Performance

1 Introduction

Performance is an important aspect of software quality. To achieve high performance, a program must fully utilize the processor architecture. Advanced-architecture includes pipelining, superscalar, and deep memory hierarchy. In this note, we use a simple example, matrix-matrix multiplication, to illustrate some major issues in developing high performance numerical software.

We note that performance can be improved even before a program is written. The following example is due to Hamming [4]. Evaluate the infinite sum

$$\Phi(x) = \sum_{k=1}^{\infty} \frac{1}{k(k+x)} \quad \text{for } x = 0.1 : 0.1 : 0.9,$$

with an error less than $tol = 0.5 \times 10^{-4}$. If we sum the series by brute force, we need to caculate at least $20,000$ terms for each value of x and requires more than two million floating-point operations for all nine values of x. Using

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$$

we can prove that $\Phi(1) = 1$. Then we can express

$$\Phi_1(x) = \Phi(x) - \Phi(1) = (1-x) \sum_{k=1}^{\infty} \frac{1}{k(k+1)(k+x)},$$

which converges faster. Repeat this process, we can prove that $\Phi_1(2) = 1/4$ and express

$$\Phi_2(x) = \Phi_1(x) - (1-x)\Phi_1(2) = (1-x)(2-x) \sum_{k=1}^{\infty} \frac{1}{k(k+1)(k+2)(k+x)}.$$

The series

$$\Phi(x) = 1 + (1-x)\left(\frac{1}{4} + (2-x) \sum_{k=1}^{\infty} \frac{1}{k(k+1)(k+2)(k+x)}\right)$$

converges even faster. For the same tolerance, it calculates at most 27 terms for each values of x and requires less than two thousand floating-point operations for all nine values of x.

In this note, we consider the impact of computer architecture on performance.

The illustrative problem is the computation of $AB + C$ and store the result in C, where A is m-by-k, B is k-by-n, and C is m-by-n. All C programs in the following sections were run on platform of Sun Sparcstation 5, 170MHZ, Solaris 2.1, Sun Workshop Compiler C4.2 using double precision. We chose $m = 1000$, $LDA = 1000$ (leading dimension of A), $k = 1000$, $LDB = 1000$, and $n = LDC = 1000$.

We used the following template (C in UNIX) for timing a function.

Program 1 (Timing) *This program records the time cost of a function using UNIX system call getrusage().*

```
long isec, iusec, esec, eusec;
double iseconds, eseconds;
struct rusage rustart, ruend;

/* get start time of seconds and microseconds*/
getrusage(RUSAGE_SELF, &rustart);

/*********************************/
/* call the function to be timed */
/*********************************/

/* get end time of seconds and microseconds*/

getrusage(RUSAGE_SELF, &ruend);
/*convert the start time to seconds */
isec = rustart.ru_utime.tv_sec;     /* seconds */
iusec = rustart.ru_utime.tv_usec;   /* microseconds */
iseconds = (double) (isec + ((float)iusec/1000000));
/*convert the ending time to seconds */
esec = ruend.ru_utime.tv_sec;       /* seconds */
eusec = ruend.ru_utime.tv_usec;     /* microseconds */
eseconds = (double) (esec + ((float)eusec/1000000));

/* time cost (in seconds) is eseconds-iseconds */
```

The performance measurement used here is MFLOPS (Million FLoating-point Operations Per Second), where the floating-point operation is either addition or multiplication. Thus, if A is m-by-k, B is k-by-n, and C is m-by-n, then $AB + C$ requires $2mnk$ floating-point operations.

First we present a straightforward implementation.

