

## CS296: Program Optimization. Due 4:00 pm 12/3/2008

Your assignment is to take a matrix matrix multiplication code and optimize it (you can choose a different code to optimize, but matrix-matrix multiplication is simple and you should be able to obtain performance improvements with just a few transformations). Below I will list a few transformations that you can apply to reduce the execution time, but you are encouraged to come up with others. Also, you can run the experiments on different machines and compare the performance benefit obtained. Most of the times you will find that what works for one architecture does not work (as well) for another, or requires different parameter values.

### Machines and software to run the experiments

1. You can run your experiments on the machines in 0216 SC or use csil-linux-ts1 / csil-linux-ts2. There is terminal server to access to these machines and the instructions can be found at:

<https://agora.cs.uiuc.edu/display/CSIL/Connecting+to+CSIL+Linux+Terminal+Servers>.

The terminal server will allow, for instance, to run graphical version of vtune from home.

The csil-linux-ts1 and csil-linux-ts2 have the newest processors in CSIL: Intel(R) Xeon(R) CPU E7320 @ 2.13GHz. Each machine has 4 quad cores for a total of 16 cores.

The machines in 0216 are half dual core, and half quad core single chip machines. On linux you can check the characteristics of the processor in the machine by running "more /proc/cpuinfo".

The INTEL software such as the icc compiler, Vtune and the MKL libraries are under the /opt/intel directory.

### How to measure the execution time

There are several ways to measure the execution time of a program:

1. Use `gettimeofday`, which will return the current time in seconds and microseconds in a "timeval" structure in the first parameter. The resolution of the system clock is unspecified. Below is an example:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define SIZE 1000

int main(int argc, char **argv) {

    struct timeval start, end;
    int i, j, k, ;

    gettimeofday(&start, NULL);

        for(i = 0; i < SIZE; i++)
            for(j = 0; j < SIZE; j++)
                for(k = 0; k < SIZE; k++)
                    C[i][j] += A[i][k] * B[k][j];

    gettimeofday(&end, NULL);
    i = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_usec - start.tv_usec);
    printf("%d iterations, %d usec\n", SIZE, i);

    return 0;
}
```

```
}
```

2. Read the time stamp counter on the processor using the RDTSC instruction. This counter counts the processor cycles. You will need to know the processor frequency to compute time if you want to compare the execution times on two different processors. Information about the time stamp counter can be found in:

([http://en.wikipedia.org/wiki/Time\\_Stamp\\_Counter](http://en.wikipedia.org/wiki/Time_Stamp_Counter)).

Next are two examples of how to read the time stamp counters when compiling with `icc` or `gcc`:

- (a) `icc` compiler: use `_rdtsc()` as shown below:

```
#include <stdio.h>
#include <stdlib.h>
#include <ia32intrin.h> // needs this include
#define SIZE 1000

int main(int argc, char **argv) {

    unsigned long long start_c, end_c;
    int i, j, k, ;

    start_c = _rdtsc();

    for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
            for(k = 0; k < SIZE; k++)
                C[i][j] += A[i][k] * B[k][j];

    end_c=_rdtsc();

    printf("It took %llu cycles\n",end_c - start_c);
    return 0;
}
```

- (b) `gcc` compiler: You will need to declare the function below, and then use the function as with `icc`.

```
static inline unsigned long long _rdtsc(void)
{
    register unsigned hi, lo;
    __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));

    return ( (unsigned long long)lo)|((unsigned long long)hi)<<32 );
}
```

3. Use Vtune. You can find Documentation on how to use VTune in the class website, under Section Notes, Section 12.

### What you are asked to do

Your task is to optimize a simple matrix-matrix multiplication code. In order to show the options that you tried you should include in your final report:

1. Architectural characteristics of the machine where you ran your code, including: frequency of the processor, number of processors, and cache parameters if you can find them (you may need to google to find them).
2. A table or a graph of the execution time of the different matrix-matrix multiplication versions that you tried.
3. List of versions that you tried. You need to either show the code for each version or indicate what transformations are applied to obtain the version. Also, for each version explain why the execution time is better or worse than other versions.

Your grade will depend on:

1. The number versions you tried. Even if you did not achieve good performance improvements, but you tried many different options, and you explain how a given version should improve performance.
2. The performance improvement obtained over the naive version.

## Transformations to improve performance

How can you generate multiple versions of the matrix-matrix multiplication code?

1. Try different compiler optimizations. You can look at the compiler flags `icc -help`. Although there are many compiler flags to try them all, you can try some of the most common and measure the difference in execution time: `-O0`, `-O1`, `-O2`, `-O3`, `-O4`, `-fast`.
2. The loops in the matrix-matrix multiplication are independent, and you can exchange them. Different loop orders result in different execution times because the matrices are traversed in different orders, resulting in a different address trace and cache interactions.
3. Linearize the arrays. Rather than declaring 2D arrays, you can declare the matrix as 1D arrays. In that case, the innermost statement in the matrix-matrix multiplication code above will be something like this (assuming matrices are squares):

