688697
All done!
```

Optimization – keep two set of data one in row major and one in column

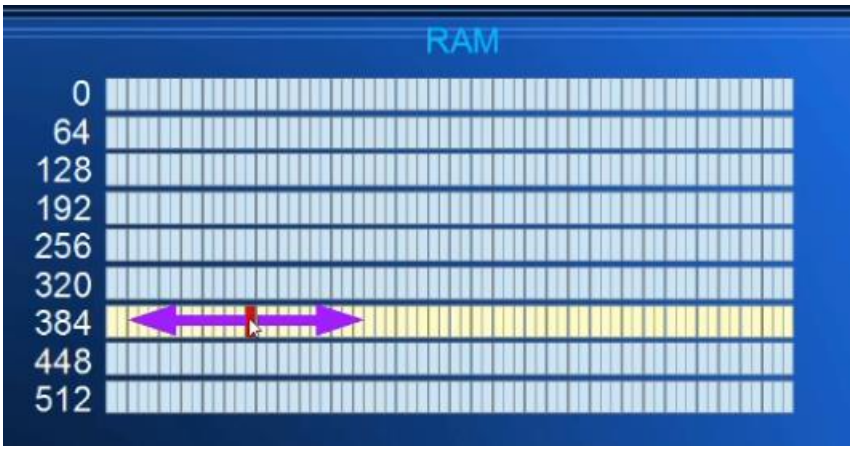
```
$ ./matrix_multiplication
Product computed in 9339397
Product computed in 9414426
Product computed in 9676411
Product computed in 9512850
Product computed in 9528594
Product computed in 9695624
Product computed in 9537854
Product computed in 9563172
Product computed in 9538766
Product computed in 9626732
All done!
```

```
double Matrix::GetCol(int col, int row) {
    return dataCol[col + row * dimension];
}

double Matrix::GetRow(int col, int row) {
    return dataRow[row + col * dimension];
}
```

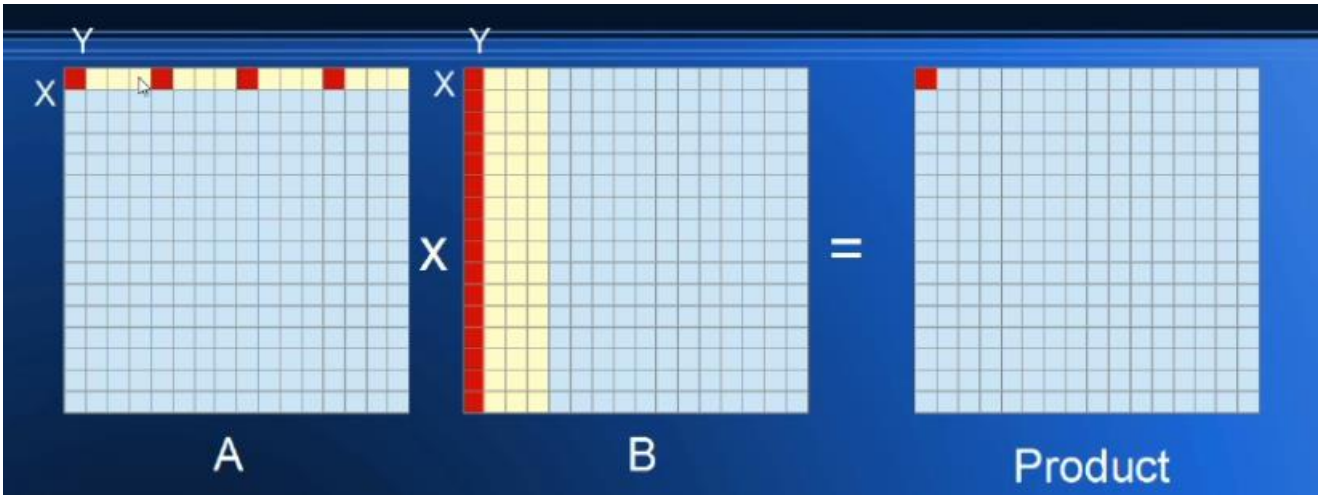
```
void Matrix::MUL(Matrix& dest, Matrix& src1, Matrix& src2) {
    int dim = dest.dimension;
    Matrix* tmp = new Matrix(dest.dimension);

    // Compute the matrix product
    for(int row = 0; row < dim; row++) {
        for(int col = 0; col < dim; col++) {
            double result = 0.0;

            // Compute the dot product of row src1 and col src2
            for(int dot = 0; dot < dim; dot++)
                result += src1.GetCol(dot, row) * src2.GetRow(col, dot);

            tmp->Set(col, row, result);
        }
    }

    // Move the tmp into the dest
    Matrix::MOV(dest, *tmp);
}
```

## Cache Lines

It does not matter if you step through an array forward or backward. The entire cache line is being read and either direction has no performance penalty.

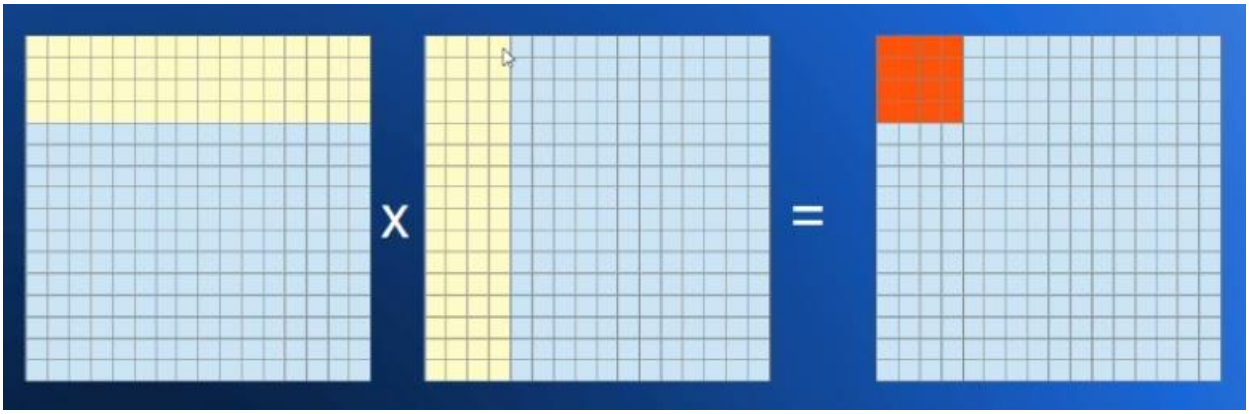


## In matrix product




## Blocking

Fit data from matrix a and b into the cache line and compute the result block by block.




```
$ ./matrix_multiplication_blocked
Product computed in 1601448
Product computed in 1599445
Product computed in 1614472
Product computed in 1601378
Product computed in 1648596
Product computed in 1622975
Product computed in 1610049
Product computed in 1602905
Product computed in 1601146
Product computed in 1599039
All done!
```

```
// Compute the matrix product
for (int row = 0; row < dim; row += blockSize)
{
    for (int col = 0; col < dim; col += blockSize)
    {
        for (int blockRow = row; blockRow < row + blockSize; blockRow++)
        {
            for (int blockCol = col; blockCol < col + blockSize; blockCol++)
            {
                double result = 0.0;

                // Compute the dot product of row src1 and col src2
                for (int dot = 0; dot < dim; dot++)
                    result += src1.GetCol(dot, blockRow) * src2.GetRow(blockCol, dot);

                tmp->Set(blockCol, blockRow, result);
            }
        }
    }
}
```