Program 2 (Naive Method) *This program computes $AB + C$ where A is m-by-k with leading dimension LDA, B is k-by-n with leading dimension LDB, and C is m-by-n with leading dimension LDC, and writes the result in C.*

```
naive(int m, int n, int k, double* A, int LDA, double * B,
              int LDB, double *C, int LDC){

              double * Acp,* Arp,
                     * Bcp,* Brp,
                     * Ccp,* Crp;

              for(Crp=C,Arp=A;
                  Crp<C+m*LDC,Arp<A+m*LDA;
                  Crp+=LDC,Arp+=LDA)

                for(Ccp=Crp,Bcp=B;
                    Ccp<Crp+n,Bcp<B+n;
                    Ccp++,Bcp++)

                  for(Acp=Arp,Brp=Bcp;
                      Acp<Arp+k,Brp<Bcp+k*LDB;
                      Acp++,Brp+=LDB)

                    *Ccp=*Ccp+*Acp**Brp;
      }
```

Actually this program is a pointer version of following program when $LDA = k$ and $LDB = LDC = n$.

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

By referencing entries by pointers instead of indices, the index calculation is eliminated. Consequently, the instruction count is reduced.

2 Fast Memory (Block Version)

Cache memories are high-speed buffers inserted between the processors and main memory to capture those portions of the contents of main memory currently in use. If the data required by an instruction is not in the cache, the block containing it is obtained from the slower main memory and put in the cache. Since cache memories are typically five to ten times faster than main memory, they can reduce the effective memory access time if a program is carefully designed and implemented.

To take advantage of the fast memory, we partition the matrices A, B, and C into square blocks. When computing $AB + C$, three blocks, one from each matrix, should be kept in the fast memory. Thus the block size is about $\sqrt{S/3}$ where S is the size of the fast memory. See [3] for a full analysis.

The following program shows the block version.

Program 3 (Block Naive Version) *This program computes $C \leftarrow AB + C$ using block size* `blksize`*.*

```
block(int m,int n,int k, int blksize,
      double *A,double *B,double *C)

  for(Crp=C,Arp=A;
      Crp<C+m*n,Arp<A+m*k;
      Crp+=blksize*n,Arp+=blksize*k)

    for(Ccp=Crp,Bcp=B;
        Ccp<Crp+n,Bcp<B+n;
        Ccp+=blksize,Bcp+=blksize)

      for(Acp=Arp,Brp=Bcp;
          Acp<Arp+k,Brp<Bcp+n*k;
          Acp+=blksize,Brp+=blksize*n) {

        bm=bn=bk=blksize;

        if(Arp==A+k*blksize*(m/blksize))
          bm = m % blksize;        /*last row block in A*/

        if(Acp==Arp+blksize*(k/blksize))
```

```
          bk = k % blksize;        /*last col. block in A*/

        if(Bcp==B+blksize*(n/blksize))
          bn = n % blksize;        /*last col. block in B*/

        naive(bm,bn,bk,Acp,k,Brp,n,Ccp,n);
      }
```

In the above program, when the naive method Program 2 is used for multiplying two blocks.

In Sparc, the cache size is 16k for data and instruction. So, the block size is $\sqrt{16000/(3\,\mathrm{sizeof(double)})} \approx 26$. After some trial, we found that the program achieved the top speed of 4.3 MFLOPS at block size of 64. The larger optimal block size is probably due to a second level cache, although it is not described in the system specification.

3 Memory Banks (Block Stride-One Version)

In most systems the bandwidth from a single memory module is not sufficient to match the processor speed. Increasing the computational power without a corresponding increase in the memory bandwidth of data to and from memory can create a serious bottleneck. One technique used to address this problem is called banked memory. Main memory is usually divided into banks. In general, the smaller the memory size, the fewer the number of banks. With banked memory, several modules can be referenced simultaneously to yield a higher effective rate of access. Specifically, the modules are arranged so that n sequential memory addresses fall in n distinct memory modules. By keeping all n modules busy accessing data, effective bandwidths up to n times that of a single module are possible. Associated with memory banks is the memory bank cycle time, the number of clock cycles a given bank must wait before the next access can be made to data in the bank. After an access and during the memory bank cycle, references to data in the bank are suspended until the bank cycle time has elapsed. This is called memory bank conflict. Memory bank conflicts can not occur when processing sequential components of a one-dimensional array or, if row major ordering as in C, a row of a two-dimensional array are being processed. This technique is called stride-one. To use this technique, we changed the naive method for multiplying blocks in Program 2 into row-ordering. Thus, memory is accessed sequentially most of the time. By replacing the function `naive` with the following stride-one version in the block version Program 3, the performance is improved to 5.4 MFLOPS.