```
C[i*SIZE+j] += A[i*SIZE+k] * B[k*SIZE+j];
```

4. Scalarize. Variable `C[i][j]` is an invariant for this loop order `i, j, k` because `C` is indexed by loop indexes `i` and `j`). Thus, we can use a scalar and move the assignment outside the innermost loop. Sometimes this results in better performance because the compiler is more likely to assign a scalar variable to a register than an array element. The resulting code will look something like this:

```
for(i = 0; i < SIZE; i++)
  for(j = 0; j < SIZE; j++){
    D = C[i][j];
    for(k = 0; k < SIZE; k++)
      D += A[i][k] * B[k][j];
    C[i][j] = D;
  }
```

5. Loop unrolling. You can unroll any of the loops and choose the degree of unroll. Below is an example, where the innermost loop is unrolled 4 times. I am assuming `SIZE` is multiple of 4, but extra code is needed when this is not the case. For this exercise you can simply choose unroll values that perfectly divide `SIZE`.

```

for(i = 0; i < SIZE; i++)
  for(j = 0; j < SIZE; j++)
    for(k = 0; k < SIZE; k+=4){
      C[i][j] += A[i][k] * B[k][j];
      C[i][j] += A[i][k+1] * B[k+1][j];
      C[i][j] += A[i][k+2] * B[k+2][j];
      C[i][j] += A[i][k+3] * B[k+3][j];
    }

```

In the example below, the j loop is unrolled 3 times.

```

for(i = 0; i < SIZE; i++)
  for(j = 0; j < SIZE; j+=3)
    for(k = 0; k < SIZE; k++)
      C[i][j] += A[i][k] * B[k][j];
    for(k = 0; k < SIZE; k++)
      C[i][j+1] += A[i][k] * B[k][j+1];
    for(k = 0; k < SIZE; k++)
      C[i][j+2] += A[i][k] * B[k][j+2];

```

In most cases, when a loop other than the innermost loop is unrolled, a transformation called unroll and jam is usually applied. The jam part of the unroll and jam occurs when the loops are fused back together. After applying unroll and jam, the code above results in the next code:

```

for(i = 0; i < SIZE; i++)
  for(j = 0; j < SIZE; j+=3)
    for(k = 0; k < SIZE; k++){
      C[i][j] += A[i][k] * B[k][j];
      C[i][j+1] += A[i][k] * B[k][j+1];
      C[i][j+2] += A[i][k] * B[k][j+2];
    }

```

## 6. Loop Tiling. (From the wikipedia: [http://en.wikipedia.org/wiki/Loop\\_tiling](http://en.wikipedia.org/wiki/Loop_tiling):

“ Loop tiling partitions a loop’s iteration space into smaller chunks or blocks, so as to help ensure data used in a loop stays in the cache until it is reused. The partitioning of the loop iteration space leads to partitioning of large array into smaller blocks, thus fitting accessed array elements into cache size, enhancing cache reuse and eliminating cache size requirements”.)

Below is a simple example that will be used to illustrate how tiling works.

```

for (j=0;j<100; j++)
  for (i=0; i<4096; i++)
    sum += a[i];

```

Suppose that you have a cache with a line size of 32 bytes, 64 lines, direct mapping, and elements of the array a are integers of 4 bytes long each:

- The first access to elements a[0], a[8], a[16], ... a[4088] will cause a miss (this miss is usually called cold or compulsory miss).

- When executing the innermost `i` loop, after accessing elements `a[0..511]` the cache is full, and a cache line has to be written back to memory before a new line is brought to the cache. Thus, when we are done with `i` loop, and execute the next iteration of `j` loop, we will find that `a[0]` is gone, and needs to be brought to the cache again.
- However, each element `a[i]` is reused 100 times, that is, the code has temporal locality that we are not exploiting in its current form. However, this can be exploited by applying tiling.

Below is a tiled version of the code above. We have introduced an extra loop, and now the innermost loop will traverse the array `a` in groups of 512 elements at a time. Notice that we chose `block` to be 512 because the cache can accommodate 512 elements of `a` (elements `a[0..511]`). This way, iteration `j = 1` will result in a cache miss for elements `a[0]`, `a[8]`, `a[16].. a[503]`, but when executing `j` iterations `2..100`, the elements of `a` will be in cache and result in a cache hit. A similar pattern will occur every `ii` iteration. (Notice that to simplify the code I have chosen an array size that is multiple of the block size).

```
for (ii=0; ii<4096; ii+=block);
  for (j=0;j<100; j++)
    for (i=0; i<block; i++)
      sum += a[i];
```

Now, if we apply tiling to our matrix-matrix multiplication code the resulting code is something like what you have below. We need to choose the appropriate value of `block` (Notice that `SIZE` here is assumed to be a multiple of `block`).