Program 4 (Block Stride-One Version) *This program computes $C \leftarrow AB + C$ using row ordering (stride-one).*

```
stride_one(int m,int n,int k,double *A,int LDA,
           double *B,int LDB,double *C,int LDC)

        for(Crp=C,Arp=A; Crp<C+m*LDC,Arp<A+m*LDA;
                                   Crp+=LDC,Arp+=LDA)
           for(Acp=Arp,Brp=B; Acp<Arp+k,Brp<B+k*LDB;
                                         Acp++,Brp+=LDB)
              for(Ccp=Crp,Bcp=Brp; Ccp<Crp+n,Bcp<Brp+n;
                                              Ccp++,Bcp++)
                 *Ccp = (*Ccp) + (*Acp)*(*Bcp);
```

Basically, the above program performs:

```
        for i=0:m-1
           for l=0:k-1
              C[i][0:n-1] = C[i][0:n-1] + A[i][l]*B[l][0:n-1];
```

Thus the entries of the matrices are accessed in rows (sequential memory locations).

4 Reducing Control Hazards (Final Version)

The concept of pipelining is similar to that of assembly line process in an industrial plant. Pipelining is achieved by dividing a task into a sequence of smaller tasks, each of which is executed on a piece of hardware that operates concurrently with the other stages of the pipeline. Successive tasks are streamed into the pipe and get executed in an overlapped fashion with the other subtasks. Each of the steps is performed during a clock cycle of the machine. That is each suboperation is started at the beginning of the cycle and completed at the end of the cycle. There are situations in pipelining when the next instruction can not execute in the following clock cycle. These event are called hazards. One of them is called control hazards which arising from the need to make a decision based on the results of one instruction while others are executing. The equivalent decision task in computer is the branch instruction. If the computer were to stall on a branch, then it would have to pause before continuing the pipeline. In Program 4, each loop involves two comparisons (conditional branches). To reduce the branches, we modified Program 4 into the following code for multiplying two blocks. By using the following function `final` in the block version Program 3, the performance was further improved to 6.45 MFLOPS.

Program 5 (Block Final) *This program computes $C \leftarrow AB + C$ by reducing conditional branch*

```
final(int m,int n,int k,double *A,int LDA,
      double *B,int LDB,double *C,int LDC)

        for (i=0;i<m;i++) {
           a = &A[i*LDA];         /* ith row of A */
           for (l=0;l<k;l++) {
              c = &C[i*LDC];      /* ith row of C */
              b = &B[l*LDB];      /* lth row of B */
              for (j=0;j<n;j++)
                 (*c++)+ = (*a)*(*b++);
              a++;
           }
        }
```

5 Conclusion

The following table summarizes the performances of the programs presented before. It also includes the performance of the vendor (SUN) high performance library function (dgemm). Our objective here is to illustrate some generic techniques in high performance computing. We did not attempt to fine tune our code to fully exploit the underlying machine architecture such as two levels of cache. So, our results are far below theirs.

Table

		Block			
Version	Naive	Naive	Stride-One	Final	Sun dgemm
MFLOPS	1.69	4.3	5.4	6.45	58.22

References

[1] David A. and John L. Hennessy. *Computer Organization & Design, the hardware/software interface, 2 ed.* Morgan Kaufmann Publishers, Inc., 1997.

[2] J.J. Dongarra, J. DuCroz, I. Duff, and Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16:1–17, 1990.

[3] Gene H. Golub and Charles F. Van loan. *Matrix Computations, 3rd ed.* The Johns Hopkins University Press, 1996.

[4] R.W. Hamming. *Numerical Methods for Scientists and Engineers.* McGraw-Hill, New York, 1962.

[5] Nicholas J. Higham. *Accuracy & Stability of Numerical Algorithm.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1996.

[6] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1998.