```
for (i0 = 0; i0 < SIZE; i0 += block)
  for (j0 = 0; j0 < SIZE; j0 += block)
    for (k0 = 0; k0 < SIZE; k0 += block)
      for (i = i0; i < i0 + block; i++)
        for (j = j0; j < j0 + block; j++)
          for (k = k0; k < k0 + block; k++)
            C[i][j] += A[i][k] * B[k][j];
```

If you apply loop tiling and loop unrolling, you should get something similar to the code shown in Figure 1.

Some experiments that my research group did in the past show that loop orders `i, j, k` or `j, i, k`, block sizes between 40 and 60, and small values of unroll for the two outermost loops (try values between `1..4`) resulted in the best performance for many of today's machines.

Other transformations that you can try are these:

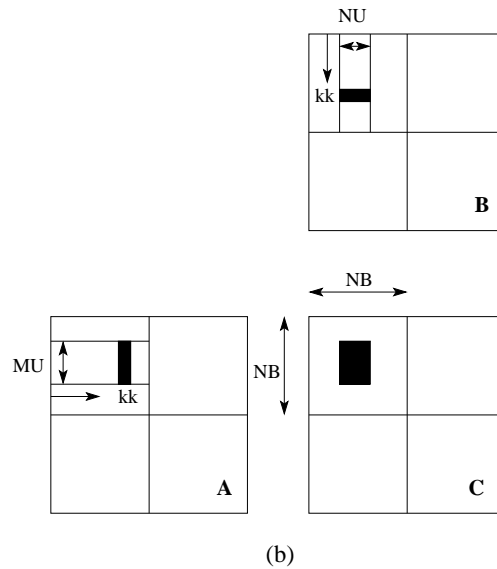
- Call the MKL library that does matrix-matrix multiplication. You should be able to find the libraries under `/opt/intel`. Otherwise, you can download them. They are free for personal use with Linux. Documentation can be found in:  
<http://www.intel.com/cd/software/products/asm-na/eng/345631.htm>
- Parallelize matrix-matrix multiplication. One simple way to do it is to use OpenMP. I put some slides about OpenMP and you can find more information in <http://openmp.org/wp/>
- Vector intrinsics. They will be covered in the last Discussion Section.
- You can use Strassen algorithm to implement matrix-matrix multiplication.

```

for (j=0; j<=SIZE; j +=block)
  for (i=0; i<=SIZE; i +=block)
    for (k=0; k<=SIZE; k +=block)
      // mini-MMM code
      for (jj=0; jj<=j+block-1; jj+=MU)
        for (ii=0; ii<=i+block-1; ii +=NU)
          for (kk=0; kk<=k+block-1; kk++)

            // micro-MMM code
            C[ii][jj]+= A[ii][kk] * B[kk][jj]
            C[ii+1][jj]+= A[ii+1][kk] * B[kk][jj]
            C[ii+2][jj]+= A[ii+2][kk] * B[kk][jj]
            C[ii][jj+1]+= A[ii][kk] * B[kk][jj+1]
            C[ii+1][jj+1]+= A[ii+1][kk] * B[kk][jj+1]
            C[ii+2][jj+1]+= A[ii+2][kk] * B[kk][jj+1]

```



**Figure 1.** MMM after cache blocking and register blocking. (a) Code where the innermost loops are unrolled by a factor of  $MU=2$  and  $NU=3$ . (b) Picture of the code on the left.

- Implement matrix-matrix algorithms using cache-oblivious algorithms. To learn about cache oblivious algorithms you can read the papers:
  - “Cache-Oblivious Algorithms” by M. Frigo, C. Leiserson, H. Prokup and S. Ramachandran, Cache-Oblivious Algorithms
  - “Cache-Oblivious Algorithms and Data Structures” by Erik D. Demaine.

A few final remarks:

- Make sure that you obtain correct results. To that end, include code at the end that checks the results.
- If you multiply matrices of different sizes you will obtain higher execution times for the larger matrices, but you will not be able to determine which one executed faster. To solve that problem you can computer FLOPS or FLOPS/s.

(From the wikipedia: “The FLOPS is a measure of computer’s performance that is similar to instructions per second.” )

In the case of matrix-matrix multiplication, the number of Floating point operations is  $2 \times SIZE^3$ , asuming you are using square matrices of size  $SIZE$ , and where the 2 comes from the addition and multiplication in the statement inside the innermost loop, and the 3 from the 3 loops. After measuring the execution time you can compute FLOPS with the following formula:

$$FLOPS = \text{Number\_of\_Floating\_Point\_OPerations} / \text{Time\_it\_took\_to\_execute\_them.}$$

If you measure clock cycles you will need to convert them to seconds by knowing the frequency of the processor.

- Multiply matrices of relatively large sizes, between 2000 and 5000. This way, you do not have to worry about measuring execution time accurately.
- After the matrix-matrix multiplication code is done you may need to print part of matrix C. Otherwise, the compiler (which sometimes is very smart) will detect that you are not using the results of the matrix-matrix multiplication loops and will remove them (this is a compiler transformation called dead code elimination).